INSTITUTE OF INFORMATICS

LUDWIG-MAXIMILIAN UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Enabling Ubiquitous Publish-Subscribe Communication in Cyber-Physical Systems

Ubiquitäre Publish-Subscribe-Kommunikation für Cyber-Physische Systeme

Author:	David Hettler
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Stefan Kugele
Submission Date:	20/05/2018

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

Munich, 20/05/2018

(David Hettler)

Abstract

Several developments in recent years are the cause of a significant increase of complexity in automotive cyber-physical systems. This trend is ongoing and is expected to reach unprecedented levels with the continuous progression towards autonomous driving. These new, increasingly complex software-driven functions impose demanding requirements on vehicular computing systems. Not only do they drive the need for high-performance hardware but also the need to collect unrivaled amounts of data for machine learning and other data-driven applications. Current E/E architectures struggle to provide the necessary means to cope with this situation. Thus, current research is investigating the possibility of offloading computations to remote data centers ("the cloud") as means to support the constrained vehicular on-board systems and to save energy. In line with this spirit, this thesis presents a novel method which enables cloud connectivity for vehicles. The method proposes to split vehicular functionality into self-contained, portable *services* which may be replicated and deployed in both, the vehicle and the cloud, in the exact same way. The services communicate anonymously via multicast-enabled publish-subscribe middleware and are organized in a ubiquitous, self-governing overlay network that spans from the vehicle into the cloud. Through that overlay network, which is realized by means of software-defined networking, services may communicate in a location-transparent fashion, i.e., they are entirely oblivious to their own and their peers' locality. Thus, the system promotes anonymity, agnosticism and isolation which greatly aids in achieving a high degree of scalability, extensibility and loose coupling. A proof of concept implementation of the approach is presented and evaluated w.r.t. its performance and reliability attributes, and other aspects. As part of this evaluation effort, an in-depth discussion about the system's aptitude and limitations is provided. The discussion is supported by a number of benchmarks which were carried out on a purpose-built testbed consisting of a cluster of Raspberry Pi computers connected to a cloud. The results reveal that the approach is indeed feasible and performs well. However, there is evidence that the employed technologies do not meet the high demands of the automotive domain. Nevertheless, the fundamental mechanisms are successfully demonstrated.

Contents

1 Introduction

1.1 Motivation

Recent trends in the automotive industry are introducing new, increasingly complex software functions into vehicles [1]. In particular, autonomous driving and advanced driver-assistance systems (ADAS) are a major force behind this development. Such systems leverage modern methods coming from computer vision and artificial intelligence (AI) research which are demanding, both in terms of computational power as well as in the volume of data they require. For instance, "Junior", Stanford's entry in the 2007 DARPA Urban Challenge,[1] uses a planning system which generates thousands of trajectories per second and then chooses the one that best fits the current situation [2]. Another example is "Boss", the winner of the challenge, which utilizes a similar planning algorithm that, under consideration of the vehicle's state, generates sequences of feasible maneuvers to dodge obstacles on the road [3]. These are just two examples of computationally demanding processes which will become commonplace with the continuous progression towards fully automated driving. The increased computational demand is in conflict with current embedded on-board computers which are typically severely limited in their computational capacities. As these fail to address the increased needs, additional, more powerful computing nodes are being implemented in automated driving test vehicles (e. g. [2]–[4]). This solution is less than optimal as the added hardware is expensive, takes up space and increases the weight of the vehicles. Furthermore, energy consumption will become an increasingly important factor to consider as electric mobility will play a major role in the future [5]. The added hardware will unnecessarily draw power which is predominantly intended to fuel the vehicle itself. Thus, while additional on-board computing hardware is inevitable, a goal should be to reduce it by as much as possible [6].

A potential remedy to overcome this problem is cloud computing [7]. Cloud infrastructures, in essence, are remote data centers which have seemingly infinite amounts of computing resources and storage space available to them. Through auto-scaling, resources can be solicited according to situational demand which prevents overprovisioning and helps to reduce operational costs and to save energy. In the future, computationally intensive functions enabling advanced functionalities, such as autonomous driving, may

[1] The DARPA Urban Challenge was a competition for autonomous vehicles.

be supported by—or even entirely offloaded to—the cloud [8]. This endeavor will be greatly facilitated by the upcoming 5G cellular network which enables high throughput data transmission while providing low latencies and reduced power consumption [9].

Despite the fact that considerable efforts are currently made to push the wide spread adoption of 5G [5], it can not realistically be guaranteed that vehicles will be connected to the Internet at all times, e. g., when navigating through geographically remote areas. In addition to the deficient availability of cellular networks, connectivity in mobile systems is characterized by frequent successions of connection losses and reconnections. For this reason it is vital that all critical vehicular functions remain "offline", i. e., the vehicle's operability should not be reliant on cloud services. Rather, the cloud should fulfill a *supportive* role.

1.2 Objective

The goal of this thesis is to devise a system that connects vehicles to the cloud as means to

- offload computations and reduce energy consumption,
- facilitate data collection for real-time monitoring and analytics purposes, and to
- introduce redundancy in an effort to improve road traffic safety.

The system ought to be unobtrusive in that the cloud shall not interfere with the operation of the vehicle itself. Much rather, the cloud shall act as an optional addendum that performs supportive tasks on the behalf of the vehicle. A major challenge in this enterprise is the mobility aspect of cars. It cannot always be guaranteed that a connection to the cloud is available. As a consequence, cloud-based functions may be available for a few seconds, and then become unavailable again. The system must be able to deal with extreme fluctuations in the network's availability. When connectivity is restored, operation must resume as if no interruption occurred. A seamless and fast transition from cloud-operation to on-board-operation and back again must be possible.

Disclaimer. It must be stressed that it is *not* the goal of this thesis to realize a fully functional, ready-to-use system, but merely to present a working proof of concept. For the implementation, concessions w. r. t. safety and other requirements relevant to automotive use cases will have to be made and questions regarding the system's practical usability will remain unanswered. This thesis presents a technical realization which allows for the migration of vehicular functionality to the cloud. As such, it is only concerned with architectural and implementation aspects. It is not the purpose to present high-level repartitioning strategies or a management and orchestration system. Such topics may build *on top* of this thesis' approach.

1.3 Contributions

In this thesis, a novel method to address the previously stated problem is presented. The prototypical implementation features a combination of technologies which, to the best of the author's knowledge, has not been subject to scientific investigations yet. Thus, in this work, experiments are presented on the basis of which the interplay of these technologies is evaluated. Moreover, in the process of writing this thesis, contributions in the form of two research papers were made. Firstly, the paper titled "Data-Centric Communication and Containerization for Future Automotive Software Architectures" [10] investigates the prospect of leveraging publish-subscribe communication and containerization technology as building blocks to devise an automotive service-oriented architecture (SOA). This paper was presented and published at the IEEE International Conference on Software Architectures 2018 (ICSA) in Seattle. Secondly, a paper titled "Elastic Service Provision for Intelligent Vehicle Functions" [11] was created in which the main contribution of this thesis was used as foundation to concept an intelligent management component that allows for vehicular E/E systems to dynamically adapt to operational situations. At the time of this writing, this paper is yet to be published.

1.4 Structure of the Thesis

The remainder of this work is structured as follows. First, preliminaries are given in Chapter 2, briefly explaining all concepts and technologies relevant to the presented approach. The next chapter, Chapter 7, is dedicated to the summarization and discussion of similar approaches and other related work. Chapter 3 provides a conceptual description of the approach and lists example use cases and requirements. In the subsequent chapter, Chapter 4, the actual realization of the approach is described, going into detail about how technologies were leveraged to exemplarily implement the idea. In Chapter 5, the approach is evaluated. The evaluation is supported by a number of benchmarks which were conducted to assess the system's feasibility and to quantify its quality attributes. What then follows in Chapter 6 is a discussion about the system's aptitude as well as its limitations. The thesis concludes with Chapter 8 where this work is summarized and future work is suggested.

2 Preliminaries

2.1 Cyber-Physical Systems

Cyber-physical Systems (CPS) are systems which are concerned with the coordinative interplay of computational and physical resources and processes. A CPS may be anything from a smart home application which allows users to unlock their doors via smart phone app, a collection of sensors dispersed in an industrial plant that report on a product's quality attributes, to a smart grid that algorithmically controls a region's electricity production and distribution. All these examples have in common that they involve physical, tangible appliances which are tightly intertwined with computing systems. CPS retrieve information about the physical world through sensors, bring the sensed data into machine readable form, process it and act upon it to exert influence on their environment. More often than not, CPS occur as collections of dispersed components which are connected not only to each other, but also to the Internet. Thus, a CPS' perception is not limited to its own physical locality, but it can also make use of globally available data and services [12]. In this context, the term *Internet of Things* (IoT) is commonly used to describe connected amalgamations of computing devices that exist ubiquitously in the environment.

The one type of CPS that this thesis is mostly concerned with are vehicles. Modern vehicles perceive the environment through a variety of sensors (e. g. cameras, lidars,[1] short and long-range radars, etc.), and use that information to make informed, autonomous decisions to ensure the continuous operability of the vehicle and to implement comfort functions. Beyond the immediate, physical awareness (distance, velocity, location, etc.) that is needed to make such decisions, the CPS needs to deduce situational awareness (*how is the information relevant to the current situation?*) as well as contextual awareness (*which other factors may influence the decision?*). The evolution of vehicular braking systems is an example of a traditional, mechanically driven subsystem that, with time, turned into a sophisticated, algorithmically controlled "system of systems" which considers more factors than just the brake pedal's position. This example is lent from the remarks of Broy *et al.* [12]. In the late seventies, the Anti-lock Braking System (ABS) was developed as a simple mechanism to ensure maneuverability of the vehicle while braking. It functions

[1] "LIght Detection And Ranging"

rather primitively: if a sensor detects that a wheel rotates too slowly, compared to the vehicle's speed (indicating a wheel lock), valves are actuated to reduce the brake's hydraulic pressure, effectively causing a reduction in braking power to ensure the wheel's traction needed for steering. This process, in its simplest form, is a purely mechanical reaction to sensor input. This is in contrast to modern braking systems which, in addition, take situational factors into consideration. For instance, Electronic Stability Control (ESC) monitors the state of the vehicle and compares that to the state that the vehicle *should* be in, according to the driver. For this, the system considers, e. g., the position of the steering wheel and thus deduces the driver's intent to maneuver the vehicle in another direction. If the vehicle's actual condition and the desired condition deviate by a too large degree, the ESC takes measures. More advanced braking systems, such as Active Brake Assist (ABA), additionally rely on cameras to detect obstacles in front of the vehicle, present a warning to the driver if danger is imminent, and, if required, brakes autonomously. The evolution of vehicular braking systems exemplifies the gradual progression from isolated, single-purpose functions to complex CPS which make decisions based on contextual information inferred from sensor data from a multitude of different subsystems. Likewise, similar evolutions can be observed in many other parts of our everyday life as ordinary "things" become connected and computing becomes pervasive.

2.2 Distributed Systems

Modern vehicular E/E-architectures[2] are comprised of a large number of distributed, connected sensors, actuators and Electronic Control Units (ECUs) [13], and hence, fall into the category of *distributed systems* [14]. The term "distributed system" entails many things and just as many definitions of the term exist. A definition that most would agree upon is the one given by Van Steen and Tanenbaum [15]:

> "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system."

The first aspect to consider in this definition is the word *collection*. Distributed systems are made up of a number of *nodes* which may occur in the form of either hardware devices, or software processes. Nodes work together to achieve a common goal. For this, they need to exchange messages. More on the communication aspect is discussed in Section 2.3. Furthermore, the definition names *autonomy* as a characteristic of distributed systems. Nodes, on their own, are autonomously acting entities, with their own, individual sets of rules and behavior. At the same time the system needs to be kept together. *Groups*, which individual nodes may join, are a tool to achieve this. There are open groups, which

[2] "Electric/Electronic architectures"

every node may join, and closed groups, which employ an authorization mechanism to control access. Groups aim to provide *coherence*, which is another aspect of the definition given above. By the given definition, however, the coherence of the system is only *perceived*. I. e., to users, whether they are humans or programs, a distributed system presents itself as a single entity, even though it is in actuality comprised of a number of physically dispersed processes and resources. This principle is called *distribution transparency*. Van Steen *et al.* [15] separate this principle into several aspects. The first one to note is **location transparency**. At the root of location transparency is the desire to hide the physical location of resources. A common method to achieve this is through the assignment of names. A user who wants to access a resource can thus refer to it by name, e. g. a URL,[3] while remaining oblivious of its actual location. Under the hood, communication is still based on location-dependent addresses, but such details can be hidden by a name resolution service. Naming furthermore facilitates another kind of distribution transparency: **Relocation transparency**. As the name suggests, relocation transparency aims to hide the fact that resources may move from one location another without the user taking notice. In the example of the aforementioned name resolution service, this can be achieved by reconfiguring the service to redirect users to a location different than the one previously known. To the user, still, the resource appears to be in the same location as it only knows its name. Related to relocation transparency is **migration transparency**, but in contrast to the former, migration transparency refers to the mobility of the *user*. A migration transparent system allows a user to roam freely, while maintaining connectivity to the rest of the system. Examples of such systems are cellular networks.

Another aspect of distribution transparency is **replication transparency**. Distributed systems often provide means to replicate nodes or resources, e. g. to improve availability and scalability. Replication transparency states that all such replicas appear as one to the user. In addition to scalability, replication can be helpful to provide failure resilience. If a given node fails, and a replica is available, the user can be automatically redirected to the replica. This is also known as **failure transparency**.

Another way in which a distributed system can be transparent is in terms of concurrency. Resources are often times shared among a number of users which are concurrently using the system. A desirable trait thereby is to hide this fact from the user such that the they are lead to believe that they are the only one with access to that resource. This is called **concurrency transparency**.

[3] "Uniform Resource Locator"

2.3 Middlewares

A prerequisite for the coherence property of distributed systems is the need for nodes to engage in collaboration. More precisely, distributed applications need a way to pass messages, or data, between different threads of execution. For this purpose, *middlewares* [16] are commonly used. Although message passing is a prime example of a middleware's use case, there are many other important concepts for which middlewares exist, e. g., transaction management in database systems. The primary goal of middlewares is to abstract away complex concepts so that programmers can focus on implementing business logic and shipping features, instead of having to deal with the underlying specifics. Such "specifics" might be, for example, protocols, hardware platforms, state management, security (encryption) etc. Middlewares are implemented as a software layer that sits between operating system and the actual applications. They are often included in the form of libraries that make the middleware's functionality available to the programmers by means of an Application Programming Interface (API).

The need for middlewares becomes particularly evident in the example of messaging. Messages need to be sent over a physical mediums which are often, by nature, unreliable. To ensure reliability, many things need to be taken care of, e. g., error detection, QoS[4] compliance measures, repetition mechanisms, etc. In addition, guarantees must be given that messages are delivered to the right receivers. Therefore, addressing and routing mechanisms must be in place. These are just a few examples of hard-to-solve problems related to messaging. Managing these things manually, and implementing according measures from the ground up is hardly feasible for programmers. Messaging middlewares greatly simplify this process.

```
auto& middleware = MW::register_by_name("Alice");
middleware.broadcast("Hello,␣World!");
```

Listing 2.1: A code snippet demonstrating a broadcast dispatch via middleware

```
void receive(const std::string& message, const std::string& sender)
{
  // message = "Hello, World!"
  // sender = "Alice"
}
```

Listing 2.2: An exemplary callback function to receive messages via middleware

Listing 2.1 and Listing 2.2 show code snippets demonstrating the sending and receiving capabilities of a hypothetical messaging middleware. In the example, all a programmer

[4] "Quality of Service"

needs to do to send messages is to invoke a simple function, e. g., `sendBroadcast` in Listing 2.1. The middleware then ensures that the message is delivered to the right recipients over whichever transport is available. To receive messages, the programmer may, in the case of this exemplary middleware, define a callback function which the middleware calls automatically whenever a new message is available. An example of such function is depicted in Listing 2.2. In the callback function's body, logic could be implemented to process the message content passed in the `message` parameter.

2.4 Cloud Computing

The idea of cloud computing is to provide access to remote computing resources (e. g., networks, servers, storage, applications, and services) in a convenient, on-demand manner [7]. Customers[5] can rent these resources to use them at their will. By outsourcing their IT infrastructure into the cloud, companies can significantly reduce capital and operational expenditures (CAPEX/OPEX) that are typically associated with running an on-premise infrastructure. Other benefits include easier maintenance and accelerated time-to-market times [17]. Several pricing models for cloud services exist. Customers often have the choice between a set monthly fee, or they may take advantage of a pay-per-use model, whereby providers bill their users, e. g., on the basis of CPU time.

Cloud infrastructures are typically implemented as multi-tenancy systems in which the same hardware is shared among many customers. This principle is known as "resource pooling". An enabling technology for this is virtualization. Each customer is assigned one or more virtual machines (VMs) running on one or more physical servers. For its users, the alloted computing environment appears as a single, isolated physical machine. The amount of disposable resources can be controlled for each VM individually, allowing for fine-grained resource tuning according to demand.

A major selling point of cloud computing is scalability. Many cloud providers allow for the dynamic allocation of resources depending on demand. To customers, the available resources appear as if they were unlimited when in actuality the substrate resources are alloted and released under the hood in an elastic manner. To steer this behavior, *elasticity controllers* can be employed which allow customers to define rules to control when and how scaling measures are performed. E. g., when a CPU utilization threshold is reached, the system can be instructed to automatically launch an application replica. Subsequently, a load balancer can be used to distribute the load between the instances [18]. This way, overprovisioning (too many resources allotted), as well as underprovisioning (too few resources allotted) can be avoided, and OPEX can be reduced.

[5] The term "Customers" is used to describe customers of *cloud providers*, i. e. ordinary companies. These customers, in turn, may use the cloud to provide cloud-based services to *their* customers, i. e. end users.

Cloud computing occurs in the form of several usage models. The most notable ones are: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS) [7], but many other models, mostly derivatives of the three mentioned, are in use today.

IaaS In the IaaS model, sheer, usually virtualized, hardware is provided, on which customers can install and run arbitrary software. Customers have no control over their virtual infrastructure's hardware composition, but may exert influence on the operating system level (from the kernel up).

PaaS The PaaS model presents a higher level view on the infrastructure. In this model, customers don't have full control over their VM instance. Instead, they can deploy their software in predefined application-hosting environments [7] which are typically centered around a certain technology, e. g. .NET, Node.js, etc.

SaaS The final cloud usage model is the SaaS model. SaaS typically describes applications which are hosted on cloud platforms. These applications are usually accessible by means of thin clients, and most notably web browsers. End users have the least control over the cloud service and can only exert influence via application-level configurations [7]. Examples of SaaS applications are browser-based e-mail services or video streaming platforms.

2.5 Overlay Networks

Distributed systems are often organized as *overlay networks*[6] [19]. An overlay network is a logical network which connects nodes, or peers, in an abstract, high-level manner. Naturally, in order to enable information exchange between the peers, overlay networks require a substrate physical network (*underlay*) over which data can be transmitted. As opposed to physical networks, which connect *physical machines*, overlay networks connect *processes*. An important thing to note is that overlays aim to be as decoupled as possible from their respective underlays, such that both networks may evolve (change topology) independently without affecting their operability [15]. For example, an added node in an overlay network does not necessarily entail the addition of a physical node. Conversely, the removal of a physical node does not necessarily result in connection loss of a logical node. An example of this is the Internet [20], which spans a worldwide network of nodes that is resilient to failures, such that, when a physical node breaks, a redundant path to the target node may be taken. Other examples of overlay networks are VPNs, Peer-to-peer (P2P) networks and voice over IP (VoIP) systems. A schematic example

[6] The term "overlay networks" is often used interchangeably with its abbreviated form, "overlays"

of an overlay network is depicted in Figure 2.1. The overlay spans over an underlay network, which in turn consists of three sub-networks. These sub-networks could be, e. g., company's internal network, the Internet, and a network within a data center.

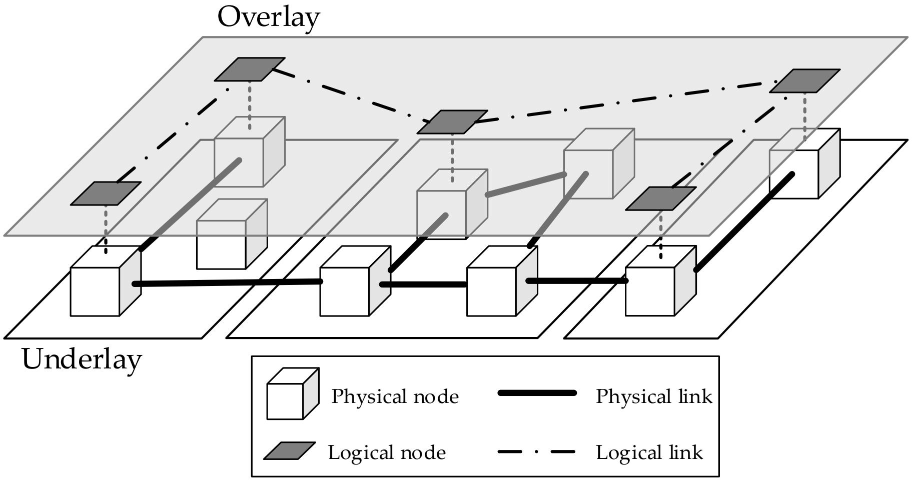

Figure 2.1: An overlay network on top of a physical underlay network which consists of three sub-networks

A distinction between two types of overlay networks is made: *structured* and *unstructured* ones. In the former, peers are organized in a specific, deterministic manner, such that each node has its firm place and an immutable set of neighbors. Unstructured overlay networks, on the other hand, allow the topology to change dynamically. In order for this to work, each node maintains an ad-hoc list of neighbors that is to be updated continuously [15]. In the context of this thesis, unstructured overlay networks are of particular interest due to the dynamic nature and the reliability characteristics of mobile systems.

Virtual Local Area Networks. There are a number of ways to realize logical network overlays. VXLAN[7] and NVGRE[8] two common examples which implement overlays on network datagram level. The technology used in the context of this work builds on VXLAN [21], an encapsulation protocol based on the long-established Virtual LAN (VLAN). VLANs make it possible to segregate a physical network into several isolated, logical networks. Each of those logical networks is assigned an identifier, a so-called *VLAN tag*. All packets sent within a VLAN network have their respective network's

[7] "Virtual eXtensible Local Area Network"

[8] "Network Virtualization using Generic Routing Encapsulation"

VLAN tag attached on them at data link level (layer 2 in the OSI[9] model). Based on the packet's tag, the substrate infrastructure can make decisions on how, and where, to forward packets. This allows for the realization of broadcast domains: each broadcast originating from a given VLAN network will stay within that network, regardless of how many other virtual networks exist alongside. This helps to significantly reduce network traffic.

Since the time when VLAN was devised much in the technological landscape has changed. In particular, virtualization has made great strides with the advent of cloud computing and multi-tenancy systems. Nowadays, thousands of virtual nodes dispersed throughout different clouds and on-premise infrastructures need to be connected. With its limited support for only 4096 virtual networks, VLAN cannot meet these increased demands. Thus, VXLAN was devised as a way to deal with the changed requirements. The most notable changes, compared to VLAN, is the support for up to 16 million virtual networks and the option to assign addresses to *processes*, rather than *devices*. In contrast to VLAN, which is an intrinsic part of layer 2 protocol frames, VXLAN is an independent protocol which is placed on top of UDP connections (in the application layer). This is illustrated Figure 2.2 in which the VXLAN protocol stack is depicted. VXLAN works by wrapping the original data packet sent from one application to the other in a VXLAN frame which is then placed inside a UDP frame. As a result of this, two sets of addresses exist: the logical (source and destination) MAC addresses embedded in the encapsulated packet, and the physical MAC addresses in the surrounding packet. Consequently, two types of packet switching need to be performed: once on a logical level, and once on a physical level. While physical switching is still performed inside physical network switches, an additional, logical type of switching needs to be introduced. Open vSwitch [22] is an example of a virtual, software-based switch that performs packet switching in the data plane (within the Linux kernel). Since VXLAN utilizes virtual MAC addresses that don't necessarily need to correspond to actual hardware addresses, addresses can be assigned at will for each process or virtual machine. This makes it possible to address single VM instances within an overlay network individually, regardless of physical locality and the underlying network design. An added benefit is that all endpoints may retain their logical address when migrating to other physical nodes. This allows for the creation of dynamically changing networks in which processes may move between different execution environments. Furthermore, VXLAN addresses the problem that WANs, in general, lack support for point-to-multipoint communication. In the context of an overlay network, peers may send messages to several receivers simultaneously, e. g. by means of IP multicast, regardless of their physical location.

[9] "Open Systems Interconnect"

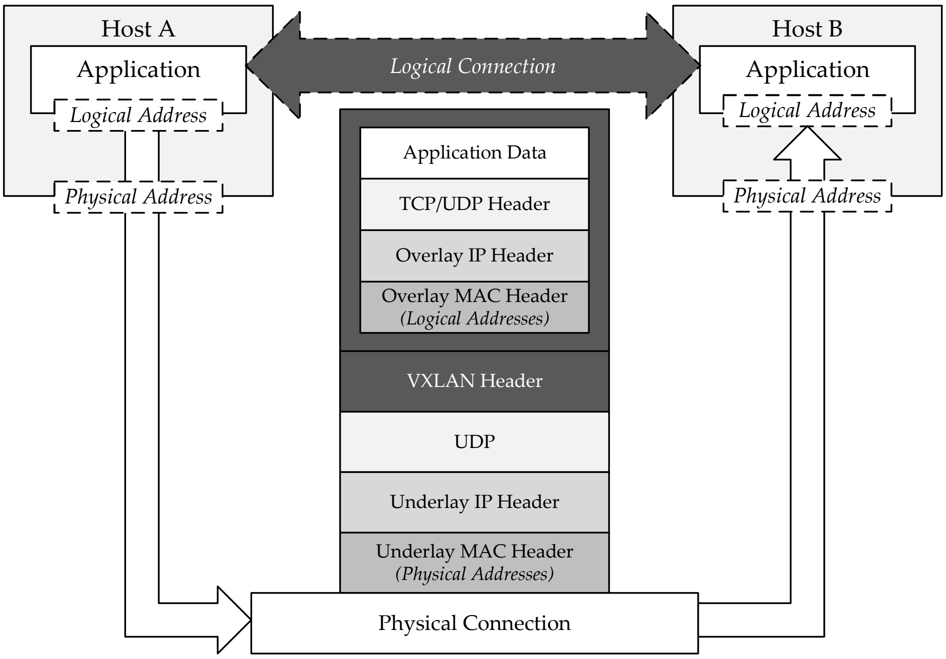

Figure 2.2: VXLAN protocol stack exemplified by two logically connected applications

In essence, VXLAN is a layer 2 overlay scheme on layer 3 networks which effectively creates *tunnels* between Virtual Tunnel Endpoints (VTEPs), indicated by the dashed "Logical Address" boxes in Figure 2.2. Thus, logical connections in VXLAN overlays are often referred to as "VXLAN tunnels".

2.6 Containerization

In recent years, container technology has gained widespread adoption in the software development world. By providing light-weight, portable execution environments for applications, containers greatly simplify the software development and deployment workflow, and thus have become a cornerstone of modern software architectures.

2.6.1 Comparison to Virtual Machines

Containerization is also known as "operating-system-level virtualization" [23], and thereby disassociates itself from traditional virtualization technologies such as virtual machines. A major difference between the two is that containerization does not rely on an intermediate virtualization layer between application and hardware, whereas

VMs employ hypervisors for this purpose. Hypervisors make it possible to run entire operating systems in their own, isolated environments, so-called VM instances. Several instances may run on the same host machine side-by-side without affecting one another, which allows for the realization of multi-tenant systems. Since hypervisors abstract the substrate hardware, a layer of indirection between hardware and the application to be executed is added. The consequence of this is a considerable overhead, compared to native execution. Containers, in contrast, run directly on the host's kernel and thus exhibit near-native performance ([24]–[26]). This approach, however, bears security issues [27].

As mentioned, each VM instance contains its own operating system and kernel. As a consequence, the disk space requirements for VMs are comparatively high. Additionally, the guest OS within a VM needs to be booted into before being operational, which slows down the VM's start-up speed. Containers, in contrast, do not include their own OSes and kernels. Thus, container start-up can be performed in a matter of milliseconds and is more akin to spawning a process, rather than booting an OS. This enables them to be created and destroyed depending on situational demand, which allows for the implementation of highly elastically scaling systems. Considering their advantages over hypervisor-based virtualization, such near-native performance, sub-second boot times, minimal disk space usage and their energy efficiency [28], containers bring qualities to the table that are relevant especially to embedded systems, which are, by their very nature, resource-constrained. Unsurprisingly, the possibility of leveraging containerization in embedded environments has received substantial attention in recent years (e. g. [29]–[31]).

Containerization and virtual machines are often regarded as competing virtualization technologies. However, both are not mutually exclusive. In fact, most IaaS providers base their infrastructure on traditional VMs, which then, in turn, host container runtimes [32]. Containers and VMs shall thus be seen as complementary, rather than competing, technologies.

2.6.2 Container Internals

Like virtual machines, containers aim to isolate processes from their environment. In the case of containers, isolation is achieved by a kernel feature called *kernel namespaces*.[10] Namespaces wrap global system resources, such as network devices and mount points, and present them to processes as if they were dedicated to them. When a process runs in the context of a certain namespace, its access and view is limited to that namespace. There are seven types of namespaces, each of which isolates a different aspect of the host system. An example for this are *process* namespaces which allow for the creation of alternative views on the process tree. For instance, consider a hypothetical process with PID 345. By assigning a process namespace to that process, it can be led to believe

[10] `www.man7.org/linux/man-pages/man7/namespaces.7.html`

that it has the PID 1 and is the only process running on the system. Thus, the processes' view on other processes is restricted, and since a process can only access what it can see, isolation is achieved. In a similar way, a processes' access to networks interfaces, the file system, user groups and other parts of the host system can be restricted.

In addition to providing isolation, containers employ resource management mechanisms which make it possible to allocate and limit the resources available to them. Resource management for containers is implemented by a kernel feature called *control groups*, or *cgroups*[11] in short. Cgroups allow for the organization of processes in hierarchical groups. On these groups, resource constraints can be imposed (e. g. CPU-time, access to devices, memory budget, etc.). Through this mechanism, processes can be bundled together, effectively forming *containerized process trees*, which can be restricted in what they can do and which resources they may take up. This opens up the possibility for container engines to exert fine-grained control over each processes' resource utilization and to further isolate processes from the rest of the system.

The two concepts (namespaces and cgroups) can be applied individually to any given process running on a Linux system. Only when combined together, such that processes are isolated through namespaces and restricted through cgroups, we speak of containers. In consideration of this, it becomes clear that a containerized process is not much different from a regular process. Rather, containers should be viewed as processes which are *augmented* by these two concepts.

While it is possible to create containers manually utilizing the aforementioned native kernel features, such undertaking is rather cumbersome. For this reason, several tools are being developed which aim to streamline the use of containers. A few notable examples are *Docker*,[12] *RKT*,[13] *CRI-O*,[14] *Railcar*[15] and *LXC*.[16] Among these, Docker is undoubtedly the most prominent one, and arguably the first one to make Linux containers accessible for general use. As it is the most mature containerization solution, Docker was chosen for the approach presented in this thesis.

[11] `www.man7.org/linux/man-pages/man7/cgroups.7.html`

[12] `www.docker.com`

[13] `www.github.com/rkt/rkt`

[14] `www.cri-o.io`

[15] `www.github.com/oracle/railcar`

[16] `www.linuxcontainers.org`

3 Approach

In this chapter, the main approach of this thesis is presented. The chapter starts with an abstract description of the approach with the aim of providing a high-level overview of the underlying concepts. Then, exemplary use cases are described, and finally, a list of requirements and desirable quality attributes is given which forms the basis for the evaluation part in Chapter 6.

3.1 Concept

The objective of this thesis is to explore a novel method to handle the increased demand for computational power and data collection in mobile cyber-physical systems, and in particular, automotive systems. For this, an approach is presented that allows for the outsourcing of vehicular functions to the cloud. A special requirement is the unreliable connectivity common to mobile systems. To deal with this, the solution must be able to perform the offloading in an instantaneous, fail-safe, and automated fashion. When cloud connectivity cannot be provided, the operation of the vehicle mustn't be affected in any way and the previous state must be reestablished as soon as connectivity is restored.

Here, this thesis' method to achieve this is presented. The method proposes to split vehicular functionality into portable, isolated units which may run within the vehicle and the cloud in the exact same way. To achieve horizontal scalability, the units may be replicated and placed in different locations, e. g., once in the vehicle and once in the cloud. All functional units communicate in a bus-like system that virtually extends from the vehicle into the cloud. How exactly this is achieved is described in the following sections.

3.1.1 Partitioning of Functionality

Fundamentally, the approach is based on the idea of splitting functionality into isolated units which may be deployed redundantly on both, the vehicle's on-board system, and on high-performance machines in remote data centers. For this, the system needs to facilitate the clean separation of functionality according to responsibilities. The *service-oriented* architectural paradigm (SOA) [33] is a proven method for achieving this. SOAs aim to divide functionality into isolated, loosely coupled modules, so-called *services*.

Each service provides an offering to users, or other services, by means of a clearly defined, preferably immutable and machine-readable interface. By guaranteeing that these interfaces rarely change, a formal *service contract* is put in place which ensures the continuous interoperability between the services. Examples of a service offering may be the provisioning of some sort of data, or functionality (algorithms and business logic) that is executed on behalf of a service consumer. All services of the system are working together like cogs in a machine to achieve a common goal, i. e., the continuous operation of a vehicle.

3.1.2 Communication

Traditionally, E/E-architectures build on the principle of message buses as means to connect the system's dispersed components (ECUs) [13]. In bus systems, all components write data samples to, and read data samples from, a shared medium (the bus). Individual components typically have no knowledge of each other's existence—they only know the data they provide and push it on the bus, or read the data they are interested in from the bus. This paradigm leads to a loosely coupled system in which components have as few dependencies among each other as possible. Loose coupling, in turn, tremendously facilitates extensibility and scalability. A concept closely related to message buses is the publish-subscribe principle by which service providers (publishers) and service consumers (subscribers) communicate on the basis of topics. In effect, a publish-subscribe system behaves in accordance with a bus system in which the shared medium is logically segregated into domains (i. e. topics).

In line with the traditional bus-centered nature of E/E-systems, the presented approach builds on a bus-like publish-subscribe paradigm in which services communicate with each other via topics. The characteristic that makes the envisaged bus special is that it virtually extends into the cloud (see Figure 3.1). To facilitate this, a network overlay mechanism [19] is needed. With the help of a virtual overlay network, which is laid on top of the substrate network, a globally dispersed *service mesh* may be created wherein services may communicate beyond the physical boundaries of the network. That way, location-transparent messaging between services is achieved: it does not matter whether a given service runs within the vehicle or within the cloud—the way they communicate remains the same.

3.1.3 Replication

In the envisioned approach, services may be replicated, and those replicated services may be deployed on any other node in the overlay network. The term "replica", or "service instance", will henceforth be used to describe instantiations of services. Replication allows for the creation of reliable and horizontally scalable systems [15]. The approach aims to leverage this principle to achieve failure tolerance and superior computing performance. The division of functionality into smaller parts alleviates replication tremendously, provided that certain design principles are applied. Hence, throughout the design and implementation of the envisaged system, several design goals need to be kept in mind. Firstly, services ought to be **stateless**. Statelessness facilitates replication as sharing state among many instances proves to be difficult. Consistency within a distributed system is hard to achieve and locking mechanisms may introduce bottlenecks [15]. A common practice to remove the need for such mechanisms is to avoid internal state altogether. Nevertheless, *some* sort of synchronization mechanism between all instances of a service needs to be present to guarantee that all instances have a consistent view on data. The publish-subscribe paradigm helps to achieve this, as all instances receive the same data simultaneously by listening on the same topic. A way to facilitate simultaneous data delivery to multiple receivers is to use point-to-multipoint communication, which may be realized, e. g., by means of IP multicast.

Furthermore, services should be **fine-grained**. High-resolution granularity allows for detailed control over which services should be replicated. That way, computational bottlenecks can be more easily singled-out and eliminated. An added benefit of fine granularity is an increase in extensibility and faster software updates. However, trade-offs need to be considered. If services are too fine-grained, they need to communicate more which may slow down the system. Furthermore, too many services unnecessarily increase complexity.

Another design goal to keep in mind is **isolation**. Services should have limited knowledge of other services' location and the service meshes' topology. Through isolation, loose coupling is further promoted. One aspect of isolation is self-containment. Service instances should be bundled together with all their dependencies as independent, portable units that may be deployed on any hardware platform. By packaging services in self-contained environments, they may be moved between different computing nodes. This property is the most important one to achieve cloud scalability. Commonly used tools to facilitate isolation and portability are virtualization and containerization.

3.1.4 Example

Figure 3.1 depicts a schematic example of the approach. The dashed container box on the left-hand side denotes a vehicle. In the vehicle, several interconnected services are deployed: ✦, ◆, ●, ▲, and ✚. The figure omits any details regarding the actual computing infrastructure but only displays the general placement of services (either vehicle or cloud). In actuality, the the vehicular services run on distributed embedded devices spread over the vehicle's on-board system. Several of these services rely on

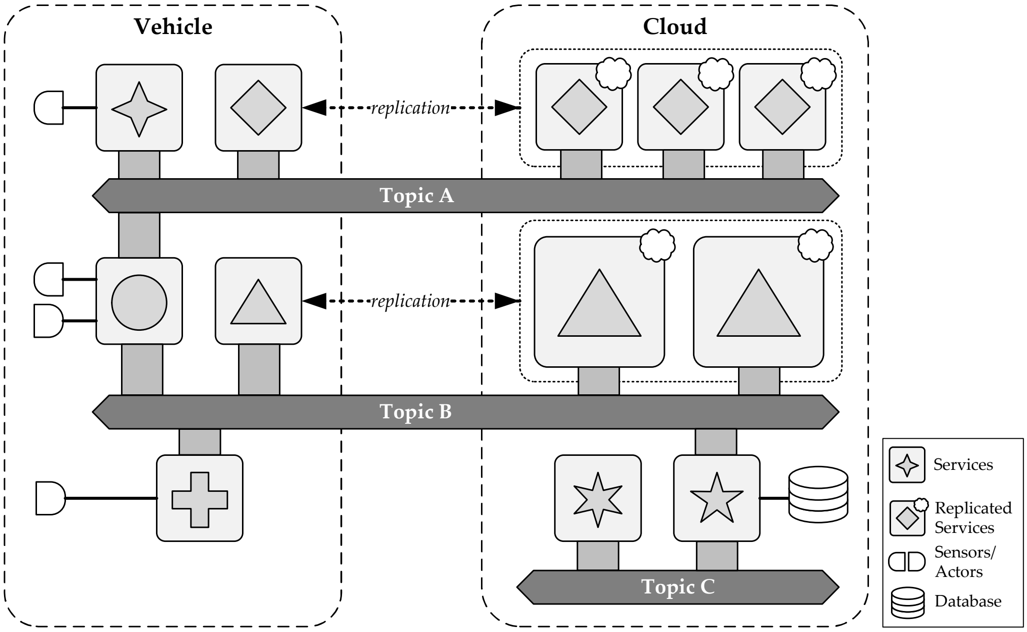

Figure 3.1: Conceptual sketch of the approach

sensor data and/or control actuators (indicated by the small semicircles).

On the right-hand side of the figure, the cloud is depicted, which, in actuality, is a collection of high-performance computing nodes. Vehicle and cloud are physically separated and connected via some sort of network, which the vehicle may access by means of cellular networking technology. On a logical level, vehicular- and cloud-based services communicate via topics (in the publish-subscribe sense) that may, or may not, span between the two otherwise isolated systems. The gap between the two systems is bridged by an overlay network which is placed on top of the cellular network.

Note that some of the vehicle-intrinsic services are replicated in the cloud (◆, ▲). Examples of such services could be functions for calculating trajectories, computer vision-based gaze detection, or machine learning applications. The duplication and varying size of the service boxes depicted in the image is indicative of horizontal and vertical scaling, respectively. It doesn't always make sense to replicate services in the cloud, e. g., because they are computationally inexpensive, or because they require access to sensors and actuators which are inherently bound to the vehicle (✦, ●, ✚). Those services have their firm place within the vehicle and are therefore referred to as *fixed* services. Their counterpart, i. e., services that may run ubiquitously, are called *volatile* services (◆, ▲). A third kind of services are *cloud-only* services, which are entirely optional and only serve to perform background tasks such as monitoring or persistence (✶, ★). Service ★, for example, would aggregate all data posted on Topic B and store it in a database. The discrepancy between fixed and volatile services emphasizes the need to split functionality into fine-grained services so that computational bottlenecks can be isolated and then eliminated. A good advice is to extract services connected to sensors and actuators in minimal "access services" (✦, ●, ✚) that do nothing but provide access to sensor data.

3.1.5 Key Challenges

In the previous sections, a conceptual approach was presented to connect a vehicle's on-board system to the cloud to facilitate computation offloading, data collection, and other use cases. To achieve this, the approach suggests to split functionality into fine-grained services that may be replicated and migrated smoothly between vehicle and cloud. Three key challenges can be identified which need to be addressed in order to realize the system:

(i) **Reliable, anonymous information exchange**: Services must be able to communicate in a reliable manner, without the need for direct references to each other. Furthermore, point-to-multipoint messaging shall be supported, such that messages can be delivered to multiple service instances simultaneously. Publish-subscribe-style communication shall be employed for this.

(ii) **Isolation**: Service instances need to be packaged in self-contained execution environments with all their dependencies to ensure portability. Virtualization or containerization are appropriate means for this.

(iii) **Connectivity**: Services need to be able to autonomously create and maintain connections between each other, regardless of their location. For this, the use of overlay networks is suggested.

3.2 Example Use Cases

The presented system may be utilized in a number of ways. To further rationalize the system's usefulness it is helpful to keep a few usage scenarios in mind.

Data Collection. The approach has the potential to greatly simplify data collection. One could think of an example where passively listening services instances, implemented as subscribers, would run in the cloud at all times for the sole purpose of accumulating telematic/telemetry data. By duplicating all subscribers from the vehicle and placing them in the cloud, it would be possible to mirror everything that happens inside the vehicle in the cloud. With this, a model of the vehicle's state could be recreated that continuously synchronizes with the actual vehicle state in real-time. Due to the anonymous nature of the publish-subscribe paradigm, the vehicle's on-board system would be entirely oblivious of the listening services. The collected data could be used for real-time monitoring and analytics purposes, remote trouble shooting, or to feed machine learning algorithms running in the backend. Furthermore, location-based services and applications utilizing crowd-sourced traffic data would be possible. An example for this type of application is Google Maps' traffic congestion detection which is based on the accumulative GPS information collected from numerous user's Android phones.

Auxiliary Fail-Operational Behavior. The presented system allows services and their replicas to run side-by-side in the vehicle and the cloud. Multicast communication ensures that all service instances are always up-to-date.[1] It is now possible to realize fail-operational behavior by providing backup replicas of certain safety-critical services that would run in the cloud at all times. In the event that one of the vehicular services fails, e. g. due to a hardware defect, the operation would not be interrupted as the cloud-based services would continue where the original service left off. In such case, the vehicle would enter a temporary *limp home* mode to protect the driver while it still can. In a conditionally automated driving scenario (automation level 3), the driver would be presented a prompt, instructing them to take over [34].

Computation Offloading. As service instances in the approach are isolated, self-contained entities, they may be moved to, and run on, any given platform. This allows for vehicular functionality to be easily outsourced to the cloud. This might be useful, e. g., for ADAS and automated driving, for which computationally intensive computer vision and machine learning algorithms are often employed. Such algorithms commonly rely on high levels of parallelization and thus benefit greatly from horizontal scaling. Consider now a service which performs such computations. The service is built in a way that makes it easy to distribute workload among many service instances in parallel. The service scales horizontally, such that the quality of its collective output improves with the number of service instances it employs. An example of such computation could be a trajectory planning algorithm that generates vast amounts of candidate trajectories and evaluates each one separately. Such algorithm would benefit from increased computational capacities as it would be able to explore greater numbers of candidate trajectories.

The service runs a few dozen service instances on the vehicle-intrinsic computing infrastructure. Just enough instances run to produce meaningful results and not to

[1] Provided that connectivity is given.

consume too much energy. This is considered the *normal* state which is active if no cloud is involved. The vehicle, which was previously in a dead zone with no network reception, now connects to the cloud. In the cloud, several hundreds of additional service instances of the same type are deployed, which are now activated to contribute to the service's collective output. Thus, the overall quality of the operation's output improves tremendously. If the vehicle now loses its reception again, the quality of the output drops back to its original level.

3.3 Requirements and Quality Attributes

The previous sections already touched on desirable quality attributes to keep in mind when implementing the approach. Still, a rigor requirement analysis is needed in order to be able to thoroughly evaluate it. To this end, a detailed requirement list is presented which an implementation of the envisaged system should aim to fulfill. The list is loosely based on the work of O'Brien *et al.* [35].

Availability. Availability states how likely it is that, at any given point in time, a system is ready to be used by its users [15]. The period in which a system or service is unavailable is called *downtime*. Sources of downtime can be, e. g., maintenance work, temporary congestion, or any kind of failure. A design goal when building and running software systems is to minimize downtime, and thereby maximize availability. This goal is not trivial to achieve as many sources of decreased availability are hard to predict, e. g. in case of hardware failures. A key technique for handling downtime is redundancy, whereby critical systems, or those susceptible to downtime, have a replacement ready to be used at all times. A prerequisite for redundancy to take effect is that the system features quick failure detection and smooth transition mechanisms so that it can quickly reroute requests to the redundant service. If a system manages to substitute an unavailable component in a way that is virtually unnoticeable, it is said to be failure transparent.

Reliability. Reliability states how long a system can continuously run without failure. A reliable system is available for prolonged periods of time without interruption. Although reliability is related to availability, there is a clear distinction between the two. While reliability concerns a continuous period of operability, availability is concerned with operability at a given point in time. For example, a system that works fine most of the time but becomes unavailable for a few milliseconds every hour is highly available, but not reliable. Hence, a highly reliable system is not necessarily a highly available one and vice versa [15]. In real-time systems, a lack of reliability could easily result in the missing of deadlines, which in turn could have disastrous effects. In case of *hard* real-time system, a single missed deadline is even equivalent to a complete system failure. Ensuring reliability is therefore of utmost importance. As with availability, a way to mitigate the effects of poor reliability is redundancy.

Special precautions need to be taken for *distributed* systems as their communication channels are inherently unreliable [15]. In practice, this means that the messaging system needs to provide guarantees for a timely and robust message delivery. In addition, certain assurances are desirable, e. g., that messages are delivered in order or that every message is delivered only once [35].

Safety. The safety of system states how resilient w. r. t. failures it is, and in case a failure occurs, how well it can handle it. A safe system manages to protect the health and wellbeing of humans involved in the operation of the system and its surrounding bystanders. As human lives are at stake in road traffic environments, vehicles are a prime example of systems that need to be safe.

Critical functions need to operate in a fail-operational fashion, i. e., in case they fail, the situation needs to be handled gracefully, without putting the passengers in danger. To this end, special precautions need to be taken throughout the whole process of development, provisioning, operation, and service of functions. ISO 26262 [36] provides the standards that vehicular E/E systems need to adhere to in order to fulfill the necessary safety requirements. Different parts of the system need to be compliant to different classes of safety, indicated by automotive safety integrity levels (ASILs).

Security. In road traffic, flaws in a system's IT security have direct implications for safety. Since safety is of utmost importance the same is true for security. In today's world, vehicles are to a large extent software-driven [1]. With an increasing number of lines of code comes an increase in complexity and error-proneness, and by that, a larger attack surface breaks open. Furthermore, vehicles are more and more connected to the outside world. Not only are they connected to third-party and OEM clouds, but also to other vehicles and the nearby infrastructure. The situation is made even worse by the fact that many of modern vehicle's subsystems can be controlled by software—even those which are safety critical. If an attacker were to gain access to, say, the steering system, consequences could be dire. Passengers would be put in severe danger.

To enhance security and to preserve the integrity, authenticity, and confidentiality of the system, state-of-the-art encryption and security measures are a necessity. To this end, proper isolation of software components is needed to build up security perimeters which are hard to penetrate. In case an attacker, or malicious code, succeeds in intruding the system, appropriate isolation measures, i. e. sandboxing, can furthermore prevent the attacker from reaching out to other subsystems.

Interoperability. In systems composed of a number of heterogeneous components that interact with each other, there needs to be a common set of rules and semantics that all involved parties must comply with. If such rules exist, so that diverse components may interact with each other, they are said to be interoperable. A major barrier to interoperability is (vendor) lock-in. Lock-in describes a situation in which a certain technology is rooted so deep within the system that the introduction of alternative technologies can hardly be accomplished. Similarly, the technology is hard to remove without considerable effort. Lock-in is detrimental to system design as it creates dependencies, and thus, introduces tight coupling. Especially in the automotive domain, in which many parts of the system are developed by a vast number of independent teams and suppliers [1], interoperability should be a priority. Furthermore, OEMs tend to prefer a slow, stepwise transition towards innovative technologies, rather than jumping in at the deep end. Hence, automotive systems need to be particularly interoperable to existing solutions. A way to achieve a high degree of interoperability is to avoid proprietary, closed-source solutions and to favor open standards instead.

Performance. Performance is the amount of work that a computer system may accomplish within a given amount of time. A system may be performant in a number of ways. Common measures to gauge performance are, e. g., the time it takes to respond to a request (response time), the rate at which work is processed (throughput), or rate in which data is transmitted. Performance also states how efficient a system is in the utilization of hardware resources. This is important especially in embedded systems, where resource constraints are commonplace. Countless methods exist to improve a system's performance. One could make use of, e. g., concurrent programming models, compiled programming languages, or binary transmission protocols, to name a few. In the end, it is important to find an appropriate middle ground between performance and usability.

Scalability. Scalability is a system's ability to expand, or rather, deal with expansion. Often times, expansion is necessary to support an increased number of users or to deal with increased computational load. Generally, a distinction between two types of scalability is made: horizontal scalability, by which workload is distributed across an increased number of nodes (scaling out), and vertical scalability, by which a single node is upgraded with more powerful hardware (scaling up) [15]. While vertical scalability is easier to realize, it is generally less desirable than horizontal scaling as there is an upper limit in how far it can scale. Furthermore, scaling up often involves downtime. Scaling out, on the other hand, is not affected by these shortcomings, and in addition to that, is in most cases more economical. However, horizontal scalability brings complexity, as many individual components need to be coordinated. Another challenge is posed by the question how load shall be distributed among the many components.

Extensibility. Extensibility expresses how easy it is to add functionality to a system without affecting other parts of the system. This includes the extension of the system itself (by adding services), as well as the extension of individual services' functionality (by means of software updates) [35]. The provisioning of new functionality must be possible not only at design-time but also at run-time. Modern automotive software architectures must deal with the fact that vehicular functions may be modified, added, or unlocked at run-time through (automatic) software updates.

Adaptability. Adaptability is a measure for the amount of effort that is needed to change a system to accommodate for changes in requirements and the environment [35]. Key properties of adaptable systems are autonomy, proper abstractions, and continuous monitoring. Adaptability needs to be addressed on many levels. Firstly, the system needs to be able to adapt to changes on software level. As a reaction to changes in the requirements, software needs to evolve. For this, software updates are inevitable, and thus, the system needs to accommodate for that. Another aspect of adaptability is concerned with the mobile nature of vehicles. Vehicles are connected to the outside world primarily by means of cellular networks. While moving, vehicles frequently change the access point, and hence, the topology changes continuously. The system must be able to reliably deal with changes in topology and exhibit great migration transparency properties. For this, service discovery must happen fully autonomously.

Lastly, the system needs to be adaptable in terms of hardware. The goal is to run the same functionality in vehicles and the cloud side-by-side. For this, software needs to be portable between different hardware platforms. Different ways to address hardware interoperability exist. Common means to achieve a high degree of hardware adaptability are virtual machines, emulators and interpreted programming languages.

Testability. There are many ways in which a distributed system may break. Since operational safety is of paramount importance in the automotive domain, testability is a key requirement. Components of the system must be testable in isolation (e. g. via unit tests) as well as in interplay with other components (e. g. via integration tests). For this purpose, modern software development employs continuous integration tools that help to continually validate the correctness of a system throughout the whole development cycle. Automotive software architectures need to be adapted to make it feasible to employ such development practices.

4 Realization

After having discussed the basic concept of the approach in the previous chapter, in this chapter, an exemplary realization is presented. Previously, three key challenges were identified: (i) *reliable, anonymous information exchange*, (ii) *isolation*, and (iii) *connectivity* (cf. 3.1.5). To tackle these challenges, three technologies are proposed:

(i) **DDS** to enable *reliable, anonymous information exchange*,

(ii) **Docker** as containerization tool to achieve *isolation* and portability of services, and

(iii) **Weave Net** as means to provide *connectivity* between the containerized services.

In the following, these three tools are presented and it is described how they are leveraged to realize the proof of concept implementation.

4.1 Data Distribution Service for Data Exchange

Data Distribution Service (DDS) is a messaging middleware standard [37] for distributed applications governed by the Object Management Group (OMG).[1] DDS is designed for mission- and business critical systems with real-time requirements. As such, it aims to function in a resource efficient, predictable and reliable manner, and is subject to minimal computational and transport overhead. Thus, DDS is a great fit for the automotive use case.

4.1.1 Data-Centric Publish-Subscribe

DDS is fundamentally based on the data-centric publish-subscribe (DCPS) communication paradigm. In the publish-subscribe-style communication, data flows between two kinds of entities: publishers and subscribers. Publishers provide data, while subscribers consume that data. A crucial characteristic of publish-subscribe is that data exchange between the peers is anonymous, i. e., publishers have no way of sending data to individual subscribers. Instead, both communicate by means of a shared, logical medium that takes data samples and forwards them to the appropriate receivers. In the context of DDS, this

[1] `www.omg.org`

medium is called *topic*. When subscribers receive a data sample they do not know where that sample originated. Similarly, publishers have no knowledge about where the sent data will end up at—or even if there are any receivers at all. Entities in this system find each other not by way of addressing, but rather on the basis of a shared understanding of what *kind of data* they want to exchange. This approach is called *data centricity*. Data centricity is in contrast to *message centricity*, in which data exchange is driven by *messages*, or *instructions*, which are directed at individual receivers. A prominent example of a message-centric technology is Remote Procedure Call (RPC). To illustrate the difference between the two paradigms consider the example of a temperature sensor (henceforth also "provider") which propagates temperature data to multiple receivers (henceforth "consumers").

Message-Centric Approach. In the message-centric paradigm, the temperature sensor transmits data samples encapsulated in *messages* to the consumers. Messages are directed at each consumer individually, similar to a letter that is addressed to a certain postal address. This requires each participant to maintain their peers' location information (i. e. their addresses) in local memory. Two patterns are common in message-centric communication: "push-based", and "pull-based" (often "request-reply"-style) communication [15].

If message-centric communication is push-based, the sensor needs to know the addresses of all interested consumers a-priori. The provider thus has to maintain a list of consumers that it needs to update continuously. When a previously uninvolved consumer decides that it wants to receive temperature data, it first needs to register to the sensor in order to make its address known. The sensor then has to add the consumer's address to its list of consumers. Similarly, when a consumer is shut down, the sensor needs to remove the respective address, and has to deal with unexpected errors in case a consumer is suddenly unreachable. This (un)registration procedure is the cause of overhead and additional management effort, and makes the system inflexible and hard to scale. In contrast, pull-based communication don't rely on lists of consumers, but consumers request information from the provider. In this regard, pull-based communication is more scalable, however, additional overhead is incurred since two messages (instead of one) need to be transmitted for each data sample: a request and a response. Furthermore, consumers can not predict when a new data sample is available. The situation is made worse when there is not one producer, but many. Thus, both approaches are highly inefficient in the use case at hand.

Data-Centric Approach. The DCPS approach, on the other hand, is exclusively push-based.[2] The temperature sensor is a publisher in the publish-subscribe relationship, and the consumers are subscribers. A subscriber that is interested in the sensor's data only knows that it wants to receive *temperature data*, i. e. data samples of type "temperature", and thus, subscribes to the "temperature" topic. The temperature topic is defined by a unique identifier and a data type. Only data of that specific type may be published on the topic. To ease the use of the middleware, DDS provides typed interfaces to the exchanged data. Through the optional *Data Local Reconstruction Layer* (DLRL) [38], data types can be defined via the DDS IDL,[3] and according language-specific stubs can be generated. The use of type safe interfaces not only improves usability, but also increases safety and error-proneness, as verifications can be performed at compile time.

In the temperature sensor example, the subscriber is entirely oblivious to the concept of *temperature sensors*, or even if there are any sensors—it just listens in on the "temperature" topic. Similarly, the temperature sensor is only concerned with the provisioning of temperature data, and doesn't know which subscribers to send the data to. Thus, the temperature sensor publishes all samples it gathers on the "temperature" topic. If more temperature sensors were to be introduced to the system, they could simply be added by registering new publishers which would post data on the same topic. Since all publishers of a topic identify themselves purely on the basis of their topic, and not by their address or location, a great deal of redundancy can be achieved—all publishers of a given topic are entirely interchangeable. Due to the agnostic relationship between publishers and subscribers, a high level of loose coupling is achieved. This allows for a simple extension of the system, making it extraordinarily scalable.

It is often helpful to think of publish-subscribe communication not in terms of messages that are sent from one entity to another, but rather as a *shared global data space* in which data is universally accessible to all entities involved in the system. In this data space, each entity views data as if it were available in a local storage, when in reality, it is distributed among many other entities.

4.1.2 Under the Hood

Naturally, in order to realize a publish-subscribe system, addresses and locations of nodes are still crucial, as publish-subscribe, like other communication paradigms, rely on IP-based computer networks. This fact, however, is hidden behind the curtains of the middleware (DDS in this case). The middleware is responsible for the registration of publishers/subscribers, the dissemination of data to the appropriate receivers, and the

[2] Although request-reply-style communication is not explicitly supported it can be mimicked (cf. Section 5.2.4).

[3] "Interface Description Language"

enforcement of network reliability policies. In order to do this efficiently, the middleware maintains lists of subscribers, publishers and topics. Many middlewares employ central components called “brokers” for this. Depending on the view point and use case, the centralized nature of brokers may be considered advantageous, or disadvantageous. On the one hand, centralization is associated with simplicity since everything is managed in one spot, and only one single source of truth exists. On the other hand, this approach brings the danger of a single point of failure that causes the whole system to fail if the broker fails. Additionally, centralized brokers may introduce bottlenecks, inhibiting the scalability of the system. As DDS is strongly focused on scalability, DDS employs a *brokerless* architecture in which the functionality of the communication system is distributed among all participating entities. Consequently, message delivery as well as service discovery is performed in a decentralized manner. For this, DDS may optionally employ IP multicast, by which all communication is directed at dedicated multicast group addresses, instead of each node’s individual address. The delivery of data is then the responsibility of the underlying infrastructure.

4.1.3 Wire Protocol

DDS, in itself, is only a standard. As such, DDS does not dictate, in detail, how to implement the concepts presented in the earlier sections. A number of DDS implementations from different vendors exist, all varying in terms of standard compliance, features beyond the standard, licensing, and other distinguishing factors. Compatibility between the respective implementations can be achieved by the use of the DDSI-RTPS[4] (DDS Interoperability–Real-Time Publish-Subscribe) wire protocol, which all major implementations support. Although the RTPS protocol is directly related to DDS, its specifics are outsourced in a separate standard [39]. As a wire protocol, RTPS is deliberately tailored to DCPS-style communication. As such, it supports reliable multicast-enabled data exchange and service discovery over unreliable and connectionless best-effort transports such as UDP/IP. Special emphasis is put on robustness, fault tolerance, scalability and performance. Thus, RTPS was conceptualized with safety-critical real-time applications in mind. Although RTPS is the preferred transport method for DDS applications, it is not the only option. Many DDS vendors support TCP, UDP, and shared memory for cases in which performance is critical.

[4] The “DDSI” in DDSI-RTPS is often omitted. The shortened form will henceforth be used.

4.1.4 DDS Components

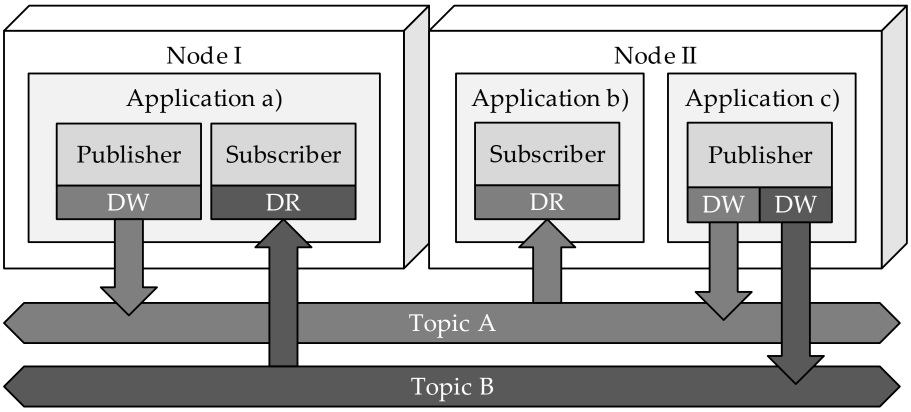

Figure 4.1: An example of a distributed application connected via DDS

The DDS specification defines a number of components which are introduced in the following. Figure 4.1 depicts an example highlighting how the components are related. In the example, there are two hardware nodes connected by an unspecified computer network. Each node can execute multiple applications simultaneously. The applications communicate with each other by means of DDS.

Topics. *Topics* form the basis on which peers communicate in the publish-subscribe paradigm. When a Publisher offers a certain kind of data, it does so by means of a Topic. When a data sample should be written, the Publisher pushes that data sample to a Topic appropriate for that sort of data. Conversely, when a Subscriber is interested in receiving data of a certain kind, then it subscribes to a Topic associated with that data. A Topic is uniquely defined by an identifier (name), a data type, and a set of QoS policies (see below). The data type represents the message format for data samples published on that Topic. For example, a suitable data type for a Topic concerned with temperature data would be a `struct` containing a single `float` value representing a temperature reading. In Figure 4.1 two Topics are depicted (Topic A and Topic B), each with a number of associated Data Writers and Data Readers (see below).

Publishers and Data Writers. *Publishers* are entities that provide information in the publish-subscribe communication paradigm. Publishers are not bound to a single Topic. Instead, they contain one or more *Data Writers*, each dedicated to a single Topic, who perform the actual data submission. Thus, Publishers may be seen as containers for a number of (unrelated) Data Writers. This concept is emphasized in Figure 4.1 where a

Publisher in Application c) controls two independent Data Writers that both write to different Topics.

Subscribers and Data Readers. *Subscribers* are the exact compliment to Publishers in that they seek to receive data from Publishers. Analogous to Publishers, Subscribers are a way to group together sets of *Data Readers*. Data Readers are entities whose purpose it is to receive data samples from their respective Topic. Through its API, DDS allows Data Writers to receive data in three ways: either by waiting for data samples (blocking the main thread), by pro-actively polling for new data samples, or by specifying asynchronous callbacks which are invoked whenever a data sample arrives.

Domains and Domain Participants. *Domains* are the DDS way of grouping together sets of coherent *Domain Participants* and to separate those sets from each other. Speaking in terms of distributed systems, Domains are a mechanism to manage group memberships of nodes [15]. Domain Participants are entities that belong to a particular Domain. Each Publisher, Subscriber and Topic is derived from one Domain Participant and is therefore dedicated to exactly that Domain. As a consequence, participants of different Domains are entirely separated from each other and there is no way for them to interface with each other. Depicted in Figure 4.1 is only one Domain. However, there could just as well be other Domains.

4.1.5 Quality of Service

One of DDS's salient features is its intrinsic support for Quality of Service (QoS), which is realized by so-called "QoS policies". QoS policies specify attributes that can be used to control each component's behavior and quality properties. They make specifying a component's behavior a matter of *configuration*, rather than *implementation*. For example, consider a Data Writer which writes data to a Topic at a high rate. A Data Reader may be interested in that data but not at such a high rate, e. g. because it runs on an embedded device powered by a battery and therefore needs to manage power consumption carefully. The Data Reader may now apply a `TIME_BASED_FILTER`, which instructs DDS to block all samples that exceed a specified frequency threshold. This way, the rate of incoming messages can be controlled via configuration. This is beneficial for the programmer as they do not need to accommodate for this at code level, but let the middleware take care of it.

QoS policies can be set for each Topic, Publisher, Subscriber, Data Writer and Data Reader individually. In Table 4.1, an excerpt of the available QoS policies is given. For the exhaustive list of all available QoS policies refer to the official standard [37].

Another example of a QoS policy is the `DEADLINE` policy. It specifies the minimum sampling frequency of a Data Writer. If the deadline period of a hypothetical Data Writer is set to, e.g., 100 ms, then this Data Writer is obligated to send a data sample at least every 100 ms. If it fails to send samples at this rate, the Data Writer and all the respective Topic's readers will be notified about that circumstance and are free to act accordingly.

Name	**Legal values**	**Description**
`RELIABILITY`	`RELIABLE`, `BEST_EFFORT`	Indicates whether a Data Writer may drop samples or whether a Data Reader approves of Data Writers that drop samples.
`TIME_BASED_FILTER`	An integer value denoting time	Specifies a Data Reader's desired data reception rate. Superfluous samples will be discarded.
`LIVELINESS`	An integer value denoting time	Defines the rules to determine whether a particular entity is "alive", e. g. by emitting heart beats.
`DEADLINE`	An integer value denoting time	Establishes a contract between Data Writer and Data Reader to determine the acceptable data rate.
`OWNERSHIP`	`SHARED`, `EXCLUSIVE`	Specifies whether multiple Data Writers may write to a given Topic simultaneously or just the one with the highest `OWNERSHIP_STRENGTH` value.
`OWNERSHIP_STRENGTH`	An integer value denoting relative priority	Determines a Data Writer's priority in cases where its Topic's `OWNERSHIP` is set to `EXCLUSIVE`.

Table 4.1: An excerpt of QoS policies

In addition to specifying quality attributes, QoS policies may also serve as "service contracts" between interacting DDS components. These contracts specify non-functional requirements that the involved components must fulfill to be able to communicate with each other. For example, a Data Writer's `RELIABILITY` policy may have been set to the `BEST_EFFORT` level, thereby allowing the writer to drop samples. A Data Reader, on the other hand, may require the writer's policy to be set to `RELIABLE`, which prohibits the dropping of samples. Since the writer does not fulfill the reader's QoS requirements, the two components are considered incompatible, and thus, they cannot interact with each other.

Despite their name, QoS policies do not only concern *quality* attributes per se. They can also be used to specify the priority of data samples, their lifespan, i. e., how long they are valid, or how many data samples of a certain type are kept in local memory.

4.1.6 Redundancy and Automatic Failover

DDS has ways to ensure reliable communication—even over unreliable transmission channels. Part of reliability is failure transparency, i. e., the ability to quickly substitute failed components by backup components. The substitution process involves two steps: first, the failure needs to be detected. Second, a backup component needs to be instructed to take over. By means of the three QoS policies `OWNERSHIP`, `OWNERSHIP_STRENGTH` and `LIVELINESS` (cf. Table 4.1), DDS may realize this process.

As the first step, a mechanism needs to determine whether a component has failed. In most cases, it is not possible for a failing component to properly shut down and "sign off", i. e., to notify the rest of the system that it will become unavailable. For this reason, the failure needs to be automatically registered. The `LIVELINESS` policy can be used to achieve this. `LIVELINESS` is the mechanism that determines whether a component is responsive ("alive"), or unresponsive ("dead"). The policy's value can be set to number indicating the maximum time interval between each liveliness signal. If a component fails to show a vital sign during that period, it is considered dead by the rest of the system. Passive components, i. e. those which do not actively emit data, can be instructed to automatically send liveliness signals, or heartbeats, in certain intervals.

After a component has been declared dead it is up to the middleware to elect a substitute component. This is done through the `OWNERSHIP` and `OWNERSHIP_STRENGTH` QoS policies. By assigning a topic the `OWNERSHIP` value "`exclusive`" one can specify that only a single data writer may write to that topic at any given time. Which data writer is given that prerogative is determined by the data writer's `OWNERSHIP_STRENGTH` value. The data writer that possesses the higher value is eligible to write to the topic. In the failover scenario, both, the primary data writer and the backup writer are assigned to the same topic, and both are configured to have exclusive `OWNERSHIP` rights. The former one has a higher `OWNERSHIP_STRENGTH` than the latter one. Based on both writer's `OWNERSHIP_STRENGTH` values, it is decided which one has precedence over the other.

4.2 Docker for Containerization

In this thesis' approach, services and their dependencies shall be encapsulated in their own, self-contained execution environments that may be deployed inside the vehicle and also in the cloud. Containerization is a promising technology to achieve this. For the exemplary implementation, Docker is used as it provides an easy-to-use interface and is available for x86, as well as for ARM-based platforms.

Docker is a holistic containerization platform which includes everything needed to build, run, deploy, and distribute containers. The Docker application is structured in a client-server model (cf. Figure 4.2). The server is a daemon process running at all times on the host machine. Its primary purpose is to manage containers, images, networks and data volumes, and to provide the means to build images and run containers. The Docker server exposes a RESTful HTTP API by which it can be controlled. The client side of the Docker application is a command line interface (CLI) which acts as a user friendly façade for the server's API.[5]

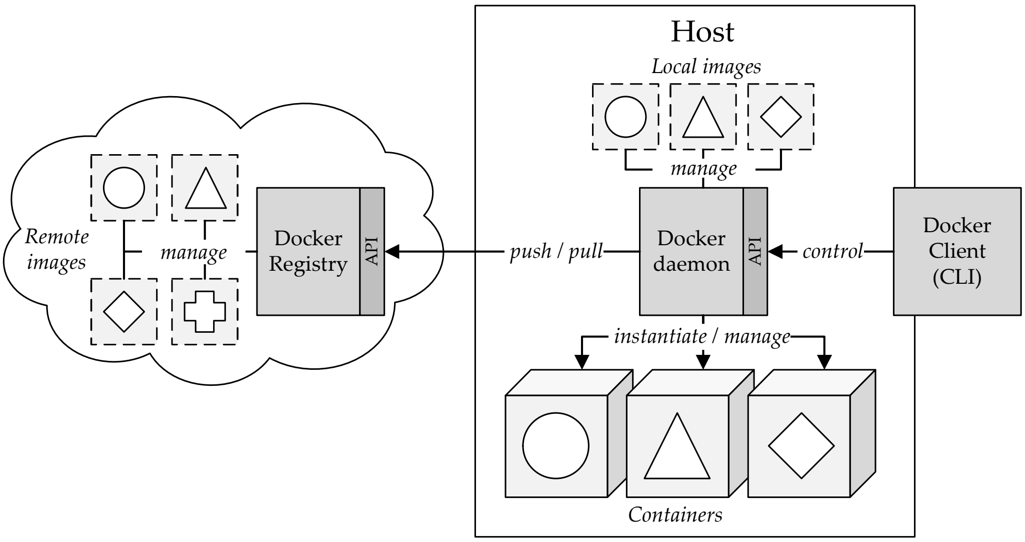

Figure 4.2: Docker architecture

[5] `docs.docker.com/engine/docker-overview`

4.2.1 Docker Components

Docker Images. Images are a central component of Docker. They may be seen as the "blueprint" on the basis of which containers are built and define exactly which files and directories are contained in a container once it comes to life. Effectively, every container is an instantiation of an image of a certain type, in the same way an object is an instantiation of a class in an object-oriented programming language. While the concept of images is very common among containerization and virtualization tools, Docker follows a rather unconventional approach. In Docker, images are made up of series of read-only file system layers stacked on top of each other. Each layer may add files and directories to its respective underlying layer. If a layer tries to modify a file from an underlying layer it creates a copy of that file and adds it to its own layer. The modification is then performed on the copy instead of the original. This principle called "Copy on Write" (CoW) ensures that layers cannot be modified which makes it possible to share individual layers with other images stored on the host. An example of this is later described in Section 4.2.3. By treating images as a collection of individual, sharable pieces instead of atomic units, massive storage space savings can be achieved. Another benefit of this is that whenever a container is started, only a thin writable layer needs to be added on top of an image, and the underlying layers do not need to be copied. That way, start-up times of containers are kept extremely low.[6] Virtual Machines, in contrast, would first need to boot into an operating system prior to operation.

Dockerfiles. Docker images are defined in so-called Dockerfiles.[7] Dockerfiles are made up of series of instructions that tell Docker how to build a certain image. Each instruction adds another file system layer to the image.

```
FROM debian
COPY . /application
RUN cd /application && make
ENTRYPOINT /application/run.sh
```

Listing 4.1: An examplary Dockerfile

An example of a Dockerfile is depicted in Listing 4.1. The first line of this Dockerfile defines the base image on top of which this particular image shall be built. In this case, an image called "debian" was chosen. As the name suggests, that base image contains a basic installation of the Linux distribution Debian.[8] Next, the line `COPY . /application` instructs Docker to copy the contents of the current (host) directory to the container's

[6] docs.docker.com/storage/storagedriver

[7] docs.docker.com/engine/reference/builder

[8] www.debian.org

`/application` directory. Due to Docker's CoW approach, the changes in the file system are added in the form of a layer—the underlying layers are not affected. The `RUN` instruction in the following line causes Docker to run the subsequent commands within the container. In this case, the newly added application directory is entered and the command `make` is executed, effectively compiling the application. Again, the resulting changes to the file system are encapsulated within another layer. In practice, the now topmost layer would contain all files produced by `make`. Finally, the `ENTRYPOINT` instruction defines the action to be performed once the container is started. In this case, the shell script `run.sh` would be executed within the container. This command, too, would add a layer to the image, albeit an empty one. By defining images as sequences of instructions, reproducibility is guaranteed. By running the command `docker build -t IMAGENAME .` in the directory of the Dockerfile, an image based on that Dockerfile would be built. Using the command `docker run IMAGENAME`, a new container of that image type would be launched.

Docker Registries. Docker images may be stored in Docker Registries,[9] where they can be browsed, managed, and distributed. The command `docker push IMAGENAME[:TAG]` causes the image with the name `IMAGENAME` to be uploaded to a registry. Analogously, to download an image from a registry to the local computer, one would have to run the command `docker pull IMAGENAME[:TAG]`. Docker registries are implemented as a server software which exposes an HTTP API (cf. Figure 4.2). The registry server is open source[10] and may be deployed on any server that supports it. The Docker company itself hosts one of such registries, Docker Hub,[11] which is provided as a publicly available and free-to-use service.

4.2.2 Multi-Platform Compatibility

The primary purpose of containerization is to provide portable execution environments that allow software to run on a broad range of computing systems. A limitation of containers is that they only provide portability across *operating systems*, and not *hardware platforms*. In other words, containers do not provide binary compatibility. The reason for this is that containerized applications run directly on the kernel of the host system, and do not build on top of a virtualization layer as classic VMs do. As a result, a containerized binary built for an x86-based processor will not run on an ARM system, and vice versa. This is problematic especially in the context of embedded systems which often rely on particular hardware architectures that favor energy efficiency over

[9] `docs.docker.com/registry`
[10] `www.github.com/docker/distribution`
[11] `hub.docker.com`

performance. Computing nodes in a data centers, on the other hand, are typically based on architectures aimed at providing a maximum level of performance. As the presented approach intends containers to be deployed on both, vehicular on-board systems and in the cloud, this poses a challenge. Two approaches exist to tackle this problem.

Universal, QEMU-enabled Images

The first approach relies on a thin platform compatibility layer that is put into the base of each container. This approach was popularized by resin.io, a company that specializes in containerization for IoT devices. On their Docker Hub page[12] they provide Docker images which have the QEMU[13] machine emulator built in. Facilitated by QEMU's user emulation mode, binaries built for a given processor architecture may be executed on otherwise incompatible processors. QEMU achieves this by translating any guest system calls into host system calls. The previously mentioned images are built in a way that allows any arbitrary binary executed within such container to be run in the context of QEMU. Thus, the container may run on a multitude of hardware platforms. This approach simplifies the software build process tremendously as only one universal container per service needs to be built. This container can then be reused for all platforms.

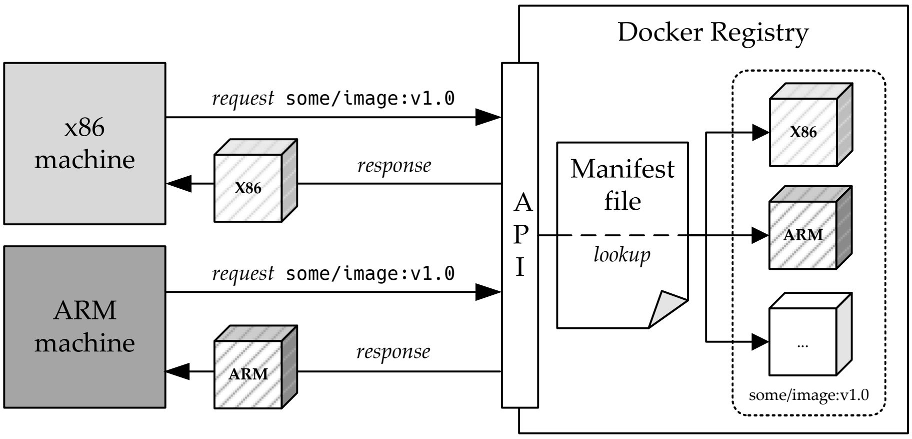

Figure 4.3: Two nodes request the same image name but receive different images which are specifically built for their respective processor architecture. Which node requires which image is determined by a look-up in the manifest file.

[12] hub.docker.com/u/resin

[13] www.qemu.org

Manifest Lookup

The QEMU approach has a significant drawback: multicast is not well supported by QEMU, which is why the idea was discarded. Thus, another approach was leveraged to solve the multi-platform compatibility issue. This approach builds on Continuous Integration (CI) in accord with Docker Hub, and in particular, Docker Hub's *image manifest* feature. The concept of image manifests builds on the following premise: by default, a Docker image name corresponds to exactly *one* image. Thus, whenever an image with a given name is requested from the Docker registry, only that specific image is returned to the requester. Image manifests extend registries by the option to define *multiple* images per image name. That way, several containers, each specifically built for a given platform, may be deployed under the same name. Depending on the requesting node's hardware architecture, a different image may be returned. Figure 4.3 depicts an example for this. In the example, an x86-based machine requests the `some/image:v1.0` image from the registry. What is then returned from the registry is an image specifically made for x86 architectures. On the other hand, an ARM-based machine which requests the very same image, `some/image:v1.0`, receives an entirely different image as a response. This kind of behavior is achieved through an image lookup in the manifest file which is performed behind the scenes.

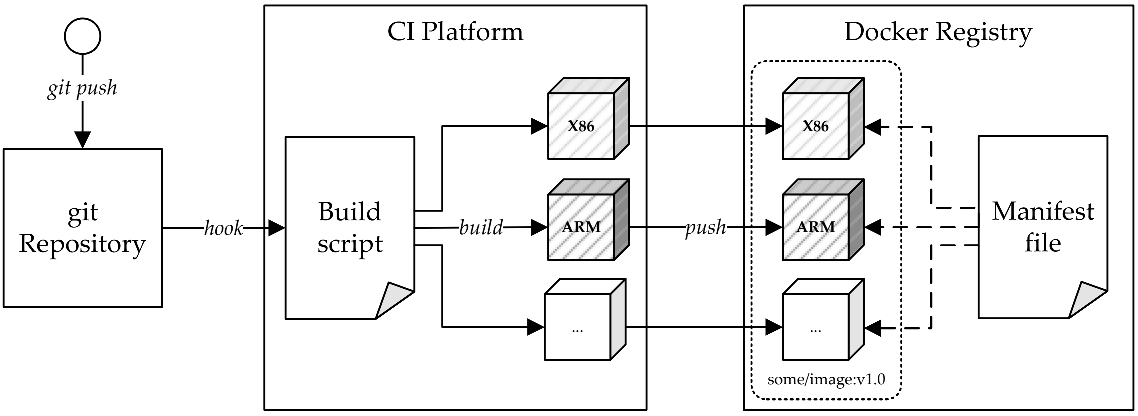

Figure 4.4: A git push to a service's git repository triggers the build process in the CI platform: the code is compiled for several platforms and the resulting images are pushed to the Docker registry.

A disadvantage of this approach (compared to the QEMU one) is that multiple versions of the same container need to be built every time a service receives a software update. To cope with this hindrance, a minimal CI pipeline was leveraged. Consider Figure 4.4 in which the employed service deployment process is depicted. The build process is initiated whenever a code change is pushed to the remote git repository. Once a push is registered, the CI Platform (Travis CI[14] was used) pulls the latest version from the git repository and executes a build script. The build script compiles several version of the service for each target platform, embeds the binaries in images and pushes those images to the Docker registry (Docker Hub). Thus, several images, each tailored to a different platform, may be built and deployed in one go.

4.2.3 Containerized Services

In the envisaged system, each service is packed into its own container with all its dependencies. That way, services may run wherever they are placed, and thus, a great deal of portability is achieved.

Figure 4.5 shows an example of three containerized services running on two different machines, Host I and Host II. On the former, two instances of Service A, and one instance of Service B are placed. On the second host only one service instance is deployed, namely one of type Service C. All services are packaged in their own, separate container. The containers are made up of three stacked images—the bottom two are common to all services. The bottommost image (*"base image"*) contains the file system layers of a minimal operating system. In this thesis, the Linux distribution Debian was used. The base image only contains a limited selection of Linux tools, such as `ls`, `cat`, `ps` etc. The purpose of a base image is to lay a solid foundation that enables users to work comfortably within the container. In the case of Debian, the *aptitude* package manager is additionally included which enables users to easily install further tools.

Next, an *intermediate image* is built on top of the base image. The intermediate image adds further layers containing the services' run-time environment, and in particular, the shared libraries of OpenDDS. This image could optionally contain additional libraries and tools that are common to all services. For the purpose of demonstration, however, DDS on its own is sufficient. The base image and the intermediate image lay the foundation of all services. Any image built on top of these could run any DDS-enabled application. Facilitated by Docker's layered approach to images, the bottom two images can be shared among all services deployed on a host. In the example, the base image and the intermediate image both only need to be stored once on Host I, which saves tremendous amounts of disk space.

At the top, finally, sit the *service images*. In these images contained is the actual service logic, and more precisely, the service's binary and configuration files. The service images are unique to each service, and are not shared, except when the same service is instantiated multiple times on the same machine (cf. Service A in Figure 4.5).

[14] `www.travis-ci.org`

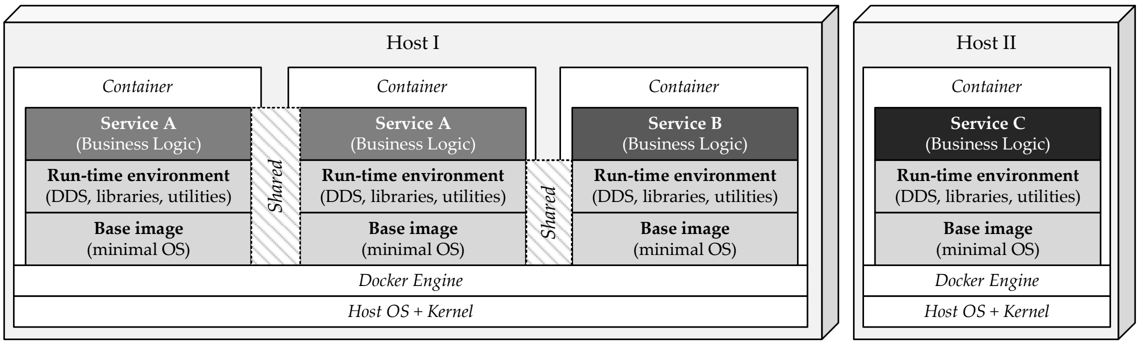

Figure 4.5: An example of four containerized service instances, some sharing common logic to save disk space and facilitate fast updates

4.3 Container Networking with Weave Net

Weave Net is an open source project[15] which is developed by a global team, mostly employed by the London-based software company *Weaveworks*.[16] The purpose of Weave Net is to provide advanced overlay networking capabilities for Docker containers. As such, is designed to make up for the shortcomings of Docker's built-in overlay networks, and more specifically, their lack of encryption and multicast support. By means of Weave overlay networks, dispersed containers deployed on physically isolated hosts around the world, may communicate as if they were connected via LAN. From a containerized applications' viewpoint, it does not matter whether its peers are located on the same host or within a data center on the other side of the world.

4.3.1 Functioning

Weave Net is implemented as client software which is installed on each machine that partakes in the overlay. The software can be started using a single command, and in the following, all containers launched on that host will automatically connected to the overlay network. This is achieved by means of a *Docker API proxy*. The proxy sits between Docker's command-line client and the Docker daemon and intercepts all communication between the two. When the Docker engine is instructed to start a container, the proxy takes all precautions needed to enable overlay networking for that container. Once the connection is established, all container traffic is routed through three dedicated network channels: one TCP connection to exchange meta data about the network, and two UDP channels for duplex data exchange.

[15] `www.github.com/weaveworks/weave`
[16] `www.weave.works`

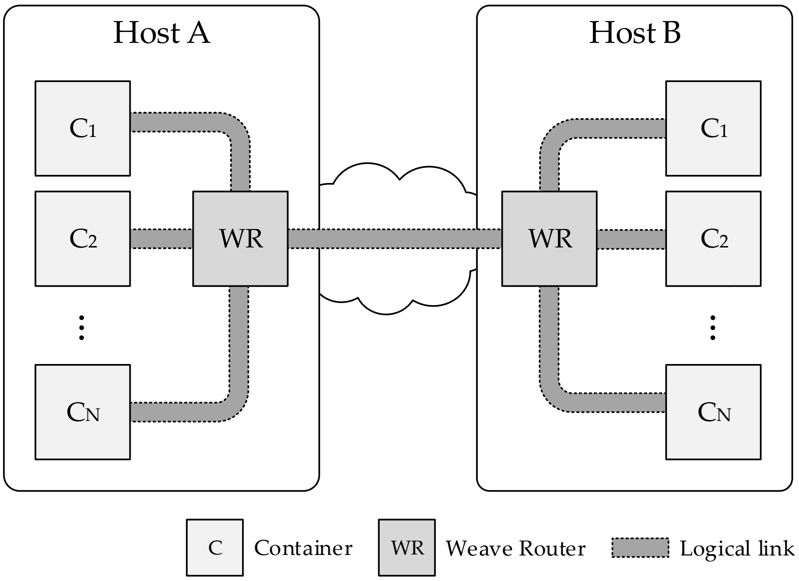

Figure 4.6: A number of dispersed containers connected by a Weave Net overlay network

When the Weave software is started, a central component of Weave Net, the *Weave router* is launched (cf. Figure 4.6). Similar to a hardware router, the Weave router is responsible for the forwarding and routing of data packets to their appropriate receivers. Weave routers can be seen as gateways through which all containers participating in a Weave network are connected. To facilitate routing on the data plane, a custom UDP encapsulation protocol, called *sleeve*, was devised. A Weave router in itself is an containerized application running at all times, in the same way a daemon would. There is one of such router containers running on every host in a Weave-enabled infrastructure.

The Weave router is a user space process. As such, a context switch is needed every time it is tasked to process a packet. This comes with a substantial performance overhead. Hence, as a faster alternative, the so-called *fast datapath* mode was added. In this mode, packets are processed by the Linux kernel instead of by the Weave router. This way, the context switch into user space is omitted. Weave Net leverages the Linux kernel's Open vSwitch datapath module (cf. Section 2.5) to achieve this behavior. Open vSwitch can be used to create a virtual, software-based network switch. That way, the kernel can be instructed to process packets in a certain way. For instance, the kernel can be commanded to add a VXLAN header to each packet, thereby achieving the same result as the Weave router, but faster. However, fast datapath mode can only be used when the underlying infrastructure allows it. The Internet is a particular example of a network where fast datapath communication is hard to achieve.

4.3.2 Topology Management

Weave's topology management is self-governing and self-healing. Peers continually exchange topology information and monitor the state of the network. Whenever peers lose connectivity, they continuously try to re-establish the connection until it is restored. All participating peers know the topology of the entire network. For this, Weave Net employs a sophisticated discovery and topology management mechanism by which changes in the network topology are rapidly propagated within the network. The topology management protocol is based on a spanning-tree broadcast mechanism known from hardware switches. To further ensure that all peers have an up-to-date neighbor list at all times, Weave Net additionally employs a custom neighbor gossiping protocol by which each peer sends update messages to a random subset of their neighbors. Updates of the network's topology are performed periodically as well as when certain events occur, such as when a node joins or leaves the network.

In order to add a new host to the overlay, the Weave software needs to be started on that host, referencing at least one other host in the network by its IP address. The information that a new host was added is then propagated to the other hosts in the overlay. After a short period, all hosts know that the new host was added and that a path to that host exists.

4.3.3 Encryption

As mentioned earlier, a salient feature of Weave Net is the ability to encrypt all traffic within the overlay network.[17] This is especially important for networks that span over insecure underlays, such as the Internet. To set up encryption, a shared secret (password) needs to be provided when the Weave router is launched. The shared secret must therefore be present on all participating hosts. From the password, salted, ephemeral session keys are generated which are then used to encrypt the network packets' payload. For each connection between any two WeaveRouters, one unique session key exists.

The way encryption is applied differs depending on the forwarding mode used (sleeve or fast datapath). In fast datapath mode, a IPsec-based protocol is used, whereby each packet is wrapped in an encapsulated security payload (ESP). Because each packet in this mode is processed by the Linux kernel, encryption is applied by means of the standard Linux Kernel Crypto API which is thoroughly tested, and generally considered secure. For sleeve mode, a custom encryption algorithm based on TLS[18] is used. As with fast datapath encryption, sleeve mode encryption utilizes shared, ephemeral session keys for

[17] Encryption applies end-to-end between Weave routers. For containers hosted on the same machine, communication is not encrypted.

[18] "Transport Layer Security"

each connection.

Figure 4.7 depicts the method used for generating and distributing session keys. In the image, two hosts (H_1, H_2) are to establish an encrypted connection. First, H_1 initiates the key exchange by sending a handshake message to H_2 ①. Then, both hosts generate their own, individual key pairs so that each host has a public key and a private key ②. The key pair for H_1 is (K_{1P}, K_{1S}) and the key pair for H_2 is (K_{2P}, K_{2S}). Once that is done, both hosts exchange their respective public keys, K_{1P}, and K_{2P} ③. Using the peer's public key and their own private key, both hosts derive an auxiliary shared key, S_A, by means of Diffie–Hellman key exchange [40]: $D(K_{1S}, K_{2P})$ ④. Finally, the actual shared key can be generated. For this, both hosts append the password (P) to S_A to provide authenticity. In order to bring the key to the desired length of 256 bit, the compounded key is additionally hashed via SHA256: $H(S_A, P)$ ⑤. The end result of this procedure is the final shared key, S_{12}, which is then used to encrypt the traffic between the two hosts.

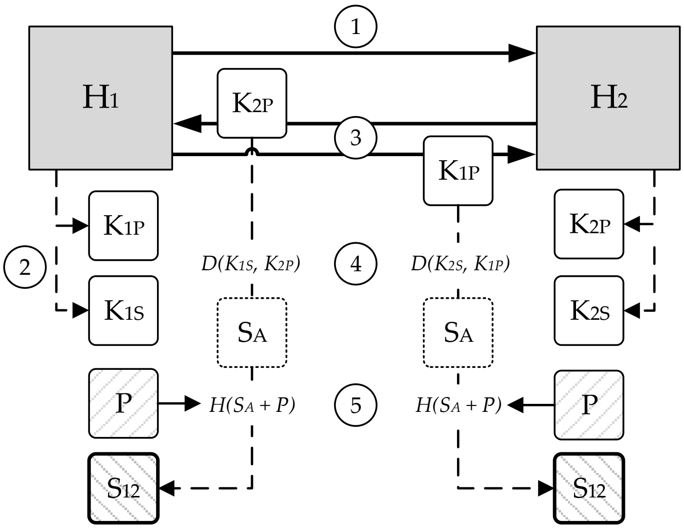

Figure 4.7: Weave Net's key exchange protocol in sleeve mode

5 Evaluation

In the two previous chapters, this thesis' main approach was presented and an exemplary implementation based on DDS, Weave Net, and Docker was suggested. What now follows is an evaluation of the implementation. The primary purpose of the evaluation is to demonstrate the system's feasibility and to quantify selected quality attributes such as performance and reliability. For this, a number of tests were conducted which are presented in the following. But first, the experiment setup is thoroughly described.

5.1 Experimental Setup

As part of the *OSBORNE*[1] project a testbed was created which was made available for this thesis. The testbed aims to simulate a vehicular computing cluster consisting of a number of Linux nodes connected by an Ethernet switch. For this thesis, the testbed was extended by a network link to the Internet to enable cloud connectivity.

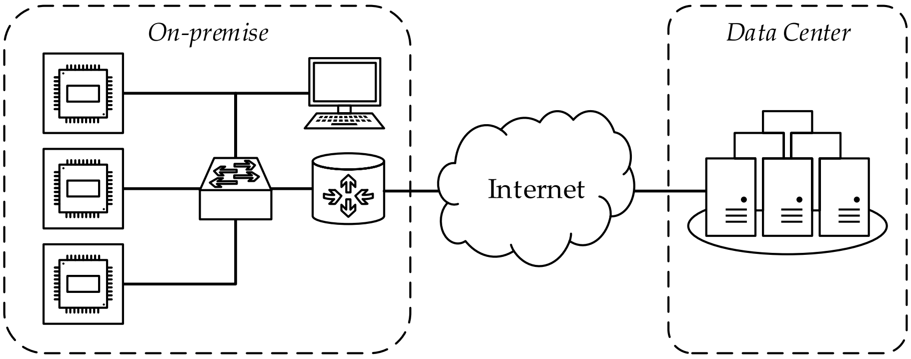

Figure 5.1: The experimental setup's network topology

The test environment's network topology is depicted in Figure 5.1. The original testbed ("on-premise") consists of three ARM-based SBCs[2] (Raspberry Pi[3] 3) whose network interfaces are connected by a Gigabit LAN switch (left-hand side in Figure 5.1). The three

[1] OSBORNE is a BMW-internal project which aims to investigate future E/E architectures.

[2] "Single-Board Computers"

[3] `www.raspberrypi.org`

nodes represent on-board computing devices in a hypothetical vehicle. Furthermore, connected to the switch is also a workstation which fulfills the sole purpose of conducting the tests, i. e., it is not directly involved in the test execution per se. The local test environment is connected to the remote data center through a 100 Mbit/s broadband Internet connection.

Representative of the "cloud" in the tests is a single root server supplied by the cloud provider DigitalOcean[4] (right-hand side in Figure 5.1). To ensure realistic latency values, a server location was chosen that was in the vicinity of the test setup—but not too close. Considering its vicinity to Munich, Frankfurt (Main) was a suitable choice. The two cities are about 300 km away from each other (as the crow flies).

	On-premise	**Remote**
Description	Raspberry Pi 3 Model B	DigitalOcean 1 GB Droplet
Number of nodes	3	1
Theoretical network bandwidth	100 Mbit/s (down), 40 Mbit/s (up)	40 Gbit/s (up- and down)
Operating system	Raspbian GNU/Linux 9.3	Ubuntu 16.04.3 LTS
Kernel	4.9.59-v7+ w/ real-time patch *PREEMPT_RT* 4.4.9-rt17	4.4.0-112-generic
CPU	ARMv7 rev 5 Quad Core (1.2 GHz)	Intel(R) Xeon(R) E5-2650 v4 Single Core (2.20GHz) x86_64
Memory (RAM)	1 GB	1 GB
DDS	OpenDDS 3.12.1	
Docker	18.03.0-ce	
Weave Net	2.2.1	

Table 5.1: Test environment specifications

Throughout the whole study, the software composition remained unchanged. As DDS implementation OpenDDS[5] was chosen. The middleware was configured to utilize dynamic service discovery (without service repository) and RTPS over UDP as wire protocol. The optional Data Local Reconstruction Layer (DLRL), by which messages

[4] www.digitalocean.com

[5] www.opendds.org

are converted into type safe data structures, was used. The overhead caused by this is evaluated and discussed in Section 5.2.4. Generally, default QoS settings were used for the tests, except in cases where a point was made to explicitly deviate from the defaults, e. g., the `LIVELINESS` settings in Section 5.2.5. Moreover, all benchmark programs were developed in the C++ language to maximize their run-time performance.

Naturally, in the benchmarks concerned with the testing of Weave Net, containers were connected by means of a Weave Net overlay network. For the connection between all local nodes (on-premise), fast datapath forwarding mode was used. Due to the characteristics of the network between the local nodes and the cloud, fast datapath was not applicable for that connection. Consequently, local containers and cloud containers were connected via sleeve mode. The detailed specifications of the involved computing nodes and the employed software are given in Table 5.1

5.2 Benchmarks

5.2.1 Latency Benchmark

Motivation. An essential non-functional requirement for computer networks is responsiveness. Especially in real-time systems, where timing requirements must be met, it is vital that information exchange is performed in the fastest, most predictable way possible. A common metric for a network's responsiveness is latency. Latency is usually measured by the time it takes for a packet to be transmitted from one peer to another and back again. The unit of this measure is round-trip time (RTT), and is typically indicated in milliseconds.

To evaluate the aptitude of the presented system it is essential to consider the latency overhead that Weave overlay networks incur. Since the approach is intended to work over the Internet—and public networks are inherently insecure—encryption is vital. Thus, in addition to the overhead induced by the overlay network itself, the overhead caused by encryption is also of interest. Hence, tests are needed measuring latency not only of *plain* overlay networks but also of *encrypted* overlay networks.

Method. In this test, latency was measured between two hosts connected over the Internet, both with, and without overlay-enabled container networking. To sample RTTs, a constant stream of ping messages was sent from one of the Raspberry Pis to the remote host and back. The Linux tool *ping*, which pings hosts via ICMP[6] Echo Requests and Replies, was employed to take the measurements.

[6] "Internet Control Message Protocol"

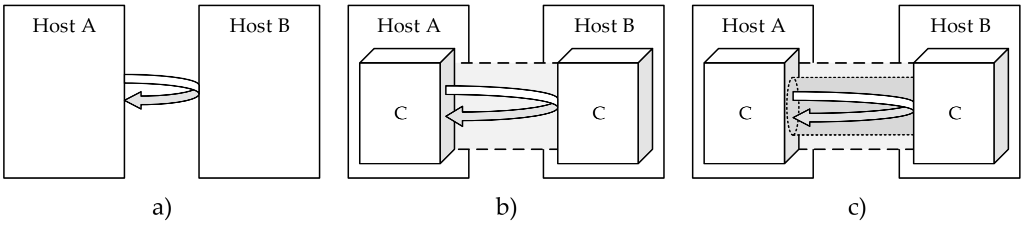

Figure 5.2: Latency experiment investigating three cases. a) PLAIN: two hosts connected without overlay network. b) WEAVE: two containers connected via Weave Net. c) ENCRYPTED: two containers connected via encrypted Weave Net.

Three experiments were conducted, comparing the average latencies of (a) plain host-to-host communication ("PLAIN"), (b) via Weave Net ("WEAVE"), and when employing (c) encryption over Weave Net ("ENCRYPTED") (cf. Figure 5.2). The results from PLAIN serve as the baseline for the two consecutive experiments. In each test run 3000 pings, each with a payload of 1 KB were transmitted. The sending peer would wait for the response of the previous message before sending another request. Thus, only one packet was in flight at a time.

Results. Figure 5.3 depicts the results of the benchmark. The Y-values of the chart represent the average round trip time of the 3000 pings. Note that the Y-axis is cropped to the interval between 28 and 29 ms to emphasize the differences in the bars' heights. The results reveal that the latency overhead incurred by Weave Net is very minor. In the case of plain host-to-host communication (PLAIN), round trips took on average 28.4 ms. When sending the pings over a Weave overlay (WEAVE), RTTs increased to an average of 28.6 ms, which is equivalent to an 0.9% increase. Lastly, in the case of encrypted overlay networking (ENCRYPTED), the average RTTs reached a maximum of 29.0 ms—an increase of 2.1% compared to PLAIN.

5.2.2 Throughput Benchmark

Motivation. In modern vehicles, unprecedented amounts of data, originating from cameras, lidars and other sensors, is accumulated and distributed to the vehicle's various computing nodes. The presented approach allows scenarios in which that data is processed in the cloud. This prospect raises the question how well the approach can deal with such high volumes of data. In particular, the maximum theoretically achievable throughput is of primary interest. Throughput is a measure for how much data per time can be transmitted over a network. Since little data is available on Weave Net's throughput performance, according benchmarks were carried out.

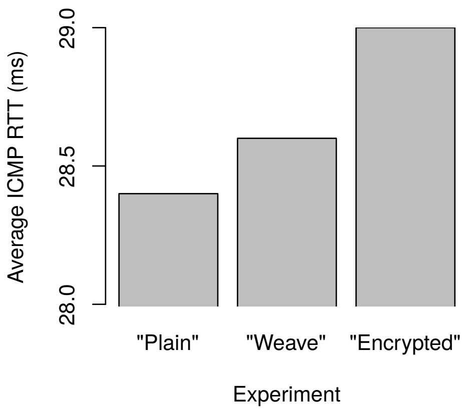

Figure 5.3: Average round trip times in the scenarios Plain, Weave, and Encrypted

Method. To measure throughput, the tool *iperf3*[7] was used. iperf continuously sends data from one node to another over a TCP/IP connection. From the volume of the transmitted data and the duration of the transmission, the effective network throughput can be calculated. In each test run iperf would continuously send as much data as possible within a timeframe of 60 seconds. As in the previous test, three scenarios were tested: Plain, Weave, and Encryped. Explanations of these scenarios can be found in the description of the previous test.

A fact to consider is that routing packets over overlays is computationally expensive—even to the extent that at a certain point the CPU may become the bottleneck of the operation. In other words, network throughput may hit a boundary incurred by insufficient computing resources, even though the networking capacity is not entirely exhausted. This experiment investigates the case where enough computing resources are available, such that the network channel's capacity is the limiting factor. The case where the CPU's performance is insufficient is investigated in the experiment after this one (cf. Section 5.2.3).

Results. The results of this benchmark are depicted in Figure 5.4. The results show that unencrypted Weave Net (Weave) slightly reduces throughput performance when compared to Plain scenario (5% reduction). When furthermore encryption is applied to the overlay network (Encrypted), an additional throughput impairment can be observed. Overall, encryption incurs an overhead of 6.7% compared to the Plain scenario. These values can be considered decent. Once more, it shall be noted that in this case enough computational resources were available. I. e., the CPU was not fully utilized but the network channel was at its limits.

[7] `www.iperf.fr`

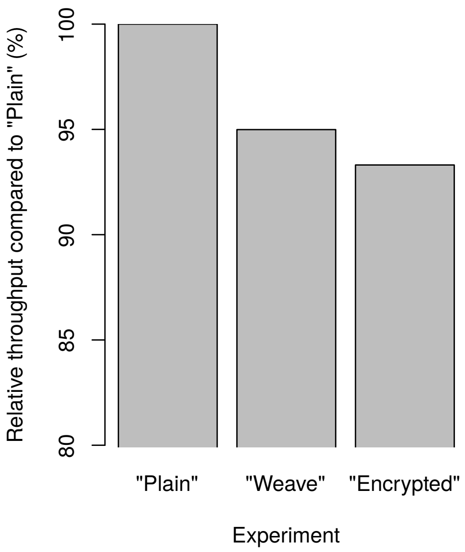

Figure 5.4: Relative throughput in the scenarios Plain, Weave, and Encrypted

5.2.3 Resource Utilization Test

Motivation. The routing of data packets through an overlay puts significant load on the host's CPU. Under circumstances, the transmission of data, which is typically an I/O-bound operation, might become CPU-bound. In this situation the CPU becomes the bottleneck of the operation. In embedded systems, in which computing resources are limited, this might be problematic. Hence, an experiment was designed aimed to investigate the impact of overlays on CPU utilization.

Method. In this experiment, sample data was sent from a Raspberry Pi to the remote host via iperf (cf. Section 5.2.2) over an encrypted Weave Net overlay. During the time of data transmission, CPU utilization was measured using the Linux tool *sar*. 25 test runs were conducted using different bandwidth settings: bandwidth was gradually increased in 1 Mbit/s increments, starting at 1 Mbit/s, and ending in 25 Mbit/s. This bandwidth will be called *target bandwidth* in the following. The target bandwidth is an artificially imposed upper bound and may differ quite substantially from the bandwidth that is actually achieved. The achieved bandwidth will be called *observed bandwidth*, or *throughput*.

Results. Figure 5.5 depicts the test results. In the diagram, each data sample represents the average CPU utilization of the sending node during a particular test run. The Y-axis represents the overall CPU utilization of the sending machine (Raspberry Pi), and includes the cumulative load of all programs running on the system. Hence, the measurements include a bit of noise. The dashed horizontal line indicates the maximum load of a single core in a quad core CPU.

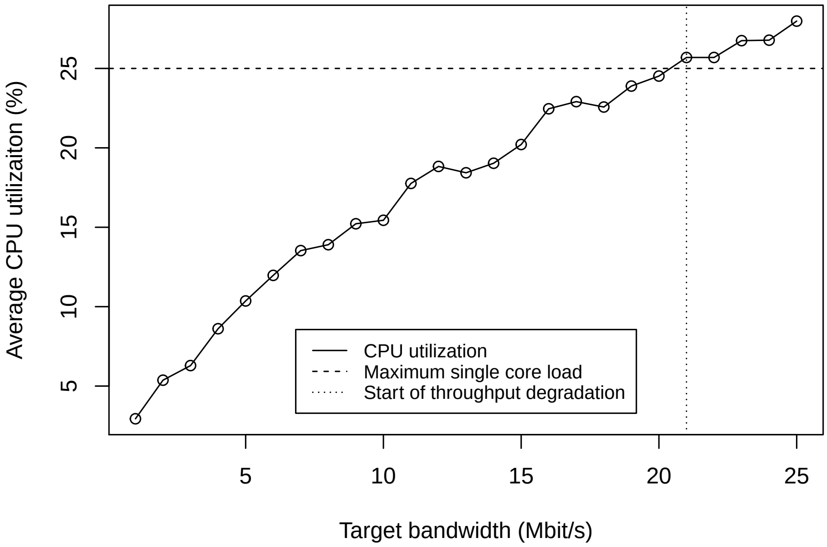

Figure 5.5: In Weave overlays, CPU utilization increases linearly with network throughput. At around 21 Mbit/s, the test node reaches a computational limit (one of the four cores is fully utilized) such that the effective throughput stagnates (not displayed).

The results show that CPU utilization increases linearly with the target bandwidth. What is striking is that the point of full resource utilization of a single core is reached quite early, at around 21 Mbit/s target bandwidth (dotted vertical line). Weave Net is a single-threaded program, and as such, experiences performance stagnation once the core it runs on is fully occupied. At this point, first throughput degradations become evident, so that the setup begins to fall short on delivering the desired bandwidth. The maximum observed bandwidth that the setup could achieve over numerous test runs was 23.6 Mbit/s. However, the maximum (theoretically) achievable throughput in the test setup was 100 Mbit/s. Thus, a large part of the network's capacity was left unused. This circumstance is a strong evidence that a CPU-bound wall was hit.

5.2.4 DDS Latency Benchmark

Motivation. So far, all experiments were concerned with Weave Net exclusively and did not involve DDS. To the best of the author's knowledge, there are no empirical studies yet on how well DDS performs in overlay networks spanning over the Internet. Hence, benchmarks were conducted to evaluate latency overhead incurred by DDS in such scenarios.

Method. The method of this experiment is similar to the previous latency experiment (section 5.2.1) in that the RTT between two hosts connected over the Internet are used as the measure of latency. Two cases are investigated: (i) RTTs via ICMP ping, and (ii) RTTs via DDS ping. For the former case, the ICMP ping results from the previous latency experiment are used. These results serve as a reference point to which DDS is compared to. In order to measure DDS RTTs, a ping application was developed that functions similar to classic ping: a message containing a recent timestamp (T_1) is sent from one endpoint (A) to another (B) via a topic named "ping", and the same message is then sent back to A without modifications. The returned message is published on a topic named "pong", which A is subscribed to. Upon reception of the returned message, A creates a reception timestamp (T_2) and subtracts that timestamp from T_1, effectively calculating the RTT. This process is then repeated multiple times. Between the reception of a returned message and the dispatch of the next message, the sender was configured to wait for a few milliseconds to make sure that only a single message was in circulation at any time.

Of course, sending data via DDS entails a lot more than simple ICMP pings, like e. g. the marshaling and parsing of messages. The comparison with ICMP is therefore a very ambitious one. As ICMP pings require very little processing, the largest part of the measured round trip time accounts for pure message transmission. ICMP ping is therefore a good reference point to measure the overhead induced by DDS, and in particular, RTPS.

The tests in both cases (ICMP an DDS) were performed in an encrypted Weave Net overlay network ("ENCRYPTED") to ensure practical experiment conditions. Both times, ping messages carried a payload of 1 kB. The results presented below are calculated from 300 pings.

Results. The results of the benchmark are presented in Table 5.2. The overhead incurred by DDS, when compared to ICMP ping, is in the 15-to-18 per cent range. The overhead can be explained by the fact that DDS does a lot more than just transmit data from point to point. Most notably, the enforcement of certain QoS policies takes their toll on latency, as well as the marshaling, serialization and parsing of message payloads by the DLRL. Considering this, the overhead of DDS can be considered reasonable.

	ICMP	DDS	DDS Overhead
Minimum	24.32 ms	28.05 ms	+15.36 %
Average	27.00 ms	30.84 ms	+14.27 %
Maximum	47.36 ms	55.96 ms	+18.16 %
σ	1.75 ms	2.39 ms	

Table 5.2: DDS latency benchmark results: ICMP and DDS round-trip-times

5.2.5 Failover Test

Motivation. The presented approach allows for many innovative usage scenarios. For instance, consider the following situation: A service running on an on-board computer within a vehicle suddenly fails, e. g. due to a hardware defect. As a reaction, a fallback service needs to take over. Consider now that the fallback service is not running on another on-board computer but within the cloud. Through DDS QoS policies, a failover mechanism can be realized (cf. Section 4.1.6) which allows for a quick transition to the remote backup service. An interesting question is how long the whole failover process, i. e. switching to a backup service in case of failure, takes in a cloud scenario.

Method. To test the performance of DDS's failover qualities an experiment was designed to replicate a scenario in which a service fails so that another, remote service must take over. In the test scenario, there are three services: two services which continuously produce data (S_{A}, $S_{\mathrm{A^*}}$) in the interval t_i, and one which consumes the data (S_{B}). S_{A} has precedence over $S_{\mathrm{A^*}}$ (i. e. it possesses a higher `OWNERSHIP` value) and is therefore the "main supplier", while $S_{\mathrm{A^*}}$ is considered the "backup supplier". S_{A} and S_{B} are both deployed on two separate Raspberry Pis (representative of two ECUs in a vehicle), and $S_{\mathrm{A^*}}$ is running in the cloud. A schematic of the experiment is depicted in Figure 5.6. It shall be noted that, contrary to what the figure suggests, communication is not point-to-point, but topic-based. S_{A} as well as $S_{\mathrm{A^*}}$ publish data samples on the same topic and S_{B} is subscribed to that topic.

In the process of sending data, a failure of S_{A} is simulated by abruptly shutting down the application process (indicated by the cross in the image). In this event, the cloud-based backup service, $S_{\mathrm{A^*}}$, needs to take over. To determine the effective failover time, t_{fo}, the time between the last message from S_{A} and the first message from $S_{\mathrm{A^*}}$ is measured. The failover is performed implicitly by DDS through the failover mechanism described in Section 4.1.6. In this experiment, `LIVELINESS` is set to 100 ms, indicating that after 100 ms

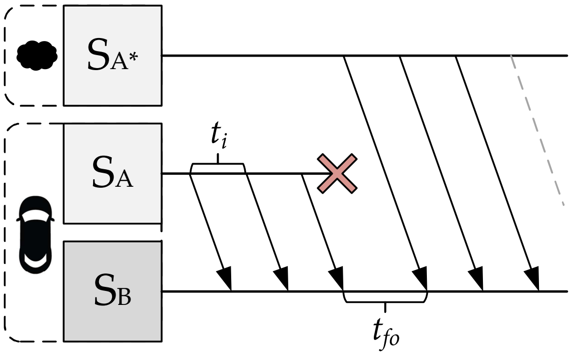

Figure 5.6: Failover scenario: vehicular service S_A fails to supply S_B with data so that cloud-based service S_{A^*} must take over

of inactivity, a service is declared "dead". The experiment is performed on an encrypted Weave overlay (case "ENCRYPTED").

Results. In 100 test runs, an average t_{fo} of 131.3 ms was measured. The fastest failover took 103.2 ms, and the slowest one took 181 ms. Of course, the `LIVELINESS` settings heavily influence the failover times as `LIVELINESS` determines how fast a failure is detected. After subtracting the 100 ms needed for failure detection, the resulting figures may be considered reasonable.

6 Discussion

In Chapter 3, the thesis' main approach was introduced. As part of this, an exemplary technology stack to realize the system was proposed (cf. Chapter 4). The exemplary solution, which is based on DDS, Docker, and Weave Net, was then assessed with the aid of a number of benchmarks (cf. Chapter 5). In this chapter, the proposed solution is discussed, and its merits as well as its flaws are pointed out. The list of requirements given in Section 3.3 and the benchmark results from the previous chapter serve as the foundation of the discussion.

6.1 Requirement Analysis

Availability/Reliability. Moving vehicles are are very likely to lose reception during operation, e. g., when navigating through remote areas. At the center of the presented approach is cloud connectivity, and hence, the reliability of the network is a major concern. Thus, a primary objective when designing the system was to make it as resilient to connection losses as possible. A prerequisite for handling reliability issues is fast failure detection and a quick fail-over mechanism so that timing requirements can be met and downtime is kept at a minimum. With its support for redundancy and failure detection, DDS is perfectly equipped for such cases. DDS' way of guaranteeing reliable information exchange is achieved by QoS policies. For failure detection, DDS employs a liveliness mechanism whereby services are either proactively, or reactively, probed for responsiveness. If a service fails to meet the demand, it is declared "dead". In such cases, fallback services may be elected as (temporary) replacement. As soon as the original service's operability restored, it may take over again. Thus, DDS provides a level of reliability that allows the system to handle (intermittent) connection losses and other failures quickly.

While Weave Net doesn't quite reach the level of replication- and failure transparency provided by DDS, it has other ways to deal with unreliable underlay networks. When a link between two nodes is interrupted Weave takes no proactive measures to find an alternative link but it reconnects as soon as connectivity is restored. Thus, a moderate resilience to connection loss is discernible.

Security. Security was a major concern when designing the approach. In fact, Weave's security features were one of the primary reasons why it was chosen as overlay networking technology. In particular, its support for end-to-end encryption ensures the confidentiality and integrity of the communication channel, even in insecure networks such as the Internet. Weave Net's security is based on IPSec and state-of-the-art encryption algorithms. However, questions remain about the usability of Weave's encryption mechanism. Weave employs a shared secret approach, i. e., a password needs to be present on all participating hosts. How this secret is to be securely stored and how situations of leakage are dealt with are an open question yet to be answered.

Aside from encryption, isolation is an important means to improve the security of a system. By integrating isolation measures, security perimeters can be created which are difficult to penetrate. The more layers of isolation there are the better the system can be protected from attackers. The approach leverages Docker to provide isolation. Each service instance is running within the perimeters of a container. However, whether Docker's isolation properties are sufficient in terms of security is debatable [27]. The fact of the matter is, however, that any degree of isolation is better than no isolation. Hence, for now, the security of Docker is considered sufficient, although a rigor security analysis is still to be done.

Another aspect that determines a system's level of security is how well it can be updated. Modern systems, and especially those building on high-level OSes, need to be supplied with security updates on a regular basis. Docker's layered images greatly facilitate software updates as single layers may be swapped out individually. Whenever a new update is to be rolled out only the layer that changed needs to be replaced, and only that one, single layer needs to be downloaded from the server. Thus, Docker greatly eases the software update process.

Interoperability. The approach is based on containerization as means to achieve a maximum of interoperability. In fact, interoperability is one of its main selling points. Docker is used as containerization technology, which allows software to run on any platform that possesses a container engine and has a kernel—properties that is not hard to come by. Docker currently supports x86 and ARM. The latter is important to guarantee binary compatibility in embedded systems, which are often based on ARM architectures. Additionally, in recent years, efforts to standardize and unify container technologies were launched. Driven by the *Open Container Initiative*[1] (OCI), standards aimed at providing interoperability between containerization technologies were created. Thus, interoperability, at least in terms of containerization, is not impaired.

Weave is, in its entirety, centered around Docker, and even the Weave software itself

[1] www.opencontainers.org

runs exclusively within Docker containers. As a consequence, and unsurprisingly, it may run on any platform which is capable of running Docker containers. Thus, hardware interoperability is given. However, due to its dependency to Docker, a lock-in situation is evident. Consequently, in order to swap Docker with a competing containerization tool, one would have to drop Weave Net from the approach. Similarly, when deciding to go without any sort of containerization, Weave could not be used anymore.

Apart from this, the chosen messaging middleware, DDS, offers decent interoperability support. The approach suggests to define service contracts by means of topics, and in particular, the combination of their associated name, message data type, and QoS policies. The type is specified in a language-independent format (IDL), and is not bound to a specific technology. Thus, platform-independent service contracts may be realized. Beyond that, DDS also supports platform-neutral communication with its wire protocol, RTPS. RTPS ensures that applications using different DDS implementations may communicate with each other, thereby ensuring a great deal of interoperability and preventing vendor lock-in.

Performance. One of the main motivations behind the approach is to support the resource-constrained on-board computing system of vehicles by offloading computations to the cloud, and thereby improve the performance of the whole system. The benchmarks in Chapter 5 demonstrate that the approach manages to fulfill the performance requirement to a satisfying degree. This is because the employed technologies were chosen, in part, on the basis of their performance properties.

DDS exhibits great performance characteristics because it is, from the ground up, designed to perform well. The fact that DDS employs multithreading-based asynchronous programming, and that virtually all DDS implementations are implemented in C++, underline this statement. In Section 5.2.4, DDS was evaluated regarding its latency characteristics in an overlay setting. The results of the benchmark show that DDS incurs reasonable protocol overhead. In that benchmark, DDS was compared to ICMP—a protocol which is not designed for information exchange. The comparison is therefore imbalanced to the detriment of DDS. The fact that DDS still performed comparatively well is a strong argument for its aptitude.

Similar to DDS, Docker was built with performance in mind. Many benchmarks have been conducted which support this statement (e. g. [24]–[27]). At the same time, Docker is very efficient in terms of energy consumption [28]. Virtually all performance analyses agree that Docker incurs minimal computational and I/O overhead, provided that it is configured properly. One reason why Docker performs well is that programs running within containers are actual processes which run directly on the host system's kernel. Thus, no computational overhead is incurred. This is in contrast to VM-based virtualization in which programs run on an intermediate hypervisor layer which affects performance negatively. Furthermore, containers do not have to boot into an operating system, as VMs do. As a result, containers may be launched in a matter of milliseconds.

A slightly less optimistic picture paint the benchmark results for the evaluation of Weave Net. The results reveal that the overlay technology adds a barely noticeable latency overhead, even when applying encryption (cf. Section 5.2.1). Likewise, network throughput takes only a minor hit (cf. Section 5.2.2). However, these properties only apply when enough computing resources are present. In cases where computing power is insufficient, severe network performance degradations may be the result. The reason for this is that overlay networking, i. e. the wrapping and unwrapping of encapsulated data, puts significant load on the CPU. In Section 5.2.3, a case was explored in which the test nodes reached their computational limit such that the CPU became the bottleneck. As a result, only a fraction of the theoretically achievable network bandwidth could be utilized. Expressed in figures, the effective throughput stalled at around 23 Mbit/s. The severe computational overhead can be explained by Weave's custom encapsulation protocol ("sleeve"), which is automatically used for overlays that span over the Internet. In this mode, packets need to traverse the Weave router. Because the router is a user space process, many context switches need to be performed during continuous data transmissions. This circumstance causes significant delays if many packets need to be sent in rapid succession. All in all, one can conclude that Weave Net is definitely not the best suited technology for the use case at hand.

Extensibility/Scalability. The extensibility of the approach is to a large extent facilitated by DDS, and in particular, the publish-subscribe paradigm. In the DCPS paradigm, data is provided "as is", i. e., there are no implications or restrictions w. r. t. *how* that data is used. A newly added service is not concerned with the original *purpose*, or intent, of the data, nor can any other service dictate how the data is supposed to be used—a service just takes advantage of the *presence* the data. Hence, data can be reused in ways that weren't even intended when first designing the system. By having no connotations about the data's semantics, new services can be easily added without making changes to a service's interface, or affecting other parts of the system. Extensibility is further improved by DDS's dynamic service discovery. A newly added service can subscribe to any topic at run-time, as long as the topic's name and type are known. Similarly, a service which provides data may register new topics at will and can engage in collaboration instantaneously. At the same time, other services are entirely oblivious to the fact that a service was added. The change of topology is handled exclusively by the middleware, and minimal manual effort is needed to integrate new services.

Weave Net also helps in building extensible systems: software as well as hardware nodes can be easily added to existing overlays. Weave overlays are set up at host-level, i. e., each host participating in an overlay has the Weave daemon running on it. The daemon causes any container that is launched on such host to be automatically added to the overlay. There is no need for per-container configurations or scripts to be executed, which significantly eases the way software nodes may be added to the system. This process is entirely transparent, so that a container management and orchestration system does not even have to know that Weave is in place. Weave's extensibility properties also carry over to the hardware-level. If a new hardware node were to be added to the system, the only thing one would have to do is launch the Weave software on that node, referencing the IP address of at least one other participating node. The other node would then propagate the addition of new node in the whole overlay. Shortly after, every other node in the system would know that a new node was added. This self-governing approach minimizes manual work, and thus, benefits extensibility greatly. Whether Weave suffers from performance degradations with an increase in nodes has not been considered in this work and needs to be investigated in the future.

Another factor that greatly promotes scalability is the fact that the system is entirely decentralized: Weave Net builds on a distributed architecture in the same way that DDS does. As a result of this, no single point of failure is evident which could potentially break with the extension of the system.

To further improve the system's run-time extensibility it must facilitate software updates. How Docker aids in realizing efficient software updates has already been discussed.

Adaptability. A major challenge in automotive systems is run-time adaptability. Due to their dynamic nature, vehicles are exposed to rapidly changing situations which require quick and adequate reactions. An example of a situation that needs to be handled quickly is a hardware failure. The approach utilizes DDS' failover mechanism to handle such situations. An example of this was demonstrated in Section 5.2.5. The same principle can be applied when the vehicle is succumbed to intermittent connectivity: when the vehicle loses reception while using a cloud-based service, the system can quickly switch to a vehicle-based service. Potential for more advanced ways to deal with operational situations is provided by Docker. The Docker daemon exposes an API by which containers can be started and destroyed remotely. One could think of a central management and orchestration component which monitors the state of the system and, with help of the API, can start and stop services, depending on situational demand.

Other aspects of adaptability are the ability to react to changes in requirements by means of software updates, and adaptability in terms of platform independence. Both aspects have already been discussed.

Testability. With the help of containers, software can be made portable so that it can run on a variety of platforms. This principle is taken advantage of by CI/CD platforms. CI/CD pipelines, in general, perform the following steps: first, the application's program code is downloaded from its source code repository to dedicated CI servers whenever changes are committed to the repository. Then, the software is built and all automated tests are run. In this step, containers are often used [41] to guarantee the processes' determinism: building and testing the software within containers ensures that all dependencies are available and that the result of successive builds is always the same. Consequently, the built program runs on the developer's workstation, the CI server, and the production server in the exact same way. When one of the tests fail, the development team is notified instantaneously so that they can fix the error as soon as possible. If all tests pass, the software is automatically deployed to the production environment. This process ensures rapid development velocity, and more importantly, helps to verify the correctness of the software during the whole development cycle. This principle was demonstrated in Section 4.2.2, were a minimal CI/CD pipeline was described. Although the presented pipeline did not include an automated testing step, such step could be easily integrated.

6.2 Limitations

The benchmarks in the previous chapter demonstrated that the approach to cloud connectivity presented in this thesis is fully functional and performs reasonably well. However, the prototypical implementation is far from perfect and comes with a number of limitations that need to be addressed.

As the presented implementation is merely a proof of concept, several aspects were insufficiently evaluated. First and foremost, little attention has been paid to the safety requirements of vehicles. Weave Net is not designed to be used in safety-critical systems. On their website, Weaveworks make no statement about any certification efforts that make the software suitable for the use in safety-critical scenarios. Thus, for the time being, the presented approach may only find applicability for non-safety critical functions, e. g. in the context infotainment systems. However, it may be added that Weave Net is open source, and as such, all underlying technologies and concepts are disclosed. Thus, it is entirely possible to implement a thoroughly verified and tested derivative of Weave Net tailored to safety-critical systems. The same is true for Docker. Docker just so happens to be the most mature containerization technology, which is why it was chosen for the prototype. However, Docker is by no means suitable for safety-critical systems. But like Weave Net , the concepts that underly Docker are based on well-known technologies and it is entirely transparent how Docker works under the hood. Furthermore, it may be noted that the Docker images presented in this thesis are not meant to be used in production. Although there was an effort to minimize the size of the images, in real scenarios, images would be shaved down to the bare minimum, both in terms of size and capabilities. Utilities such as compilers, build tools and generally everything which is not a prerequisite to execute a given service would be removed. The images used in this thesis serve the purpose of demonstration and thus, such measures were not taken to ensure a rapid progression of this work. More time would be needed to optimize the images for production use. Additionally, more measures are required to improve the containers' isolation. For example, the number of system calls that a containerized process can invoke should be reduced significantly through tools like *SELinux*.

Another limitation related to the used technologies is that they are not universally usable on all hardware platforms. For example, Docker requires the substrate system to run a Linux kernel and a container runtime. This requirement is hard to fulfill in many cases, e. g., for the use with smart phones. This circumstance further restricts the system's potential use cases.

An additional aspect which was negligently evaluated is security. Although Weave Net promises to provide end-to-end encryption, it is unknown how effective and robust the employed encryption scheme is. A rigor security analysis is needed to be able to make a well-founded statement about this aspect. Furthermore, questions about the encryption mechanism's usability remain unanswered. To encrypt a Weave overlay, all hardware nodes need to be supplied a shared secret. The question how this secret shall be distributed and stored securely was not answered in this thesis. Another major caveat concerning Weave's usability is its extensive resource utilization. When overlays span over long distances, Weave employs a custom encapsulation protocol which is applied by a user space process. Thus, a context switch is needed for every packet that is sent, making this process highly inefficient. This circumstance was demonstrated in Section 5.2.3, where the effect of overlay networking on CPU utilization was investigated. The results of this experiment show that Weave overlays may introduce CPU-bound bottlenecks for high-throughput applications. As energy efficiency is one of the main motivating factors behind the devised system, one can conclude that Weave Net is unsuitable in automotive use cases.

Overall, the provided implementation is not, in any way, suitable for production use in vehicular environments. This, however, was never the goal of this thesis. Weave Net and Docker were both chosen for the lack of better alternatives. Furthermore, both tools provide easy-to-use interfaces which allowed for quick prototyping, ensuring the rapid progression of this thesis.

7 Related Work

Cloud computing in the context of automotive allows for the realization of disruptive innovations which have great potential to improve driver satisfaction and to open up new fields of business for car manufacturers. For instance, Wollaeger *et al.* propose a framework which allows for the computation of an optimal velocity profile for vehicles in an effort to minimize fuel consumption [42]. Their solution benefits greatly from the vast amounts of computing resources available to clouds as it relies on dynamic programming which is characterized by high resource demands. Furthermore, large amounts of data are accessible in the cloud which may be used to make better-informed decisions, e. g. on the basis of crowd-sourced data pools.

While some approaches exist which explore possible use cases of automotive clouds, very little work can be found that addresses the *architectural* challenges of creating cloud-supported systems specifically for the automotive domain. Thus, much of the work related to this thesis is concerned with other, more researched domains, such as IoT and fog computing. First and foremost, the thoroughly executed and exhaustive survey on fog computing [43] by Nath *et al.* shall be mentioned as highly relevant and recommended reading. In their literature study, the authors present and summarize work pertaining to a variety of subjects related to IoT and embedded computing. An important and much-discussed aspect in this domain is the *management* of the of widely dispersed collections of embedded computing nodes that make up pervasive IoT systems. Similar to the system proposed in this thesis, many approaches exist that attempt to master the complexity inherent to these systems by means of service-oriented computing. Notable examples of service-oriented architecture frameworks tailored specifically to the IoT and CPS are the SOCRADES project [44], the IMC-AESOP project [45] and the ARUM project [46]. Further noteworthy approaches are presented in [47], [48], and [49]. A common denominator among these works is the idea to represent heterogeneous physical resources and processes as software objects that can be managed and controlled programmatically and may interact with each other by means of machine-to-machine interfaces. Of particular interest for this thesis are furthermore service-oriented approaches specifically tailored to *automotive* systems. Contributions in this domain are provided, e. g., by Kugele *et al.* In their paper [34], the authors present the results of an interview study investigating the needs of automotive software architects and conclude that SOA-based approaches to E/E-system modeling bring promising benefits to the automotive development process.

A concrete example of how service oriented modeling can have great benefits for the development of ADAS is presented by Wagner *et al.* [50].

Most of these architectural approaches stay in the local domain and do not consider cloud connectivity. A common method to bridge the gap between local computing clusters and remote data centers is to employ middlewares. Farahzadi *et al.* analyze the problem domain of cloud connectivity for the IoT ("Cloud of Things")[51]. For this purpose, they present a survey in which they analyze a number of different IoT middleware solutions and compile a set of key challenges and requirements for such systems. They furthermore introduce a taxonomy to group types of middlewares and map each of the 20 investigated middlewares to one of the groups. Particularly interesting are middlewares that, similar to this thesis' approach, focus on publish-subscribe-style communication. For instance, Antonić *et al.* present COPUS [52], a cloud-based publish-subscribe middleware for the IoT. In the paper, the authors perform benchmarks to evaluate the middleware with respect to scalability and elasticity and present a case study in which test subjects were equipped with mobile air quality monitoring sensors that would continuously send sensor data to the cloud. COPUS relies on a broker as cloud gateway which must be installed on a mobile device (e. g. a smartphone). Thus, this solution is unsuitable for the automotive domain. Further noteworthy publish-subscribe middlewares for the IoT are HoPP [53] and VIRTUS [54].

Publish-subscribe enables anonymous point-to-multipoint communication which brings universal benefits, such as scalability and extensibility, which are also interesting for the automotive domain. Thus, several projects exist that aim to implement publish-subscribe in vehicular E/E architectures. An example for this is the RACE project [55], an effort to create a holistic platform for next generation cars. An industry-driven project with the same goal is the AUTOSAR Adaptive Platform [56]. Noteworthy in this regard is also SOME/IP [57], an IP-based automotive middleware by which vehicular functions can be modeled in a service-oriented fashion. SOME/IP integrates flawlessly into AUTOSAR and supports publish-subscribe and other communication paradigms.

The exemplary implementation presented in this thesis relies on the DDS middleware for data exchange. Unfortunately, at the time of this writing, only few attempts were made that aim to leverage DDS in the context of vehicles (e. g. [58], [59]). However, some work was done that is concerned with the use of DDS in other safety-critical systems. Pérez *et al.*, for instance, explore DDS for ARINC-653-based avionic systems [60]. The same authors furthermore present a way to combine DDS and hypervisor-based inter-process communication to interconnect a time and space partitioned system [61]. Their focus lies on mixed-criticality applications running atop a hypervisor designed for safety-critical scenarios. The performance tests they conduct show that DDS adds a reasonable overhead to the communication performance but the overhead caused by hypervisor-based partitioning is quite significant, albeit reducible by the use of multicore processors. Significant contributions in this field were also made by Serrano-Torres *et al.*, who investigate the performance impact of virtualization on DDS-based applications ([62], [63]).

DDS in combination with overlay networks is another highly relevant topic for this thesis. Only one contribution was found that investigates this opportunity: Hakiri *et al.* devise a system to interconnect IoT devices via DDS over overlay networks [64]. However, the description of their approach lacks details and no real effort were made to evaluate it. Another relevant work related to overlay networking is presented by Kratzke. In his paper [65], the author investigates the performance impact of using encrypted overlay networks (in particular, Weave Net) for Linux containers on top of hypervisors. This setup—containers atop hypervisors—is very common in cloud scenarios as IaaS providers often provision servers in the form of virtual machines. This is also the case for the test setup described in this thesis. The author concludes that containers add a non-negligible impact on networking performance. In Weave Net-based overlay networks, the performance is further impaired, and another, less-significant impact is observed when employing encryption. This is in contrast to the benchmark results presented in this thesis, which suggest that both technologies incur a very minor overhead.

This thesis' approach relies on containerization as means to encapsulate functionality in isolated, portable units. A challenge in this is to marry containerization with the traditionally resource-constrained embedded systems. In recent years, this prospect has received substantial attention in the research community. For instance, Morabito provides an in-depth analysis on the utilization of container technology for a variety of Single-Board Computers [31]. The author's evaluation includes benchmarks that assess energy consumption, device temperatures, as well as I/O and computing performance and other factors. Further analyses and benchmarks are presented in [66] and [29]. The evaluations reach a consensus that containerization for resource-constrained devices presents a promising prospect. In fact, the industry has already caught on. resin.io[1] is a project dedicated to bringing containers to embedded devices. In the past, resin.io have made several significant contributions to enable container-based Continous Integration for embedded systems, and have also taken initiative to further minimize resource requirements of containerization. To this end, resin.io provide their own light-weight container engine called "balena"[2] and an accompanying host OS that is deliberately crafted to run containers ("resinOS"[3]).

[1] www.resin.io

[2] www.balena.io

[3] www.resinos.io

Finally, after having covered related work concerning the basic building blocks of this thesis' approach (architecture, communication and virtualization), there are some pieces of related work that use similar methods (combining the aforementioned concepts) to achieve similar goals.

One example is presented by C. Berger *et al.* In their paper [67], the authors describe their experiences of employing containerization in the context of self-driving vehicles. In accordance with the microservice architecture pattern, vehicular functionality in their approach is split into a number of fine-grained services deployed on distributed nodes within the vehicle. The services are connected via OpenDaVINCI, a real-time-capable middleware for distributed embedded systems. This approach is very similar to the one proposed in this thesis in that communication is based on multicast-enabled publish-subscribe and that they employ Docker containers. However, their approach lacks cloud connectivity. The authors furthermore struggle to get multicast to work over Docker overlay networks—a problem that was solved in this thesis.

Another approach that resembles the one presented in this thesis is described by Asad Javed. In his master's thesis [30], he explores the possibility of managing IoT nodes via Docker containers and the Kubernetes[4] container orchestration platform. The employed experimental setup is very similar to the one used in this thesis in that a local cluster of Raspberry Pis is connected in a LAN with cloud access. Attached to some of the Raspberry Pis are cameras that capture images which are then sent to the cloud via Apache Kafka. Like in this thesis, communication between the nodes and the cloud is publish-subscribe-based. However, the approach has a serious drawback: Kubernetes and Kafka are technologies designed for sophisticated cloud environments and are by no means suitable for the use within embedded computer networks due to their tremendously high system requirements. Moreover, no serious effort is evident to evaluate the approach in a neutral manner.

Großmann *et al.* describe another approach [68] related to the previous one. But instead of heavy-weight enterprise container management solutions they follow a more light-weight philosophy. A drawback of their approach is a lack of failure resiliency inherent to their centralized architecture in which a master node may become a single point of failure. A follow-up to that work is presented by Celesti *et al.* in which several weak points are addressed [69].

Not only in the research community, but also in the industry, there is interest in ubiquitous publish-subscribe. For instance, PubNub[5] is a commercial service that aims to provide publish-subscribe-style communication for large numbers of devices dispersed on a global scale. Their focus lies on IoT devices and mobile phone applications. Thus,

[4] `www.kubernetes.io`

[5] `www.pubnub.com`

there is a pronounced focus on handling mobility challenges such as frequently changing IP addresses and general reliability issues. PubNub is a holistic, all-in-one solution, i. e., their service includes everything from client APIs to infrastructure provisioning. The fact that PubNub is a closed platform severely inhibits its adaptability and limits its applicability—especially for the automotive use case, which is characterized by highly specific requirements. While it may be true that PubNub is a good starting point to kick off new businesses (as advertised), it is doubtful that it is the right tool for the intended use case.

Another commercial product that follows the same goal, but is focused more on Industrial IoT (IIoT), is offered by ADLINK (formerly PrismTech). ADLINK provides a holistic data sharing platform based on DDS named "Vortex DDS". Aside from their own DDS implementation, OpenSplice, the Vortex platform includes the service called "Vortex Cloud".[6] Like this thesis' approach, Vortex Cloud aims to connect globally dispersed DDS components, and also provides means to deploy DDS-based applications in the cloud. Unfortunately, no information about the inner workings of Vortex Cloud and its aptitude can be found online. As it is a commercial, closed-source product, this option was not further pursued.

Real-Time Innovations (RTI) is another company which provides an DDS implementation called "RTI Connext DDS".[7] A salient feature of Connext is that it supports DDS-based data transmission over WANs natively. However, this "RTI Secure WAN Transport" has several drawbacks. Firstly, it needs to be configured on code-level in all applications that participate in the communication network. This is in contrast to this thesis' approach in which the applications themselves are kept unaware of the fact that they are communicating with peers outside their LAN. This agnostic approach promotes loose coupling, simplifies network management, and ultimately enables applications to migrate freely between computing nodes. Further disadvantages of RTI's approach are that it doesn't support multicast, that it requires a central "rendezvous" server which acts as a participant repository, and that it is not resilient to connection loss.[8]

[6] `www.prismtech.com/vortex/vortex-cloud`

[7] `www.rti.com/products`

[8] `community.rti.com/static/documentation/connext-dds/5.3.0/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/index.htm`

8 Conclusion

To conclude this thesis, a summary is first given in which the context, problem statement, and the chosen approach is recapitulated. Then, ideas for future research opportunities are provided.

8.1 Summary

8.1.1 Context

Current trends in the field of automotive mobility, such as autonomous driving, are the cause of an increased demand for computational power and the need for advanced data collection techniques. Associated with this development is an inherent increase in energy consumption. However, an objective should be to keep the energy usage of vehicles' electronic systems as low as possible. This applies especially in the wake of the ever-increasing importance of electro mobility where every bit of energy is needed to fuel the vehicle. In the face of this dilemma traditional vehicular on-board technologies are reaching their limits, and thus, new ways to tackle tomorrow's mobility challenges need to be investigated. One possibility to overcome the constraints imposed by the limited capabilities of current vehicles' computing infrastructure is offered by *cloud computing*. Cloud computing has been around for many years now and many modern businesses thrive on the innovation opportunities that cloud technologies facilitate. However, comparatively little attention has been paid to the prospect of integrating them into the automotive domain. A reason for this might be the special requirements that apply to this particular industry. The most prominent challenge is the mobility aspect of vehicles which makes it difficult to maintain a reliable communication channel to the outside world. This issue will be greatly alleviated with the upcoming 5G cellular network that is currently being rolled out. Supported by such technologies, cloud computing paves the way for new, innovative functions for tomorrow's vehicles. For example, computationally intensive tasks could be offloaded to high-performance computing systems with the aim of relieving the vehicle's constrained on-board system, and to boost the system's performance. Further potential lies in the simplified collection of telemetry data for real-time health monitoring and remote troubleshooting as well as in the provisioning of backup services for safety-critical functions.

8.1.2 Approach

This thesis was created as a response to the situation laid out in the previous section. The goal of this work was to present a system that connects vehicles to clouds as means to facilitate innovative applications. For this, an approach was chosen that lends many concepts from classic bus systems in which dispersed components communicate with each other anonymously by reading and writing messages to a shared, logical medium. The characteristic that makes the approach special is that this medium is virtually extended into the cloud with the help of network virtualization. As a result of this, everything that is put onto the bus can be read from within the cloud. Similarly, applications in the cloud can push data onto the bus, which can then be consumed by the vehicle's on-board system. The approach employs anonymous multicast communication in order to provide location transparency: in the eyes of the vehicle it does not matter whether a given data sample originated in the vehicle or anywhere else. This greatly facilitates the offloading of vehicular functionality into the cloud. The approach builds on a service-oriented architectural paradigm in which the vehicle's functionality is split into a discrete number of *services*. Each service fulfills a purpose which, in traditional E/E architectures, would correspond to an ECU's functionality.

Three key challenges were identified that need to be tackled in order to realize such system. Firstly, reliable, anonymous information exchange needs to be facilitated such that individual services are able to communicate in a reliable manner, without the need for explicit references to one another. For this purpose, DDS was proposed. DDS, being a messaging middleware designed for mission- and business critical systems with high reliability and real-time requirements, is a perfect match for the automotive use case. The foundation of these properties is laid by *QoS policies*, which serve to control the level of reliability and to establish interface contracts between services. For data dissemination, DDS employs the data-centric publish-subscribe communication paradigm by which data is made available through *topics*. Contrary to RPC-style communication, in which data exchange is based on *instructions*, DDS places *data* at center-stage. Any service that wishes to expose a certain kind of data may *publish* respective data samples on a topic associated with that data. Conversely, any service that is interested in the reception of that sort of data may *subscribe* to that topic. Communication takes place entirely anonymously, i. e., services do not know their peers, and consequently, can not address each other directly. The subsequent dissemination of data is then performed by the DDS middleware "under the hood". This approach to information exchange greatly facilitates scalability and extensibility.

In order to enable services to run on both, the vehicle and the cloud, they need to be portable, which poses the second challenge to be tackled: isolation. The approach intends services to be packed into self-contained execution environments with all their dependencies. This would allow them to run independently from each other, on a variety of platforms. To achieve this, the use of containerization technology, and in particular, Docker, is proposed. Containers provide OS-level isolation through kernel namespaces. Namespaces wrap a set of system resources and present them to the container process as if they were dedicated to it. Each aspect of a container runs in its own namespace and its access is limited to that namespace. Similarly, containers have by default no access to the host's filesystem. Instead, they come with their own "cutout" of a filesystem which is specified in each container's respective image. In that image, all of a containerized application's dependencies and its runtime are contained. Containers thus provide portability across a multitude of operating systems. The only prerequisite is that the host OS runs on a kernel and provides a container runtime. What makes containers especially appealing for the intended use case is that they exhibit excellent performance attributes, that they may be launched in a matter of milliseconds, and that they may be controlled via HTTP API. The latter characteristic makes it possible to programmatically launch and tear down individual containers in a uniform manner, establishing the basis for orchestration tools like Kubernetes and elasticity controllers.

The last remaining challenge to realize the system is to connect the dispersed services with each other in a location transparent manner. As means to achieve this, SDN-assisted overlay networks were suggested. To demonstrate the general feasibility of the approach, Weave Net was selected for this purpose. Weave Net is a tool which aims to enable advanced overlay networking capabilities for Docker containers which are dispersed throughout different cloud environments. Docker itself comes with intrinsic overlay networking out of the box, however, encryption and multicast is not supported, which is a crucial requirement for the intended use case. Weave is implemented as a daemon that runs on all hosts that participate in the overlay. Whenever a container is launched on such host, the daemon automatically adds the container to the overlay. Changes in the topology are constantly propagated to other containers via spanning tree-based broadcast and a gossiping protocol. Weave effectively creates a self-governing service mesh in which nodes communicate through VXLAN tunnels. In this mesh, containers may communicate freely with each other across physical boundaries. From the viewpoint of a container it does not matter whether a communication partner is located on the same host or on a remote server in a data center.

The three technologies described above (DDS, Docker, and Weave Net) were combined in a novel way to realize an exemplary implementation of the envisaged system. Selected aspects of this setup were evaluated in Chapter 5. In that chapter, a number of benchmarks were presented which serve to demonstrate the approaches' general feasibility and to evaluate its performance attributes and fail-over characteristics. The results of the benchmarks prove that the approach, in itself, performs extraordinarily well. Network latency as well as throughput take only a minor hit in Weave-enabled overlays. Similarly, DDS adds a moderate, but justifiable, overhead. However, severe deficiencies were discovered in Weave's overlay implementation as it puts a tremendous load on CPUs. The (admittedly feeble) test nodes hit a CPU-bound wall quite early in the throughput benchmark, effectively prohibiting high-volume data transmissions. Especially in consideration of the goal to reduce energy consumption, this insight leads to believe that Weave Net is not the right tool for the job. Nevertheless, the general feasibility of using VXLAN-based overlays and DDS as means to connect widely dispersed containers was demonstrated successfully.

8.2 Future Work

Although considerable effort was made to quantify the quality attributes of the approach, it is far from being fully evaluated. Future research could investigate many questions which were left unanswered in this thesis. Especially Weave Net, which is a technology rooted in the software industry, and has no bonds to the scientific domain, would greatly benefit from further research efforts. Questions of interest would be, for example: To which extent can Weave Net scale to a greater number of nodes, and does it suffer from performance degradations with an increase in participating entities? How fast can Weave Net adapt to changes in the underlay's topology? Can it deal with nodes getting assigned new IP addresses? It was furthermore stated that nodes in a Weave Net overlay automatically reconnect when they lose connection to the other nodes—how fast can the reconnection be accomplished? To which degree does Weave's sleeve mode impact performance, compared to fast datapath mode? Can Weave be used in elastically scaling PaaS systems or Amazon's EC2? Apart from all that, how does Weave compare to other, competing technologies, such as *flannel*[1] and Docker's native overlays?

These questions assume that it is worth pursuing Weave Net as overlay networking technology. This, however, in itself is an unanswered question. Because, as it stands, Weave Net is the major weak point of the approach. While it is still the only viable solution available, as it supports encryption and IP multicast, many deficiencies relevant to the automotive use case are evident. However, the underlying principles are well known and well understood, and the general feasibility of the these principles was successfully demonstrated in this work. Thus, there are no technical barriers to realizing a system based on the underlying concepts which is better suited for the automotive use case.

[1] www.github.com/coreos/flannel

A weak point in the testing methodology which pervades throughout all benchmarks in Chapter 5 is that the tests were performed under laboratory conditions. The setup was connected to the cloud by means of a stable broadband Internet connection. These conditions obviously don't apply to real life scenarios. Thus, further testing in such scenarios would be required to make a grounded assessment of the approaches' real-world applicability. Optimally, the system would be integrated into a real, moving vehicle with 5G-enabled Internet access. In this context, it would also be interesting to see to which extent the approach would benefit from 5G's rich set of innovative features, such as *network slicing*—a novel way to facilitate advanced network scaling techniques. By means of slicing, the network is separated into a number isolated slices which are then used by different applications. Each slice is highly specific to its application's use case and can enforce corresponding QoS requirements. Intuitively, the approach, and especially systems based on DDS, would benefit greatly from this. However, further tests would be required to investigate this possibility.

Further research potential lies in the continuing exploration of farther-reaching use cases. This work investigated only a small subset of possible applications facilitated by publish-subscribe communication in overlay networks. A major strength of the approach lies in its ubiquity, i. e. it is meant to be applied in systems consisting of many interconnected entities. So far, however, only the "vehicular cloud" use case was investigated. While a vehicle, with all its contained ECUs, is in itself a distributed system, only dozens of components are connected. It would be interesting to see the approach being used in other use cases in which not dozens, but thousands, of entities are present, e. g. in the context of sensor networks and other (industrial) IoT applications.

An aspect that was neglected in this thesis is the management and orchestration of nodes. This thesis only presents the technical foundation to create a system to distribute computational workloads. A possible next step would be to create a central management component which controls the placement of services, similar to container orchestration systems like Kubernetes or Docker Swarm. Using Docker's programmatic API, containers can be created and destroyed remotely, allowing containers to migrate between different computing nodes. The management component could determine the optimal placement of each service and then use Docker's API to change each service's location dynamically, according to demand. In case of a temporary peak in computational load, the management component could launch additional service instances in the cloud, leveraging horizontal scaling mechanisms. A first step in this direction was already taken with a research paper that was composed to investigate this very prospect [11].

List of Figures

List of Tables

List of Code Listings

Bibliography

[1] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*, ACM, 2006, pp. 33–42.

[2] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, *et al.*, "Towards fully autonomous driving: Systems and algorithms," in *Intelligent Vehicles Symposium (IV), 2011 IEEE*, IEEE, 2011, pp. 163–168.

[3] C. Urmson, C. Baker, J. Dolan, P. Rybski, B. Salesky, W. Whittaker, D. Ferguson, and M. Darms, "Autonomous driving in traffic: Boss and the urban challenge," *AI magazine*, vol. 30, no. 2, p. 17, 2009.

[4] F. W. Rauskolb, K. Berger, C. Lipski, M. Magnor, K. Cornelsen, J. Effertz, T. Form, F. Graefe, S. Ohl, W. Schumacher, *et al.*, "Caroline: An autonomously driving vehicle for urban environments," *Journal of Field Robotics*, vol. 25, no. 9, pp. 674–724, 2008.

[5] A. Festag, M. Rehm, and J. Krause, "Studie Mobilität 2025: Koexistenz oder Konvergenz von IKT für Automotive?," 2016.

[6] J. White, S. Clarke, C. Groba, B. Dougherty, C. Thompson, and D. C. Schmidt, "R&D challenges and solutions for mobile cyber-physical applications and supporting Internet services," *Journal of internet services and applications*, vol. 1, no. 1, pp. 45–56, 2010.

[7] P. Mell, T. Grance, *et al.*, "The NIST definition of cloud computing," 2011.

[8] S. Liu, J. Tang, C. Wang, Q. Wang, and J.-L. Gaudiot, "A Unified Cloud Platform for Autonomous Driving," *Computer*, vol. 50, no. 12, pp. 42–49, 2017.

[9] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang, "What will 5G be?" *IEEE Journal on selected areas in communications*, vol. 32, no. 6, pp. 1065–1082, 2014.

[10] S. Kugele, D. Hettler, and J. Peter, "Data-Centric Communication and Containerization for Future Automotive Software Architectures," in *2018 IEEE International Conference on Software Architecture, ICSA 2018, Seattle, USA, 2018*, IEEE, 2018, (to appear).

[11] S. Kugele, D. Hettler, and S. Shafaei, "Elastic Service Provision for Intelligent Vehicle Functions," 2018, to appear.

[12] M. Broy, M. V. Cengarle, and E. Geisberger, "Cyber-physical systems: imminent challenges," in *Monterey workshop*, Springer, 2012, pp. 1–28.

[13] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[14] G. Leen and D. Heffernan, "Expanding automotive electronic systems," *Computer*, vol. 35, no. 1, pp. 88–93, 2002.

[15] M. Van Steen and A. S. Tanenbaum, *Distributed Systems, Principles and Paradigms*, Third edition. Maarten van Steen, 2017.

[16] P. A. Bernstein, "Middleware: a model for distributed system services," *Communications of the ACM*, vol. 39, no. 2, pp. 86–98, 1996.

[17] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[18] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[19] S. Tarkoma, *Overlay Networks: Toward Information Networking.* CRC Press, 2010.

[20] M. Vaezi and Y. Zhang, "Virtualization and Cloud Computing," in *Cloud Mobile Networks*, Springer, 2017, pp. 11–31.

[21] M. Mahalingam, D. G. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): a framework for overlaying virtualized layer 2 networks over layer 3 networks," RFC Editor, RFC 7348, Aug. 2014, pp. 1–22.

[22] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The Design and Implementation of Open vSwitch.," in *NSDI*, 2015, pp. 117–130.

[23] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, ACM, vol. 41, 2007, pp. 275–287.

[24] T. Adufu, J. Choi, and Y. Kim, "Is container-based technology a winner for high performance scientific applications?" In *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*, IEEE, 2015, pp. 507–510.

[25] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, IEEE, 2015, pp. 171–172.

[26] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, IEEE, 2015, pp. 386–393.

[27] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, IEEE, 2013, pp. 233–240.

[28] R. Morabito, “Power consumption of virtualization technologies: an empirical investigation,” in *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*, IEEE, 2015, pp. 522–527.

[29] P. Bellavista and A. Zanni, “Feasibility of fog computing deployment based on docker containerization over raspberrypi,” in *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ACM, 2017, p. 16.

[30] A. Javed *et al.*, “Container-based IoT sensor node on raspberry Pi and the Kubernetes cluster framework,” 2016.

[31] R. Morabito, “Virtualization on internet of things edge devices with container technologies: a performance evaluation,” *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[32] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs containerization to support paas,” in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, IEEE, 2014, pp. 610–614.

[33] T. Erl, *SOA: Principles of Service Design*. Prentice Hall Upper Saddle River, 2008, vol. 1.

[34] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub, and W. Hopfensitz, “On Service-Orientation for Automotive Software,” in *Software Architecture (ICSA), 2017 IEEE International Conference on*, IEEE, 2017, pp. 193–202.

[35] L. O’Brien, P. Merson, and L. Bass, “Quality attributes for service-oriented architectures,” in *Proceedings of the international Workshop on Systems Development in SOA Environments*, IEEE Computer Society, 2007, p. 3.

[36] I. ISO, “26262: Road vehicles-Functional safety,” *International Standard ISO/FDIS*, vol. 26262, 2011.

[37] "Data Distribution Service (DDS)," Object Management Group (OMG), Standard, Apr. 2015, v1.4.

[38] "DDS Data Local Reconstruction Layer (DDS-DLRL)," Object Management Group (OMG), Standard, Apr. 2015, v1.4.

[39] "The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification," Object Management Group (OMG), Standard, Sep. 2014, v2.2.

[40] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater, "Provably authenticated group Diffie-Hellman key exchange," in *Proceedings of the 8th ACM conference on Computer and Communications Security*, ACM, 2001, pp. 255–264.

[41] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable DevOps," in *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, IEEE Press, 2017, pp. 65–75.

[42] J. Wollaeger, S. A. Kumar, S. Onori, D. Filev, Ü. Özgüner, G. Rizzoni, and S. Di Cairano, "Cloud-computing based velocity profile generation for minimum fuel consumption: A dynamic programming based solution," in *American Control Conference (ACC), 2012*, IEEE, 2012, pp. 2108–2113.

[43] S. B. Nath, H. Gupta, S. Chakraborty, and S. K. Ghosh, "A Survey of Fog Computing and Communication: Current Researches and Future Directions," *ArXiv preprint arXiv:1804.04365*, 2018.

[44] A. Cannata, M. Gerosa, and M. Taisch, "SOCRADES: A framework for developing intelligent systems in manufacturing," in *Industrial Engineering and Engineering Management, 2008. IEEM 2008. IEEE International Conference on*, IEEE, 2008, pp. 1904–1908.

[45] S. Karnouskos, A. W. Colombo, T. Bangemann, K. Manninen, R. Camp, M. Tilly, P. Stluka, F. Jammes, J. Delsing, and J. Eliasson, "A SOA-based architecture for empowering future collaborative cloud-based industrial automation," in *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*, IEEE, 2012, pp. 5766–5772.

[46] C. A. Marín, L. Monch, P. Leitao, P. Vrba, D. Kazanskaia, V. Chepegin, L. Liu, and N. Mehandjiev, "A conceptual architecture based on intelligent services for manufacturing support systems," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, IEEE, 2013, pp. 4749–4754.

[47] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, IEEE, 2016, pp. 1–6.

[48] F. Kart, G. Miao, L. E. Moser, and P. Melliar-Smith, "A distributed e-healthcare system based on the service oriented architecture," in *Services Computing, 2007. SCC 2007. IEEE International Conference on*, IEEE, 2007, pp. 652–659.

[49] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, "Service oriented middleware for the internet of things: a perspective," in *European Conference on a Service-Based Internet*, Springer, 2011, pp. 220–229.

[50] M. Wagner, A. Meroth, and D. Zöbel, "Developing self-adaptive automotive systems," *Design Automation for Embedded Systems*, vol. 18, no. 3-4, pp. 199–221, 2014.

[51] A. Farahzadi, P. Shams, J. Rezazadeh, and R. Farahbakhsh, "Middleware Technologies for Cloud of Things-a survey," *Digital Communications and Networks*, 2017.

[52] A. Antonić, M. Marjanović, K. Pripužić, and I. P. Žarko, "A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things," *Future Generation Computer Systems*, vol. 56, pp. 607–622, 2016.

[53] C. Gündoğan, P. Kietzmann, T. C. Schmidt, and M. Wählisch, "HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things," *ArXiv preprint arXiv:1801.03890*, 2018.

[54] M. Bazzani, D. Conzon, A. Scalera, M. A. Spirito, and C. I. Trainito, "Enabling the IoT paradigm in e-health solutions through the VIRTUS middleware," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, IEEE, 2012, pp. 1954–1959.

[55] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, and A. Knoll, "Race: A centralized platform computer based architecture for automotive applications," in *Electric Vehicle Conference (IEVC), 2013 IEEE International*, IEEE, 2013, pp. 1–6.

[56] S. Fürst and A. Spokesperson, "AUTOSAR adaptive platform for connected and autonomous vehicles," in *Proc. conf., 8th Vector Congress, Alte Stuttgarter Reithalle*, 2016.

[57] L. Völker, "SOME/IP-Die Middleware für Ethernetbasierte Kommunikation," *Hanser automotive networks*, 2013.

[58] R. Bouhouch, H. Jaouani, A. B. Ncira, and S. Hasnaoui, "DDS on top of FlexRay vehicle networks: Scheduling analysis," *International Journal of Computer Science and Artificial Intelligence*, vol. 3, no. 1, p. 10, 2013.

[59] R. Bouhouch, H. WafaNajjar, and S. Hasnaoui, *Implementation of Data Distribution Service Listeners on Top of FlexRay Driver*, 2011.

[60] H. Pérez and J. J. Gutiérrez, "Handling heterogeneous partitioned systems through ARINC-653 and DDS," *Computer Standards & Interfaces*, vol. 50, pp. 258–268, 2017.

[61] H. Pérez and J. J. Gutiérrez, "Enabling Data-Centric Distribution Technology for Partitioned Embedded Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3186–3198, 2016, ISSN: 1045-9219. DOI: `10.1109/TPDS.2016.2531695`.

[62] R. Serrano-Torres, M. García-Valls, and P. Basanta-Val, "Virtualizing DDS middleware: performance challenges and measurements," in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, IEEE, 2013, pp. 71–76.

[63] M. García Valls, P. Basanta Val, and R. Serrano Torres, "Benchmarking communication middleware for cloud computing virtualizers," 2013.

[64] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif, "Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications," *IEEE communications magazine*, vol. 53, no. 9, pp. 48–54, 2015.

[65] N. Kratzke, "About microservices, containers and their underestimated impact on network performance," *ArXiv preprint arXiv:1710.04049*, 2017.

[66] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures–a technology review," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*, IEEE, 2015, pp. 379–386.

[67] C. Berger, B. Nguyen, and O. Benderius, "Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, IEEE, 2017, pp. 7–12.

[68] M. Großmann, A. Eiermann, and M. Renner, "Hypriot cluster lab: an ARM-powered cloud solution utilizing docker," 2016.

[69] A. Celesti, L. Carnevale, A. Galletta, M. Fazio, and M. Villari, "A Watchdog Service Making Container-Based Micro-services Reliable in IoT Clouds," in *Future Internet of Things and Cloud (FiCloud), 2017 IEEE 5th International Conference on*, IEEE, 2017, pp. 372–378.