A Lightweight Virtualisation Platform for Cooperative, Connected and Automated Mobility

Fabian Gand, Ilenia Fronza, Nabil El Ioini, Hamid R. Barzegar, Van Thanh Le and Claus Pahl
Free University of Bozen-Bolzano, Bolzano, Italy

Keywords: Digital Mobility, CCAM, Road Mobility, Edge Cloud, Container, Orchestration, Performance Engineering.

Abstract: Digital mobility systems such as autonomous cars or traffic management build on connectivity and automated cooperation. In order to facilitate various use cases such as vehicle manoeuvres, infotainment support or state share functions, a distributed layered computation and communication infrastructure is needed that connects vehicles and other devices through mobile networks, linking them to edge and cloud services. Of particular relevance are lightweight clustered infrastructures close to the edge of the network that provide nonetheless sufficient compute, storage and networking capabilities. Clusters consisting of single-board devices are used in a variety of these use cases. In most cases, data that is accumulated on the devices has to be sent to remote cloud hubs for processing. However, with the hardware capabilities of these controllers continuously increasing, it is now possible to directly process data on these edge cluster. This concept is known as Edge Computing. We propose an edge computing architecture for cooperative, connected and automated mobility that relies on industry-standard technologies such as the MQTT protocol for communication, Prometheus for monitoring and Docker swarm in conjunction with openFaas for deploying containerized services.

1 INTRODUCTION

Digital mobility systems such as autonomous cars or traffic management build on connectivity and automated cooperation. In order to facilitate use cases such as vehicle manoeuvres, infotainment support and state share applications, a distributed layered computation and communication platform is needed that connects vehicles and other devices through mobile networks, linking them to edge and cloud services.

We propose an architecture for *cooperative, connected and automated mobility (CCAM)* that relies on widely used technologies such as the MQTT protocol for communication, Prometheus for monitoring and Docker swarm in conjunction with openFaas for deploying services across the cluster. We demonstrate the proposed architecture by referring to the functionality of the European Union-supported 5G-CARMEN project that aims at building a 5G-enabled motorway corridor across 3 countries to conduct cross-border trials of 5G technologies (5G-CARMEN, 2019). Three areas that the project is focused on are vehicle manoeuvre negotiation, infotainment and state sharing, which will also be addressed here in the proposed architecture.

The 5G mobile standard aims at enabling communication between devices almost in real time (5G-CARMEN, 2019). In addition to higher transfer rates, new technologies such as Network Function Virtualization are supported allowing the execution of code functions on generic hardware nodes without having to install specific hardware. Software Defined Networking allows third parties to directly use hardware resources by defining the desired set-up at software level.

Clusters at the edge of the communication network often consist of smaller (e.g., single-board) devices and are used in a variety of these use cases. In many configrations, data that is accumulated on the devices has to be sent to remote cloud hubs for processing. However, with the hardware capabilities of these controllers continuously increasing, it is now possible to directly process data on the cluster. This concept is known as Edge Computing. Edge Computing is defined as a concept where most processing tasks are computed directly on hardware nodes at the edge of a network or cluster and are not send to remote processing hubs. The edge technologies we deploy shall be based on the serverless concept. Serverless aims at shifting the responsibility of deploying, scaling and maintaining software to the resource provider. Instead of deploying and running the

Gand, F., Fronza, I., El Ioini, N., Barzegar, H., Le, V. and Pahl, C.
A Lightweight Virtualisation Platform for Cooperative, Connected and Automated Mobility.
DOI: 10.5220/0009379802110220
In *Proceedings of the 6th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS 2020)*, pages 211-220
ISBN: 978-989-758-419-0
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

application on designated hardware themselves, developers simply need to hand it over to the serverless platform (Baldini et al., 2017).

In order to demonstrate the suitability of our proposed lightweight CCAM edge architecture built on principles of microservices, containers, serverless and auto-scaling, we implemented a proof-of-concept version for experimental evaluation.

The paper is structured as follows. In Section 2, a use case and architecture requirements are introduced. In Section 3, related work is discussed. Section 4 first reports on the high-level architecture of the proposed platform before describing each component in detail. Section 5 evaluates the architecture.

2 CCAM USE CASES

We target here a traffic management (TM) system for autonomous cars as a use case for illustration that needs to be deployed on hardware nodes in relatively close proximity to the road sections it is supposed to cover. The users of the system are: (a) Passive User: Passengers of self-driving vehicles that make use of the autonomous driving functionality, using the parts involved in providing autonomous driving and additional functionality for vehicles such as video streaming. (b) Active User: Administrators observing the system status, using a dashboard function to gain an overview over the system status. The system shall be deployable on microcontrollers in the form of single-board devices. The aim is to judge the computational capabilities of constrained devices at the real edge of the network. In our case the application will be deployed and evaluated on Raspberry Pis, but other, comparable devices, could also be supported. The goal of this work is to identify components that are suitable for the demands of a 5G mobility system. The focus needs to be on the architectural and software related issues of such an implementation. Topics related to the technical details of the 5G standard will not be addressed. Please note that we focus on the *outer edge* in terms of performance evaluation. More central and larger so-called *mobile* or *multi-access edge clouds (MECs)* are also needed, but will not be the focus here in the experimental work.

A platform solution must provide a way to scale and configure parts of the application in such a way that changing external factors, such as the numbers of active cars, are taken into consideration and the application is configured and scaled accordingly. Therefore, there needs to be a mechanism to rescale at runtime. Another important aspect was to keep the used storage space on the devices to a minimum.

We will consider a *Cooperative, Connected and Automated Mobility (CCAM)* scenario, specifically manoeuvering (e.g., lane changing) and in-car video streaming of a mobile entertain service, as our motivating scenarios. Cars (and other vehicles) are becoming more and more connected. Even today, many cars communicate status information to manufacturers or other parties, or receive data for the navigation system. Towards autonomous cars, even more communication and coordination can be expected with more data to be processed at higher speed. We illustrate this with the following two use cases:

- Our first use case deals with manoeuvering, for which supported lane changing is a concrete example. Cooperation between vehicles is needed to ensure safe and efficient navigation through in lane changing or overtaking. Vehicle automation can help here. However, recognizing and communicating the intentions of traffic participants with minimal delay is key to an optimized behaviour. To this end, wireless communication systems can be used to exchange speeds, positions and intended trajectories/manoeuvres. The on-board systems can use this information to derive an optimized driving strategy (in case of automated operation) or derive a recommended course of action to actively optimize traffic flow and avoid dangerous situations. Cooperative lane changing can be realized either in localized or centralized way. The former involves direct exchanges between vehicles, while the latter builds upon a MEC/back-end server and a cellular network, which support the vehicles in determining optimal behaviour. Latency requirements require the use of edge cloud technology to be used.
- Our second use case considers the on-demand streaming of movies or live broadcasts into cars, ideally in quality HD format. This application already dominates internet traffic today. More demand can be anticipated with autonomous driving where drivers can then consume for instance videos instead of being occupied with driving. However, in a mobile context, the speed and latency needed for high-quality HD video streaming, irrespective of the location and network conditions, is a challenge.

The overall architecture is shown in Fig. 1, showing key management functions from core cluster management and monitoring at MEC level (e.g., Kubernetes for container clusters) to service orchestration including placement, lifecycle and quality management.

To get a better understanding of the requirements, we use the video streaming use case for a deeper analysis. We review the technical platform and technol-

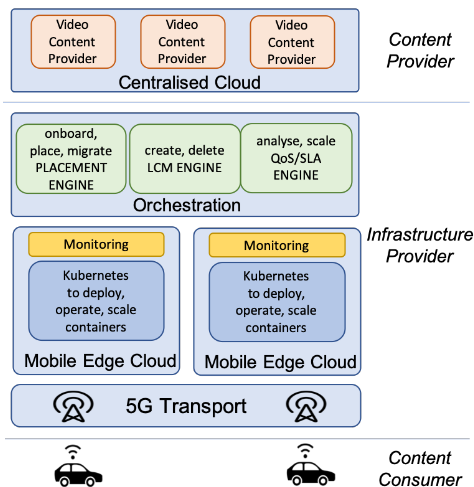

Figure 1: 5G-enabled Edge Architecture – for a Video Streaming Service.

ogy requirements now, as in Figure 1. This application context is characterised by different parties:

- Content consumer – consuming video in a vehicle,
- Content provider – delivers content, using video servers in a cloud back-end,
- Infrastructure/QoS management entity – provides the infrastructure and maintains QoE.

This needs to work in cross-operator and also in cross-border environments. The video streaming application has many sources of input, apart from the actual video content. Many parameters need to be monitored: the traffic on a road with the location/trajectory of cars, but also the resource utilisation on edge and core cloud resources (storage, compute, network), to allow the right sections of video to be provided at the right time into a moving vehicle that connects to different network points. These resources are consumed by the application function (streaming), but are managed by an orchestrator (local mobile edge cloud orchestrators and also central cloud orchestrators) and a platform manager. Telecommunications companies talk about Mobile (or Multi-access) Edge Clouds (MECs) that bring cloud technology closer to the edge of the network and the user.

The use cases of the EU H2020 project 5G-CARMEN show the need to consider layered, 5G-enabled edge architectures. For instance, in order to provide video content into a moving car, information is needed in order to predict the car trajectory and buffer sufficient video content close-by in these MECs to maintain quality. To solve this, we need to employ prediction techniques and pro-active scaling of platform and application services.

Prediction is a critical component in the dynamic processing, both for manoeuvering and video use cases. The prediction of the car movement or the future allocation of storage (video caching needs) to bridge connectivity losses around borders are challenges. Prediction needs to work with a controller, which manages configuration settings for video streaming functions in order to maintain non-functional requirements (NFR) such as quality and performance. An intelligent controller can deal with orchestration and dynamic adaptation needs, for example for the video streaming case to:

- Buffer more (if necessary allocate more storage at suitable MEC). The size of the allocated resource is the control parameter: provide more storage for buffering, if needed due to expected outages/disconnections. The location of allocated resources, i.e., storage at the most suitable MEC/device in cross-border settings depending on predicted trajectory, is another factor.
- Reduce video quality, e.g., reduce video frames per second to deal with latency problems arising from high data volumes being transferred.

Adaptation has an optimization goal. The resource utilisation needs to be as low as possible, while still maintaining NFRs (e.g., required latency or reliability limits to maintain user Quality-of-Experience, with a fixed message size).

The focus of our investigation is on the lower layers – from the edge to 5G infrastructure to roadside devices (assuming small-board devices – SBD) and cars (on-board units – OBU), see Fig. 2.

3 RELATED WORK

This chapter provides an overview over work that has been conducted in the past regarding the elicitation of requirements and the modeling of distributed IoT systems in the context of Edge Computing for CCAM.

(Kiss et al., 2018) gather requirements for applications making use of edge computing. Specifically in the context of combining them with the capabilities of the 5G standard. They mention that recently released single-board devices open up the possibility of processing some of the data at the edge of the cluster.

There exist a variety of different examples of comparable IoT Systems that have been documented. We consider here systems comprised of different hardware nodes that communicate with each other as well as with a central node. (Tata et al., 2017) outline the state of the art as well as the challenges in modeling the application architecture of IoT Edge Applications. One of the scenarios they introduce is a system

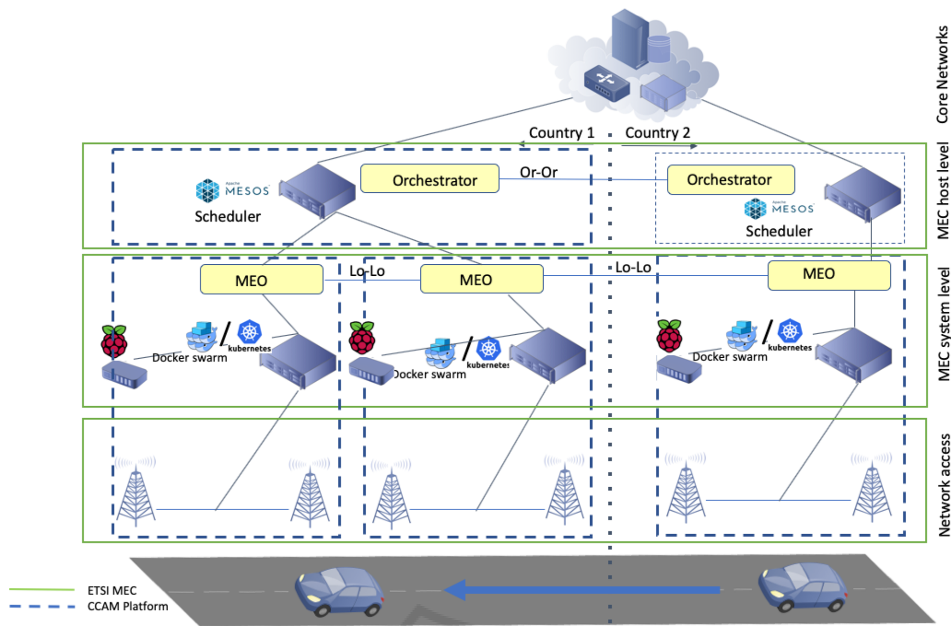

Figure 2: 5G-CARMEN architecture.

for smart trains based on the principles of edge computing. The system is comprised of a set of different sensors attached to crucial components of a car. Data is gathered by a central unit for each car that sends its data to the central processing unit. The task of gathering and processing data is delegated to edge nodes of the network before passing it on to the cloud.

There exist already 5G-oriented platforms for edge processing. An example is the lightMANO platform (Riggio et al., 2018). This presents a MANO standards-compliant implementation for lightweight processing that allows NFV and SDN capabilities to be supported. We build on similar idea, providing here an experimental evaluation specifically considering lightweight device clusters.

4 CCAM ARCHITECTURE

In this section, we introduce important architecture concepts as well as specific tools and technologies for our platform. The final subsection describes how the tools and technologies can be combined to create a complete lightweight CCAM edge platform.

4.1 Architecture Concepts

Cloud Computing is based on data centers that are able to process large amounts of data in the Cloud (Kiss et al., 2018). Data from the local system is usually sent to and processed by the cloud. This approach, however, leaves the potential local processing power of the network unused and comes with significantly increased latency. Edge Computing leverages the processing power of nodes at the edge of the local network. These nodes are an intermediate layer between the devices by processing parts of the data within the local network that would otherwise by handled by an external cloud.

Microservices have become increasingly popular in recent years. Traditional architectures usually package and deploy software as a monolith: The entire application was bundled into one executable that was deployed on specific hardware. When switching to a microservice architecture, the monolith is split into different parts. These parts, the microservices, run in an independent process and have their own deployment artifacts (Jamshidi et al., 2018).

Serverless Computing is a new concept for the deployment of cloud applications (Baldini et al., 2017). The term serverless represents the idea that developers can focus on the application code without being concerned about the servers it is deployed on. The tasks of managing and scaling the application are handled by the resource provider. Usually, serverless computing is accompanied by a concept called Functions-as-a-Service (FaaS). Here, small chunks of functionality are deployed and scaled dynamically by the provider (Kritikos and Skrzypek, 2018). These functions are usually even smaller than microservices. They are short lived and have clear input and output parameters, similarly to functions used in most

programming languages. If the component to be deployed is more complex than a simple function and is supposed to stay active for a longer period of time, a stateless microservice should be considered (Ellis, 2018). Managing and deploying these microservices is similar to serverless functions.

4.2 Tools and Technologies

The concrete hardware, software and standards used in the platform are introduced here.

The *Raspberry Pi* is a single-board computer based on an ARM-processor. The version 2 B models used here include a 900MHz quad-core ARM Cortex-A7 CPU and 1GB of RAM.

Docker is a containerization software. Containerization is a virtualization technology that, instead of virtualizing hardware, separates processes from each other by utilizing features of the Linux kernel . Docker containers bundle an application with all of its dependencies. Docker allows to create, build and ship containers. *Docker swarm* is the cluster management tool integrated into the Docker Engine. Instead of running services and their corresponding containers on one host, they can be deployed on a cluster of nodes that is managed like a single, Docker-based system. By setting the desired number of replicas of a service, scaling can also be handled by the swarm.

Hypriot OS is an operating system based on Debian that is tailored towards using Docker technology on ARM devices such as the Raspberry Pi. The OS comes prepackaged with Docker. *MQTT* is a network protocol designed for the Internet of Things. It is primarily used for unreliable networks with limited bandwidth. MQTT uses a publisher-subscriber approach.

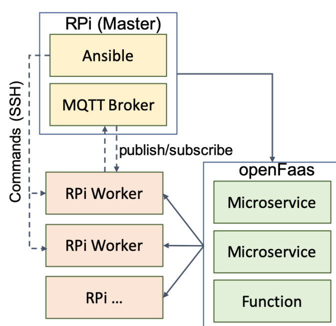

Figure 3: CCAM Architecture.

Prometheus is a monitoring tool that can be used to gather and process application metrics. Contrary to other monitoring tools it does not rely on the application delivering the metrics to the monitoring tool. Prometheus scrapes the metrics from a predetermined interface in a given interval. This means that the metrics are expected to be exposed by the application.

OpenFaas is a Function-as-a-Service framework. It can be deployed on top of Docker swarm or Kubernetes. When starting the openFaas framework a few standard Docker containers are deployed. openFaas contains a simple autoscaling rule that leverages the default metrics aggregated by Prometheus and scales based on predefined thresholds (openFaaS, 2019).

4.3 Combining Tools and Technologies

To meet the set requirements, tools and technologies used to realize the platform need to provide a high level of flexibility while managing the limited hardware resources of the cluster. Combining the concepts and technologies presented above are one way of achieving this: by splitting the application into microservices and containerizing it, the hardware can be reallocated dynamically. This also enables scaling the different parts of the application in a simple manner. The Docker images leave a minimal footprint on the devices, making efficient use of the hardware. MQTT, as a lightweight protocol, has a similar advantage while its underlying publisher/subscriber pattern simplifies communication in an environment where services are added and removed constantly. Establishing peer-to-peer communication would be significantly more complex. OpenFaas is a simple way of building and deploying services/functions across the cluster. An overview of the different technologies in the platform is provided in Figure 3.

5 ARCHITECTURE USE CASE

This section introduces a concrete use case for the proposed architecture based on the 5G-CARMEN project, i.e., an application, deployed on a cluster of single-board devices, that is based on the concepts of serverless computing and the microservices architecture. Additionally, the system will support adaptive auto-scaling of its components.

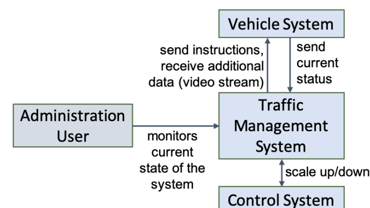

Figure 4: Interaction between systems.

The application is composed of three main layers.

The platform layer realizes the core architecture of the cluster and the application. The system layer includes the components of the system. On top of these two layers, the controller layer scales the components of the platform. Figure 4 shows the interaction between system layer, controller layer and additional components. The Traffic Management System (TMS) contains all core components of the system. Administrators can access an interface to monitor its current state. There is a constant exchange of messages between the TMS and the Vehicle System (VS) that contains simulations of vehicles. The Control System (CS) is used to scale parts of the TMS based on the current situation and a number of predefined factors.

5.1 Platform

The platform is deployed on a cluster managed by Docker Swarm. The cluster includes one master node and an arbitrary number of worker nodes. *Ansible* is used to execute commands on all nodes without having to connect to each node individually. All nodes are able to connect to the MQTT broker that is running on the master device after startup. *Avahi* allows all nodes to establish a connection to the master by addressing it by its hostname. Using Docker swarm and openFaas, the RPIs are connected in such a way that they can be seen as one system. If a service is supposed to be deployed, openFaas will distribute it among the available nodes. There is no need to specify the specific node as this abstraction layer is hidden behind openFaas. OpenFaas is also utilized to scale the services independently. Communication between the services is achieved by relying on the openFaas gateway as well as on the MQTT broker.

The cluster is comprised of eight Raspberry Pi 2 Model B connected to a mobile switch via 10/100 Mbit/s Ethernet that is powering the RPIs via PoE (Power over Ethernet) (Scolati et al., 2019). All nodes of the cluster run Hypriot OS.

5.2 Traffic Management System

The platform in its current state serves as a functional, concrete proof-of-concept of a Traffic Management System. Figure 5 visualizes the interaction between the most important services of the system. The Vehicle System VS is used to simulate vehicles. The vehicles continuously publish their current position and status to the MQTT Broker. The gatherer receives these messages and passes them on to the decision-function, which is an openFaas function that calculates a corresponding command (slow down, accelerate, keep current speed) for each message. The gatherer relays these commands back to the vehicle via the MQTT Broker. In addition, a video service provides a video stream to be consumed by the vehicles and the dashboard service enables administrators to access a web-interface that provides an overview of the system. All of the aforementioned services are deployed and managed via the openFaas gateway.

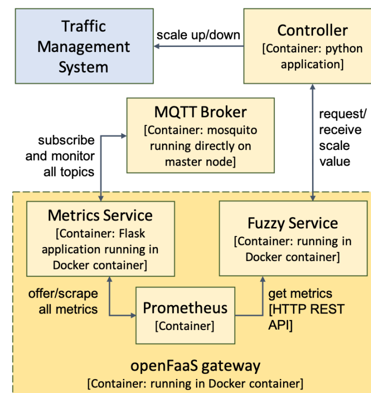

Figure 5: Traffic Management System.

5.3 Communication

The system uses the mosquito MQTT broker to establish a communication bus between systems and services. The messages are based on the JSON data exchange format. The first advantage of using MQTT is the minimized network latency (Light, 2018), which is crucial for CCAM use cases. Additionally, not all services may be accessible from all other services within the cluster. Therefore, having only a single central master node, running the mosquito broker, that needs to be accessible from any other part of the system, saves a significant amount of additional configuration work. Making use of the publisher/subscriber pattern is another advantage since messages may need to be addressed to more than one service at a time. Hence, MQTT greatly reduces configuration efforts and network latency.

5.4 Vehicle System

The vehicle system VS provides a service simulating here in the experimental set-up a vehicle on the road section that is to be monitored. The VS may contain an arbitrary number of vehicle services. The service randomly creates a vehicle and continuously publishes MQTT messages to the broker. The messages contain JSON objects that include the following fields: the current speed of the vehicle, the lane of the vehicle, the license plate number, the vehicle type (manually driven vehicle, an autopiloted car,

a convoy truck), the UUID of the vehicle, the altitude and whether the vehicle supports video streaming. The vehicle service subscribes to a specific commands topic that is defined as "commands/" followed by the UUID of the vehicle. Once it receives a command on this topic, the vehicle updates its speed according to the command.

5.5 Traffic Management Components

The **gathering service** receives information about the current location of the vehicles. The gatherer holds a four-dimensional array of length 1000 that represents the part of the road that is to be monitored. Arrays 0 and 1 are two lanes leading south, while array 2 and 3 are the lanes leading north. It will update the road array with the current location of each vehicle, thus creating an actual representation of the road at a given point in time. Once a vehicle has left the area that is covered, the gatherer will delete the last location of that vehicle. The gatherer also passes specific messages to each vehicle. However, it does not include the logic for choosing a command for a specific car in a specific situation. Instead, it relays this information along with the current road-array to the decision-function that will decide what command is to be executed. The returned command will then be published by the gatherer. The decision-making logic is a built-in component of the gathering service.

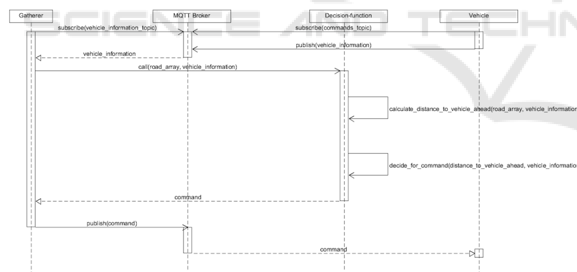

Figure 6: Sequence diagram showing the interaction between the gatherer and the vehicle instances.

The **video service** provides video streaming functionality. The implementation implements a streaming application using flask. Using this approach, independent jpeg images can be streamed continuously on a specific route. The video service was implemented as an openFaas stateless microservice. This way, video streaming data can be served from different instances of the video service by accessing the stream via the gateway.

The **decision function** receives the current road-array and information about the current location of a vehicle and uses this information to pick a suitable action that is to be taken by that vehicle. In its current configuration, it aims at maintaining a minimum distance of four blocks (elements of the array) between vehicles. Should the vehicle at the back get to close to the vehicle in front, the command slows it down until the desired distance is reached. If the distance is sufficient and the car has not reached the advised speed for this section of the road, the command will instruct the vehicle to accelerate.

The **dashboard function** provides an overview over the MQTT messages that are passing through the broker. It shows the configuration messages, issued commands as well as information for every active vehicle. The dashboard is based on an open-source, MIT licensed project that contains sample code for a simple MQTT web-dashboard.

The **config function** is used to automatically set the configuration of certain system-wide parameters. The function takes a JSON object as a parameter that contains values such as the hostname of the MQTT broker or topic definitions. The config function publishes this object to a certain topic and flag it as a message to be retained. On startup, all concerned services will subscribe to this configuration topic and immediately receive the last retained message. The received global values are used to update the internal variables of the respective services.

5.6 Monitoring

Monitoring uses the openFaas Prometheus instance.

The **metrics service** is used to acquire metrics about the system by serving as a central hub that accumulates all cluster-wide metrics and by publishing those metrics via a flask HTTP endpoint. This endpoint is the central interface for Prometheus to scrape from. The prometheus python API is used to implement the metrics. The metrics service currently exposes the number of messages, the number of active cars as well as the cumulative memory and CPU usage. The number of messages is implemented as a counter that continuously increases. CPU/memory usage, on the other hand, is realized by using a gauge, which can be a set to an arbitrary number.

The **Prometheus instance** is used to store metrics and query them when needed. Prometheus provides a rest API along with a language called PromQL to aggregate and query metrics. The aggregated data is returned in the JSON format.

Before startup, Prometheus needs to be informed about the endpoints that metrics should be scraped from. The Prometheus instance that is shipped with openFaas only scrapes metrics from the openFaas gateway since only metrics related to function execution are being monitored by default.

6 EVALUATION

Now, the proposed architecture shall be evaluated. The focus lies on evaluating the performance of the proposed serverless microservice solution and the auto-scaling approach in the context of the CCAM non-functional requirements. To give this a concrete perspective, we also evaluated the maximum number of vehicles the cluster architecture (including the network it was operated in) could support for the use cases in a given road section.

6.1 Requirements, Metrics, Setup

We start discussing at a high level the non-functional CCAM requirements and how we addressed them. *Safety:* Downtime during reconfiguration and slowdowns caused by high load are avoided by using fuzzy-adaptive scaling. *Performance:* The scaling algorithm is based on Service-Level-Objectives that maintain performance needs. *Reconfigurability:* The application is split into different containerised services that can be easily deployed across the cluster from a central node. Physical access to individual nodes is not necessary. *Scalability:* The system is reconfigured on the fly by using the auto-scaling functionality based on adaptive fuzzy scaling for serverless containers. *Dependability:* Several gatherer services are usually deployed at the same time. If a gatherer is added or removed the vehicles are automatically distributed among the available gatherers, providing better reliability and availability. The scaling functionality ensures that the performance remains above a certain SLO. *Interoperability:* The application may be deployed on different hardware nodes. Docker containers decouple the application from the actual hardware layer and the host OS. The only technical requirement is the availability of Docker. In the following, dependability, scalability and and performance will be addressed in detail.

These objectives were evaluated for two different cluster set-ups. In order to obtain a first understanding of the system and the possible range of variables, a first calibration pilot was conducted on a small cluster of only three devices. Then, the evaluation procedure was repeated for a complete cluster of eight devices.

All evaluation steps report on a number of performance metrics that indicate the effectiveness of the system or provide insight into an internal process. The *Message Roundtrip Time (MRT)* is the central variable of the system that reports on the effectiveness of the autonomous driving functionality. Included in the MRT is the openFaas-supported *Function Invocation Time (FIT)* that is listed separately in order to individually report on the serverless performance. In this evaluation, all MRT and FIT values are considered average values aggregated over the last 20 seconds after the previous scaling operation was completed. For our use case, the maximum scale value was unknow, but could generally be specified beforehand. Over the course of this evaluation, different MRT thresholds are applied. For all set-ups and iterations that were evaluated, the hardware workload was measured by computing the average CPU and memory usage over all nodes of the cluster, combining it into a single value. This is feasible since the entire cluster can be seen as one system by combining the individual nodes using Docker swarm and openFaas.

Table 1: Results for a cluster of eight RPIs using the full version of the system. The FIT was not measured since the decision-function was removed here.

Vehicles	Mem Usage	CPU Usage	MRT	FIT
2	2.92	36.53	0.02	-
4	3.29	37.02	0.025	-
6	3.89	37.90	0.029	-
8	5.48	38.41	0.028	-
10	5.81	39.37	0.17	-
12	5.17	39.78	0.028	-
14	6.98	40.07	0.028	-
16	4.2	40.33	0.025	-
18	4.5	40.67	0.025	-
20	4.94	41.1	0.027	-
22	5.19	41.43	0.028	-
24	5.74	41.7	0.027	-
26	5.99	42.14	0.028	-

Table 2: Setup as in Table 1 with higher vehicle numbers.

Vehicles	Mem Usage	CPU Usage	MRT	FIT
50	8.63	46.75	0.03	-
75	11.79	51.7	0.032	-
100	14.48	56.57	0.04	-

An initial calibration pilot was conducted to obtain an initial idea of the system's capabilities and adjust the manually-tuned parameters accordingly. It was also used to evaluate whether the scaling functionality yields promising results before putting it to use in a bigger set-up. The evaluation was started with a cluster consisting of three RPIs: a master and two worker nodes. The maximum scale value was initially set to 5 in order to avoid scaling the system to inadequate (overly resource-demanding) configurations.

6.2 Evaluation of the Complete System

A cluster of eight RPIs was used. The decision-making functionality is included in the gatherer service, which can be scaled independently. This avoids the need to call the decision function for each message. The results can be found in Tables 1 and 2. Values of a sample run can be found in Table 3.

Table 3: Sample scaling for a cluster of eight RPIs. Manually set variables: Maximum Scale Value: 12, Number of cars: changing, MRT Threshold: 0.3 seconds.

Iteration	# cars	Fuzzy Scale Value	MRT after scaling	Adjustment Factor
1	25	4.77	Initial	0.0
2	25	6.94	0.03	2.58
3	25	7.79	0.04	0.0
4	25	8.80	0.03	-2.58
5	50	10.11	0.178	-7.74
6	50	10.82	0.44	-10.32
7	50	10.82	0.44	-10.32
8	75	11.06	0.09	-12.9
9	75	11.24	0.033	-18.06
10	75	11.3	3.95	-20.64
11	75	11.34	0.07	-23.22

Considering the results in Tables 1 and 2, adequate MRT values can be noted. We recorded values for different numbers of vehicles. The MRT values appear to be growing exponentially. Between 16 and 24 vehicles, the MRT only increases by about 8%. If we look at the CPU usage, we can again see suitable values, here it consumes about 8-10% more of CPU with a bundled and separately scaled gatherer. The additional computing power and time that is needed to make a decision internally is neglectable when compared to the significant overall MRT advantage.

The system accommodates around 75 vehicles. The bottleneck appears to be the network: when trying to increase the number of vehicles beyond this point, the network was unable to handle the amount of messages that were exchanged, which resulted in connections and packets being dropped continuously.

6.3 Evaluation Summary

Containerization comes with a small performance loss compared to traditional set-ups. However, the advantages are generally more significant than the downsides. The evaluation indicates that serverless function calls should be restricted since they introduce network latency problems. Refactoring the proposed solution to reduce the number of necessary calls to openFaas functions resulted in a significant increase in performance. The implemented scaling algorithm works as intended. Based on give service-level objectives, the only values that need to be set manually are the maximum scale value and the adjustment factor. The limiting factor appears to be the network. Our configuration was not able to process more than 75 vehicles at a time. CPU and memory usage numbers as well as the steady, but slow increase of the MRT imply that the hardware itself should be able process a higher number of vehicles. The architecture thus yields satisfying results in terms of hardware consumption and performance (MRT).

7 CONCLUSIONS

We introduced a containerized traffic management architecture for the CCAM context deployable on a cluster of edge devices. The applied approach results in a reconfigurable, scalable and dependable system that provides built-in solutions for common problems such as service discovery and inter-service communication. An implementation of a proof-of-concept has experimentally demonstrated the suitability.

The evaluation also showed that the given lightweight and constrained set-up is only able to process up to 75 vehicles simultaneously in common data streaming and coordination activities. The bottleneck that prevents the system from scaling even higher appears to be the network infrastructure as well as the limited internal networking capabilities of the RPI. This shall be addressed in a future extension.

Implemeting full security and a trust environment is part of future work (El Ioini et al., 2018; El Ioini et al., 2018a). For instance, the communication between the services using the openFaas gateway could be encrypted using TLS. The MQTT connections can be secured. Most MQTT brokers, including mosquitto, also support TLS for securing the communication channels.

ACKNOWLEDGEMENTS

This work has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement 825012 - Project 5G-CARMEN.

REFERENCES

5G-CARMEN (2019). 5G-CARMEN - 5G for Connected and Automated Road Mobility in the European UnioN. https://www.5gcarmen.eu/. Accessed: 2019-11-10.

Baldini, I., Castro, P. C., Chang, K. S., Cheng, P., Fink, S. J., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R. M., Slominski, A., and Suter, P. (2017). Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178.

Ellis, A. (2018). Introducing stateless microservices for openfaas. https://www.openfaas.com/blog/stateless-microservices/.

Fang, D., Liu, X., Romdhani, I., Jamshidi, P. and Pahl, C. (2016). An agility-oriented and fuzziness-embedded

semantic model for collaborative cloud service search, retrieval and recommendation. In *Future Generation Computer Systems*, 56, 11-26.

Fowley, F., Pahl, C., Jamshidi, P., Fang, D. and Liu, X. (2018). A classification and comparison framework for cloud service brokerage architectures. IEEE Transactions on Cloud Computing 6 (2), 358-371.

Gand, F., Fronza, I., El Ioini, N., Barzegar, H. R., Azimi, S. and Pahl, C. (2020). A Fuzzy Controller for Self-Adaptive Lightweight Edge Container Orchestration. In *International Conference on Cloud Computing and Services Science CLOSER*.

Gand, F., Fronza, I., El Ioini, N., Barzegar, H. R. and Pahl, C. (2020). Serverless Container Cluster Management for Lightweight Edge Clouds. In *International Conference on Cloud Computing and Services Science CLOSER*.

El Ioini, N. and Pahl, C. (2018). A review of distributed ledger technologies. OTM Confederated International Conferences.

El Ioini, N. and Pahl, C. (2018). Trustworthy Orchestration of Container Based Edge Computing Using Permissioned Blockchain. Intl Conf on Internet of Things: Systems, Management and Security (IoTSMS).

El Ioini, N., Pahl, C. and Helmer, S. (2018). A decision framework for blockchain platforms for IoT and edge computing. IoTBDS'18.

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

Jamshidi, P., Pahl, C., Chinenyeze, S. and Liu, X. (2015). Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective. In *Service-Oriented Computing - ICSOC 2014 Workshops*. 6–19.

Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A. and Estrada, G. (2016). Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. *Intl Conf Quality of Software Architectures*, 70–79.

Jamshidi, P., Pahl, C. and Mendonca, N. C. (2016). Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Computing, 50-60.

Jamshidi, P., Pahl, C. and Mendonca, N. C. (2017). Pattern-based multi-cloud architecture migration. Software: Practice and Experience 47 (9), 1159-1184.

Kiss, P., Reale, A., Ferrari, C. J., and Istenes, Z. (2018). Deployment of iot applications on 5g edge. In *2018 Intl Conf on Future IoT Technologies*, pp. 1–9.

Kritikos, K. and Skrzypek, P. (2018). A review of serverless frameworks. In *2018 Intl Conf on Utility and Cloud Computing (Companion)*, pp. 161–168.

Lama, P. and Zhou, X. (2010). Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *2010 Intl Symp on Mod, Anal and Sim of Comp and Teleco Syst*, pp. 151–160.

Le, V. T., Pahl, C. and El Ioini, N. (2019). Blockchain Based Service Continuity in Mobile Edge Computing. In *6th International Conference on Internet of Things: Systems, Management and Security*.

Light, R. (2018). Mqtt man page. https://mosquitto.org/man/mqtt-7.html. Accessed: 2019-11-12.

Mendonca, N. C., Jamshidi, P., Garlan, D. and Pahl, C. (2020). Developing Self-Adaptive Microservice Systems: Challenges and Directions. In *IEEE Software*.

openFaaS (2019). openfaas: Auto-scaling. https://docs.openfaas.com/architecture/autoscaling/. Accessed: 2019-11-11.

Pahl, C., El Ioini, N., Helmer, S. and Lee, B. (2018). An architecture pattern for trusted orchestration in IoT edge clouds. Intl Conf Fog and Mobile Edge Computing.

Pahl, C., Jamshidi, P. and Zimmermann, O. (2018). Architectural principles for cloud software. ACM Transactions on Internet Technology (TOIT) 18 (2), 17.

Pahl, C. (2003). An ontology for software component matching. International Conference on Fundamental Approaches to Software Engineering, 6-21.

Pahl, C. (2005). Layered ontological modelling for web service-oriented model-driven architecture. In *Europ Conf on Model Driven Architecture – Foundations and Applications*.

Riggio, R., Khan, S. N., Subramanya, T., Yahia, I. G. B., and Lopez, D. (2018). Lightmano: Converging nfv and sdn at the edges of the network. In *2018 Network Operations and Management Symp*, pp. 1–9.

Pahl, C., Jamshidi, P. and Zimmermann, O. (2020). Microservices and Containers. Software Engineering (2020).

Pahl, C., Fronza, I., El Ioini, N. and Barzegar, H. R. (2019). A Review of Architectural Principles and Patterns for Distributed Mobile Information Systems. In *14th Intl Conf on Web Information Systems and Technologies*.

Scolati, R., Fronza, I., Ioini, N. E., Samir, A., and Pahl, C. (2019). A containerized big data streaming architecture for edge cloud computing on clustered single-board devices. In *9th International Conference on Cloud Computing and Services Science CLOSER*.

Tata, S., Jain, R., Ludwig, H., and Gopisetty, S. (2017). Living in the cloud or on the edge: Opportunities and challenges of iot application architecture. In *2017 Intl Conf on Services Computing*, pp. 220–224.

von Leon, D., Miori, L., Sanin, J., El Ioini, N., Helmer, S. and Pahl, C. (2019). A Lightweight Container Middleware for Edge Cloud Architectures. Fog and Edge Computing: Principles and Paradigms, 145-170.

Xi, B., Xia, C. H., Liu, Z., Zhang, L., and Raghavachari, M. (2004). A smart hill-climbing algorithm for application server configuration. In *13th Int. Conf. on WWW*.