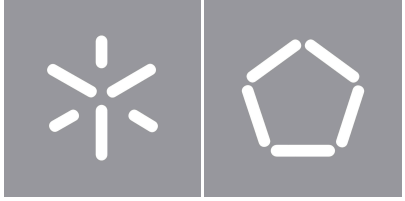


University of Minho
School of Engineering

Rúben Samuel Alves Santos

Software Defined Vehicular Networks: Mobility of Microservices in an Emulated Environment

December 2024



University of Minho
School of Engineering

Rúben Samuel Alves Santos

Software Defined Vehicular Networks: Mobility of Microservices in an Emulated Environment

Master's Dissertation
Master's Degree in Informatics Engineering

Dissertation supervised by
Prof. Dr. António Luís Duarte Costa
Prof. Dr. Pedro Nuno Miranda de Sousa

December 2024

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY-SA

https://creativecommons.org/licenses/by-sa/4.0/

Acknowledgements

I would like to express my gratitude to both my supervisors, Professor António Luís Duarte Costa and Professor Pedro Nuno Miranda de Sousa, for their guidance during every stage of this project. This field is still largely academic, which made it difficult to find practical advice. As a result, the expertise and insights from my supervisors was essential in overcoming these challenges.

The very existence of this work is not the result of individual effort, but rather relies on the countless contributions and foundational technologies that have come before. To the researchers, engineers, and visionaries who laid the groundwork for these complex systems: my sincere thanks.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, December 2024

Assinado por: **Rúben Samuel Alves Santos**
Num. de Identificação: 15626973
Data: 2024.12.26 15:58:18 +0000



Rúben Samuel Alves Santos

Abstract

With an estimate of 57% of the world population already living in cities and with no signs of slowing down, traffic becomes a critical problem and the demand for a more intelligent transportation system increases. Another strong motivator for this rising demand is road safety, which was the main driving factor of research in the area. Questions such as the sustainability of the *mobility of the future* can also addressed in an easier and programmable way.

We can model a transportation system by thinking of vehicles as nodes with communication capabilities that can interact with each other creating a Vehicular Ad-hoc Network (VANET). This network can then be connected to roadside infrastructure that in turn will collect information generated by the vehicles and propagate it to, for example, a nearby edge computing device. To enable an efficient and flexible traffic flow we can control the network from a centralized controller in what is called as Software-Defined Networking (SDN). This dissertation aims to integrate VANETs and SDNs in a Software-Defined Vehicular Network (SDVN).

By looking at our model and defining vehicles as service consumers, where services can range from road safety to infotainment applications, we are challenged to develop a dynamic microservice mobility strategy to ensure their optimal positioning and latency requirements. In this dissertation the goal is to implement this strategy in an emulated environment and optimize parameters like the global delay or service availability. The choice of service also impacts the results and optimization parameters since that a road safety service would have a different set of requirements compared to a video streaming service. In this project, a service with real-time and low latency requirements is needed to demonstrate the efficiency of the mobility strategy.

Keywords SDN, VANET, SDVN, Microservices

Resumo

Estima-se que cerca de 57% da população mundial viva atualmente em cidades. Sendo que esta tendência não mostra sinais de desaceleração, o tráfego rodoviário torna-se num problema crítico e a procura por sistemas de transporte inteligentes aumenta. Outro fator que motiva este aumento é a segurança rodoviária, que é o principal fator impulsionador da investigação na área. Questões como a sustentabilidade da *mobilidade do futuro* também podem ser abordadas de uma forma mais fácil e programável.

Podemos modelar um sistema de transporte pensando nos veículos como nós com capacidades de comunicação que podem interagir entre si criando uma Rede Veicular Ad-hoc (VANET). Essa rede pode ser ligada à infraestrutura rodoviária, que coleta principalmente informações geradas pelos veículos e propaga-as para, por exemplo, um dispositivo de computação de borda próximo. Para permitir um fluxo de tráfego eficiente e flexível, podemos controlar a rede a partir de um controlador centralizado no que é chamado de Redes Definidas por Software (SDN). Esta dissertação tem como objetivo integrar VANETs e SDNs em uma Rede Veicular Definida por Software (SDVN).

Ao olhar para o nosso modelo e definir os veículos como consumidores de serviços, onde os serviços podem variar desde a segurança rodoviária até aplicações de infoentretenimento, somos desafiados a desenvolver uma estratégia dinâmica de mobilidade de microsserviços para garantir o seu posicionamento ideal e requisitos de latência. Nesta dissertação o objetivo é implementar esta estratégia num ambiente emulado e otimizar parâmetros como o atraso global ou a disponibilidade do serviço. A escolha do serviço também impacta os resultados e parâmetros de otimização, uma vez que um serviço de segurança rodoviária teria um conjunto de requisitos diferente de um serviço de streaming de vídeo. Neste projeto é necessário um serviço com requisitos de tempo real e baixa latência para demonstrar a eficiência da estratégia de mobilidade.

Palavras-chave SDN, VANET, SDVN, Microserviços

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Objectives . . . 3
1.4 Dissertation Organization . . . 4

2 State of the Art 5

2.1 Vehicular Ad-hoc Networks . . . 5
2.2 Software-Defined Networks . . . 9
2.3 Software-Defined Vehicular Networks . . . 15
2.4 Microservices . . . 19
2.5 Related Work . . . 27

3 Conceptual Architecture 31

3.1 Problem Context . . . 31
3.2 Conceptual Solution . . . 33

4 Framework Architecture and Components 37

4.1 Software Components . . . 37
4.2 Implementation Guide . . . 39
4.3 Framework Evaluation . . . 43

5 Experiments and Results 46

5.1 Experiments Framework . . . 46

5.2 Scenario I: Replication 47
5.3 Scenario II: Migration 60
5.4 Scenario III: Video Streaming 63
5.5 Scenario IV: Load Balancing 71
5.6 Scenario V: Mobility of Microservices 82

6 Conclusions **94**
6.1 Summary 94
6.2 Contributions 95
6.3 Future work 95

7 References **98**

List of Figures

1 Example of V2V and V2I communication. 6
2 The Three-Tiered SDN Architecture. 11
3 OpenFlow switch components. 12
4 Generic SDVN architecture. 16
5 Comparison between monolithic and microservices architecture. 20
6 Components of a Kubernetes cluster. 24
7 Detailed view of Pod. 26
8 Conceptual Architecture of microservice mobility framework. 33
9 Data flow of the Conceptual Architecture. 35
10 Implementation architecture with Mininet-WiFi, Ryu and Kind. 41
11 **SUMO** map of the city of Braga. 48
12 Visualization of network elements for the Replication scenario. 53
13 Visualization of vehicle position in range of *ap2*. 55
14 Wireshark screenshot of Openflow packet associated with **DNAT**. 58
15 Wireshark screenshot of Openflow packet associated with **SNAT**. 59
16 Visualization of vehicle position in range of *ap3*. 60
17 **SUMO** map of the city of Braga, updated with more **AP** coverage. 64
18 Streaming protocol diagram. Example of initial chunk list request and the following individual chunk requests. 65
19 Initial view of the telemetry graphic for Video Streaming scenario. 66
20 Screenshot of video sample being reproduced in the vehicle. 67
21 **SUMO** map of the borough of Manhattan. 72
22 Architecture of the Load Balancing scenario program. 73
23 View of the initial telemetry screen for the load balancing scenario. 76

24 Configuration of the nine **APs** used in the Load Balancing scenario. 77
25 Load of each node during the simulation with random routes and ten vehicles. 80
26 Load of each node during the simulation with random routes and one hundred vehicles. 81
27 Wireshark screenshot of Openflow packet associated with **DNAT** for specific vehicle with IP 10.0.0.8 in Load Balancing scenario. 82
28 Global latency of both strategies for a simulation using a single deployment at any node and one hundred vehicles. 89
29 Global latency of both strategies for a simulation using three deployments and 100 vehicles. 90
30 Global latency of both strategies for a simulation using five deployments and 100 vehicles. 91

List of Tables

1 Docker containers created for the first scenario. 49
2 Kubernetes nodes and their IP addresses, all in *Ready* state. 49
3 Services available in the Kubernetes cluster. 50
4 Deployments managing pod replicas in the Kubernetes cluster. 50
5 Pods running in the Kubernetes cluster. 51
6 Deployments managing pod replicas in the Kubernetes cluster after first replication. . . 56
7 Pods running in the Kubernetes cluster after first replication. 56
8 Kubernetes nodes and their IP addresses, all in *Ready* state. 62
9 Deployments managing pod replicas in the Kubernetes cluster. 62
10 Pods running in the Kubernetes cluster. 62
11 Deployments managing pod replicas in the Kubernetes cluster, after migration. 63
12 Pods running in the Kubernetes cluster, after migration. 63
13 Pods running in the Kubernetes cluster for scenario III Video Streaming. 70
14 Kubernetes nodes used in the Load Balancing scenario, all in *Ready* state. 77
15 Initial deployments for the Load Balancing scenario. 78
16 Kubernetes nodes used in the Mobility Strategy scenario, all in *Ready* state. 87
17 Initial deployments for the Load Balancing scenario. 87
18 Simulation results comparing latency metrics across different strategies and node configurations. 92

Acronyms

AP Access Point.

API Application Programming Interface.

c-c-m Cloud Controller Manager.

c-m Controller Manager.

C-V2X Cellular Vehicle-to-Everything.

CAM Cooperative Awareness Message.

CAV Connected and Autonomous Vehicle.

CD Continuous Delivery.

CI Continuous Integration.

DASH Dynamic Adaptive Streaming over HTTP.

DNAT Destination Network Address Translation.

DoS Denial-of-service.

DSRC Dedicated Short-Range Communications.

ESB Enterprise Service Bus.

EV Electric Vehicle.

G-VSPA Greedy V2X Service Placement Algorithm.

IoT Internet of Things.

ITS Intelligent Transport Systems.

K8s Kubernetes.

MANET Mobile Ad-hoc Network.

MEC Multi-access Edge Computing.

MVC Model-View-Controller.

NETCONF Network Configuration Protocol.

NFV Network Function Virtualization.

OBU On-Board Unit.

OOP Object-Oriented Programming.

OVSP Optimal V2X Service Placement.

PV Persistent Volumes.

PVC Persistent Volume Claims.

RSU Road Side Unit.

SDN Software-Defined Networking.

SDVN Software-Defined Vehicular Network.

SNAT Source Network Address Translation.

SOA Service Oriented Architecture.

SOAP Simple Object Access Protocol.

SUMO Simulation of Urban Mobility.

V2I Vehicle-to-Infrastructure.

V2V Vehicle-to-Vehicle.

V2X Vehicle-to-Everything.

VANET Vehicular Ad-hoc Network.

VM Virtual Machine.

WSDL Web Services Description Language.

Chapter 1

Introduction

1.1 Context

This dissertation brings together several relevant topics in the field of **Intelligent Transport Systems (ITS)**. These systems encompass the application of advanced technology in the transportation sector, with the goal of improving traffic management, safety and efficiency. This dissertation joins **Vehicular Ad-hoc Networks (VANETs)** and **Software-Defined Networking (SDN)** in a concept known as **Software-Defined Vehicular Networks (SDVNs)**. These networks combine the mobility-oriented features of **VANETs** with the flexibility and efficiency of **SDN**, offering a new layer of intelligence to vehicular communications. On top of a **SDVN** we will deploy and manage microservices with a primary focus on developing an effective mobility strategy for these services.

VANETs are a key component of this project because they enable vehicles to communicate with each other and with relevant infrastructure. While it is important to understand the advantages and limitations of these networks (highlighted in section 2.1) the project, in practical terms, is more focused on the implementation of an **SDN** controller that can manage the network infrastructure in an emulated environment. Therefore, the understanding of the paradigm shift that came with **SDN** (section 2.2 and 2.3) is where the project situates its core emphasis. But the transition from a theoretical analysis to a hands-on implementation will require much more than writing and studying the state of the art on these singular topics: it requires practice and experimentation.

The project, from an operational point of view, depends not only on the intricacies of the development and deployment of an **SDN** controller but also on the integration of microservices into **SDVNs**. This integration will depend on an existent interface between the controller and an orchestrator for an effective management of microservices. This connection will allow us to centrally control an orchestration layer that facilitates the dynamic deployment, scaling, and coordination of microservices. This approach, generally and without starting to define the system's architecture, allows this project to have a competitive chance

if deployed in a contemporary transportation system.

In conclusion, this project has a multifaceted context that will center on the development and deployment of an **SDN** controller and the seamless integration of a microservice's architecture in a **SDVN**. Despite the potential that this area has to transform and enhance traffic solutions, these technologies are not yet mainstream and their full-scale deployment remains a subject of ongoing research and development. This is in part due to evolving standards and continuous technological advancements that come as a result of a solid research and industry interest in the area. It is expected that these technologies will mature, and their adoption will reshape the future of **ITS**.

1.2 Motivation

There are lots of reasons that lead us to believe in the importance of this project and area in general. One possible motivating factor is that the share of the population living in urban areas has been increasing in the past few decades, and it is reported that in 2023 around 57% of the world population lived in these areas [1]. For the United States (US) and Europe this figure sits at 83% and 75%, respectively. The same study estimates that around 70% of the world population will live in urban areas by 2050. When we join this data with the car ownership data, which slightly increased in Europe in the last five years and sits at 559 cars per 1000 inhabitants [2], and is no different in the US with 91.7% of households reporting to have at least one vehicle [3], we have the perfect recipe for a more intense and chaotic urban traffic. While more traffic does not necessarily mean more severe accidents, *the increase in traffic congestion is positively associated with the frequency of fatal and serious injury accidents* [4].

There is an actual safety concern that derives from an increase in urban traffic. This concern, along with vehicle safety regulation enforced by governments, fuels investment from vehicle manufacturers to produce safety technology. Examples go from the good old seat belts, the airbags and simple rearview parking cameras to more modern blind spot detection and driver assistance systems. This project, which focuses on a mobility of microservices strategy for **SDVNs**, can contribute to the building of a platform that will guarantee certain latency and bandwidth requirements essential for real-time safety applications.

Another motivating factor is related with a more contemporary question: climate change. There is already a trend with the increasing **Electric Vehicle (EV)** sales that poses as a mitigation factor for the climate change issue [5]. Despite the known supply chain distribution issues, the **EV** market sales more than doubled in 2021 and account for approximately 5% of global passenger car sales. This electrification phenomenon remains concentrated in China, Europe and the US that together capture about 95% of the

global **EV** sales. In our context of either enhancing road safety or improving traffic efficiency in order to reduce CO2 emissions, the **EVs**, with their electric motors, instant torque and precise control, offer the optimal responsiveness to a dynamic traffic management system. Such system could in turn leverage these capabilities to implement a wide range of traffic efficiency solutions. For example, an adaptive traffic light control system could use the responsiveness of an **EV** to dynamically adjust signal timings thus minimizing waiting times at intersections. Another possible application would be to leverage the **EV** instant torque and precise control to improve traffic flow when travelling in a busy highway or after being stopped at a red light. These improvements would not only reduce the time of travelling but would also have the possibility to improve road safety, all while saving more energy.

In conclusion, it is important to notice that a faster, more reliable, safer and energy efficient transportation system has the potential to dramatically transform society and the way we live for the better. The microservice's mobility strategy, which is the focus of the dissertation, is just a small part of a bigger picture. In the future, **Connected and Autonomous Vehicles (CAVs)** will most likely populate our (smart) cities and will, on a lower level, rely on strategies and technology that is currently being developed and researched today.

1.3 Objectives

The primary goal of this dissertation is the development of a microservice's mobility strategy that will be tested in a realistic emulated traffic scenario. This goal can be subdivided into the following objectives:

- Define a use case for the microservice, e.g., real time image recognition.
- Design a mobility strategy compatible with the particularities of **SDVNs** (e.g., high mobility) and the previously defined use case.
- Create and configure an emulated environment where vehicles are defined as service consumers.
- Enable dynamic relocation of microservices as vehicles move.
- Define and optimize the metrics of the emulation e.g., global delay.
- Validate the model in at least one traffic scenario, e.g., highway, intersection.

By achieving these objectives the main goal of the dissertation would be completed.

1.4 Dissertation Organization

This dissertation is composed by six chapters with the following contents:

- **Introduction**: This chapter contextualizes the dissertation by briefly presenting the topics and fields related to the study. We also outline the motivations and objectives being the research.
- **State of the Art**: This chapter presents the four different areas related to this dissertation: Vehicular Ad-hoc Networks, Software-Defined Networking, Software-Defined Vehicular Networks and Microservices. It also includes a section dedicated to exploring related research.
- **Conceptual Architecture**: The first introductory chapter to the solution, exposes the conceptual architecture designed to tackle the challenges proposed.
- **Framework Architecture and Components**: In this chapter we define the architecture of the solution from a practical point of view. We also give an implementation guide that can be used to replicate the framework. The chapter ends with some comments on the advantages and limitations of the solution.
- **Experiments and Results**: This is the chapter that displays the results obtained in the five experiments of this dissertation. The first two scenarios, Replication and Migration, respectively, are meant to test the basic functionality of the dissertation. The third and fourth scenarios, Video Streaming and Load Balancing, respectively, introduce two new use cases to the framework. The final scenario, Mobility of Microservices, combines the knowledge of all the previous experiments and presents two different strategies to migrate microservices in a highly dynamic environment.
- **Conclusions**: We conclude with a summary of the dissertation achievements and difficulties, along with a section dedicated to future work and possible research avenues.

This work was conducted as part of Project No. 179491, with funding reference SIFN-01-9999-FN-179491, under the COMPETE 2020 program.

Chapter 2

State of the Art

This chapter introduces and examines the key concepts of **VANETs** (Section 2.1), **SDN** (Section 2.2) and Microservices (Section 2.4). It is also imperative to explore the interconnections among these topics, given their importance to the dissertation's objectives. Consequently, a dedicated section on **SDVNs** (Section 2.3) is included. The chapter concludes with an analysis of other related works in the field (Section 2.5).

2.1 Vehicular Ad-hoc Networks

Vehicular Ad-hoc Network (VANET), like their name might suggest, are ad-hoc networks where the nodes are vehicles. An ad-hoc network is decentralized, meaning that it does not rely on pre-existing infrastructure such as routers or wireless access points, which in turn leads them to have a dynamic topology where nodes move and adjust their own configuration. The lack of centralized control allows the network to be more flexible but also introduces significant complexities evident in the network's routing algorithms, which must constantly adapt to the changing positions and availability of nodes [6]. Additionally, the decentralized nature of ad-hoc networks raises important considerations regarding security, as the open and often unsupervised communication environment is susceptible to various threats [7].

VANETs, being ad-hoc networks, share the same decentralized and dynamic environment and inherit most of the challenges of their *ad-hoc* predecessors **Mobile Ad-hoc Networks (MANETs)**. However, one aspect that sets **VANETs** apart is their confinement to road networks. Unlike **MANETs**, where nodes can move in virtually any direction, the movement of vehicles in **VANETs** is largely restricted by road layouts. This constraint on movement can lead to more predictable patterns within the network, facilitating more efficient route planning and network management.

Initially, **VANETs** focused primarily on **Vehicle-to-Vehicle (V2V)** communication, where vehicles directly exchanged information with each other. This method is straightforward but can become complex

in scenarios involving multi-hop communication, particularly in environments with high-speed moving vehicles. In multi-hop **V2V** communication, messages are passed through several vehicles to reach their destination. However, the rapidly changing topology of a network of moving vehicles makes routing these messages more challenging. To address these complexities, later developments in **VANETs** incorporated dedicated roadside infrastructure. These systems use a communication mode referred to as **Vehicle-to-Infrastructure (V2I)**, where vehicles also exchange information with fixed infrastructure.

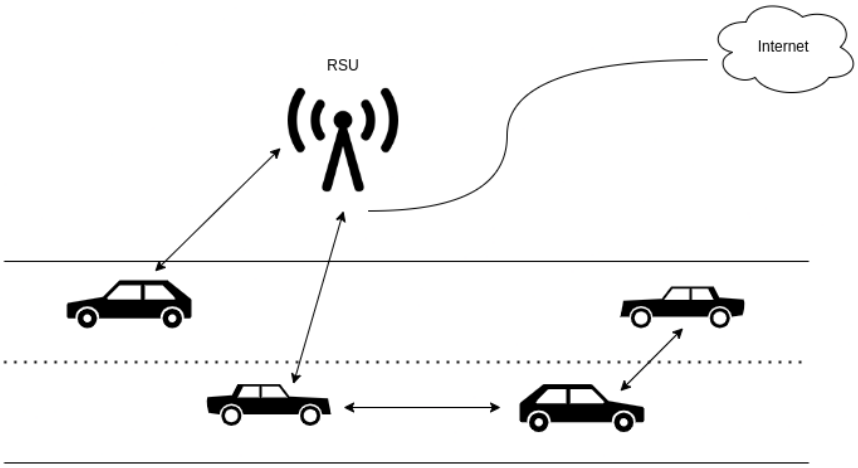


Figure 1: Example of V2V and V2I communication.

Figure 1 illustrates the dual nature of communication in **VANETs**, where vehicles can interact with each other and with fixed infrastructure. The **V2I** communication is demonstrated by the connection between the vehicles and the **Road Side Unit (RSU)**. By serving as a fixed communication point, the **RSU** can offer an array of services that enhance both the functionality and safety of vehicular travel. It also serves as a point of communication between the vehicle and the broader internet infrastructure. This connectivity is essential for smart city initiatives, where vehicles can receive information about parking availability, electric vehicle charging stations, and other urban infrastructure elements. In addition to these conveniences, **RSU**s play a crucial role in safety-related services. They broadcast alerts about road hazards, accident spots, and traffic conditions, significantly contributing to proactive safety measures for drivers [8].

Another important component of **VANETs**, that is not explicitly represented in Figure 1, is the **On-Board Unit (OBU)**. These devices are equipped within the vehicles themselves and serve as the primary interface for all vehicle communications. **OBUs** provide the necessary hardware and software to allow vehicles to communicate with each other and to roadside infrastructure. Moreover, **OBUs** can receive crucial updates and information from **RSUs**, which can include traffic management instructions, safety

warnings, or even firmware updates for the vehicle's systems. With the integration of **OBUs** into vehicles, we gain access to relatively powerful computers without significant concerns about energy consumption. This is the exact opposite scenario of **MANETs**, where energy efficiency is a major constraint due to the reliance on battery-powered devices. This integration opens the door for many applications, especially in improving road safety.

One of these applications is the Collision Avoidance System. This system relies on **Cooperative Awareness Messages (CAMs)** that are periodically broadcasted by vehicles to ensure situational awareness among nearby vehicles. These messages contain information about a vehicle's position, speed and direction that is used by the system to make accurate predictions and avoid potential collisions [9].

The Emergency Vehicle Warning System is designed to decrease the emergency response time for vehicles like ambulances and fire trucks. By informing drivers about the approach of these vehicles, the system helps in clearing the path more efficiently. This not only aids in reducing response times in critical situations but also enhances the overall safety of both the emergency responders and other road users.

The Road Hazard Warning system alerts drivers to various potential dangers on the road, such as accidents, road construction, or hazardous weather conditions. By receiving real-time updates about such hazards, drivers can make informed decisions to either slow down, take alternate routes, or exercise additional caution while driving through the affected areas.

Intersections are often hotspots for vehicle collisions, particularly in areas with limited visibility or non-functioning traffic signals. The Intersection Collision Warning system addresses this challenge by alerting drivers about potential collision risks at intersections. It ensures that drivers are aware of other vehicles approaching from different directions, enabling them to take timely actions to avoid accidents.

Vehicle platooning is an application of **VANETs** that involves groups of vehicles traveling closely together at high speeds, controlled by interconnected driving systems. This method is not only efficient in terms of fuel consumption but also significantly reduces the risk of accidents caused by human error. By maintaining optimal distances and synchronizing movements, platooning ensures a safer and more controlled driving environment, particularly on highways.

Implementing applications in **VANETs** comes with its unique set of challenges, one of the most significant being scalability [10]. **VANETs** operate in environments where a large concentration of vehicles is common, making it essential to design a network that can efficiently handle numerous nodes without compromising connection speed. This requirement becomes more complex due to **VANETs** limited bandwidth, a consequence of the challenges in maintaining a stable network connection in mobile conditions.

An essential feature of a scalable **VANET** system is its ability to manage broadcast messages effec-

tively. Since **VANET** applications often transmit safety or warning data to all nearby vehicles, it's crucial to avoid overloading the network with excessive broadcasting, which could lead to a broadcast storm, consuming significant network capacity. To address these challenges, **VANET** technologies have been focusing on developing network systems capable of efficiently handling a high volume of nodes [6]. This includes algorithms that prioritize the relevance of messages based on factors such as the message's age, the current **VANET** environment, and the importance of the information to nearby vehicles. For instance, a message about a sudden brake by a vehicle ahead would be tagged as high priority and fast-tracked across the network. Furthermore, the concept of clustering has been introduced to enhance network management in **VANETs**. Clusters group vehicles in proximity, with a nominated cluster head managing communications within the cluster. This system reduces the number of packets needed for broadcasting a message, as the cluster head can efficiently disseminate information to all vehicles in the cluster [11].

As we delve into the scalability of **VANETs**, it's crucial to understand two key communication technologies that underpin these networks: **Dedicated Short-Range Communications (DSRC)** and **Cellular Vehicle-to-Everything (C-V2X)** [12]. **DSRC**, a technology based on Wi-Fi, is designed for short to medium range communication between vehicles and between vehicles and roadside infrastructure. It is known for its low latency and reliability in high-velocity environments. On the other hand, **C-V2X**, an emerging technology, leverages cellular networks, including 4G and 5G, extending beyond **V2V** and **V2I** communications to encompass a broader concept known as **Vehicle-to-Everything (V2X)**. **V2X** is a vehicular communication system that enables the transfer of information from a vehicle to moving parts of the traffic system that may affect the vehicle: infrastructure, vehicles, network, pedestrians, etc.

DSRC faces scalability challenges in dense traffic areas due to its limited spectrum and dependence on Wi-Fi technology, necessitating adaptive strategies for more effective channel management. In contrast, **C-V2X** notably surpasses **DSRC** in scalability. It benefits from the broader bandwidth and advanced features of cellular networks, including 4G and 5G, enabling more efficient performance in dense vehicular environments. Additionally, the evolution of 5G technology promises even lower latencies and higher data throughput, significantly enhancing **C-V2X**'s scalability.

Considering the strengths and limitations of both radio access technologies, an integrated approach can offer a comprehensive solution to **VANET** scalability. **DSRC**, with its low latency, is well-suited for immediate, safety-critical communications, while **C-V2X** can handle broader, non-safety-related data transmissions and benefit from the ongoing advancements in cellular technologies.

To conclude this section on **VANETs** it is essential to recognize that there are a few challenges that put mainstream adoption a step further. This holds true despite the interest from automotive companies

and the vast array of potential applications. This current situation is primarily due to a mix of regulatory barriers and the intrinsic characteristics of **VANET** technology, such as high mobility, decentralized network structure, variable network density, real-time communication needs, and security and privacy concerns. A significant barrier to the adoption of this technology is the network effect - the benefits of **VANET** systems become truly apparent only when there is widespread adoption among vehicles. This creates a paradoxical situation where early adopters see limited benefits, leading to reluctance from both manufacturers and consumers to invest in **VANET** technologies.

Another question that is worth addressing is the need for **VANETs** when cellular networks already exist. The main argument is that localized communication means that **VANETs** can offer lower latency and more reliable data exchange in specific vehicular contexts, which is vital for safety-critical applications.

In conclusion, while **VANETs** have not yet reached mainstream status, their potential to revolutionize vehicular communication and transportation systems is undeniable. With regulatory hurdles being addressed and technological advancements on the horizon, the future for **VANETs** is indeed promising. As more vehicles become VANET-capable and as infrastructural support grows, we can expect a significant shift in how we perceive and interact with our vehicular environments, ultimately leading to safer, more efficient, and connected roadways.

2.2 Software-Defined Networks

The appearance and prevalence of **Software-Defined Networking (SDN)** [13, 14] was motivated by the need for a more scalable, flexible and programmable network. In traditional networks the control and data plane are tightly coupled within network devices like switches and routers. In practice this means that whenever network administrators need to change network policies they have to do it so individually and often using a vendor specific-interface. This design, while robust and reliable for static and predictable network traffic, struggled to keep pace with the dynamic demands of modern data-intensive applications and cloud computing environments.

SDN is a networking paradigm where the main characteristic is the separation of the control and data plane. This new approach leads the network devices to be less complex, since they only have to forward traffic while all the control decisions are implemented in a centralized controller. This architecture not only brought greater flexibility but also opened the door to innovation, allowing network operators to tailor network behavior through software applications without needing to modify the hardware.

In this new era of cloud computing, big data and mobile networks, traditional network infrastructures

simply don't have enough flexibility to adapt to rapidly changing demands and complex traffic patterns. This can lead to operational inefficiencies, as each network device requires individual configuration and management, a process that is not only time-consuming but also prone to human error. The decoupling of the control and data plane allows administrators to dynamically adjust network resources and policies via software, which in turn increases overall network efficiency. The centralized control model also allows for a more informed and rapid decision process, instead of the distributed and often isolated decision-making inherent in traditional network architectures. Furthermore, this centralization facilitates easier integration with higher-level services and applications. By providing programmable interfaces to the controller, **SDN** opens up opportunities for automation, orchestration, and innovation in network management. Thus, the centralized control model not only addresses the inefficiencies of traditional networking but also lays the foundation for a more agile, responsive, and user-centric networking environment.

The concept of **SDN** was further reinforced by the advent of OpenFlow [15], the first standard communications interface defined between the control and forwarding layers of a **SDN** architecture. OpenFlow allows the direct access to the forwarding plane of both physical and virtual network devices, that in turn enables the decoupling of the control plane from the data plane. The relationship between **SDN** and OpenFlow offered cloud providers unprecedented flexibility in deploying and managing devices from different vendors [16]. Traditionally, cloud providers like Google and Amazon were constrained to using high-performance switches and routers from the same vendor to ensure seamless network management. Each vendor's devices often required custom configurations using their unique language syntax, making network management a complex and time-consuming task. **SDN** by standardizing network management, enables cloud providers to rapidly reconfigure routing parameters and resource distribution, regardless of the vendor. The adoption of **SDN** and OpenFlow in cloud environments, as described, presents significant economic incentives where cloud providers can benefit from reduced costs and increased operational efficiency, enabling more competitive service offerings in an increasingly dynamic market. The integration of **SDN** and OpenFlow in cloud computing thus represents a paradigm shift. It not only simplifies network management and enhances flexibility but also underpins the modern cloud computing environment's scalability, efficiency, and adaptability.

The diagram depicted in Figure 2 tries to represent **SDN** architecture by dividing it into three different layers: Application, Control and Infrastructure.

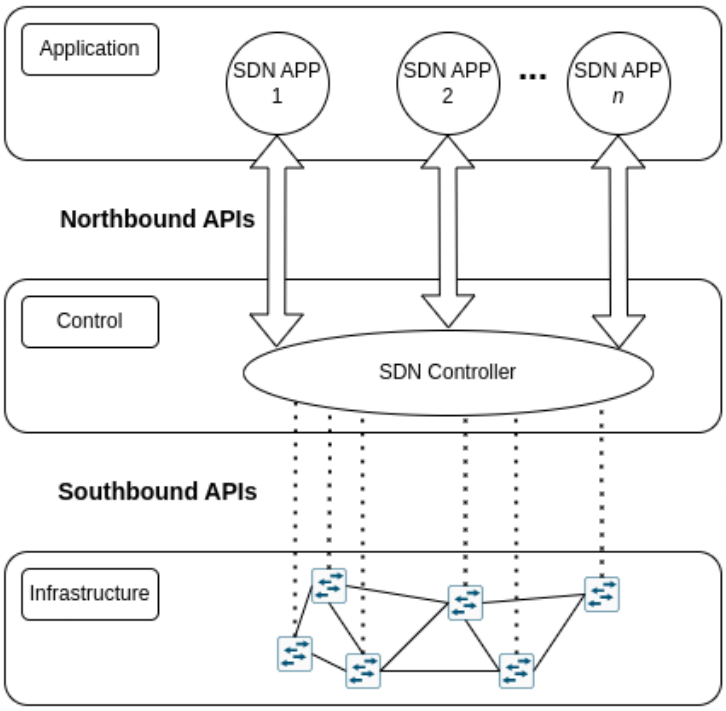


Figure 2: The Three-Tiered SDN Architecture.

At the top of Figure 2 we have the Application Layer, featuring multiple **SDN** applications, labeled as *SDN APP 1* to *SDN APP n*, indicating that we can have numerous applications within the ecosystem. These applications communicate with the Control Layer through Northbound **Application Programming Interfaces (APIs)**. The central component of the Control Layer is the *SDN Controller*, the intelligent component that manages the traffic flow across the network by translating the high-level application requirements into low-level data plane behaviors. Below the *SDN Controller*, Southbound **APIs** provide the necessary interface for the Control Layer to program and manage the Infrastructure Layer. This lower layer consists of network devices such as switches or routers. The bidirectional arrows between the layers imply that there is a dynamic flow of information that allows the controller to make decisions based on the network state and application needs, which can then be enforced on the network hardware.

An example of a standard Southbound **API**, shown in Figure 2, is Openflow. Openflow enables the **SDN** controller to directly influence the behavior of network devices by manipulating their flow tables. It also facilitates the fine-tuning of network traffic for example by controlling specific network paths and dynamically adjust or reconfigure them in response to network conditions, policies or application requirements. The diagram below represents the main components of an OpenFlow Switch, described in depth in the OpenFlow Switch v1.5.1 specification [17].

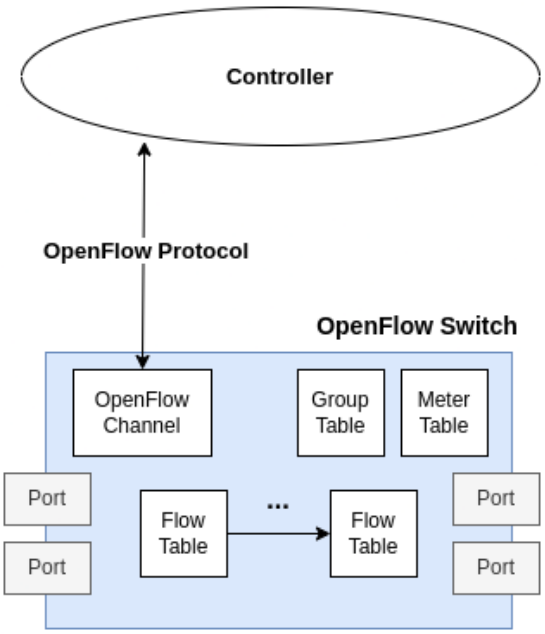


Figure 3: OpenFlow switch components.

Figure 3 illustrates an OpenFlow Logical Switch architecture consisting of flow tables, a group table, a meter table and OpenFlow channels connecting to an external controller. The controller, using the OpenFlow protocol, dynamically manages the switch by adding, updating, or deleting flow entries within these tables.

Flow tables are a crucial component of an OpenFlow Switch, acting like a set of rules that govern how packets of data are handled. Each entry in a flow table has three parts: match fields, counters, and instructions. The match fields are the criteria used to identify packets, sort of like filters that decide if a packet should be acted upon according to a specific rule. Counters are used to keep track of how many packets match each entry, which is useful for monitoring and managing the network. Instructions tell the switch what to do with packets that match the criteria: whether to forward them to another part of the network, modify them, or drop them entirely. In addition to flow tables, a Group Table allows packets to be processed collectively in a batch for actions like forwarding to multiple destinations. Meter Tables are used to measure the rate of packets, ensuring the flow does not exceed a set bandwidth.

The simple yet powerful design of the OpenFlow switch, centered around the direct manipulation of the flow tables within the network, made OpenFlow the *de facto* protocol in the domain of **SDN**. Beyond OpenFlow, the **SDN** landscape is enriched by a variety of other technologies and protocols that complement or offer alternatives to it. One other protocol is the **Network Configuration Protocol (NETCONF)** [18], designed for network configuration. **NETCONF** stands out as a prominent protocol in the realm of

network management as it provides mechanisms to install, manipulate, and delete the configuration of network devices through a set of standardized operations. Despite this, **NETCONF** and OpenFlow are two distinct protocols used in network management that serve different purposes. While **NETCONF** is about configuration management across various devices, facilitating a broad range of management tasks, OpenFlow is specifically designed for **SDN** environments, offering fine-grained control over traffic flows. They can be complementary, with **NETCONF** managing the setup and the state of network devices, and OpenFlow handling the real-time routing of traffic within SDN-enabled networks.

Implementing and operating **SDN**, including OpenFlow-based networks, presents several challenges. Three of the main challenges are Scalability, Compatibility and Security. The scalability challenge in **SDN**, especially with OpenFlow, centers around managing a growing network without compromising performance.

As the network expands, the **SDN** controller must handle a higher number of switches and an increasing volume of data traffic which requires advanced algorithms for network optimization and powerful hardware to prevent processing bottlenecks. The issue intensifies in data-intensive environments, where the need for real-time data processing and quick adaptation to changing network conditions is predominant. Being able to efficiently scale is crucial to maintain performance and the overall effectiveness of the **SDN** central management approach.

The compatibility challenge in **SDN**, particularly with OpenFlow, revolves around ensuring harmonious integration of diverse network components. This includes compatibility between **SDN** controllers and various network devices, each potentially having different capabilities and firmware. The heterogeneity of current network environments that blend legacy systems with modern SDN-capable devices further complicates the issue. The overall success of an **SDN** deployment hinges on its ability to integrate and manage these components which in turn requires extensive customization and is quite resource-intensive.

The security challenge in **SDN** comes primarily from the centralized control that, while efficient, creates a potential single point of failure, making the entire network vulnerable if the controller is compromised. This calls for the need of robust security protocols and systems to protect the controller and the communication between the controller and network devices. Apart from the centralized controller, other security risks in **SDN** include the increased attack surface due to the programmability of the network. This makes **SDN** potentially more vulnerable to advanced, targeted attacks like spoofing or man-in-the-middle attacks on the controller-switch communication. Furthermore, the dynamic nature of **SDN** also poses a risk, as rapid configuration changes might inadvertently create security loopholes.

While addressing the inherent security challenges in **SDN**, it's crucial to also recognize the importance

of **SDN** architecture in shaping emerging technologies. It is the same programmability and centralized control that introduces security complexities that also allows for a more efficient network. This duality of **SDN**, as both a challenge and a facilitator, is especially evident in its integration with evolving fields like for example, 5G networks [19]. In the 5G architecture, **SDN** enables improved control through modularization and virtualization, allowing for dynamic resource allocation and management. This adaptability is essential for 5G's diverse service requirements, from enhanced mobile broadband to ultra-reliable low latency communications. Building on **SDN**'s integration in 5G networks, its impact goes beyond mere network management improvements. **SDN** allows for 'network slicing', a technique where a single physical network can support different virtual networks tailored for specific services. This improves network efficiency and service quality. **SDN** also enhances edge computing in 5G, speeding up data processing closer to users, which is crucial for reducing delays in applications like streaming or online gaming. These capabilities illustrate **SDN**'s importance in maximizing 5G's potential, offering both flexibility and improved performance.

To conclude this section on **SDN** it is important to take a look at the current research trends and future directions. It's evident that security remains one of the main concerns, especially with the growing integration of connected devices [20]. The existing solutions in **SDN**, while effective to an extent, still face limitations that restrict their practical application. Future research is expected to develop more sophisticated protection mechanisms, taking into account the latest advancements like stateful data planes. A significant research direction involves enhancing the security of SDN-enabled forwarding devices, possibly through designing secure hardware based on the latest software requirements. The unique architecture of **SDN** introduces new security threats, necessitating innovations in network security tools. The centralized network controller, a critical component of **SDN**, poses vulnerabilities to attacks such as **Denial-of-service (DoS)**, which can disrupt the entire network. Intrusion detection systems, adapted for **SDN**, could offer effective countermeasures against such threats. Moreover, the intersection of **SDN** with cyber-physical systems and **Internet of Things (IoT)** opens further research opportunities in securing exchanges in these environments.

SDN's journey represents a shift in network paradigms, breaking away from traditional complexities and embracing programmability and logical centralization. However, this disruption brings its own set of challenges, particularly in security. As **SDN** matures, addressing these security concerns becomes as crucial as enhancing its performance and functionalities.

2.3 Software-Defined Vehicular Networks

In the previous sections the concepts of **VANETs**, networks that enable vehicles to wirelessly communicate with each other and roadside infrastructure, and **SDN**, a networking paradigm that allows for centralized management by decoupling the control plane from the data plane, were presented. In this section, we once again showcase the adaptability of **SDN** and its capacity to fundamentally alter traditional **VANET** operations. The integration of **SDN** into **VANETs** gives life to a novel concept termed **Software-Defined Vehicular Networks (SDVNs)**.

This integration brings three major advantages to traditional **VANETs**: improved traffic efficiency, optimization of wireless resources and increased communication reliability [21].

The traffic efficiency results from the global vision offered by the centralized controller. This perspective allows for more precise routing decisions and a common example is that in a traditional network all nodes might try to use the shortest path which might result in network congestion. However, with a centralized controller this scenario is detected and traffic is rerouted through alternative paths, increasing network utilization and decreasing blockages.

The optimization of wireless resources is due to the fact that in this new paradigm of **SDVNs** the use of various wireless interfaces and configurable radios is facilitated. This means that the controller can dynamically manage the radio interfaces based on the type of traffic and the current network conditions. The result is the optimal use of frequencies or channels, particularly for emergency services because it ensures that this traffic is prioritized and has reliable access to the necessary channels. Overall this enhances the efficiency of the network while, at the same time, plays a critical role in maintaining public safety.

The final advantage of increased communication reliability comes from the ability of managing the transmission power of wireless interfaces. Assuming that the controller is collecting neighboring vehicle information, which is typical in a **SDVN** scenario, the power level can be dynamically changed to ensure an effective packet delivery. This dynamic adjustment helps in maintaining an optimal signal strength that minimizes interference and packet loss, especially in scenarios where vehicles are farther apart or in less dense traffic because it prevents unnecessary spread of signals. Additionally, this power management contributes to a more energy efficient vehicle which might also extend the lifespan of onboard communication devices.

Figure 4 represents the generic architecture of a **SDVN**. We can look at this diagram as an extension of Figure 1 but with the integration of the **SDN** controller.

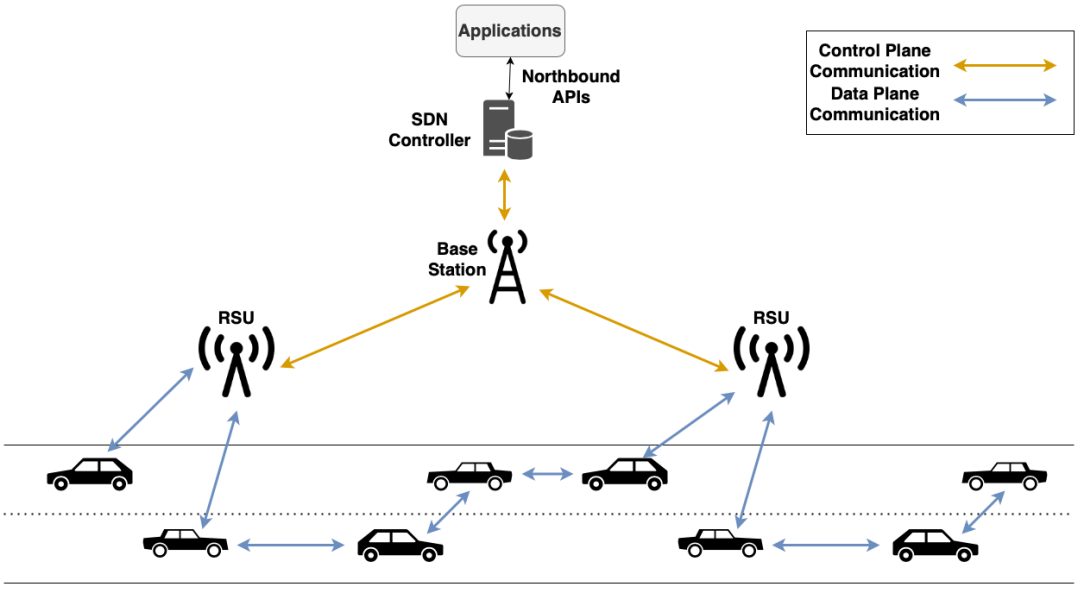


Figure 4: Generic SDVN architecture.

As we have mentioned before, Figure 4, exemplifies the concept of **SDVN**, a paradigm that integrates **SDN** into vehicular communication systems. The central component here is the **SDN** controller that orchestrates traffic flow and network management and communicates (most likely using OpenFlow) with the **RSUs**. As we have seen before, the **RSUs** are critical interfaces for **V2I** communication because they directly connect vehicles to the broader infrastructure. This diagram also adds the base station component, which serves as a mean of communication between the **RSUs** and the **SDN** controller and also extends the network's reach by providing connectivity through various communication protocols such as LTE, 5G and others. This cellular base station (also known as BS, NodeB, or eNodeB) operates under the control of the **SDN** controller. From the base station upwards we have the already familiar Northbound **API** that connects the **SDN** controller to the application layer. This framework marks an evolution in traffic systems where **VANET** communications are now more robust and efficient due to the integration of **SDN**.

It is also possible to extrapolate from Figure 4 that **SDVN** brings a three-tier architecture with an infrastructure, control and application plane. The vehicular information flows from the bottom up, while the decisions and instructions come the other way, top to bottom. In the architecture, the infrastructure layer includes all physical infrastructure and network devices such as **OBUs** and **RSUs**. These are the devices responsible for generating data like speed, routes and positions which is then propagated by routers and switches, also part of this layer. As for the control plane, its function is to gather information from the infrastructure plane given the application requirements. There are two types of control planes: fixed and mobile control plane. In the fixed control plane, controllers are stationary and integrated into the infrastructure. The global controller is located in a central place or cloud server, while local controllers

are installed on **RSUs** or base stations, primarily handling data collection and filtering. Conversely, the mobile control plane features a global controller located in the cloud, with local controllers embedded in vehicles. The global controller handles complex computing tasks and bandwidth-intensive demands while local controllers manage routine requests due to limited memory and computational power.

Network Function Virtualization (NFV) [22, 23] is also instrumental to maximize the efficiency of **SDVN** and therefore is crucial to the services that it includes. This approach to networking decouples network functions from traditional network hardware (e.g., routers, load balancing, firewalls, etc.), allowing them to run as software applications. The agility of **NFV** allows network administrators to deploy and scale them without the need for the physical hardware to be modified which also contributes to the reduction in operational costs. In environments that require quick deployment and frequent updates to network services (e.g., data centers) the **NFV** ends up being particularly beneficial.

We have previously presented use cases for **VANETs** more focused on the safety aspects of vehicle traffic. In the realm of **SDVNs** there are many more applications to present due to new capabilities like real-time response. But even regarding safety services there are clear improvements derived by features like dynamic channel allocation, where sets of frequencies or channels are specifically allocated for emergency traffic, resulting in a more efficient **CAM** distribution.

A widely discussed application is related with on-demand surveillance [24], where **SDVNs** empower the authorities to create surveillance requests. In comparison with traditional **VANETs** where these requests involved either direct or broadcasting communications, **SDVNs** have a centralized controller with a global network overview. The controller can even create rules to direct any required data straight to the requesting vehicle thereby eliminating redundant data transmission and reducing network congestion. In essence, the on-demand surveillance service in **SDVNs** is more a streamlined and responsive approach to data dissemination.

The virtualization of wireless networks also enhances network efficiency as it involves creating virtual networks that operate over a shared physical infrastructure, allowing for more informed and therefore efficient use of network resources. A key to this process is the dynamic grouping of wireless nodes and **RSUs** which aid in traffic management and minimize interference between services. By grouping together wireless nodes that need to communicate with each other we guarantee isolation between groups and reduced interference. This technique is essential not only because it optimizes the overall functioning of the network but also because it maximizes the utility of the physical infrastructure.

A different kind of use case but also very relevant in today's society is the air quality monitoring. By having sensors placed in the infrastructure (e.g., **RSUs**) we could easily create a sensor network that can

gather data to be analyzed by a central system [25]. This information is particularly useful for health and environmental agencies as it gives them a clear picture of where and when air pollution is at its worst, enabling a better decision to protect public health.

Smart Parking in **SDVNs** [26] is a relevant use case for the ever-growing cities especially with the rise of autonomous vehicles. Since information from parking spots and cars is captured in real-time, the controller could easily reserve a parking spot and guide the driver to its location. Due to the flexibility of **SDVNs** they are perfect for managing lots of cars and parking spots at the same time, meaning less time driving and more efficient use of parking spaces.

The concept of in-vehicle entertainment is still fairly recent but accounts for a few applications such as streaming music or videos and participate in video conferences, all while on the move [21]. Expanding on this, recommendations such as neighboring dining options, gas stations or cultural landmarks not only makes travel more enjoyable but also supports local businesses and tourism. For long drives, the passengers can still be productive and finish a school assignment or reliably access a work-related resource. These infotainment systems can even be customized for each passenger with the use of individual profiles, ensuring an overall better travel experience. Vehicles can truly become moving hubs of digital interaction and entertainment and enhance the journey no matter the destination.

To conclude this section it is important to understand the current challenges of **SDVNs**. Some of them derive directly from the **VANET** and **SDN** challenges. One example of this is the high mobility of vehicles causing frequent changes in network topology, posing a challenge for **SDN** controllers and **RSUs** in maintaining stable communication. The solution might be related to both developing more robust algorithms and enhancing, for example, **DSRC** connectivity to withstand rapid speed fluctuations. The other example that comes from **SDN** is that security is still one of the biggest challenges, particularly the risk of attacks on centralized **SDN** controllers. Decentralizing control to mitigate the risks of single-point failures is one of the solutions, but it must be carefully implemented so that we don't lose the benefits of the centralized controller.

In the future, the progress of **SDVNs** depends on several key developments. Firstly, there's a need for creating better and more comprehensive OpenFlow protocol stacks. Additionally, making sure these protocols can work well together, which is known as standardization, is crucial for a seamless integration of various technologies within the **SDVN** framework. This is demonstrated by the complex taxonomy of **SDVNs** [21].

There are also possible improvements in the way the physical components of the network interact with their virtual counterparts. This interaction happens in the so-called east and westbound **APIs** and these

need to be built with the goal of smooth communication between components. In summary, the future of **SDVNs** lies in developing robust, well-integrated systems that can be thoroughly tested and optimized through real-world implementations and simulations. By overcoming the current challenges we pave the way for more efficient, responsive, and smarter vehicular networks.

2.4 Microservices

In this section we introduce a popular software architecture that appeared as an evolution of **Service Oriented Architecture (SOA)** and is commonly referred to as *Microservices* [27]. From an historical point of view, the need for a software design came around the 1960s, where developers started to notice the problems with large-scale software systems. In the following decades, software architecture became a discipline, meaning that it was integrated in the development process in both academia and industrial contexts. With the emergence of **Object-Oriented Programming (OOP)** in the 1990s, previously proposed patterns like **Model-View-Controller (MVC)** became instrumental in the development of software. The further popularization of **OOP** greatly contributed to the field of software architecture.

In the last couple of decades the focus has been in the separation of concerns to achieve better control over design, implementation and the evolution of the system in question. **SOA** came into the picture to address the complexities of these distributed components and is characterized by its approach to modularize large-scale systems into a collection of self-contained modules, each designed to perform a business functionality [28]. Each and every service is independent of the state and context of the others, is defined by standard languages like **Web Services Description Language (WSDL)** and communicate using protocols such as **Simple Object Access Protocol (SOAP)**. But the central component of a **SOA** is the **Enterprise Service Bus (ESB)**, which provides interoperability among **SOA** applications. It offers services like orchestration, reliable messaging and security, enabling the distribution of services across different systems and computing environments. In essence, the **ESB** connects the different **SOA** applications and systems, allowing them to exchange data efficiently.

Microservices emerged as a second iteration of the **SOA**. This new architecture consists of small, independent and highly modular services, each focused on a specific business function. The goal is to strip down the previous larger and more interdependent services and develop even smaller and lighter services i.e., microservices. The main difference is that microservices do not rely on centralized management and typically do not use components like the **ESB**, instead they favor direct and lightweight communication methods like RESTful **APIs**.

It is also useful to compare microservices with the traditional monolithic application, composed of one or more modules that are tightly connected and cannot function independently. Microservices split the application into smaller and self-sufficient pieces, making it easy to update or scale parts of the system without affecting the whole. In contrast, monoliths often need to be rebuilt and redeployed entirely after each change, which becomes time-consuming. In Figure 5 the emphasis is on the differences in terms of components and interactions of both architectures.

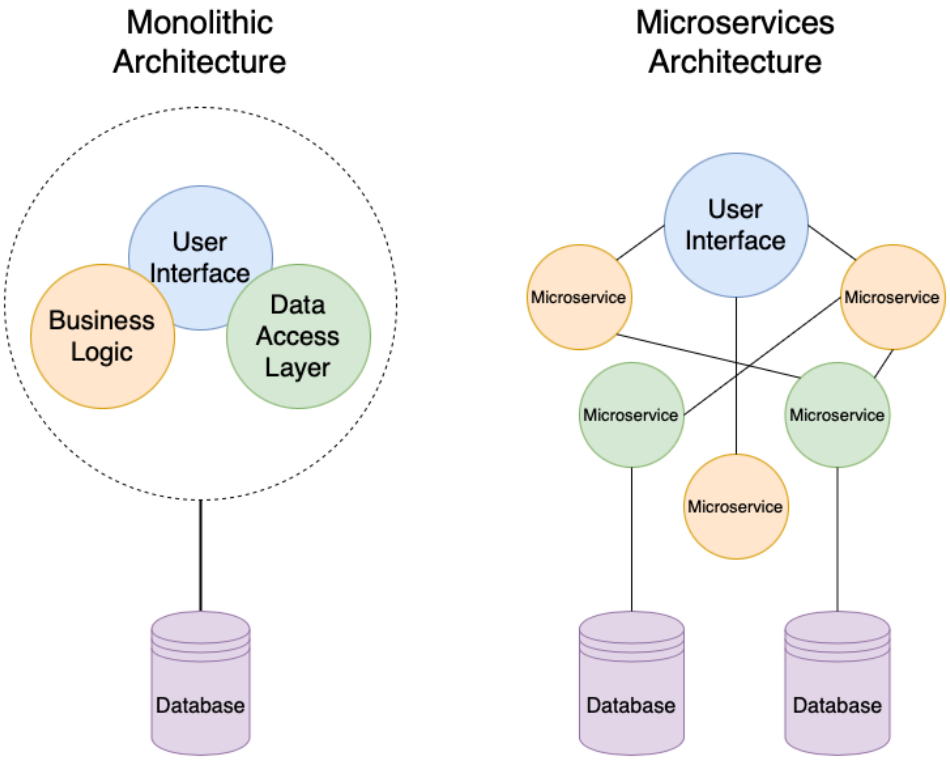


Figure 5: Comparison between monolithic and microservices architecture.

In Figure 5, on the left, we have the Monolithic Architecture as a unified structure where all components are developed and deployed as a single unit, sharing the same database shown at the bottom. In this architecture, each component is interdependent, meaning changes or updates to one area can impact the entire system.

On the right side, Microservices Architecture is illustrated as a network of separate entities, called microservices. The color of these microservices indicates that they originated from a monolithic module on the left. They operate independently, there is no overlap between them, and communicate with each other via dedicated channels i.e., HTTPS. We also gain more abstraction between our data models and our database architecture because the microservices related to the data access layer have to remain independent and are the only ones that have database connection. It is also presented the idea that a

microservice architecture is more scalable because we can have more than one database, meaning that we can scale a specific component individually without scaling the other components.

In fact, microservice architectures include several benefits such as scalability, as each service can be scaled independently based on its demand, and flexibility, since different services can be developed in different programming languages and frameworks. By no longer having a single point of failure, the modularity introduced with microservices offers better fault isolation and an increased resilience of the overall system. This means that if one microservice fails, it doesn't necessarily bring down the entire application.

Despite the benefits, microservices also introduce new challenges, particularly in complexity. By having to manage multiple codebases and deal with inter-service communication, the deployment, coordination and monitoring of the services gets more complex. This complexity can be mitigated with automation tools and **Continuous Integration (CI)**/**Continuous Delivery (CD)** pipelines that streamline these processes. Data consistency across services is another challenge, as each microservice can have different databases. Transactions that span across multiple services end up being more complex to handle than in a monolithic architecture where a single database is used. Ensuring high availability and creating robust communication between services are additional complexities that require sophisticated strategies, like implementing **API** gateways or using messaging queues for asynchronous communication.

After discussing the topic of Microservices, even superficially, one might start to ask if it is even possible to apply this concept in practice. There is a main technology responsible for enabling the microservices architecture referred to as *Containers* [29]. Containers, essentially, are an operating system virtualization mechanism that can isolate and control resources for a group of processes. Unlike the traditional **Virtual Machines (VMs)** that emulate the entire hardware system, containers are more lightweight and efficient as they only isolate and manage resources and don't have to simulate hardware. So as we can see containers are currently the best solution for microservices that need isolation and rapid scalability. One of the underlying mechanisms that allows containers to have isolation are Linux namespaces which are a feature of the Linux kernel that provides isolation for a group of system resources, allowing processes to run in separate environments as if they were on different machines. This approach to microservices with containers is not only resource-efficient but also enhances the deployment flexibility resulting in the containers being able to operate in different environments while maintaining consistency.

This concept of containerization can also be implemented in several ways. The leading solution for it is Docker [30] - a platform that simplifies the process of developing, shipping and running applications by using container technology. Docker containers are based on Linux Containers and use kernel-level names-

paces for isolation as well as control groups for resource accounting and limiting. One of the problems that Docker tries to solve, and is actually mentioned in its white paper, is the issue of *dependency hell* which arises when managing software libraries and their dependencies. Normally, in software development, applications rely on a number of libraries that in turn depend on other libraries originating a web of dependencies that leads to conflicts and difficulties in ensuring that the correct version of each library is used. It gets even worse when different components require different versions of the same library. Docker addresses this issue by allowing each part of an application to be packaged with its dependencies into a container. These containers are, once again, lightweight and portable across different environments.

To conclude the topic of Docker containers, it is worth reinforcing the idea that this tool had a great impact in the realm of software development, particularly microservices. Other abilities of Docker include the version of container images, its compatibility across Linux and non-Linux systems and the fact that it is application-centric with the focus on running applications or services in containers. By the reasons mentioned above, Docker stands out as a robust platform for building, running, and distributing applications, effectively solving many of the complexities associated with software dependencies and environment inconsistencies.

Going back to the discussion on microservices and their relevance to this dissertation, we are missing a critical component that ensures the actual mobility of the microservices - or rather, containers - that is yet to be presented: the orchestrator. In general, the job of an orchestrator, in the context of containerized applications, is to automate the deployment, scaling, and management of containers. It should handle the lifecycle of containers, ensure they are running as intended, allocate resources, balance loads and manage networking and communication between containers. Without an orchestrator we would only have the unrealized potential of the microservices and container technology and would not be able to build highly distributed systems and applications.

This is where **Kubernetes (K8s)** [31] appear in the ecosystem: as a powerful orchestrator that manages, scales and deploys containers efficiently. **K8s** also automate the distribution and scheduling of containers, handle load balancing, monitor the health of applications and provide mechanisms for service discovery. Essentially this is a platform that addresses the complexity of running microservices in containers, with the goal of guaranteeing an optimal performance in a dynamic environment. The biggest difference between Docker and **K8s** can be seen from a scalability perspective: while Docker manages individual containers, **K8s** manages a group of containers that will possibly run microservices that need to be harmoniously interconnected. This makes **K8s** very important in the realm of modern, cloud-native applications where scalability and resilience is critical.

In the following paragraphs are presented some key features of **K8s**. The first of these features, and also particularly relevant to this project, is automated scheduling. This allows the system to *intelligently* place containers based on their resource needs and other constraints, optimizing the use of underlying resources. In a scenario where we have a set of microservices with different resource demands, **K8s** can automatically assign these services to different nodes in a cluster, ensuring that each service has the necessary resources without manual intervention. This automation improves efficiency but also ensures a balanced workload across the cluster.

The self-healing capability is one of the standout features of **K8s**. This function enables the system to automatically detect and fix issues such as failed containers. Typically, if a container crashes due to an application error, **K8s** would detect the failure and attempt to restart the container, ensuring minimal downtime and enhancing the reliability of the application. In practice, it might happen that a web service that is running multiple instances across a **K8s** cluster has one of the instances failing. This function would allow for a quick replacement without human intervention and is crucial in maintaining stability of applications in dynamic environments.

Horizontal scaling is another crucial feature for managing workloads in **K8s**. This capability allows the system to automatically increase or decrease the number of running container instances based on the demand. For instance, if a web application experiences a sudden spike in traffic, Kubernetes can automatically deploy additional instances of the relevant containers to handle the load. The opposite is also true and, during periods of low traffic, the number of instances can be brought down to conserve resources. This dynamic scaling ensures that applications remain responsive and efficient under varying load conditions, without the need for manual intervention.

Service discovery allows services within the Kubernetes cluster to find and communicate with each other easily, using DNS names or IP addresses. This automation, once again, simplifies networking configuration and ensures that services can reliably interact, even as they scale and change in dynamic environments.

Load balancing is critical for handling incoming network traffic. For example, if a service is receiving a high load of traffic, **K8s** can automatically spread this load evenly across all available instances, preventing any single instance from becoming a bottleneck. This will in turn result in a more responsive application and a robust user experience, as the workload is managed to accommodate fluctuating demands seamlessly.

K8s automated rollouts and rollbacks help in managing containerized applications with precision. When defining the desired state of your application, such as specifying the number of service instances or the version of a container image, **K8s** actively manages your application to match this state. For example,

when deploying an update, **K8s** incrementally rolls out changes according to this predefined desired state. If problems are detected, it can automatically revert to a previous version, minimizing potential disruptions.

With some key features of **K8s** already presented above, the next focus is on the core components and the architecture that make up **K8s**.

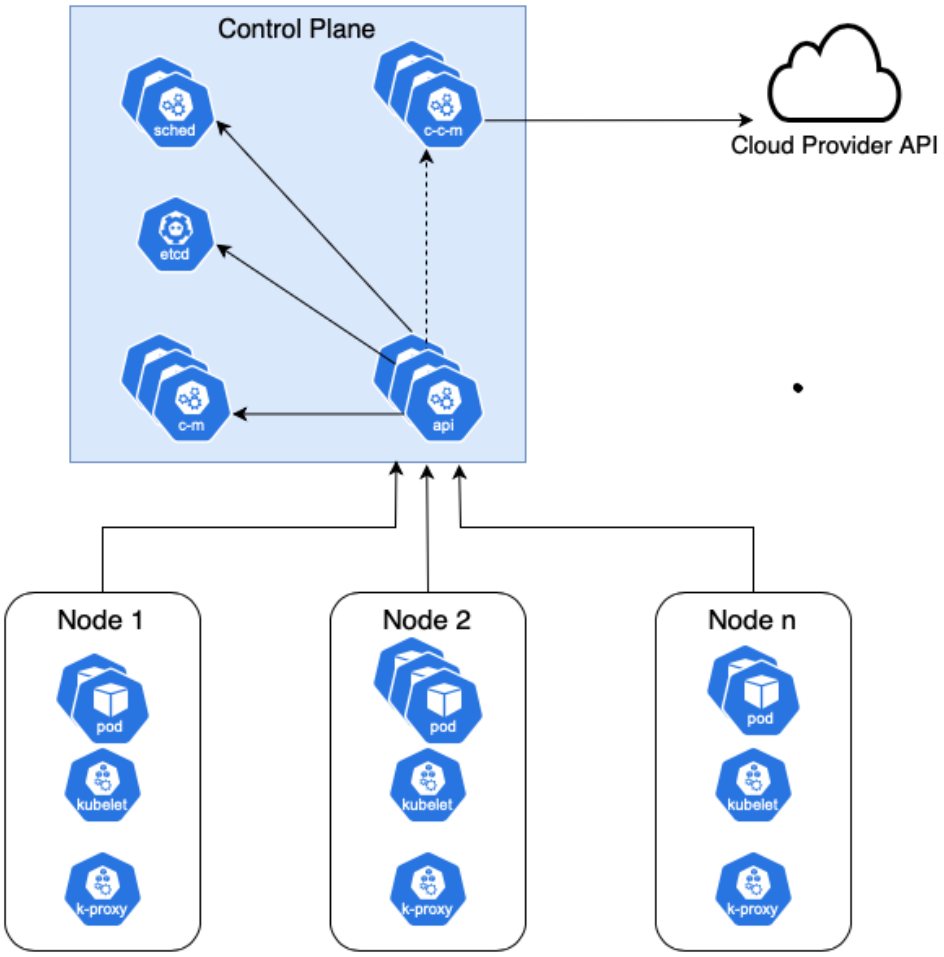


Figure 6: Components of a Kubernetes cluster.

In Figure 6 we can observe the typical architecture of a **K8s** cluster with the roles of the Control Plane and the Nodes. As one might expect, the Control Plane is the brain of the **K8s** system, responsible for global decision-making and management. This component includes the **Cloud Controller Manager (c-c-m)** which integrates the cluster with the cloud provider **API** and manages interactions that are specific to the cloud environment. The **c-c-m** allows for the outsourcing of cloud-related functionality to the cloud provider like for example node lifecycles, routing setup, handling load balancers, etc. If we don't need to run our kubernetes cluster in a cloud environment and prefer to run it on-premises then this component is not necessary. As seen in the picture we can scale the **c-c-m** horizontally if we need to handle increased load in the management layer and also to improve performance and reliability.

Another apparently central component in the Control Plane is the **API** Server, *api* in Figure 6. The **API** Server exposes the Kubernetes **API** and can be seen as the front end for the Kubernetes Control Plane. Since nearly all the components need to make **API** calls to perform their tasks, ensuring that the **API** Server is available at all times is key. This component also scales horizontally i.e., by deploying more instances.

The **Controller Manager (c-m)** is another component of the Control Plane that is meant to run controller processes. It ensures that the current state of the containers, nodes, and other resources aligns with the state defined by the user. It also manages the lifecycle of resources (e.g., pods) and handles node failures. According to the **K8s** documentation there are many types of controllers such as Node Controllers, for managing nodes' lifecycle, Job Controllers, for creating pods to run specific jobs, Service Account Controllers, to create Service Accounts for new namespaces, and many others.

The *etcd* is another service included in the Control Plane whose job is to store the current configuration and run state of the cluster. It is important that the *etcd* is consistent and highly available so that other services have the necessary and updated information necessary to coordinate and manage the cluster. Also, the *etcd* is not a general purpose database, instead it is implemented as a lightweight distributed key-value store.

The last component of the Control Plane is the Scheduler, *sched* in Figure 6. The Scheduler reacts when pods are created with no assigned node and selects a node where they will run on. This decision is based on resource availability and constraints such as hardware/software/policy constraints, data locality or even deadlines. This ensures that the workload is distributed in the most efficient manner possible, in alignment with both the pods needs and the cluster policies.

Now focusing on the nodes, it is evident that there is more than one node because, as we have seen, **K8s** have the ability to expand or contract the cluster size according to demand. The nodes are responsible for running the actual workloads (containers) and usually run two important processes: *kubelet* and *kube-proxy*. The *kubelet* is a process that communicates with the Control Plane and manages the containers on its Node. The *kube-proxy*, to put it simply, is a network proxy that is used to ensure network connectivity to the pods from both the inside and outside the cluster. It also enables service discovery and load balancing within the cluster.

Now we look into the detailed architecture and components that make up a Pod. As we see in Figure 7 a Pod is a collection of containers and volumes that usually share a common resource e.g., file system, IP address.

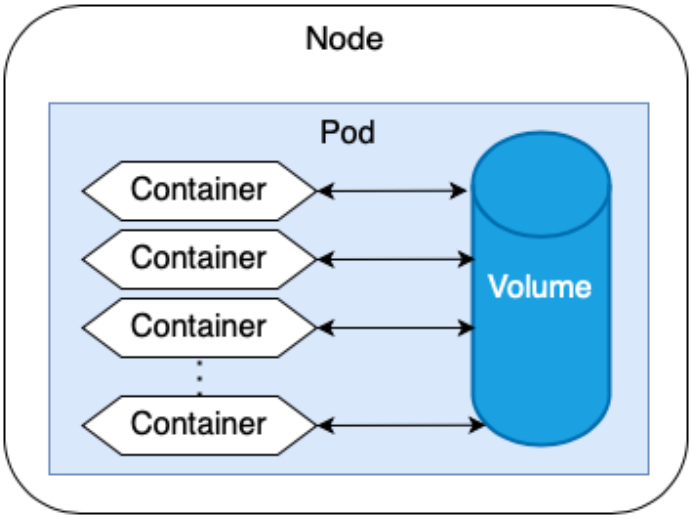


Figure 7: Detailed view of Pod.

In **K8s**, a Pod is the smallest deployable unit of computing that can be created. It represents a single instance in the cluster, but it encapsulates one or more containers that should be managed as a single application. Each Pod is given a unique IP address within the cluster, allowing the containers to communicate with each other using *localhost* as if they were on the same machine which simplifies networks configuration and inter-container communication. Since Pods are ephemeral - can be terminated and rescheduled to different nodes in the cluster as needed - there needs to be a mechanism of data persistence for applications that need to maintain a state, even after a container restart or after the pod being rescheduled to another node. This is where the Volume comes into play. In Docker, volumes are attached to a single container, but in **K8s**, volumes are associated with a Pod which allows data to persist across container restarts within that pod.

After analyzing the components of the **K8s** cluster, exploring how the Control Plane and Nodes function, as well as diving deeper into specific components like the *etcd* or the *kube-proxy*, we have highlighted the orchestration and operational efficiency that Kubernetes brings to cloud computing. The next step would be to look around to find alternatives for this technology. One of the alternatives is Docker Swarm [32], also a container orchestration tool similar to **K8s**, that focuses on simplicity and ease of use. As the name suggests this tool is directly integrated into the Docker platform which means that we can create similar Docker Swarm clusters from the Docker CLI. These two alternatives serve the same purpose of container orchestration but differ significantly in their approach and capabilities. **K8s** provides a more comprehensive ecosystem with detailed management of networking, storage, security, and automation, while Docker Swarm focuses on simplicity and an easier setup and deployment process, making it ideal

for simpler applications.

Throughout this section on Microservices, we have discussed relevant topics such as the role of containers, the practicality and utility of Docker and the need for orchestration with Kubernetes. The Microservices' architecture has changed the development and deployment of applications drastically by providing a modular and highly scalable approach that benefits both businesses and developers alike. With containers, we have seen how the encapsulation and efficient resource utilization were able to streamline the development process. Docker only took this a step further and simplified the container management and addressed the *dependency hell*, promoting an application-centric development model. Kubernetes, with the automated scheduling, self-healing and horizontal scaling emerged as the orchestrator that unified these containers into a functioning system. While this ecosystem has a lot to offer in terms of agility and resilience, it is not without challenges such as the setup and management steep learning curves. Despite everything, the application of all these technologies is widespread across industries, proving once again their significance in building and maintaining robust, scalable and fault-tolerant systems. In conclusion, the Microservices paradigm, enabled by containers and orchestrated by systems like Kubernetes represents a shift in how we build and operate applications at scale.

2.5 Related Work

In this section the focus is on the exploration of existing literature and research in the context of mobility of microservices in **SDVNs**. The selected work has strong ties with the topics of **Multi-access Edge Computing (MEC)**, edge and fog computing, all of them relevant to implementations of the project in a real-world environment. In summary, the concepts of **MEC**, edge computing and fog computing all are used to bring computational capabilities closer to end-users or devices. Instead of relying solely on cloud computing, this decentralization is needed to mitigate latency concerns and optimize data processing efficiency. Edge computing is a broader framework that advocates for the processing of data near its source; **MEC** further refines this principle, focusing specifically on telecommunication environments, particularly optimizing 5G networks. Meanwhile, fog computing is an intermediary layer between the edge and the centralized cloud. All of these terms are often used interchangeably as the factors that help in distinguishing them lie on their focus and architectural scopes with **MEC** tightly integrated with telecom, edge as a broader concept and fog as an intermediary layer. In essence, this section abstracts from the concepts that make part of the related work and focus on the technologies we can use in our context.

In the following paragraphs a brief analysis of selected research is presented and critically analyzed.

From the selected works there is almost no focus on the actual microservices or application architectures used. This dissertation focuses on mobility strategies, so that is where almost all investigation delves into. Additionally, few of these works address the challenges of emulating realistic mobility scenarios, so there is still more research to do in that aspect.

The first research article of this section actually uses the technologies we went over in the past sections i.e., Docker and Kubernetes. In [11] the authors start by recognizing the high demand to process the considerable amount of data that each vehicle generates. This process, since it mostly needs to occur in real-time (for example due to self-driving features), requires an efficient resource utilization as close as possible to where the data is produced initially. The authors propose a fog computing solution for this problem called *Vehicular-OBUs-As-On-Demand-Fogs* that brings cloud intelligence closer to the edge, specifically to each vehicle's **OBUs**. The emphasis on transforming **OBUs** into on-demand fog devices reflects the understanding of the challenges posed by vehicular mobility and the need for lower latencies and localized cloud intelligence. It is the author's understanding that the project was the first that considered the use of containers to host services on vehicles. In summary, this research considers impractical the use of **RSUs** for real-time vehicular applications therefore supporting a hybrid model between the infrastructure and **OBUs**. The experiments showcase a 90% improvement in response time, particularly in scenarios involving **SDN** delays, limited **RSU** coverage, and **RSU** handovers. Key components of the solution include an adapted master election, recovery algorithm, and an evolutionary Memetic Algorithm for efficient service placement. The author suggests that in the future the recovery algorithm can be improved with a reinforcement learning approach and also there is a need to develop a security model for safeguarding vehicular networks.

The work from [33] focuses on the **Connected and Autonomous Vehicle (CAV)** paradigm where **V2X** services are essential to guarantee the benefits it advertises e.g., enhance road safety. In the presented highway scenario the authors delve into the challenges associated with the efficient placement of **V2X** basic services such as latency, limited resources at the edge, and the optimization of deployment. The authors formulate a binary integer linear programming model to minimize delays for multi-component **V2X** applications taking into account the varying demands of traffic conditions. From the realistic simulations, the work demonstrates that the approach is effective in minimizing the delays for the **V2X** services in varying traffic conditions. One of the limitations of the research is the assumption that the rate at which requests are generated for each **V2X** application remains constant. The authors plan to expand the model to incorporate varying request distributions so that it becomes more adaptable to changing communication demands. Also, the very environment where the **V2X** applications are deployed assumes constant

service availability. The future research mainly focuses on optimizing the model to better adapt to the unpredictable nature of **V2X** communication demands.

In [34] the authors characterized vehicles as data hubs with ever-increasing computing needs. The article focuses on the prediction of vehicle mobility within 5G systems in order to minimize service discontinuity and to achieve substantial energy savings. Unlike previous works that adopt a reactive method (e.g., after a user moves to a new radio cell) for migrating **VMs** the authors propose a proactive allocation using a predictive model. The chosen method is a Lyapunov-based algorithm that is both low-complexity and distributed. This approach integrates a mobility prediction algorithm that combines neural networks and Markov chains. The neural network processes radio link measurements within eNB (evolved NodeB) sites, which are essentially 4G and 5G base stations responsible for managing communication with user devices. Meanwhile, the Markov chains statistically characterize the sequence of previously visited eNBs across various locations. Notably, this approach is able to self-adapt and meet specific risk targets, such as minimizing service interruption during handovers. The simulations are then conducted in a realistic vehicular mobility scenario, featuring many vehicles and a dense network of 5G base stations. The results reveal a 50% reduction in energy consumption compared to benchmark migration strategies. As for the future work the authors aim to extend the approach to scenarios involving hardware constraints on computing servers and the execution of computation tasks both on-board and off-board.

In [35] the authors look once again into the integration of **V2X** services in **ITS**. By leveraging **MEC** the services can be brought closer to vehicles, but this comes with the challenge of determining the optimal placement of **V2X** services, especially considering the limited computational resources available at edge nodes. The authors formulate an **Optimal V2X Service Placement (OVSP)** problem in a hybrid core/edge environment as a binary integer linear programming problem. This solution considers computational resource availability, as the authors say this is a current gap in research. The solution comes in the form of a low-complexity greedy heuristic called the **Greedy V2X Service Placement Algorithm (G-VSPA)** that is meant to balance the efficiency with performance. Some metrics taken into consideration by the algorithm are average latency and average node computing utilization. The authors consider their work a pioneering effort, as no prior research has considered the **V2X** service placement problem while accounting for the availability of computational resources in the core and edge nodes of a hybrid computing environment. As for the results, the **G-VSPA** achieved close-to-optimal performance, outperforming both the **OVSP** model and the Genetic algorithm. The authors highlight the need for future work to explore the trade-off between delay performance and cost efficiency in **V2X** service placement, especially considering the cost-efficiency of core placements. Considering higher throughput requirements

would also increase the utility of the system for a future practical application.

In [36] the authors delve into the complexities of vehicular service placement, addressing the unique aspect of unknown service demands. This unknown service demands, say the authors, are intricately tied to their context, meaning that the specific requirements and preferences of vehicle users vary based on factors such as mobility patterns, vehicle types, and other contextual parameters. It is also accepted that migrating computing capabilities to **RSU** at the edge has a positive result in terms of enhancing the quality of experience for the end-users. This work formulates the service placement problem with a context-aware algorithm that relies on an improved utilization of **RSU** resources. The context-aware vehicular service placement algorithm involves two main steps: service demand estimation and service placement. In the service demand estimation step, **RSUs** estimate the service demands based on the observed contexts and user connections. In the context-aware service placement step, the vehicular service provider rents resources based on the estimated demands to optimize the utility of **RSUs**. **RSUs** with well-estimated demands receive more resources, while those with less accurate estimations receive minimal resources for continued observation. The authors conducted a simulation that considered several traffic scenarios in the region of Shenzhen (China) to evaluate the proposed context-aware vehicular service placement algorithm and the results demonstrated superior performance compared to benchmark algorithms, including Combinatorial UCB, RSU-OPT, Oracle, and a Random approach. Furthermore, after sufficient learning, the algorithm can make rental decisions more accurately due to the increase accuracy in estimating the service demands.

To conclude this section on related work it is worth noting that while all the projects presented here seem similar they differ in terms of how they are implemented, how are the simulations performed, the metrics optimized, the test scenarios and many others. The common denominator was in fact the reduction in latency that was generally obtained by bringing resources near the edge. Then the mobility challenge, as we have seen, can be tackled in many ways. This diversity shows the complexity of addressing the specific demands of vehicular applications but, at the same time, opens the door for more exploration and development in this dynamic and evolving field.

Chapter 3

Conceptual Architecture

In this chapter, we begin by detailing the context and nature of the problem addressed by this dissertation. Following this, we introduce a theoretical solution designed to address these challenges comprehensively.

By the end of this chapter the reader will have a complete understanding of the problem and the conceptual framework designed to address it. This chapter is crucial to understand the implementation of the solution that is presented in the next chapter.

3.1 Problem Context

This project consists of designing and implementing a mobility strategy for microservices deployed on an emulated vehicular network. That is one of the main objectives of the project. The other, more subtle objective is to create a framework for testing, visualizing, and gathering information about our simulations. Therefore, our main contribution would be the development of a framework that enables researchers to run their custom mobility strategies for microservices in a vehicular environment.

Developing said framework requires knowledge of several areas, especially from **SDN** and **K8s**. **SDN** will provide us with a programmable interface that can be used to adapt in real-time to the highly variable context of **VANETs**. **K8s**, on the other hand, will handle the deployment and scaling operations of the microservices consumed by vehicles. With these technologies, we can ensure that the microservices are deployed, migrated, and replicated in our cluster, making them accessible to vehicles from any point in the network.

There are also unique challenges posed by deploying microservices in **VANETs**. The dynamic environment characterized by rapid topology changes and variable network conditions, makes it harder for maintaining service continuity and performance. When developing a testing framework we can only aim at being the closest possible to a real world scenario. Relying on the latency and availability results of any framework that is not actually deployed in a real environment is not advised. However, a mobility strategy

for microservices that aims to load-balance vehicles across different **K8s** nodes while maintaining connectivity can indeed be effectively tested with our framework. Additionally, the scalability of the solution is crucial, as the system must efficiently handle a growing number of vehicles and services.

Regarding the microservices applications that should be supported by the framework, the focus should be on enabling real-time data processing that is either generated or consumed by the vehicle. Meeting this requirement can facilitate the implementation of a wide range of services. For instance, gathering vehicle information and making decisions related to safety, as well as providing general information about the surroundings, such as traffic conditions or the location of any points of interest e.g., gas station. Applications related with image processing could be used for tasks like detecting pedestrians or obstacles, but would also require low latency and significant computational resources to process images quickly and accurately. While not related to safety, a video streaming application shares many of the same requirements, such as low latency and stable connectivity. However, it also requires higher bandwidth to ensure smooth and uninterrupted streaming. By supporting these diverse applications, the framework becomes valuable for more research and development use cases.

Now that we have briefly explored the kinds of applications we wish to develop within our framework, we must discuss the key requirements necessary for making this possible. The simulations will run in a local environment, preferably using a single computer. It is essential to have some type of live dashboard of the simulation metrics and state, along with the vehicle mobility. The services must operate without failures, even in the face of changes in network topology, to ensure continuous operation and service availability. Low latency is crucial for communication between microservices, as delays could affect the real-time processing and decision-making capabilities of the applications. Scalability is also a key requirement; the framework must handle an increasing number of vehicles and services efficiently, maintaining performance and responsiveness as the system grows. These requirements are critical to ensure the framework's robustness and usability. If they are met, we could possibly support a diverse ecosystem of applications within our simulation.

In this section, we have explored the key areas relevant to developing our framework, including the diverse applications it aims to support. With this in mind, we are now prepared to delve into the conceptual architecture of the framework in the next section, where we will outline the theoretical design and structure that will address these identified needs and constraints.

3.2 Conceptual Solution

This section starts by defining a conceptual architecture that serves as the underlying infrastructure needed to implement a microservices mobility strategy. We follow this explanation with an overview of the components that could theoretically be a part of the final solution. After that, we describe how data might flow through the system. Finally, we wrap up with a summary of the key points.

In order to implement our testing framework we need at least three main components: a network emulator, an orchestrator and a mobility simulation tool. The network emulator will allow us to create our vehicular network and any other infrastructure needed to support it e.g., access points, switches, **SDN** controllers. The orchestrator manages and deploys microservices efficiently across the emulated network and the mobility simulation tool models the movement patterns and behaviors of vehicles, providing realistic input for how vehicles interact within the network environment. Because there are no readily available frameworks that satisfy this particular use case, we had to design one ourselves. In Figure 8 we can see a conceptual design of the framework.

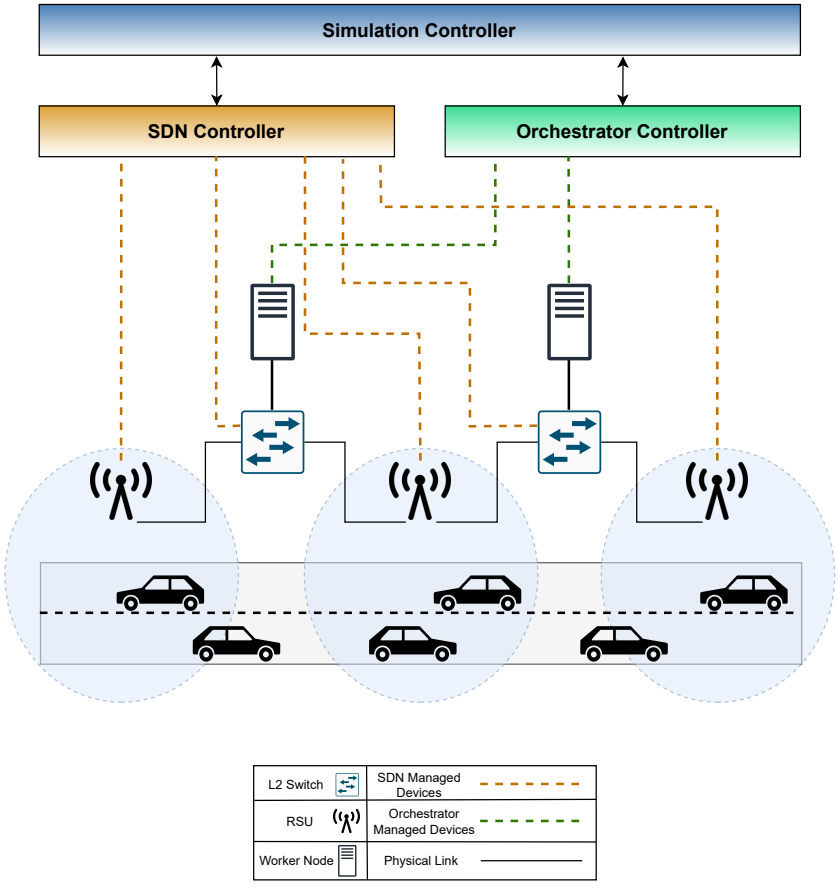


Figure 8: Conceptual Architecture of microservice mobility framework.

There is a clear hierarchy displayed in our conceptual architecture. We must have a simulation controller, which, as the name suggests, is responsible for managing the entire environment. This component is where decisions are made, so we can assume it compiles all the relevant information, such as the location of vehicles, the SDN rules for each network element, and the details of the microservices deployed and available at each worker node. Only with a global view of the current status can the simulation controller make informed decisions regarding the mobility of microservices. Deciding to replicate or migrate a specific microservice assumes that the controller has the ability to interact with other components, not only to gather data but also to take action. In summary, it is expected that the simulation controller can compile all the necessary information, make decisions on microservice mobility, and then collaborate with the SDN controller and orchestrator to enforce those decisions.

The vehicular network, created by a network emulator, relies on the SDN controller to route requests from each network entry point (**RSUs**) to the appropriate worker node. The goal of the architecture is to make the network components adapt to the microservice migration. From the client (vehicle) perspective, the worker node that is handling the requests is irrelevant. The process of writing applications for this framework is therefore simpler since the developer does not have to know if a specific worker node is overwhelmed by requests or if there is another node that is closer and therefore has a better latency. The intelligence, as it is common in SDN scenarios, is centralized in a single component. This architecture has its own advantages and disadvantages that are inherited from the typical **SDN**: centralized control simplifies network management and makes it easier to implement and enforce policies across the entire network but comes with the cost of creating a single point of failure and an attractive target for cyberattacks.

The orchestrator controller is a component rarely seen in vehicular networks. It remains an unknown entity in our framework due to the lack of literature on its use. In our conceptual architecture, we can assume that this component consists of two entities: the worker node and the control node. The worker node is expected to host the microservices and be accessible from within the emulated vehicular network. The control node manages the various worker nodes by enforcing policies that primarily determine where the different microservices are hosted. Although this control node is not shown in the diagram, we can envision it as part of the orchestrator controller, which will expose an interface to receive instructions from the simulation controller and another interface to communicate with the control node.

Apart from all the network elements that we see in Figure 8, there is still another component that must be a part of this architecture: the mobility controller. In order for the vehicles to move and justify the need for replicating and migrating microservices, we need a mobility tool. It is not relevant how is this mobility achieved, it can be offered by the network emulator, or it can be an external component.

In order to further extend our framework concept we designed a diagram that exposes a possible data flow and migration scenario. In Figure 9, we try to show a vehicle that is moving along a road and interacting with different **RSUs**. A microservice is then migrated from the first worker node (1), to the second worker node (2).

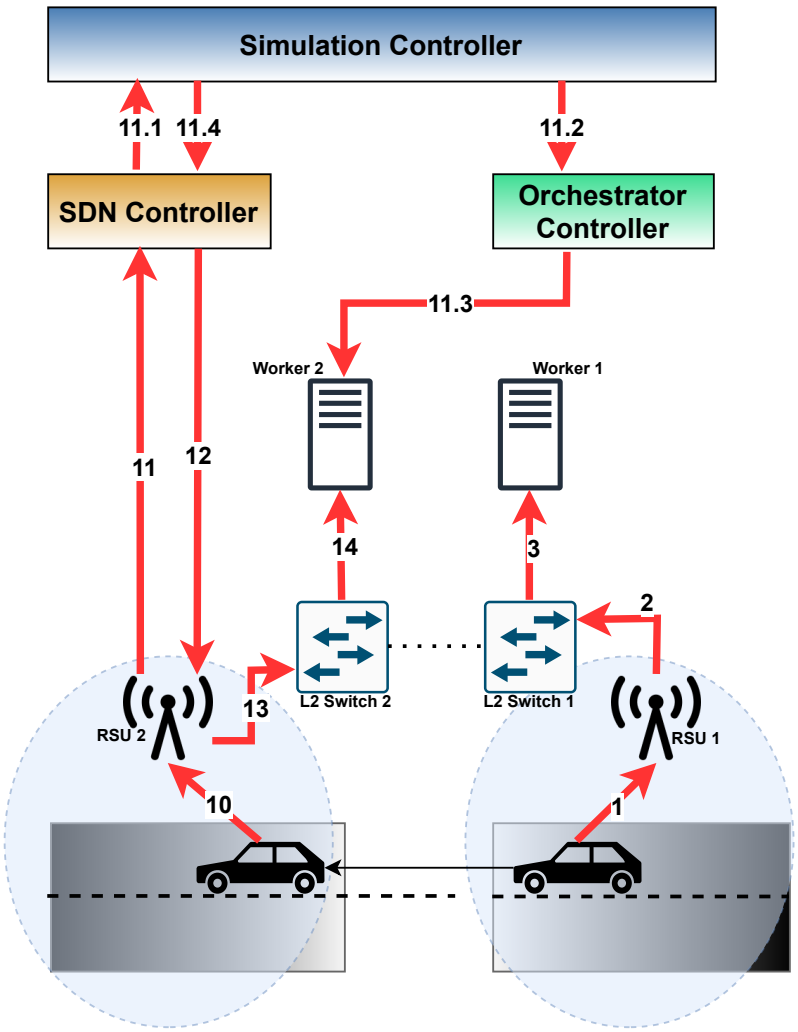


Figure 9: Data flow of the Conceptual Architecture.

In the first request the microservice is already available at *worker 1*, so the **RSU** already knows the destination of the packets that are received by the vehicle. Steps 1 to 3 constitute the best case scenario for the microservice consumer since there is no *decision overhead* and the node available is the closest to our network entry point (*RSU 1*). The vehicle keeps moving and at some point in the future it makes the 10th request. This request is sent to RSU 2, which does not know the packet destination. After the packet being

sent to the **SDN** controller for inspection, the simulation controller gets a hold of this piece of information: the vehicle has now moved, and it is in range of RSU 2. There are some things that are being abstracted here, for example the vehicle might still use worker 1 to maintain connectivity to the microservice, also this assumes that the migration is reactive meaning that the controller only takes decisions after the vehicle moves and does not predict its future position. In any case, the simulation controller now knows about this new vehicle location (11.1) and proceeds to migrate the microservice to *worker 2* (11.2, 11.3). After this process is finished, the simulation controller can install a new **SDN** rule in both the *RSU 2* and the *L2 Switch 2* (11.4). These new rules ensure that any further requests should be redirected from *RSU 2* to *worker 2*, passing by the *L2 Switch 2*. After the flow rules being installed, the packet is returned to *RSU 2* and is now redirected to the *L2 Switch 2* (13) and then to the *worker 2* (14).

In summary, we defined a conceptual architecture and a data flow that can now be implemented using any available technology. There are many alternatives that could possibly instantiate this proposal. We will explore and define one in Chapter 4.

Chapter 4

Framework Architecture and Components

Now that we have a general understanding of the architecture for a solution to the microservice mobility problem, we can delve deeper into the challenges of instantiating this architecture. In this chapter, we begin with a description of the software components used to implement the framework, along with the rationale behind each selection. This is followed by an implementation guide designed to assist the reader in replicating the framework, which includes detailed architecture explanations and diagrams. The chapter concludes with a critical evaluation of the framework.

4.1 Software Components

As we have said in the previous chapter, the framework needs at least three main components: a network emulator, an orchestrator and a mobility simulation tool. Our choices must balance between the learning curve of a new tool and its flexibility to adapt to our requirements, since there is no current single solution that can be used as a framework for our experiments.

At first glance, using Veins [37] seems an optimal choice given that we are trying to implement a realistic vehicular network simulation. The software is open-source and already includes a road traffic simulation tool: **Simulation of Urban Mobility (SUMO)** [38]. So far so good, we have a network emulator and a mobility simulation software, but we still need to add a **SDN** controller and an orchestrator, as well as find a way to have custom applications being executed on mobile hosts i.e., vehicles. The main focus, after being certain that this framework could offer the basic functionality, was to investigate if we could include a microservice orchestrator. Our goal was always to use a real microservice and not a simulated one. This proved harder to integrate with Veins because it seemed impossible to have connectivity between a simulated vehicle node and another network outside of Veins, where the microservices would be hosted. Along with the lack of documentation and experience on this specific issue, we were forced abandon this conquest early on and look to a simpler approach.

Simplicity is the key feature that Mininet-WiFi [39] has to offer. Mininet-Wifi is a fork of the Mininet SDN network emulator [40] and extended its functionality by adding virtualized Wi-Fi Stations and **Access Points (APs)** based on the standard Linux wireless drivers and the 80211_hwsim wireless simulation driver. Since Mininet-WiFi extends the Mininet code base, it still supports all of Mininet's functionality. Mininet-WiFi allows us to create a realistic virtual network of both mobile Wi-Fi stations (vehicles) and **APs** (**RSUs**). We can, similarly to Veins, create our own custom **SUMO** simulations and launch them from within Mininet-Wifi so that we are able to simulate the movements of the vehicles. But Mininet's versatility extends beyond just emulating these wireless mobile networks. One of the much-needed features that is made available by this software is the ability to connect to external networks outside the emulated network, meaning that our vehicular network nodes can reach other networks that do not need to reside inside Mininet-WiFi. It will be clear in the following paragraphs why do we have this requirement, but for now we can say that it is a necessary requirement in order to have real microservices in our simulation.

Integrating **SDN** in our network is also made easy by Mininet-WiFi. We can program our custom controller logic and then run it while we start the network. This process is seamless, and we can avoid unnecessary complications by using one of the most recommended **SDN** Controllers for Mininet: Ryu [41]. Ryu is a productivity-focused solution to software defined networks. It provides a well-defined **API** that makes it easier for developers to create and manage control applications. Ryu supports many protocols for managing network devices, such as OpenFlow, Netconf and OF-config. OpenFlow seems to be the standard or at least the most used **SDN** protocol in both academic research and industry applications. In our project we will use OpenFlow version 1.3 due to its balance between features and stability. There are newer versions of OpenFlow, but with version 1.3 we have all the features (and more) that we need without the added complexity of newer iterations. Also, this version is widely supported and has become a standard in many SDN environments.

The decision to use the Ryu controller in this project comes naturally due to previous contact with the software. It is a user-friendly and flexible tool that emphasizes productivity, which is very important in an ambitious but rather time-constrained project. Ryu also offers a web **API** that allows us to manage our network devices. This ends up being favorable to us since we have in mind a Simulation Controller (Figure 8) that is capable of controlling all aspects of the simulation be it the network traffic flow or the orchestration of microservices.

Now that we have a network emulator, a vehicle mobility simulation and a **SDN** controller, we need one last component: the microservice's orchestrator. We had already decided to use Kubernetes as an orchestrator, since it offers the most powerful platform for managing containerized applications, as

mentioned on section 2.4. By using Kubernetes we can test our framework locally and, in the future, we can deploy it at almost any cloud provider (e.g., Google Cloud Platform, Azure, Amazon Web Services). For our local environment there are a few options, such as Minikube [42] and Kind [43]. Both of these tools allow us to simulate a full Kubernetes environment on a single machine or small cluster. Upon further investigation and experimentation it was decided that Kind was the preferred tool. Minikube works by setting up a virtual machine to emulate our cluster. For each cluster node that we have, Minikube will create a separate virtual machine. If we, for example, want to have a realistic vehicular network we need more than just one or two worker nodes. This is resource intensive and in practice was not possible by using the hardware available. On the other hand, Kind offers a lightweight approach to multi-node clusters. When using Kind, the cluster nodes are Docker containers, and so we can easily scale our cluster network. In terms of usability and features, both the frameworks are comparable.

The software components were all presented in this last few paragraphs, but we still need to write custom software to integrate and manage the framework. The preferred programming language to do that was Python 3 [44]. This is the logical choice, since almost all the software tools chosen are either written or have a powerful and well-defined Python **API**. Python is also a flexible and productive programming language that allow us to quickly iterate over ideas for our implementation.

In summary, the framework will use a Network Emulator (Mininet-WiFi), a traffic simulation tool (**SUMO**), a **SDN** controller (Ryu) and a microservice's orchestrator (Kind). All the management and general purpose code will be written in the Python programming language.

4.2 Implementation Guide

This section describes the implementation of our framework from a practical standpoint. By following the steps ahead one can configure the same setup and hopefully achieve the same results described in the following chapters. Before starting to describe the implementation, it must first be noted the hardware available. There were two different machines available, a MacBook Pro M3 (arm64) and an older MSI Laptop (amd64) running Manjaro Linux. The MacBook is packing 36 GB of RAM, which makes it our desired machine to run our experiments.

Trying to use the MacBook however proved difficult. It was impossible to install Mininet-WiFi, so the first alternative was to use a virtual machine. Since the chip architecture is ARM, the VM was actually translating in real-time the instruction set from x86_64 to ARM, which resulted in a rather slow experience. Nevertheless, it was possible to take the VM made available by Mininet-WiFi and convert it to a format that

could be used by UTM (the virtualization software) [45]. The problems came when the development of test scenarios was introduced. When creating a network with Mininet-WiFi that contains wireless network devices, we must rely on a software module to simulate the wireless medium. There are two approaches for simulating the medium: Traffic Control (TC) and wmediumd. The team behind Mininet-WiFi strongly recommends the use of wmediumd due to its advantages over TC. Some of them being the isolation of the wireless interfaces from each other, the implementation of a back-off algorithm, the decision of the association based on the signal level and many others, making it more realistic than its alternative. Unfortunately, when we started testing the framework with simple simulation scenarios, there was an error related with the initialization of wmediumd. Upon further investigation, it was apparent that it could not be fixed without rewriting a part of this tool and also its possible interaction with the mac80211_hwsim kernel.

The next possible solution was to simply use another machine. The MSI Laptop has almost 10 years and its condition is not the best but, the success of the framework's configuration in this machine proved that it is a lightweight solution for our problems and that almost anyone could replicate this setup at a minimum cost. At first, we decided to use a virtual machine so that our framework could be easily shared among those interested in replicating our experiments. Similarly to all the other attempts, this failed due to insufficient resources, and we were forced to set up the framework directly on the host machine. This ensured a better performance and eliminated all the issues we had with virtualization overhead and hardware limitations. But once again it was not straightforward to install all the necessary dependencies, and we had to manually edit a few installation scripts.

The installation of **SUMO** was performed via AUR and Kind was installed with a pre-built binary available on their Quick-Start Guide. The installation of Ryu was trickier and required us to clone their GitHub repository and run the following two commands:

```
$ pip install -r tools/pip-requires
$ sudo python setup.py install
```

The first instruction is necessary to install specific version of all of Ryu's dependencies. A specific version of pip (20.3.4) is installed, which may interfere with your local environment. Of course that before installing any tools we have created a virtual environment using Conda [46]. Also, both Mininet-WiFi and Ryu require a slightly older version of Python. In our experiments we came to the conclusion that version 3.8 was the most stable, so that is the one we are using.

Now, last but not least, the installation of Mininet-WiFi. Since this software was meant to be used on Ubuntu, there is no installation script for Manjaro, and so we had to adapt the current script(s). This

process of installing Mininet-WiFi can be comprised in five steps: Mininet-WiFi dependencies, Mininet dependencies, OpenFlow, OpenvSwitch and wmediumd. We can easily clone and compile from source the last three dependencies. Installing both Mininet-WiFi and Mininet dependencies requires us to check which packages are being installed in the script and install them manually in Manjaro.

With all the tools installed and working properly, we can start the development of our simulation controller, responsible for managing both the vehicular network and the microservice's orchestration. But before that, let's take a look at Figure 10 on how the different tools interact with each other.

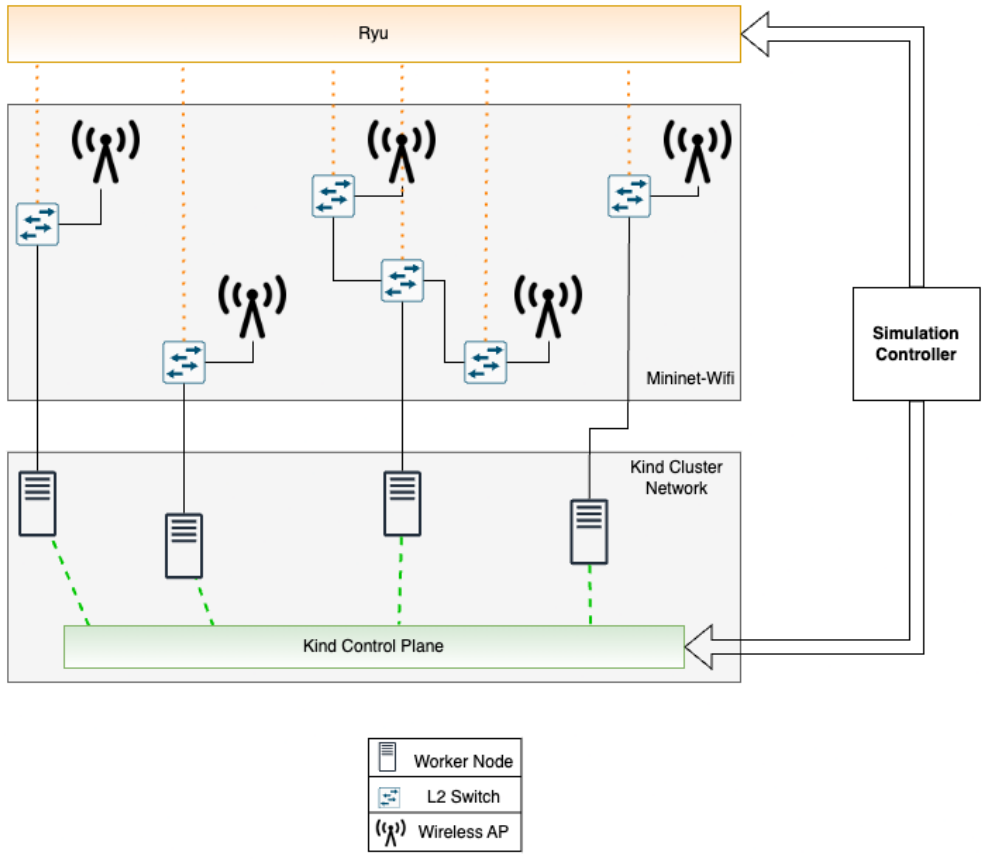


Figure 10: Implementation architecture with Mininet-WiFi, Ryu and Kind.

The vehicular network is being emulated with Mininet-WiFi and is composed of several layer two switches and wireless **APs**. The connections between these components are irrelevant, the only requirement is that each **AP** (or its associated switch) needs to have connectivity with a worker node from the cluster network. Each network component is registered with the **SDN** controller (Ryu) when the network starts. Regarding the cluster network, there are worker nodes and a control plane. The worker nodes will host the microservices, and the control node is responsible for enforcing the orchestration policies.

When developing this framework, the goal was to simplify the process of starting a new simulation.

Therefore, with a single command, we can start a new simulation, ensuring that the cluster network, the vehicular network and the **SDN** controller are all initialized. To achieve this we have implemented configuration files. When we want to run a simulation we can simply use the following command.

```
$ sudo python app.py <config-file-path>
```

The configuration file is expected to be a simple JSON file containing yet another three configuration file paths, one for each main component: Kind, Mininet and Ryu. The kind config file is a YAML file mainly focused on the definition of the cluster network, e.g., its name and the number of worker nodes. The Ryu configuration is actually just the file path of the python module that we want to use, and we can also specify more than one module by separating them with an empty space. Lastly, the simulation configuration file contains a path for the Mininet configuration file. This configuration contains the path for the **SUMO** simulation file, the number of vehicles we want to simulate and, most importantly, the desired network configuration. In this network configuration we can specify parameters such as the number of **APs** (**RSUs**), their geographic position, the worker node that they're closest to and the ID of the AP that they're linked with. Still in the Mininet configuration, we can also specify the propagation model for wireless signals and other general configuration for the telemetry window.

After the script is executed and all the configuration files are read, we start creating the cluster network with Kind. We first check if the same cluster already exists and, if so, we check if we need to overwrite it. The cluster creation includes the creation of the worker nodes and control plane, a service and a deployment. While the service is responsible for exposing our application, the deployment describes a desired state for our cluster, for example, by specifying the number of instances running on the worker nodes or where these instances are running. The migration and replication of a specific application will come as a result of a new deployment.

After creating the cluster network, we launch the **SDN** controller using the *ryu-manager* utility. Finally, we launch Mininet-Wifi controller that is responsible for starting the vehicular network with the previously given configuration. In a separate thread we also start a control module. This control module will be the brain of the whole simulation. Here we gather information about the real-time location of the vehicles so that we can make decisions regarding the replication or deployment of a given application, depending on the simulation objective. A simple control module would check the **AP** in range of a given vehicle. It would then check the worker node that is closer to this **AP** and migrate/replicate the application on this node, if necessary.

Although we are able to understand the components that make up this framework, we might still not grasp its limitations. In the next section we go over the main advantages and disadvantages of the

framework.

4.3 Framework Evaluation

Throughout the development of this framework, there were several critical moments that required us to either improvise solutions or select between alternative paths, each presenting unique trade-offs. In this section, we will start by presenting some limitations of the framework, a few advantages and, in the end, improvements that could increase the usability of the platform.

Starting with the limitations, we have mentioned that the microservices would be hosted by the worker nodes but did not specify how will they be made available to the nodes. When starting the simulation, one of the steps is to load the docker images of the desired applications into each worker node. This process of downloading the docker images to each node is slow, and it would not be accurate to only do this when we are replicating or migrating a service to a node. The result is that, when the simulation starts, all worker nodes know the applications referenced by the Kubernetes service. At the beginning of the implementation we experimented with a docker image repository that would host the images so that they could be downloaded from anywhere with an internet connection. This was not possible and Kind only allows to load docker images into the cluster nodes via its command line interface.

When first studying the mechanics of microservice migrations in our framework, it became apparent that **K8s** was not intended to be used as a way to move our application to different nodes in different geographic locations in order to obtain a lower latency. The primary goals of replication and migration in Kubernetes are centered around maintaining application availability, ensuring fault tolerance and optimizing resource utilization within a cluster. Kubernetes achieves this by automatically replicating pods and migrating workloads to healthy nodes within the same cluster, often prioritizing efficient resource distribution rather than geographic proximity to end users.

In contrast, our framework's goal focuses specifically on reducing latency by migrating microservices closer to the end user, leveraging geographic distribution to optimize performance. We are effectively deploying microservices to specific worker nodes and then, when a vehicle moves into the range of another node, we move this service to another specific node. There is no consideration of, for example, resource utilization. Although in one of the test scenarios we use the framework to load balance incoming traffic from the vehicles. To conclude with this issue, the way we are using **K8s** here (where we specifically choose which node will host a microservice) is not the intended use case for this technology.

Another issue we encountered was the proximity of each worker node to an **AP** (**RSU**). Initially, during

the early stages of development, we believed it would be feasible to run each node on a separate virtual machine. This setup would allow the worker nodes to be in distinct networks, providing the desired level of separation and justifying the need to migrate services. However, this approach did not work due to our local setup, which not only lacks sufficient resources but also relies on Kind, a tool that does not support separating nodes into different networks. Ultimately, this approach didn't make much sense. As our ideas evolved, we figured that if all nodes had to be within the same network, we would need a predefined mapping of each node to a geographic location or set of **RSUs**. This means that our framework relies on a configuration file that contains this information, enabling it to determine the appropriate worker node to use based on the traffic's originating **RSU**.

One of the later issues that we found was the way we were handling the migrations of microservices. While replications demand only for an application to be made available at a given node, migrations usually involve the transfer of data related to each client's session. We found it impossible to do this using our framework. The applications we develop to test the framework are stateless, meaning the client must store any relevant information. For example, in a video streaming scenario, we would use the **Dynamic Adaptive Streaming over HTTP (DASH)** protocol. So, in summary, when we migrate a service we are essentially just deleting the existing deployment that assigned a specific node to host a particular service and creating a new deployment for a different node. Similarly, for replication, we create a new deployment without deleting the previous one.

To conclude with the limitations of the framework, we should note that this setup is not easy to replicate. If we start the process using the Mininet-Wifi virtual machine, that comes pre-configured with Mininet-Wifi and its dependencies, the process is straightforward. On the other hand, if we don't use Ubuntu and can't use the virtual machine, we basically have to figure out how to install all the required tools. This is far from ideal, but it is the only serious and difficult limitation of this framework to overcome.

Focusing on the more positive side of the framework, there are two very strong features that must be pointed out. First, it is a very lightweight alternative to test a fairly realistic vehicular network. It is also open-source, meaning that we have access to the entire codebase that makes up the framework and can, at any time, modify the source code or even build on top of it and contribute to the open-source community. The fact that it supports a variety of different applications is also one of its key advantages that will be displayed in the following chapter.

To conclude this section on the evaluation of the framework, we recognize that while the current approach provides a functional method for handling microservice migrations and replications, there is significant room for improvement in making these migrations more realistic. The transfer of state between

nodes during migrations would be a key improvement. This feature could be added to the simulation controller, assuming that the services would have endpoints to handle the status upload. Furthermore, to facilitate the distribution of the framework, we could create a pre-configured virtual machine with all necessary components, configurations, and dependencies. This would make it simpler for others to experiment with the framework and contribute to its further development. By tackling these two areas we can significantly improve the realism and usability of the framework, ultimately making it a reliable tool for the evaluation and optimization of microservice's mobility in vehicular networks.

Chapter 5

Experiments and Results

The framework could not be concluded without a thorough evaluation of its capabilities. To that end, we developed five different experiments, each of them unique in terms of functionality, displaying a broad range of applications that can be tested using this framework. Before presenting the scenarios, we briefly go over the common structure of all of our experiments in Section 5.1. We follow with five sections, each dedicated for a specific scenario, containing a detailed description of the experiment and the results obtained.

5.1 Experiments Framework

The experiment's framework consists of a set of configuration files and Python classes that are shared among all the scenarios. As it was mentioned in the previous chapter, we developed a simple script that initiates all the needed components of a specific scenario. This script takes as an argument the path for a configuration file, as shown in Listing 5.1.

```
{
    "kind-config" : "kind/configs/<filename>.yaml",
    "mininet-config": "mininetwf/configs/<filename>.json",
    "sdn-controller": "sdnRyu/<filename>.py"
}
```

Listing 5.1: Initialization script configuration file example.

Depending on the scenario we are experimenting with, the script instantiates its class and calls its *run* method. This method is implemented by all the scenario specific classes (e.g., Replication, Migration, etc.). Each of these classes also inherits the *Scenario* class, which is responsible for all the common functionality such as creating the Kind cluster given a specific configuration file, adding deployments to the cluster or creating **SDN** flows, among many other operations. The main purpose of this class is to improve code reusability and abstract the concrete scenarios implementations.

For the *mininet-config* file in particular, we have a custom configuration format with the following main parameters, as described in Listing 5.2.

```
{
    "sumoConfig": "sumo/<filename>.sumocfg",
    "cars": { "count": 1 },
    "aps": [
        {
            "id": "1",
            "position": "2000,1500,0",
            "channel": "1",
            "linkTo": "2",
            "kindNode": "1"
        },
        {
            "id": "2",
            "position": "1500,1300,0",
            "channel": "6",
            "kindNode": "1"
        }
    ]
}
```

Listing 5.2: Mininet configuration file example.

There are a few other parameters that we can customize, that are not displayed in the previous JSON file. Apart from those, the main parameters are the file path for the **SUMO** configuration file, the number of vehicles we want in the simulation and the access point configuration. For the *aps* list, each element represents an **AP** with a unique identification and position on the **SUMO** map. We can also specify the wireless channel to reduce the interference between neighboring **APs**. The *linkTo* parameter is optional and is used to create links between **APs**. And at last, the *kindNode* parameter represents the closest or, in our framework, the preferred worker node for the **AP**.

Given this codebase we can start focusing on the particular topologies and strategies for each of the following scenarios.

5.2 Scenario I: Replication

The first scenario was meant as a proof-of-concept to test the overall usability of the framework. In this experiment we build a simple topology with a single car moving on the same road. Along this road, a few **APs** are scattered. As we know from the previous section, each **AP** is associated with a worker node that hosts our microservice. Our single vehicle is consuming that microservice while it travels and passes

through several **APs**. The goal of this experiment is to replicate the microservice on the closest node to the vehicle, so that the latency is always minimal.

As for the topology used in this scenario, we decided to start with a familiar sight and choose the city of Braga (Portugal), to be specific, the surrounding roads of the Universidade do Minho. Figure 11 presents the area of the experiment.

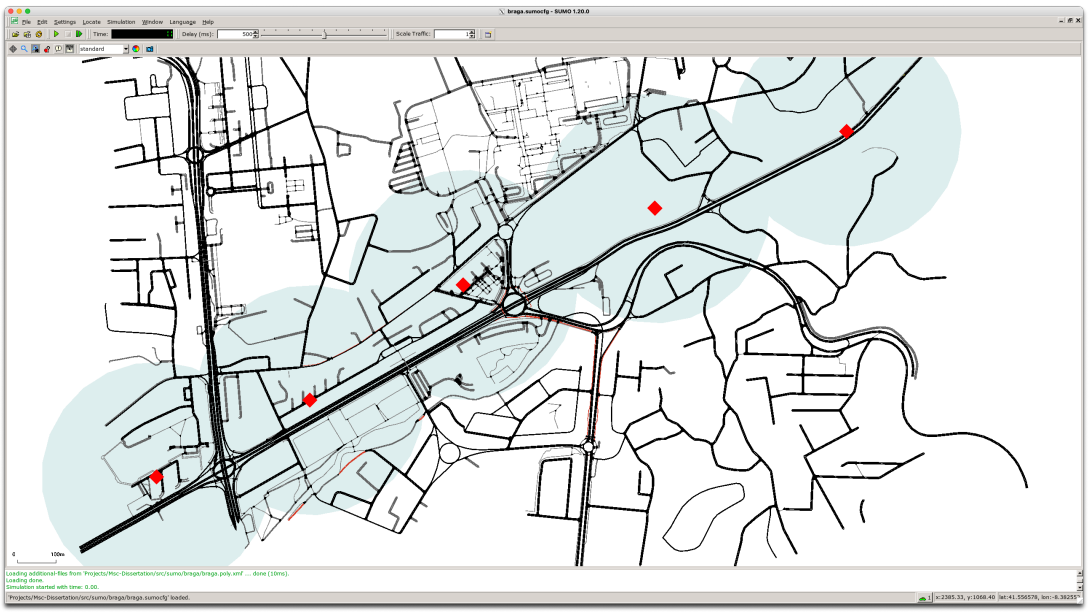


Figure 11: **SUMO** map of the city of Braga.

The map was retrieved from OpenStreetMap [47] with the help of the utility *OSM Web Wizard* from **SUMO**'s repository. This tool makes it easy to crop a specific portion of the map and also extract only the elements we need for our simulations. In our case we decided to export only the network of nodes that represent each intersection and define our roads. Also, in order to keep the simulation lightweight, the buildings, pedestrians, vehicles and all the other extra elements were not exported. Later on we generated the vehicle route manually using the *netedit* utility from **SUMO**.

For this scenario we have five **APs**, each represented by a small red square on the map. The grayish circles surrounding each **AP** represent their area of reach. It was defined for all the scenarios that the range of the **APs** is three hundred meters in a straight line, so we draw three hundred-meter radius circumferences over the map. The objective of this was to help visualize when vehicles are approaching or leaving a specific **AP**'s reach.

Now that we have seen the topology that we are going to use for the first experiment, it's time to launch the simulation, with the same command we have seen in section 4.2.

```
$ sudo python app.py scenarios/configs/1_Replication.json
```

For this first scenario, let's go over each component's initial state. The first component to be created is the Kind cluster. In this scenario, to support the five **APs** we have created three worker nodes that will be hosting services. Since Kind creates a container per node, we can first verify the creation of our nodes with the following command. The result of this command is presented at Table 1.

```
$ docker container ps
```

IMAGE	PORTS	NAMES
kindest/node:v1.31.0	127.0.0.1:6443->6443/tcp	poc-replication-control-plane
kindest/node:v1.31.0		poc-replication-worker
kindest/node:v1.31.0		poc-replication-worker2
kindest/node:v1.31.0		poc-replication-worker3

Table 1: Docker containers created for the first scenario.

With the previous command we were able to verify the created containers that support our Kubernetes cluster network. A total of four containers were created, one per each worker and a control plane container to handle the management operations of the cluster. The following command uses the Kubernetes command line tool, *kubectl*, to view the IP addresses for each of the created nodes, their status, and a few other less relevant parameters that were omitted from Table 2.

```
$ sudo kubectl get nodes
```

NAMES	STATUS	INTERNAL-IP
poc-replication-control-plane	Ready	172.18.0.4
poc-replication-worker	Ready	172.18.0.2
poc-replication-worker2	Ready	172.18.0.3
poc-replication-worker3	Ready	172.18.0.5

Table 2: Kubernetes nodes and their IP addresses, all in *Ready* state.

At this point, we can already assume that the cluster network with the desired number of worker nodes was created. Now, we need to verify the service creation. The service's job is to provide stable network

access to a set of pods, regardless of their dynamic lifecycle. To view the current Kubernetes services we use the following command. The output follows on Table 3.

```
$ sudo kubectl get services
```

NAME	TYPE	CLUSTER-IP	PORTS	SELECTOR
kubernetes	ClusterIP	10.96.0.1	443/TCP	<none>
mysimpleserver-service	NodePort	10.108.161.116	8080:30001/TCP	app=mysimpleserver

Table 3: Services available in the Kubernetes cluster.

In our case, we see two services: the *kubernetes* service of type ClusterIP, which allows internal components within the cluster to communicate with the Kubernetes **API** server, and the *mysimpleserver-service* of type NodePort, which exposes the application to external traffic by mapping an internal port (8080) to an external node port (30001). With this output we can assume that the services are properly configured, and we have both internal and external communication to the necessary components. Without the service, we can't reliably access the pods in the cluster, as their IPs are ephemeral and subject to change.

However, services alone aren't enough; we also have the need for a deployment, which manages the desired state of the application. With the following command we verify the creation of the deployment. The output follows, on Table 4.

```
$ sudo kubectl get deployments
```

NAME	READY	IMAGES	SELECTOR
mysimpleserver-deployment-1	1/1	mysimpleserver:latest	app=mysimpleserver

Table 4: Deployments managing pod replicas in the Kubernetes cluster.

This deployment instructs the kubernetes control plane to maintain one replica of the mysimple-server:latest docker image, which as the name implies, is a simple HTTP server that will be used on this experiment. The deployment will not only ensure that the correct number of pod replicas are running, but it will also handle updates and provide fault tolerance.

The pods are the components that handle the actual workload. To see the current existing pods we can use the following command and observe the results at Table 5.

```
$ sudo kubectl get pods
```

NAME	STATUS	IP	NODE
mysimpleserver-deployment-1-5b86f964b8-jl6gk	Running	10.244.1.3	poc-replication-worker

Table 5: Pods running in the Kubernetes cluster.

We can once again confirm that one pod is running smoothly on the desired worker node. Since the service routes traffic via the app=mysimpleserver selector, even if the pod's IP changes or if more pods are added, the traffic will continue to be distributed to all active instances of the application. We might also see this IP again in the future, since our simple HTTP server will return the IP of the pod that is running the service.

Now that we have successfully verified the creation of our Kubernetes cluster, we can inspect the **SDN** flows that exist at each **AP**. As we have mentioned previously in this section, we have five **APs** in our Mininet-WiFi network. Each of these **APs** can be connected to a switch that is in turn connected to a broader network. In this initial scenario, to simplify the experiment, all the **APs** are connected in sequence: the first to the second, the second to the third, and so on. Each **AP** is configured as a wireless Open vSwitch, a virtual switch that supports the OpenFlow protocol. We can control the flow of packets between **APs** by defining custom rules and policies. Using Ryu's REST **API**, we can monitor how many **APs** have been registered with the controller, as well as their unique identifiers. Later on we will inject flows to control how traffic is forwarded across the network. We can start with a GET request to the switch's endpoint, the response is given at Listing 5.3.

```
$ curl -X GET http://localhost:8080/stats/switches
```

```
[
    1152921504606846977,
    1152921504606846978,
    1152921504606846979,
    1152921504606846980,
    1152921504606846981
]
```

Listing 5.3: Response to the GET `/stats/switches` endpoint containing the list of switch IDs.

The IDs of the **APs**, contrary to the typical switches, start on the number *1152921504606846977* and continue incrementally, instead of starting on number 1, which would make it more readable. Nevertheless, we can continue our exploration of the **API** by listing all flows of each **AP**. Since they are all the same, at least at the start of the experiment, we will request the flows for the first **AP** using the following command. The response follows at Listing 5.4.

```
$ curl -X GET http://localhost:8080/stats/flow/1152921504606846977
```


```
{
    "1152921504606846977": [
        {
            "priority": 0,
            "cookie": 0,
            "idle_timeout": 0,
            "hard_timeout": 0,
            "byte_count": 3432,
            "duration_sec": 43,
            "duration_nsec": 390000000,
            "packet_count": 32,
            "length": 80,
            "flags": 0,
            "actions": [
                "OUTPUT:CONTROLLER"
            ],
            "match": {

            },
            "table_id": 0
        }
    ]
}
```


Listing 5.4: Response to the GET `/stats/flow/<dpid>` endpoint containing the list of flows of the given switch.

The **AP** *1152921504606846977* (and all the others) contains only one flow entry on its table. This rule has the lowest possible *priority*, zero, which indicates that it does not take precedence over any other rule i.e, is the last resort rule. The *cookie* is an arbitrary value assigned to the flow, often used for tracking or filtering specific flows, but in this case, it's set to zero, indicating no special identification. Both the *idle_timeout* and *hard_timeout* are zero, meaning that this flow doesn't expire based on activity (idle timeout) or after a fixed duration (hard timeout). It will persist until explicitly removed. The *byte_count* indicates that already 3432 bytes have matched this flow rule since it was installed, and the two *duration* properties express the lifetime of this rule. The *packet_count* indicates the total number of packets that

have matched this flow and the *flags* being zero means that there are no special flags are set for this flow. The most important fields of all are the *actions* and *match*. The *actions* contain the value *"OUTPUT:CONTROLLER"*, that means that packets matching this flow are sent to the controller. And the *match* field is empty, meaning that this flow matches all packets by default. Since that the flows are matched by priority and then by specificity of the match, we have to create flows with higher priorities or more complex matches (trivial) in order to have the expected impact. Also, since we are using a single flow table, only one flow needs to be matched per packet. Once a packet matches a flow, no other flows will be considered, even if they are present in the table. Therefore, we can assume that only one match is needed to direct the packet correctly, and subsequent flows won't be evaluated for that packet.

Now let's continue with our initial verification and check if our network was created and is working as expected. To better visualize our simulation, we enabled the telemetry feature from Mininet-WiFi, as we can see in Figure 12.

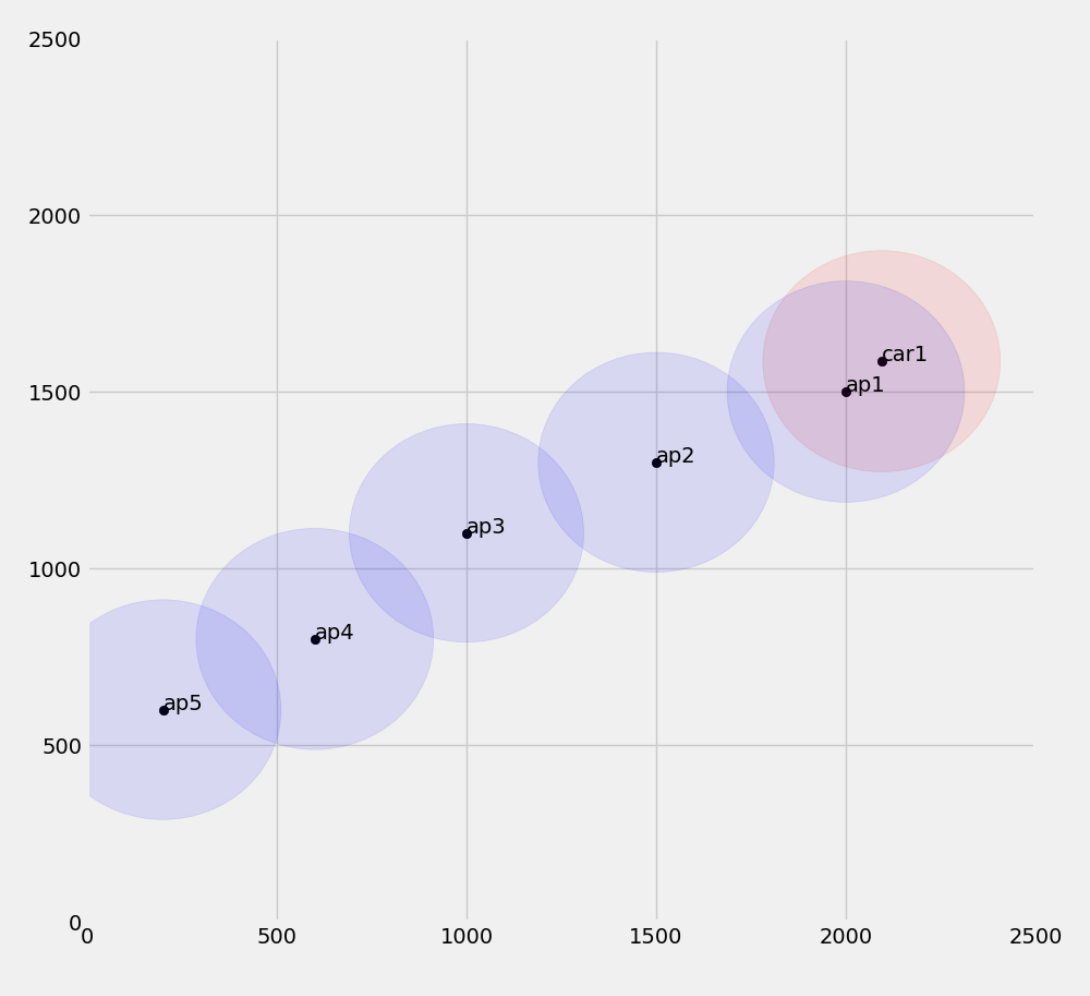


Figure 12: Visualization of network elements for the Replication scenario.

We can also use Mininet-WiFi's integrated terminal to check the network nodes and their connections. To that end, we can use the *links* command.

```
mininet-wifi> links
ap1-eth2<->ap2-eth2 (OK OK)
ap2-eth3<->ap3-eth2 (OK OK)
ap3-eth3<->ap4-eth2 (OK OK)
ap4-eth3<->ap5-eth2 (OK OK)
car1-wlan0<->wifi (use iw/iwconfig to check connectivity)
nat0-eth0<->ap1-eth3 (OK OK)
```

There are a couple observations to be made about the previous result. First is that it appears that all our nodes were created and are linked correctly. The second, most important, is that of a *nat0* object, linked to *ap1*. The NAT feature, that was missing on other frameworks, allows us to route traffic from our Mininet network to our local network. This makes it possible for our vehicle to interact with the Kubernetes worker nodes and therefore is essential for this project. Unfortunately, we cannot create multiple *nat* nodes and have them linked to each **AP**.

Before starting moving the vehicle, let's check if the vehicle can reach the worker node. Remember that we created a deployment that ensures one pod replica on the *poc-replication-worker*, which has the IP address *172.18.0.2*. We can make a GET request to the *getPodIP* endpoint of this service. Also, given our Kubernetes NodePort service, we have to use port 30001. The following *curl* command makes the request, the response follows at Listing 5.5.

```
$ curl -X GET http://172.18.0.2:30001/getPodIP
```

```
{
    "PodIP": "10.244.1.6"
}
```

Listing 5.5: Response to the GET `/getPodIP` containing the IP of the pod that is hosting the service on the *poc-replication-worker* node.

From the output of the *curl* command we notice that the pod's IP changed, compared with the address from Table 5. There is a very simple explanation: Pods are ephemeral, meaning they do not persist through events like system restarts.

To make it easier to interact with the service, we developed a simple client application that makes this previous request in a loop, and saves all the responses in a log file. Another important detail that we

should mention is that the client application always uses the same IP address to interact with the service. It's the job of the framework to create flow rules, deployments and any available mechanisms to redirect the packets to a better, closer worker node. The address used is always guaranteed to belong to a node that has an active deployment with the services we are testing at the moment. Subsequent flows that modify the destination or source IPv4 addresses do so in relation to the so-called bootstrapped IP.

After starting the client application and verifying that the vehicle reaches the service on the *poc-replication-worker* node, it's time to start the simulation i.e., vehicle movement. For demonstration purposes, we stop the simulation at the position seen in Figure 13.

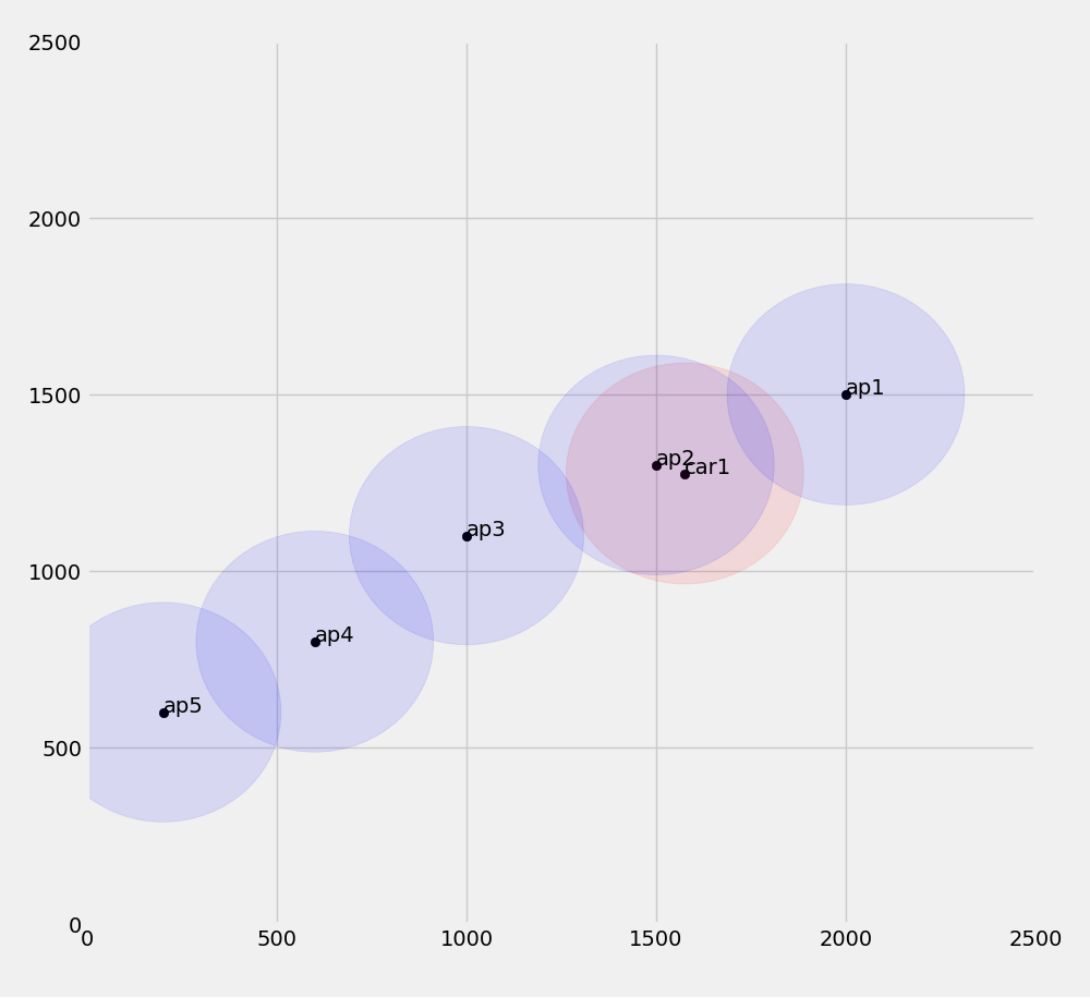


Figure 13: Visualization of vehicle position in range of *ap2*.

The vehicle is well in range of *ap2* and can't reach the previous *ap1*. The client application is still running normally and displays the same pod IP address as obtained in Listing 5.5. We continue the simulation and soon the vehicle is reaching the edges of *ap2*'s range. The service replication is triggered when the vehicle enters the last quarter of the access point's range as it begins to move out of coverage. We can stop the simulation again, still in range of *ap2* and check our new deployment and flow rules. The

kubectl command for the deployments is, as the result, presented below.

```
$ sudo kubectl get deployments
```

NAME	READY	IMAGES	SELECTOR
mysimpleserver-deployment-1	1/1	mysimpleserver:latest	app=mysimpleserver
mysimpleserver-deployment-2	1/1	mysimpleserver:latest	app=mysimpleserver

Table 6: Deployments managing pod replicas in the Kubernetes cluster after first replication.

Another deployment was successfully created: *mysimpleserver-deployment-2*. More importantly we need to know what pods are currently running in our Kubernetes cluster. For that we use the following *kubectl* command.

```
$ sudo kubectl get pods
```

NAME	STATUS	IP	NODE
mysimpleserver-deployment-1-5b86f964b8-sntst	Running	10.244.1.6	poc-replication-worker
mysimpleserver-deployment-2-67fcd79d8-d49f2	Running	10.244.2.3	poc-replication-worker2

Table 7: Pods running in the Kubernetes cluster after first replication.

As expected, one more pod was created, and it's associated with the *poc-replication-worker2*. Now, let's see what flows were created. We made requests to all the **APs** and only the *ap3* with ID *115...979* had custom rules apart from the basic flow seen at Listing 5.6. We use *curl* to interact with Ryu's REST **API** and retrieve the flows for the *ap3*.

```
$ curl -X GET http://localhost:8080/stats/flow/1152921504606846979
```

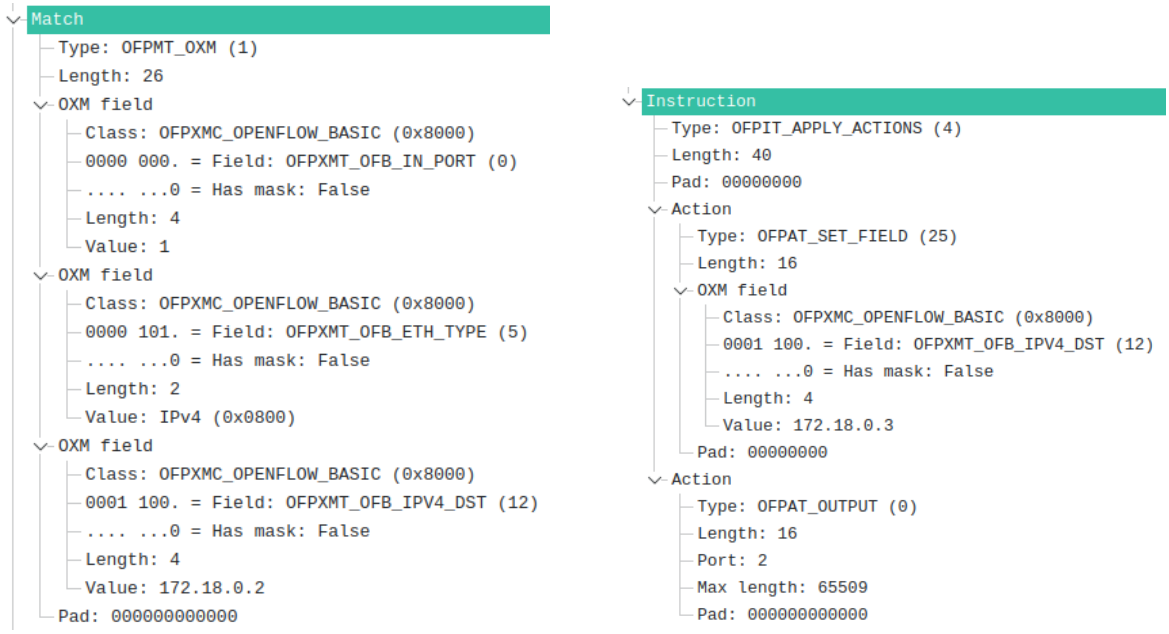
```
{
    "1152921504606846979": [
        {
            "priority": 10,
            "cookie": 0,
            "idle_timeout": 0,
            "hard_timeout": 0,
```

```
            "byte_count": 0,
            "duration_sec": 1048,
            "duration_nsec": 834000000,
            "packet_count": 0,
            "length": 120,
            "flags": 0,
            "actions": [
                "SET_FIELD: {ipv4_dst:172.18.0.3}",
                "OUTPUT:2"
            ],
            "match": {
                "in_port": 1,
                "dl_type": 2048,
                "nw_dst": "172.18.0.2"
            },
            "table_id": 0
        },
        {
            "priority": 10,
            "cookie": 0,
            "idle_timeout": 0,
            "hard_timeout": 0,
            "byte_count": 0,
            "duration_sec": 1048,
            "duration_nsec": 830000000,
            "packet_count": 0,
            "length": 120,
            "flags": 0,
            "actions": [
                "SET_FIELD: {ipv4_src:172.18.0.2}",
                "OUTPUT:FLOOD"
            ],
            "match": {
                "in_port": 2,
                "dl_type": 2048,
                "nw_src": "172.18.0.3"
            },
            "table_id": 0
        },
        ...
    ]
}
```

Listing 5.6: Response to the GET `/stats/flow/<dpid>` endpoint containing the list of flows of the given switch.

There are two new flows in *ap3*. We know already that the address 172.18.0.2 is our bootstrap IP, meaning that all requests are made to this address, and we inject flows to modify this address in the

packets to our preferred one. The most important changes in these two new flows are the priority, that increased faced to the base flow (Listing 5.4), the matches and the actions fields. Starting with the first flow, that matches IPv4 packets coming through port 1 (default wireless port in this framework), its job is to perform **Destination Network Address Translation (DNAT)**. This is when packets originally destined for 172.18.0.2 are intercepted and the destination IP is rewritten to 172.18.0.3, according to the actions field. The *actions* field also instructs the **AP** to forward the packet through port 2, redirecting it to the wired network, where it will ultimately reach the cluster node. Figure 14 shows two screenshots of Wireshark, one related to the *match* field and the other to the *actions*.

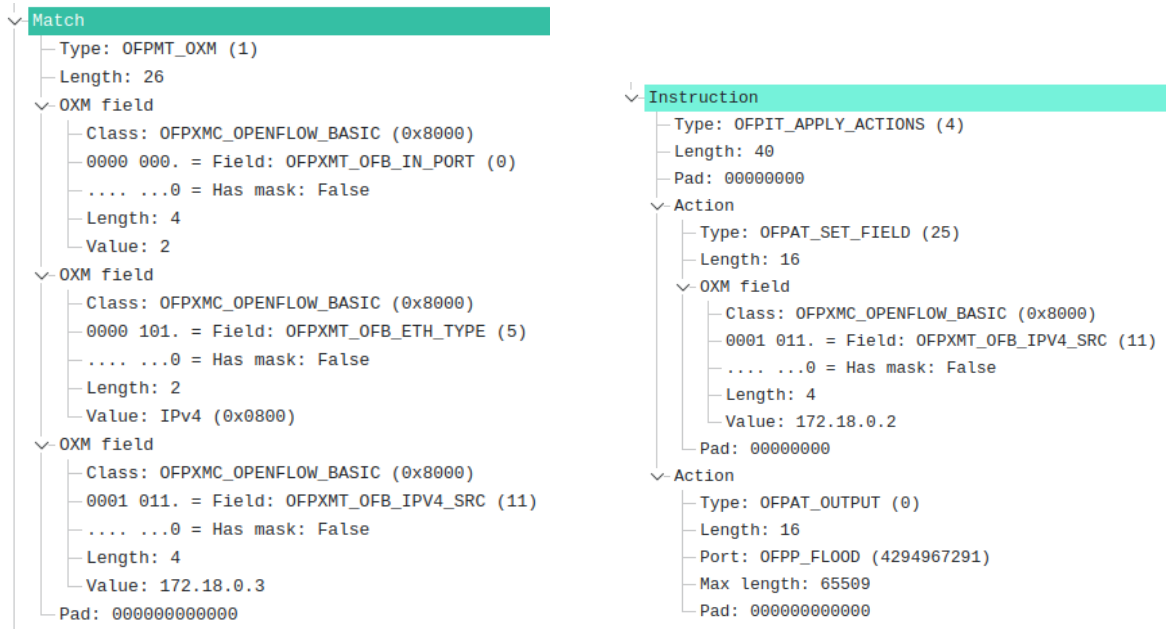


(a) Match field.

(b) Instructions field.

Figure 14: Wireshark screenshot of Openflow packet associated with **DNAT**.

The second flow handles the response from the server and performs **Source Network Address Translation (SNAT)**. In this case, packets returning from 172.18.0.3 are intercepted when they arrive at port 2, and their source IP is modified from 172.18.0.3 back to 172.18.0.2. This ensures that the communication appears to originate from 172.18.0.2 on both ends of the exchange. The packet is then flooded to both ports 1 and 3. The reason we are sending the packet to both ports is that if the vehicle is not in range of the **AP**, it will not receive the packet. Our goal is for the vehicle to receive the packet from any **AP** that is using the same worker node. This approach maintains consistency in the communication flow, making the source IP address transparent to the devices involved. Figure 15 shows two screenshots of Wireshark, one related to the *match* field and the other to the *actions*.



(a) Match field.

(b) Instructions field.

Figure 15: Wireshark screenshot of Openflow packet associated with **SNAT**.

Now let's keep the simulation going until the vehicle is in reach of *ap3* as seen in the Figure 16.

The client application is still running from the moment we started the simulation. There were a couple of moments where the vehicle lost connectivity, precisely when it is located in the middle of two **APs**. This happens because the **APs** are not close to each other and don't overlap much. Nevertheless, we can see the following output in the console.

```
{
    "PodIP": "10.244.2.3"
}
```

Listing 5.7: Response to the GET `/getPodIP` containing the IP of the pod that is hosting the service on the *poc-replication-worker* node.

This result demonstrates that the service was successfully replicated on another node, confirming that the network is functioning as intended. The **SDN** flows were able to dynamically modify the packet headers, specifically the source and destination IP addresses, ensuring communication between the client and the newly replicated service.

We could continue further with this scenario, and we would still trigger another deployment when we reach the edges of ap3's range. But for this scenario the goal was to develop a proof of concept that illustrates the capabilities of our **SDN** approach in dynamically managing service replication and traffic redirection. In this first section we laid the groundwork for future optimizations and applications, and most importantly we validate our current implementation. We will continue to enhance the framework and add

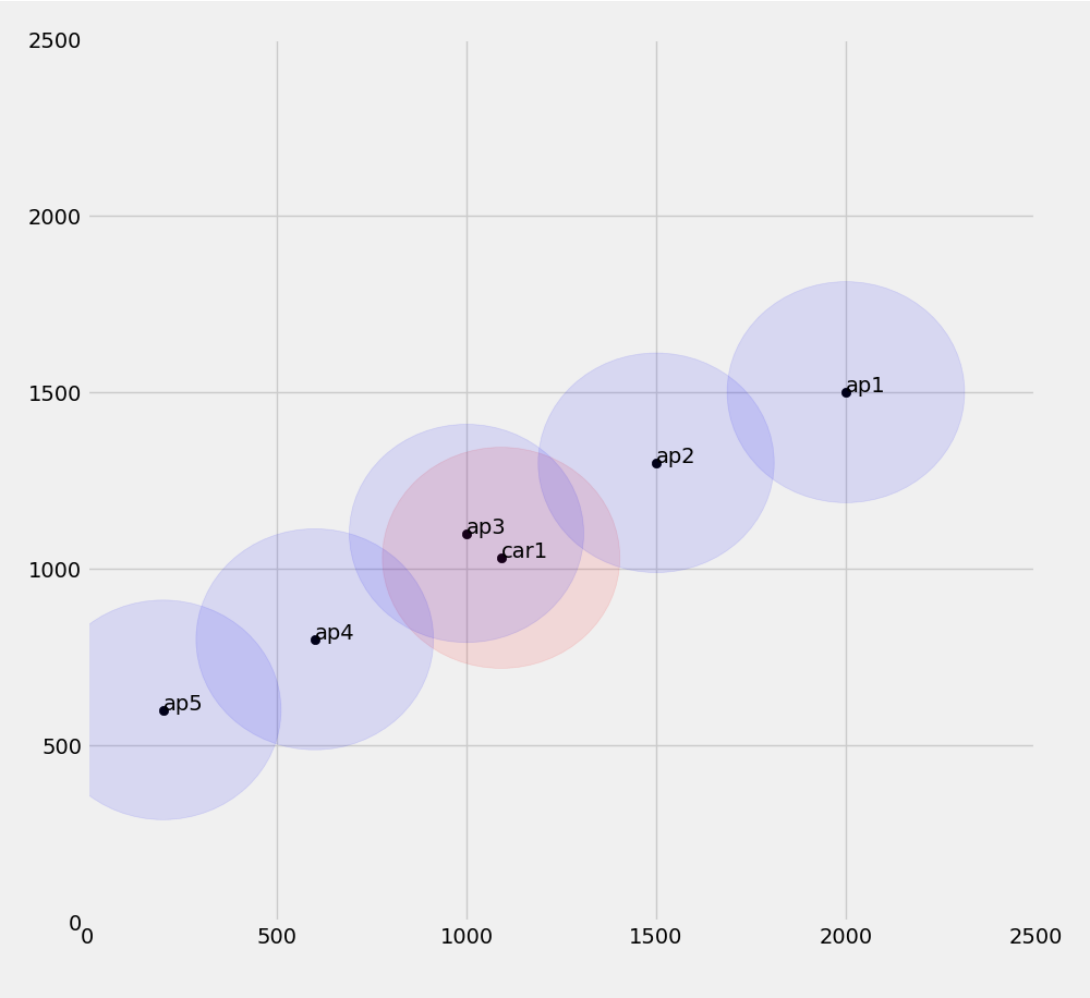


Figure 16: Visualization of vehicle position in range of *ap3*.

new functionality in the following sections.

5.3 Scenario II: Migration

In this experiment, the focus shifts from service replication to the full migration of a microservice to another node. This migration process involves transferring a running service from one worker node to another in real time, without disrupting its functionality. In this section, we describe how service migration can be performed within our framework, though the primary goal is to demonstrate the feasibility of such a process. In a real-world scenario, service migration would typically be driven by factors like load balancing (section 5.5), where services are moved to less busy nodes to optimize resource utilization and maintain performance. Another plausible use case is migrating a service to a node closer to end users (section 5.6), as proximity improves responsiveness, particularly in edge computing environments. Additionally, if there are costs associated with deployment, as is often the case, migration might be the most viable

option, since maintaining numerous replicas indefinitely may not be financially sustainable.

In our experiment, we are focusing on the migration of a stateless application, which simplifies the process significantly. Our *mysimpleserver* app, and the ones that we use in the following sections, don't retain data or state between sessions, meaning that in order to perform a migration, we can simply create a new deployment on another node and terminate the old one without worrying about losing information. However, this process isn't a *real* migration in the traditional sense—it's more about recreating the app in a different node and keeping only a single instance of the service.

To migrate a stateful application e.g., one that uses a database, is a much more complex task. If we are in a microservice architecture we could move our **API** to a different node, and it would still work as long as it has connectivity to the database. But to migrate data and actually reduce the latencies of any data queries, we would also need to migrate the underlying storage. Kubernetes offers some solutions for this such as **Persistent Volumess (PVs)** and **Persistent Volume Claimss (PVCs)**. A **PV** is a storage resource that exists independently of pods, and we could use **PVCs** to bind them to pods, allowing them to access the persistent data even if it moves or restarts. Ensuring data consistency and minimizing downtime is a difficult task that requires coordination and planning from the Engineers responsible for it. Also, for our use case, real-time migration of a stateful application might not be the best alternative since we have a very small window of time when the vehicles are in a specific location and also there is always some downtime during the detachment, reattachment, and pod restart processes.

For all the reasons stated above, we decided not to pursue this *real* migration process and instead decided to use a stateless application, that does not require a database. The migration process involves creating a new deployment in another node, and delete the previous one. In the Load Balancing scenario of section 5.5, we use the concept of deployment cost to reduce the amount of replicas we have instantiated. In this section we simply migrate services and use the topology from Figure 11 and all the same network configuration from the replication scenario at section 5.2.

The starting point for this scenario is very much similar to the previous replication experiment. We have created a new Kind cluster with three worker nodes, as seen in Table 8.

```
$ sudo kubectl get nodes
```

NAMES	STATUS	INTERNAL-IP
poc-migration-control-plane	Ready	172.18.0.4
poc-migration-worker	Ready	172.18.0.5
poc-migration-worker2	Ready	172.18.0.2
poc-migration-worker3	Ready	172.18.0.3

Table 8: Kubernetes nodes and their IP addresses, all in *Ready* state.

Similarly to the previous scenario, we have created a deployment that creates a single pod replica in the *poc-migration-worker* node. We can access more information about this deployment with the following command and respective result at Table 9.

```
$ sudo kubectl get deployments
```

NAME	READY	IMAGES	SELECTOR
mysimpleserver-deployment-1	1/1	mysimpleserver:latest	app=mysimpleserver

Table 9: Deployments managing pod replicas in the Kubernetes cluster.

The pod replica created with the previous deployment is presented at Table 10.

```
$ sudo kubectl get pods
```

NAME	STATUS	IP	NODE
mysimpleserver-deployment-1-f6778c8c4-wnmg2	Running	10.244.1.2	poc-migration-worker

Table 10: Pods running in the Kubernetes cluster.

Starting the simulation and having the vehicle at a location near *ap1*, like we see at Figure 12, we can gather about the same information as we did in the replication scenario. It is only when the vehicle moves past ap2 and into the range of *ap3*, somewhere in the middle of the positions at Figures 13 and 16, that the migration is triggered. The result of this migration is the deletion of the deployment named *mysimpleserver-deployment-1* and the creation of a new one that targets the *poc-migration-worker2* node. After the migration happens, these are the deployments being enforced on our cluster (Table 11).

NAME	READY	IMAGES	SELECTOR
mysimpleserver-deployment-2	1/1	mysimpleserver:latest	app=mysimpleserver

Table 11: Deployments managing pod replicas in the Kubernetes cluster, after migration.

Now let's take a look at the newly created pod replica presented at Figure 12.

NAME	STATUS	IP	NODE
mysimpleserver-deployment-2-6b4878858-t8fjt	Running	10.244.2.2	poc-migration-worker2

Table 12: Pods running in the Kubernetes cluster, after migration.

Regarding the **SDN** flows created, since the network configuration is the same and the **AP** that is responsible to redirect traffic to the *poc-migration-worker2* is also *ap3* with ID *1152921504606846979*, the resulting flows are the same as the ones presented in Listing 5.6.

In conclusion, the connectivity results and the Wireshark proofs are the same as in the replication scenario at section 5.2. This scenario's behavior is consistent with the previous one and there are no noticeable differences in terms of traffic redirection and packet handling. But, we noticed that when the *mysimpleserver-deployment-1* got deleted, and because the vehicle is still in range of *ap2*, the vehicle looses connectivity to the service. This is because the **APs** are not overlapping each other and the vehicle, in this topology, will always lose connectivity when in the middle of two **APs**. This issue is fixed in the next scenario topology, where we need full coverage of the vehicle route.

5.4 Scenario III: Video Streaming

For our third scenario we bring a different use case for the framework i.e., video streaming. The main goal of this experiment is to have a vehicle fetching a video while it moves under the range of different **APs**. This new service will be replicated in the necessary nodes, similarly to the previous two scenarios. Also, we will use **SDN** functionality to always redirect the traffic to the closest kubernetes node.

Following the comments of the previous sections, we updated the topology from Figure 11 to include more **APs**. This is meant to eradicate the areas where we had no **AP** in range, so that the streaming can run smoothly. We can see the new topology in Figure 17.

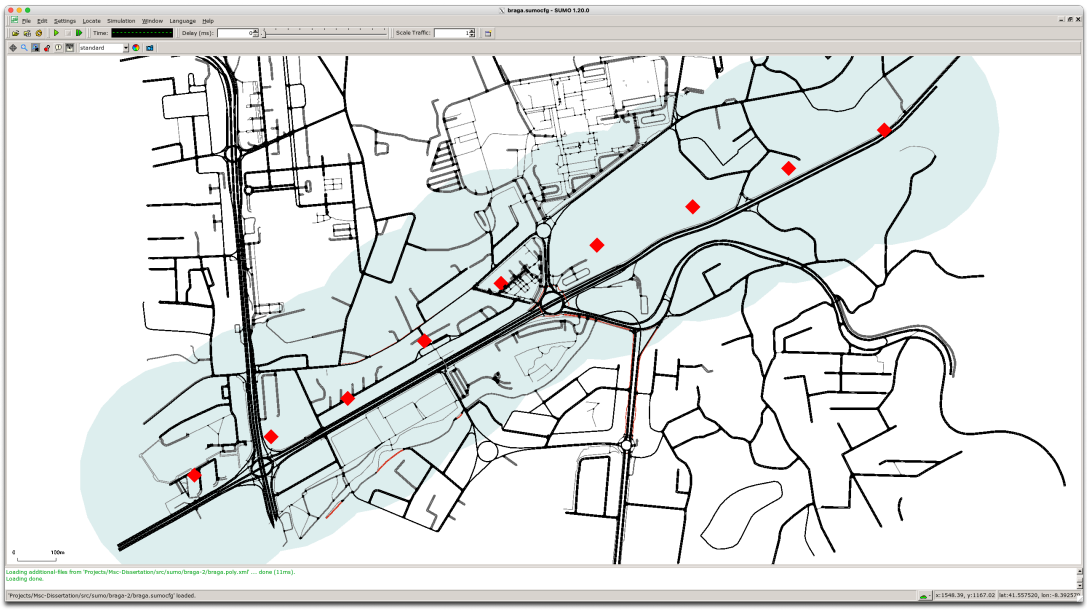


Figure 17: **SUMO** map of the city of Braga, updated with more **AP** coverage.

The map used for the topology is still the same, the only change was the number of **APs** that went from five to nine. The four additional **APs** were placed in between the previous ones. As a reminder, the **APs** are represented by the small red squares and in its surrounding area, in a grayish color, we see its range.

Our deployment trigger logic also remains the same and happens when the vehicle begins to move out of coverage, when it enters the last quarter of the **AP**'s range. There are no relevant changes in the flow rules that shape the traffic of the framework, they still look the same as in the previous scenarios (Figure 5.6).

This scenario introduces a new service that supports video streaming. Given the limitation about stateful applications that was discussed in the migration scenario (section 5.3), we opted for a stateless service where all the session information is stored on the client side. This service is a simplified version of **Dynamic Adaptive Streaming over HTTP (DASH)**, where video content is broken down into smaller chunks that the vehicle can request as it moves through different network zones. This stateless service stores video chunks on the server, and the vehicle fetches them as needed. The only issue with this approach is that the client has to track its previous requests, apart from that this method offers the flexibility and scalability we need for our vehicular environment. A very simple protocol diagram is presented in Figure 18.

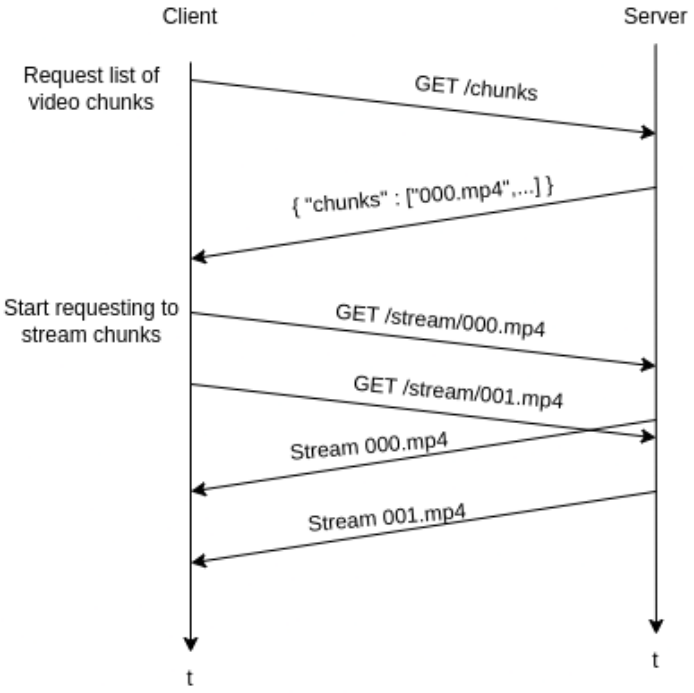


Figure 18: Streaming protocol diagram. Example of initial chunk list request and the following individual chunk requests.

The diagram that describes the streaming protocol illustrates its simplicity. The client starts by requesting a list of video chunks to the server and, after receiving it, starts requesting a stream of the individual video chunks, one by one.

Now let's start the simulation and, since the topology changed slightly, use Mininet-WiFi's integrated terminal to check the new network nodes and their connections. Below is the output of the *links* command.

```
mininet-wifi> links
ap1-eth2<->ap2-eth2 (OK OK)
ap2-eth3<->ap3-eth2 (OK OK)
ap3-eth3<->ap4-eth2 (OK OK)
ap4-eth3<->ap5-eth2 (OK OK)
ap5-eth3<->ap6-eth2 (OK OK)
ap6-eth3<->ap7-eth2 (OK OK)
ap7-eth3<->ap8-eth2 (OK OK)
ap8-eth3<->ap9-eth2 (OK OK)
car1-wlan0<->wifi (use iw/iwconfig to check connectivity)
```

```
nat0-eth0<->ap1-eth3 (OK OK)
```

Figure 19 presents the telemetry graphic of the current scenario.

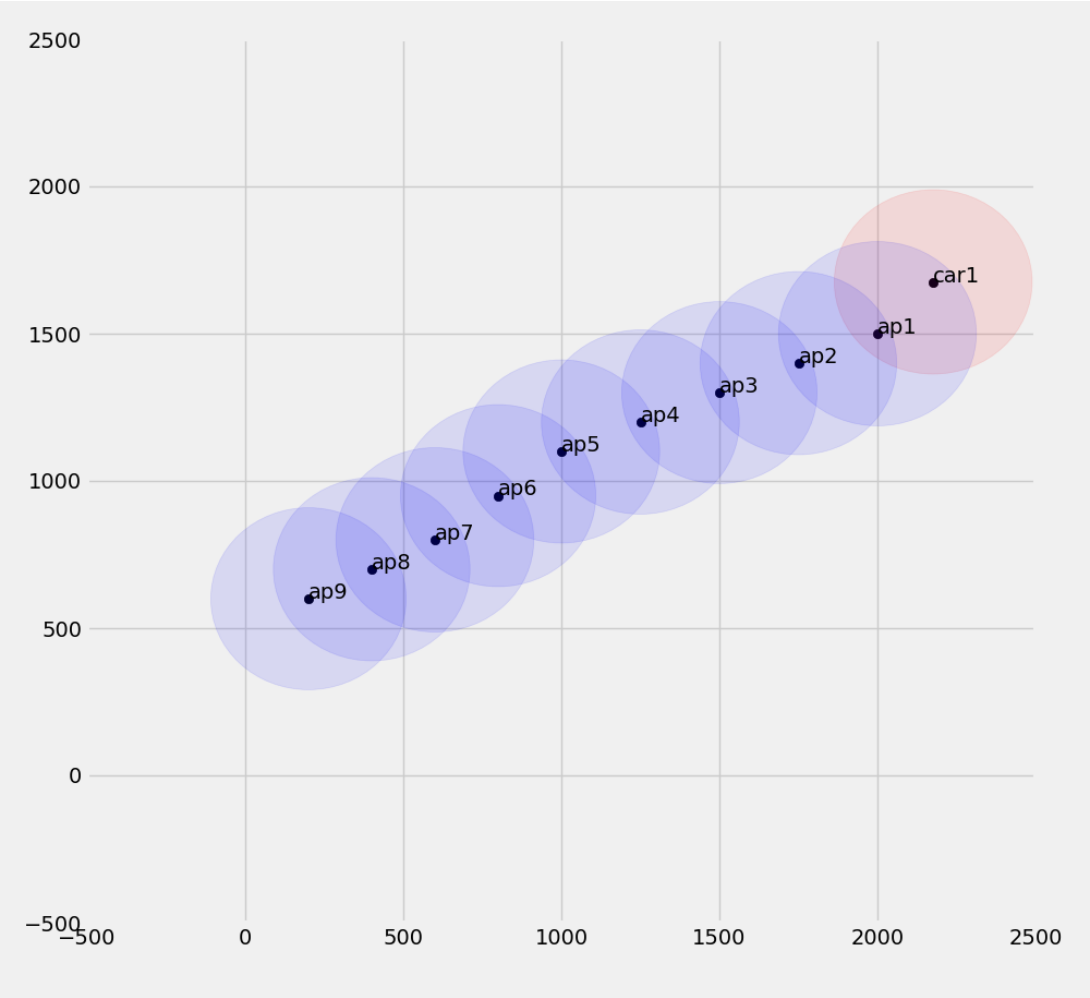


Figure 19: Initial view of the telemetry graphic for Video Streaming scenario.

Regarding the initial state of the simulation, as usual, we have a single deployment that instantiates a replica pod on one worker node. So, there are no changes apart from the topology and the service being used when compared to the replication experiment (section 5.2). Because of this, we are going to start by showing a screenshot of the sample video that we are using. This screenshot, presented at Figure 20, was taken from the Blue Planet II show from BBC Earth [48].

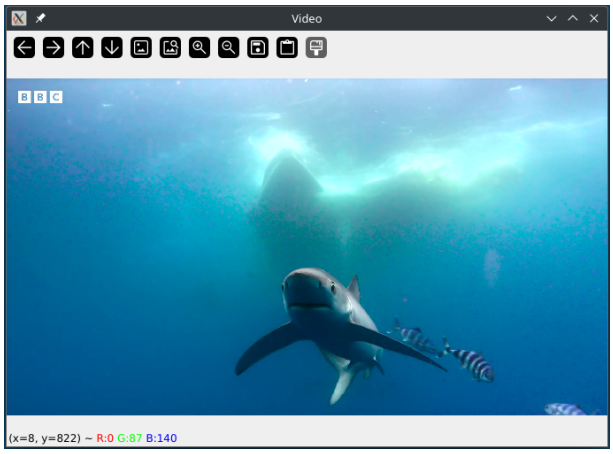


Figure 20: Screenshot of video sample being reproduced in the vehicle.

With the vehicle still in the same position as in Figure 19, we can start our client application with the following command.

```
mininet-wifi> xterm car1
car1# python services/client/streaming-client/client.py
```

The listing below shows the output of the first 30 seconds of streaming. The vehicle is stationary and located near *ap1*.

```
15:25:03.914 : Fetched output000.mp4 from "Pod 10.244.3.2"
15:25:05.092 : Fetched output001.mp4 from "Pod 10.244.3.2"
15:25:07.198 : Fetched output002.mp4 from "Pod 10.244.3.2"
15:25:08.107 : Fetched output003.mp4 from "Pod 10.244.3.2"
15:25:08.958 : Fetched output004.mp4 from "Pod 10.244.3.2"
15:25:09.691 : Fetched output005.mp4 from "Pod 10.244.3.2"
15:25:11.015 : Displayed: output000.mp4
15:25:11.628 : Fetched output006.mp4 from "Pod 10.244.3.2"
15:25:13.563 : Fetched output007.mp4 from "Pod 10.244.3.2"
15:25:15.884 : Fetched output008.mp4 from "Pod 10.244.3.2"
15:25:16.199 : Fetched output009.mp4 from "Pod 10.244.3.2"
15:25:17.151 : Fetched output010.mp4 from "Pod 10.244.3.2"
15:25:17.224 : Displayed: output001.mp4
15:25:18.479 : Fetched output011.mp4 from "Pod 10.244.3.2"
15:25:21.107 : Fetched output012.mp4 from "Pod 10.244.3.2"
15:25:24.651 : Fetched output013.mp4 from "Pod 10.244.3.2"
15:25:24.795 : Displayed: output002.mp4
15:25:26.235 : Fetched output014.mp4 from "Pod 10.244.3.2"
15:25:26.781 : Fetched output015.mp4 from "Pod 10.244.3.2"
15:25:28.733 : Fetched output016.mp4 from "Pod 10.244.3.2"
15:25:29.145 : Displayed: output003.mp4
15:25:31.834 : Fetched output017.mp4 from "Pod 10.244.3.2"
15:25:34.868 : Fetched output018.mp4 from "Pod 10.244.3.2"
```

```
15:25:35.707 : Fetched output019.mp4 from "Pod 10.244.3.2"
15:25:35.967 : Displayed: output004.mp4
```

Listing 5.8: Output of the streaming client application when the vehicle is stationary and in range of *ap1*.

From Listing 5.8 we assume that the streaming is working. Also, the video runs smoothly without interruptions. After about 30 seconds stationary we displayed four chunks and fetched nineteen. These buffered frames are important because it creates some margin for when the vehicle starts moving and eventually its connection is delayed.

Now let's stop our current stream and start over. This time we move the vehicle along the same road of the previous chapters. Here is the output when the vehicle changes **APs**.

```
15:54:57.217 : Fetched output000.mp4 from "Pod 10.244.3.2"
15:54:58.464 : Fetched output001.mp4 from "Pod 10.244.3.2"
15:55:00.620 : Fetched output002.mp4 from "Pod 10.244.3.2"
15:55:01.510 : Fetched output003.mp4 from "Pod 10.244.3.2"
15:55:02.354 : Fetched output004.mp4 from "Pod 10.244.3.2"
15:55:03.033 : Fetched output005.mp4 from "Pod 10.244.3.2"
15:55:04.279 : Displayed: output000.mp4
15:55:04.896 : Fetched output006.mp4 from "Pod 10.244.3.2"
15:55:07.018 : Fetched output007.mp4 from "Pod 10.244.3.2"
15:55:09.740 : Fetched output008.mp4 from "Pod 10.244.3.2"
15:55:10.185 : Fetched output009.mp4 from "Pod 10.244.3.2"
15:55:10.627 : Displayed: output001.mp4
15:55:11.532 : Fetched output010.mp4 from "Pod 10.244.3.2"
15:55:13.008 : Fetched output011.mp4 from "Pod 10.244.3.2"
15:55:16.043 : Fetched output012.mp4 from "Pod 10.244.3.2"
15:55:18.251 : Displayed: output002.mp4
15:55:21.528 : Fetched output013.mp4 from "Pod 10.244.3.2"
15:55:22.547 : Displayed: output003.mp4
15:55:25.778 : Fetched output014.mp4 from "Pod 10.244.3.2"
15:55:26.916 : Fetched output015.mp4 from "Pod 10.244.3.2"
15:55:30.845 : Fetched output016.mp4 from "Pod 10.244.3.2"
```

Listing 5.9: Output of the streaming client application when the vehicle starts moving and is located in range of *ap2*.

We can easily detect the moment when the vehicle came into range of *ap2*. This is indicated by the fact that, within a timespan of about 10 seconds, the client fetched only one chunk (output013.mp4). It is also becoming increasingly evident that the speed at which the chunks are fetched slows down when the vehicle is in motion.

To proceed with our experiment, let's continue the simulation and move the vehicle past ap3, triggering a deployment, and let's observe and analyze the performance of our streaming application. The following listing displays the complete logs of the streaming application from the first vehicle position near *ap1* to the *ap9*.

```
18:03:53.703 : Fetched output000.mp4 from "Pod 10.244.1.6"
18:03:54.995 : Fetched output001.mp4 from "Pod 10.244.1.6"
18:03:57.285 : Fetched output002.mp4 from "Pod 10.244.1.6"
18:03:58.151 : Fetched output003.mp4 from "Pod 10.244.1.6"
18:03:59.049 : Fetched output004.mp4 from "Pod 10.244.1.6"
18:03:59.908 : Fetched output005.mp4 from "Pod 10.244.1.6"
18:04:00.939 : Displayed: output000.mp4
18:04:01.989 : Fetched output006.mp4 from "Pod 10.244.1.6"
18:04:04.128 : Fetched output007.mp4 from "Pod 10.244.1.6"
18:04:07.279 : Displayed: output001.mp4
18:04:08.201 : Fetched output008.mp4 from "Pod 10.244.1.6"
18:04:08.722 : Fetched output009.mp4 from "Pod 10.244.1.6"
18:04:14.735 : Displayed: output002.mp4
18:04:18.950 : Displayed: output003.mp4
18:04:19.772 : Fetched output010.mp4 from "Pod 10.244.1.6"
18:04:22.928 : Fetched output011.mp4 from "Pod 10.244.1.6"
18:04:25.703 : Displayed: output004.mp4
18:04:26.841 : Fetched output012.mp4 from "Pod 10.244.1.6"
18:04:29.879 : Displayed: output005.mp4
18:04:37.987 : Error fetching output013.mp4: Read timed out.
18:04:39.946 : Displayed: output006.mp4
18:04:44.593 : Fetched output013.mp4 from "Pod 10.244.2.5"
18:04:44.887 : Displayed: output007.mp4
18:04:47.266 : Fetched output014.mp4 from "Pod 10.244.2.5"
18:04:48.223 : Fetched output015.mp4 from "Pod 10.244.2.5"
18:04:49.922 : Displayed: output008.mp4
18:04:51.388 : Fetched output016.mp4 from "Pod 10.244.2.5"
18:04:51.863 : Displayed: output009.mp4
18:04:56.362 : Fetched output017.mp4 from "Pod 10.244.2.5"
18:04:58.851 : Displayed: output010.mp4
18:05:04.894 : Displayed: output011.mp4
18:05:10.637 : Error fetching output018.mp4: Read timed out.
18:05:10.759 : Displayed: output012.mp4
18:05:17.935 : Fetched output018.mp4 from "Pod 10.244.2.5"
18:05:18.867 : Displayed: output013.mp4
18:05:19.893 : Fetched output019.mp4 from "Pod 10.244.2.5"
18:05:22.263 : Displayed: output014.mp4
18:05:24.971 : Fetched output020.mp4 from "Pod 10.244.2.5"
18:05:26.478 : Displayed: output015.mp4
18:05:30.626 : Error fetching output021.mp4: Read timed out.
18:05:32.041 : Displayed: output016.mp4
18:05:39.006 : Displayed: output017.mp4
18:05:45.767 : Displayed: output018.mp4
18:05:47.829 : Fetched output021.mp4 from "Pod 10.244.3.3"
18:05:49.073 : Displayed: output019.mp4
18:05:55.016 : Fetched output022.mp4 from "Pod 10.244.3.3"
18:05:56.576 : Fetched output023.mp4 from "Pod 10.244.3.3"
18:05:58.222 : Displayed: output020.mp4
```

```
49 18:06:03.549 : Fetched output024.mp4 from "Pod 10.244.3.3"
50 18:06:06.159 : Displayed: output021.mp4
51 18:06:12.550 : Fetched output025.mp4 from "Pod 10.244.3.3"
52 18:06:12.550 : Uptime: 133.11275908991047
53 18:06:12.550 : Downtime: 0
```

Listing 5.10: Output of the streaming client application capturing the entire vehicle movement from *ap1* to *ap9*.

For the first 36 seconds, until line 20 of Listing 5.10, the vehicle moves across the first three **APs**, buffering a total of 12 chunks. On line 20, we see the first timeout error, that indicates that the client application did not obtain a response from the server in time. The request is repeated very shortly and a response is obtained containing the 13th chunk. This chunk is obtained from another pod with IP 10.244.2.5, belonging to the *streaming-service-worker2* node. The vehicle continues its movement, and approximately 1 minute later reaches the edge of the **APs** associated with the second worker node. But before this happens, a new deployment, the third, is created. The connection throws another timeout error (line 40), and shortly after the same chunk is fetched from the third worker node with pod IP 10.244.3.3. In the middle of this whole process, there is another apparently unexpected timeout error (line 32). This timeout is due to an abrupt change of **AP**, and it ends up delaying the whole process for a few seconds. We can also see that after the third timeout error, the fetch for the 21st chunk is delayed for almost seventeen seconds. The reasons for this are unknown.

Regarding the deployments used in this scenario, there were three, each one of them creating a single pod replica. Below we can see the pods created for this scenario.

NAME	STATUS	IP	NODE
streamingserver-deployment-1-5fcc8d6cd9-q7n42	Running	10.244.1.6	streaming-service-worker
streamingserver-deployment-2-65fb9f4555-d2pwt	Running	10.244.2.5	streaming-service-worker2
streamingserver-deployment-3-54b5b579dd-dg64n	Running	10.244.3.3	streaming-service-worker3

Table 13: Pods running in the Kubernetes cluster for scenario III Video Streaming.

The performance of the streaming service, while functional in certain scenarios, is not yet optimal, particularly in environments with high mobility. In urban settings, where vehicles frequently stop at traffic lights or intersections, the service performs adequately, as these pauses provide windows of time for the system to reconnect and fetch video chunks. However, when the vehicle is in continuous motion, particularly at higher speeds or in areas with sparse access points, the frequent *handoffs* between access points cause significant delays. The client often fails to retrieve video chunks in time due to the high

reconnection time associated with these handovers. This results in interruptions in the video stream and a degraded user experience.

One possible solution for this is to implement a more sophisticated client architecture. Enabling asynchronous chunk requests will allow the client to fetch multiple video segments in parallel or from different access points simultaneously. Additionally, we could have a predictive algorithm to anticipate an upcoming connectivity issue and have the client adjust its chunk fetching behavior proactively. For example, if the vehicle is approaching an area of low connectivity, it could request more chunks to the server to keep the buffer at maximum capacity. These enhancements and others that we haven't focused on, have the potential to improve the video streaming performance, especially in highly dynamic vehicular environments.

In conclusion, the implementation of a video streaming use case within the framework highlights both the potential and challenges of supporting multimedia services in dynamic vehicular environments. The streaming service serves as a real world use case that demonstrates how the framework handles a more data intensive application, while facing an intermittent connectivity and frequent handovers. By addressing these challenges, we have showed that the framework is able to support a multimedia streaming application. Even if we go beyond this use case, the framework should support various vehicular applications that require reliable communication and data exchange, such as an autonomous driving coordination system or a real-time traffic monitoring system, to name a few outside the realm of infotainment services. All of these applications rely heavily on on-time and efficient data transmission, similar to the requirements of streaming services. Therefore, the development of this framework invites future work to deploy and test various vehicular applications, further exploring its performance, scalability and potential in real-world scenarios.

5.5 Scenario IV: Load Balancing

In this fourth scenario, we introduce yet another use case for our framework: network load balancing. Network load balancing is a traffic distribution process that optimizes resource usage and ensures an even distribution of network traffic across multiple servers (or nodes). In our vehicular network, load balancing can be used in highly congested environments, like urban areas. This service can be used not only to reduce latency but also to increase availability, ensuring that all applications such as the ones used in previous scenarios, can perform reliably under varying conditions. This section will explore how our framework supports load balancing and its importance in achieving a stable and scalable vehicular

network.

We can first start with a new topology that will be used on this last two scenarios. The most densely populated borough of New York City, Manhattan, with its grid-like roads, is the perfect topology for testing a load balancing solution. We can take a look at the simulation map with the screenshot from **SUMO** below (Figure 21).

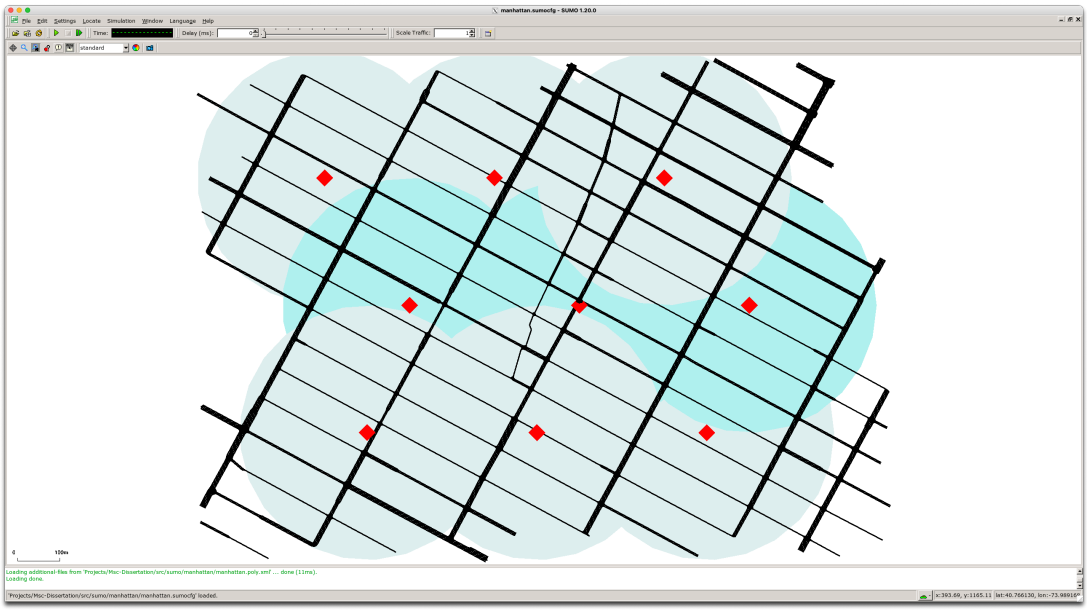


Figure 21: **SUMO** map of the borough of Manhattan.

We have nine **APs**, represented by the red squares, and, in contrast with past scenarios, we will have more than one vehicle. We ran tests using a wide range of vehicles, starting with a single one to a few dozens. Their routes were randomly generated using the *randomTrips.py* utility from **SUMO**.

Overall, as we can see in Figure 22, this scenario uses a simple architecture. After the user runs the Python command that starts the experiment, a thread is created to track the position of all the vehicles present in the simulation. Let's call this thread the *position-tracker* thread. Before the main thread starts the Mininet-WiFi software, another thread is created, the *controller* thread.

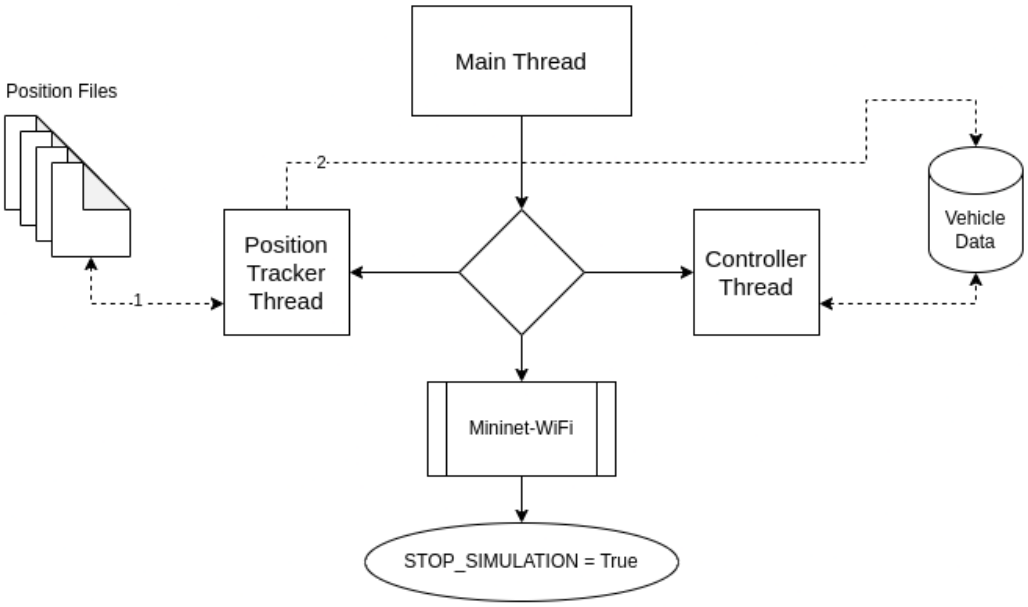


Figure 22: Architecture of the Load Balancing scenario program.

The goal of the *position-tracker* thread is to, as the name suggests, track location related information about the vehicles. This thread will update a shared structure with the parameters about the vehicles in the simulation seen in Listing 5.11.

```
{
    "car1": {
        "position": (x, y),
        "direction": (dx, dy),
        "associated_ap": "ap1",
        "using_node": "worker2",
        "flows" : [
            {
                'ap': "ap1",
                'node': "worker2",
            },
            ...
        ]
    },
    ...
}
```

Listing 5.11: Vehicle data structure collected and maintained by the *position-tracker* threads.

With the information presented at Listing 5.11, the *controller* thread will have access to each vehicle's position, direction, currently associated **AP**, node and the list of **SDN** flows installed at each **AP** (with the associated node). This information only makes sense because the **SDN** flows have also changed for this specific scenario. We still have to install two flows, one to change the source IP (**SNAT**) and the other to

change the destination IP (**DNAT**). But now, instead of matching with the bootstrap IP (destination) that is present in the packets entering through the wireless port (1), we match with the vehicle IP (source) directly. As expected, when the packet returns, we match it by the vehicle IP (destination). This change in the **SDN** flows allows us to create rules for specific vehicles instead of matching traffic from multiple vehicles that use the same **AP** and bootstrap IP. This change was needed for this scenario because we have to be able to send a specific vehicle traffic to another worker node, other than the one closest to the **AP** in range. When doing this we would be balancing the load of our worker nodes, but this will also result in higher latencies, because the node is not the closest from the **AP**. Later, when we start the experiment, we will see examples of the **SDN** flows used.

About the controller thread, its job is to make decisions using the data collected by all the *vehicle* threads. The process used to make decisions can be summarized with some pseudocode, as seen in the Algorithm 1 below. In the strategy described, we need an initial set of data i.e., Listing 5.11, and the goal is to keep the load of requests balanced through all the nodes, while the simulation is running. At each iteration of the algorithm, we start by collecting the number of vehicles that is using each node. This gives us an initial idea of which nodes have the most and the least number of connected vehicles. Our strategy is to move vehicles from high load nodes to low load nodes. To achieve this we calculate something that we call the *retention rate* of each vehicle that is connected to the node with the highest load. This value is the distance that this vehicle will still travel in range of the node (which includes several **APs**). We calculate this assuming the current movement direction, which might not be a correct calculation because it does not necessarily follow the road. The error associated with this assumption is mitigated when using a grid topology. After calculating this distance for all the vehicles, we sort it and determine the number of vehicles to move to the *lightest* node. The calculation aims to balance the *heaviest* node with the *lightest*. The vehicles that qualify for this operation are selected and two **SDN** flows (**SNAT** and **DNAT**) are installed in the respective **AP**.

Algorithm 1: Load Balancing Algorithm used for scenario IV.

Data: Vehicle positions, directions, associated nodes and flows.

Result: Balanced load across nodes.

while *simulation is running* **do**

  Collect current load of each node: `nodes_load`;

  Identify the node with the heaviest load: `heaviest_node`;

  Identify the node with the lightest load: `lightest_node`;

  if *both nodes are different* **then**

    Calculate retention distance for each vehicle in the heaviest node;

    Sort vehicles by retention distance;

    Determine number of cars to move:

$$\mathrm{N} = \left\lfloor \frac{\texttt{len(cars_in_heaviest)} + \texttt{len(cars_in_lightest)}}{2} - 1 \right\rfloor$$

    if $N > 0$ **then**

      Select N vehicles with the shortest retention time;

      Install flows to move selected vehicles to the lightest node;

    end

  end

end

In summary, the vehicles that are predicted to stay within the **AP** range for the longest time are prioritized for reassignment when the nodes become unbalanced. After selecting the vehicles to move, the algorithm installs new **SDN** rules to migrate their traffic to the lighter-loaded node, thereby optimizing resource usage.

Now let's start the simulation and briefly go over the initial configuration of this experiment. In Figure 23 we can observe the telemetry screen captured just a few seconds after starting the vehicle mobility.

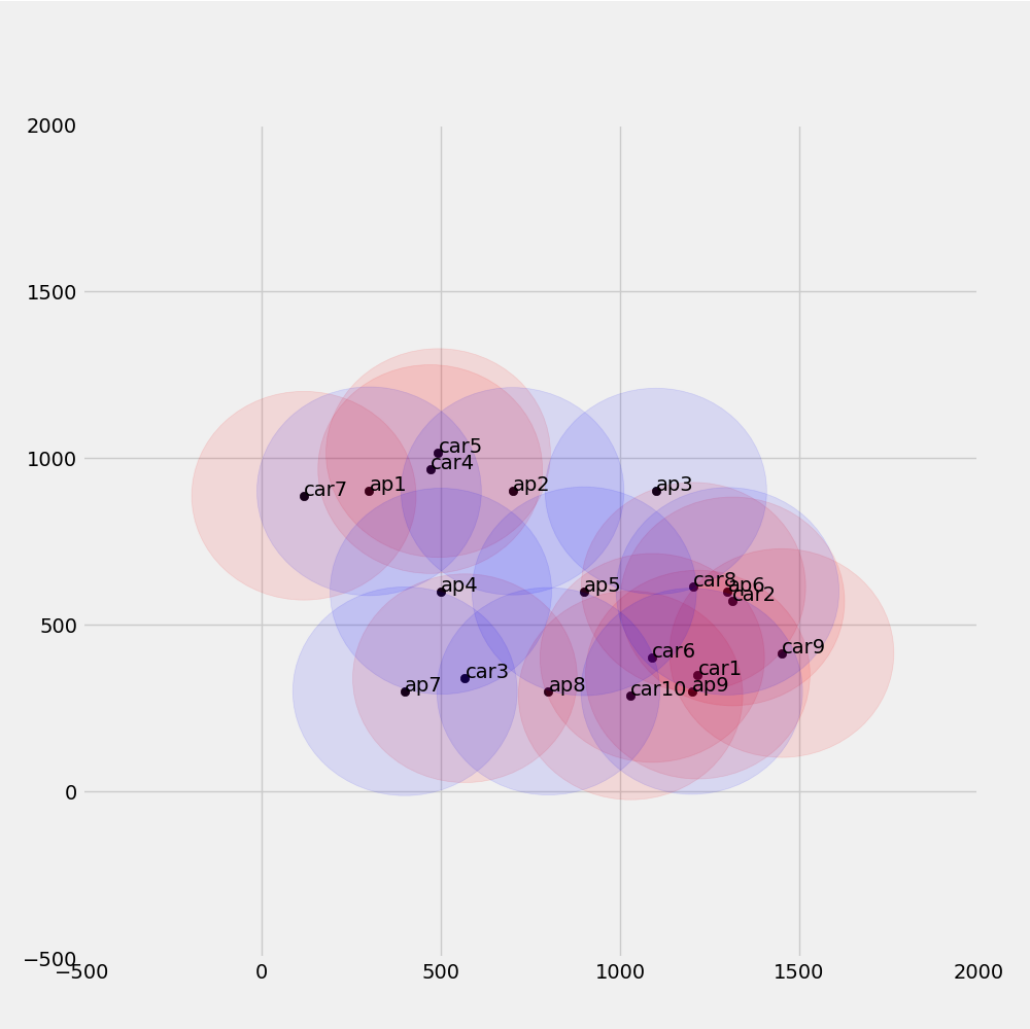


Figure 23: View of the initial telemetry screen for the load balancing scenario.

For this experiment we have started with ten vehicles, in order to keep the telemetry screen clear enough to debug any bugs related to the load balancing algorithm. As for the rest of the configurations, they have also changed quite a bit. Before jumping to the deployments and **SDN** flows, it is worth mentioning that we are using, as seen in Table 14, four worker nodes.

NAMES	STATUS	INTERNAL-IP
load-balancing-control-plane	Ready	172.18.0.4
load-balancing-worker	Ready	172.18.0.6
load-balancing-worker2	Ready	172.18.0.5
load-balancing-worker3	Ready	172.18.0.2
load-balancing-worker4	Ready	172.18.0.3

Table 14: Kubernetes nodes used in the Load Balancing scenario, all in *Ready* state.

Each of the four worker nodes is associated to its *closest* **AP**. Below, on Figure 24, we can see a representation of the **APs** used and their configuration.

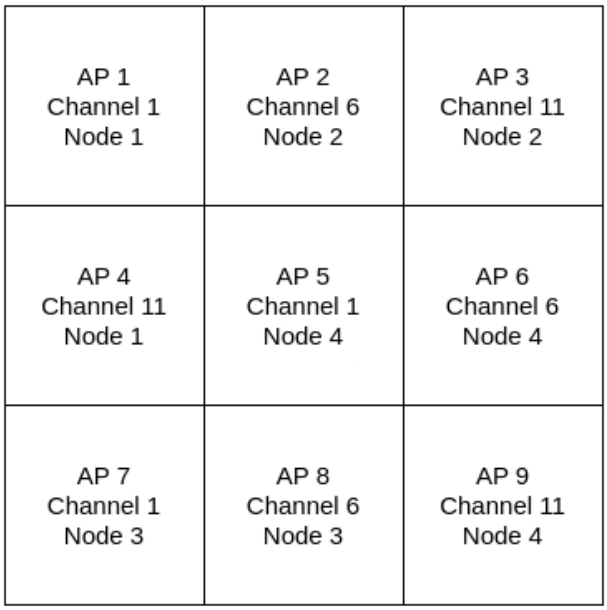


Figure 24: Configuration of the nine **APs** used in the Load Balancing scenario.

This mapping between **APs** and worker nodes is useful to create all the initial **SDN** flows, and to have a reference of the optimal routing for each **AP**. Now, moving to the deployments, we are using the *mysimpleserver* service, as there is no need to overcomplicate the application because we are testing a load balancing algorithm, and we are treating every vehicle as if they were using the same amount of resources. Our decisions assume that all the vehicles in range are interested in the service and therefore their traffic has to be balanced. This leads us to the defining change of this scenario, in relation to the previous ones. The service is now fully deployed on all worker nodes prior to the start of the experiment. This means that

we will have exactly one pod replica per worker node, as we decided that it is not relevant for the tests to increase the number of replicas. Below, on Figure 15, we can see the four initial deployments.

NAME	READY	IMAGES	SELECTOR
mysimpleserver-deployment-1	1/1	mysimpleserver:latest	app=mysimpleserver
mysimpleserver-deployment-2	1/1	mysimpleserver:latest	app=mysimpleserver
mysimpleserver-deployment-3	1/1	mysimpleserver:latest	app=mysimpleserver
mysimpleserver-deployment-4	1/1	mysimpleserver:latest	app=mysimpleserver

Table 15: Initial deployments for the Load Balancing scenario.

Given that we start the experiment with four worker nodes, we also have to make sure that each **AP** has at least three initial flows. The first one we have already seen in Listing 5.4, and its job is to redirect any packet that does not match any rules to the controller. The second and third one (**SNAT** and **DNAT**) are meant to create the flows that guarantee that packets are redirected to the correct worker node, accordingly with Figure 24. In terms of priorities, the default rule that redirects packets to the controller will have the lowest priority (0), then the two flows that are meant to redirect packets without caring for the source vehicle will have a higher priority (10). Any added flows meant to redirect traffic based on the source vehicle will have the highest priority (100) so that we can make sure that the load balancing happens. In the listing below we can see an example of the initial **SDN** flows for the *ap8*, which as Figure 24 suggests, is associated with the third worker node. As we see on Table 14, the third worker node has the IP address of 172.18.0.2. The bootstrap IP, as discussed previously, is the IP of the first worker node, 172.18.0.6.

```
{
    "1152921504606846984": [
        {
            "priority": 10,
            "cookie": 0,
            "idle_timeout": 0,
            "hard_timeout": 0,
            "byte_count": 0,
            "duration_sec": 1048,
            "duration_nsec": 834000000,
            "packet_count": 0,
            "length": 120,
            "flags": 0,
            "actions": [
                "SET_FIELD: {ipv4_dst:172.18.0.2}",
                "OUTPUT:2"
```

```
            ],
            "match": {
                "in_port": 1,
                "dl_type": 2048,
                "nw_dst": "172.18.0.6"
            },
            "table_id": 0
        },
        {
            "priority": 10,
            "cookie": 0,
            "idle_timeout": 0,
            "hard_timeout": 0,
            "byte_count": 0,
            "duration_sec": 1048,
            "duration_nsec": 830000000,
            "packet_count": 0,
            "length": 120,
            "flags": 0,
            "actions": [
                "SET_FIELD: {ipv4_src:172.18.0.6}",
                "OUTPUT:FLOOD"
            ],
            "match": {
                "in_port": 2,
                "dl_type": 2048,
                "nw_src": "172.18.0.2"
            },
            "table_id": 0
        },
        ...
    ]
}
```

Listing 5.12: Initial **SDN** flows for *ap1*. The basic flow, that redirects traffic to the controller, is not in this listing, but it is also a part of the flow rules of *ap1*.

For the **APs** that use the bootstrap node, the flow rules are not necessary. Since the vehicle already makes the requests to the optimal node (bootstrap node is the optimal node for some **APs** i.e., *ap1*, *ap4* and *ap5*), the flows are not needed.

Now that we have defined the initial configuration of this experiment, we can start the simulation. As we have mentioned, the vehicles move randomly through the simulation space and our goal is to balance the load of all the worker nodes. But, to be able to display the load balancing information and make any conclusions on the performance of the algorithm, we need to implement a visualization system. In this scenario, we gather, at every iteration of the algorithm, the load of each node. With this information we

can plot our first graph that displays the load of each worker node in a simulation with 10 vehicles.

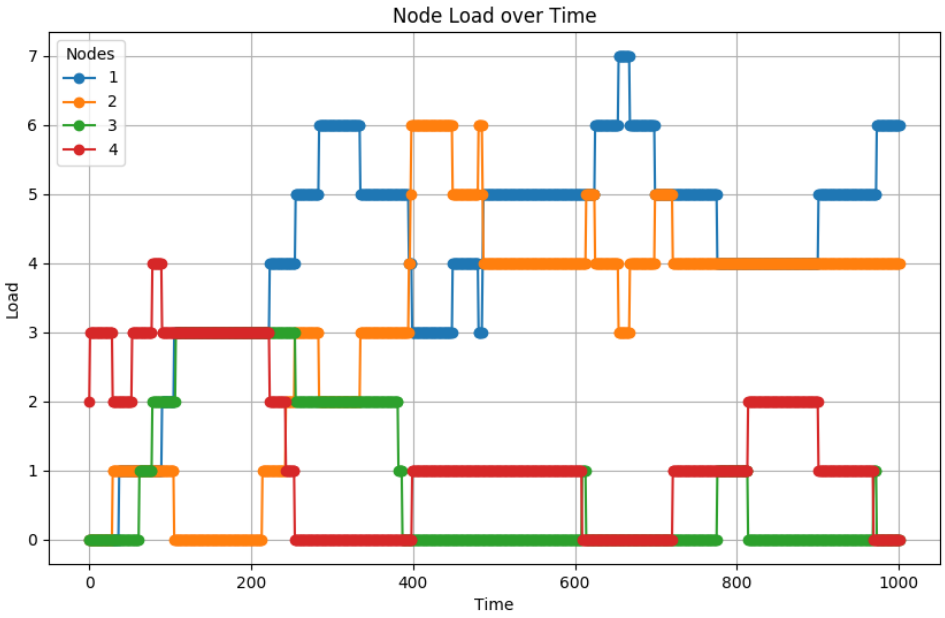


Figure 25: Load of each node during the simulation with random routes and ten vehicles.

In the plot displayed at Figure 25, the Y-axis shows the evolution of the load on each worker node i.e., the number of associated vehicles. It is obvious that there is not an overall balance between all the nodes. We can observe that node 1 and 2 experience the highest load levels, frequently oscillating between 4 and 7, while node 3 and 4 have relatively lower loads, often remaining at values between 0 and 3. The constant movement of vehicles, and the consequent change between different nodes, might be the reason of this apparent lack of balance. But these are the results of one simulation, with one set of routes and also a pre-defined relation between **APs** and nodes, but mainly, only ten vehicles. By increasing the number of vehicles to, for example, 100, we should be able to realize the real effectiveness of the load balancing algorithm. In the Figure below we can see just that, a simulation with 100 vehicles.

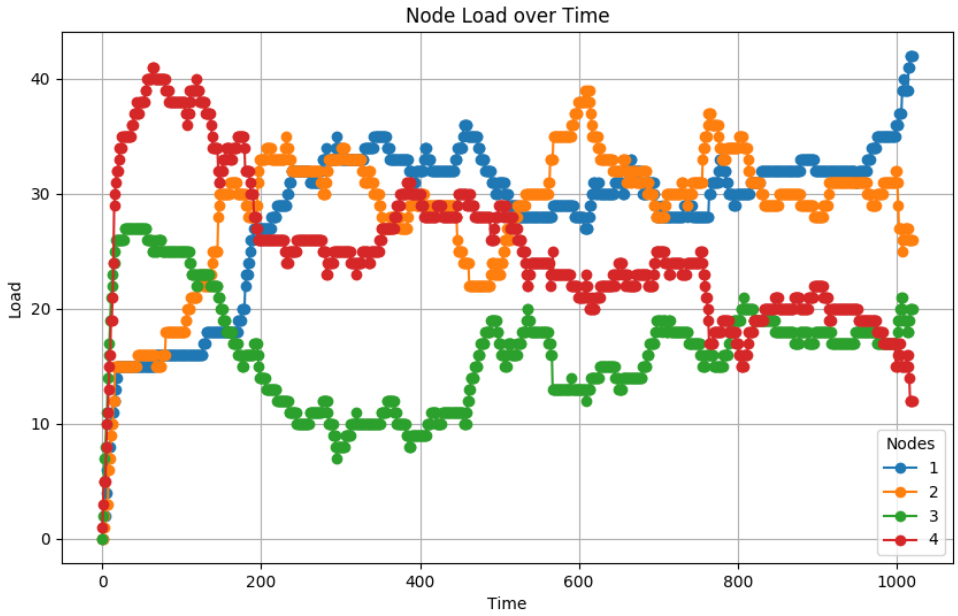
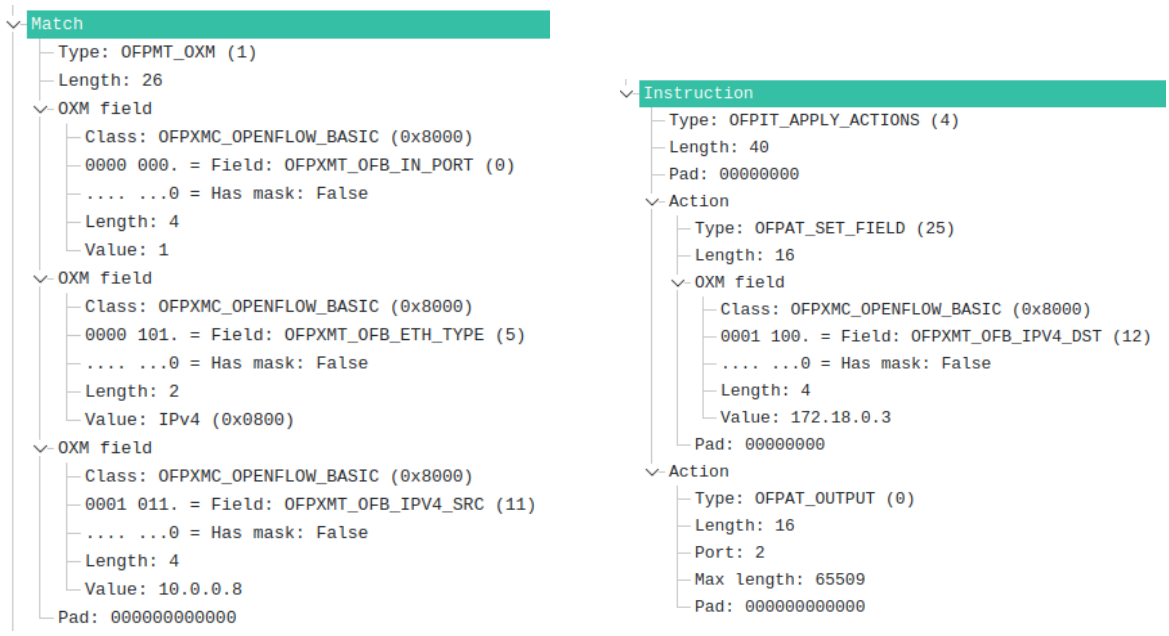


Figure 26: Load of each node during the simulation with random routes and one hundred vehicles.

There is a similar trend that is noticeable in the plot of Figure 26, the first two nodes (1 and 2) tend to accumulate more vehicles (Y-axis). We have made a lot of simulations, and it seems that the most important parameters for a good balanced result is the route that the vehicles take and the configuration that defines the *ideal* nodes for each **AP**. Even when generating the routes randomly, we did not always get a well distributed set of 100 routes. Meaning that we had to manually adjust the routes so that the traffic is more or less homogeneous i.e., not having all the vehicles travelling the in the same block. With a good traffic distribution and the configuration seen at Figure 24 we are able to achieve the results shown above.

Now that we have seen that the load balancing algorithm works adequately, we can take a look at the new **SDN** flows that we are using specifically in this scenario. Apart from the flows presented in Listing 5.12, we now have flows that match a specific vehicle and redirect its traffic to the desired node.



(a) Match field.

(b) Instructions field.

Figure 27: Wireshark screenshot of Openflow packet associated with **DNAT** for specific vehicle with IP 10.0.0.8 in Load Balancing scenario.

In Figures 27a and 27b we see that the traffic generated from vehicle with IP 10.0.0.8 and that is coming from port 1 (wireless port) of this specific **AP**, will all be redirected to the node with IP 172.18.0.3, i.e., worker 4 according to Table 14. These flows are key to balance the load of each node with a high level of precision, since we are handling each vehicle individually.

In conclusion, Algorithm 1, along with the architecture design illustrated in Figure 22, demonstrates another practical use case for the framework we are presenting. We have provided a detailed explanation of how the experiment operates, from the **SDN** flows to the simulation settings, as well as the results achieved. While many more simulations with varying parameters can be made using this framework, our primary focus is not on benchmarking. Although we used one hundred vehicles, which pushed our resources to the limit, the key objective is to develop different and valuable applications on top of this framework. This scenario is the fourth of five, and next, we will present the final scenario, which addresses the mobility of microservices in a similar environment.

5.6 Scenario V: Mobility of Microservices

The final scenario presented in this document, describes the use case that names this dissertation, i.e., the mobility of microservices. This use case was the main driver for this dissertation, but before implementing

it, we had to define our framework's architecture and test it using at least the first two simpler scenarios. Also, due to its complexity, this experiment took longer to implement and test thoroughly, compared to the previous ones.

The goal of this experiment is to create a simulation, using the Manhattan topology (Figure 21) of the previous scenario, where vehicles are consuming a service, and we must guarantee that, despite the movement of the vehicle, it will have the lowest possible latency. We have seen that in our framework we might not be able to measure latency accurately, but we know the *ideal* **K8s** node for each vehicle, at every instant. To guarantee the lowest latency we must migrate the microservice closer to the vehicle. This is possible, similarly to the first three scenarios (but in a bigger scale and complexity now), by a combination of creating **SDN** flows and **K8s** deployments that target a specific **AP** and node, respectively.

There are many ways in which we can migrate microservices. In this experiment we have two strategies that influence how the microservices are migrated. We have a reactive strategy, that triggers a microservice deployment and respective **SDN** flow injection, after the vehicle is in a non-optimal position in terms of latency and/or proximity to the node. Our other strategy is predictive. It aims to predict where the vehicle will be located in the next few instants, with enough time to trigger a deployment and flow injection. At the end of the experiment we compare the two strategies given two very important metrics: the latency, measured as proximity to the optimal node, and the cost, measured as the number of deployments we make. The introduction of the cost variable brings a new dynamic to this simulation and adds another layer of realism, since there is no such thing as *free resources*.

Starting with the description of the simulation environment, the topology used is the same as in the last Section 5.5. Manhattan's grid-like roads, with nine **APs** and nine **K8s** worker nodes. In this experiment, because our goal is to manage the deployments per worker node, we decided to have one worker node per **AP**. The routes used to simulate the movement of vehicles are randomly generated, for one hundred vehicles, which, according to our experiments, is an acceptable number given our resources.

This scenario's architecture follows the exact same structure of the previous one, displayed at Figure 22. We have a main thread that creates a controller and a position tracker thread. The controller thread launched depends on the strategy, we either launch the *reactive* or the *predictive* controller thread. This thread is responsible for the decision-making of the experiment, and relies on a shared structure with the following parameters.

```
{
    "car1": {
        "position": (x, y),
        "direction": (dx, dy),
        "associated_ap": "ap1",
```

```
    },
    ...
}
```

Listing 5.13: Vehicle data structure collected and maintained by the *position-tracker* threads.

The information displayed in Listing 5.13 is not enough to support this experiment. We have two more structures that are not shared but instead are managed entirely by the controller thread. The first one is the list of flow rules.

```
[
    {
        ap_id: 1,
        node_id: 2
    },
    ...
]
```

Listing 5.14: Structure with list of flow rules used by the controller thread.

This list (Listing 5.14) associates each **AP** with a specific worker node. By doing this we know that a specific **AP** is redirecting traffic to a certain node, and therefore know if the vehicle is using the optimal node or, for example, the bootstrap node. Another important data structure used and maintained exclusively by the controller thread is the deployments structure.

```
[
    {
        node_id: 1,
        apps: [
            "app_name_1",
            ...
        ]
    },
    ...
]
```

Listing 5.15: Structure with list of deployments used by the controller thread.

The deployments structure (Listing 5.15) informs the controller of which applications (*apps*) are deployed at each node. The framework has not been extensively tested for a multiple application scenario where vehicles consume different applications. Nonetheless, we left the option to test this alternative scenario in this experiment. With the deployment's information we can make sure that our **SDN** flows are correct (we don't want to redirect traffic to a node that does not host our application) but also we can keep track of the deployments for cost management reasons.

The position tracker thread manages all the vehicles in the simulation. This thread is responsible for creating and updating the structure at Listing 5.13. As we have mentioned in the previous section, we

gather information about each vehicle by reading their *position file*, an automated file created and updated by Mininet-WiFi, that contains the position of a given network element at any given instant.

A distinguishable characteristic of this experiment is the existence of two different strategies to migrate microservices. Starting with the reactive strategy, we describe its operation with the pseudocode below, Algorithm 2.

Algorithm 2: Reactive Controller for Deployments and Flows.

```
Data: Vehicle data, associated access points (APs), node information, vehicle positions, and
      directions.
Result: Updated deployments and flows for optimal routes.
while simulation is running do
    Collect the latest vehicle data;
    foreach vehicle in the simulation do
        if vehicle is not connected to an AP then
            Skip to the next vehicle;
        Determine the best available node for the vehicle based on its AP;
        if no suitable node is found then
            Skip to the next vehicle;
        Check if the vehicle is currently using the optimal node;
        if vehicle is already assigned to the best node then
            Continue without making changes;
        Check the vehicle's movement direction and position;
        if the vehicle is not leaving the AP coverage area then
            Identify a new node for deployment and route adjustment;
            Prepare a new deployment for the vehicle's application, if needed;
    Apply new deployments and modify network flows based on vehicle movements and node
      assignments;
```

The pseudocode presented above goes on to describe the reactive algorithm, and highlights its simplicity. We iterate all the vehicles in the simulation, and for each of them we want to know which one is in range of a **AP** that in turn is not redirecting traffic to the optimal node. After iterating all the vehicles, we know exactly to which nodes we need to deploy our microservice, and also the **SDN** flows. Regarding the cost parameter, we are not optimizing it for now, and simply, at each iteration of the simulation, we register the current global cost.

We also have a predictive strategy that is described with the pseudocode below, Algorithm 3.

Algorithm 3: Predictive Vehicle Routing Controller

Data: Vehicle positions, directions, associated access points (APs), node assignments, and flow data.

Result: Proactive deployments and flow routes for vehicles based on future movements.

while *simulation is active* **do**

- **Collect** the latest vehicle data;
- **foreach** *vehicle in the simulation* **do**
  - **if** *vehicle is not connected to an AP* **then**
    - **Skip** to the next vehicle;
  - **Check** the vehicle's current position and movement direction;
  - **if** *vehicle is not about to leave its current AP coverage area* **then**
    - **Skip** to the next vehicle;
  - **Predict** the next AP based on the vehicle's movement direction;
  - **if** *no valid next AP is found* **then**
    - **Skip** to the next vehicle;
  - **Determine** the node associated with the predicted next AP;
  - **if** *the next AP is managed by the same node as the current one* **then**
    - **Skip** further actions for this vehicle;
  - **Check** if a deployment exists on the predicted node;
  - **if** *no deployment exists for the vehicle's application* **then**
    - **Prepare** a new deployment on the predicted node;
  - **Add** the predicted node and corresponding AP to the list of new deployments and flows;
- **Apply** new deployments and update network flows based on predicted vehicle movements;

With the predictive algorithm, we iterate the vehicles in the simulation and try to predict the next **AP** for each vehicle. This calculation uses the direction of movement and assumes that the vehicle will keep the same direction. After we verify that the vehicle is moving into the range of another **AP**, and that this **AP** does not use the same worker node, we schedule the creation of a deployment and a **SDN** flow.

Having described both strategies, its time now to start the simulation. The initial configuration for this experiment is the same as seen in the telemetry of the previous scenario (Figure 23). This time we start with a 100 vehicle configuration to obtain more realistic results. The **K8s** nodes created for this

experiment are shown below in Table 16.

NAMES	STATUS	INTERNAL-IP
mobility-strategy-control-plane	Ready	172.18.0.8
mobility-strategy-worker	Ready	172.18.0.9
mobility-strategy-worker2	Ready	172.18.0.4
mobility-strategy-worker3	Ready	172.18.0.5
mobility-strategy-worker4	Ready	172.18.0.10
mobility-strategy-worker5	Ready	172.18.0.11
mobility-strategy-worker6	Ready	172.18.0.6
mobility-strategy-worker7	Ready	172.18.0.3
mobility-strategy-worker8	Ready	172.18.0.2
mobility-strategy-worker9	Ready	172.18.0.7

Table 16: Kubernetes nodes used in the Mobility Strategy scenario, all in *Ready* state.

Another very important piece of information is the mapping between **AP** and nodes. Since we have nine **APs** and nine nodes, each **AP** has its own node. Similarly to what is shown in Figure 24, we have the worker node 1 for the *ap1*, the node 2 for the *ap2*, and so on. The initial deployments of this scenario are also different. We have a single initial deployment that can be seen in Table 17.

NAME	READY	IMAGES	SELECTOR
mysimpleserver-deployment-1	1/1	mysimpleserver:latest	app=mysimpleserver

Table 17: Initial deployments for the Load Balancing scenario.

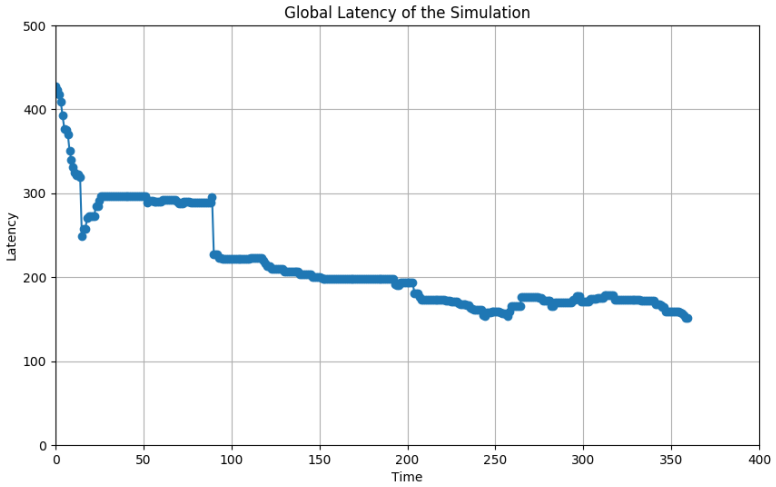
This deployment is of our simple HTTP server that exposes the `GET /getPodIP` endpoint. This endpoint, as the name suggests, returns the IPv4 address of the pod that is hosting the microservice. It is useful to check where the microservice that we are interacting with is hosted.

Regarding the **SDN** flows used in this experiment, we have exactly three flows in every **AP** at the start of the simulation. We have the usual basic flow that redirects traffic that does not match any rules to the controller (Listing 5.4) and also we have the two flows (Listing 5.6) that redirect traffic to the bootstrap node (mobility-strategy-worker). With these flow rules, every **AP**, if it receives a request that targets our

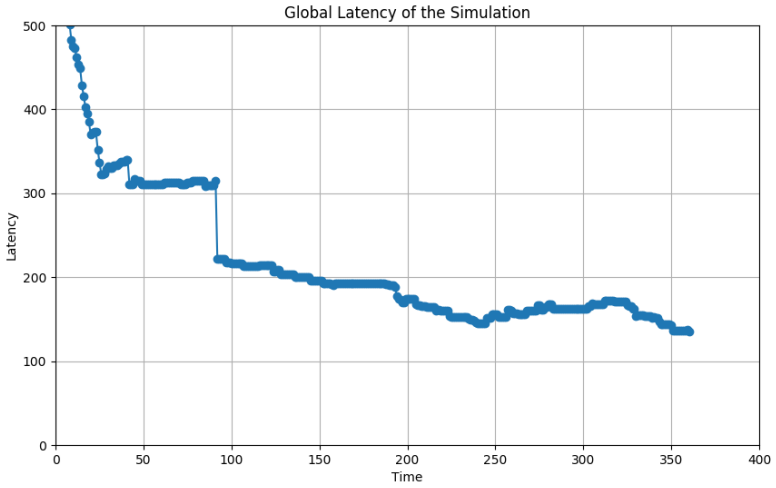
service, can redirect it to the bootstrap node. In this scenario there are no new deployments or flow rules, we just have a different logic on how to apply them.

Having defined the components of this experiment, it is now time to start the simulation. For each simulation we must define two parameters: the strategy that will be used (reactive or predictive) and the maximum cost (number of deployments) that we can have. We will obtain data regarding the global latency of the simulation at any given instant. Latency is measured in terms of proximity units, indicating how far a vehicle is from the ideal node. A latency of zero means the vehicle is consuming the service from the nearest possible node (e.g., if a vehicle is using Node 1 when Node 1 is the closest, latency is zero). If a vehicle is using a node that is further away, the latency increases by one unit for each additional node it must use (e.g., if the closest node is Node 1, but the vehicle is using Node 5, the latency is 4). The global latency presented in each graph is the sum of the individual latencies for all one hundred vehicles. We split each run into time instants, which correspond to the iterations performed by our controller thread. At each iteration it checks the data from all vehicles and makes a decision to migrate a microservice, to create a **SDN** flow or to do nothing. Each simulation takes a little less than ten minutes, which means that, given that the simulations are all around 350 iterations, the controller thread is making and enforcing a decision each 1.5 seconds.

We ran simulations with one hundred vehicles and tested all possible values for the maximum deployments. In Figure 28 below, we can see the initial results for a simulation using a single deployment. In this graph, and in the one in Figure 29, the Y-axis represents the total latency for the entire simulation, while the X-axis shows the progression over time.



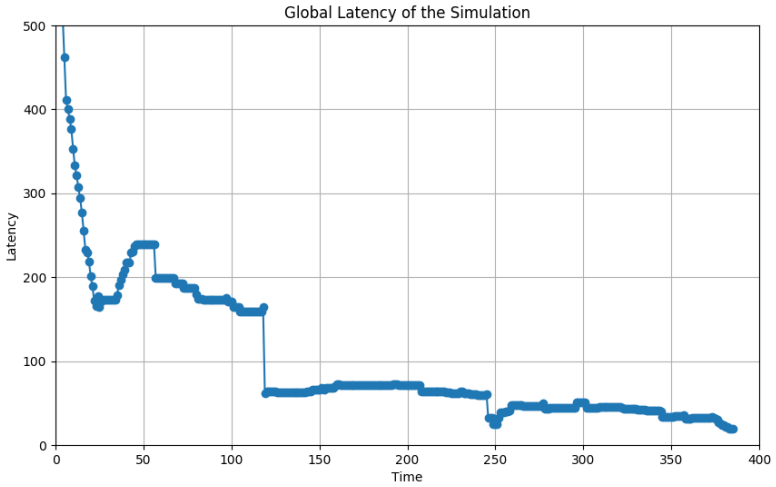
(a) Reactive strategy.



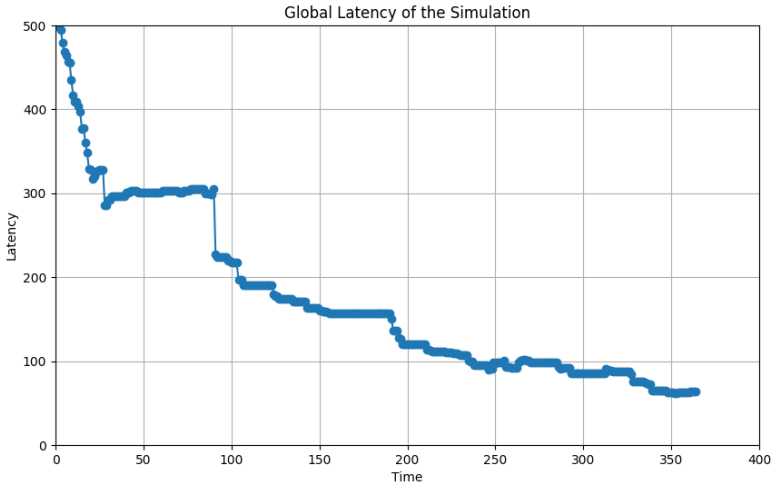
(b) Predictive strategy.

Figure 28: Global latency of both strategies for a simulation using a single deployment at any node and one hundred vehicles.

In the results presented above, we notice a slight trend for the global latency to reduce over time. In this particular case we are using a single deployment on a single node, so we must keep this deployment closest to the area with more vehicles. This task, for now, its performed better by the predictive strategy, by a small margin. Below, on Figure 29, we increased the number of deployments in the simulation to three.



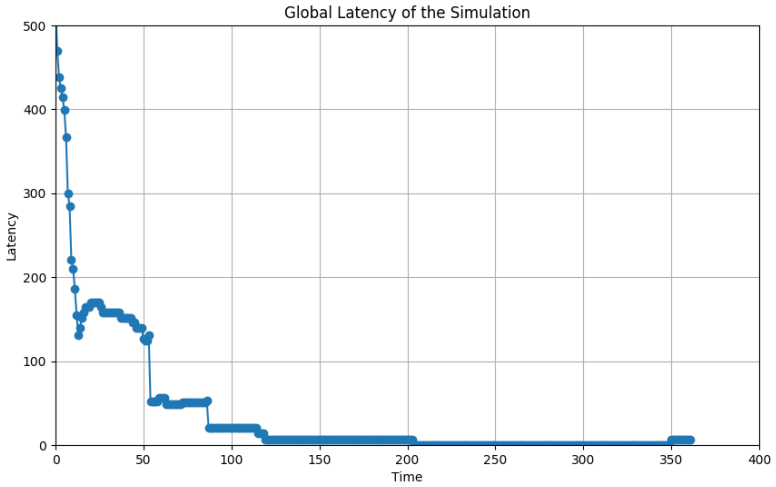
(a) Reactive strategy.



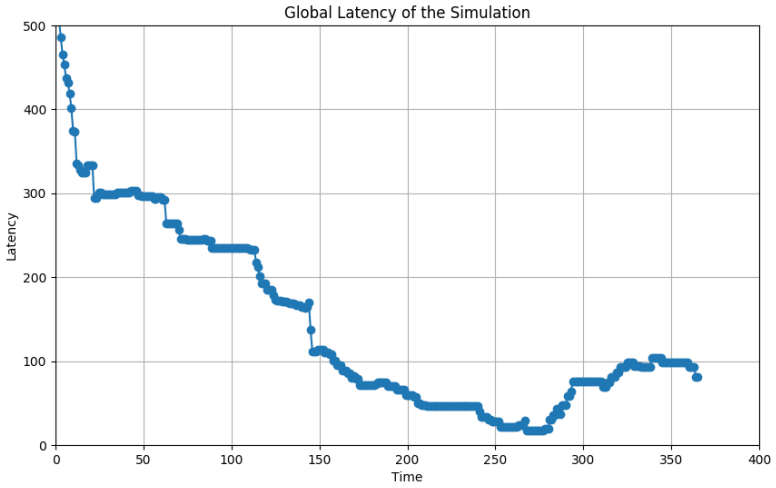
(b) Predictive strategy.

Figure 29: Global latency of both strategies for a simulation using three deployments and 100 vehicles.

Now the downwards trend is even more noticeable and the reactive almost reaches zero latency at the end of the simulation. In the first few dozen iterations, the latency is always higher because we have not reached the limit of deployments of the simulation. Once we reach that limit, we just have to guarantee that the deployments are in the busiest areas. Now, the reactive strategy at Figure 29a seems the best strategy since it achieves a lower latency earlier in the simulation. Below is yet another plot, this time with five deployments.



(a) Reactive strategy.



(b) Predictive strategy.

Figure 30: Global latency of both strategies for a simulation using five deployments and 100 vehicles.

It is now clear to us that the reactive strategy is more efficient in terms of achieving a lower latency earlier on in the simulation. With increasing number of nodes, we get lower and lower latencies, as it would be expected. To better analyze all the data, Table 18 presents a comparison of simulation results for the different mobility strategies and maximum costs, i.e., deployments.

Cost	Strategy	Avg Latency	Min Latency	Iterations to Min Latency	Max Latency	Std Dev	Median Latency
1	Reactive	214.22	151	358	427	56.62	198.0
1	Predictive	221.32	135	360	576	89.52	193.0
2	Reactive	106.11	40	357	510	82.98	57.0
2	Predictive	170.86	48	356	457	95.22	152.0
3	Reactive	103.84	19	385	543	91.72	64.0
3	Predictive	175.27	62	352	516	102.24	157.0
4	Reactive	94.71	21	346	556	96.73	48.0
4	Predictive	145.80	7	370	463	127.44	96.0
5	Reactive	37.76	0	204	524	78.20	7.0
5	Predictive	146.02	17	268	535	111.62	99.0
6	Reactive	31.85	0	180	495	74.55	0.0
6	Predictive	123.10	12	281	588	129.16	51.0
7	Reactive	5.56	0	18	540	39.54	0.0
7	Predictive	105.14	0	283	531	121.74	41.0
8	Reactive	18.43	0	21	582	84.11	0.0
8	Predictive	105.08	0	380	491	110.89	50.0

Table 18: Simulation results comparing latency metrics across different strategies and node configurations.

The table includes key metrics like average latency, time to reach minimum latency and a few more statistics. The average latency lowers with the number of deployments, but also depending on the strategy. It seems that the reactive strategy achieves better results. The consequences of using a reactive strategy are that, theoretically, the vehicles will be consuming a service from a non-optimal node for at least the amount of time needed to migrate the service to a closer location. A prediction algorithm that could always guess the next position of the vehicle would, in theory, perform better. What we see in practice is that mistakes are made by the prediction algorithm and the reactive strategy, by capitalizing on fast deployments and perfect information, achieves a better result. The predictive model is outperformed also due to the lack of real-time data on the road and environmental conditions. For example, if the simulation controller knows that a certain vehicle is in a street that forces it to make a right turn, it will not contemplate other alternatives, and it will immediately migrate the service, if needed. This additional data could allow for the development of a more sophisticated predictive algorithm, one that could easily outperform the reactive strategy.

In summary, this last experiment presented two different strategies to migrate microservices closer to vehicles. It also served as a final endurance test for the framework, and the results have been satisfactory. Both predictive and reactive strategies demonstrated their unique strengths: the predictive strategy offered more stable latency over time, while the reactive strategy as quicker to adapt to changes in vehicle locations, leading to lower minimum latencies in certain scenarios. Despite these differences, the overall performance of the system has proven that the framework is capable of dynamically managing microservice migrations in real-time environments.

It is fair to say that, based on the results obtained throughout all the experiments, we have validated the functionality and usability of the framework for at least three scenarios: video streaming, load balancing and microservice mobility.

Chapter 6

Conclusions

In this final chapter we summarize the key findings of this dissertation and highlight the contributions made by this work to the field of Software Defined Vehicular Networks. The chapter finishes with a few suggestions on potential future research directions.

6.1 Summary

The primary goal of this project was to develop a microservice's mobility strategy that can be tested in a realistic emulated traffic scenario. To achieve this goal we had to design a framework that allows us to emulate a vehicular network. The framework must also include a kubernetes cluster that will be hosting the microservices. We defined this framework conceptually at Chapter 3, and delve into the details of our implementation at Chapter 4.

Developing the software, referred to as the *framework*, which integrates all the individual components, was arguably the most challenging aspect of this project. This framework consists of three components: (1) a vehicular network capable of emulating realistic traffic scenarios, (2) a Kubernetes-based network for hosting microservices, and (3) a custom software solution that triggers deployments and configures **SDN** flows based on real-time traffic information.

This framework enabled us to conduct five distinct experiments, as detailed in Chapter 5. These experiments demonstrated the versatility of the system by allowing us to configure various traffic scenarios, test different simulation scales (e.g., number of access points, vehicles), and experiment with diverse applications. The key for this versatility was to use configuration files that allowed us to customize every detail of the simulation. We developed and tested scenarios for microservice replication, migration, video streaming, load balancing, and mobility management.

There are limitations to the capabilities of the system, most of them are presented in Section 4.3. The main limitation is the fact that Kubernetes was not designed for geographic microservice migration.

Additionally, we haven't found a solution to migrate stateful applications, meaning that we can only use stateless microservices and there is no easy way to implement this feature. There are other limitations, for example the slow process of loading Docker images onto worker nodes, which made it impossible to do this as part of the migration process. Moreover, the framework lacks a proper emulation of network latencies between vehicles and worker nodes. These last limitations have to do with how the vehicular network and cluster network are connected, which is through a single component, over a single link.

Despite these limitations, the framework proved to be a strong candidate as a lightweight, open-source solution for simulating realistic vehicular networks and supporting a variety of applications. There are many possible improvements to the framework, some of them aimed at the realism and performance, but also to its usability. Despite this, this project offers a solid foundation. These possible enhancements will be discussed in detail in Section 6.3.

6.2 Contributions

The primary contribution to the field of Software-Defined Vehicular Networks, particularly in the area of microservice migration, is the development of a framework capable of emulating complex vehicular networks and dynamically creating deployments and SDN flows based on real-time traffic data.

This framework serves as a convenient workbench for researchers to quickly test mobility strategies or load balancing algorithms in specific locations. The process is straightforward, with all tools being accessible and open-source, allowing for extensive customization without licensing restrictions. This comes with the disadvantage of being hard to configure and requiring some level of technical expertise.

The set of experiments that served to validate the framework not only demonstrate its capabilities but also offer a starting point for researchers who want to test and build something on top of it. These experiments provide a reference for future research aimed at optimizing microservice deployment and network efficiency in vehicular environments.

6.3 Future work

There are a few promising avenues for future research and improvements to the framework. We can start looking into the improvements to the current solution.

The question of usability has been presented many times across this dissertation, and it might be the single most important improvement. It can be solved by creating a pre-configured virtual machine. Users with newer ARM64 chips will still not be able to use it, as we have discussed in Section 4.2.

Creating realistic simulations requires an accurate representation of the network. The current solution for enabling communication between the vehicular network and the kubernetes cluster nodes is limited, since it relies on a single entry point. By tackling this issue and by simulating network delays we would no longer need a configuration file that specifies which worker nodes are closest to which access points **APs**. Instead, we could implement a software module to dynamically measure delays, allowing the system to determine node proximity based on actual network conditions.

The microservice orchestrator should be capable of handling stateful application migrations. Support for these applications would open many more use cases to the framework. Also, the orchestrator has to natively support mechanisms to deploy nodes on different locations and deploy microservices on specific nodes i.e., locations. Despite our success on adapting Kubernetes to our project, we would lack mechanisms to scale our applications and use multi-application clusters or even multi-cluster system. Since we rely on static configuration files for manual node selection, we could not dynamically scale by creating new nodes. In an ideal scenario, we would have several small edge servers that allow us to control resource usage and deploy nodes specifically on these servers. This setup is not possible, or at least not convenient to create using Kubernetes. The most efficient solution has to be designed specifically for the network edge, where resources are not abundant.

The current framework can not scale to emulate high density scenarios with thousands of vehicles. This is partially a resource limitation since we were only able to validate the framework for a maximum of one hundred vehicles. But it is also a problem with the way Mininet-WiFi creates and manages the network components. Improving the scalability of the framework will require addressing both factors. The first one is easy, since we can simply use a more powerful system e.g., by using a cloud provider. The latter is a harder challenge, since it might require to use another network emulator and consequently change the whole dynamic of the framework. These enhancements would enable the framework to be used not only for academic purposes but for more realistic and useful emulations of real-world applications.

Another smaller improvement to the framework is the integration with **SUMO**'s Traffic Control Interface, TraCI. This integration would allow the simulation controller to gather more accurate data on traffic patterns in real time. This information would allow for more sophisticated migration algorithms and therefore improve the overall functionality of the framework, making it even more valuable.

Regarding the results presented in Section 5.6, specifically those indicating that the predictive strategy was not as efficient as the reactive strategy, it is important to note that the predictive model implemented was relatively simple. As a future improvement, we should focus on implementing a more advanced predictive strategy, potentially leveraging machine learning techniques or more sophisticated mobility prediction

algorithms. To implement this improvement, we would need to leverage the integration with TraCI. A more sophisticated strategy requires more data, especially about the surroundings of the vehicle and the road where it is located.

The seventh and final improvement to the framework is related to its security. The security of vehicular networks is still an active area of research and lacks the standardization found in other domains. The framework inherits the security issues associated with **SDN**, particularly the single point of failure in both the **SDN** controller and the simulation controller. Implementing redundancy measures such has having multiple controllers could mitigate the damages caused by potential attacks or failures. Additionally, vehicle authentication standards would ensure that only authorized vehicles can access the network (and its resources) and communicate with other devices.

Now that we have seen some of the improvements that we could use to make this framework more usable and robust, there are also some interesting research topics that can consolidate this work and the field in general. We propose the following three research topics.

The first of the research topic is the investigation of stateful microservice migrations within **SDVNs**. This study would focus on maintaining application state during migrations between edge servers while minimizing latency and ensuring data consistency. There are many applications that might require a state and have real-time processing needs, such as autonomous vehicle navigation systems.

Another research topic involves optimizing resource allocation and load balancing across multiple edge servers. This topic focus on the dynamic scaling strategies that allows for an effective distribution of workloads in a vehicular environment. With the integration of real-time traffic data into the framework, this study could lead to an intelligent and efficient microservice deployment strategy that aligns with typical vehicular patterns.

Lastly, the framework could host comparative studies of various orchestration tools and their effectiveness in managing microservices within vehicular environments. Researchers could investigate the performance, latency and resource impacts that each orchestrator has, and conclude which one is adequate for this scenario.

In conclusion, this work explores the novel field of **SDVNs** with a focus on emulating support for microservices and their respective migrations with the goal of maintaining service availability and lower latencies. This dissertation essentially bridges the theoretical concepts and practical implementations of this field. As **VANETs** continue to evolve, and with the advent of **ITSs**, this work stands as a testament to the potential of combining the flexibility of software-defined networking with the dynamic needs of vehicular environments.

Chapter 7
References

[1] Population Reference Bureau. Share of urban population worldwide in 2023, by continent. https://www.prb.org/international/indicator/urban/table, December 2023.

[2] Eurostat. Key figures on europe. https://ec.europa.eu/eurostat/en/web/products-key-figures/w/KS-EI-23-001, 2023.

[3] U.S. Census Bureau. 5-year american community survey (2017-2021). https://www.census.gov/newsroom/press-kits/2022/acs-5-year.html.

[4] Chao Wang. *The relationship between traffic congestion and road accidents: an econometric approach using GIS*. PhD thesis, Kingston University, 2010.

[5] IEA. Global ev outlook 2023. https://www.iea.org/reports/global-ev-outlook-2023.

[6] Maryam Gillani, Hafiz Adnan Niaz, Muhammad Umar Farooq, and Ata Ullah. Data collection protocols for vanets: a survey. *Complex & Intelligent Systems*, 8(3):2593–2622, 2022.

[7] Muhammad Arif, Guojun Wang, Md Zakirul Alam Bhuiyan, Tian Wang, and Jianer Chen. A survey on security attacks in vanets: Communication, applications and challenges. *Vehicular Communications*, 19:100179, 2019.

[8] G. Rakesh and Meena Belwal. Vehicle collision avoidance in a vanet environment by data communication. In *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pages 238–242, 2019.

[9] Alessandro Bazzi, Barbara M. Masini, and Alberto Zanella. Cooperative awareness in the internet of vehicles for safety enhancement. *EAI Endorsed Transactions on Internet of Things*, 3(9):e5, Jan. 2017.

[10] Michael Lee and Travis Atkison. Vanet applications: Past, present, and future. *Vehicular Communications*, 28:100310, 2021.

[11] Hani Sami, Azzam Mourad, and Wassim El-Hajj. Vehicular-obus-as-on-demand-fogs: Resource and context aware deployment of containerized micro-services. *IEEE/ACM transactions on networking*, 28(2):778–790, 2020.

[12] Gaurang Naik, Biplav Choudhury, and Jung-Min Park. Ieee 802.11 bd & 5g nr v2x: Evolution of radio access technologies for v2x communications. *IEEE access*, 7:70169–70184, 2019.

[13] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[14] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51, 2015.

[15] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[16] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206, 2014.

[17] Open Networking Foundation. *OpenFlow Switch Specification Version 1.5.1*, 2015. Available at https://opennetworking.org/software-defined-standards/specifications/.

[18] Rob Enns, Martin Bjorklund, Juergen Schoenwaelder, and Andy Bierman. Network configuration protocol (netconf). Technical report, IETF RFC 6241, 2011.

[19] Qingyue Long, Yanliang Chen, Haijun Zhang, and Xianfu Lei. Software defined 5g and 6g networks: A survey. *Mobile networks and applications*, 27(5):1792–1812, 2022.

[20] Yassine Maleh, Youssef Qasmaoui, Khalid El Gholami, Yassine Sadqi, and Soufyane Mounir. A comprehensive survey on sdn security: threats, mitigations, and future directions. *Journal of Reliable Intelligent Environments*, 9(2):201–239, 2023.

[21] Jitendra Bhatia, Yash Modi, Sudeep Tanwar, and Madhuri Bhavsar. Software defined vehicular networks: A comprehensive review. *International Journal of Communication Systems*, 32(12):e4005, 2019.

[22] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.

[23] Yong Li and Min Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[24] Corrado Rametta, Gabriele Baldoni, Alfio Lombardo, Sergio Micalizzi, and Alessandro Vassallo. S6: a smart, social and sdn-based surveillance system for smart-cities. *Procedia Computer Science*, 110:361–368, 2017. 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops.

[25] Somansh Kumar and Ashish Jasuja. Air quality monitoring system based on iot using raspberry pi. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1341–1346, 2017.

[26] Syeda Zeenat Marshoodulla and Goutam Saha. Sdiotpark: A data analytics framework for smart parking using sdn-based iot. *IEEE Internet of Things Journal*, 11(11):20030–20039, 2024.

[27] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[28] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[29] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 ieee 14th international symposium on network computing and applications*, pages 27–34. IEEE, 2015.

[30] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[31] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly Media, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[32] Docker. Docker swarm. https://docs.docker.com/engine/swarm/, 2024. Accessed: 2024-10-25.

[33] Ibrahim Shaer, Anwar Haque, and Abdallah Shami. Multi-component v2x applications placement in edge computing environment. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2020.

[34] Ibtissam Labriji, Francesca Meneghello, Davide Cecchinato, Stefania Sesia, Eric Perraud, Emilio Calvanese Strinati, and Michele Rossi. Mobility aware and dynamic migration of mec services for the internet of vehicles. *IEEE Transactions on Network and Service Management*, 18(1):570–584, 2021.

[35] Abdallah Moubayed, Abdallah Shami, Parisa Heidari, Adel Larabi, and Richard Brunner. Edge-enabled v2x service placement for intelligent transportation systems. *IEEE Transactions on Mobile Computing*, 20(4):1380–1392, 2020.

[36] Wanlu Zhang, Chenhui Tao, Jingjing Luo, Fu-Chun Zheng, and Lin Gao. Context-aware service placement at the edge in vehicular networks. In *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, pages 1–5. IEEE, 2023.

[37] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, January 2011.

[38] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[39] Ramon R. Fontes, Samira Afzal, Samuel H. B. Brito, Mateus A. S. Santos, and Christian Esteve Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 384–389, 2015.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[41] Ryu sdn framework documentation. https://ryu.readthedocs.io/en/latest/index.html, 2024. Accessed: 2024-10-25.

[42] Minikube documentation. https://minikube.sigs.k8s.io/docs/, 2024. Accessed: 2024-10-25.

[43] Kind documentation. https://kind.sigs.k8s.io/, 2024. Accessed: 2024-10-25.

[44] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[45] Convert ova to qcow2 and start it with utm. https://simo9265.medium.com/convert-ova-to-qcow2-and-start-it-with-utm-13fa3fc4c3db, 2024. Accessed: 2024-10-25.

[46] Conda - package, dependency and environment management for any language. https://docs.conda.io/en/latest/, 2024. Accessed: 2024-10-25.

[47] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[48] BBC Earth. Sharks feast on whale | 4k uhd | blue planet ii | bbc earth. https://youtu.be/SzEJIXKTM-8?si=ogUk0LYaaL7Ekj0_, 2023. Accessed: 2024-10-25.