Application of a Distributed Software System on an Autonomous Maritime Vehicle

May 21, 2021

Fredrik Brushane, 35973

Master of Science Thesis

Supervisors: Dr. Sébastien Lafond, Annamari Soini, Kai Jämsä

Faculty of Science and Engineering

Åbo Akademi University

Åbo 2021

Abstract

In the recent decade, the interest in technologies enabling autonomous vehicles has increased significantly in both industry and academia. Spurred by this interest, an autonomous boat research platform has been set up at Åbo Akademi University to serve as a test bed for autonomous maritime vehicle-related technologies. As a starting point, the boat platform needs a system to both enable remote monitoring and control, as well as to interconnect the various hardware entities of the boat such as sensors and motor. The autonomous vehicle-oriented software ecosystem OpenDLV is used to build an initial control and communication system for the boat platform.

In this thesis, the technologies relevant to the core building blocks of the OpenDLV software ecosystem are explored, including the use of a standardized method of data serialization, multicast communication, containerization, and a distributed system based on the novel microservice architecture. A high-level diagram depicting the planned distributed system of the boat platform is created to provide an overview of the different building blocks of the system and of how they communicate. A number of microservice applications are developed using the OpenDLV software ecosystem to handle the tasks of manual remote control of the boat, video streaming, collection and display of GPS-derived data, and LTE telemetry data collection and display.

The applications developed for this thesis using the OpenDLV ecosystem form the basis of a distributed communication and control system that seems well suited for both an autonomous maritime vehicle in general and the development environment of the boat platform at Åbo Akademi University. Ultimately, however, the strengths of a distributed system such as the one developed for this thesis also come with drawbacks in the form of increased system complexity, which should be carefully considered before and during future development.

Keywords: Autonomous Maritime Vehicle, OpenDLV, Microservice Architecture, Containerization, Distributed System

List of Figures

1	Comparison of type-2 virtualization and lightweight containers [6]	11
2	List of Linux namespace types [14]	13
3	Average CPU utilization, pagerank workload [17]	16
4	Average memory utilization, pagerank workload [17]	17
5	OpenDLV *AccelerationReading* message specification	20
6	Excerpt from the system specification diagram	23
7	Diagram of the central element of the system, "the brain/AI"	25
8	Motor remote control system overview	26
9	System overview diagram, video streaming	27
10	Control schemes for the original vs. the modified gamepad microservice. Base controller image from [23].	31
11	*ActuationRequest* message specification	33
12	State diagram for the controller input thread	34
13	State diagram for the message sending thread	36
14	The 8-byte payload of the NMEA2k message with the PGN 65332	39
15	State diagram for the control loop of the motor control microservice	42
16	Top-down diagram of the error margin for motor rotation	44
17	Network and IPSec configuration of the boat and shore LANs	45
18	Message propagation in the motor control subsystem	47
19	The single session problem	48
20	Cluon-Relay as a possible solution the single session problem	49
21	Diagram of the OpenDLV video streaming process	51
22	Shell script for launching a Gstreamer pipeline	55
23	Message specification for the *RouterDataMessage*	57
24	Message overview in vehicle-view	59
25	Plotting code user interface in vehicle-view	60
26	Signal-to-noise ratio plot in vehicle-view	60
27	Map pane in vehicle-view	61
28	Latest high-level system diagram	73
29	Router script for sending GPS and LTE-telemetry data using NetCat	74

Abbreviations

- API *Application Programming Interface*
- CAN *Controller Area Network*
- DDS *Data Distribution Service*
- ESB *Enterprise Service Bus*
- GPS *Global Positioning System*
- IMU *Inertial Measurement Unit*
- IP *Internet Protocol*
- IPSec *Internet Protocol Security*
- JSON *JavaScript Object Notation*
- KVM *Kernel-based Virtual Machine*
- LAN *Local Area Network*
- LiDAR *Light Detection and Ranging*
- LTE *Long-Term Evolution*
- LXC *Linux Containers*
- MSA *Microservice Architecture*
- MTU *Maximum Transmission Unit*
- NMEA *National Marine Electronics Association*
- N2K and NMEA2k *NMEA2000 bus*
- OpenDLV *Open DriverLess Vehicle*
- PGN *Parameter Group Number*
- Protobuf *Protocol Buffers*
- ROS Robot Operating System

- RSOA Robot System Onboard Architecture
- RTP Real-time Transport Protocol
- SINR Signal-to-Noise Ratio
- SOA Service-Oriented Architecture
- TCP Transmission Control Protocol
- UDP User Datagram Protocol
- VM Virtual Machine
- VMM Virtual Machine Manager
- YAML YAML Ain't Markup Language

Contents

1 Introduction **1**
1.1 Object of the Thesis . . . 1
1.2 Thesis Structure . . . 2

2 Technical Background **3**
2.1 Technologies in Autonomous Vehicles . . . 3
2.1.1 Maritime-Specific Application . . . 4
2.2 Microservice Architecture . . . 5
2.2.1 Characteristics of a Microservice Architecture . . . 5
2.2.2 Advantages of a Microservice Architecture . . . 6
2.2.3 Pitfalls in a Microservice Architecture . . . 7
2.2.4 Smells in the Automotive Domain . . . 9
2.2.5 Service-to-Host Mapping . . . 10
2.3 Virtualization . . . 10
2.3.1 Virtual Machines . . . 10
2.3.2 Containerization . . . 12
2.3.3 Docker . . . 13
2.3.4 Comparison of KVM, Xen, and Docker on ARM . . . 14
2.3.5 Virtual Machines and Containers at Scale . . . 14
2.3.6 Combining Containers and Virtual Machines . . . 18
2.4 OpenDLV . . . 19
2.4.1 Communication . . . 19
2.4.2 Data Format . . . 20
2.4.3 Libcluon . . . 21

3 Implementation **23**
3.1 System Design . . . 23
3.1.1 The Brain . . . 24
3.1.2 Video Streaming . . . 27
3.1.3 System Requirements . . . 28
3.2 Motor Control . . . 29
3.2.1 System Description . . . 29
3.2.2 Motivation . . . 29
3.2.3 Remote Control Microservice . . . 30

3.2.4 Motor Control Microservice 37
3.3 Networking 44
3.3.1 OpenDLV communication in practice 46
3.4 Video Streaming 50
3.4.1 System Description 50
3.4.2 Video Streaming with OpenDLV 50
3.4.3 Containerization 52
3.4.4 Video Streaming with Gstreamer 53
3.5 GPS and LTE Telemetry 55
3.6 Visualization 58
3.6.1 Message Overview 59
3.6.2 Plotting 59
3.6.3 Map 61

4 Results and Evaluation **63**
4.1 Motor Control System 63
4.2 Video Streaming and Visualization 64
4.3 OpenDLV 64

5 Conclusion **66**

6 Svensk Sammanfattning - Swedish Summary **67**
6.1 Inledning 67
6.2 Systembeskrivning 67
6.3 Metoder 68
6.4 Implementation 69
6.5 Resultat 69

A System Diagram **73**

B Router Script **74**

1 Introduction

In the recent decade, the interest in technologies enabling autonomous vehicles has increased significantly in both industry and academia. Although a large part of contemporary research and development is focused specifically on autonomous cars, there is also a significant amount of research taking place in the field of autonomous maritime vehicles. An autonomous boat research platform at Åbo Akademi University has been created with the purpose of serving as a test bed for the design and implementation of remote control and navigation infrastructure, situational awareness and sensor technologies, and maritime oriented machine learning models.

1.1 Object of the Thesis

The object of this thesis is to design and implement an initial control and communication system for the boat project, which consists of an inflatable pontoon boat equipped with various computing hardware and sensors, as well as control and monitoring station on the shore. In lieu of creating such a complex system from the ground up, an open source software ecosystem for autonomous vehicles, OpenDLV [1], was through initial evaluation chosen as a suitable starting point from which development of a control and communication system for the boat platform could begin.

The intent of this thesis is to look into how the core features of the OpenDLV software ecosystem affect both high-level system design as well as low-level application design and implementation. A handful of applications will be developed for the boat platform and shore control station using OpenDLV, which will perform varying tasks such as controlling an outboard motor and gathering data from sensors. The thesis seeks to evaluate the strengths and weaknesses of the OpenDLV architecture, particularly concerning communication, system design, and application development.

As the OpenDLV software ecosystem is realized using a relatively new and somewhat immature technology, the microservice architecture, any implementation thereof is likely to present some issues that will need to be worked out. At the same time, designing and implementing a novel system such as this lays a good foundation for innovation, and the lessons learned from the process of refining the architecture and

finding solutions to emergent problems will certainly be interesting from a research viewpoint.

1.2 Thesis Structure

In Chapter 2 of this thesis, the technical background for the thesis is presented. In Section 2.1, some general technological information concerning autonomous vehicles is presented, including a maritime-oriented autonomous system application. In Sections 2.2 and 2.3, some of the underlying technologies used by the OpenDLV software ecosystem are discussed, while in Section 2.4, OpenDLV itself is presented. In Chapter 3, the practical implementations and experiments performed for this thesis are presented, opening with a system design description and continuing with descriptions of the microservice-based functional application developed for the boat platform as well as a description and discussion of the communication systems of the boat platform. Chapter 4 contains the results of the thesis, including an evaluation of OpenDLV, the applications developed, and some discussion and suggestions concerning future development of the boat platform. In Chapter 5, the thesis is concluded.

2 Technical Background

2.1 Technologies in Autonomous Vehicles

An overview of key technologies for autonomous cars is presented by Zhao et al. in their paper [2]. In the paper, environmental perception is identified as an important mechanism to provide data for the control systems of a vehicle. Three sensor types are presented, each with different use cases and weaknesses: LiDAR (Light Detection and Ranging), a laser sensor capable of providing location, shape, and velocity data of surrounding objects, a sensor impaired by its high cost and the large amount of complex data generated; radar, a sensor useful for long range detection; and visual perception, i.e. using camera sensors to gather data about the environment, useful for parsing environmental objects intended for humans such as traffic signs. Data from camera-based visual perception systems is noted as being difficult to exploit, necessitating the use of machine vision and machine learning technologies for the extraction of useful information.

In their article, Kuutti et al. [3] present an overview and comparison of different localization techniques for land-based autonomous vehicles, such as LiDAR, radar, GPS, and camera-based. In the article, localization is used in the context of positioning either a vehicle or an object or road fixture near the vehicle with respect to a reference map. The different localization methods are compared using metrics for e.g. computational and monetary cost, accuracy, reliability, and general sensor performance limitations. Accurate localization data is established as being crucial to the operation of other functional systems of an autonomous vehicle; specifically perception, control, and (path) planning.

The authors conclude that the best solution is a hybrid approach of fusing data from multiple different sensors. The analysis performed suggests that no sensor technology used alone can provide the accuracy and reliability in different conditions needed for a truly autonomous vehicle, and that a sensor fusion approach can potentially provide an accurate and robust solution while keeping the system cost-effective. As an example, the authors note that while a combination of integrated GPS, IMU (Inertial Measurement Unit), and camera sensors can provide high accuracy localization at a low cost, the accuracy of this combination is not sufficient for autonomous vehicles,

and should be supplemented by LiDAR or radar sensor-based techniques.[3]

2.1.1 Maritime-Specific Application

The RSOA (Robot System Onboard Architecture) developed by Barbier et al. [4] is an attempt to produce a modular, generic, and decentralized software architecture for a fleet of autonomous maritime vehicles. The architecture is one of three parts of a larger control system for a number of cooperating, heterogeneous autonomous vehicles, with the other two parts being a distributed middleware system for communication, and a central mission management tool used in a remote command and control station.

The architecture presented is based on the ROS (Robot Operating System) framework. In the developed architecture, each vehicle communicates with the remote control station using radio, Wi-Fi, or acoustic signals (in the case of underwater vehicles). The communication is implemented using an open DDS (Data Distribution Service) protocol. The architecture was tested first using pure simulations, then simulations with hardware in the loop, and finally by experimentation on the open sea using near-complete systems, with some functionalities simulated due to safety concerns. [4]

In the presented system, the usage scenarios consist of a human operator remotely assigning high level tasks to the system, tasks being transit to a location, survey of an area, inspection of an object, etc. The tasks are then decomposed into generic actions, e.g. motion, perception, and manipulation, and relayed to a number of available vehicles, which then set out to perform the tasks given. The vehicles themselves do not have any onboard planning systems, and rely on the simplified action instructions provided to them from the mission management tool. [4]

During a real-life experiment, the authors encountered difficulties in implementing the DDS-based communication system on some of the processing hardware used, but report that the developed onboard architecture performed the received high-level tasks correctly. The authors conclude by stating that the modularity of the developed on-board architecture provided great benefits for both development and integration, as the genericity of the architecture facilitated deployment on the het-

erogenous hardware platforms of both vehicles and control stations, and the modularity of the framework allowed for platform development being done by several different project partners. [4]

2.2 Microservice Architecture

Microservice Architecture (MSA) is a relatively new software architectural style that has started seeing a lot of use in recent years. The style itself is considered by some to be either an improvement upon or a refined subset of a slightly older architecture, Service-Oriented Architecture (SOA).

2.2.1 Characteristics of a Microservice Architecture

While the microservice architectural style is yet lacking any clear definition, there are a number of common characteristics that can be used to outline the architecture. One characteristic that suggests an MSA being a distinctly different architecture from an SOA is decentralized control. A common part of an SOA application is the Enterprise Service Bus (ESB), a central message bus that manages application workflow, choreography, and communication. Microservice architectures, however, typically lack such a central governance mechanism, relying instead on the services themselves to apply their own communication and business logic etc. while leaving all the thinking out of the message bus, which is left to simply act as a router. [5]

The characteristic communication structure of "*smart endpoints and dumb pipes*", as Fowler and Lewis put it [5], results in an application that is ideally very cohesive and loosely coupled. This in turn provides benefits in evolvability, as the independent components (services in this case) are easier to develop, test, and deploy. Evolvability is especially beneficial in a novel system such as that of a research platform, as parts of the system are likely to change frequently.

Loose coupling means that a service should know as little as possible about other services it collaborates with, and that services are integrated into the system in such a way that changes to one service do not require changes to another. High cohesion is achieved by having related behavior collected in one place, and unrelated behavior elsewhere. Thus, if some behavior in the system needs to be changed, changes have to be made in as few places as possible. [6, p. 30]

Dragoni et al. [7] define a microservice as a "*minimal independent process interacting via messages*" and a microservice architecture as a "*distributed application where all its modules are microservices*". The term minimal is used to mean that a service should only be responsible for functionality to serve a specific purpose, such as a calculator service only providing functionality to perform calculations.

While there are no standards governing the size of microservices, a term often seen where best practices are discussed is "*small enough*". As the sizes of services grow smaller, the benefits from low interdependence increase, and managing a single service becomes easier. However, the resulting increase in the number of services does make the system more complex. The size of the individual services thus also depends on how well the overall architecture can handle the increased complexity. [6]

2.2.2 Advantages of a Microservice Architecture

Whether the intention is to migrate an existing system from a monolith to a microservice architecture or to develop a microservice-based system from the bottom up, it is worth considering what is to be gained by doing so. There are a number of issues and disadvantages with monolithic architectures that are either absent or whose impact can be minimized when using a microservice architecture, as presented in [7]:

- The complexity of large monolithic systems can limit their maintainability and evolvability, as adding and updating libraries can result in unwanted behavior in the system due to dependencies. With microservices, gradual transition to a new version is possible. New and old versions of a service can be deployed and run in tandem, and any dependency issues can be solved gradually, fostering continuous integration.

- Bugs can be hard to track down in a monolithic system due to the large code base. Conversely, the small size and independent nature of microservices can limit the impact of bugs while making finding and fixing them easier.

- Changes to a part of a monolithic system require a reboot of the entire system. In a microservice architecture, changes to a part of the system typically only involve a number of microservices, which can be rebooted without disrupting

the rest of the system. This reduces overall system downtime when rebooting, and greatly improves testability and maintainability.

- Developing a monolithic application entails being locked in to certain frameworks and programming languages. Microservices offer a bit more leniency in this regard, as only the communication technologies and protocols used between the services need to stay consistent; the technologies, languages, and frameworks used for the microservices themselves are up to the developers.
- The modules in a monolithic system may have significantly different needs regarding resources such as memory or computational capacity and may even require some additional components e.g. databases. Optimal deployment of such an application is difficult as a single deployment environment must satisfy all the needs of the system. Microservices, on the other hand, are inherently suitable for containerization, which enables a more flexible and configurable deployment environment.

2.2.3 Pitfalls in a Microservice Architecture

As with any system architecture, a microservice-based system does present some challenges that need to be acknowledged. Being an architectural style that is still considered to be in its infancy [7], an implementation of a system using microservices is especially vulnerable to bad design practices stemming from inexperience and from the use of a somewhat immature architectural style.

In a study conducted among developers experienced with microservice architectures, a number of so-called architectural "*bad smell*s" are identified. The term bad smells is defined as "*indicators of situations—such as undesired patterns, antipatterns, or bad practices—that negatively affect software quality attributes such as understandability, testability, extensibility, reusability, and maintainability of the system under development*" [8]. The five smells that are reported as most harmful are *wrong cuts, hard-coded endpoints, cyclic dependency, shared persistency,* and *API versioning.*

The *wrong cuts* smell occurs when microservices are split on the basis of horizontal layers (e.g. presentation, business, and data layers) instead of according to processes. This can lead to wrongful separation of concerns and increased complexity stemming

from unnecessary data-splitting. This smell can be avoided by analyzing the business processes and resource needs of the system.

Hard-coded endpoints refer to hardcoded IP addresses and ports between interconnected microservices. These become an issue when the location of a microservice needs to be changed, and can be avoided by using a service discovery approach instead of hardcoding.

Cyclic dependency is a result of an existing cyclic chain of calls between microservices such that e.g. service A calls service B, B calls C, with C finally calling A again. This smell can be avoided by detecting cycles and refining the data flow and dependencies in the system, as well as by applying the API gateway pattern [8]. The latter involves having microservices exposed to each other through a gateway layer that manages the connections between services [9]. The related smell *not having an API gateway* is also mentioned in the study [8], though reportedly this smell only becomes an issue when the amount of microservices in a system becomes so large that the communication between them becomes hard to manage.

The *shared persistency* smell occurs when multiple services access the same (relational) database, resulting in high coupling and reduced service independence. Proposed solutions include private data stores for each service or private tables in a database that different services have exclusive access to. Similarly, loss of service independence can also result from different microservices using the same libraries, indicated by the *shared libraries* smell. Modification of the shared libraries requires coordination between developers, and the resulting loss of independence could either be accepted or, alternatively, the common functionality from the shared libraries could be extracted to a new service.

A lack of semantic and consistent *API versioning* (e.g. v1.1, 1.2 etc.) can result in connection issues and unexpected behavior between services when API changes are made, e.g. data are sent in a different way or need to be requested in a different way than before. Consistent versioning will help indicate if services need to adapt to changes made in the way other services communicate.

2.2.4 Smells in the Automotive Domain

Lotz et al. posit in [10] that not all of the bad smells listed in [8] are necessarily applicable in automotive or embedded systems. In some cases, the bad smells are still relevant even in automotive or embedded applications, but the proposed solutions are less than ideal.

As presented in [10], the negative impact of the *wrong cuts* smell is significantly lessened in automotive or embedded systems. Since the capabilities in e.g. the car industry are typically separated along clear boundaries such as powertrain, chassi-, comfort, and entertainment systems, embedded systems operating within these clear boundaries are less affected by issues stemming from cross-functionality. As an example, if the services are assigned to perception (sensing), data processing, and finally actuation, the service boundaries are quite clear. The services can be developed and deployed independently, and the only cross-functionality required is the exchange of data.

Whether *hard-coded endpoints* are an issue or not is heavily dependent on the method of message transfer between services. In the automotive industry the commonly used method is broadcasting, where messages are sent to all control units and a message identifier tells the receiving microprosessor if the message is to be processed. This means that the issue of *hard-coded endpoints* never comes up.

Just like in more general applications, *shared persistency* (multiple microservices sharing data-storage spaces) can lead to problems even in the automotive domain due to decreased system independence and higher coupling. However, due to resource constraints and cost issues, private data storage for each service is not always feasible. Private data storage could nonetheless be advantageous in some safety-critical systems and in systems where resource constraints are less of an issue.

The usage of *shared libraries* could pose significant risk through the introduction of a single point of failure, as any faults in a shared library could lead to issues in multiple subsystems. In the case study by Lotz et al. this problem was entirely avoided by fully integrating the required common libraries into the deployment image of the service, leading to preserved independence.

2.2.5 Service-to-Host Mapping

In their case study [9], Taibi et al. found that the most prevalent deployment pattern in microservice architectures is the *multiple services per host* pattern. In this pattern, multiple microservices are deployed on a single host (node), each running either in a container or a virtual machine. This fits well with the general principles of an MSA, as it promotes scalability and performance; containerized service instances are easy to deploy according to the system's needs. Containerization and virtual machines will be discussed at more length in the next section.

An alternate method, the *single service per host* pattern is also mentioned in [9]. In this pattern, each service is deployed on a separate host. The authors note that while this could eliminate resource conflicts by way of isolation, overall system performance and scalability would be drastically reduced. Additionally, this pattern is considered counterproductive, as deploying each service on a separate node violates the basic principles of the microservice architectural style.

As Newman points out [6, pp. 116-118], deploying multiple services on a single host does have some drawbacks. For one, it makes resource monitoring more complicated, as tracking e.g. the CPU usage of a host as a whole might not be enough; a single service using an inordinate amount of resources could lead to throttling the other services sharing the same host. To detect this, the resource usage of the individual services also needs to be tracked. Multiple services per host also introduces somewhat of a single point of failure, as a host failing can take out multiple services.

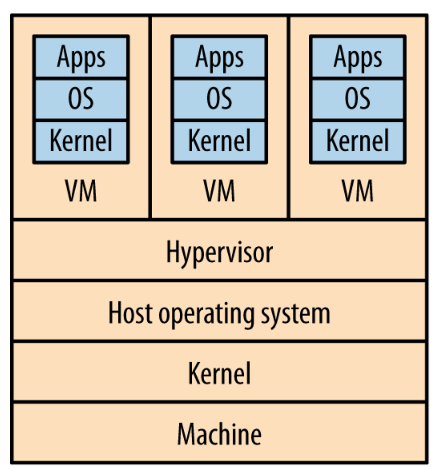
2.3 Virtualization

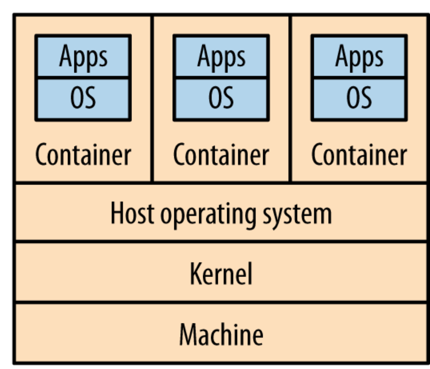
In order to deploy multiple services on the same host, the hardware needs to be split up somehow. A common way to achieve this is the use of virtual machines (VM).

2.3.1 Virtual Machines

When using virtual machines, a *hypervisor*, also called a virtual machine manager (VMM), runs on top of the native (host) operating system (OS) of a machine, providing one or more VMs with a share of the system's resources (CPU, I/O, memory etc.) through virtualization. The VMs in turn run a guest OS and kernel, which

are able to run as if they were running on physical hardware thanks to the resource abstraction provided by the hypervisor. [6, pp. 123-124] The kernel is the central part of an OS and handles most of the crucial tasks in a system, such as memory, process, and disk management. Every OS needs a kernel, and e.g. the Linux kernel is used in many different operating systems.





a) Standard virtualizations b) Container-based virtualizations (LXC)

Figure 1: Comparison of type-2 virtualization and lightweight containers [6]

A hypervisor running on top of the OS of the host machine is called *type-2 virtualization,* in contrast to *type-1 virtualization,* where the hypervisor instead operates directly on top of hardware. An illustration of type-2 virtualization can be seen in Figure 1a.

Virtual machines seem a good fit for microservices at a glance; the isolation provided by a VM can easily provide the operational independence which is a characteristic of services in an MSA [7], but there are some drawbacks. Both the hypervisor and the VMs need to set aside resources to do their jobs, and as the number of VMs increases, so does the overhead from managing them. At some point, this overhead becomes a limit to the number of times a physical infrastructure can be sliced up. [6] As one of the core principles of a microservice architecture is a small service size, even a moderately sized, MSA-based application might run into significant overhead if VMs are used.

According to [11], cyber-physical systems (CPS) such as those found in a self-driving vehicle can benefit greatly from the isolated environment and encapsulation provided by a VM. A *cyber-physical system* is comprised of a number of computerized systems that control physical resources to interact with their surroundings through collaboration and coordination, e.g. the collision avoidance and various autonomous driving functions in a self-driving vehicle. The ease of deploying new features and easier rollback in case of unwanted behavior are also mentioned as advantages in favor of virtualization. However, the prevalence of real-time systems and strict resource constraints in the automotive domain make VMs seem less appealing due to their resource overhead. Response time violations can lead to system failures, which can have disastrous consequences in the operating environment of a vehicle.

2.3.2 Containerization

An alternate method of virtualization is the use of container-based virtualization, also known as *containerization*. Container management tools such as Linux containers (LXC) and Docker provide benefits similar to VMs, but are much more light-weight and do not use a hypervisor, thus removing some of the issues with VMs that were mentioned earlier. Linux containers share the same kernel as the host machine and can run any OS as long as it uses the same kernel as the host. The containers work by operating in a separate subtree of the overall system process tree and are allocated physical resources by the kernel. [6] Linux containers are depicted in Figure 1b.

More precisely, the Linux kernel features responsible for the isolated user space instances used in containerization are *cgroups* (control groups) and *namespaces* [12]. Cgroups allow processes to be organized into hierarchical groups whose resource usage can be monitored and limited. The resource assignment and tracking are implemented in a number of subsystems (also called resource controllers) that each operate on a single resource (CPU, memory etc.). The hierarchical structure of cgroups enables imposing different limits on different levels of the hierarchy, with lower-level groups not being able to exceed limits set on higher levels but possibly having stricter resource limits of their own. [13]

Namespaces wrap global system resources in an abstraction, providing processes within a namespace a seemingly isolated instance of the resource in question. Figure

Namespace	Flag	Page	Isolates
Cgroup	**CLONE_NEWCGROUP**	**cgroup_namespaces**(7)	Cgroup root directory
IPC	**CLONE_NEWIPC**	**ipc_namespaces**(7)	System V IPC, POSIX message queues
Network	**CLONE_NEWNET**	**network_namespaces**(7)	Network devices, stacks, ports, etc.
Mount	**CLONE_NEWNS**	**mount_namespaces**(7)	Mount points
PID	**CLONE_NEWPID**	**pid_namespaces**(7)	Process IDs
User	**CLONE_NEWUSER**	**user_namespaces**(7)	User and group IDs
UTS	**CLONE_NEWUTS**	**uts_namespaces**(7)	Hostname and NIS domain name

Figure 2: List of Linux namespace types [14]

2 lists the namespace types available and what resources they isolate. Changes made to a global resource are visible to the other members of a namespace and invisible to other processes. [14]

2.3.3 Docker

Docker is an application management and development platform focused on separating applications from infrastructure through containerization. To containerize an application, it is first made into an *image* using a *Dockerfile*, which is a read-only template containing build and run instructions for a Docker container. Typically, an image is based on an operating system image, which can then be customized by adding e.g. an application or a web server on top. The image can then be run, turning it into a *container*; the runnable instance of an image [15]. Built images can be stored locally or in the Docker *registry*, a repository which is used to hold and distribute user-submitted images. Images can also be pulled from the Docker Hub; a service provided by Docker which hosts images for community-driven projects, applications from external vendors, and official images such as operating systems and programming language runtimes [16].

Docker uses a client-server architecture, in which a user-interactive Docker *client* communicates with a local or remote Docker *daemon* using the Docker API through e.g. UNIX sockets or a network interface. The Docker daemon is responsible for managing images, containers, networks, and volumes. Linux kernel features such as the aforementioned namespaces are used to deliver functionality and isolation to the Docker architecture. [15]

2.3.4 Comparison of KVM, Xen, and Docker on ARM

In their study, Raho et al. [12] presented a performance comparison between Docker and two open-source hypervisors, KVM (Kernel-based Virtual Machine) and Xen, which was conducted on ARM platforms. Xen on ARM is a bare metal (type-1) hypervisor that only supports paravirtualization, i.e. the guest OS is aware of the virtualization and uses drivers and extensions to talk to the hypervisor. The hypervisor implementation of KVM on ARM is split into two parts, a 'highvisor' and a 'lowvisor', for security and portability reasons. The highvisor operates in kernel space and handles most of the typical hypervisor functionalities, while the lowvisor handles context switches and enforcing isolation.

The performance overhead of the three virtualization methods was tested using benchmarks for system performance (floating-point operations, system call overhead, process creation overhead etc.), network performance, scheduling, and I/O. The experimental results showed that there was very little performance overhead overall for all three virtualization solutions, with slight variations depending on the benchmark used. [12]

The authors note that while containers are nowadays preferred because of their ease of use and faster deployment due to multiple containers sharing an operating system, they do not provide the same level of isolation as hypervisors and are thus considered less secure. Because both methods have comparably low performance overhead, the authors suggest using a combined approach of containers running inside virtual machines to provide the benefits of both technologies. [12]

No scalability tests with multiple concurrent virtual machines or containers were performed during the study. While single instances using the different virtualization methods may have comparable performance overhead, the question still remains of whether containers are more resource efficient when the number of instances on a single host machine increases.

2.3.5 Virtual Machines and Containers at Scale

In a study by Zhang et al. [17], virtual machines and containers were compared within a big data environment. Although a big data environment is not strictly

relevant to e.g. autonomous vehicles, the study does provide some valuable insight into how VMs and containers behave at scale, as well as how the virtualization methods behave under heavy load.

According to the authors, VMs have an issue w.r.t. scaling in that they are not very good at sharing resources; multiple VMs can easily share the same hardware but cannot easily shift resources between each other. This can lead to an imbalance at high loads, where application performance degrades even though there are free resources available. Multiple VMs requiring their own operating systems and corresponding images without the ability to share them between VMs are also factors that can lead to inefficient scaling. Containers avoid these issues by sharing the host OS and by using layered images, respectively. [17]

The study used KVM and Docker to manage the VMs and containers respectively; an open-source cluster computing system, Apache Spark, was used as a platform to perform the computation workloads for the study by running a variety of compute-intensive Spark applications. In addition to investigating the convenience and ease of setting up a system using containers vs. using VMs, the study sought to evaluate the impact VMs and containers have on the performance and scalability of different big data workloads. [17]

The deployment convenience comparison was carried out by measuring the time it took to set up a Spark cluster comprised of three VMs or containers on a single machine. Setting up a containerized cluster took on average 23 minutes, while setting up a cluster using VMs took on average 46 minutes. The differences in setup times were mostly in the image building phase, where setting up the VMs took longer because of the need to install an OS for every VM and because the VM images needed to be copied multiple times. [17]

Significant differences were also observed w.r.t. bootup efficiency. A single machine could reasonably hold many more containers than VMs with a machine starting to become unusable at 250 deployed VMs compared to the same machine happily running 512 Docker containers. When sequentially booting up the images, it took 479 seconds to boot up 256 containers while the same amount of VMs took over 50 times longer to boot up. [17]

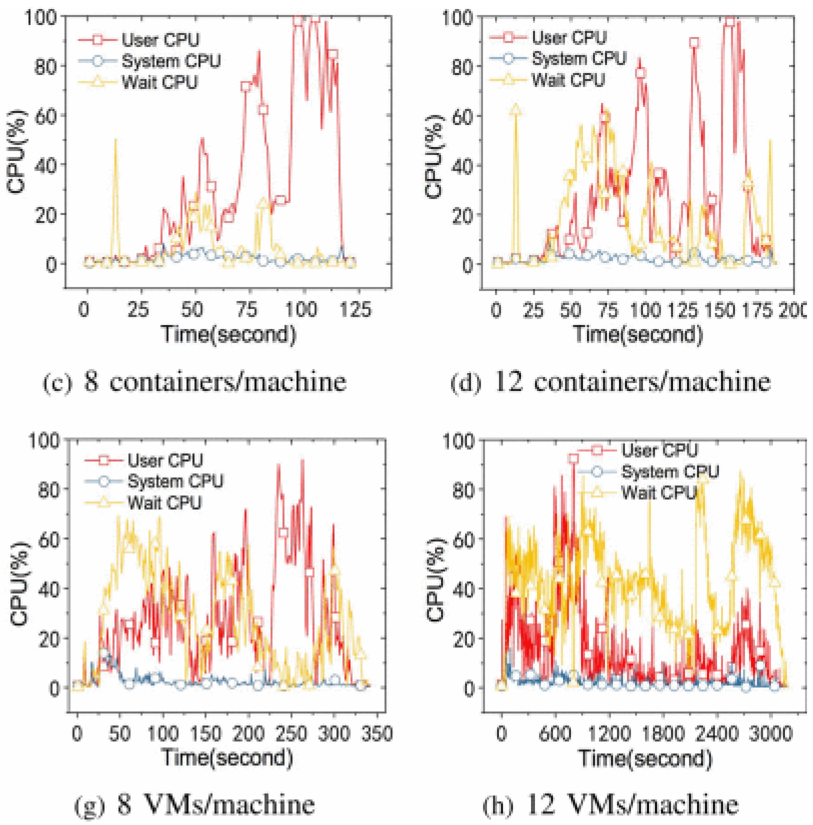


Figure 3: Average CPU utilization, pagerank workload [17]

Performance-wise, there were few differences between VMs and containers when the number of instances was low. At a cluster size of 4, i.e. one instance on each of the four physical machines used, the measurements of task execution times, CPU usage, and memory usage were comparable between the two virtualization methods. When the cluster sizes increased further, however, the differences started growing larger with containers performing noticeably better than VMs. At a cluster size of around 40, the tasks performed in the VM environment took over ten times longer to complete than those performed in a containerized environment. Beyond a cluster size of 48, execution of the workloads in the VM environments failed to complete altogether while the containerized applications still managed to complete their tasks in a reasonable amount of time. [17]

Figures 3 and 4 illustrate average CPU and memory utilization measurements for clusters of different sizes running the pagerank workload. Pagerank is an algorithm that counts the number and quality of links to a webpage in order to provide an

estimate of how important a given page is. The input data used for the workload consists of 300 000 pages, with the job running for three iterations. [17] The figures are cropped to only include the larger clusters, as the smaller clusters show comparable results for both VMs and containers.

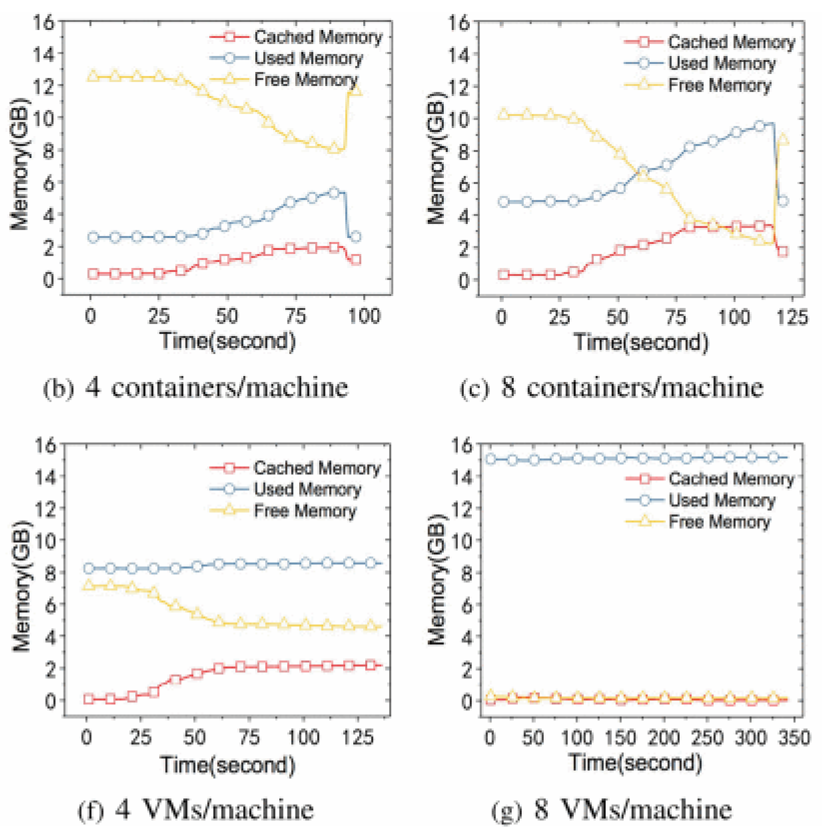


Figure 4: Average memory utilization, pagerank workload [17]

As the pagerank workload is a user-level application, most of the CPU usage is at the user level, away from the kernel. As the cluster size grows larger, however, the CPU wait times increase significantly, mostly due to long disk I/O latency and memory swap. As seen in Figure 3, at the cluster sizes of 32 and 48 (8 resp. 12 VMs or containers per machine) the CPUs in the VM environment in particular spend a significant amount of time waiting, leading to prolonged execution times. The proportional increase in CPU wait time is also true for the containerized environment, albeit to a much lesser extent. At high cluster sizes (≥ 48), tasks in the VM environment started failing due to timing out, likely because the CPUs spent most of their time waiting. [17]

The average memory utilization statistics paint a similar picture to the CPU utilization measurements, but the better scaling capabilities of containers become apparent earlier, at smaller cluster sizes. Because containers are generally better at memory allocation than VMs, the difference in memory usage is already apparent at two containers/VMs per machine, with the containers using less than 2 GB and VMs using 4 GB. As seen in Figure 4, the differences increase as the cluster size grows, and at 8 instances/machine the VMs are already using up all the available memory while the containers still have some free memory available. [17]

2.3.6 Combining Containers and Virtual Machines

Despite the advantages of using containers, there are situations where they might not be the best approach for virtualization. Firstly, containers are less secure as they are not as isolated from the host machine as VMs are, since containers use the same kernel as the host machine. It is thus easier to e.g. gain access to a container as a root user on Linux. Secondly, some applications on a host might need to use entirely different operating systems, which is not achievable with containers. [18] The negative aspects of the trade-off between containerization and virtual machines could feasibly be minimized by using a combined approach of running containers inside a VM. This approach could, however, come with the added cost of significant resource overhead, which may prove to be prohibitive especially for resource-constrained systems.

According to Mavridis et al. [19], containers perform best when run on bare metal as one of their main advantages is low performance overhead. In their study, the authors sought to investigate how much additional overhead is introduced by running containers inside virtual machines, and whether the resulting benefits of increased security and flexibility were enough to justify any additional overhead. The comparison in the study was performed between containers on their own and containers running within VMs, and the benchmarks used tested network performance, disk I/O, memory I/O, and CPU usage.

The study results showed that there was indeed a performance penalty that occurred when running containers inside VMs; containers on their own performed best during all the benchmarks, though the extent varied. The biggest overheads were observed in the benchmarks for disk I/O and network throughput, with performance reduc-

tions as high as 28% for disk I/O and 33% for network throughput. The biggest performance differences between just containers and VMs + containers were generally observed during the execution of smaller tasks, such as transmitting small (4-16 byte) packets with TCP during the network benchmark or solving smaller linear equation sets (200x200 matrices) during the CPU benchmark. [19] These results indicate that while running containers inside VMs does cause additional overhead, the amount is dependent on both the task performed and its size.

2.4 OpenDLV

OpenDLV (Open DriverLess Vehicle) [1] is an open-source software environment created to support the development and testing of self-driving vehicles. The system is entirely microservice-based, where applications are designed to be as self-contained as possible. In addition to source code, in order to facilitate deployment and ease-of-use the environment is also distributed in the form of Docker images, which contain all the libraries necessary to run the microservice in question.

2.4.1 Communication

Most of the communication that takes place within OpenDLV is in the form of UDP *multicast*. In multicast, information is sent to multiple recipients at once; in contrast to *broadcast*, where messages are sent from a single source to all members of a local network, and *unicast*, where messages are sent to a single recipient. When an OpenDLV microservice is started, it is given a numerical *session ID* from within the range [1,254] as a parameter. This ID is in fact part of an IP address; OpenDLV microservices exchange data by sending messages to UDP multicast address 225.0.0.X, where X is the session ID. All microservices within an OpenDLV session are able to communicate with each other, while services located in different sessions do not see each other and are thus separated.

It is worth noting that the term *session* in the context of the OpenDLV system carries a slightly different meaning than what is usually meant. In computer networking, a session typically refers to a limited-time communication, often stateful, between two parties or systems. In OpenDLV, a session is more akin to a communication group; by default, messages sent by a participant in a session are sent to all other

participants in that same session. Unless stated otherwise, the term *session* is used in this thesis to mean the latter.

2.4.2 Data Format

OpenDLV services communicate by sending message *envelopes*, which in addition to the actual payload (e.g. sensor readings, actuation requests, or information requests) contain metadata, i.e. some information about the contents of the message. The metadata contains timestamps indicating when the message was sent and received as well as when the contents (payload) were sampled, and also a *message identifier* which identifies the contents of the message.

An integral part of the OpenDLV software ecosystem is the *OpenDLV standard message set* [20]. All the data sent between the microservices in OpenDLV are encoded in Google's Protobuf [21] format. Protobuf is a platform- and language-neutral mechanism to serialize structured data. Serialization is the process of translating data structures into a format fit for storage or transmission so that they can be reconstructed later through deserialization. In Protobuf, the desired data structures are defined in a file, from which data access classes can be generated for various platforms and programming languages.

```
message opendlv.proxy.AccelerationReading [id = 1030] {
float accelerationX [id = 1];
float accelerationY [id = 2];
float accelerationZ [id = 3];
}
```

Figure 5: OpenDLV *AccelerationReading* message specification

OpenDLV natively implements Protobuf, and the data structure definitions are stored in a single file called the *OpenDLV standard message set*. The file itself is portable and is intended to be bundled with each service to enable the serialization and deserialization of data in a manner that is consistent between the different parts of the system.

As an example, Figure 5 shows an excerpt from the standard message set. The ID 1030 uniquely identifies the message as an acceleration reading, and the fields numbered 1-3 identify the readings themselves, in this case acceleration readings for each of the three axes. The number 1030 is the *message identifier* which is part of the metadata of an envelope, and the fields 1-3 are accessed when the envelope is deserialized at the receiving end. A message can thus be identified either by its name, or by its unique ID.

By convention, OpenDLV imposes one additional restriction upon its data structures which is not a standard part of Protobuf. To preserve backwards and forwards compatibility, message identifiers are unique across the system and should not be changed. This is to prevent future issues, e.g. a service receiving a message with the ID 1030 containing something else than an acceleration reading.

The standard message set is one of the system's great strengths as it creates a layer of abstraction between the hardware-level components and the higher-level system functions. As an example: Suppose a service C expects to receive latitude and longitude data from a GPS service in the form of a *GeodeticWgs84Reading* message, which is one of the standard messages. Initially the messages are provided by service A which interfaces with a GPS module. Then, the GPS module is changed, and a new service B is created to interface with it. The two GPS modules need to be interfaced with in different ways, e.g. one could be providing NMEA data via TCP and the other via a serial interface, but because both services A and B send *GeodeticWgs84Reading* messages, the change has been made in a completely transparent manner to service C. A change in one service not requiring a change in another is an example of loose coupling, which is one of the cornerstones of a microservice architecture. On the other hand, if both services A and B are used simultaneously, the messages they send can be assigned unique sender identifiers, so that the receiver can differentiate between the two senders.

2.4.3 Libcluon

Libcluon [22] is a single-file, header only C++ library which is used to realize the core functionality of OpenDLV. The main features of Libcluon primarily revolve around data representation and a portable implementation of publish/subscribe communi-

cation. Libcluon natively supports multiple data serialization formats such as JSON, csv, Protobuf, and OD4. OD4 is a data serialization format used by OpenDLV. Runtime conversion between these formats is also supported, and the message compiler implementation of Libcluon solely depends on a modern C++ compiler, i.e. C++14 or newer.

The Libcluon library is designed around compatibility and portability; all OpenDLV microservices use the single-file library, and developing microservices compatible with the OpenDLV software ecosystem is as simple as including the library in a project. The Libcluon GitHub page [22] includes a link to the API documentation of the library, as well as a set of tutorials covering some basic features.

3 Implementation

In this chapter, the practical implementation done for this thesis will be presented, beginning with the high-level design of the overall system and continuing with the design and function of the individual microservices.

3.1 System Design

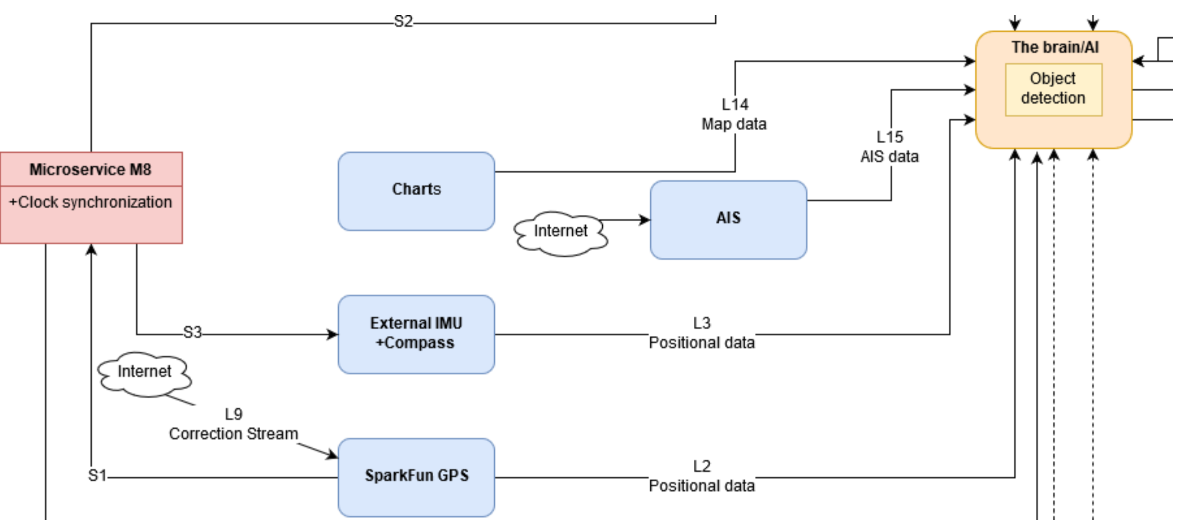


Figure 6: Excerpt from the system specification diagram

In the early stages of development, a high-level system design diagram, with some accompanying documentation, was created for the boat platform. This was not primarily intended to provide a comprehensive description of a finished system. Rather, it was made to be used as a tool to get a rudimentary overview of the different systems on the envisioned autonomous boat platform, be they planned or already partially implemented. The diagram would also serve to illustrate the communication pipelines in the system and to identify where in a pipeline a data-processing microservice should be placed. This system description could then be used to guide the development on the autonomous boat platform. The full diagram can be found in Appendix A.

The system diagram was fully expected to change rapidly during the early design process as understanding of the software ecosystem improved and new hardware acquisitions were planned, and was as such developed and refined in an iterative

process. As the envisioned system is a decentralized one, the diagram reflects that design as well. Each blue element in the diagram (as seen in the excerpt in Figure 6) is assumed to have a microservice associated with it, responsible for the functionality of the element. As an example, the microservice associated with the SparkFun GPS module in Figure 6 would be responsible for receiving GPS data, possibly combining positional data with a correction stream received over the internet, and performing a conversion into the format expected by other parts of the system.

3.1.1 The Brain

The most significant change between diagram versions was the expansion of the central element, labeled "The brain/AI" in the main diagram, into a separate diagram, which can be seen in Figure 7. While the overall system is at its core decentralized, many of the mission-critical elements in the system, i.e. those responsible for navigation, path planning, situational awareness, and steering, are closely connected among themselves and together form an arguably central element in the system. This element is also the recipient of a majority of the data flow in the overall system. While this somewhat blurs the line between centralization an decentralization, the overall system can still be considered a decentralized one as all of the other elements operate independently. In addition, the system specification makes no assumption regarding what hardware each of the elements will be running on. Even the software of the central navigation/situational awareness module is likely to consist of a number of microservices running on multiple discrete pieces of computing hardware.

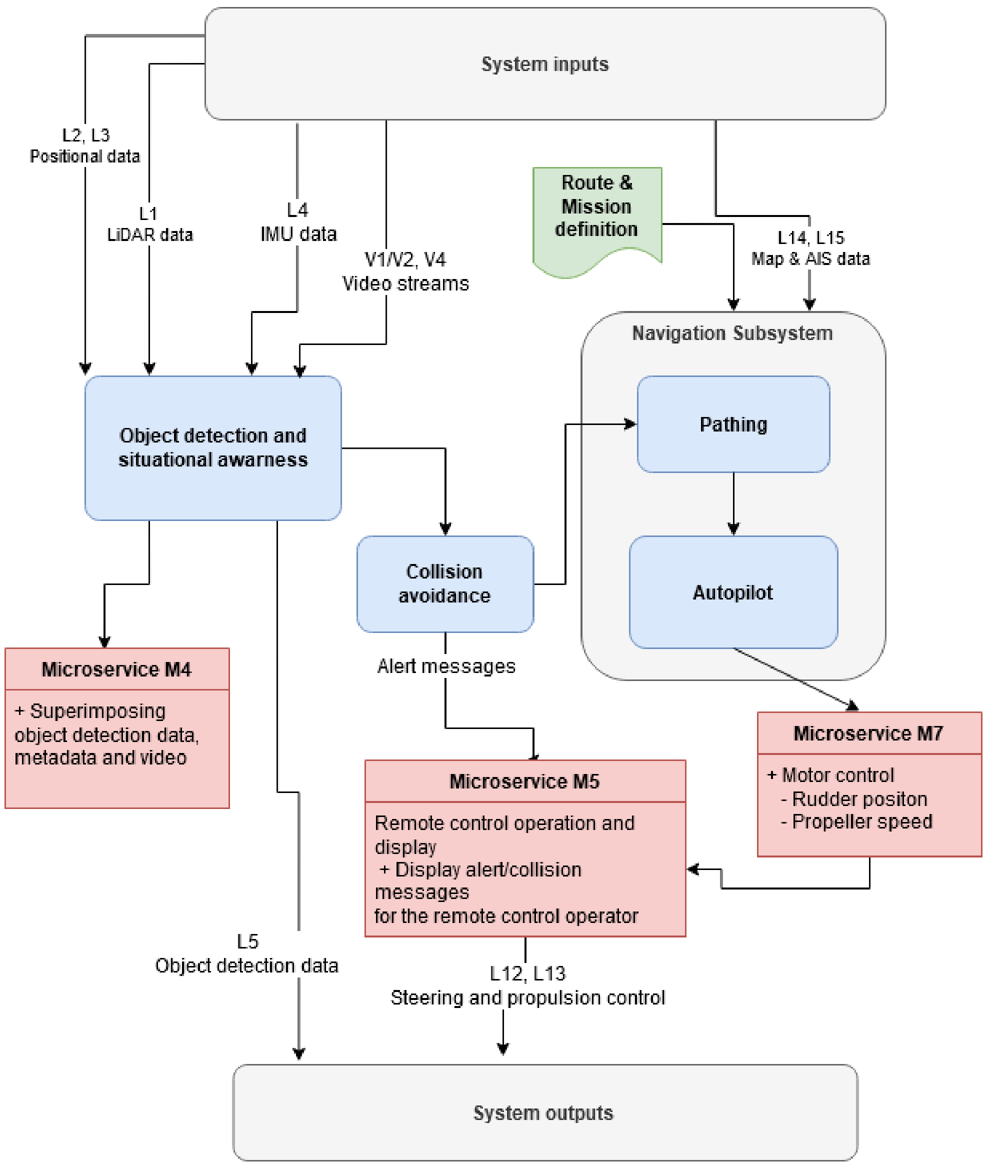


Figure 7: Diagram of the central element of the system, “the brain/AI”

The primary purpose of the diagram is to depict what data are needed from the rest of the system as well as which elements need what data to operate. A few key pieces of functionality are defined as microservices and placed into the diagram to illustrate their position in the information pipeline, but otherwise the diagram is kept at high level of abstraction. As an example, the navigation system in Figure 7 uses route and mission definition data to perform long-term navigation, while the situational awareness and collision avoidance modules provide data needed for short-term path planning to the navigation system. In turn, the situational awareness module needs

data from a number of on-board sensors to perform its tasks. The data flow defined in the system design diagram can then be used to guide the development of the different modules.

While the diagram in Figure 7 shows the microservice responsible for controlling the motor receiving input from an (autonomous) navigation subsystem, the motor control system developed for this thesis is somewhat different in design. As the eventual goal of the boat platform is autonomous operation, the system design diagram reflects that fact. However, the implementation of an autonomous control system is beyond the scope of this thesis, and the manual remote control system presented in this thesis is intended as an intermediate step on the path towards autonomous operation, and will be discussed in more detail in Section 3.2.

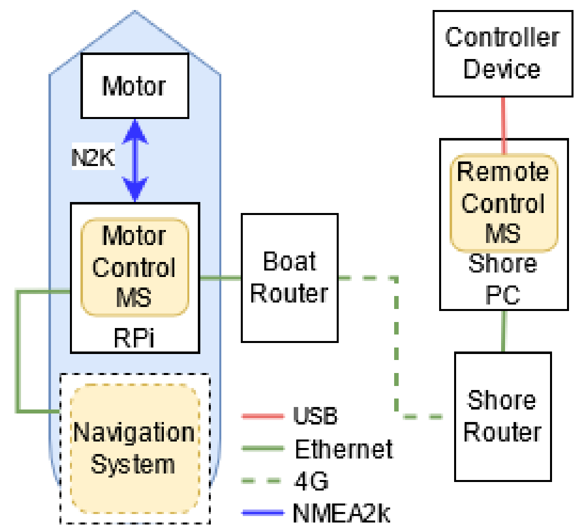


Figure 8: Motor remote control system overview

A system diagram over the remote control system can be seen in Figure 8, where a microservice interfacing with a controller device on the shore sends control messages over to the boat, and the motor control microservice actuates the motor. The blue arrow labeled "N2K" in Figure 8 indicates the placement of an NMEA2000 bus, which will be discussed in Section 3.2.4. The common elements between the system design diagrams depicted in Figures 7 and 8 are the motor control microservice, i.e. the software that actuates the physical motor, and the future navigation system. The difference is what element sends control messages to the motor control microservice.

For autonomous control, the messages are sent by the navigation system, and for remote control, the messages are sent by the remote control microservice. The motor control microservice is designed in such a way that the transition from manual control to autonomous control will require no changes to the motor control microservice, which will be used in both cases.

3.1.2 Video Streaming

Along with some of the elements providing e.g. maps/charts, IMU data, and GPS data to the navigation/object detection part of the system, the system design diagram excerpt in Figure 9 also includes one of the most crucial parts of the system of the boat platform, i.e. video streaming. While the hardware configuration of the video streaming system is clear in that a 360 degree video view is defined as a requirement for autonomous operation, there are some options when it comes to the data flow and architecture of the video streaming system, as indicated by the dashed lines coming out of the camera module element in Figure 9.

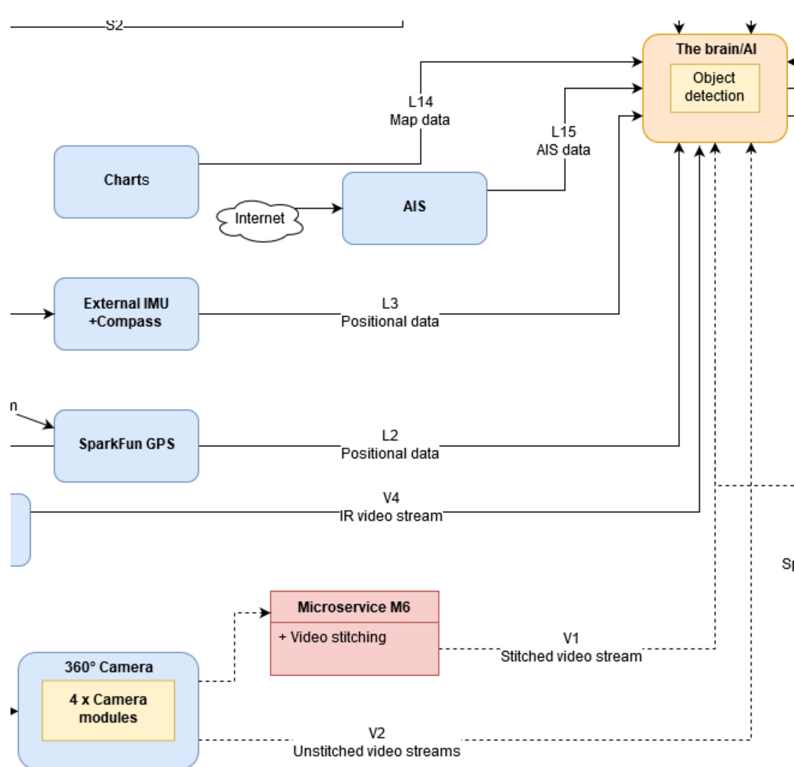


Figure 9: System overview diagram, video streaming

The existence and placement of the video stitching microservice, the purpose of which would be to stitch together and eliminate overlap from four video streams into one single 360 degree video, depend on whether e.g. object detection can be feasibly performed on 360 degree video, or whether it is to be performed on a number of discrete video streams instead. Likewise, there are some options regarding video display on the shore; while a display of four video streams is enough to provide the required 360 degree view, a stitched-together 360 degree video could e.g. be used in conjunction with a virtual reality headset. Finally, the system design also leaves room for the object detection being performed either on the boat itself or on the shore side. In the latter case, raw video data would be sent to the shore or edge to be processed before the relevant information needed for e.g. collision avoidance is sent back to the boat. It should be noted that implementing object detection, video stitching, and a 360 degree video display are beyond the scope of this thesis, and only the implementation of sending the four discrete video streams from the boat to the shore will be presented.

3.1.3 System Requirements

Table 1: List of requirements for thesis-related implementations

1: Motor control and Remote control
1.1: The boat should be controlled remotely from the shore with the operator having a visual contact with the boat.
1.2: Input from a controller on the shore should be translated to motor actuation.
1.3: The relation between controller input and actuation should be defined and consistent.
2: Visualization
2.1: The remote control operator should have a display of the information required to operate the boat:
2.1.1: Heading
2.1.2: Position
2.1.3: Speed
2.1.4: Location of nearby obstacles (video feed)
2.1.5: Network diagnostics with color-coded lines

During project planning, a preliminary requirements analysis was performed for the boat project. The analysis primarily focused on the functionality needed for what

were defined as the two first major milestones in the development of the boat project; *remote control operation*, i.e. steering the boat from the shore with a direct line of sight to the boat itself, and *remote control using visualization*, i.e. steering the boat using a combination of a live video feed and telemetry data instead of direct line-of-sight. The requirements concerning the implementations discussed in this thesis are largely functional in nature, and can be seen in Table 1.

3.2 Motor Control

The purpose of the motor control subsystem is to enable manual remote control of an electric outboards motor which is situated on the boat. The subsystem consists of two microservices; one for sending steering information to the boat, and one for translating the steering requests into motor actuation on the boat.

3.2.1 System Description

The boat platform is equipped with an electric outboard motor, an Xi5 trolling motor manufactured by Motorguide. A trolling motor is typically used as a secondary means of propulsion on a fishing boat, to facilitate precision maneuvering. The motor is equipped with a magnetometer/GPS module and has some rudimentary navigation capabilities in the form of recording a route being traversed by logging position data which can then be used later to automatically traverse the same route. The motor can also be controlled externally through an NMEA 2000 (NMEA2k) bus.

3.2.2 Motivation

The implementation of motor control was considered a good first step towards autonomous operation, primarily because it could be developed and implemented well before any actual navigation systems were in place. The eventual navigation system of the boat platform will be implemented using machine learning models, and as such it is beyond the scope of this thesis. Similar situations, where a system is to be implemented with future extensions in mind, are expected to arise as the development of the boat platform continues.

Another deciding factor was that tests of the motor control system could be performed on dry land and indoors, unlike a navigation system that relies on e.g. GPS;

indoor tests of a navigation system would be problematic because GPS positioning requires an unobstructed view of the sky. Additionally, because the position of the boat platform would remain static, the position would have to be simulated somehow for any realistic tests to take place.

The intent is that the lessons learned from the development and implementation of the motor control subsystem will be useful in the further development of the boat platform. The subsystem as it is designed will encompass multiple aspects and features that can be reasonably assumed to also be present in other, future subsystems on the boat platform:

- The receiver service takes direct action based on the messages from the sender.
- The subsystem is comprised of multiple microservices.
- The subsystem contains a microservice that is shipped as part of the OpenDLV ecosystem, which would have to be modified to better suit the boat platform.
- The subsystem contains a microservice that is not shipped as part of OpenDLV, but would still have to be developed with interoperability in mind.
- Creating one of the microservices requires integrating OpenDLV source code with source code not part of OpenDLV.

3.2.3 Remote Control Microservice

The purpose of the remote control microservice is to enable manual control of the boat platform's outboard motor. Initially, the service would be used for testing and tuning both the manual control of the motor and the rudimentary control loop of the steering subsystem as a whole. The microservice will also eventually be used during the late stage development and operation of the boat platform. Depending on the current mission of the boat, the ability to manually control it would either be required for the mission (e.g. during the field testing of some onboard equipment or systems) or useful as a fallback in case of a failure of the navigation system during autonomous operation.

To start the development of the remote control microservice, the microservice *opendlv-device-gamepad* was used as a starting point. The original microservice and its source code are distributed as part of OpenDLV, but as it is intended for steering a car, it would have to be modified in order for it to be used to control the outboard motor of the boat platform.

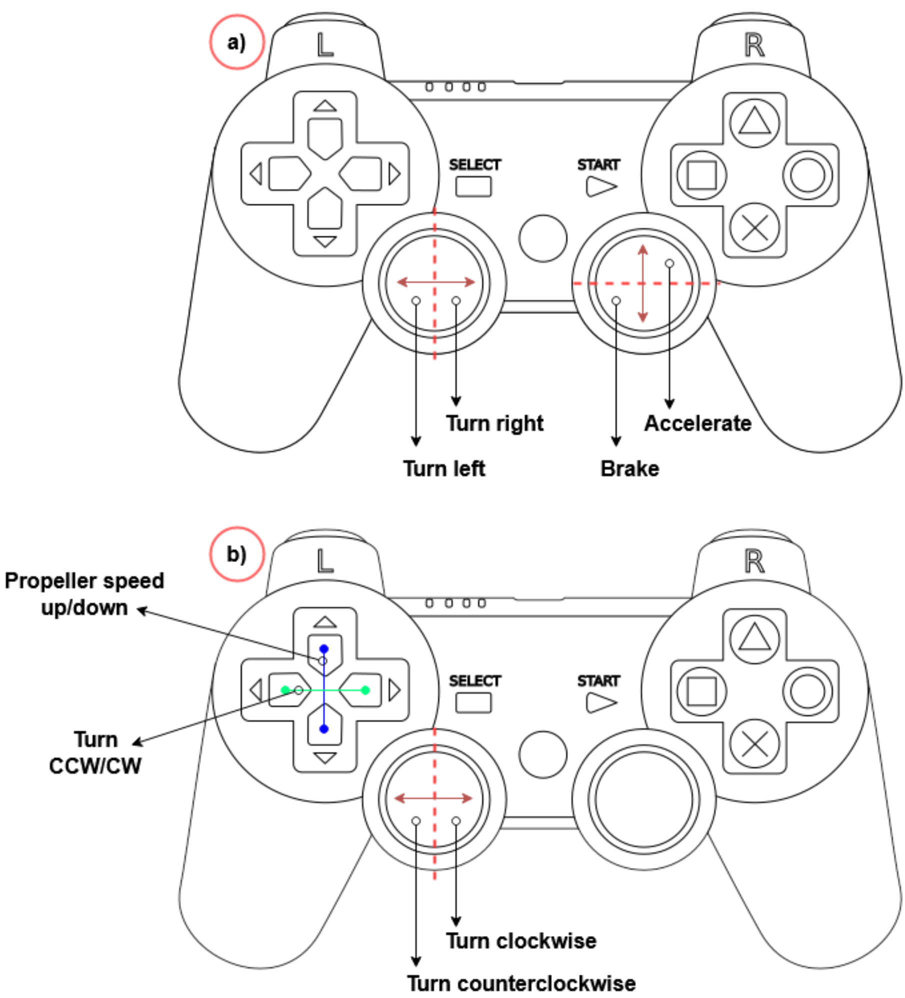


Figure 10: Control schemes for the original vs. the modified gamepad microservice. Base controller image from [23].

First, the control scheme would have to be changed. Figure 10a) illustrates the original control scheme, which is reminiscent of the controls used to steer a small remote-

controlled car. In the original scheme, the left analog joystick controls the steering and the right one controls the acceleration and braking. The amount of steering and acceleration is determined purely by the distance of the tilted joysticks from the center axes; the horizontal axis in the case of acceleration/braking and the vertical axis in the case of turning. When the joysticks are tilted, the input is then continuously translated into appropriate steering/acceleration requests by mapping joystick input to a range between minimum and maximum acceleration/deceleration/steering. These minima and maxima are given as parameters when the microservice is started.

The control requests are then serialized into *ActuationRequest* messages, each of which contains both the desired acceleration and steering in addition to a boolean value indicating whether the message contents are valid. As it is the standard for most OpenDLV microservices, the messages are then sent using UDP multicast to the session to which both the gamepad microservice and any intended receiver microservices belong. As discussed in Section 2.4.1, multicast is similar to broadcast but the intended message recipients are a select group of nodes on a network instead of all of them. The rate at which the messages are sent over the network is also given as a parameter (in Hz) when the microservice is started, but the internal check for whether there is new input data to be processed and serialized into a message occurs at a fixed rate of 50 Hz.

The new control scheme is illustrated in Figure 10b). The directional pad buttons on the left (indicated in green and blue in the figure) were designated for stepwise control; a single press of either the up or down button would change the propeller speed by some set amount and a press of the left or right button would rotate the motor by a set amount. Because step control was deemed adequate for changing the propeller speed (i.e. the speed of the boat), the acceleration and braking controls on the right joystick were removed entirely.

```
message opendlv.proxy.ActuationRequest [id = 160] {
float acceleration [id = 1]; // Unit: percent
float steering [id = 2]; // Unit: degrees
bool isValid [id = 3]; }
```

Figure 11: *ActuationRequest* message specification

Initial testing suggested, however, that stepwise control of the motor's rotation was insufficient. If a single increment or step were too large, fine-grained control and subtle adjustments of the rotation would be difficult. Conversely, if a single increment were too small, large changes in the motor's rotation would be difficult. To resolve this issue, the left joystick was assigned for continuous control. During continuous control, keeping the left joystick tilted in either direction will result in the continuous creation and sending of *ActuationRequest* messages, with the intent of turning the motor at a steady rate. The exact rate of steering is determined by the distance between the tilted joystick and the vertical center axis. The maximum and minimum steering rates, which determine how fast the motor will turn when the joystick is tilted all the way to the left or right, are given as parameters when the microservice is started. The contents of the *ActuationRequest* message are shown in Figure 11.

Because the outboard motor can freely rotate 360 degrees and can thus be placed on either the bow or the stern of the boat, and because the ultimate placement of the motor was unknown at the time of development, it was decided that the values of the *steering* field would represent counterclockwise and clockwise turns of the motor in degrees for positive and negative values respectively. As the correlation between the turning direction of the boat and the direction the motor rotates depends on the placement of the motor, the intent is that the translation of motor rotation into boat turn direction will be implemented later, in the navigation subsystem. For similar reasons, the *acceleration* field solely represents the desired speed of the propeller of the motor and not the speed of the boat as a whole. This distinction is important since adjusting the propeller speed can have multiple effects on the speed of the boat depending on the position and rotation of the motor.

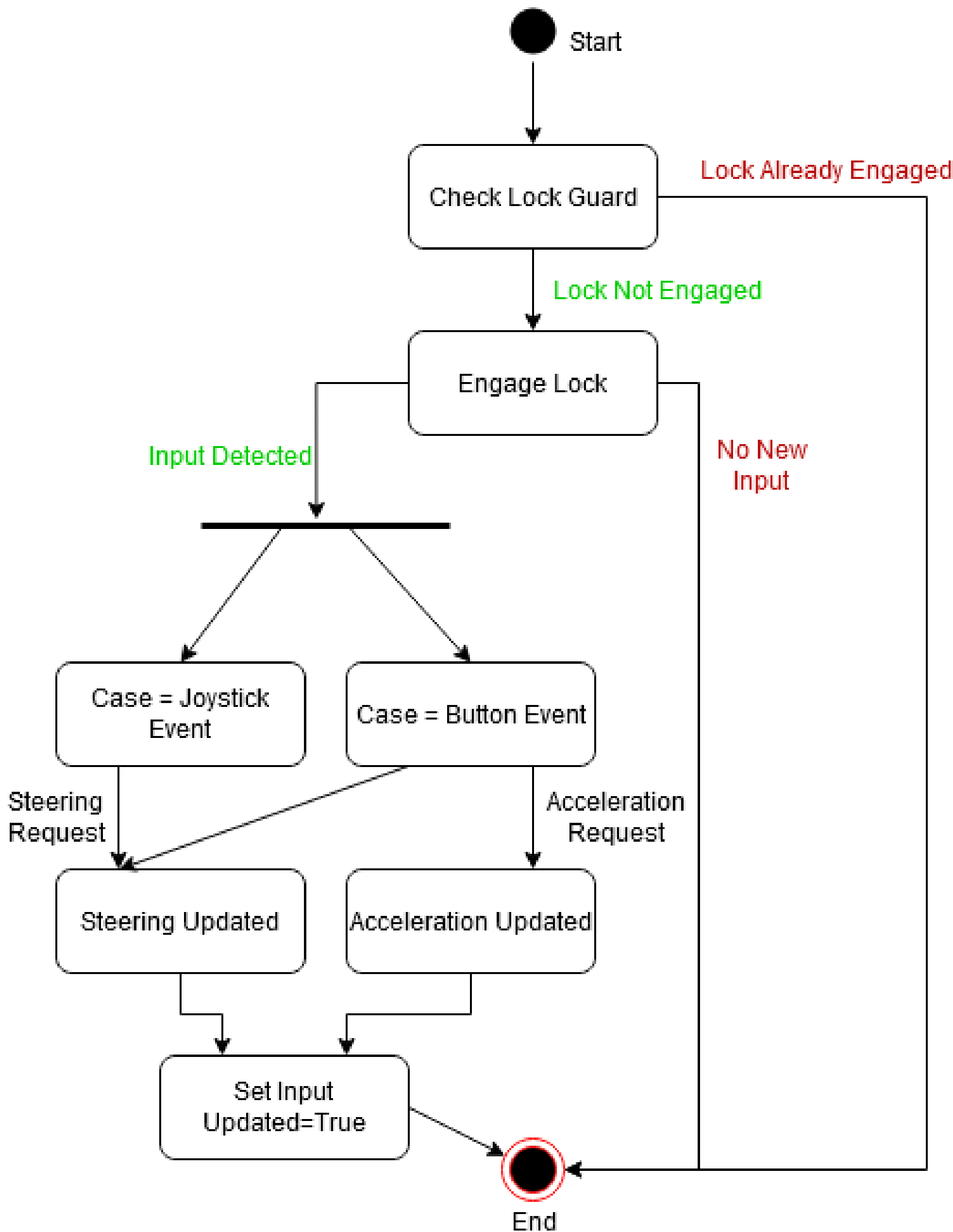


Figure 12: State diagram for the controller input thread

In the remote control microservice, the reading of controller input and the sending of messages are conducted in two separate threads. Multithreading is used to facilitate setting different rates for the sending of messages e.g. to lower network usage while maintaining the ability to quickly react and respond to controller input.

To prevent race conditions where the two threads attempt to access and/or modify the same information simultaneously or in the wrong order, a *mutex* is used. The mutex class provides a mechanism to protect shared data from simultaneous access by way of a locking mechanism [24]. Figures 12 and 13 depict state diagrams for the threads that handle controller input reading and message sending respectively. Before anything else happens in a thread, an attempt to lock the *valuesMutex* is made. If the mutex is locked, thread execution blocks until the mutex is unlocked. If the mutex is found to be unlocked, it is subsequently locked and thread execution proceeds. When thread execution is finished, the mutex is automatically unlocked.

As illustrated in Figure 12, after the mutex is locked the input reading thread proceeds to update the appropriate *steering* and *acceleration* variables. In the case of button events, which occur when the directional pad buttons on the controller are pressed, the variables are changed by a set amount of ten, indicating a requested change in motor rotation by ten degrees or propeller speed by ten percent. In the case of joystick events, which occur when the left joystick is tilted, the steering is updated by an amount which corresponds to how much the joystick is tilted. Lastly, one of two *input updated* variables is updated, signaling to the message sending thread that there is new information to process and whether the objective is step control or continuous control.

The execution of the message sending thread is time-triggered, and it occurs at a variable frequency given as a parameter to the microservice. When there is new input to process, the message sending thread first differentiates between requested step control and continuous control, as seen in Figure 13. The distinction is important mainly in the case of continuous input as the objective with continuous control is to rotate the motor at a set rate rather than by a set amount. Because the rate at which steering requests are sent to the motor can vary while the steering rate should stay the same, the value of the *acceleration* field in the constructed *ActuationRequest* is modified relative to the message sending frequency before the message is sent.

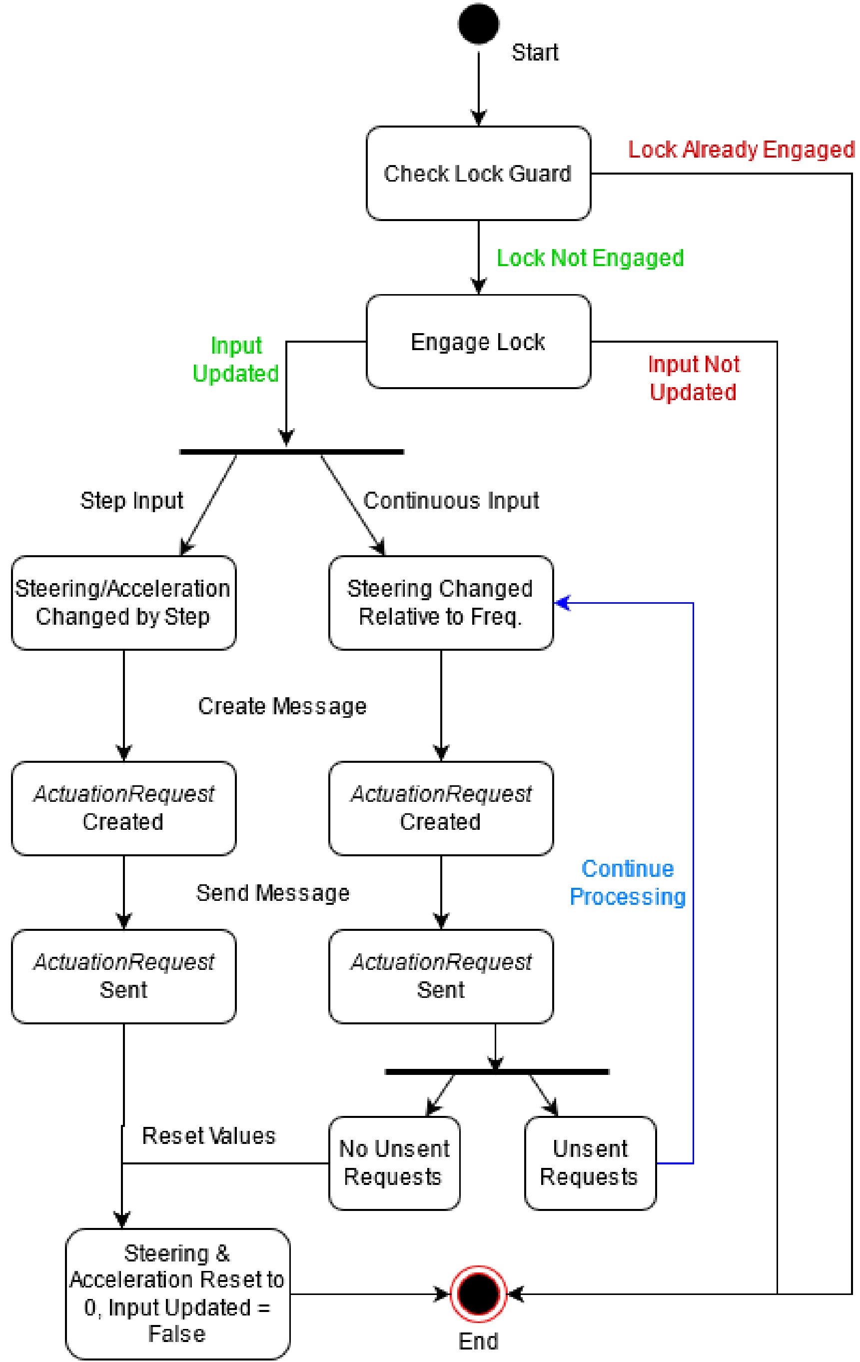


Figure 13: State diagram for the message sending thread

For step control, all the steering and acceleration requests from the controller that occur between iterations of the message sending thread can be packed into a single *ActuationRequest*, and after the message is sent a flag is reset indicating that there are no more requests to process. For continuous control, messages are continually created and sent as long as there are unsent requests as a result of the joystick being tilted.

3.2.4 Motor Control Microservice

The purpose of the motor control microservice is to receive *ActuationRequest* messages from other microservices, deserialize them and translate them into actuation instructions for the outboard motor. Initially, the only source of messages will be the remote control microservice, but in the future these messages will also be sent by an (autonomous) navigation system. The motor control microservice was designed in such a way that the eventual move from manual remote control to autonomous control would be as painless as possible from a development standpoint. Essentially, this was done by having the motor control microservice make no assumptions about whether or not the origin of received *ActuationRequest* messages is the remote control microservice described in Section 3.2.3. This means that the motor control microservice will readily accept actuation requests from e.g. a future navigation subsystem.

Before the development of the motor control microservice could begin, there was an obstacle that needed to be removed. Apart from a wireless remote controller, the only way to control the motor is through the use of NMEA2k messages sent over the NMEA2k bus. Therefore, for programmatical control of the motor to be at all possible, the use of NMEA2k messages would be unavoidable. There was no public API for the motor to be found, nor could a specification of the manufacturer-proprietary NMEA2k messages used to control the motor be found. Thus, the messages needed to be reverse-engineered, which was done purely for interoperability purposes, and in cooperation with Dr. Sébastien Lafond.

The NMEA 2000 (NMEA2k or N2K) standard describes a serial data communications network based on the CAN (Controller Area Network) used primarily in automotive applications, and is designed to interconnect devices on marine vessels.

The network is bi-directional and supports multiple transmitters and receivers without the need for a central controller. [25] The network consists of a backbone or bus, which provides both power and a communication interface for up to 50 physical devices, or up to 252 network addresses. NMEA2k messages consist of one or more CAN frames, which each contain up to 8 bytes of data with a 29-bit identification field and some additional bits reserved for e.g. acknowledgment and error detection. NMEA2k messages are organized into *parameter groups* based on their function, and the group identifier PGN (Parameter Group Number) is included in the header of the message frame as part of the CAN identifier. [26]

It should be noted that the NMEA2k bus was included on the boat platform strictly by necessity, due to it being the only way to programmatically control the motor. If the motor is ever replaced with another model, the NMEA2k bus will likely be removed. Nevertheless, the presence of an NMEA2k bus does provide an interesting integration challenge to the project.

For the reverse-engineering setup, the following were connected to the NMEA2k bus:

- A fishfinder/chart plotter display unit capable of controlling the motor
- An Actisense NGT-1 NMEA2k-to-PC interface module
- The outboard motor
- A Raspberry Pi single board computer equipped with a Pi2CAN module capable of recording and replaying CAN messages over an NMEA2k bus
- A battery to power all of the above as well as the bus itself

Once the hardware setup was completed and with everything up and running, the NMEA2k messages sent over the idle system were observed through a PC connected to the bus via the Actisense NGT-1 module. By observing the headers of the messages sent on the bus (e.g. status messages to and from the motor) first when the system was idle and subsequently when the motor was controlled via the display unit, some NMEA2k messages were flagged as interesting i.e. likely to contain the sought-after motor control messages.

The next step was to attempt to verify that the previously flagged messages were in fact motor control messages. This was done by recording CAN messages from the bus during short time intervals while certain control actions of the motor were performed, e.g. increasing propeller speed by one step or rotating the motor clockwise for one second. The display unit was then disconnected and the recordings were replayed over the system. The software applications Actisense NMEA reader and Actisense EBL reader were used for recording and replaying NMEA network traffic, respectively. During the replays, bus traffic was observed on a PC and special attention was paid to the messages flagged as interesting in the previous step.

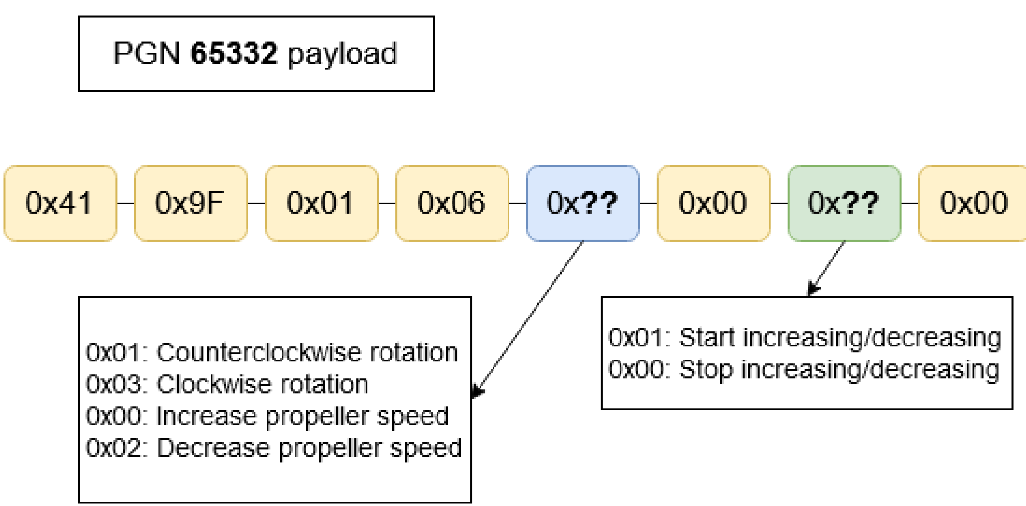


Figure 14: The 8-byte payload of the NMEA2k message with the PGN 65332

The NMEA2k message with the PGN 65332 was discovered to be used for all the desired actuation requests to the motor: *propeller speed up, propeller speed down, turn motor clockwise,* and *turn motor counterclockwise.* The byte of the message payload that determines the control action is marked with blue in Figure 14.

What determined the amount of actuation, however, was not readily apparent. After some more recording and replaying of NMEA2k bus traffic, it seemed that there was some relationship between the amount of actuation and the rate at which PGN 65332 messages were sent. Additionally, it was observed that for each discrete control action, the value of another byte in the payload was changing between 0x01 and 0x00 (marked with green in Figure 14). The assumption was made that a full actuation request consists of two parts; a start and a stop message, with the delay between the

messages determining the amount of e.g. turning the motor. To attempt to confirm this, an ad-hoc program to send the two-part messages with varying intervals was created.

To create the above program, a C++ library for creating NMEA2k bus devices and sending messages was used [27]. The library includes several tools related to the NMEA2k bus, but for now only the ability to create a PC-to-NMEA2k interface and to construct and send NMEA2k messages were needed. By using the ad-hoc program, it was confirmed that the assumption described above was true; sending a pair of PGN 65332 messages with a delay in between did result in actuation proportional to the delay. During testing, a slight inconsistency was noticed, however; turning the motor in one direction and then back in the other direction with the same delay between the start and stop messages did not result in the motor turning back to the starting position as expected. The difference was small, but still perceptible.

What remained was to create the actual motor control microservice to translate OpenDLV messages into motor actuation. The purpose of this microservice is to serve as the glue between OpenDLV and the NMEA2k bus, and its functionality is roughly as follows:

1. Receiving *ActuationRequest* messages either from the remote control microservice described in Section 3.2.3, or some other source, such as an autonomous navigation system.

2. Translating actuation requests into the desired motor rotation and propeller speed using a control loop.

3. Creating PGN 65332 messages and sending them to the motor over the NMEA2k bus.

The main libraries used for the motor control microservice are Libcluon and the NMEA2k C++ library mentioned above. The former provides OpenDLV-related functionality such as message receipt and deserialization, and the latter is used to facilitate communication over the NMEA2k bus. The extent of the NMEA2k-related functionality of the motor control microservice is registering the microservice as a device on the NMEA2k bus to enable communication, and sending control messages

to the motor. Any messages from the bus to the microservice are ignored, but functionality to receive e.g. status messages from the bus is planned to be added in the future.

Similarly to the remote control microservice, the motor control microservice makes use of multiple threads. Multithreading is in this case necessary, because the motor turns relatively slowly; rotating the motor 180 degrees takes several seconds. During the time the motor is turning, the desired motor rotation could change, and the motor being in the process of turning should not prevent changes in the variable that holds the current desired rotation. For similar reasons, the actuation of the motor is undertaken in multiple steps, with a limit for the maximum amount of actuation that can occur at once. Essentially, this means that small changes in motor rotation and propeller speed remain unaffected while big changes have checkpoints in between which allows for periodical re-evaluation of the current desired rotation. Also like with the remote control microservice, a lock guard in the form of a mutex is used to prevent multiple threads from accessing the same variables simultaneously or out of order.

The four main control variables of the motor control microservice are *currentPower, targetPower, currentRotation,* and *targetRotation.* As motor actuation is controlled by a delay between start and stop messages, the unit of the values stored in the control variables is a unit of time (microseconds) for ease of comparison. On startup, these variables are all initialized to a value of zero. It should be noted that this results in any changes in the rotation or the propeller speed of the motor being relative instead of absolute, i.e. there is currently no way for the motor control microservice to know in which direction the motor is pointing relative to the boat. Only the changes made to the rotation and propeller speed of the motor since startup are tracked. The total change in rotation is measured by tracking the total time that has elapsed between the start and stop NMEA2k messages, and is thus only an approximation. This issue and possible solutions to it are discussed in the conclusion of this thesis, in Chapter 4.

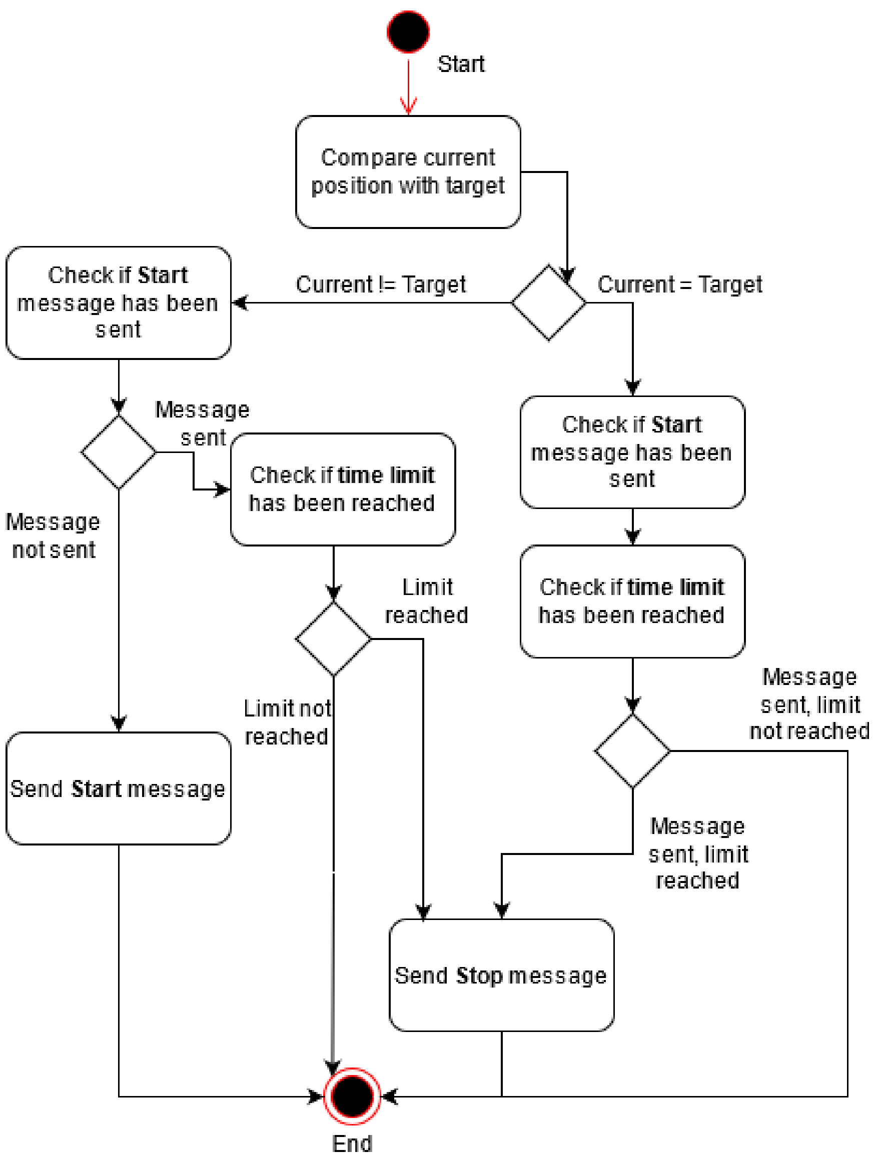


Figure 15: State diagram for the control loop of the motor control microservice

The first thread of the motor control microservice is event-triggered, with the event in question being the receipt of an *ActuationRequest* message. When the event is triggered, a mutex is locked, and the contents of the received *ActuationRequest* are written into *targetPower* and *targetRotation*, indicating that changes are to be made to the propeller speed and/or the rotation of the motor, respectively. The message-receiving thread then yields, and the mutex is unlocked.

The second thread of the microservice houses the actual control loop, a state diagram for which can be seen in Figure 15. One iteration of the control loop consists of two sequential iterations of the loop depicted in the state diagram; the first iteration controls the propeller speed and the second controls the rotation of the motor. In the beginning of the loop, the current value of the control variable is compared against the target value of the control variable. If there is a difference between the two values, a start message is sent over the NMEA2k bus, which is the first of the two messages that are required to actuate the motor.

The time at which the message is sent is also recorded. On subsequent iterations of the loop, the current time is compared against the previously recorded time to see if it is time to send a stop message. There are two states in the control loop that trigger the sending of a stop message:

- Sufficient time has passed since the start message was sent, which means sending the stop message will result in e.g. the motor rotating to the target position.
- A set time limit is reached. As mentioned previously, large actuations are split into stages to facilitate responsive control of the motor.

If the time limit is reached, the motor will actuate towards the target position, but will not reach it yet. On the next iteration of the control loop, the comparison between the target and current positions will trigger the sending of another start message, provided there is still a difference between the current position and target position. Both motor rotation and propeller speed are controlled by the same control loop, although different NMEA2k messages are sent for the two control variables.

Tests of the motor control microservice during development resulted in some interesting unexpected behavior. Frequently, when the motor was rotated, it started making minute back-and-forth movements close to the intended end position. It seemed that the motor was unable to reach the precise desired end position, and tried to correct back and forth repeatedly. This resulted in the control thread in the microservice not yielding, and the microservice was subsequently unable to receive and process further actuation requests. The exact source of this issue is unknown, and was believed to have something to do with the internal control mechanism of the motor. This could also be the cause of the inconsistent amount of rotation observed in earlier tests. To

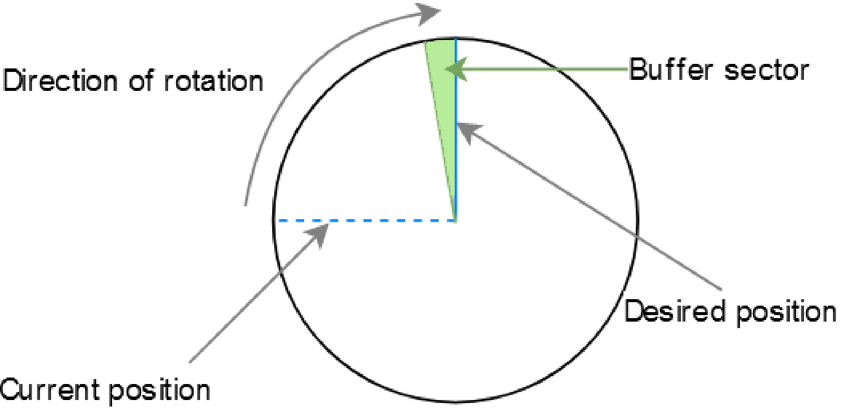


Figure 16: Top-down diagram of the error margin for motor rotation

prevent this behavior, a buffer zone was added to the control loop, as depicted in Figure 16.

The buffer is designed in such a way that if, after a pair of start and stop messages has been sent and the motor has rotated, the motor is within some set distance from the precise desired position, it will stop turning to prevent the back-and-forth correcting behavior. Note that this buffer is very small, and the size of the sector marked with green in Figure 16 is exaggerated for clarity. In reality, the motor will end up within approximately a few degrees of the precise target position. The buffer is implemented by removing one millisecond from the delay between each start and stop message, resulting in the motor rotation stopping just before the target position is reached. The buffer size was determined through trial-and-error, with the intended size being large enough to prevent the problem but still small enough so that the steering could be considered reasonably accurate.

3.3 Networking

The networking hardware of the boat platform consists of a pair of Teltonika RUTX11 routers, which can connect to the internet using 4G LTE and are capable of gigabit Ethernet speeds on the local network. One router (RUTX11-DASBOAT) is located on the boat itself and the other (RUTX11-LAND) is to be used as part of the shore control station, as illustrated in Figure 17.

As the development of the boat platform is still at an early stage, the requirements related to the network are only loosely defined. Communication latency should be kept minimal, preferably at around 50 ms or less. The bandwidth requirements of the Ethernet network on the boat are determined by the high outgoing data rate of the onboard LiDAR, which necessitates the aforementioned gigabit Ethernet. The bandwidth requirement of the communication link between the boat and the shore control station depends largely on whether e.g. video processing and LiDAR data processing are to take place on the boat or on the shore, as the data rate of most of the other communication in the system is relatively low. Generally, the communication between the boat and the shore should be kept at a minimal data rate, as the available bandwidth of the link between the two nodes can fluctuate heavily due to the nature of the 4G network used. Latency and network throughput measurements were taken during an on-water test of the boat platform, which showed that while the latency was sufficiently low, the network throughput varied between 5 Mbits/s and 25 Mbits/s over the course of two hours, with the average available throughput at any given time being roughly 20 Mbits/s.

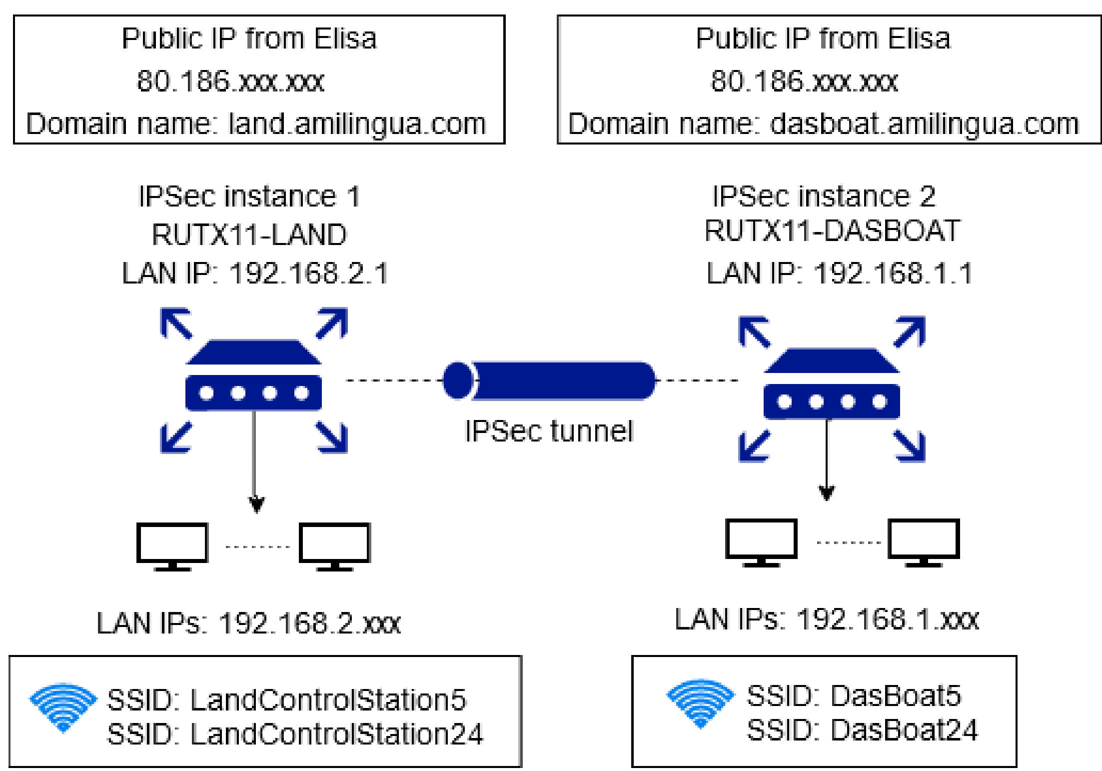


Figure 17: Network and IPSec configuration of the boat and shore LANs

To facilitate secure communication between the two routers, and thus between the boat and the shore control station, the routers are configured with an IPSec tunnel (Internet Protocol Security) between them. IPSec is a security protocol suite for IPV4 that enables the encryption and authentication of data packets over IPV4 networks, and uses cryptographic key negotiation and mutual authentication to secure the communication between hosts or networks [28]. In cryptographic key negotiation, a combination of public keys and private keys is used to encrypt communication between two parties. With the IPSec tunnel, the LANs of each router become subnetworks in a single local network, with an added layer of security for communication taking place between devices on the shore and the boat, respectively. Though the communication between the boat and the shore is sufficiently secure thanks to the IPSec tunnel, the security requirements of the internet connections of the boat and shore networks need some additional work, to prevent outside interference; at the very least, unused ports should be closed.

3.3.1 OpenDLV communication in practice

As mentioned in Section 2.4.1, OpenDLV microservices are grouped into communication groups called *sessions*, where all the participating microservices are recipients to all the messages sent within that session, regardless of whether the different microservices are hosted on a single or multiple devices. The exception is that a microservice will never receive its own messages. An example can be seen in Figure 18, which illustrates the propagation of messages in the motor control system.

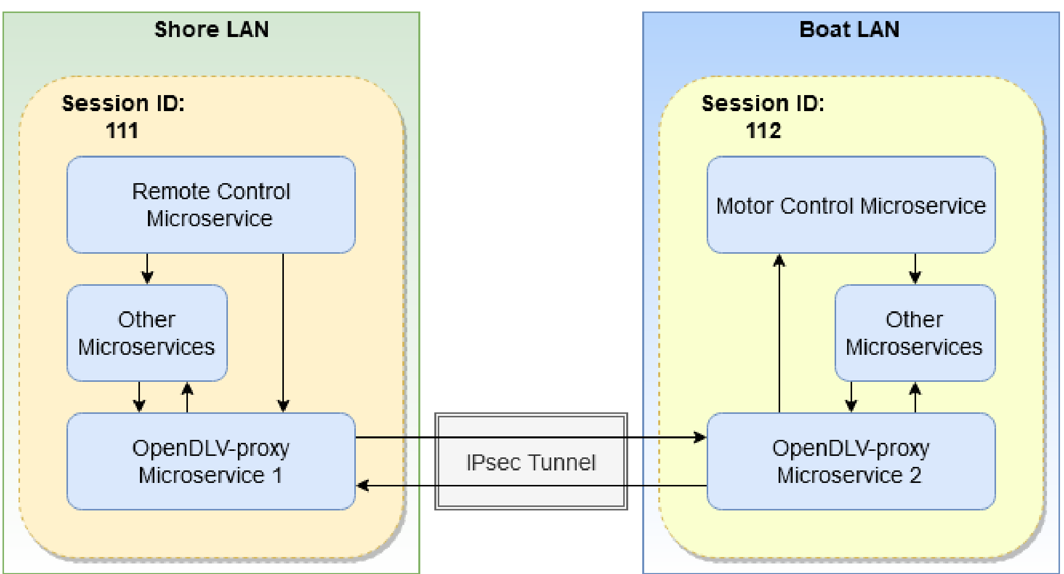


Figure 18: Message propagation in the motor control subsystem

During the development of the two microservices that make up the motor control subsystem, both microservices were located in the same session and on the same local network, and only one router was used. When the microservices were moved to separate LANs, however, the messages from the remote control microservice no longer reached the boat as intended. It was soon discovered that while the LANs of the two routers were bridged by the IPSec tunnel, messages from one subnet were not reaching the other, despite all microservices belonging to the same session. Evidently, the UDP multicast communication groups used by OpenDLV microservices are not only constrained to one session, but also to one subnet.

To remedy this unwanted behavior, a tool called OpenDLV-proxy was used. OpenDLV-proxy is a message flow control tool that can be used to either combine two sessions into one, or to act as a bridge between two sessions to tunnel OpenDLV messages between them even if they are on separate networks, or in this case subnets. [29] As depicted in Figure 18, one instance of OpenDLV-proxy is running in each session. On startup, both instances of the proxy microservice are given as a parameter a port number and the IP-address of the device where the other instance is running. This pair will then relay messages between the two sessions as if they were on the same network and session, and send any incoming messages to all other microservices in the session they belong to.

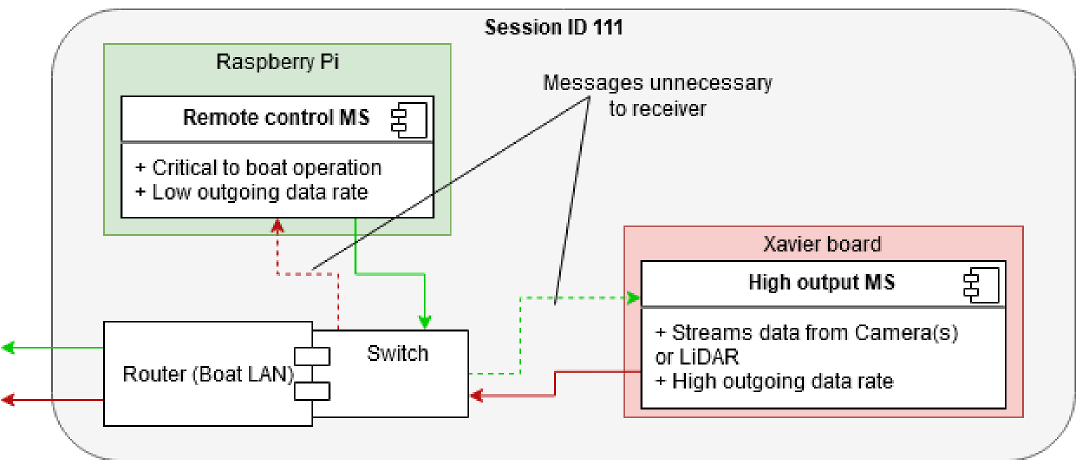


Figure 19: The single session problem

Figure 18 illustrates the message flow in the motor control system when the proxies are used. As the remote control microservice receives input from a controller, the input is serialized into messages which are then sent to all other microservices in the same session on the shore LAN, including one instance of the proxy microservice. The proxy microservice then sends the messages over to the boat LAN through the IPSec tunnel, where another instance of the proxy microservice relays the messages to all microservices in a session on the boat, including the motor control microservice.

While the communication structure in OpenDLV is decidedly simple to implement thanks to the sessions, there is one potential issue; having a large amount of microservices in the same session could potentially lead to a problem wherein high-output microservices end up throttling other microservices in the session with unwanted messages. This could end up in wasting network bandwidth or processing power in the devices hosting receiving microservices when they discard a large amount of unwanted messages.

An example of the above is illustrated in Figure 19. The boat is equipped with two modules that are capable of generating tremendous amounts of data; a 64-beam LiDAR and a set of four full HD 60 fps video cameras. Any microservice in the same session as the microservices that operate the LiDAR or cameras will be on the receiving end of a large amount of unwanted data. In Figure 19, the critical remote control microservice is in the same session as the aforementioned high-output microservices. If the normal operation of the remote control microservice

were disrupted, it could have a serious negative impact on the operation of the whole boat. The outgoing messages of the remote control microservice are small and infrequent, and are thus unlikely to cause any issues related to unwanted receipt by other microservices in the system.

This problem could be solved using another communication control tool called Cluon-relay. Cluon-relay is a microservice realized with Libcluon, and has the following features:

- message filtering through keeping/dropping envelopes with certain message IDs
- downsampling in the form of only forwarding every nth envelope, with functionality to set different downsampling levels for different message IDs, potentially useful for sending low-frequency diagnostics data from the boat to the shore
- sending messages over TCP instead of the standard UDP, with optional MTU (maximum transmission unit) and timeout parameters

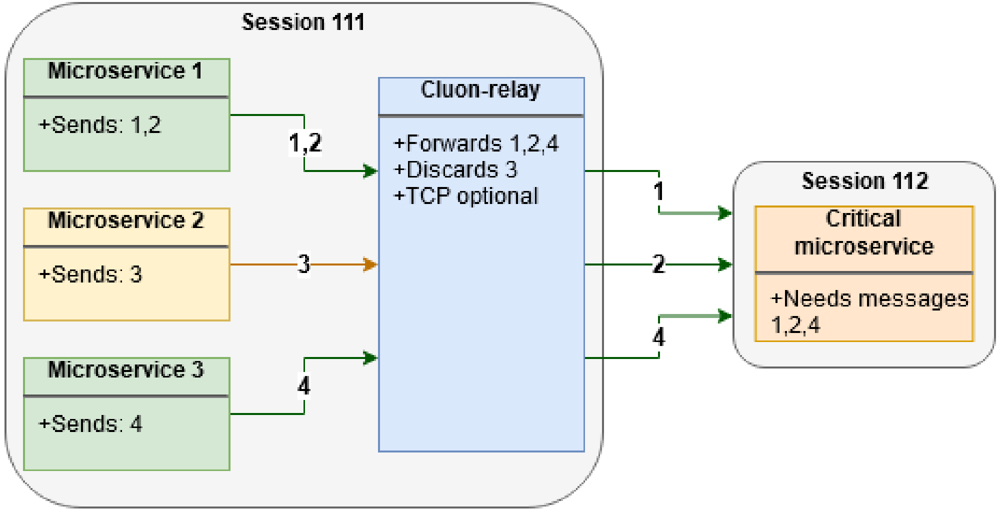


Figure 20: Cluon-Relay as a possible solution the single session problem

Cluon-relay could be used to isolate a microservice, as illustrated in Figure 20. A microservice that is critical to the operation of the boat could be placed in a separate session, and cluon-relay could be used to forward it only the messages it needs to prevent resource expenditure that results from discarding unwanted messages. Additionally, the TCP mode of cluon-relay could be used to make message transmission more robust, at the cost of lower data rates.

Cluon-relay will undoubtedly be useful for controlling what messages are transmitted between the boat and the shore control station. Many microservices will only need to send messages to other microservices on the same subnet, as is the case for the communication taking place between the microservices in the navigation subsystem of the boat, for example. The link between the boat and the shore is the most bandwidth-restricted link in the network, and unnecessary communication between the boat and shore should be kept to a minimum, to lessen the risk of mission-critical messages to the boat being lost.

3.4 Video Streaming

The purpose of the video streaming system is to stream a video feed from a set of camera sensors on the boat over to the shore monitoring station for display.

3.4.1 System Description

The boat platform is equipped with an Nvidia Jetson AGX Xavier GPU development kit and the e-con SurveilsQUAD, a set of four camera sensors. While video data from the cameras is eventually planned to be internally used on the boat for e.g. object detection and situational awareness, the first steps in development have focused on a rudimentary system to grab video frames from the four camera modules and streaming them to the shore control station for display. This would then enable more advanced testing of the implemented systems on the boat platform by remotely controlling the boat from a distance using a video feed instead of line-of-sight.

The initial evaluation of the OpenDLV video streaming pipeline was done using other video cameras than those on the boat, and the results seemed promising. Unfortunately, none of the OpenDLV camera microservices were compatible with the camera modules on the boat, and the video streaming was eventually implemented using the open source multimedia framework Gstreamer. [30]

3.4.2 Video Streaming with OpenDLV

The process of streaming video using OpenDLV microservices is roughly as follows:

1. A camera microservice interfaces with a camera device, writing captured frames to a shared memory area.

2. Another microservice on the same device attaches to the same memory area, encodes them into some format (e.g. h.264) and sends them as ImageReading messages to an OpenDLV session.

3. On the receiving end, a microservice decodes video frames from an ImageReading message into shared memory.

There are a few different choices offered by OpenDLV as far as the video streaming microservices are concerned, as the framework is designed with modularity in mind. For the purposes of the boat platform, *opendlv-camera-opencv* was chosen as the camera-interfacing microservice, and *opendlv-video-h264-encoder* and *opendlv-video-h264-decoder* were used for encoding on the sender end and decoding on the receiver end.

The camera-interfacing microservices use a third-party library called libyuv [31] to facilitate pixel format conversion. Libyuv is an open source library that provides functionality for YUV color space conversion and scaling. *Opendlv-camera-opencv* uses functions from libyuv to convert the image frames grabbed from the camera into the ARGB and i420 formats, which are then stored into two separate memory areas on the device; the former format is useful for local image processing and video display, while the latter is used for encoding and transmission.

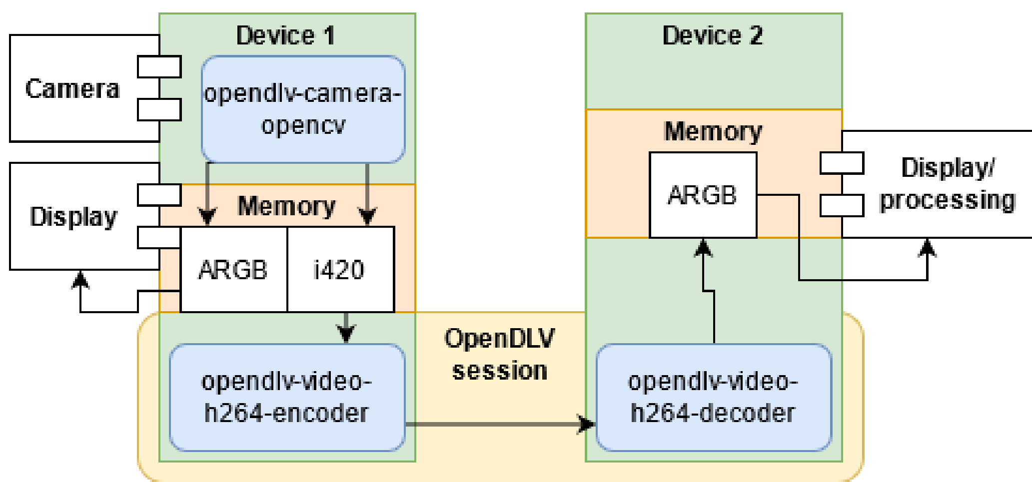


Figure 21: Diagram of the OpenDLV video streaming process

The memory area which now contains image data in the YUV i420 format is then accessed by the encoder microservice, in this case *opendlv-video-h264-encoder*, and the frames are encoded into h264 and subsequently serialized into OpenDLV *ImageReading* messages and broadcast to the communication session to which the encoder microservice belongs. The *opendlv-video-h264-encoder* microservice supports several configuration parameters allowing for granular control of both the h264 encoding and transmission rates. On the receiving end, an instance of the *opendlv-video-h264-decoder* microservice then deserializes any received *ImageReading* messages, decodes the h264 frames and saves them in the ARGB format into shared memory for processing or display. A diagram illustrating the OpenDLV video streaming process can be seen in Figure 21.

When streaming from multiple cameras, the encoder/decoder pair can make use of OpenDLV's sender ID to differentiate between camera streams. When running multiple instances of the camera and encoder microservices, the shared memory area(s) used by each pair need to be given different identifiers as parameters when the microservices are started, otherwise the default identifiers would be used which would cause conflicts.

3.4.3 Containerization

The process of implementing a video streaming system using OpenDLV was thought to be a good opportunity to test out the containerization of microservices. One instance of each of the three microservices needed for the video streaming pipeline would be needed per camera used, the possible exception would be the third microservice, e.g. if the messages containing video frames are simply recorded locally instead of displayed or used for post-processing on the receiving end. The total amount of microservices needed would thus be between 8 and 12 if all cameras are used.

Unlike many of the ready-made microservices provided in the OpenDLV repository, no pre-built Docker images are provided for the h264 encoding and decoding microservices due to patents around the AVC/h264 format used. Instead, the build process involves using a Docker-compose file, which is a YAML-formatted (YAML Ain't Markup Language) file containing automatic build and startup instructions

for a number of Docker images. In the case of *opendlv-video-x264-encoder*, the files needed for using h264 encoding are downloaded during the build process of the container, as defined in the Docker-compose file. Building the Docker images from the source code using the provided Docker-compose files for the *opendlv-video-h264-encoder* microservice did not work, however. The build process involved downloading some other additional packages and libraries from the internet, and their locations had seemingly changed so that their sources in the Dockerfiles were incorrect. Because of this, the encoding microservice used for the video streaming setup was changed to *opendlv-video-x264-encoder*, whose Docker build files did not have the aforementioned build problem. The difference between the two encoding microservices is the h264 encoding library used, otherwise their function is the same.

With a single Docker-compose file, multiple containers can be built and started easily, with individually set startup parameters for each container if needed. Additionally, creating different container configurations is as simple as creating a Docker-compose file for each configuration. For the purposes of the boat platform, a total of two Docker-compose files would be needed for the whole video streaming setup. On the boat, a compose file would be used to start four pairs of camera and encoding microservices. On the shore side, a compose file would be used for the four decoding and display microservices. Only one Docker image is needed to start each type of microservice, i.e. the same image would be used to start each copy of the encoding microservice. The Docker-compose files proved particularly useful in testing the video streaming system, as multiple different streaming configurations could easily be set up and started. This method was used to fine-tune the streaming primarily by visually comparing what effect the multitude of encoding parameters had on the latency and framerate of the video.

3.4.4 Video Streaming with Gstreamer

While running the prepared OpenDLV video streaming setup worked without issue using ordinary webcams, running the same setup on the Xavier board using its cameras resulted in garbled green video on the receiving end, suggesting either a pixel format mismatch or an encoding issue. The problem was narrowed down to the pixel format of the frames captured by the cameras on the Xavier board. A few lines of code were added to the *camera-opencv* microservice to print out the FourCC

code identifying the image format that OpenCV detects the incoming frames to be. A FourCC (Four Character Code) is a sequence of four bytes used to uniquely identify a data format. The output in this case was 'YUYV' also known as YUY2. This format is also supported by libyuv, and an attempt to fix the problem was made by changing what pixel format conversion function was used in the camera microservice. This did not work either, and it seemed that OpenCV detected the format incorrectly.

Using the command *$v4l2-ctl -d[X] --list-formats-ext* (replacing X with the video device number) the actual pixel formats of the frames from the Xavier-connected cameras were revealed to be BG10 and BG12, i.e. 10-bit and 12-bit Bayer GBGB/RGRG; an image format that is not supported by libuyv. The two workarounds to this issue would be to either develop additional functions to libyuv that could perform the correct pixel format conversion, which would be somewhat laborious, or to implement the video streaming using another framework than OpenDLV. After some deliberation, the multimedia framework Gstreamer was chosen as the basis for the video streaming system for the boat platform, at least for the time being; support for the BG10 and/or BG12 formats could be added to libyuv in the future, or the camera hardware on the boat platform could change. In either case, a revisit to the OpenDLV video streaming framework could be made.

Gstreamer is a framework that is designed for creating streaming media applications. The fundamental building blocks of Gstreamer are elements, pads and pipelines. An element is an entity with one specific function, such as encoding, reading data from a file or stream, or outputting data to a screen. Chains of elements are connected together through pads, which are essentially connectors between elements. A number of connected elements form a pipeline, through which data flows in one direction from a source to a sink. [32]

An example of a Gstreamer pipeline can be seen in Figure 22, which depicts the shell script containing the single-line Gstreamer pipeline that runs on the Xavier and interfaces with one of the four camera sensors on board.

```
gst-launch-1.0 -vv nvarguscamerasrc sensor-id=$1 do-timestamp=true !
'video/x-raw(memory:NVMM), width=(int)1280, height=(int)720,
format=(string)NV12, framerate=(fraction)30/1' ! nvvidconv !
queue ! x264enc tune=zerolatency ! rtph264pay !
udpsink host=192.168.2.197 port=$2
```

Figure 22: Shell script for launching a Gstreamer pipeline

The Gstreamer pipelines used for the sender and receiver work in much the same way as the OpenDLV video streaming setup. The element *nvvidconv* performs format conversion from the native format of the camera sensor to the format needed for encoding, and the *x264enc* element performs the encoding, which is configured for minimal latency using the *tune=zerolatency* option of the encoding element. The *rtph264pay* element then packages the encoded data into RTP (Real-time Transport Protocol) packets, and the *udpsink* element sends the packets to a socket over UDP. On the receiver side, a Gstreamer pipeline does essentially the same thing in reverse; the incoming packets are received, depayloaded, and decoded in that order, after which they are displayed on screen. One instance of the camera-interfacing pipeline is used for each camera sensor and one instance of the receiving pipeline is used for each stream, for a total of four senders and four receivers.

3.5 GPS and LTE Telemetry

The Teltonika RUTx11 router on the boat runs an operating system called RutOS, which is based on OpenWRT, a Linux OS aimed for embedded systems. The operating system of the router can be accessed using SSH (Secure Shell), and its firmware includes a pair of utilities that can provide useful data to the systems of the boat platform and shore control station; *gsmctl*, which provides GSM- and LTE-related telemetry data, and *gpsctl*, which is used to query the GPS module of the router for positioning data, provided an external GPS antenna is connected to the router. For the purpose of aggregating the GPS and LTE telemetry data and converting it into a format which would coincide with the OpenDLV communication structure used by most of the other systems on the boat platform, a GPS and LTE telemetry

microservice called *opendlv-device-router* was created.

Table 2: Data exctracted using the *gsmctl* and *gpsctl* utilities on the boat router

Data	**Additional information**
Time	Epoch timestamp
GSM RSSI	Received Signal Strength Indicator, a measurement of radio signal power
RSRQ	Reference Signal Received Quality, a measure of LTE signal quality level
RSRP	Reference Signals Received Power, same as above
SINR	Signal-to-Noise Ratio
Operator Name	
Data Carrier Type	e.g. LTE, GSM, or no carrier
Module Temp	Temperature of the modem
Network Info	Information related to the mobile network, such as frequency band and channel ID
Datetime	GPS date and time
Lat, Long	Latitude and Longitude
Speed	Speed over ground
Angle	North-relative heading
Altitude	Altitude over sea-level

As the hardware of the router would likely not be able to run the microservice due to resource constraints, the GPS and LTE telemetry system was designed such that the actual microservice would be running on another device on the local network, and a shell script on the router would send the data over to the microservice for further processing. A shell script running on the router periodically executes a number of *gsmctl* and *gpsctl* commands and sends the data to a socket over UDP using the netcat utility. The *opendlv-device-router* microservice attaches to the socket, parses the incoming data, serializes them into the appropriate OpenDLV messages, and finally sends the messages to the OpenDLV session. A list of the data acquired using the *gsmctl* and *gpsctl* utilities on the boat router can be seen in Table 2. The script on the router can be found in Appendix B.

Where possible, the data acquired from the script on the router are serialized into OpenDLV messages already supported by the OpenDLV standard message set; as is the case for the values for altitude, speed, heading, and position, which are serialized

into *AltitudeReading*, *GroundSpeedReading*, *GeodeticHeadingReading*, and *GeodeticWGS84Reading* messages, respectively. The utilization of messages already part of the OpenDLV standard message set for the GPS-related data was done specifically because the opendlv-vehicle-view web interface, which will be discussed in Section 3.6, has some built-in map functionality which requires the latitude and longitude contained in the *GeodeticWGS84Reading* message.

```
message opendlv.proxy.RouterDataMessage [id = 3000]
uint32 epochTime [id = 1];
int16 GSMRSSI [id = 2];
int16 LTERSRP [id = 3];
float LTESINR [id = 4];
int16 LTERSRQ [id = 5];
uint32 cellIDParam [id = 6];
string operatorName [id = 7];
string dataCarrierType [id = 8];
float moduleTemp [id = 9];
string networkInfo [id = 10];
```

Figure 23: Message specification for the *RouterDataMessage*

Additionally, a new message was added to the OpenDLV standard message set. The *RouterDataMessage* contains all the data obtained from the gsmctl commands in the script on the router, as seen in Figure 23. All fields in the message are single values, with the exception of the *networkInfo* field, which is a string containing a few comma separated values which can be parsed as needed on the receiving end.

The script on the router runs the gsmctl and gpsctl utilities once per second and concatenates all the data into a single string with a delimiter between each field, after which it is sent to a socket using netcat. On the receiving end, the *opendlv-device-router* microservice reacts to incoming data on the socket, and the data is converted to the correct type, as defined by the entries for each message in the standard message set. The appropriate OpenDLV messages are then constructed,

after which they are sent using the OpenDLV-standard method of UDP multicast to the session in which the microservice belongs.

3.6 Visualization

One of the milestones in the development of the boat project was defined as remote control using visualization, i.e. remotely operating the boat without having a direct line of sight to it. With the remote control system implemented and tested and with a rudimentary live video streaming system in place, the next step was to implement an informative interface for the shore control station. The purpose of the interface is to not only provide the remote control operator of the boat the information needed to steer the boat, but to also provide a good overview of the system, including but not limited to a real-time map, a list of messages being transmitted in the system, a network status display, etc. The system overview would be especially important during the later stages in the development of the boat project, when the development eventually starts to shift from manual control to autonomous operation.

Early on during project planning, it was agreed upon that at some point a system overview/visualization interface would be built from the bottom up so that it could be better tailored to the needs of the boat project. As an intermediary step, however, the *opendlv-vehicle-view* [33] microservice included in the OpenDLV distribution would, with some modification, serve as a good temporary substitute for a bespoke interface. The features of *opendlv-vehicle-view* include:

- Display of a real-time list of OpenDLV-messages received from the session.
- Display of video frames and LiDAR point clouds sent using OpenDLV *ImageReading* and *PointCloudReading* messages, respectively.
- A virtual joystick for remote control operation.
- A map drawn around the position obtained from a message.
- Plotting functionality for OpenDLV messages.
- Functionality to record and replay all messages sent in a session over a period of time.

The microservice creates a web page displaying the information listed above, which can be viewed locally using a web browser.

3.6.1 Message Overview

action(s)	ID	senderStamp	message name	sample timestamp [µs]	signal(s)
	3000	0	opendlv_proxy_RouterDataMessage	1615294097012286	epochTime: 1615286896 GSMRSSI: -51 LTERSRP: -60 LTESINR: 22.1 LTERSRQ: -6 cellIDParam: 1345025 operatorName: elisa dataCarrierType: LTE moduleTemp: 45 networkInfo: (base64) K1FOV0lORk...
	1033	0	opendlv_proxy_AltitudeReading	1615294097012383	altitude: 46.8
	19	0	opendlv_proxy_GeodeticWgs84Reading	1615294097012440	latitude: 60.455188 longitude: 22.284663
	1051	0	opendlv_proxy_GeodeticHeadingReading	1615294097012489	northHeading: 123.9
	1046	0	opendlv_proxy_GroundSpeedReading	1615294097012537	groundSpeed: 0

Figure 24: Message overview in vehicle-view

The message overview tab depicted in Figure 24 displays a regularly updated list of all the OpenDLV messages that have been received from a running session since startup, along with the message contents. The messages seen in Figure 24 come from the GPS and LTE telemetry microservice described in Section 3.5, which was the only other microservice running at the time. The list in the message overview, along with the map and plot tabs, is updated after a certain (hard-coded) number of messages have been received. The reason for this is presumably that the web page would update too rapidly if messages were arriving at a high frequency. This led to a slight issue, however, when the frequency of received messages was low. With only the GPS and LTE telemetry microservice running, the view only updated once every several seconds, and the update threshold had to be reduced in the code of *opendlv-vehicle-view* so that the view would update more often.

3.6.2 Plotting

The plotting functionality of *opendlv-vehicle-view* is realized using a built-in implementation of Gnuplot [34], a freely distributed graphing utility. The plotting interface consists of two parts, the plotting code pane as seen in Figure 25, and

the plot output, depicted in Figure 26. A plot of any received OpenDLV message can be drawn using a pre-made plotting template by pressing a button on the message overview tab, and the plotting code can be edited during run-time using the code pane, after which the plot will automatically update. Predefined plots can be created by storing the Gnuplot code in a JavaScript variable in the main.js file of *opendlv-vehicle-view* and adding an element to the drop-down menu using html.

Select pre-defined plot: LTE Signal-to-Noise Ratio

```
        # Plotting Signal-to-Noise Ratio, last 50 values (actually last 50 "refreshes"),
using palette to color line
        # depending on signal to noise ratio
        set terminal svg size 700,500 enhanced fname 'arial' fsize 10 solid
        set output 'out.svg'    # Output must always be named 'out.svg' to display.

        set title 'opendlv_proxy_RouterDataMessage.0'
        set xlabel 'DATAPOINTS'    # Define label for x axis.
        set xrange [GPVAL_DATA_X_MAX-50:] # uncomment for display of last 50 values
        set ylabel 'dB'    # Define label for y axis.
        set key inside top left # Add legend box located at top/left.
        set palette defined (  0 'red', 6 'orange', 15 'yellow', 22 'green')


        # Plot first numerical signal of message 'opendlv_proxy_RouterDataMessage.0'.
        # Replace 'using 1' with 'using 1:2' or similar to access other fields.

        # Actual call to plot data.
        plot[] "opendlv_proxy_RouterDataMessage.0" using 0:4:4 title 'LTE-SINR' with lines
lc palette lw 2
```

Figure 25: Plotting code user interface in vehicle-view

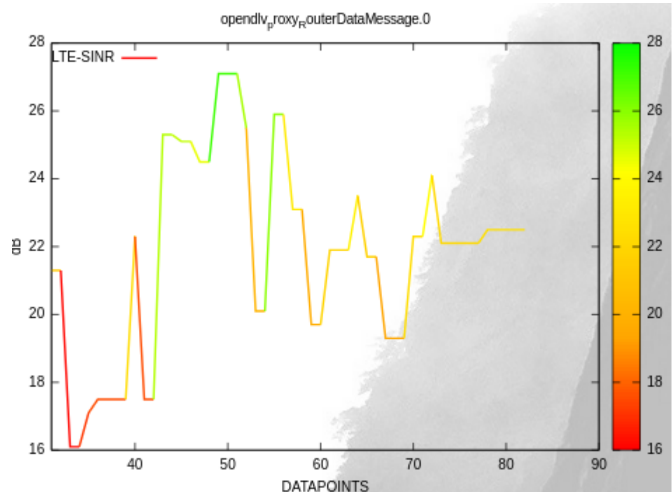


Figure 26: Signal-to-noise ratio plot in vehicle-view

A number of predefined plots were created for the boat project, most notably to draw plots of data related to the network of the boat platform. The parameters for these

plots are obtained from *RouterDataMessages.* An example can be seen in Figures 25 and 26, which depict the plotting code and resulting plot for the signal-to-noise ratio of the router on the boat, respectively. The line of the SINR and other signal-quality related plots change color based on thresholds defined by the router manufacturer [35], as the relation between the actual value and the quantity being measured are dependent on the hardware used.

3.6.3 Map

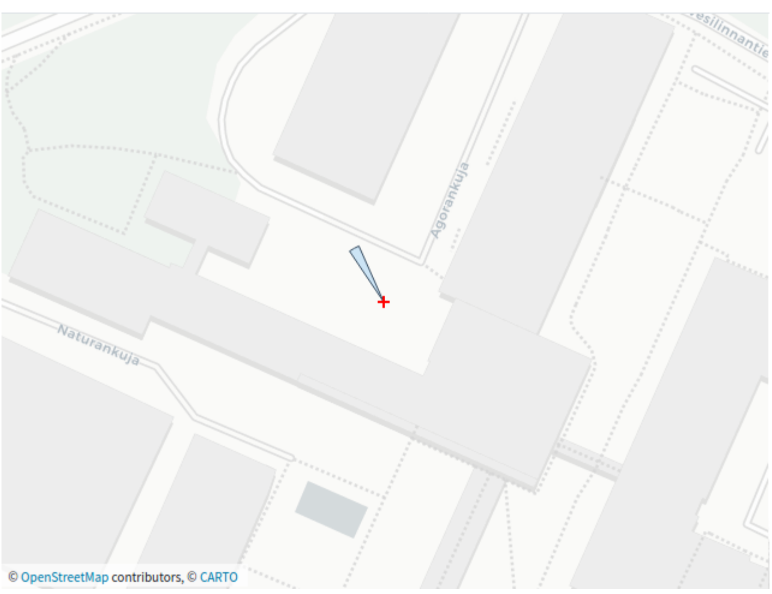


Figure 27: Map pane in vehicle-view

The map pane of *opendlv-vehicle-view* displays a map, which is centered on the coordinates obtained from the *GeodeticWgs84Reading* message. A zoom-able map is drawn by using the open-source javascript library Maptalks [36] to query the OpenStreetMap [37] database for map data. A base layer is drawn containing the map itself, and a small cross is drawn in the center, indicating the current position. As a display of the boat heading was also needed for the boat platform, this functionality was added to vehicle-view, also using Maptalks. To display the heading, a vector layer is drawn on top of the map layer, containing a small sector originating from the current position and pointing in the direction of the heading value obtained from a *GeodeticHeadingReading* message. While the variables containing the position and heading values are updated whenever the appropriate messages are received, the map

is only redrawn whenever the view updates, as discussed in Section 3.6.1. The map can be seen in Figure 27.

4 Results and Evaluation

In this Chapter, the microservices developed for this thesis and the OpenDLV software ecosystem as a whole is summarized and evaluated. Some future improvements to the systems of the boat platform are also discussed.

4.1 Motor Control System

The current motor control system implementation fulfills its requirements in that the boat can be remotely controlled using visual contact. However, a substantial issue with the motor control subsystem as it is implemented is that there is currently no way to get the exact status of the motor, i.e. rotation and propeller speed. Having visual contact with the boat and its motor lessens the need for motor status information, but for autonomous navigation or remote control without visual contact to be at all possible, this information is essential. Reasonably accurate propeller speed information can be inferred from the status of the motor control microservice, and if needed, functionality to read the NMEA2k status messages the motor independently sends out could be added to the microservice. As the status message concerning propeller speed is simply a measure in percent of how much power the propeller motor is receiving, it can be considered reliable.

The same cannot be said about the motor rotation status message. During network traffic observation of the NMEA2k bus in the lab, the motor rotation readings in the status messages from the motor were found to be fluctuating even when the motor was static. This is likely due to the source of the rotation status information being a magnetometer in the GPS module of the motor, and the readings having been influenced by external magnetic field disturbances from surrounding electronic equipment. This issue could be solved by either evaluating the reliability of the magnetometer readings or by installing another, for instance mechanical rotation sensor on the motor. If the magnetometer readings were through evaluation found to be acceptable, i.e. within some small range, the status messages from the motor could be used and any fluctuations could be accounted for within the microservice. Adding a dedicated rotation sensor such as a rotary encoder or hall effect sensor implementation would likely be more accurate, but would also be more labor intensive.

The control loop in the motor control system contains a parameter for the current motor position, which is currently an approximation of the amount the motor has turned since service startup. If this parameter would be changed to instead hold accurate readings from a rotation sensor, the accuracy of the motor control system would improve dramatically. This addition is likely to be one of the next steps in the further development of the boat platform.

4.2 Video Streaming and Visualization

Though the video streaming implementation using OpenDLV was eventually scrapped due to a hardware conflict, it was found to work quite well in preliminary testing. The stream latency between camera and screen was acceptably low, considering the wireless connection between the boat and the shore. Splitting the video streaming pipeline into three microservices (camera sensor interface, encoding, and receiving/display) facilitated granular control over the whole pipeline, which was particularly useful for the encoding microservice and its many parameters.

The OpenDLV web view microservice provides a good overview of the system, with its map, message overview, and plotting functionality. As the web view by default only supports video display if the frames arrive in the form of OpenDLV messages, support for other methods of video streaming will have to be added later, if it is needed. The web view also has support for displaying LiDAR point clouds, which will be useful in the future as a LiDAR system is one of the planned additions to the boat platform, as per the system design diagram.

4.3 OpenDLV

An important aspect to consider when evaluating a system architecture is how it handles complexity. The current system of the boat platform consists of a relatively low number of microservices, and has yet to present any major issues stemming from complexity. The high-level system design created as part of this thesis predicts a much higher level of complexity in the future, and the question of how OpenDLV and its tools could be used to handle this is hard to answer. While the standardized communication structure of OpenDLV makes setting up the communication links between elements in a system easy, it is hard to discern the intent of the developers of OpenDLV when it comes to some issues. These issues include the potentially high

computational cost of a service discarding unwanted messages, and messages in a large communication group potentially failing to arrive due to low available network bandwidth.

Correspondence with the developers revealed that typical systems on their platforms are supported by powerful hardware, and are thus unlikely to run into these issues. Therefore, while developing an implementation of OpenDLV on a more resource-constrained platform such as the boat platform, complexity is definitely an issue that needs to be considered. It should be noted that problems arising from complexity are by no means unique to OpenDLV but are inevitable in distributed systems and especially microservice-based systems, as discussed earlier in this thesis.

Generally, development using OpenDLV suffers somewhat due to lack of documentation; getting started with the system could be made significantly easier if the information about the system and its features were consolidated in one place and documented more extensively. At the time of writing, the documentation for OpenDLV, its microservices, and its main library Libcluon, is scattered over a number of GitHub pages and difficult to piece together. Many of the ready-made OpenDLV microservices were not in the main repository, including some very useful communication flow control microservices. The developers seem to be aware of this, however, as the landing page for OpenDLV includes a message indicating that a web page is in development.

Disregarding the documentation issues, developing completely new functionality to a system using the OpenDLV architecture is relatively straight-forward. The defining features of the communication structure, i.e. the data structure definition in the standard message set and the reliance on UDP multicast for transport, both serve to simplify incremental additions of new functionality to an established system. Libcluon, the small yet flexible main library, provides a host of useful basic features to microservices, and the use of Docker containers provides flexibility to deployment by separating the applications from the underlying hardware.

5 Conclusion

The purpose of this thesis was to implement and evaluate a rudimentary communication and control system for an autonomous boat platform using the OpenDLV software ecosystem and its microservice-based architecture. In order to lay theoretical foundation for the development of the system, some of the key aspects of OpenDLV, e.g. the microservice architecture, distributed systems, and containerization, were explored in more detail. A high-level diagram of the planned system was created for the purpose of mapping planned system functionality into small elements that would later be developed into microservices. The system design diagram was developed with a distributed system in mind, and was also used to describe the communication flow of the system.

A number of OpenDLV microservices were then developed to handle the tasks of manual remote control, video streaming and display, collection and display of GPS-derived data, and LTE telemetry data collection and display. Some of the microservices were built from the ground up and some were built by using an existing OpenDLV microservice as a starting point.

The focus on small, contained applications in lieu of a more monolithic approach makes OpenDLV a particularly good fit for the development environment of the boat platform. Because the bulk of the development is intended to be done by many different researchers as well as project and thesis workers, the ability to easily develop small additions to the system is invaluable. While OpenDLV and the systems developed using it for this thesis have their drawbacks, these can and should be mitigated by careful system planning before and during future development. In conclusion, the benefits that are to be gained from the flexibility of a distributed system implementation for an autonomous vehicle platform should not be underestimated.

6 Svensk Sammanfattning - Swedish Summary

6.1 Inledning

Under det senaste årtiondet har intresset för teknologier som möjliggör autonoma fordon ökat avsevärt både inom industrin och i den akademiska världen. Även om uppmärksamheten ofta fästs speciellt på självstyrda bilar, kan man också se ett ökat intresse för teknologi ämnat för autonoma sjöfartsfordon. På fakulteten för naturvetenskaper och teknik vid Åbo Akademi har en forskningsplattform för en självstyrd båt grundats för att experimentera med och utveckla olika teknologier som kunde göra framsteg inom området autonoma sjöfartsfordon.

Syftet med denna avhandling är att implementera ett grundläggande mjukvarusystem till båtplattformen. Till systemets uppgifter hör bl.a. datainsamling från olika sensorer, styrning samt dataöverföring mellan olika hårdvaruenheter och delsystem. Eftersom ett heltäckande styr- och kommunikationssystem för ett autonomt fordon är rätt så invecklat och eftersom ett flertal av de till plattformen planerade systemen är tillsvidare bara konceptuella, kommer avhandlingen mestadels att handla om ett fåtal delsystem. Målsättningen är att utvecklingen och implementeringen av dessa delsystem kan användas som en bra grund för att sporra framtida vidareutveckling av båtplattformen.

6.2 Systembeskrivning

Den grundläggande teknologin som tillämpas i denna avhandling är OpenDLV [1]. OpenDLV (eng. *Open DriverLess Vehicle*) är en mjukvarumiljö med öppen källkod skapat för att främja utvecklingen och testandet av system för autonoma fordon. Även om miljön är ursprungligen avsedd särskilt för självstyrda bilar, finns det de facto inga hinder för att tillämpa den till ett system för ett autonomt sjöfartsfordon. Den största skillnaden mellan bilar och sjöfartsfordon är i detta sammanhang systemens kommunikationsteknologier; emedan enheter inom datasystem i bilar typiskt kommunicerar via en CAN-buss *(Controller Area Network)*, använder sjöfartsfordonet som denna avhandling berör ett lokalt Ethernet-nätverk. I praktiken har valet av kommunikationsteknologi dock en liten betydelse.

OpenDLV har ett antal egenskaper som är utmärkande för systemets uppbyggnad: OpenDLV Standard Message Set, Microservices och Libcluon.

OpenDLV Standard Message Set är en fil som beskriver hur data struktureras och serialiseras för kommunikation mellan olika enheter i ett system. Filens huvudsakliga uppgift är att upprätthålla en standard för kommunikation som gör det lätt att utveckla ny funktionalitet till systemet.

Miljön är fullständigt baserat på microservices, en modulär mjukvaruarkitektur där de diskreta enheterna i ett system eller en applikation är minimala dvs. varje enhet har en enda, väl begränsad funktion. Enheterna är också sinsemellan frånkopplade och kommunicerar självständigt med varandra över ett nätverk utan någon form av central styrning.

En stor del av funktionaliteten som erbjuds av OpenDLV ligger i programbibliokteket Libcluon. Biblioteket innehåller en stor mängd generella funktioner som förenklar utvecklingen av programvara till distribuerade system och speciellt system för autonoma fordon. Själva källkoden i biblioteket delas ut i form av en enda fil, vilket gör det enkelt att använda i t.o.m. små program.

6.3 Metoder

I avhandlingen utarbetas en grundlig helhetsbild om vad som bör uppmärksammas vid utformningen, utvecklingen och implementeringen av ett distribuerat system för ett sjöfartsfordon. Innan någon praktisk tillämpning förverkligas, studeras de grundläggande egenskaperna av OpenDLV i detalj för att få en uppfattning om varför OpenDLV är uppbyggt som det är, samt för att försöka identifiera vilka element är kritiska till systemets funktionalitet.

Speciell uppmärksamhet fästs på microservice-arkitekturen och virtualisering, eftersom dessa teknologier anses vara speciellt viktiga med tanke på en välfungerande praktisk implementation av ett distribuerat system. Microservice-arkitekturen är dessutom en något ung teknologi, och en implementation därav kan ha brister som kunde leda till negativa konsekvenser i ett system ämnat för ett autonomt fordon.

6.4 Implementation

Den praktiska implementationen som valdes till utgångspunkt är ett delsystem för motorkontroll. Delsystemet för motorkontroll ansågs vara en bra utgångspunkt eftersom det är av en lämplig storlek och för att det omfattar flera funktioner som kommer att vara närvarande i många andra delsystem på plattformen. Förutom delsystemet för motorkontroll utvecklades också ett antal mindre delsystem för bl.a. direktuppspelning av video samt insamling och uppvisning av GPS-data. För att skapa en grundläggande helhetsbild av systemet, utvecklades därtill ett diagram på hög nivå över båtplattformens system.

Delsystemet för motorkontroll består av en microservice som läser inmatningar från en spelkontroll (eng. *gamepad*) och översätter dem till instruktioner för styrningen av en utombordsmotor. Styrinstruktionerna mottas av en annan microservice som översätter instruktionerna till meddelanden som kan skickas över en NMEA2000 buss för att driva själva motorn. Kommunikationen i delsystemet sker m.h.a. UDP över ett lokalt nätverk, vilket består av två trådlösa nätverksväxlar. Även om delsystemet tillsvidare enbart stöder manuell styrning, är det är uppbyggt med tanke på eventuell automatisk styrning; microservicen som driver utombordsmotorn är konstruerat för att göra en framtida implementation av självstyrning möjligast smärtfri.

6.5 Resultat

Implementationen av motorkontroll fungerade som väntat. Den teoretiska utredningen av viktiga systemegenskaper ledde till gynnsam information som påverkade utvecklingen av båtplattformens olika delsystem. Det tillgodogjordes t.ex. att både microservicen för inmatning och den för motorkontroll borde kunna agera som buffert för meddelanden eftersom frekvensen med vilken meddelanden skickas kunde sänkas för att lätta på nätverkets belastning. Under utvecklingen av motorkontroll och de andra delsystemen ökade förståelsen för OpenDLV och distribuerade system i allmänhet, vilket förde med sig erfarenheter som kommer att vara nyttiga då båtplattformen utvecklas vidare i framtiden.

References

[1] "Opendlv". (), [Online]. Available: `https://github.com/chalmers-revere/opendlv` (visited on 02/04/2020).

[2] J. Zhao, B. Liang, and Q. Chen, "The key technology toward the self-driving car", 2018.

[3] S. Kuutti, S. Fallah, K. Katsaros, M. Dianati, F. Mccullough, and A. Mouzakitis, "A survey of the state-of-the-art localization techniques and their potentials for autonomous vehicle applications", *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 829–846, 2018. DOI: `10.1109/JIOT.2018.2812300`.

[4] M. BARBIER, E. BENSANA, and X. PUCEL, "A generic and modular architecture for maritime autonomous vehicles", in *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, 2018, pp. 1–6. DOI: `10.1109/AUV.2018.8729765`.

[5] M. Fowler and J. Lewis. "Microservices: A definition of this new architectural term". (Mar. 2014), [Online]. Available: `https://martinfowler.com/articles/microservices.html` (visited on 01/16/2020).

[6] S. Newman, *Building Microservices - Designing Fine-Grained Systems*, 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, Feb. 2015, ISBN: 978-1-491-95035-7.

[7] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, *et al.*, "Microservices: Yesterday, today, and tomorrow", Jun. 2016.

[8] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells", *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018, ISSN: 1937-4194. DOI: `10.1109/MS.2018.2141031`.

[9] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study", Mar. 2018. DOI: `10.5220/0006798302210232`.

[10] J. Lotz, A. Vogelsang, O. Benderius, and C. Berger, "Microservice architectures for advanced driver assistance systems: A case-study", in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 45–52. DOI: `10.1109/ICSA-C.2019.00016`.

[11] P. Masek and T. Magnus, "Container based virtualisation for software deployment in self-driving vehicles", 2016.

[12] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing", in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2015, pp. 1–8. DOI: 10.1109/AIEEE.2015.7367280.

[13] "Linux man-pages, cgroups". (), [Online]. Available: http://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 02/24/2020).

[14] "Linux man-pages, namespaces". (), [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 02/24/2020).

[15] "Docker documentation, overview". (), [Online]. Available: https://docs.docker.com/get-started/overview/ (visited on 04/07/2020).

[16] "Docker documentation, docker hub". (), [Online]. Available: https://docs.docker.com/docker-hub/ (visited on 04/07/2020).

[17] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment", in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 178–185. DOI: 10.1109/CLOUD.2018.00030.

[18] U. Gupta, "Comparison between security majors in virtual machine and linux containers", *CoRR*, vol. abs/1507.07816, 2015. arXiv: 1507.07816. [Online]. Available: http://arxiv.org/abs/1507.07816.

[19] I. Mavridis and H. Karatza, "Performance and overhead study of containers running on top of virtual machines", in *2017 IEEE 19th Conference on Business Informatics (CBI)*, vol. 02, 2017, pp. 32–38. DOI: 10.1109/CBI.2017.69.

[20] "Opendlv standard message set". (), [Online]. Available: https://github.com/chalmers-revere/opendlv.standard-message-set/tree/master (visited on 02/12/2020).

[21] "Google protobuf documentation". (), [Online]. Available: https://developers.google.com/protocol-buffers (visited on 02/12/2020).

[22] "Libcluon". (), [Online]. Available: https://github.com/chrberger/libcluon (visited on 02/04/2020).

[23] "Controller clip art". (), [Online]. Available: http://www.clker.com/clipart-game-console-controller-outline.html (visited on 05/18/2021).

[24] "Cplusplus reference, mutex". (), [Online]. Available: https://en.cppreference.com/w/cpp/thread/mutex (visited on 04/23/2020).

[25] "Nmea2000 standard page". (), [Online]. Available: https://www.nmea.org/content/STANDARDS/NMEA_2000 (visited on 04/10/2021).

[26] L. A. Luft, L. Anderson, and F. Cassidy, "Nmea 2000 a digital interface for the 21st century", Jan. 2002. [Online]. Available: https://www.nmea.org/Assets/nmea-2000-digital-interface-white-paper.pdf (visited on 04/10/2021).

[27] "Github page, nmea2000 library for c++". (), [Online]. Available: https://github.com/ttlappalainen/NMEA2000 (visited on 02/16/2021).

[28] "Rutx11 ipsec wiki page". (), [Online]. Available: https://wiki.teltonika-networks.com/view/IPsec_configuration_examples (visited on 02/02/2021).

[29] "Github page for opendlv-proxy". (), [Online]. Available: https://github.com/chalmers-revere/opendlv-proxy (visited on 02/02/2021).

[30] "Webpage for gstreamer". (), [Online]. Available: https://gstreamer.freedesktop.org/ (visited on 02/09/2021).

[31] "Webpage for libyuv". (), [Online]. Available: https://chromium.googlesource.com/libyuv/libyuv/ (visited on 03/02/2021).

[32] "Gstreamer foundations". (), [Online]. Available: https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html?gi-language=c (visited on 03/04/2021).

[33] "Github page for opendlv-vehicle-view". (), [Online]. Available: https://github.com/chalmers-revere/opendlv-vehicle-view (visited on 03/15/2021).

[34] "Landing page for gnuplot". (), [Online]. Available: http://www.gnuplot.info/ (visited on 03/16/2021).

[35] "Router manufacturer wiki page for sinr". (), [Online]. Available: https://wiki.teltonika-networks.com/view/SINR (visited on 03/16/2021).

[36] "Maptalks api". (), [Online]. Available: https://github.com/maptalks/maptalks.js/wiki (visited on 03/22/2021).

[37] "Openstreetmap". (2021), [Online]. Available: https://www.openstreetmap.org/about (visited on 03/22/2021).

A System Diagram

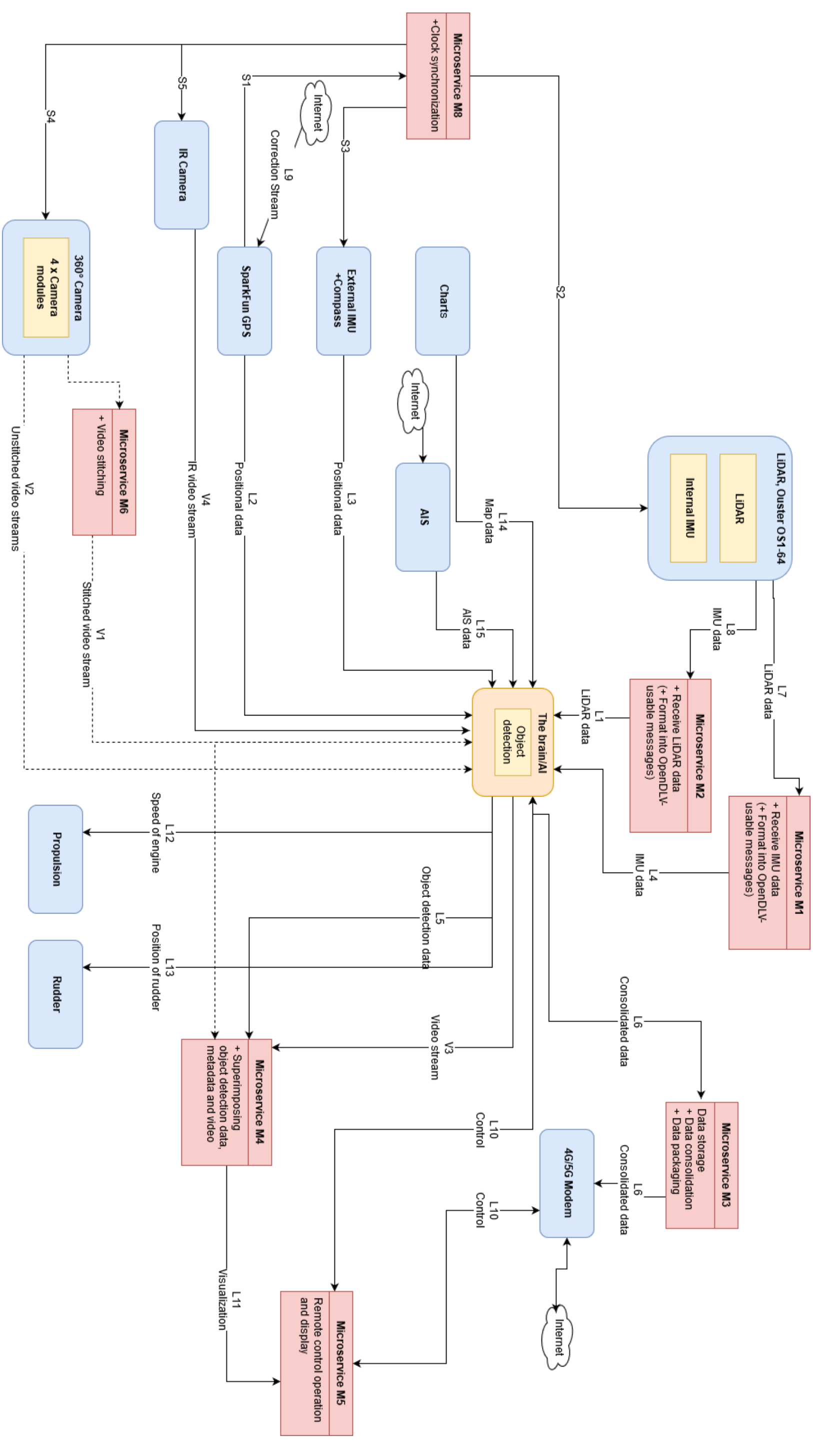


Figure 28: Latest high-level system diagram

B Router Script

root@RUTX11-SHORE: ~

```
# Author : Sebastien Lafond
# This script logs a set of parameters from the router and log this via a curl comman
d to a VM hosted by CSC
# RIT11X uses BusyBox (ash)

while 1>0                                    # log indefinitely
do

# MODEM DATA
TOPOST="$(date "+%s%N")&"                    #get the epoch timestamp
TOPOST=$TOPOST"$(gsmctl -q)&"                #Get GSM signal (RSSI) level
TOPOST=$TOPOST"$(gsmctl -W)&"                #Get LTE RSRP level
TOPOST=$TOPOST"$(gsmctl -Z)&"                #Get LTE SINR level
TOPOST=$TOPOST"$(gsmctl -M)&"                #Get LTE RSRQ level
TOPOST=$TOPOST"$(gsmctl -C)&"                #Get cell ID parameter
TOPOST=$TOPOST"$(gsmctl -o)&"                #Get name of operator used
TOPOST=$TOPOST"$(gsmctl -t)&"                #Get data carrier type
TOPOST=$TOPOST"$(gsmctl -c)&"                #Get module temperature
TOPOST=$TOPOST"$(gsmctl -F)&"                #Get network information

#TOPOST="${TOPOST//[[:blank:]]/_}"            # remove blank spaces
TOPOST="${TOPOST%??}"                        # remove last two charaters (does not work wi
thout)


# GPS DATA
TOPOST=$TOPOST"&$(gpsctl -i)&"               #get latitude
TOPOST=$TOPOST"$(gpsctl -x)&"                #get longitude
TOPOST=$TOPOST"$(gpsctl -t)&"                #get timestamp
TOPOST=$TOPOST"$(gpsctl -a)&"                #get altitude
TOPOST=$TOPOST"$(gpsctl -v)&"                #get speed
TOPOST=$TOPOST"$(gpsctl -p)&"                #get satellite count
TOPOST=$TOPOST"$(gpsctl -g)&"                #get angle
TOPOST=$TOPOST"$(gpsctl -s)&"                #get fix status
TOPOST=$TOPOST"$(gpsctl -u)&"                #get accuracy
TOPOST=$TOPOST"$(gpsctl -e)&"                #get datetime (YYYY-MM-DD HH:MM:SS)
TOPOST=$TOPOST"$(gpsctl -c)&"                #get course over ground Depreciated! but work
ing....

TOPOST="${TOPOST//[[:blank:]]/}"            # remove blank spaces

echo $TOPOST                                # for debuging pupose

echo $TOPOST | nc 192.168.2.197 3000        # IP/port of the computer on which the OpenDLV
 router microservice is running

#curl -k https://86.50.169.5/log.php?"$TOPOST"  #Post the data via curl. Current IP p
oints to a tiny VM on Pouta
sleep 1
done
```

Figure 29: Router script for sending GPS and LTE-telemetry data using NetCat