AMERICAN UNIVERSITY OF BEIRUT

On-Demand Deployment Of Containerized Micro-Services On Vehicular Fogs

by

Hani Osamah Sami

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
May 2019

AMERICAN UNIVERSITY OF BEIRUT

On-Demand Deployment Of Containerized Micro-Services On Vehicular Fogs

by

Hani Osamah Sami

Approved by:

[illegible]

Dr. Wassim El-Hajj, Associate Professor and Chair — Advisor

Computer Science - AUB

[illegible]

Dr. Azzam Mourad, Associate Professor and Chair — Member of Committee

Computer Science - LAU

Haidar Safa

Dr. Haidar Safa, Professor — Member of Committee

Computer Science - AUB

Date of thesis defense: April 23, 2019

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name:	Sami	Hani	Osamah
	Last	First	Middle

[✓ Master's Thesis] Master's Project Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☒ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after:

[✓ One] ___ year from the date of submission of my thesis, dissertation or project.
Two ___ years from the date of submission of my thesis, dissertation or project.
Three ___ years from the date of submission of my thesis, dissertation or project.

[illegible]	May 06, 2019
Signature	Date

Acknowledgements

I first would like to thank my supervisor Dr. Wassim El-Hajj, Chair of the Computer Science Department, for his continuous support and guidance throughout this work. Without his passionate participation and input, this thesis could not have been successfully conducted.

I would also like to thank my co-supervisor Dr. Azzam Mourad, chair and associate professor at LAU. I always found the door of his office opened whenever I ran into trouble or I have any questions and concerns about my research. He always steered me in the right direction whenever he thought I needed help.

I would also like to acknowledge Professor Haidar Safa of the Computer Science Department at AUB as the second reader of this thesis, and I am gratefully indebted to his valuable comments on this thesis.

Finally, I am very grateful to my parent and friends who endured this long process with me and always offered love and support.

An Abstract of the Thesis of

Hani Osamah Sami for Master of Computer Science
Major: Computer Science

Title: On-Demand Deployment Of Containerized Micro-Services On Vehicular Fogs

With the vehicular manufacturing advancement, real-time vehicular applications require fast processing of the vast amount of generated data by vehicles, thus a maintained service availability and reachability while driving. These applications use sensed data generated by IoT devices on board to support vehicular applications such as, the self-driving cars, real-time traffic signs updates, or even video surveillance and analysis applications. Fog devices are capable of bringing cloud intelligence near the edge, making them a candidate to such requests. However its location, processing power, and technology used to host and update services affect its availability and performance while considering the mobility patterns of vehicles. Contemporary work in the literature examines the use of virtual machines (VM) to host the essential services on Road Side Units (RSU). The RSU usage raises many limitations including the difficulty of updating services hosted in VMs, RSUs range of coverage, and other handover problems when considering SDN controller to route traffic between RSUs. On the other hand, the evolvement of the On-Boarding Units helps in performing some of the required processing locally. However, one OBU is still not enough to perform real-time processing of generated data and to enable efficient decision making in critical applications like self-driving cars. In this thesis, we overcome the mentioned limitations by introducing a Kubeadm OBUs clustering technique to enable on-demand micro-services deployment with the least costs possible. Docker Containerization technology is adapted to offer light service installment and smooth services updates based on the application's needs. A hybrid multi-layered networking architecture is proposed to maintain network reachability between the requesting user and available Kubeadm fog cluster. We present a master node election algorithm to select the cluster orchestrator in the most effective way taking into consideration the mobility conditions of vehicles. Cluster failures can be recovered following a proposed recovery algorithm. Moreover, our framework leverages a vehicular container placement scheme that produces optimal vehicles selection and services distribution. An Evolutionary Memetic Algorithm is elaborated to solve our multi-objective vehicular container placement problem. We also present a simulated testing environment with real datasets to demonstrate various improvements interpreted by the relevance and efficiency of (1) forming Kubeadm vehicular fog clusters with maximum time availability and user support, and (2) deploying services on selected OBUs based on the vehicular container placement approach.

Keywords — Vehicular Fog Computing, Vehicular Clustering, Orchestration, Container, Micro-Services, Kubeadm, Docker, Memetic Algorithm

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Problem Statement

The OBU evolved from a simple device that can track the vehicle location and speed, to a networking device capable of communicating with neighboring vehicles and opening a stable connection with servers on the cloud. The OBUs are now also capable of performing various computation tasks depending on their CPU capabilities. Buses in modern cities are now equipped with multiple OBUs that can support applications helping drivers and providing luxury to passengers. An example of these applications is the recording of violating cars driving on buses lines, letting drivers abide by the speed limit, calculating the number of passengers, measuring air pollution, voice communication with passengers and drivers, etc. This type of applications requires enough resources which one OBU only cannot handle.

The improvement in the wireless technology, new development of services, and the evolution in onboard unit's (OBU) physical resources capabilities, disclosed a potential in enhancing the vehicular applications by employing the available technologies and resources into vehicular fog computing clusters that can enrich the user experience and allow a new development paradigm targeting vehicular market. One of the essential applications that can be supported in this case is self-driving cars. This application is very time sensitive, and more processing power is needed to provide more accurate analysis of sensed data from all vehicles around to perform lane changing, detect humans or objects crossing the roads, and be alarmed by any accident happened in front.

Vehicles are mobile sensors generating a lot of data that should be processed in real time. For this reason, it is not feasible for time-critical applications to receive their support from servers on the cloud. Here arises the concept of fog computing capable of bringing cloud intelligence near the edge by hosting services next to the user. Fog devices minimize the load on the cloud by doing the major work, reducing networking delay, enhancing user's devices battery life, improving QoS for various types of applications.
This being said, vehicular devices should be supported by fog servers whom the applications can rely on to provide the best user experience and meet the networking and processing needs. Many literature works tried to develop a workaround to solve the below-mentioned limitations; however, there is no architecture until now that can provide full support for vehicles all the time with any service required. These limitations can be summarized as follows:

1. The limited resources on the OBUs makes it impossible to download a VM and start it. So there is a need for a technology that can better utilize disk space and provide the ability to update services with the least delay possible to host these services on cars themselves. Besides, very few works considered the use of electronic devices carried by passengers.

2. The dynamic car mobility and erratic behavior of drivers make it a challenge to maintain a stable network connection between cars and service provider, which is in most cases the RSU. The limited capabilities of base stations that use current cellular technologies is another challenge to host services on cars. If every vehicle connects to the BS, the base stations are overwhelmed with requests and can suffer from computational and networking problems (performance issues). So there is a need for a networking architecture that can keep vehicles connected all the time.

3. RSU concept was developed to partially overcome the challenges mentioned above by hosting services on them and keep the car attached to it while in range to access the required services. Several problems arise here, such as, the limited coverage of RSUs on the road where it is costly to deploy RSU server everywhere, the limited number of vehicles that an RSU can serve, the cost of pushing new VMs every time the user request a new service.

On the other hand, the usage of containerization technology is rising because of its capability of replacing VMs with low deployment costs and faster booting time. This is because containers use the actual operating system of the device, can share the bin and lib files, and occupy much smaller OS image sizes on disk. On the other hand, the VM uses their copy of the operating system with large OS files and denser deployment on the machine. Also, companies are shifting to re-engineering their services as microservices to improve the deployment costs, enhance distributed computing, and provide easiness in maintenance and upgrading procedures.
In this paper, we make use of one of the well-known containerization technology Docker to deploy our services on cars on the fly, making cars as fog devices supporting other users. Vehicles in our architecture form a cluster that can host micro-services

related to one service or more. These clusters are managed by a cluster node using Kubernetes utility Kubeadm developed recently by Google. We also adopt a technique proposed in [33] to provide a long time stable connectivity between the user and the serving cluster. To the best of our knowledge, we are the first to propose a methodology that uses vehicles and devices on-board as fog devices to push containers of micro-services on the fly on them, and at the same time provide long term support for moving users regardless of their connection status. Through our approach, we can open the door for a new development area serving a new generation of vehicle's applications.

1.2 Objectives

The main objective of this thesis is to provide real-time support for vehicular applications by pushing services next to vehicles when needed, and to be able to keep the car connected to its service provider all the time after achieving a tolerable end to end delays and an enhanced quality of service with negligible delay and fast processing when possible. Our main objectives are:

1. Develop a framework that efficiently initializes clusters of vehicles to benefit from OBUs and onboard resources to push services most efficiently with the least initialization time possible.

2. Build a networking architecture combining cellular technology and 802.11p for an ad-hoc wireless network to keep cars connected at all times.

3. Elaborate an efficient resource selection and micro-services assignment when deciding to use a mobile cluster to serve a particular user.

1.3 Methodology

In this thesis, we take advantage of the OBUs and onboard devices availability into becoming fog devices that can serve nearby or far users. We make use of an orchestration technology that can manage containers pushed to worker vehicles, so we first start by preparing Kubeadm clusters of vehicles after electing the orchestrator. We then elaborate an efficient on-demand deployment approach on vehicles using Docker technology. Second, we build five layers of networking architecture that connect vehicular fogs with users to achieve a stable connection between users and serving fogs. Third and last, we formulate and provide a solution for the intelligent vehicular container placement which selects the best vehicles that can host a group of requested micro-services by the user. Below we list the fundamental contributions of this thesis:

1. Introducing the on-demand vehicular fog clusters creation and management counting on OBUs and personal devices using containerization and orchestration technologies for pushing and monitoring micro-services.

2. Introducing a networking architecture leveraging the use of LTE and 802.11p under five layers of communication including BS, RSUC, RSU, orchestrator, workers (fog devices), and users.

3. Formulating the Vehicular Container Placement problem (VCP) as a multi-objective optimization problem after proving its complexity, and providing a heuristics to solve it.

4. Presenting a local master election algorithm and introducing a recovery algorithm to overcome potential cluster failures.

5. Showing the architecture feasibility and the VCP solution advantage by building

our customized testing environment. This is done by merging Containernet, Mininet-Wifi, and SUMO simulators functionalities.

1.4 Thesis Organization

The remainder of this thesis is organized as follow:

In chapter II, we provide a piece of background information about the OBU, RSU, RSUC, Docker, Kubernetes, Kubeadm, 802.11p, base stations, cellular technology LTE and how 5G can improve the performance of our framework. We also go over some surveys and reviews that point out the need for creating vehicular fog clusters including the main challenges solved by our proposed methodology. Some of the literature work from where we inspired parts of our architecture is provided, and others in the area of vehicular fogs.

In chapter III, we describe our on-demand vehicular fog clusters creation, the methodology used, and detailed writing of each component functionalities per layer. This also includes the networking architecture proposed to keep vehicles connected. Finally, we evaluate the performance of our primary approach using Containernet, Minient-Wifi, and SUMO simulators.

In chapter VI, we mathematically formulate our vehicular container placement problem by providing the input, output, constraints and objective functions. A Memetic algorithm that solves this problem is provided. Finally, we evaluate the proposed VCP solution and show its impact on our framework performance.

In Chapter V, we conclude our work and summarize the thesis contribution. Future goals are also discussed.

Chapter 2

Background and Related Work

2.1 Introduction

In this chapter, we start first by presenting some of the well known time sensitive vehicular applications that are not fully supported yet. We then explain the terminologies used in this paper and the technology behind them. We start by defining the On-Board Unit (OBU) and its revolution throughout the years including the smart devices carried by passengers. Next, we present the common usage of the roadside unit (RSU) and roadside unit controller (RSUC) to help vehicular communication and enhance the performance of their applications. In addition, the LTE and 5G cellular networks that we count on in our architecture are reviewed.

Moreover, we explain the motivation behind using the 802.11p networking protocol in VANET communications. We define virtual machines and containers and divulge into the difference between them. After that, we discuss the concept and need behind orchestration technologies available nowadays and compare them. We describe the multi-objective optimization problem and the multiple methods used to solve it. We then discuss the Memetic algorithm and the improvements achieved over the Genetic algorithm, as well as the different steps that build such an algorithm. Finally, we present a literature review of recent surveys that mention the vehicular environment limitations and others that try to serve vehicular applications using VMs deployed on RSUs or OBUs to perform basic tasks. We also review a work that proposed the use of hybrid network architecture to maintain vehicular connections.

2.2 Applications in Vehicular Fog Network

Some applications heavily rely on vehicular computational resources to achieve their purpose. These applications are categorized into safety, traffic control, and infotainment applications[15]. Route navigation updates use real-time road congestion updates from cars on the road to defining the shortest path for the user in terms of traffic congestion to reach their destination, where e-road signs can be adjusted to redirect traffic flows accordingly. This application requires the sensing and data processing services to be installed near a group of cars on each suggested road on the way. Video surveillance is another application that can be used to report accidents on the street, track vehicles or people, or report any traffic violations. This also requires applications to be hosted on cars to capture and process videos and to report them to the concerned security department. Re-scheduling traffic lights in case of an emergency car passing or traffic jam problems requires processing the sensed data to study road conditions to make proper and accurate decisions on changing the traffic light in real time. In these cases, we need an architecture that builds vehicular fog cluster formed using the vehicular infrastructure to collect and process sensed data using the micro-services hosted as containers on them. This way, the processing is no longer done on RSUs or the cloud, as well as utilizing available vehicular resources.

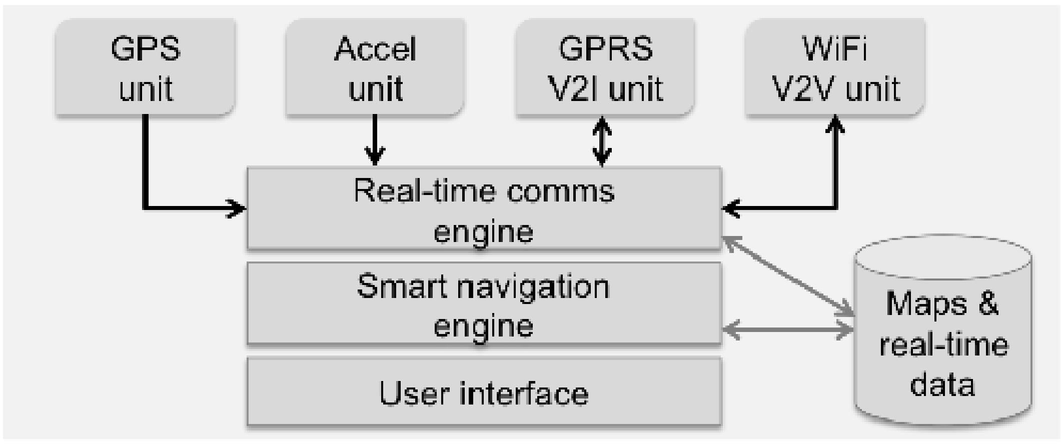


Figure 2.1: OBU Internal Architecture [4]

2.3 On-Board Units and Personal Devices

On-board units (OBUs) started from simple devices that can track vehicles locations and monitor their speeds, to networking devices capable of connecting to servers placed on the cloud. Even-though one OBU is not enough to perform fast processing; some modern cities are equipping buses with more than one OBU to enable the integration of smart vehicular applications to positively enhance passengers' and drivers' experience by improving its processing capabilities. Therefore, the buses can identify violating cars driving on its lanes, counting the number of passengers, alarming the driver of any traffic violations, measuring air pollution, and opening conversations with passengers. According to a study conducted in 2017 [1], on average, every person nowadays uses at least two smart mobile devices a day. These smart devices are becoming more powerful with up to six cores [30] in one phone. These devices can be a great source of computation, especially when combined with the power of clustered OBUs. As discussed in the next section, researchers are counting on RSUs to do the heavy processing required to maintain a fast response time when serving ve-

[1]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5680647/

hicular applications. Although RSUs are the only solution to support heavy vehicular computations, their limitations are affecting the development of other applications that are non-delay tolerant and must be supported in real time. An example of these applications is self-driving cars. This application requires a real-time update on the road conditions from its sensors and neighboring cars. This vast amount of data generated by these cars should be processed fastly for the car to take an instant decision of lane changing, identify an object is crossing the road, or if an accident is happening in front to change directions. Separate applications can be real time traffic update, and video surveillance applications to analyze accident scenes, to identify traffic violations, or for police officers to track criminals. In this paper, we count on OBUs and passengers' smart devices to form a fog computing infrastructure to process the huge amount of data in a distributed way.

2.4 RSU and RSUC

Roadside units can be any device embedding processing unit(s) with the ability of wirelessly covering parts of the road by connecting with mobile vehicles using 802.11p protocol. In some contexts, an RSU can be a fog device that processes the sensed data generated by cars. The main purpose of an RSU is to host services that process vehicular requests in range. Usually, virtual machines are hosted inside RSUs to preconfigure and run required services [1]. There are many limitations behind the use of RSUs which limit their ability to support some applications. These restrictions can be summarized as, the limited resources to serve a huge number of vehicles, the restricted range of coverage, and the high delay of updating services when needed. To overcome these limitations, some workarounds were proposed in the literature such as, solving an optimization problem to find the best distribution of RSUs on the roads in a way

that maximizes its range of coverage to continuously serve vehicles [17], however it is very expensive to deploy RSUs everywhere, so it is not feasible to fully count on them to cover all vehicles moving on the roads. A different work proposed the integration between the SDN controller to manage the route between RSUs and minimize the number of service migrations. SDN was also used to manage routes between cars to maximize the chance of reaching the RSUs through other neighboring vehicles. On the other hand, SDN controllers are subject to networking and computational delays that prevent them from being a good candidate to redefine routes in some time-sensitive vehicular environments.
In case an RSU has limited computational resources, the sensed data of the vehicles are forwarded to a roadside unit controller (RSUC) which is a centralized server linked to more than one RSU using Ethernet with more processing power. The RSUC reports the analyzed data to the cloud and can control the behavior of other RSUs.
Depending on the processing power available on the RSU or RSUC, they can play the role of fog devices capable of processing the sensed data generated by vehicles, provided the mentioned limitations.

2.5 LTE, 5G and 802.11p

LTE (Long Term Evolution) is a 4g wireless communication technology that is designed to overcome 3G limitations and provide a better quality of service compared to WiMAX while benefiting from the existing cellular network infrastructure. LTE was invented because of the constant user demand for higher data rates at lower costs with fewer delays. 3G networks introduce delays of around 100ms; this causes great difficulties for more demanding time-sensitive applications. LTE is now able to deliver data rates at peaks of 300 Mbps on the downlink, and 75 Mbps on up-link with a delay

of 1ms. More importantly, mobility is better supported with LTE and can offer high performance up to 120 km/h. Moreover, coverage is enhanced with lower latency and simultaneous user support [9].
Higher data rates and greater capacity continue to be the key driver behind the network evolution in 5G, in addition to energy consumption on devices, spectrum, and latency. 5G is expected to provide a peak download data rate of 20Gbps and an upload rate of 10Gbps. Moreover, 5G reduces network energy usage by 90%. A high number of user can access simultaneous connection at a maintained high rate. 5G will positively impact a user's experience and opens new opportunities for the development of new technologies and eras. Self-driving cars performance and time to take decision can now be improved because of the fast data transmission it can achieve. 5G uses mmWave that refers to a specific part of the radio frequency that has a very short wavelength spectrum to achieve high bandwidth. mmWave in 5G can be used to enhance Vehicles to everything communications (V2X) to incorporate low latency links between moving vehicles themselves and cloud servers [8].
802.11p is another amendment of the 802.11 standard to offer vehicular wireless communications by allowing the vehicle to vehicle (V2V) communication even when moving at fast speeds, and vehicle to infrastructure (V2I) like from a vehicles to an RSU in range. 802.11p operates in an ad-hoc mode, where there is no need to establish a basic service set between the two connecting entities because cars can be moving at fast speed and require a short time connection. The building block of the dedicated-short range communication DSRC is the 802.11p. This protocol was deigned from the beginning to support all the intelligent transport systems requirements.

LTE can be used to support V2I applications, but not V2V low delay communication. Roaming can cause problems when connecting two vehicles through eNodeB.

Moreover, the high number of vehicles connections to the LTE BS can cause performance degradation issues. Furthermore, 5G will not completely replace 802.11p until it is widely adopted and its vehicular advantage overcomes that of 802.11p. 5G offers fast data transmission speeds. However, this does not mean that the computation can be sent to the cloud for processing. These data should be processed on vehicles to neglect any additional network or computational delay by decreasing the load on the cloud. 5G can improve the way data are processed locally with faster communication between vehicles.
In this work, we consider the use of 802.11p and LTE because of their current wide availability that can be maintained and enhanced in the future.

2.6 IoT and Fog Computing

Devices with available connection to the internet can be an IoT device. The number of IoT devices are expected to reach 24 billion in 2020 [10]. A sensor that is able to transmit data is considered as IoT devices. These data are considered very valuable for analysis to form what is known as Big Data. Current technologies are not able to support data generated by the new generation of vehicles at a rate of one terabyte per hour [10]. The battery and energy level of consumption are considered one of the major challenges in IoT devices. These devices should serve for a long period with negligible energy consumption to extend its working time. Moreover, sending all these data to the cloud directly is not feasible in terms of energy consumption, and can overwhelm the cloud with requests. For this reason, we have the rise of many networking protocols and the Fog Computing concept by Cisco in 2011. Fog devices are meant to reduce energy consumption on IoT devices, do the processing on the data before sending it to the cloud, and serve IoT devices and users in need of better QoS[3]. In other words,

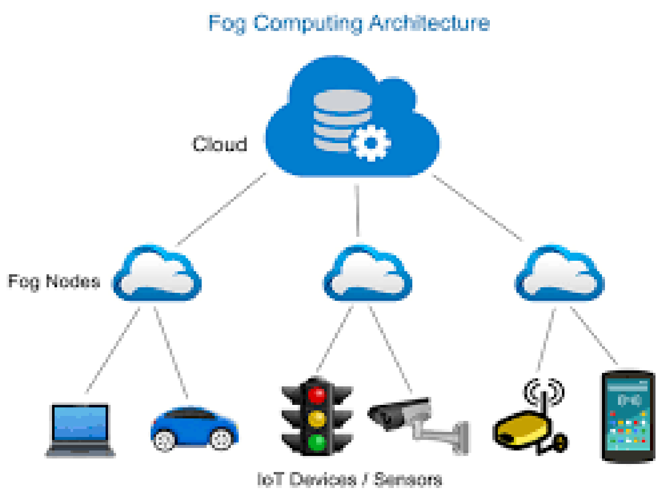


Figure 2.2: Fog Layer Supporting IoT [19]

fog computing (or what is known as edge computing) brings cloud intelligence near the edge, by hosting cloud services on devices or server placed as close as possible to the sensors.

In our work, these fog devices are the OBU and smart devices carried by passengers. At the same time, these are the IoT devices capable of generating a vast amount of data that we can process on the same and other cars using our approach.

2.7 Virtual Machines and Containers

Virtual machines are used to host services on computers without worrying about hardware maintenance costs. A virtual machine uses its copy of the operating system, where each VM is seperated from the other. In vehicular environments, the service requirement changes a lot. The RSU usually host a VM and adapt the services running based on vehicles requests for applications. When using a VM, updating service will either need human intervention or a full download of another VM containing the requested service. Because a VM uses a full copy of the OS, the file to download a

VM image is too big. This will result in delays when trying to download new VMs to update the running services.

On the other hand, containerization technology is becoming more important because of its capability to overcome VMs limitations while maintaining a better performance. Containers are more lightweight than VMs because they share the same OS of the machine they are running on. Besides, containers on a machine can share not only the base OS image but also the bin and lib files associated with the services. The defacto technology that implements containers is Docker [23]. Services run inside containers on the machine. Each container is a running copy of a downloaded image. This image is created using Docker commands and pushed to an image repository. One of the well-known repositories is DockerHub [24]. Furthermore, the size of an OS Docker image is much smaller than any VM instance because containers can integrate with another component on the kernel level of the machine. This does not only add to the performance, but also to the size of the image that should be pushed every time an update of the service is required [7].

In this work, we use Docker as the main containerization technology to offer a fast and lightweight deployment on fog devices that are OBUs and smart devices in our context.

2.8 Containers' Orchestration

Every day, Google pushes more than two million containers; this causes a large number of containers running everywhere without monitoring or management. Different implemented container orchestration technologies solve this problem.

Kubernetes is an orchestration technology implemented by Google for pushing services and managing containers. It is a framework designed to monitor the life cycle

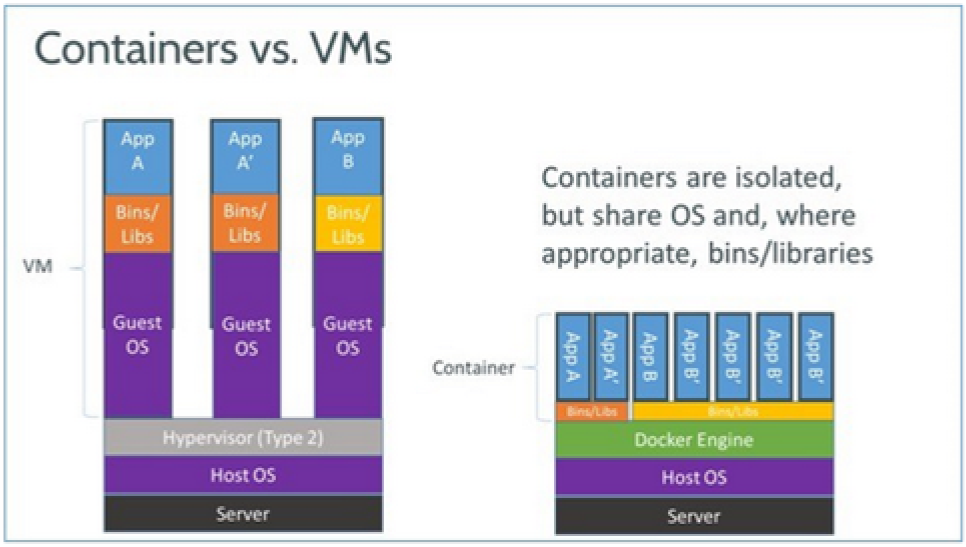


Figure 2.3: Containers vs VMs [29]

of containers through methods that provide predictability, scalability, and high availability [28]. The architecture of Kubernetes is based on clustering the available server that host the containers with a master node or an orchestrator that manages its worker nodes and monitors their performance. A request to push service is sent to the master node in the form of a Yaml file. This service is then pushed to an available worker node. The worker nodes can run multiple pods. A pod resembles a group of containers that are related. A failure in the service leads to an automatic restart and a report sent to the orchestrator. Kubeadm is a Kubernetes utility that provides the ability to create custom Kubernetes cluster formed of heterogeneous devices. Additionally, Kubeadm offers the high cluster availability feature that allows the creation of Kubeadm cluster with multiple orchestrators. In order to run Kubeadm on a machine, there are specific packages that should be installed like Docker, Kubelet, Kubectl, and a networking solution like Flannel to let pods communicate together.

Another containers orchestration technology is Docker Swarm by Docker. This technology offers the same functionality as Kubernetes, but it is much simpler to use.

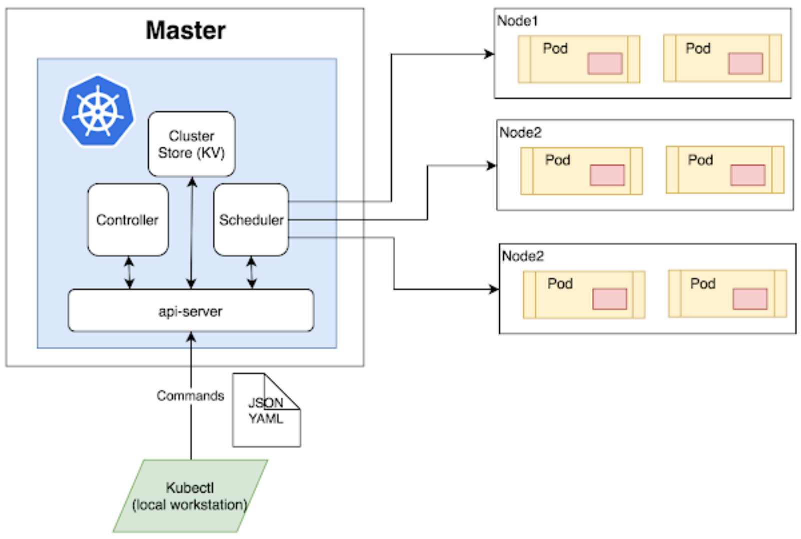


Figure 2.4: Kubeadm Orchestration Architecture [28]

However, Kuberenetes is a more advanced tool that supports auto-scaling and cluster customization features which are not offered by Swarm out of the box.

In this work, we cluster the vehicles to form a Kubeadm cluster with an orchestrator that pushes and manages containerized services life-cycle.

2.9 Multi-Objective Optimization Problem

In an optimization problem, the objective is to find the optimal solution from a set of possible ones. This includes minimizing or maximizing a single or a set of objective functions. These functions are evaluated based on a set of permitted inputs that are selected based on a set of constraints that are defined. A single optimization problem contains one objective function that should be maximized or minimized to retrieve the best solution. Function F is an example of a single objective optimization problem. The aim is to maximize the function $x3+2x2+1$ where: $x \in \mathbb{R}$

$F = max(x3+2x2+1)$

In contrast of the single objective optimization, a multi-objective optimization tries to minimize or maximize a set of objective functions at one time where it is hard to decide on a solution that best optimizes all the provided functions. Function G below is a multi-objective optimization function that finds the value of x that minimizes $g_1, g_2, ..., g_k$ such that k $\geqslant 2 objective functions, and X is the set of decision variables:$

G = min (g$_1(x), g_2(x)...g_k(x))$ where $x \in X$

The multi-objective optimization problem finds the optimal solution based on a trade-off between all the objective functions. Sometimes, no solution optimizes the evaluation of all the functions. Therefore, these functions are considered as conflicting operations. Here comes the concept of a Pareto solution where the improvement of one objective will affect at least one other objective. Therefore, solving a multi-objective optimization problem is not as straightforward as solving a single objective because there might be more than one optimal Pareto solution [6]. Accordingly, several methodologies were defined to solve multi-objective optimization problems such as scalarization, lexicographic, and evolutionary algorithms. Scalarization aims to transform a multi-objective optimization problem to a single objective problem where its solution is the Pareto solution for the multi-objective problem. The lexicographic method uses the decision maker to order the objective functions by importance and solve each one starting from the highest rank. The evolutionary algorithm uses the Pareto-ranking scheme to solve the problem by generating all Pareto optimal solutions in one run so that each solution is better than any solution not within the front.

2.10 Genetic and Memetic Algorithms

When the problem faced is an NP-hard problem, there is no algorithm capable of solving it. However, in another issue, there is a solution for the problem that an algorithm can reach, but this requires a lot of processing and might take a long time to reach a solution or a set of them. Hence, heuristic algorithms are the escaping solution to such a problem where speed is favored over accuracy and completeness of the solution. When an ideal solution fails to provide an exact answer or is too slow to reach it, heuristic approaches, based in randomness, can give an approximation of the solution or near-optimal solutions. Genetic algorithms are meta-heuristics and a class of evolutionary algorithms that are adapted to solve hard search-problems by emulating natural selection. Genetic algorithms build a set of individuals and try to enhance them by generating more fitted solutions using natural evaluation techniques such as selection, crossover, and mutation. An individual having a high fitness value is more likely to be selected for refinement in the new generation. In the crossover, the algorithm select two parents from the previous generation to create a new child having a higher average fitness in the new offspring. The mutation is meant to diversify the generation by altering one to more genes via mutating some children.

The genetic algorithm works as follows: a set of individuals is randomly selected first. These individuals are then evaluated using the objective functions provided. The fitness value is determined based on the criteria of minimization or maximization of the objective functions. Selection and mutation are applied to the fittest solution to reproduce a new population. Individuals having better fitness value are most likely to be considered in the offspring. A new offspring is then reproduced by placing the selected individuals in a mating pool and applying crossovers between two parents. The two parents might undergo a mutation process before a crossover to maintain diversity in the population. The number of iterations the individual undergoes is specified by stopping criteria like a maximum number of iterations, reaching a desired optimal solution, or a manual intervention by a decision maker. Consequently, a better set of optimal solutions is generated to solve the given problem. Genetic algorithms are adopted in many research problems to solve sophisticated search, scheduling, and optimization problems benefiting from its evolutionary process [5].

On the other hand, genetic algorithm is well known for their premature convergence. This prevents such algorithm from exploring and searching through other solutions and might get stuck in a local optimal. Memetic algorithms are built on top of the genetic algorithms to reduce the prospect of premature convergence. This is done using a local search algorithm that allows a broader exploration of possible solutions that can lead to better results. Memetic algorithms are proven to generate more accurate and improved results than genetic algorithms [21]. In this work, we build a Memetic algorithm that combines genetic with probabilistic local search[22] to solve our vehicular container placement (VCP) NP-hard problem.

2.11 Related Work

In this section, we explain the contribution of several related literary works and present their limitations that are thoroughly explored in our experiments later. The areas discussed are related to supporting vehicular applications through the use of the cloud and RSUs as processing power, the use of vehicular nodes as computing infrastructure to serve other vehicles, the use of containerization in a fog-related contexts to serve IoT devices, a related work that proposes the use of hybrid network architecture to keep vehicles connected, and finally a review of a research idea that uses the Memetic algorithm to solve a cloud virtual machine problem which can be mapped to our VCP.

Methods Supporting Vehicular Applications

Several architectures were proposed in the literature to integrate SDN in VANET, such as [32] and [27].

An architecture that supports vehicular devices in providing them with services at any time was proposed in [32]. The problem it tackles is how to maintain a stable connection between vehicles and RSUs even if the topology of the vehicles is dynamically changing. This approach assumes that SDN is placed away on the cloud hosting the controller and a fog orchestrator. The fog orchestrator is responsible for deciding upon the distribution of services on the RSUs, as well as managing and updating them based on the user's needs. Vehicles are treated as switches, where they can communicate to reach their nearest RSU. We believe that this architecture is not applicable in all real-life scenarios and thus cannot be considered as a good solution for VANET environments. Indeed, SDN being on the cloud is an issue by itself where it is subject to networking delays. Moreover, Controller on the cloud is subject to a single point of failure and cannot handle all user's requests because of the humongous number of vehicles and the fast-changing network topology while randomly moving. In addition, this work fails to provide any real-life simulations to prove their claim.

The authors in [27] proposed the idea of RSU clouds that are controlled by an SDN controller to support the internet of vehicles. Services on RSUs are running inside virtual machines. When a user connects to an RSU to get services, it asks the controller for a route to the service if running on any other RSU in its area, if not, the RSU has to update the service running on its VM to start serving the user. It is not feasible to update services on VMs especially in a dynamic environment like VANET because human intervention is needed to update the service manually, or a new VM has to be downloaded on the RSU again. This behavior repeats when the user moves from one cloud of RSUs to another. Another limitation is the load that can be caused on the SDN controller in terms of computational delay. Hence, in some scenarios where the user is moving at a fast speed, when the RSU range of coverage is small, or when the virtual image size containing the service is big, the user might leave the cloud of RSUs without being able to reach the requesting service hosted on that RSU. Time-sensitive applications malfunctions under any of the mentioned circumstances.

Volunteering Vehicular Fogs as Infrastructure

The authors in [31] proposed the use of vehicular fogs as infrastructure named by vehicular fog computing (VFC) where a central main fog covering a shopping mall can benefit from volunteering cars present in the parking area. This paper was limited to the use of known resources such as cars' computation power only, and the authors did not mention what technology they counted on to push and run services on newly joining fog volunteers.

Authors in [14] proved an elaboration on the ability to use vehicular nodes as infrastructure to build the fog environment. They illustrated the ability to use moving and parked vehicles as infrastructure to support moving users using VFC. This paper is limited to proposing a new idea of having vehicles on the road as infrastructure to build the fog environment for computation and communication purposes without any knowledge of the underlying technology of how the services are pushed to cars, managed/orchestrated, and distributed on them based on the resources requirements and available capacities.

In [18], the authors make use of the vehicular infrastructure by forming vehicular cloud on-demand counting on the RSUs present on the roads. Finding a Star on the road is the key idea. This star is the one offering the computation and storage power while moving on the road. This information is published by the nearest RSU so that any user can reserve the star's resources to host its requested services. The idea of profit is added where every star offers its resources for a time asking for money in return. If the star moves out of range of the RSU, the requesting user will lose connection with the service. In their proposed architecture, they try to find the best match of a star node to the user requirement by looking at its available resources, distance, and time to serve inside this VANET. This work lacks a method of recovery when the star moves out of the VANET. Moreover, the work does not consider the feasibility and time required to push the required services on the star.

Fog Containerization

None of the available work in the literature considers the use of containerization technology to host services on vehicles. However, many others considered the use of containers in the context of fog devices to serve IoT having statically fixed locations. In this section, we discuss some of these contributions.

The authors in [2] prove the potential of using Docker technology to run containers on fog devices with the ability to adjust services hosted whenever needed. The authors in [13] proposed a framework to cover the limitations present by the fog orchestrator technology after comparing them with the help of the OpenIotFog toolkit. To build their framework, they used Docker Swarm as an orchestration technology. This work was limited to having the fog running on the device generating data itself and requesting the services it needs rather than considering limited resource devices that should have a serving fog nearby. The difference is that our framework can let the fog run on-demand on any vehicle anywhere and anytime whenever needed without user intervention.

A model was proposed by the authors in [12] where dynamic deployment of services on helper nodes of the main server using Docker is possible. They called these helper nodes ”fogs”. So it is feasible to remove, add, stop, and run any service on a physically known fog anytime.
In the recent work done in [20], the authors focused on the ability to use lightweight Docker containerization technology to support service provisioning over IOT devices. Their main contribution is to show how lightweight containerization technology can manage IOT resources by hosting services on them.

Vehicular Hybrid Network Architecture

A promising work done by the authors in [33], that uses 802.11p and LTE in hybrid architecture to keep the cluster connected all the time with the help of the e-NodeBs even if topology changes. The master node uses two network adapter. LTE to connect to other vehicles outside its range, and 802.11p to connect to other vehicles in its cluster. Even-though all work is counting on services being installed on cars (fogs) when needed, and it is still unclear in what form each computing service is installed or realized by requesting vehicles. In this thesis, we are going to solve the aforementioned problems of having a stable vehicular infrastructure to serve cars requesting time-sensitive applications and fast sensed data processing by proposing a framework that benefits from the containerization, micro-services and orchestration technologies, and adopt a variation of the Vehicular Multi-hop algorithm for Stable Clustering (VMaSC) [33]

Vehicular Container Placement Problem

Lopez-Pires et al. [16] proposed an interactive Memetic Algorithm to solve the proposed multi-objective formulation of the virtual machine placement problem. The goal is to find the optimal distribution of VMs on corresponding available hosts concerning the conflicting objective functions. This problem can be mapped to our vehicular container placement problem by considering the VMs as containers, and the available hosts to run the VMs as the vehicular fog devices. Vehicular fog devices have the mobility feature that is taken into consideration during the formulation of our placement problem.

2.12 Conclusion

In this section, we explain the importance of supporting vehicular applications with some examples. Then, we present background information about some terminologies, technologies, and algorithms needed in our proposed architecture, such as OBUs, RSU, RSUC, IoT, fog, LTE, 5G, 802.11p, containerization, containers orchestration, Multi-Objective Optimization problem, and Memetic Algorithm. Finally, we review some related work showing limitations that we aim to solve in our approach, and other contributions that we used to build our architecture. In summary, the main limitations that we try to overcome in our approach are:

- RSUs are limited to supporting vehicular applications because of their constraint range of coverage, limited resources in case of congested traffic, and heavy VM deployment. All of these factors can lead to non availability of the service and poor user experience, especially that real time vehicular applications require low networking delay and high processing power.

- VMs are heavy on the machine, therefore updating a service on it might need user's intervention or another deployment of a new VM having the newly requested service hosted.
- Hosting a service on one OBU is a problem because of the constraint resources on such devices.
- Vehicles follow random patterns, which makes it difficult to cluster vehicles and keep the requesting and serving fogs connected.

To the best of our knowledge, none of the previous work can fully support vehicular applications by hosting containerized micro-services with an orchestration layer and a multi-hybrid network approach. We are the first also to formulate and solve the vehicular container placement problem.

Chapter 3

Kubeadm Vehicular Fog Clustering Using Hybrid Network Architecture

3.1 Introduction

Vehicular applications are emerging at rapid speed and require fast and real-time processing of sensed data generated by cars. As discussed in the previous chapter, RSUs (if available) are a good candidate to serve non-time sensitive applications only. Therefore, there is a need for an architecture capable of serving time-sensitive applications taking into consideration vehicles mobility. To consider the mobility factor, it is a good idea to host services on neighboring cars having the same driving patterns.Vehicles must be connected for the maximum time possible in case they are used as fogs. The connection time should be maximized in order to maintain a high rate of request/response packets delivery. Furthermore, one OBU is not enough to host heavy services requiring a lot of resources. For this reason, OBUs must be clustered and monitored in a Master-Slave approach in order to increase resources capacity, and services should be divided into smaller and lighter micro-services. Another problem arises while trying to initialize the vehicular Kubeadm cluster is the master node election for a group of available OBUs. Also, whenever a failure or disconnection happens in the cluster, it is costly to re-initialize a new cluster while the user is waiting for the service. Therefore, a recovery method must be applied to maintain service availability and cluster connectivity.

In this section, We address the problems as mentioned earlier by elaborating a Kubeadm Vehicular Fog Clustering methodology that consists of OBUs connected using a hybrid network architecture. Our approach uses Docker containerization technology to push micro-services on OBUs when needed. A master election algorithm to improve cluster monitoring and connectivity is presented. We also propose a recovery algorithm to help in maintaining cluster connectivity when possible. Our approach guarantees a long time service availability on vehicular devices initialized on the fly with fast deployment on pre-configured clusters according to users' needs.

The rest of this chapter is organized as follows: Section 3.2 shows the overall proposed architecture. Section 3.3 explains the components of each layer inside the architecture and how they interact together. Section 3.4 presents the master node election algorithm. In section 3.5 we propose our cluster connectivity recovery algorithm. We dedicate section 3.6 to show the experimental setup and analysis. We finally conclude in section 3.7.

3.2 Approach Overview and Architecture

Literature work supporting users through VANET resources suggests him being present in the RSU/VANET range to access its services. Because of the high and random car mobility in VANETs, this approach is only feasible when cars are moving at low speeds. For this reason, [33] suggests the use of cellular technology with 802.11p to support users all the time even if not within the VANET range. To make the architecture more realistic and to avoid wireless collisions, they proposed that only master nodes of clusters are connected to the BS using LTE and to the worker nodes using 802.11p. This minimizes the load on the BS compared to all cars connected to it. In case all cars are connected to the BS, the master and worker nodes have to keep an open connection with the BS which leads to an increase in the complexity of supporting mobility and handovers. In our work, we added another connection layer in case of RSUs presence on the road. If RSUs are covering users and VANETs, the user can connect through them without the need of passing through the BS. For each group of connected RSUs, there is a roadside unit controller (RSUC) that manages them. The purpose of adding this layer of communication is to connect RSUs and minimize the traffic load on the BS. On RSUs, there is a service registry that minimizes micro-services migration costs on nearby fogs. Our architecture also offers a lightweight services migration technique benefiting from the containerization, orchestration, and micro-services technologies. The architecture runs independently of the cloud and is made functional without the need for user intervention. The clusters are formed based on purely vehicular resources on board like OBUs or any computational device capable of hosting one or more micro-services carried by the passengers. Devices are required to support containerization and orchestration technologies: Docker and Kubeadm. Upon user's request of a service, micro-services are deployed on a nearby pre-built Kubeadm cluster.

The architecture shown in Figure 1 is composed of five linked layers: base stations, RSUCs-RSUs, Kubeadm masters, fog devices (Kubeadm worker nodes), and requesting users. This architecture is layered by power and importance from top to bottom. In other words, a layer above has supervision of what is happening on the layers below. Moreover, if a layer below fails to maintain a connection between a fog and a user, the request is escalated to the layer above. A connection between BS-RSUC-RSU is called infrastructure to infrastructure (I2I). The top layer is composed of the base stations (BS) or cellular towers that are connected using Ethernet. In our architecture, the BS tasks are to connect a requesting user to a cluster master node in another BS range and to broadcast a request for service hosting to the underlying RSUs through RSUCs using Ethernet. The RSUC routes requests between RSUs and shares their information including the position of master nodes they have in range, the services hosted by every Kubeadm cluster, and the list of Docker images they have. The third layer is composed of RSUs storing images of users' requested services and tracks the position of the serving Kubeadm master node to connect users with services using 802.11p. This type of connection is called infrastructure to vehicle (I2V). The next layer comprises the Kubeadm master nodes that are dual interface devices able to connect using 802.11p or through the cellular network. Master cars are elected locally and keep vital connection with the BS, RSU (V2I) and fogs (V2V) in the range all the time. The main job of the master node is to decide on the best distribution of services on the set of selected vehicular fogs, and monitor the status of the containers running on them. The master node sends resources offers to the user and waits for approval before pushing services on vehicular fogs. A failure on the master node can be recovered whenever enough resources on the cluster are available to enable the high availability feature implemented in Kubeadm. This feature allows the creation of a secondary master node that replaces the primary one in case of failure. A device in the fog layer can communicate to a nearby car or the RSU using 802.11p only. It is responsible for hosting the assigned micro-services by the master node. The fog also updates its master node with its profile and availability. The communication between the master and fog nodes is kept through exchanging "Hello" packets (This feature is by default provided in Kubeadm, any failure in a worker node is reported to the master). The user initiates the request of hosting a service nearby; he then receives several offers coming from an available nearby cluster. A special decision algorithm running on the user side decides to accept the most suited cluster to maximize his support time. The user can send another request to host other or similar services if not satisfied with the QoS level received by the serving cluster.

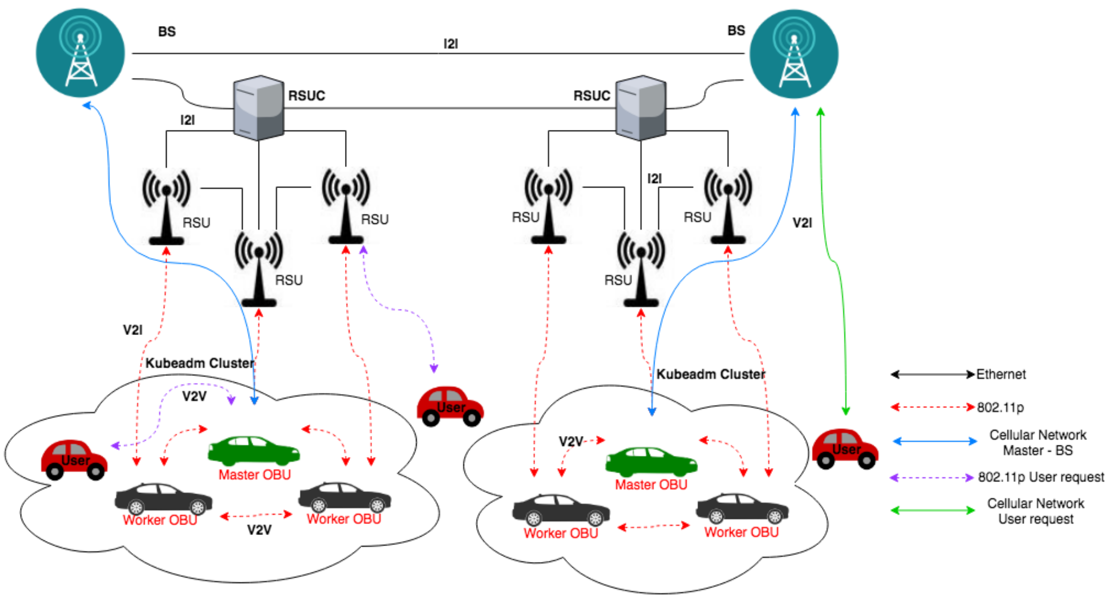


Figure 3.1: Proposed Architecture

3.3 Architecture Components

In this section, we discuss the functionalities of each component per layer in our proposed approach. The BS, RSUC, and RSU communicate together to keep the user connected with the master node by running the Master Manager. Moreover, the RSUC runs the Container Registry Manager responsible for providing Docker images to vehicular fogs. The Kubeadm Master and Fog nodes coordinate to form a stable Kubeadm cluster with high service availability and maximum duration of user support while driving. The master manages fogs and containers running on them through the Fog/Micro-Service Manager, and decides on the best distribution of services on the set of available vehicles in its cluster. Containerization required modules should be already installed on the cluster nodes. The user accepts an offer using the Decision Module for Offer Acceptance and monitors the performance of the serving Kubeadm cluster through the QoS manager. The components of our architecture per layer are depicted in figure 2.

In the following; we provide a detailed explanation of the functionalities within each component.

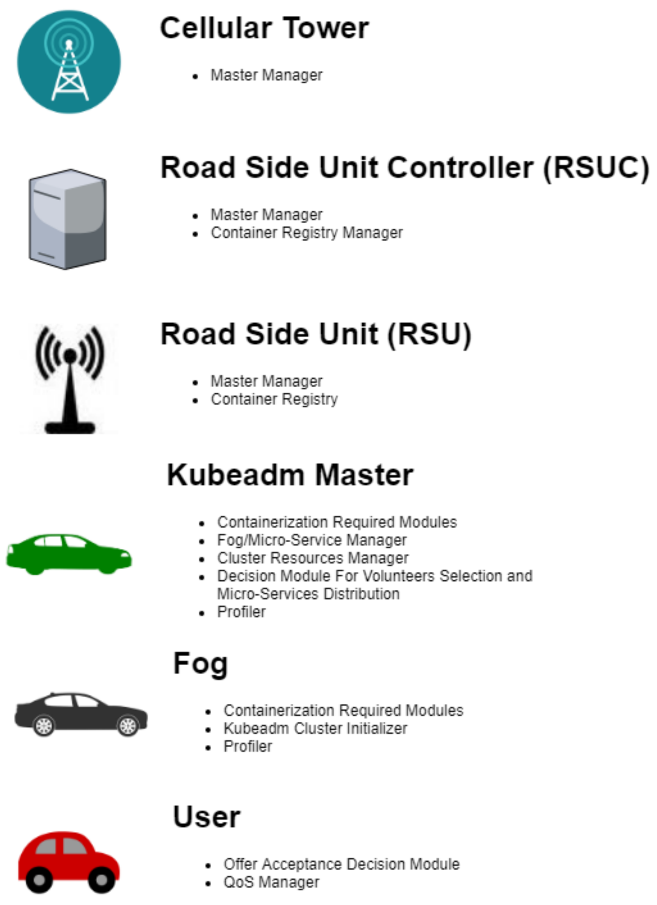

Cellular Tower
Master Manager
Road Side Unit Controller (RSUC)
Master Manager
Container Registry Manager
Road Side Unit (RSU)
Master Manager
Container Registry
Kubeadm Master
Containerization Required Modules
Fog/Micro-Service Manager
Cluster Resources Manager
Decision Module For Volunteers Selection and Micro-Services Distribution
Profiler
Fog
Containerization Required Modules
Kubeadm Cluster Initializer
Profiler
User
Offer Acceptance Decision Module
QoS Manager

Figure 3.2: Node Architecture Per Layer.

Master Manager

The RSU shares the master node information in range with the RSUC and BS. When the master node moves into the coverage range of a new RSU on the road, the old RSU notifies its neighboring RSU of the joining master node. The new RSU, in turn, starts sending ”Hello” packets to the master to ensure a vital connection. The Hello reply messages contain the master node profile. The main functionalities of the master manager are to

Connect User and Master Node

1. In case the module is running on BS: The BS receives a connection request from a user to a master node in two cases, either they are in the range of the BS but cannot communicate through an RSUC/RSU, or they are in the range of two different BSs. This connection uses the wireless cellular network.

2. In case the module is running on RSUC/RSU: the RSUC/RSU is used to connect a user with a master node even if they are under different RSU ranges. The RSU knows if a master node is under its range via periodic updates from the master node profiler. Knowing the master node location within an RSU range minimizes the network load by limiting the broadcast messages into one RSU range. This connection uses 802.11p.

Send Requests and Collect Offers

The BS, RSUC or RSU can receive a request from a user to allocate resources in available Kubeadm VANET clusters. This request is broadcasted to all master nodes either through the BS, or the RSU. The master nodes reply with the available resources.

The BS or RSU, in turn, replies to the user with the available clusters to serve (offers sent from the masters) and waits for an acceptance or rejection decision from him.

Master status monitor and failure recovery

In the case of the master node leaving the cluster, a recovery algorithm (present in section 3.5 - algorithm 5) is triggered by the master manager to maintain the cluster connectivity and service availability.

Container Registry Manager

Usually, the container registry is placed on the cloud for users to push and pull images from. In our architecture, we bring these registries closer to the user to minimize the time of pulling the required micro-services. The RSU has a container image registry holding micro-services requested by users. All RSUs hosting images of services share information about what services they are hosting to their RSUC by replying to the container registry manager's request. In case a fog node asks the RSU for an image it does not have, it asks the RSUC for the location of this service on another RSU. Once found, the RSUs uses Ethernet to transfer the image between them and 802.11p at a maximum rate of 54Mb/s between the RSU and fog.

Containerization Required Modules

Kubeadm is a Kubernetes utility that allows the formation of the cluster on any devices. Kubeadm is a suitable orchestration technology in our architecture because of the various type of devices forming the cluster. Kubeadm orchestration uses the defacto containerization technology, Docker. Kubelet should be installed on the device to ensure communication with the master node and allows different pods to communicate together. Kubelet also helps the master node in checking services health. Kubectl is used to run Kubeadm commands on the master and worker nodes. Docker and Kubeadm with the mentioned dependencies should be installed on the vehicles to form the clusters.

Profiler

This component runs on the master and worker nodes (fog) of a Kubeadm cluster. The module uses GPS to get information about the current vehicle coordinates to calculate the speed and get its location. The profiler also gathers information about the available CPU, memory, and disk space on the vehicle. The route is followed to reach the destination is also provided through the car system. The car might also offer its services for a specific period (can be used for billing purposes), so the time availability is also provided by the profiler.

Fog/Micro-Service Manager

This component runs on the master node and is responsible for managing the worker nodes in the Kubeadm cluster as well as the containers running on them. The purpose of this component is to keep the fog nodes available and to overcome any physical or service failures. The main functionalities can be described as follows:

Keep connection alive

Because of the unpredicted random behavior of cars on the road, a connection has to be checked periodically between fogs and master nodes. Similar to the level of RSU - Master, Hello packets are exchanged periodically containing the current profile of the fog extracted from the profiler module. Connection checks functionality is implemented in Kubeadm.

Assign services to fogs

After getting the offer approval from the user, the master assign services to fogs based on the output of the vehicles selection and services assignment model. The master node sends a pull command containing the list of micro-services to be installed on each node. Each worker node either directly communicates with the RSU if possible, or uses the master node to reach the RSU to download a copy of the Docker image using 802.11p.

Fog status monitor and failure recovery

This functionality is triggered in two cases. Either a container on the fog stopped running and failed to restart, or the fog suddenly went out of the cluster. In this case, the master calls the recovery algorithm. This algorithm is discussed in details in section 3.6, algorithm 4.

Load balancing on fogs

The master node is also responsible for monitoring the load of requests on fog devices. When needed, the master node either creates more copies of the overloaded containers on the same machine or another one in its cluster. This functionality is already provided by Kubeadm to use.

Cluster Resources Manager

After receiving a service installment request from a user, the master calls the Resources Manager component to get a matrix containing the resources available and if they meet

the user requested service requirement, the maximum time availability of the cluster for the given resources requirement, cluster speed, and maximum destination reached by all nodes in the current cluster. This information is received from the decision module to select vehicles and assign services to form an offer message to be sent as a reply to the user. The user based on this data can accept or reject the offer.
The user also informs the master node if the services are not in use anymore to release the allocated resources.

Decision Module For Vehicles Selection and Micro-Services Distribution

The input to this module is a list of services with CPU, memory, disk requirements, and a set of available cars with the CPU, memory and disk resources offered, an average speed, the route followed, the destination, the time availability, and the elected orchestrator. Trying to find a linear solution to this problem is impossible. The problem can be reduced to the bin packing problem to prove it is NP-hard. In this work, we formulate this vehicular container placement problem as a multi-objective optimization problem, where we can find an optimal selection and distribution using heuristics. The problem formulation and solution are explained in Chapter Four.

Kubeadm Cluster Initializer

Kubeadm cluster initialization contains the time to initialize a master node and the time to let worker nodes join the cluster and have all required pods and modules installed. We avoid this initialization cost by making the cluster ready beforehand. For any group of available vehicles, we create the cluster and make it ready to host services. The master node is elected locally by cars based on some parameters. Some issues should be taking into consideration while running the cluster and maintaining its connectivity/availability such as the state transition of the nodes and the possibility of merging master nodes.

Vehicles States

Inspired from work in [33], we define four different states for the vehicles as follows:

1. Undecided: The car does not participate in any cluster. This scenario can also happen when the node failed to connect to the master node.

2. Master: The car becomes a Kubeadm master.

3. Fog: The car becomes a Kubeadm worker.

4. Isolated: The car is a master node but without fogs.

State Transition

Because of the highly dynamic and mobile nature of the vehicle, they transition between states a lot. The possibilities are: An Undecided car can decide to join a Kubeadm cluster and offer some of its resources, so the node becomes a fog. If the car was a master node but lost all the worker nodes, it becomes undecided and vice versa. A fog device can become a master node when it is elected before the master leaves the cluster. A master node can become a fog as well in case of merging masters. A cluster can lose all of its worker nodes in case of the master changing direction or going out of the worker nodes range. In this case, the master becomes isolated waiting for worker nodes to join.

Master Election

To have a more stable cluster, the master has to have speed close to the average cluster speed. Master node election is not only based on speed, but also on the time availability and resources offered. Because the master is overloaded with request and has to monitor all worker nodes and containers, the elected node should meet the minimum requirement in terms of available resources. Furthermore, if the master node goes down, a new cluster has to be created, so we also check the time availability while electing the master. A detailed algorithm is discussed in section 3.5.

Cluster Formation

After electing the master node, Kubeadm init command is executed on the master. Kubeadm init outputs a unique token for the cluster to be shared with any node willing to join. The required modules in our architecture are installed on the master node. Every one-hope away the vehicle from the master is asked to join the cluster using Kubeadm join command and the provided token. The architecture modules on the fog layer are also installed on the worker nodes to makes them ready to pull and run services on the fly.

Masters Merging

To avoid clusters interference and for larger services deployments, the master nodes detect potential interference and ask for merging. Worker nodes vote for one of the master nodes that better support them. After a decision is taken, the RSUC updates its database about the new status of the cluster.

Decision Module for Offer Acceptance

A user receives multiple offers from different master nodes. The user has to accept the offer that best supports him to achieve his minimum required QoS for the longest period while moving towards his destination following the predefined path. A hybrid reinforcement, time series machine learning model, is useful in this situation to study the history of user's activities that includes accepting different offers. This is out of the scope of this work. For now, we assume that the user accepts the best offer by default.

QoS Monitor

The user checks the processing and networking delays happening on the serving cluster to estimate the QoS received. If the minimum QoS required by the service is not met, the user releases the resources on the serving cluster and initiates another request to host a service on a closer cluster with higher computation power.

3.4 Architecture Components Interactions

In this section, we give two examples of how the components in our architecture interact together in case of cluster initialization step or a request from the user to push a fog is sent.

Cluster Initialization Setup

When several cars group together and realizes a potential of creating a Kubeadm cluster, the components of our architecture starts interacting together to form that cluster automatically without user intervention as shown in Figure 3.3. In the cluster initialization case, the components start acting as follows:

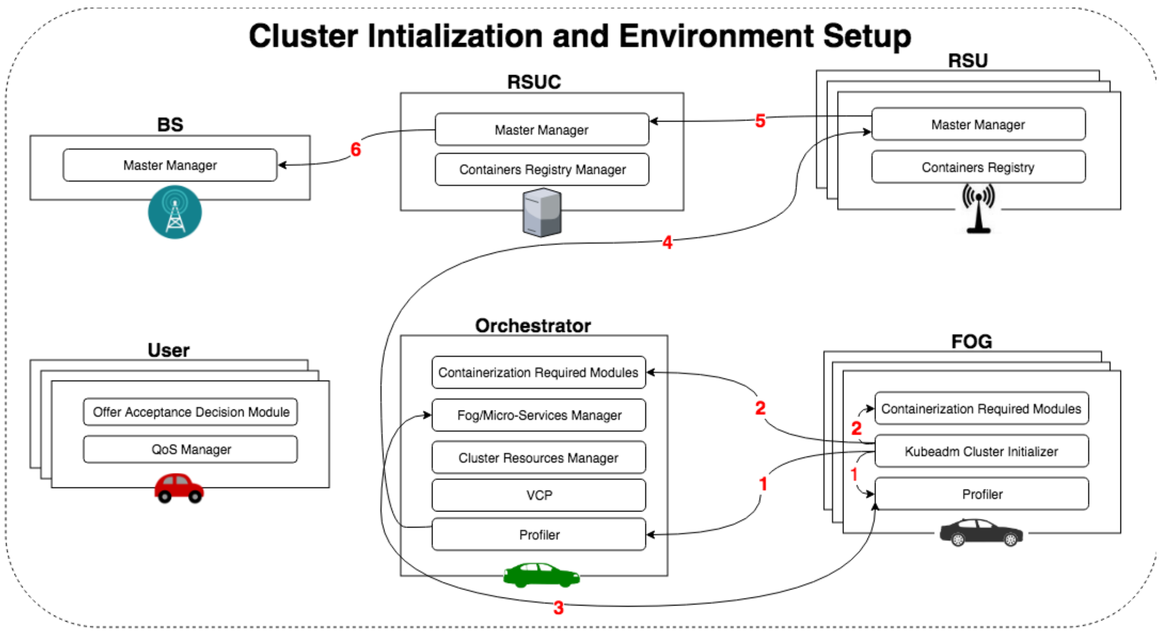

Cluster Intialization and Environment Setup
BS
Master Manager
RSUC
Master Manager
Containers Registry Manager
RSU
Master Manager
Containers Registry
User
Offer Acceptance Decision Module
QoS Manager
Orchestrator
Containerization Required Modules
Fog/Micro-Services Manager
Cluster Resources Manager
VCP
Profiler
FOG
Containerization Required Modules
Kubeadm Cluster Initializer
Profiler
1
2
3
4
5
6

Figure 3.3: Components Interaction While Initializing Kubeadm Cluster

1. Kubeadm cluster initializer extracts the profile information of its node to be considered for the master node election.

2. When the master is elected, the initializer calls the Containerization Required Modules to start the process of initializing the Kubeadm cluster on the master node using Kubeadm Init command. This command generates a unique token to be sent to electors or future fog devices to run the following command: Kubeadm Join token and join the created cluster.

3. After cluster formation, the orchestrator sends asks for profile updates from its worker nodes to keep itself posted with the resource and the time availability.

4. Assuming that the vehicles are connected to an RSU, the information about the

cluster are then shared with that RSU. The master manager of the RSU asks for a periodic update about the cluster profile.

5. The information about the newly created cluster are propagated and keeps on updating from the RSU to its RSUC.

6. The RSUC, in turn, keeps the BS posted with the latest information about the cluster.

In case of the orchestrator connected to the base station, the base station then propagates the information to the neighboring BSs and underlying RSUs. In this case, a user can ask for services hosted on the newly created cluster form any place.

Flow of User's Request To Improve QoS

For the clusters created to host service and start serving users, they wait for a service request from the users. The components interaction in such a scenario from the starting point of receiving a service request from the user until the point of hosting it are shown in Figure 3.4. When the user initiates the request, the components work as follows:

1. The QoS manager on the user side studies the application performance running on its side and decides if an improvement is required. In case the QoS manager wants to improve the QoS level, it initiates a request to host one or group of services to the nearest RSU and already serving orchestrators if any.

2. Assuming that the user and the available cluster are under the coverage range of an RSU. The master manager of this RSU receives the request of the user and looks for the list of available master nodes it has. The RSU then broadcasts the user's request including the service resources and availability requirements to the master nodes available.

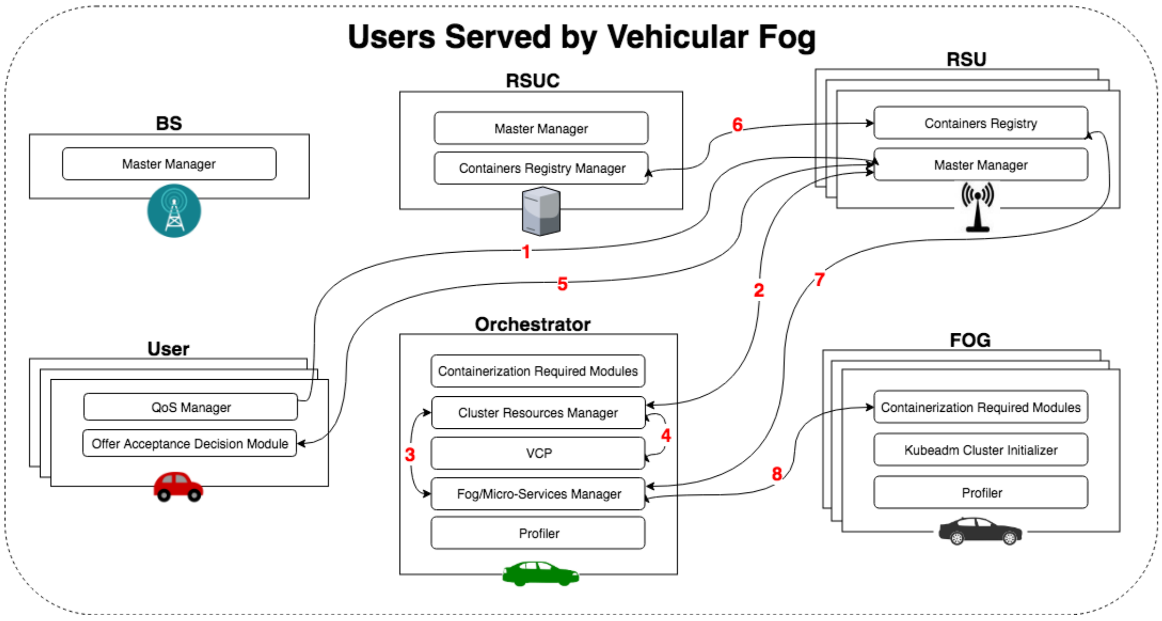

Users Served by Vehicular Fog
BS
Master Manager
RSUC
Master Manager
Containers Registry Manager
RSU
Containers Registry
Master Manager
User
QoS Manager
Offer Acceptance Decision Module
Orchestrator
Containerization Required Modules
Cluster Resources Manager
VCP
Fog/Micro-Services Manager
Profiler
FOG
Containerization Required Modules
Kubeadm Cluster Initializer
Profiler
1
2
3
4
5
6
7
8

Figure 3.4: Components Interaction When a User Requests Services - QoS improvement

3. The master node of the cluster receives the service request and asks the cluster resources manager to calculate its profile and availability.

4. The cluster resources manager collects all the worker's profile information from the fog/micro-services manager and asks the vehicular container placement component (VCP) to prepare the output containing the cluster's resource and time availability. This information is sent back to the user in the form of an offer message.

5. At this point, the RSU informs the orchestrator of the offer acceptance to reserve the resources and sends the offer message to the user. The offer acceptance decision module studies this offer. When the user accepts the offer, the RSU starts preparing the micro-services that should be hosted on the new fogs.

6. In case the RSU does not have the required services, it asks for them from the RSUC's container registry manager.

7. The container registry forms the Yaml file containing the list services to push. This file is then sent to the nominated cluster master node to start pushing those services on its available fogs.

8. The fog/micro-services manager uses the placement matrix generated by the VCP to select the vehicles and push the proper micro-services on each one.

The same scenario can happen in case the user and the master node is connected through the BS. However, the micro-service has to be downloaded from the cloud in the case because RSUs are not available on these roads.

3.5 Masters Election

One certain car approaches each other; a master node election takes place to initialize the Kubeadm master node on the proper device taking into consideration its bandwidth, the speed, and distance concerning the neighboring cars, the serving time, and the resources available to host the required orchestration modules. High bandwidth to ensure reliability with the base station, the speed, distance, and serving time to maintain cluster stability. Implementation of the master election algorithm is provided in algorithms 1, 2, and 3.
Our Kubeadm master election algorithm is inspired by [34]. The election between vehicles is happening locally, and each car calculates its QoS score. Every car receives all QoS scores from neighboring cars and based on the highest; a master node is nominated. The QoS is calculated based on the proportional bandwidth and proportional speed as follows:

$$QoS_j = \frac{BW_j}{N_j} \times \frac{RatioD_j}{RatioAvgS_j} \tag{3.1}$$

where:

j corresponds to a vehicle in the set of available ones.

BW_j is the bandwidth that j can offer.

N_j is the set of neighbors of j.

$RatioD_j$ is the ratio distance Bypassed by j before reaching its destination.

$RatioAvgS_j$ is the average ratio speed of j.

In the first algorithm, the average ratio speed is calculated. Therefore, we used the time spent on the onward and backward trips to get an ideal average speed. The minimum and maximum speed parameters are set based on the speed of the gathered cars. These cars should be grouped for more than a certain period, so the minimum and maximum speed are extracted. When calculating the ratio speed, if the value is less than one (¡ 1), this means that the car is moving with its neighboring cars. This also increases the QoS score of this car (dividing by $RatioAvgS_j$). As for the average distance, the ratio of the residual distance towards the destination is calculated in algorithm 2.
In algorithm 3, the master election process takes place. The algorithm pass through the available vehicles and checks if a particular one meets the resource requirements to be considered for the QoS score calculation. If a vehicle meets the requirement, the QoS is calculated based on equation 3.1; otherwise, the QoS of this vehicle is zero. This vehicles receives all the QoS vehicles from N2 and selects the one having the maximum QoS. The vehicles then broadcast its nomination to all its neighbors. Once the master is elected, the Kubeadm master is initialized, and an ack message is sent to all electors. The election is done on every node to select the highest QoS value from its neighbors, therefore the complexity of this algorithm is $O(log_{N_j})$ where N_j is the number of neighbors of j.

Algorithm 1 Average speed of V_j

```
1: Input:
2:     D: Distance traversed by the car in each direction
3:     t1: Time spent on onward trip
4:     t2: Time spent on backward trip
5: Output: $RatioAvgS_j$
6: $AvgS = 2D/(t1+t2)$
7: foreach j in m
8:     $V_{s_j}$ = random value between $minSpeed_j$ and $maxSpeed_j$ of the road
9:     $RatioAvgS_j = V_{s_j}/AvgS$
```

Algorithm 2 Average Distance of V_j

```
1: Input: $MaxD_j$ = Distance traversed by the car j from the starting point until
2:                     reaching its destination
   Output: $RatioD_j$
3: foreach j in m
4:     $V_{l_j}$ = current position of $V_j$
5:     $ResidualDistance_j = MaxD_J - V_{l_j}$
6:     $RatioD_j = ResidualDistance_j/MaxD_j$
```

Algorithm 3 Kubeadm Master Election Algorithm

```
1: for each $j \in m$ :
2:     if $V_{cpu_j} >= MinRequiredOrchCPU$ and $V_{m_j} >= MinRequiredOrchMem$ and
3:     $V_{d_j} >= MinRequiredOrchDisk$
4:         $RatioAvgS_j$ = Average Speed of $V_j$
5:         $RatioD_j$ = Average Distance of $V_j$
6:         let $N2_j$ = two hops neighbors of j
7:         Calculate $QoS_j$
8:         Broadcast hello message containing $QoS_j$ to $N2_j$
9:     else
10:        $QoS_j = 0$
11:    let k $\in \{N2_j\} \cup \{j\}$ be s.t.
12:        QoS(k) = max{QoS(x) | x$\in \{N2_j\} \cup \{j\}$}
13:    Vote for k through Election Messages
14: for each elected k:
15:     Run Kubeadm Init command to inititalize orchestrator
16:     Broadcast ack messages to N2 of k
```

3.6 Cluster Recovery Algorithm

Vehicles inside a Kubeadm cluster can host one service/micro-service or more. If a serving fog or master leaves the cluster, the user loses its connection with the service. Hence he is back to the default case of requesting the service either from an RSU or from the cloud. To avoid such scenarios from happening, we propose the Kubeadm cluster recovery algorithm.

Because the master node detects any potential cluster communication breaks with any of the fog devices, it can identify the vehicle that will soon leave the cluster by tracking its speed and location. Whenever an alarm is raised, the master node calls the Kubeadm fog recovery.

Furthermore, it is possible for the master node to disconnect from the fog devices and leave the cluster. However, the RSU's track the master availability and can raise the alarm whenever a potential cluster leave is about to happen. The RSU uses the Kubeadm Master recovery algorithm.

In this section, we propose a Kubeadm Fog Recovery Algorithm (algorithm 4), as well as a Kubeadm Master Recovery Algorithm (algorithm 5).

Kubeadm Fog Recovery - Algorithm 4

If a fog vehicle is about to leave the cluster, or a connection is lost, the master node runs Algorithm 4 to recover from any failures that are possible to happen. The master tries to check if another fog in its cluster can host the service of the leaving car. In case there are no resources available on the workers, the master uses its resources to host the service if possible. If not, the master asks the RSU or BS for temporary resources to host the missing service and maintain its availability. As a last resort, the RSU examine the available clusters it has and sends additional offer messages to the user for acceptance. Once the user accepts the offer of the new cluster, the missing service(s) is/are migrated to the new cluster.

Algorithm 4 Kubeadm Fog Recovery Algorithm

```
1: procedure: Fog Recovery
2:     The master Searches for another fog availability by following these checks:
3:         Run VCP to check if another fog node can host the service
4:         Check if the master node can host the service
5:         Check if another cluster or a single vehicle can temporally host
           the service by contacting the RSU or BS
6:     if checks fail, then:
7:         The RSU prepares a backup cluster: generate new offer messages and
           ask for the user acceptance.
8: end procedure
```

Kubeadm Master Recovery - Algorithm 5

In Kubeadm, if the master node leaves its cluster, the cluster goes down. The RSU monitors the behavior of all the underlying master node. If a potential leave for a master node is detected, the RSU calls algorithm 5 to recover from cluster failures before occurring. The algorithm check first if a secondary master node is running to replace the primary. If no secondary is found, the RSU asks for the election algorithm to run locally and elect a new master node in case available resources are found. As a last resort, the RSU collects offers from another available Kubeadm cluster. Once an offer is accepted from the user side, the services are migrated to the new cluster. After that, there is a backup cluster that can replace the original one in case of failures.

3.7 Experiments Showing Architecture Advantage

In the following section, we show a list of experiments to prove the advantage and feasibility of adapting our architecture to support real time vehicular applications. We

Algorithm 5 Kubeadm Master Recovery Algorithm

```
   Procedure: Master Recovery
1:     The RSU searches for another master availability by following these checks:
2:         Check for a running secondary master
3:         Run master election and Check for a fog ability to transition
           for a master state
4:     if checks fail, then:
5:         The RSU prepares a backup cluster: generate new offer messages and
           ask for the user acceptance.
6: end procedure
```

start first by reproducing the work of [27] and prove that if the authors used the containers instead of VMs, they would get better results. We then then show that VMs are not feasible to be hosted on vehicles, however, lightweight containers can. We then move to show the advantage of our approach vs using RSUs to serve vehicular applications. This is done by showing the drawbacks caused by RSUs when SDN delays are high and handovers take place, and the consequences of not having an RSU covering a certain part of the road. At the end, we create an experiment combining all of the above mentioned scenarios, and provoked a worst case scenario when the vehicular cluster in our approach fails to maintain connection. In this case, we show the importance of using our recovery algorithm to maintain serving cluster's connectivity. In some of our experiments, we compare our approach to the work of authors in [27]. This implies that our architecture works better than any other approach using RSUs to host services. We assume in our experiments that whenever the requesting vehicle is not able to access its services from the RSU or neighboring vehicle, the requests of the user are redirected to the cloud. For this reason, simulated the cloud behavior by deploying a tier t2.small VM instance on amazon web service. This instance is running our pinging service.

In our experiments, we use Containernet simulator as a base environment. Our

proposed methodology counts on containerization technology to push and run services on demand for moving cars. Simulating this technology is made possible with the flexibility of the open source simulator Containernet built on top of Mininet to support the use of Containers. Mininet offers the ability to run a full network on one machine. This is done using the notion of processes as hosts where each one can connect and ping the other. Therefore, Wireshark can be used on the machine to see the processes interactions during a simulation. Containernet uses Mininet to allow hosts (processes) to run Docker containers. Because of the need to represent VANETs and simulate them in real life scenarios, we enhanced the integration between Containernet and Mininet-Wifi with the help of SUMO simulator to build real-life scenarios of moving cars. Sumo is a road traffic simulator capable of programming and displaying the movement of cars on any chosen map.
After all, we can build any scenario required to show the advantage of our main approach. Cellular Networks and RSUs are represented in SUMO as well as the moving cars at predefined speeds and routes to follow. The car's behavior is predefined in a Container not script. The networking connection and access to any simulated node's terminal are possible. Automated python scripts using Containernet libraries are used to build experiments discussed in the following Sections. A ping web service is built using the Flask framework and pushed to the container's registry Docker Hub. In this service, the hosting device listens on a certain port waiting for the user's request. The fog then replies with a string message. The aim of this service is to measure the networking delays between users and serving vehicles. The networking delays are enough to show the ability of our approach to hosting services on the fly on fogs.

Experiment 1: Proving Containers Advantage Over Virtual Machines

Proof of containers' performance enhancement over VMs

This experiment aims to show the importance of using containers over VMs on an RSU. The comparison is based upon image files sizes, booting time, and ability to change services smoothly.
Most of the proposed architectures serving vehicular nodes in the literature count on VMs on RSUs to serve vehicles, such as [27] who proposed the creation of a cloud of RSUs. In this section, we reproduced the work of [27] to prove that their approach can perform better if using containers instead of VMs.

Experiment Setup

In this experiment, a core i7 machine with 32 GRAM is used. We fixed the bandwidth rate coming into the machine to 50Mb/s to make a fair comparison. 50Mb/s is the peak download rate for LTE cellular technology. An image of minimal Ubuntu VM having a size of 520 MB holding the flask service is placed on Google Cloud Platform (GCP) to be downloaded on the machine. A container image of size 30MB holds a copy of the same Ubuntu minimal image with the service installed. The virtualized and containerized images are downloaded with the same bandwidth. A moving car with a speed of 10m/s in Sumo is implemented and connected to an RSU offering a 300m range of coverage. Three RSUs are placed next to another on the same road in SUMO. Moreover, an SDN controller is connecting all RSUs. This controller is used to configure routes between RSUs when needed. No delays are added to the controller. Afterward, we let the car send simultaneous HTTP requests to the RSU. After one minute from the simulation time, we let the user request a new service from the RSU.

This service is another copy of the original one.

The response time of each HTTP request sent by the vehicle is recorded. When the RSU receives the first user's request of the web service, it starts downloading the VM vs. the Container in two scenarios. The results are displayed in the form of a signal in Figure 3.5.

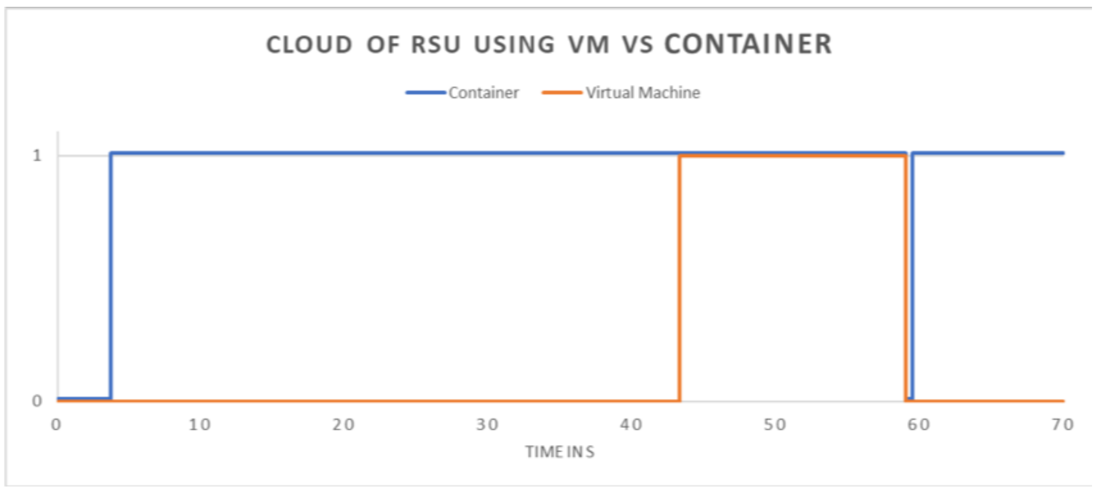


Figure 3.5: Cloud of rsu using VM vs. Container.

Experimental Results

In figure 3, the service reachability is represented as a signal of zero (reachable) or one (not reachable). The x-axis represents the simulation time. The user does not receive a response for the request sent until the signal in the graph is 1. It takes around 4 seconds for the container to install and boot on the RSU. However, around 50 seconds are needed to install and launch the VM Ubuntu minimal instance. The user requests a new service after one minute. As shown in the signal output, it takes less than a second to download the service files in the case of the container and launch it; on the other hand, the RSU has to install another full copy of the VM for it to run the new service. This is another advantage of containers over VM where containers can share the base

image hosted on the machine. There is no need to install a container base image in case the OS of the machine itself is compatible with services requirements. Containers can also share the lib and bin files. On the other hand, each VM uses its copy of the operating system. Therefore, a new copy of the VM has to be downloaded again on the device (without user intervention).
The car is moving at a constant speed of 10m/s. This vehicle leaves the first RSU after the 30s and joins the second one. In this case, the RSU receives the request of the user and tries to connect to the original RSU hosting the service with the help of the SDN controller. The time for the RSU to receive the new configuration of its routing table is not shown in the results of figure 3.5 because there are no delays added to the controller. It took 0.34s for the controller to get back to the RSU with its routing table. However, if delays are considered, which is the case in most real-life scenarios, the approach of [27] suffers from delays reaching the service as proved in the experiment of section 3.7.2.

Feasibility to Host Containers On Vehicles vs. Virtual Machines

In this experiment, we illustrate the irrelevance of pushing virtual machines to vehicles vs. the advantage of pushing containers in terms of image size and booting time.

Experiment Setup In this experiment, we use the SUMO-Containernet testing environment to simulate the advantage of migrating a container on vehicles vs. a VM. The car is moving at a speed of 10 m/s. VM and Docker images use Ubuntu as a base image with the pinging service installed. We assume that the disk space on the vehicles can handle the large image size of the VM. The vehicle tries to download the VM image from the GCP when the simulation starts. The other serving vehicle downloads the Docker image of the service from Docker Hub also when the simulation starts.

To show the effectiveness of using containers on vehicles, we let the user request the service from the serving car running a VM vs. the one using hosting a container. The response time of each HTTP request sent to the serving car is shown in Figure 3.6. If the serving vehicle does not have the requested service, we let the user redirects his requests to the cloud.

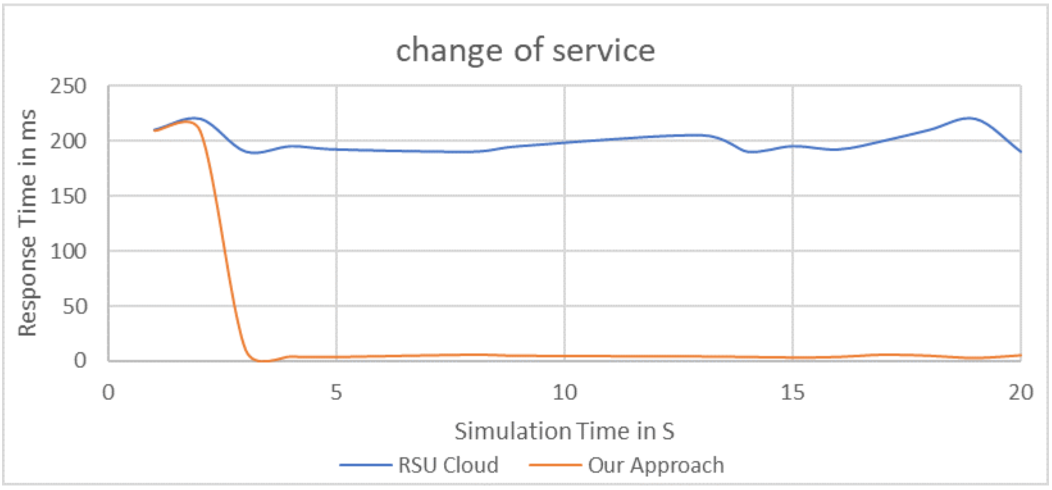


Figure 3.6: Container vs. VM installment on a vehicle.

Experimental Results The x-axis in the graph corresponds to the simulation time in seconds, vs. the response time at different stages or time of the simulation.
A VM instance is being downloaded on the vehicle, and it takes around 50s to get the download. Therefore, all vehicle's requests of the user are served by the cloud in case of pushing the VM. Following our approach, the container contains the Ubuntu instance as a base image with the pinging service. It takes 5s to download the container of size 30MB. After the container is pulled and running, the service is always made available on the running car. We take advantage of Docker technology in our architecture to provide the flexibility of pushing services with the least costs possible. In other words, if the Ubuntu base image were already running on the vehicles, the car would

only pull the required python files to run a new service. Moreover, because OBUs has limited resources in terms of CPU, Memory, and Disk space, it might be impossible to download or run the VM image on the car boarding unit. We also introduced the use of micro-services to overcome the limitation of the resources.

Experiment 2: RSU Handover Costs vs. Hosting Services On Cars

In case of service hosted on RSU1, the car will at one point leave the range of this RSU to join another one. RSU2 should know about the service running on RSU1 to forward user's requests to it. [27] uses the concept of an SDN controller that is responsible for doing the routing table calculations when needed. In this experiment, we prove that using SDN controllers to route requests between RSUs is not feasible in two scenarios that are most likely to occur in real life. First, the controller might be hosted on the cloud, which makes it subject to networking delays because of the distance separating the RSU and the controller. Second, in case the controller is placed near the cloud of RSUs where the traffic on the road is very congested, the number of services requests will increase on the controller, making it subject to computational delays.

Experiment Setup

In our simulator's environment, the car is set to move at a speed of 10m/s from the point it joins the RSU range. Five RSUs are aligned next to each other without any coverage gaps on the same road. The service needed is running on the first RSU only. The car starts moving from the second RSU and tries to keep connected to that service. Based on [27]'s approach, the SDN controller helps in computing the routes, so we connected all RSUs to an SDN controller. We considered a real-life scenario where the controller is busy doing computations causing a delay of 50 ms, and a network delay of around 40ms. These delays are manually provoked during the simulation using bash commands. In our approach, two cars are running together following the same path on the road where they can ping each other all the way. The same service hosted on the RSU is also running on one of the cars to serve the other. The results of this experiment are shown in figure 3.7.

Another experiment is done in the context, but having the car moving at a speed of 20m/s. The results of this second experiment are shown in figure 3.8.

A screenshot of the setup of the experiment is shown in Figure 3.9.

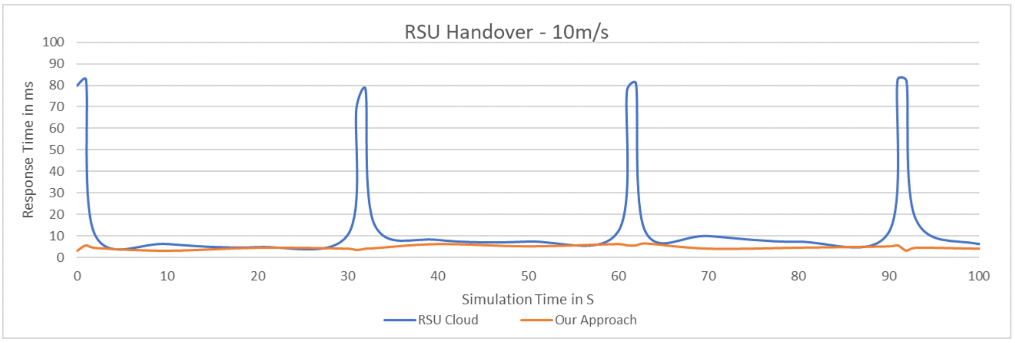


Figure 3.7: RSU Handover Issue vs. Vehicular Fogs. at a speed of 10m/s

Experimental Results

The graph of figures 3.7, 3.8 displays the http response time of requests sent at different time-stamps during the simulation. On the x-axis we have the simulation time vs the response time of the http requests on the y-axis. In the first experiment, When the simulation starts, and HTTP request takes around 80ms to get a response, where the controller computation and networking delays are counted. Afterward, when the RSU receives the HTTP requests, it can route them properly with the help of the controller. This explains the response time drop to 1-5ms. When the car moves to a new RSU, the controller updates the routing table on the RSU, and the same process happens again on the two remaining RSUs. Although these jumps in the response time do not happen

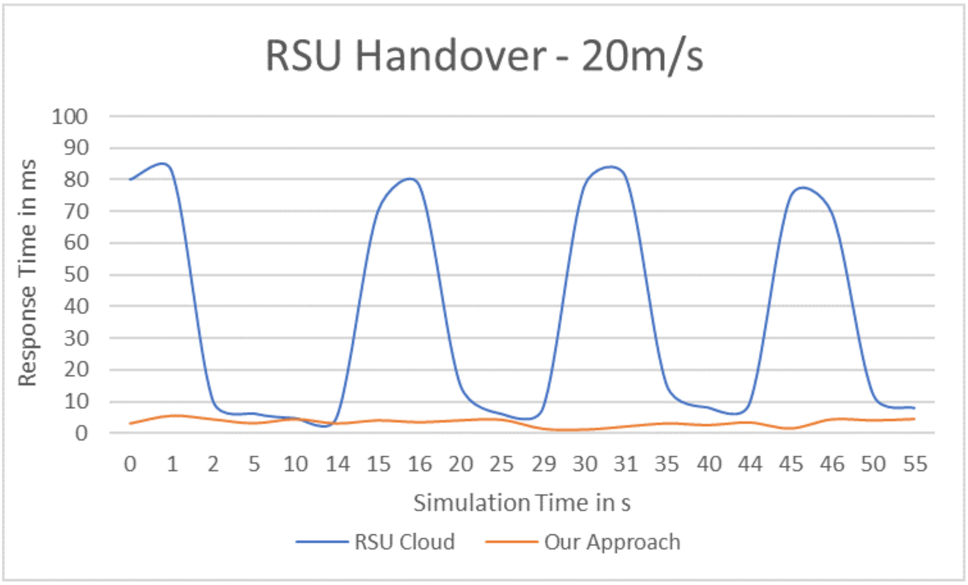
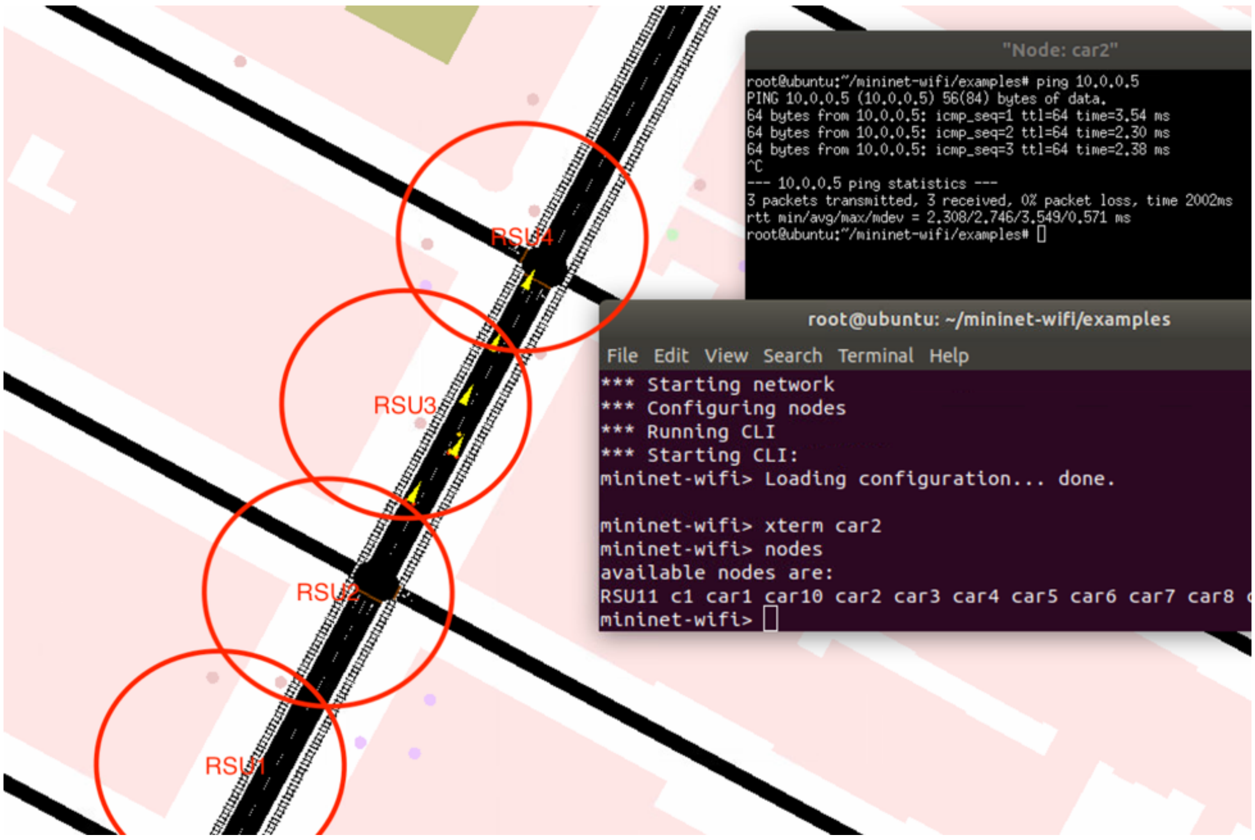


Figure 3.8: RSU Handover Issue vs. Vehicular Fogs. at speed of 20m/s


RSU4
RSU3
RSU2
RSU1
"Node: car2"
root@ubuntu: ~/mininet-wifi/examples
File Edit View Search Terminal Help

```
root@ubuntu:~/mininet-wifi/examples# ping 10.0.0.5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=3.54 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=2.30 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=2.38 ms
^C
--- 10.0.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 2.308/2.746/3.549/0.571 ms
root@ubuntu:~/mininet-wifi/examples# 
```

```
*** Starting network
*** Configuring nodes
*** Running CLI
*** Starting CLI:
mininet-wifi> Loading configuration... done.

mininet-wifi> xterm car2
mininet-wifi> nodes
available nodes are:
RSU11 c1 car1 car10 car2 car3 car4 car5 car6 car7 car8 c
mininet-wifi> 
```

Figure 3.9: Experiment Screenshot

very frequently and might not harm the overall application performance, we can see in the results of figure 3.8 that these jumps occur more frequently because of the car moving at a faster speed causing it to change more RSUs. Therefore, the use of SDN controllers to route the requests between RSUs, if available, is not a good candidate because it is subject to networking and computational delays where the car can move at fast speeds.

In our approach, the best fog is selected and keeps moving in the user's connection range. Because of our proposed network architecture and the ability to push containerized services on the fly, the service is always available on nearby cars even if not in the same range. Our approach achieves a stable small response time which is a critical factor in time-sensitive applications like self-driving cars.

Experiment 3: Drawback of Limited RSU Range of Coverage

In this experiment, we show the drawback of having parts of the roads not covered by RSU's wireless technology (802.11p). In such a case, vehicles will not be able to request their services as usual. In our approach, the serving car is always accessible because of the location tracking and updated routing tables on the RSUs, RSUCs, and base stations.

Experiment Setup

Using the same testing environment, two RSUs are aligned on the road. 300m separates the two RSUs. Two cars are moving together on this road at a speed of 10m/s. Our service is running on the first RSU and in the neighboring car. In this simulation, if the car cannot reach the RSU, the request is redirected to the cloud instance running our service. Services are already installed and running on the RSU and the vehicular fog. The results of this experiment are shown in Figure 3.10.

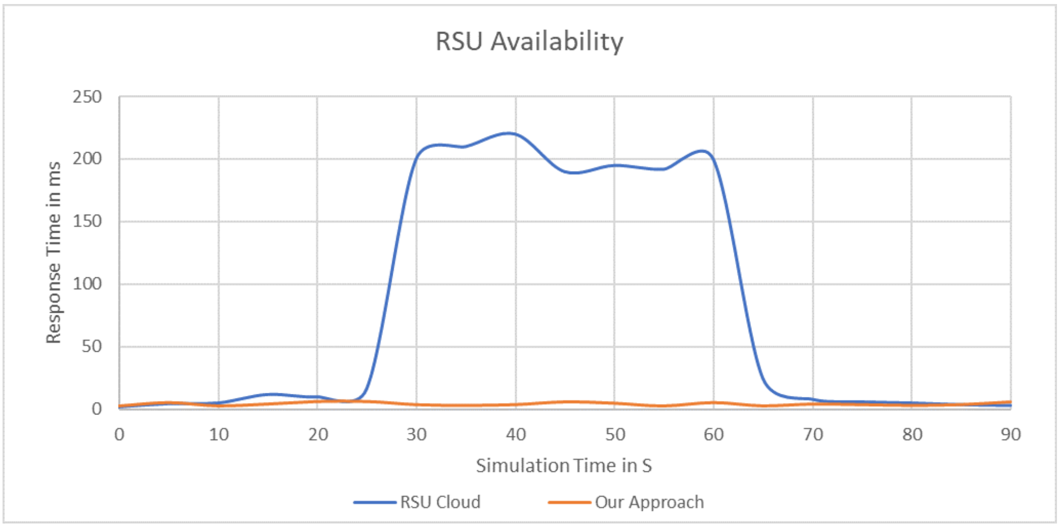


Figure 3.10: RSU limited network coverage drawback vs. vehicular fogs.

Experimental Results

Figure 3.10 presents the response times of HTTP requests sent at different simulation times to the RSU vs. the serving vehicular fog. The response time of user's requests is negligible at the beginning of the simulation because the car is served by the first RSU. After that, the response time reached almost 200ms per request after the 30s of the simulation time because vehicle left the RSU range and the cloud starts serving. In some cases, the vehicles rely on RSU's processing of the generated sensed data by nearby cars. Whenever the RSU is not available, the car application remains idle until joining the next RSU on its way. In our case, the user waits for 10s to join the second RSU and regain the acceptable QoS level. On the other hand, our approach is capable of hosting micro-services on nearby Kubeadm cluster, making the service always available following our multi-layered network architecture. Therefore, as seen in the results of Figure 4.10, our approach achieves a high service availability with low response time because the serving and requesting vehicles are moving together.

Experiment 4: Combined Testing Scenarios Showing Our Approach Advantage

In this experiment, we combine the three previous experiments considering more number of cars having different speeds. This leads to an unstable cluster connection. Therefore, we try to cover a case where the car randomly changes its behavior to show how our framework can react to it. We compare our approach to the use of the cloud of RSUs.

As the second part of this experiment, we show the advantage of using the recovery algorithm in the case where the cluster loses its connectivity.

Experiment Setup

Four cars are installed on the road following the same path. Also, five RSUs are arranged next to each other on that road. All RSUs are aligned in a way that covers all parts of the road, except RSU3 and RSU4 where we have a gap of 300m in between. All cars start moving at a speed of 10 m/s. V1, V2, V3, and V4 starts moving at time 5s, 10s, 5s, and 0s respectively, creating a distance of 50m between every pair of cars. V1 is the user who starts requesting the service after 5s of the simulation time. We assumed that V3 is elected as the master node of the cluster V2, V3, V4, and that the Kubeadm cluster is already initialized on them, given that all vehicles have the same resources capabilities. At 5s, the RSU starts pulling the VM containing the flask service. At the same time, V4 starts pulling the same service from Docker Hub. After 170s, V4 speed is doubled to reach 20m/s. This behavior was provoked to simulate randomness in the vehicle movement. V4 leaves the cluster (range of the master) at 197s. In this part of the experiment, we do not use any recovery algorithm when V4 leaves its cluster. The response time of each ping request sent to RSU vs. V3

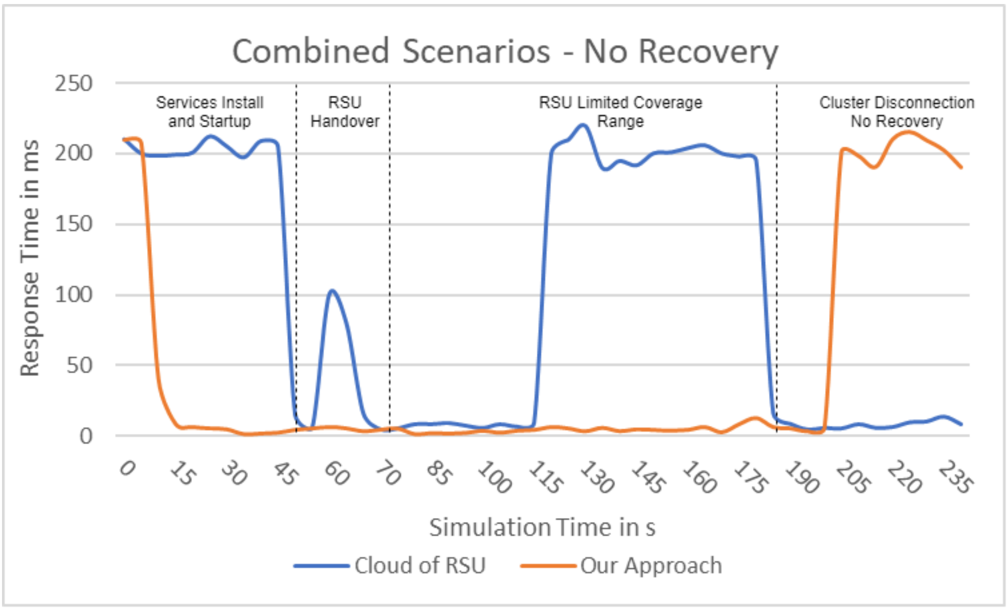


Figure 3.11: Approach Advantage in Combined Scenarios - No Recovery

is recorded and present in the graph of Figure 3.11.

Experimental Results

The results in Figure 3.11 are separated into four parts. Exp1, Exp2, and Exp3 results are reproduced sequentially in this experiment. After 180s of the simulation starting time, V4 doubles its speed to 20m/s causing it to leave the cluster at around 198s. Because V4 is the only running fog in V3's cluster, the service requested by V1 is not available anymore. In this case and Following our approach, V1 has to request the service from the cloud. In the cloud of RSUs approach, the service is available on RSU1 and reachable through RSU5. Moreover, we can notice the jump in the response time; this is because RSU5 is trying to update the routing table from the SDN controller to reach RSU1. Therefore, there is a need for a recovery algorithm that can avoid any failures in our proposed framework.

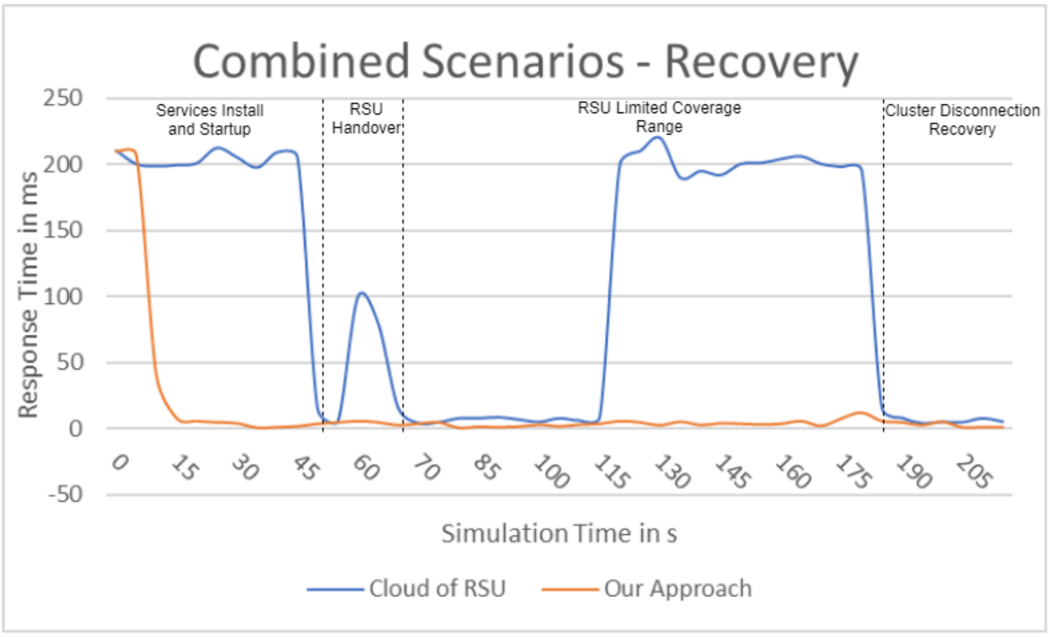


Figure 3.12: Approach Advantage in Combined Scenarios - Recovery

Recovery Algorithm

In the second part of this experiment, the recovery algorithm is Installed to run on V3 to avoid any cluster connectivity or physical failure. We reproduced the experiment, and we can maintain the cluster connectivity as shown in the results of Figure 3.12 (Recovery part). V3 is now able to push the service to V2 before the current running fog (V4) leaves the cluster. A similar scenario can happen in case of a master node failure by running the recovery algorithm on the RSU.

In the combined experiments, our approach is compared to the RSU cloud approach only; however, this comparison applies to all approaches trying to serve vehicular applications using RSUs and virtual machines. In table 3.1, we summarize all advantages of our approach compared to existing ones such as the cloud of RSUs. Adapting the containerization technique allowed us to download and update services faster. Our approach does not consider the use of SDNs because of the timely low cost reporting between master nodes and RSUs. In addition, the use of our proposed hybrid networking architecture makes it possible to users and vehicular fogs to keep connected. The recovery algorithm plays a vital role here by avoiding any potential clusters failures.

Table 3.1: Performance Comparison Between Our Approach vs Cloud of RSUs

Compare Based On	Our Approach	RSU Cloud
Time to download service	low	high
Time to update service	low	hight
SDN Delays	None	Network and Computation
Service Availability	Always	Only within RSU range
Push service again	Use same base OS	Download VM again

3.8 Conclusion

In this chapter, we addressed the problems related to the performance issues when serving vehicles through RSUs running virtual machines. We introduced an On-Demand service deployment on Kubeadm vehicular fog clusters using vehicular resources only. We also adopted a multi-layered hybrid networking architecture capable of maintaining a stable connection between the user and serving fog vehicles. A description of the components of the architecture is discussed. A cluster election algorithm to elect Kubeadm orchestrators is proposed as well as the recovery algorithm that maintains the cluster connectivity for longer service availability and user support. Through our approach, we were able to push services on-demand on vehicles and greatly improve the response time, service availability and cluster maintainability through time
A testing environment was built using Containernet and SUMO simulators. Promising results were shown through a series of deducted experiments studying the feasibility and advantage of adapting our approach over other existing ones. More than 90% improvement in image size and booting time is shown to be achieved when using con-

tainerization technology for pulling and running services. RSUs limitations such as handover and range of coverage are solved in our approach because of the ability to host services on vehicles by forming a cluster and electing the right orchestrator. An improvement in cluster connectivity is also illustrated in the last experiment by running our recovery algorithm.

Chapter 4

Vehicular Container Placement

4.1 Introduction

Vehicles in VANET tend to change their speed randomly, making it a challenge to maintain cluster stability. In our architecture, vehicles are the primary source of computational power including any devices on board to host micro-services. Furthermore, fog nodes are asked to host different services based on users request, which makes it another challenge of mapping each micro-service to the proper car. Finding the optimal migration of micro-services to available cars is an NP-hard problem. The objectives of getting the optimal solution of this problem are to maximize the cluster stability/connectivity or serving time, maximize the connection time between micro-services, maximize the number of pushed services, and minimize the number of active vehicles. This is done after checking if the solution meets the given constraints.

In this section, we define the Vehicular Container Placement Problem (VCP) and prove it is NP-Hard. We mathematically formulate our problem by identifying the input, output, constraints, and the objective functions for our multi-objective optimization problem. A table of notations is provided in table 1.

Table 4.1: Table of Notations

Variable	Description
m	Set of available vehicles
n	Set of requested services
$orch_j$	Kubeadm master node of V_j
E$_{id}$	Number of micro-services of service having id
S	Set of services
V	Set of vehicles
L	Set of serving vehicles
R	wireless coverage range in 802.11p
S_{id}^{i}	Micro-service i of service id
C_{id}^{i}	CPU usage of $S_{id,i}$
M_{id}^{i}	Memory usage of $S_{id,i}$
D_{id}^{i}	Disk usage of $S_{id,i}$
T_{id}	Minimum time to host S_{id}
V_{cpu_j}	CPU available on V_j
V_{m_j}	Memory available on V_j
V_{d_j}	Disk space available on V_j
V_{l_j}	Current position of j as long and lat
V_{S_j}	Current speed of V_js
T_j	Time for V_j to reach its destination
$AvgS_j$	The average speed of worker V_j
U_d	Current location of the user
$T_{cluster}$	Minimum guaranteed cluster time availability without any recovery techniques
$D_{cluster}$	Minimum guaranteed end distance between user and Kubeadm cluster

VCP Problem Definition

In this problem, we have a set of services with different requirements and a set of available vehicles that would potentially host a micro-service or more. The aim is to find the best distribution of these services on the set of available cars taking into account the resources available on them, requirements of services, the networking stability of the moving cluster, the maintenance of attached micro-services, and the proximity from the user during the serving time (required for some applications). This problem is complex to solve and is an NP-hard problem.

By reducing our problem to the Bin-Packing problem, we can prove that our problem is NP-Hard. The traditional bin packing problem is described as follows. Suppose we have a set of objects with different volumes that you need to pack inside a finite number of bins of some capacity or volume. The aim is to try to maximize the total object packed in the bin, and to minimize the number of used bins. This problem can be mapped to our problem as follows. Each bin is our available vehicle that has resources capacities, and the objects are the service images we need to assign for these cars. Our objective is to maximize the number of pushed service while minimizing the number of active fog/vehicles, in addition to other objective functions. Thus our problem is NP-hard.

4.2 VCP Problem Formulation

We aim to optimize the number of pushed services, the number of actively serving vehicles, the stability of the cluster, and the attachment of related micro-services under similar vehicular conditions.

Input Data

For the input data, we have a set of available services S with different requirements, a set of available vehicles V with different offerings, and the mobility parameters to locate the user U.

Vehicle:

The set of vehicles V is represented as a matrix $V \in \mathbb{R}^{n*7}$ that illustrates the seven attributes of each car moving on the road. Each car is described as follows: $V_j = [V_{l_j}, T_j, AvgS_j, V_{cpu_j}, V_{m_j}, V_{d_j}, orch_j]$. For simplicity, we assume that all vehicles forming the cluster are moving in the same direction until reaching their destinations and that they are available all the time during their trip.

Service:

The set of micro-services S, each with a service id, is represented as a matrix $S \in \mathbb{R}^{n*5}$ having five attributes for each micro-service. A row in the matrix represents the microservice requirement as $S^i_{id} = [C^i_{id}, M^i_{id}, D^i_{id}, T_{id}, id]$. We assume that all cars have the same resources specifications, so a service performs the same on any device.

User:

The user is represented as an array of length two that describes the mobility factors of the user and helps VCP in keeping the cluster close to the user when needed (depending on the application). A user U is represented as [$AvgS_u$, T_u] where $AvgS_u$ is the average speed of the user and T_u is the time for the user to reach his destination.

Output data

The optimization solution aims to map each requested service to a vehicle node in its cluster. The output is a binary matrix K_{ij} of size $(n \times n_{id}) \times m$ where $K_{ij} \in 0,1$. i and j represent the service S and the vehicle V that will host S_i respectively. Moreover, an offer message is constructed by K_{ij} and represents the serving time of the cluster for one requested service at a time, as well as the predicted distance from the user. Each component of the offer message is calculated as follows:

Cluster Serving Time

Either the time for the master node to reach the destination is returned, or the time until a worker node goes out of the cluster. This approach is formulated to calculate the minimum guaranteed availability time to serve (without any recovery pattern) such that the distance between this node and master is R given that the master node is also moving.

$$T_{cluster} = min\{\frac{R}{|AvgS_{orch_j} - AvgS_j|} - |V_{p_{orch}} - V_{p_j}|\} \ \forall j \ | L_j = 1 \tag{4.1}$$

If $T_{cluster} > T_{orch} \ then \ T_{cluster} = T_{orch}$

Where V_{p_j} is departure time of the vehicle.

Cluster Distance from user (current and end)

The end distance between the cluster and the user is calculated. In other words, we use the cluster serving time and the master node average speed to get the end position of the master without any recovery techniques. The calculated end orchestrator position is then subtracted from the end user's position to get the distance. This requires geographical coordinates calculations. Therefore, we used **nvector** python library. The end distance between the cluster and requesting user is calculated as follows:

$$
\begin{aligned}
&OrchestratorEndDistance = AvgS_{orch} \times T_{cluster} \\
&UserEndDistance = AvgS_{u} \times T_{cluster} \\
&OrchestratorEndPosition = getNewPosition(V_{l_{orch}}, OrchestratorEndDistance) \\
&UserEndPosition = getNewPosition(V_{l_{u}}, UserEndDistance) \\
&D_{cluster} = calculateDistance(OrchestratorEndPosition, UserEndPosition)
\end{aligned}
\tag{4.2}
$$

Where:

getNewPosition: takes the inital coordinate position and the end distance, and calculates the new position's coordinate.

calculateDistance: takes two positions and calculates the distance between them.

The initial distance between orchestrator and user is calculated using the inital position of the orchestrator and user.
The current and end distance between the user and orchestrator are sent in the offer message. During the offer acceptance decision, the user takes into consideration the orchestrator's proximity to the cluster in case the application is time sensitive.

Constraints

In this section, we present the different constraints that makes a solution feasible.

Resources Limit

The total CPU, Memory, and Disk resources required by the hosted service on a vehicle should be less than its available resources. This constraint can be formulated as follows:

$$\sum_{id=1}^{n}\sum_{i=1}^{nid} C_{id}^{i} \times K_{ij} \leq V_{cpu_j} \tag{4.3}$$

$$\sum_{id=1}^{n}\sum_{i=1}^{nid} M_{id}^{i} \times K_{ij} \leq V_{m_j} \tag{4.4}$$

$$\sum_{id=1}^{n}\sum_{i=1}^{nid} D_{id}^{i} \times K_{ij} \leq V_{d_j} \tag{4.5}$$

$\forall j \in m$, i.e., for all available hosts V_j

Minimum Cluster Serving Time

To guarantee that the cluster can serve the user for a reasonable time, we set a time threshold to be considered before sending an offer message to the user.

$$T_{cluster} > T_{id} \;\; \forall id \in n \tag{4.6}$$

All Coupled micro-services to be hosted

Micro-services should be coupled together to keep them connected in the same cluster and to avoid service availability and delay issues. If all the micro-services composing service requirements cannot be met by a vehicular cluster, any micro-service of this service must not be pushed to the cluster. This constraint is formulated as :

$$\sum_{j=1}^{m}\sum_{i=1}^{n_{id}} K_{ij} = n_{id} \;\; \forall id \in n \tag{4.7}$$

Distance Threshold To User

In case the application requires the fog to be hosted near the user to avoid networking delay, the VCP ensures that the distance between the cluster and the user does not exceed a certain value D already set based on the application need. This cluster-user distance constraint can be formulated as:

$$D_{cluster} < D \tag{4.8}$$

Weights Summation

Each objective function is multiplied with a weight associated with it. All the weights should add up to one. The purpose of these weights is to have a tradeoff in terms of the importance of each objective function. For example, to push as many services as possible no matter what the conditions are, we should set W_{f3} to a value greater than all other weights. In this case, the evaluation of f3 is affecting the sum of the optimization functions more than all other objective functions. It is expressed as follows:

$$W_{f1} + W_{f2} + W_{f3} + W_{f4} = 1 \tag{4.9}$$

Objective Functions

Cluster Lifetime Maximization

Maximizing the cluster lifetime leads to maximizing the user serving time. This objective function aims to maximize the time availability of the kubeadm cluster by selecting fog vehicles that can keep connected to the master node for a long period. We formulate this function by maximizing the minimum time it takes one fog to go out of the cluster as follows:

$$F_1 = max(\sum_{p=1}^{c} T_{cluster} \times W_{f1}) \tag{4.10}$$

Where c is the number of used clusters.

Maximize Micro-Services Connection Time

An alive connection between all micro-services of service is important to ensure service availability. One micro-service does not function without the other. Therefore, in this function, our objective is to host the micro-services in similar mobile conditions on different cars if possible. We make sure that all micro-services are of approximately at the same distance from the master node. This is formulated by minimizing the total sum of distances between the worker nodes hosting the micro-service and their Kubeadm master vehicle for all given ids (services) as follows:

$$F_2 = min(\sum_{j=1}^{m} \sum_{id=1}^{n} \sum_{i=id}^{n_{id}} ((V_{l_{orch_j}} - V_{l_j}) \times K_{ij}) \times W_{f2}) \tag{4.11}$$

Maximize number of pushed services

The aim is to Maximize the number of pushed services to vehicles in the chosen population.

$$F_3 = max((\sum_{j=1}^{m} \sum_{id=1}^{n} \sum_{i=id}^{n_{id}} K_{ij}) \times W_{f3}) \tag{4.12}$$

By maximizing the number of pushed services, we guarantee that all users' requests for services are satisfied and that services are deployed on vehicular fogs.

Minimize number of active vehicles

The aim of this objective function is to minimize the number of active vehicles in order to save initialization time of devices while joining the cluster, using less energy with fewer vehicles running, and a better way to let the orchestrator monitor less number of devices. It is expressed as:

$$F_4 = min((\sum_{j=1}^{m} L_j) \times W_{f4}) \tag{4.13}$$

4.3 Memetic Algorithm To Solve VCP

It is important to get the optimal set of solutions for the container placement problem in a short time. The memetic algorithm is the most suitable solution for such problems. It is built on top of the genetic algorithms. However, in addition to the optimization operators, it has a local optimization (local search) algorithm that can guarantee optimal solutions in early generations [22]. The memetic algorithm proposed in [16] and adapted to solve our optimization problem is illustrated in Algorithm 6.

First, we check that the problem has at least a feasible solution where containers of services can be hosted on the available vehicles before moving to step 2. Next, we initialize a random set of solutions P_0 by randomly assigning images of services to available cars. In step 3, we look at the set of available solutions in P_0, and repair any violation of our constraints (e.g., a service to be hosted on a vehicle with resource consumption greater than available capacity of this vehicle). These violations are repaired in three ways and illustrated in Algorithm 7: (1) Moving containers to other available vehicles in the cluster, (2) adding available vehicles to the list of running ones and move Docker containers to them, and (3) removing containers from the list of services to be pushed. Step 4 of the MA is to apply a probabilistic local search method presented in Algorithm 8 to optimize feasible solutions. If the probability is less than 0.5, we maximize the number of pushed services following line 4. On the other hand, if we have probability >0.5, we minimize the number of available volunteers. This way, we are trying to converge to an optimal solution at early stages using this probabilistic local search. Then the Pareto set approximation is generated at step 5. After the initialization of step 6, standard selection, crossover, and mutation operators are applied, infeasible solutions are repaired, optimization of solutions is done using probabilistic local search, iteration counter is incremented, and finally, the Pareto set is updated if any improvements happened. After that, a new population is selected. This process keeps on iterating until the algorithm reaches a number of iterations. Finally, the fittest set of the solution p_{known} is returned. In this MA, we use the binary tournament for selecting individuals from the population to apply crossover and mutation on them. The crossover operator used is the single point cross-cut. For mutation, each gene is mutated with probability $1/n$ where n is the number of services. This prevents stagnation in a local optimum.

The complexity of this algorithm is divided into four parts as discussed in [25]: The generation of M chromosomes, the crossover, the mutation, and the local search complexity time. Let M and N be the number of chromosomes and number of nodes respectively. The MA starts off using $O(M \times (n-1) \times log(n-1))$ time units to generate the random population. Also let p_c and p_m be the probability of the mutation and crossover respectively. The number of offsprings generated by the crossover uses $O(N \times p_c \times [M \times (N+1)])$, while the ones created by the mutation consumes $O(p_m \times [M \times (N+1)])$ of time units. The local search algorithm consumes $O(n)$. Therefore the combined time complexity of the MA is shown in equation 4.14 (given $p_m = 1/2$)

$$O((M \times (n-1) \times log(n-1)) + (N \times pc \times [M \times (N+1)]) + (1/2 \times [M \times (N+1)] + N)) \tag{4.14}$$

Algorithm 6 Multi-objective memetic algorithm

```
    Data: Set of containers
    Result: Pareto set approximation $p_{known}$
 1: Check if the problem has a solution
 2: Initialize set of solutions $P_0$
 3: $P_0'$=repair infeasible solutions of$P_0$
 4: $P_0''$ = apply local search to solutions of $P_0'$
 5: Update set of non-dominated solutions $p_{known}$ from $P_0'$
 6: $t = 0$
 7: $P_t = P_0''$
 8: While (stopping criterion is not met), do
 9:     $Q_t$ = selection of solutions from $P_t \cup p_{known}$
10:     $Q_t$’ = crossover and mutation of solutions of $Q_t$
11:     $Q_t$” = repair infeasible solutions of $Q_t$’
12:     $Q_t$”’ = apply local search to solutions of $Q_t$”
13:     increment t
14:     Update set of non-dominated solutions $p_{known}$
        from $Q_t$”’
15:     $P_t$ = fitness selection from $P_t \cup Q_t$”’
16: End while
17: Return Pareto set approximation $p_{known}$
```

Algorithm 7 Infeasible solution reparation

```
    Data: Infeasible Solution
    Result: Feasible Solution
 1: $feasible = false$;  $i = 1$
 2: While $i \leq n$ $and$ $feasible = false$ do
 3:     if $it$ $is$ $possible$ then
 4:         move $S_i$ to $V_j'$ $(j \neq j')$
 5:     else
 6:         if $S_j$ does not have priority level
 7:             Remove $S_i$ from list of services to be pushed
 8:         else
 9:             Moving $S_i$ to other available volunteers $V_j$
                 in $P_{known}$
10:         end
11:     end
12: end while
13: return feasible solution
```

Algorithm 8 Probabilistic local search

```
    Data: set of feasible solutions $P_t$'
    Result: set of feasible optimized solutions $P_t$”
 1: Probability: Random value between zero and one
 2: While there are solutions not verified do
 3:     if Probability $< 0.5$ then
 4:         We remove containers placed on $V_j$ and run them
            on $V_j$' if resources available are enough, and then
            assign any unselected service on $V_j$ if resources
            are available after sorting them with priority level
 5:     else
 6:         We assign all services $S_i$ needed to available $V_j$
 7:         devices depending on resources requirement, and
 8:         then we discard all $V_j$ and assign all services $S_i$
 9:         to new set of volunteers $V_j$' that can host them
10:     end
11: end while
12: return set of optimized solutions $P_t$”
```

4.4 VCP Experiments

In the previous experiments, the best selection of volunteers and optimal distribution of services on cars are taken by default. In the following experiments, we build test cases with three scenarios to show the importance of each of our objective functions and their effect in taking the selection and placement decision. The first experiment illustrates the importance of keeping micro-services connected, because loosing one micro-service before the other leads to non-availability of the main service. A decision can be taken with the best distribution of connected micro-services; however, services can be redistributed in a way that maximizes the number of pushes while maintaining the same fitness value of the micro-services connectivity or tolerating a bit of fitness loss. In the second experiment, we show the ability of the VCP to maximize the number of pushed services. while these two objectives are maintained, the service will not be available for a long time if the cluster is about to reach its time availability limit.

Therefore, maximizing cluster lifetime is another objective function considered in our VCP solution. The advantage of this function is studied by the third experiment.
In all of the mentioned experiments, we display the fitness value of each objective function along with the output matrix compared to an adhoc algorithm to place the services. This is done to show the ability of our VCP to maintain an equilibrium between all of the objective functions in one decision.
To develop our scenarios, we used two of the well-known datasets, the Google trace 2011 dataset [26], and another one generated by Mobisim tool [11]. Google trace dataset contains data about micro-services' resources requirements in terms of CPU, memory, and disk, as well as data about the available cluster's resources to host these services. In our case, each node in this cluster is a vehicle. In their data, all nodes have identical specs where the given values are normalized. The mobility conditions of our scenarios are selected from a generated Mobisim dataset. In Mobisim, we can generate a behavior of real vehicles with random directions and speeds on the roads. From this data, we selected the average speed and distance crossed by a group of cars. The services and vehicles datasets are shown in tables 4.2 and 4.3. Table 4.4 shows the combination of available vehicles and services used to build our test cases.
The testing environment is created using Containernet and SUMO simulators (Similar to the experiments of Chapter 3).

Datasets

Vehicles Dataset

Table 4.2: Vehicles Dataset

	Starting Position	Dest Time	AvgS	CPU	Memory	Disk	Dep Time	Orch
V1	(40.740, -73.994)	20	7.5	0.5	0.55	0.6	0	V1
V2	(40.740, -73.994)	5	7.35	0.6	0.6	0.4	1	V1
V3	(40.740, -73.994)	30	25.0	1	1	1	2	V1
V4	(40.743, -73.996)	40	7	1	1	1	1	V4
V5	(40.743, -73.996)	35	7.1	0.7	0.7	0.7	1	V4
V6	(40.743, -73.996)	30	6	0.5	0.5	0.8	0	V6
V7	(40.165, -73.736)	4	3	0.8	0.7	0.4	3	V7
V8	(40.365, -73.756)	100	20	0.2	0.3	0.1	4	V8

Services Dataset

Table 4.3: Services Dataset

Service	ID	CPU	MEM	DISK
S1	1	0.4	0.3	0.5
S2	1	0.1	0.1	0.05
S3	1	0.2	0.2	0.25
S4	1	0.5	.0.4	0.4
S5	2	0.2	0.1	0.25
S6	2	0.3	0.4	0.25
S7	3	0.6	0.5	0.58

User Data

The current location of the user is (40.740513, -73.994568) that depart at time 4 after the simulation starts.

Scenarios

Table 4.4: Scenarios

Scenario	Available Vehicles(s)	Requested Service(s)
Scenario1	V1, V2, V3, V4, V7, V8	S1, S2, S3, S4
Scenario2	V4, V6, V7, V8	S5, S6, S7
Scenario3	V2, V4. V7, V8	S7

Experiment 4: Importance of keeping micro-services connected

All micro-services must reach each other to keep the service available. In this experiment, we show the ability of our VCP algorithm in keeping the microservices connected for the maximum time possible. We compare the performance of the VCP algorithm to a search algorithm that looks into the available vehicles, find the first vehicle close to the user and capable to host the service.

Experiment Setup

Scenario 1 is used in this experiment, where we have to push S1, S2, S3, S4 and we only have V1, V2, V3, V4 available. The car has a network coverage range of 50m, the user is moving with V1 at almost the same average speed during the first 2s and then connects to it through the RSU. We implemented the services in a way that each one pings the other whenever the user sends a request to V1 (selected orchestrator). If all services can reach each other, the user receives its response. The response time is recorded in the graph of Figure 4.1. Whenever a micro-service is not reachable, we represent it as a value of -1 in the graph. This is where the user does not receive any response from V1. We gave scenario one as input to our VCP algorithm, and we compared its results to the ones generated by the simple search algorithm. 5ms of networking delay is added on the link between the user and V4, V5. Concurrent requests are sent to the serving vehicles neglecting the time of initializing the cluster and downloading the service. The x-axis in Figure 4.1 represents the request number, while the y-axis represents the response time of the request sent by the user to test the service availability.

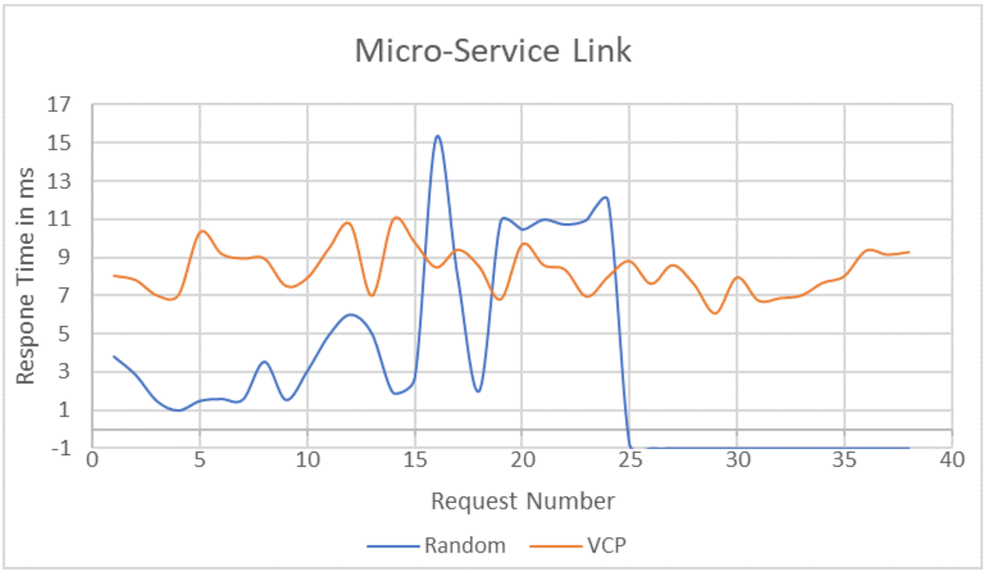


Figure 4.1: Maintain micro-services connection using VCP.

Matrix output for scenario 1 - VCP:

$$K_{ij} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrix output for scenario 1 - Search Algorithm:

$$K_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

	VCP	Search	weights
F1	20	3.8	0.01
F2	0	-4	0.39
F3	4	4	0.3
F4	-2	-2	0.3
Total	2	-0.922	-

Table 4.5: Objective Functions Evaluation - Scenario 1

Experimental Results

As shown in the above output matrices, the VCP algorithm assigns S1, S2 to V5 and S3, S4 to V4, whereas the search algorithm assigns S1, S2 to V1 and S3, S4 to V3. In the simple search case, the user is being served until sending the 25^{th} request where he stops receiving any response. At this time, the response drops to -1 and the service becomes unavailable. This is because V3 is moving at a faster average speed and leaves V1's range after around 3s from the simulation starting time. A jump in the response time for the search algorithm is shown during the 17^{th} request; this is because the user connects to V1 through RSU after a certain time rather than a direct connection (the speed of the user is different than the cluster's speed). VCP tries to push micro-services to cars in a way that maximizes their time connectivity to their orchestrators. Following the VCP decision, the service is made available for the user all the time. We can notice that VCP is causing more delay when the service is available because V4 is far from the user and connected through an RSU from the beginning.

Experiment 5: Importance of maximizing the number of services

We aim to push the maximum number of services on available cars to enhance QoS for all requesting users.

Experiment Setup

In this experiment, we let the user move next to V1 and use scenario two as a testing environment. To show the importance of this objective function, we compare the results of our VCP to the search algorithm. Using Containernet integration with SUMO, we can simulate the behavior of both criteria. A networking delay of 5ms is added on the link between the user and all the vehicles. A comparison of the response time of each HTTP request sent to both services hosted on selected is shown in the two graphs of figure 4.2.

Matrix output for scenario 2 - VCP:

$$K_{ij} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Matrix output for scenario 2 - Search:

$$K_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

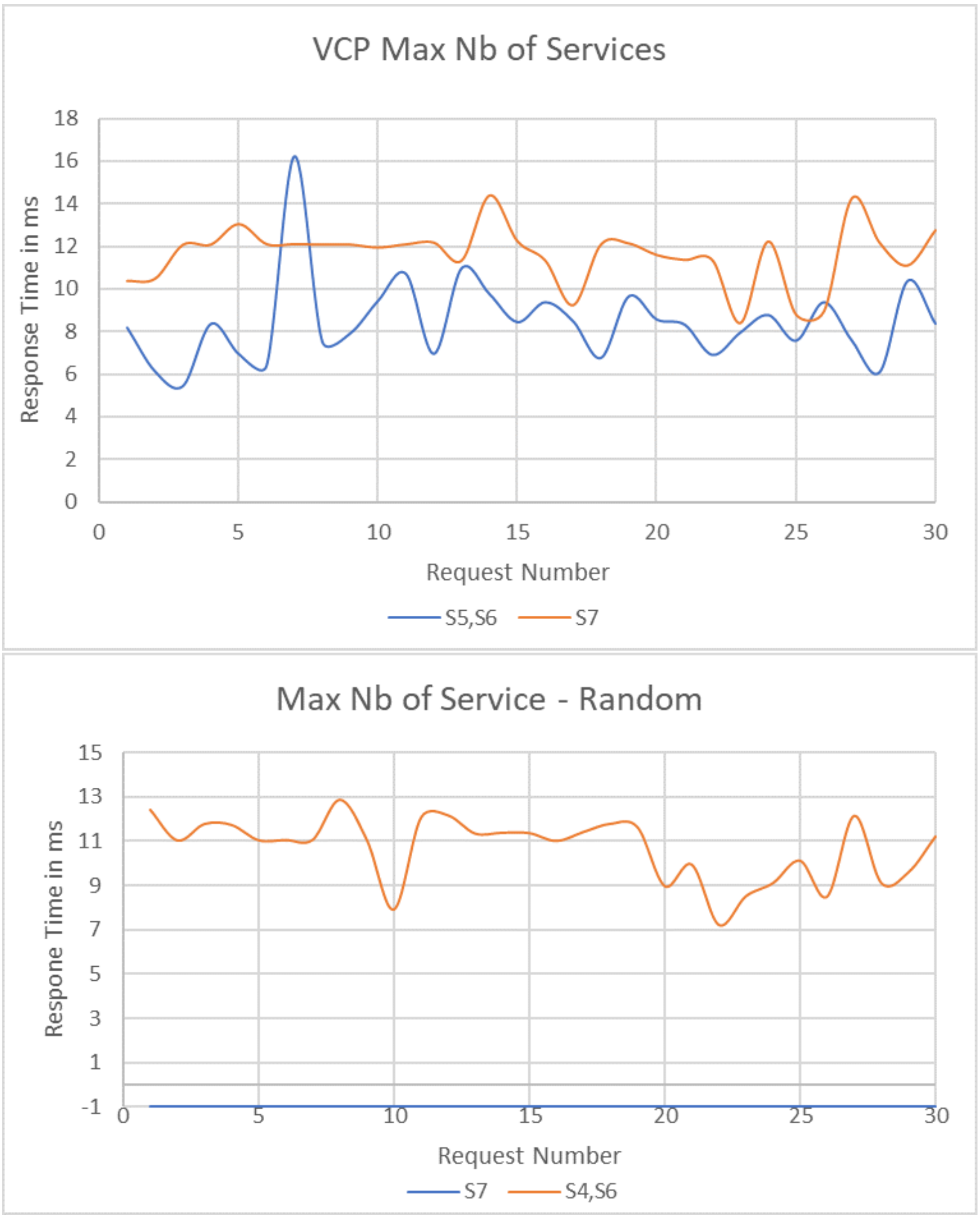


Figure 4.2: Maximize number of pushed services using VCP.

Experimental Results

As per the output matrices shown, the VCP output is to push the micro-services S5, S6 together to V6 and the service S7 to V4. In the searching selection, S5, S6 are pushed

	VCP	Search	weights
F1	30	30	0.01
F2	0	0	0.29
F3	3	2	0.5
F4	-2	-1	0.2
Total	1.4	1.1	-

Table 4.6: Objective Functions Evaluation - Scenario 2

to V4 and S7 to V6. Following the VCP decision, we can notice that both services (S5/S6 and S7) are available to the user all the time. The services response time is high a bit because the user is connected to the fogs through RSUs. For the searching algorithm output, S5 and S6 services are available, but S7 is not. Because S5 and S6 were pushed to V4, half of the resources are not being used. Therefore, we cannot host S7 anymore on V6 because S7 resources requirements are more than V6 capacity. Hosting S5/S6 to V1, allows us to utilize almost the full available resources of V6 and allows S7 to be hosted on V4. V7 and V8 are not considered in the search algorithm or VCP because they are further away from the user than V4 and V6. Based on the experiment's results, we proved that our Memetic solution is capable of maximizing the number of pushed services.

Experiment 6: Importance of increasing the cluster lifetime

While taking the VCP decision, services can be placed on nearby cars having enough resources that results in the least delay possible. However, if the services are being pushed to clusters that do not have a stable connection, or where the orchestrator/worker nodes will soon reach their destinations and stop serving, this leads to shorter serving time. In this case, the decision of selecting serving vehicles is greatly affected by their serving time. The aim is to maximize the cluster serving time.

Experiment Setup

We use Scenario 3 for this experiment. Services are ready and running on the fogs before the vehicles start moving. We also let V2 leave the cluster after 54s from its starting time. The user drives next to V2 at the same speed following the same direction. A delay of 5ms is added between the user and V4. We assumed that the service is already running on both cars. The response time of user's requests sent to S7 either hosted by V2 or V4 is shown in figure 4.3.

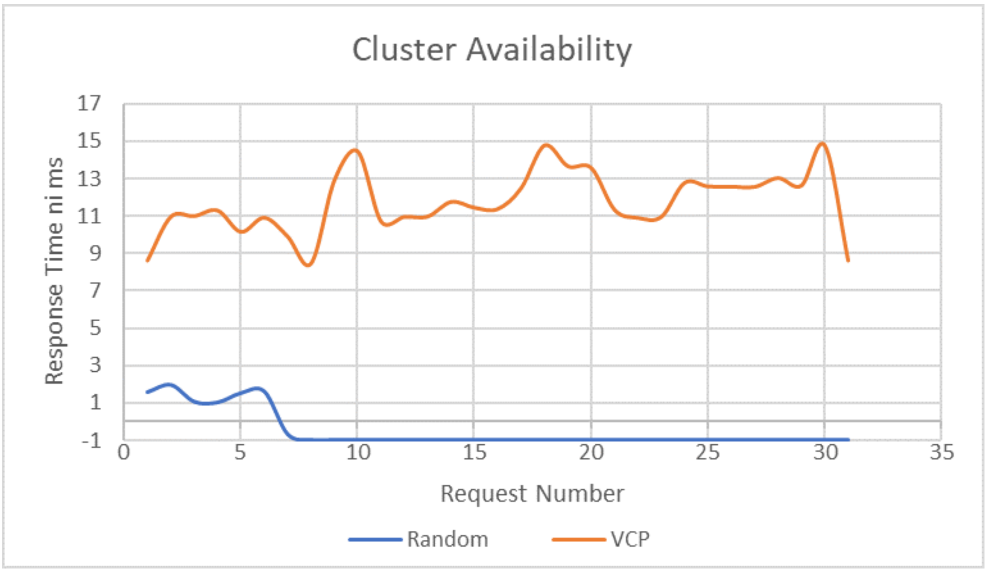


Figure 4.3: Maximize cluster availability using VCP

Matrix output for scenario 3 - VCP:

$$K_{ij} = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$$

Matrix output for scenario 3 - Search:

$$K_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

	VCP	Search	weights
F1	40	5	0.1
F2	0	0	0.2
F3	1	1	0.3
F4	-1	-1	0.3
Total	4	0.5	-

Table 4.7: Objective Functions Evaluation - Scenario 2

Experimental Results

The VCP decides to push S7 to V4, whereas the search algorithm pushes S7 to V2. As shown in the results of Figure 4.3, after 1.2s, S7 is no longer available on V2 because it reached its destination and stopped serving. We can see that S7 is available on V4 throughout the simulation time because V4 can serve for 40s with enough resources. Depending on the available resources, user's requirements, and services importance, VCP tries to provide the optimal selection and distribution of services. In terms of service importance, the user decides what services to request.

4.5 Conclusion

In this chapter, we defined the VCP as a multi-objective optimization problem and proved its hardness. A mathematical formulation is presented stating the inputs and expected output matrices. A solution to the problem is subject to different constraints that should be considered to build feasible solutions. Various objective functions are also formulated, such as the optimization of the number of pushed services, the number of moving vehicles, the cluster lifetime, and the micro-service time connectivity. A Memetic algorithm built on top of the genetic algorithm is used with a combination of a probabilistic local search algorithm to guarantee fast and optimum solution reachability.

A series of experiments are conducted showing the improvement achieved by our architecture in terms of vehicle's selection and services assignment. Google Cluster Traces dataset of 2011 is used to represent service resources requirements and vehicles capacities. Mobisim to generate a real-life mobility data of vehicles is also used. We first showed the ability of the VCP to improve micro-services connectivity by pushing them on vehicles having the same driving patterns. Second, we illustrate the gain achieved by trying to host the maximum number of services on the set of available vehicles. Finally, we experimented with the capability of VCP to select the group of vehicles that leads to longer service lifetime through higher cluster availability.

Chapter 5

Conclusion

5.1 Conclusion

In this paper, we proved that the use of RSUs to support real-time vehicular applications is not possible. Moreover, On-Board Units are still constrained by the availability of their resources. Therefore, one or two OBUs are not enough to handle the processing of the vast amount of data generated by vehicles. There is no work in the literature capable of hosting services on the vehicle. In case vehicles are used as fog devices, they should keep an alive connection with the user to provide longtime support. Another problem arises in this context, which is finding the best fit of micro-services on available vehicles in a way that maximizes the vehicles serving time and maintains its reachability. This thesis focuses on addressing these concerns by elaborating a framework for efficient vehicular clusters formation and service deployment on demand. Below we present a summary of the thesis contributions:

1. Elaborating an on-demand service deployment approach on Kubeadm vehicular clusters.

2. Adapting a hybrid multi-layered networking architecture to keep a vital connec-

tion between the user and available fogs.

3. Adapting a local master election procedure.
4. Introducing a vehicular recovery algorithm to overcome cluster failures.
5. Integrating a flexible multi-objective optimization model for intelligent container placement.
6. Implementing a memetic solution for the vehicular container placement problem.

5.2 Future Work

In our approach, the recovery decision is taken 2s before the vehicle leaves the cluster. As future work, the recovery algorithm should be improved by introducing a reinforcement learning approach to identify the proper time to perform the restore the cluster state.

We can benefit from our approach by introducing a security model for securing vehicular networks. This is a susceptible area because any attacks on the network can lead to data and decisions changes.

On the other hand, upcoming cellular network technologies like 5G are very promising towards the support of real time vehicular applications. Thus, our architecture and methodology can be adapted and enhanced based on the networking technologies advancements in the years to come.

Bibliography

[1] Saif Al-Sultan, Moath M Al-Doori, Ali H Al-Bayatti, and Hussien Zedan. A comprehensive survey on vehicular ad hoc network. *Journal of network and computer applications*, 37:380–392, 2014.

[2] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *Proceedings of the 18th international conference on distributed computing and networking*, page 16. ACM, 2017.

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[4] Luca Carafoli, Federica Mandreoli, Riccardo Martoglia, and Wilma Penzo. Evaluation of data reduction techniques for vehicle to infrastructure communication saving purposes. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, pages 61–70. ACM, 2012.

[5] Lawrence Davis. Handbook of genetic algorithms. 1991.

[6] Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

[7] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[8] Andreas Festag. Standards for vehicular communication—from ieee 802.11 p to 5g. *e & i Elektrotechnik und Informationstechnik*, 132(7):409–416, 2015.

[9] Borko Furht and Syed A Ahson. *Long Term Evolution: 3GPP LTE radio and cellular technology*. Crc Press, 2016.

[10] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[11] Jérôme Härri, Fethi Filali, Christian Bonnet, and Marco Fiore. Vanetmobisim: generating realistic mobility patterns for vanets. In *Proceedings of the 3rd international workshop on Vehicular ad hoc networks*, pages 96–97. ACM, 2006.

[12] Hua-Jun Hong, Pei-Hsuan Tsai, and Cheng-Hsin Hsu. Dynamic module deployment in a fog computing platform. In *Proceedings of 2016 18th Asia-Pacific on Network Operations and Management Symposium (APNOMS)*, pages 1–6. IEEE, 2016.

[13] Saiful Hoque, Mathias Santos de Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. Towards container orchestration in fog computing infrastructures. In *Proceeddings of the 2017 IEEE 41st Annual on Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299. IEEE, 2017.

[14] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. *IEEE Transactions on Vehicular Technology*, 65(6):3860–3873, 2016.

[15] Kang Kai, Wang Cong, and Luo Tao. Fog computing for vehicular ad-hoc networks: paradigms, scenarios, and issues. *the journal of China Universities of Posts and Telecommunications*, 23(2):56–96, 2016.

[16] Fabio López-Pires and Benjamín Barán. Many-objective virtual machine placement. *Journal of Grid Computing*, 15(2):161–176, 2017.

[17] Sara Mehar, Sidi Mohammed Senouci, Ali Kies, and Mekkakia Maaza Zoulikha. An optimized roadside units (rsu) placement for delay-sensitive applications in vehicular networks. In *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pages 121–127. IEEE, 2015.

[18] Khaleel Mershad and Hassan Artail. Finding a star in a vehicular cloud. *IEEE Intelligent transportation systems magazine*, 5(2):55–68, 2013.

[19] Thabit Sultan Mohammed, Omer F Khan, Ahmmed S Ibrahim, and Rustom Mamlook. Fog computing-based model for mitigation of traffic congestion. *International Journal of Simulation–Systems, Science & Technology*, 19(3), 2018.

[20] Roberto Morabito, Ivan Farris, Antonio Iera, and Tarik Taleb. Evaluating performance of containerized iot services for clustered devices at the network edge. *IEEE Internet of Things Journal*, 4(4):1019–1030, 2017.

[21] Ferrante Neri and Carlos Cotta. Memetic algorithms and memetic computing optimization: A literature review. *Swarm and Evolutionary Computation*, 2:1–14, 2012.

[22] Quang Huy Nguyen, Yew-Soon Ong, and Meng Hiot Lim. A probabilistic memetic framework. *IEEE Transactions on evolutionary Computation*, 13(3):604–623, 2009.

[23] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[24] Pethuru Raj, Jeeva S Chelladhurai, and Vinod Singh. *Learning Docker*. Packt Publishing Ltd, 2015.

[25] Rahab M Ramadan, Safa M Gasser, Mohamed S El-Mahallawy, Karim Hammad, and Ahmed M El Bakly. A memetic optimization algorithm for multi-constrained multicast routing in ad hoc networks. *PloS one*, 13(3):e0193142, 2018.

[26] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.

[27] Mohammad Ali Salahuddin, Ala Al-Fuqaha, and Mohsen Guizani. Software-defined networking for rsu clouds in support of the internet of vehicles. *IEEE Internet of Things journal*, 2(2):133–144, 2015.

[28] Gigi Sayfan. *Mastering Kubernetes*. Packt Publishing Ltd, 2017.

[29] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111):2, 2014.

[30] Mahendra Pratap Singh and Manoj Kumar Jain. Evolution of processor architecture in mobile phones. *International Journal of Computer Applications*, 90(4):34–39, 2014.

[31] Mehdi Sookhak, F Richard Yu, Ying He, Hamid Talebian, Nader Sohrabi Safa, Nan Zhao, Muhammad Khurram Khan, and Neeraj Kumar. Fog vehicular computing: Augmentation of fog computing using vehicular cloud computing. *IEEE Vehicular Technology Magazine*, 12(3):55–64, 2017.

[32] Nguyen B Truong, Gyu Myoung Lee, and Yacine Ghamri-Doudane. Software defined networking-based vehicular adhoc network with fog computing. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 1202–1207. IEEE, 2015.

[33] Seyhan Ucar, Sinem Coleri Ergen, and Oznur Ozkasap. Multihop-cluster-based ieee 802.11 p and lte hybrid architecture for vanet safety message dissemination. *IEEE Transactions on Vehicular Technology*, 65(4):2621–2636, 2016.

[34] Omar Abdel Wahab, Hadi Otrok, and Azzam Mourad. Vanet qos-olsr: Qos-based clustering protocol for vehicular ad hoc networks. *Computer Communications*, 36(13):1422–1435, 2013.