Halvor Groven

Developing a Testbed for VANET Protocols using Drones

Master's thesis in Communication Technology
Supervisor: Peter Herrmann
Co-supervisor: Ergys Puka
August 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Halvor Groven

Developing a Testbed for VANET Protocols using Drones

Master’s thesis in Communication Technology
Supervisor: Peter Herrmann
Co-supervisor: Ergys Puka
August 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Title:	Developing a Testbed for VANET Protocols using Drones
Student:	Halvor Groven

Problem description:

In vehicular ad hoc networks (VANETs), an essential aspect of protocols to be implemented and commercialized is rigorous testing. Simulations are often used in combination with field tests to accurately predict the behavior and performance of vehicle-to-vehicle (V2V) protocols in real-life scenarios. However, field tests require a substantial amount of time and resources to realize, to which researchers and developers may not have access. This calls for further research into ways of improving the current situation of testing VANET protocols. Uncrewed aerial vehicles (UAVs), or drones, are remotely controlled, pilot-less vehicles. Compared to landbased vehicles that would typically be used in field tests, they are relatively cheap. In addition, drones are highly programmable and automatable and can be maneuvered relatively easily through open-source software and with little modification, depending on the drone. In this thesis, we design and implement a testbed that integrates traffic simulation software with drone automation to investigate the potential for using drones in field tests of VANET/V2V protocols.

Date approved:	2021-04-01
Supervisor:	Peter Herrmann, NTNU

Abstract

The number of vehicles on our roads is increasing every day. This results in growing challenges related to road congestion and traffic safety, and novel technology and solutions must be developed to mitigate these issues. In the future, Intelligent Transportation Systems (ITS) and Vehicular ad hoc networks (VANET) will play a crucial role in overcoming these challenges, by making traffic more interconnected, unified, informed, and safe. Due to the unique properties of VANET, like rapidly changing network topology, high mobility vehicles and high requirements for reliability, new and novel communication protocols need to be developed specifically for this purpose.

These communication protocols need to be rigorously tested before commercialization, as correct operation may be vital for the safety of drivers and passengers. Today, the most widespread way of testing such protocols is through simulated experiments. However, simulations have their shortcomings, and real-world experiments should be conducted to obtain the most accurate and reliable results. Furthermore, conducting real-world experiments with cars can become very resource-demanding, time-consuming, and expensive. In this thesis, we have explored a novel approach to address this, by investigating how Uncrewed Aerial Vehicles (UAV)s can benefit the domain of testing new communication protocols designed for VANET. In particular, we approach this by designing and developing a novel testbed for routing- and data dissemination protocols created for VANET, by combining the traffic simulator SUMO with UAV technology. Our solution revolves around simulating vehicles on a computer, and duplicate their behavior in airborne UAVs. From our work, we conclude that UAVs can be used instead of cars when conducting real-world testing of VANET protocols, but that it should not be used as a replacement for real-world tests with real cars. We also identify other limitations.

The code developed in this thesis, as well as example files and setup instructions, has been published on GitHub[Gro21].

Sammendrag

Antallet biler i trafikken øker hver dag. Dette resulterer i voksende utfordringer relatert til økt trafikkbelastning- og sikkerhet, og ny teknologi og nye løsninger må utvikles for å adressere disse utfordringene. I fremtiden kommer Intelligente transportsystemer (ITS) og Vehikulære ad hoc-nettverk (VANETs) til å spille en avgjørende rolle i måten vi håndterer disse utfordringene på, ved å muliggjøre et trafikkbilde som er mer sammenkoblet, enhetlig, informert og trygg. På grunn av de unike egenskapene til VANET, som raskt skiftende nettopologi, kjøretøyer med høy mobilitet og høye krav til pålitelighet, må nye kommunikasjonsprotokoller utvikles spesielt for dette domene.

Disse kommunikasjonsprotokollene må testes grundig før kommersialisering, ettersom korrekt oppførsel kan være avgjørende for sikkerheten til sjåfører og passasjerer. I dag er den mest utbredte måten å teste slike protokoller på gjennom simulerte eksperimenter. Simuleringer har imidlertid sine mangler, og eksperimenter i den virkelige verden bør utføres for å få de mest nøyaktige og pålitelige resultatene. Videre kan utførelse av virkelige eksperimenter med biler bli svært ressurskrevende, tidkrevende og dyrt. I denne oppgaven har vi utforsket en ny tilnærming for å adressere dette, ved å undersøke hvilken nytteverdi droner kan gi ved å brukes i testing av nye kommunikasjonsprotokoller designet for VANET. Mer spesifikt tilnærmet vi oss dette ved å designe og utvikle en testbed for ruting- og dataformidlingsprotokoller, ved å kombinere trafikksimulatoren SUMO med droneteknologi. Løsningen vår går ut på å simulere biler på en datamaskin, og duplisere oppførselen deres til flyvende droner. Fra arbeidet vårt konkluderer vi med at droner kan brukes istedenfor biler når vi utfører virkelige tester av VANET-protokoller, men at det ikke burde brukes som en erstatning for virkelige tester med virkelige biler. Vi identifiserer også andre begrensninger.

Koden som ble utviklet i sammenheng med denne masteroppgaven, i tillegg til eksempelfiler og installasjonsinstruksjoner, har blitt publisert på GitHub[Gro21].

Preface

This thesis concludes my Master of Science education in Communication Technology at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway, and was carried out in in the spring of 2021.

I would like to thank my responsible professor and supervisor Peter Herrmann and my co-supervisor and PhD candidate Ergys Puka at the Department of Information Security and Communication Technology for valuable support and guidance during this thesis. I would also like to thank Pål Sturla Sæther, also at the Department of Information Security and Communication Technology for assisting with equipment. Finally I would like to express my gratitude to the several drone pilots at Trondheim Modellflyklubb for guidance and motivating words at Udduvoll Airfield outside Trondheim.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

DTW Dynamic Time Warp.

FC Flight Controller.

GBC Ground Based Computer.

GCS Ground Control Station.

GNSS Global Navigation Satellite Systems.

GPS Global Positioning System.

GUI Graphical User Interface.

HIL Hardware-in-the-loop.

ITS Intelligent Transportation Systems.

MAVLink Micro Air Vehicle Link.

NTNU Norwegian University of Science and Technology.

OBU On-Board Unit.

RSUs Road-side Units.

RTL Return To Launch.

SITL software in the loop.

SUMO Simulation of Urban MObility.

SV simulated vehicle.

TraCI	Traffic Control Interface.
UAV	Uncrewed Aerial Vehicles.
V2I	Vehicle-to-infrastructure.
V2V	Vehicle-to-vehicle.
VANET	Vehicular ad hoc networks.

Chapter 1
Introduction

In this chapter we introduce the motivation for this thesis. In addition we define the research questions and scope of as well as our approach to answer them.

1.1 Motivation

As the demand for transportation and mobility is increasing in major cities, traditional means of traffic planning and expansion are becoming insufficient for handling the issues that arise with the number of vehicles increasing every day. Escalating issues with congestion followed by a higher risk for accidents and emergency scenarios call for new approaches for tackling continuously growing challenges. Intelligent transportation systems (ITS) are a necessary supplement to traditional approaches to mitigate these issues.[SH18][its]

Intelligent Transportation Systems (ITS) describe systems that incorporate telecommunications, electronics, and information technology to enhance safety and mobility in transportation. ITS aims to employ real-time data flows between infrastructure and vehicles to create a beneficial situation where every part of the transportation domain is more interconnected, unified, and informed[GMSG12]. In 2012, a report from the European Commission[Dir] listed ITS as the most important factor for achieving seamless integration between modes of transportation across Europe, besides the development of infrastructure.

Vehicular ad hoc networks (VANET) is widely perceived as a promising technology to realize the future of ITS[EZL14]. VANET facilitates data exchange between vehicles, either directly through Vehicle-to-vehicle (V2V) communication or indirectly through roadside infrastructure with Vehicle-to-infrastructure (V2I) communication[LCQ$^+$18]. The goals of VANET closely resemble that of ITS in that it aims to provide enhanced safety and comfort for passengers and increase traffic efficiency. When wide adoption of VANET technology becomes the norm, vehicles and passengers will have more information about the traffic environment which can be crucial for making the correct decisions at the right time. VANETs have unique characteristics that spark new challenges. Most notable are the mobility patterns of vehicles as well as a network topology that changes rapidly[CJPZ14]. Due to these characteristics, designing and developing communication protocols for routing and data dissemination in VANETs is challenging.

New applications and technology designed and created for VANET needs to be rigorously tested before being implemented in the real world. One reason that testing technology related to VANET-applications is crucial is that ensuring correct operation of the underlying technology can be vital for drivers and passengers. If applications receive any data that is too late or erroneous, vehicle behavior that may put passengers in danger or even cause accidents may be triggered[WSGY19]. Examples of this include wrongful broadcasting of road conditions to surrounding vehicles, or a safety application that is falsely triggered in a vehicle.

Today, a widespread method of evaluating communication protcols designed for VANET is through simulated experiments. For simulating a realistic VANET scenario, mainly two components are needed: A traffic simulator and a network simulator. Respectively, these are used to simulate vehicle and driver behavior, and network behavior, including signal propagation and protocol stack[BMM13a]. There are several challenges related to this way of testing and evaluating the performance of VANET protocols. Creating realistic simulations to model real world scenarios can quickly become very complex when including different aspects of VANET, like signal propagation- and reflection, network protocol stacks, and vehicle mobility[RGFW15]. Simulations are only as good as the underlying mathematical models used for each simulation layer. Thus, even though both network- and mobility simulators have grown more accurate over the years, the goal of perfectly reflecting every aspect of real scenarios is very difficult.

A real-life field experiment should be conducted to obtain the most valuable and accurate results when assessing VANET protocols or applications. However, as pointed out by many authors, deploying and testing VANET applications and protocols in real-life experiments are both resource-demanding and expensive[WNF21][CSAZ09] [WCC$^+$07] [BMM13a][ZHC$^+$10][MSS17]. Renting a location suitable for testing as well as buying or renting vehicles quickly become very expensive. In addition, obtaining, preparing, and installing On-Board Unit (OBU)s, the hardware and software installed into each vehicle to handle the V2V/V2I communication, can be demanding in terms of resources and time consumption. These resources may not be available to researchers and developers designing novel technology for VANET purposes. As a result, even though simulated tests are not flawless, they are a necessary substitute to assess the performance of new VANET applications and communication protocols. This thesis aims to combat this issue by investigating a new approach for executing field experiments of VANET protocols, aiming to reduce the costs and resource demand involved, using Uncrewed Aerial Vehicles (UAV)s.

UAVs, or drones, are aircrafts that do not have an onboard pilot but can be controlled either manually with a controller or with different levels of automation through software. Initially used for military purposes, drones have entered the consumer market in later years and are now used for multiple applications like aerial photography, express delivery, and traffic monitoring[WFZ$^+$16]. Other applications include area mapping, precision agriculture, and search and rescue. In this thesis we will focus on a specific class of drones, namely multi-copter drones[mul]. We argue that this drone type, have interesting abilities that may make them suitable for imitating the movement patterns of cars. For instance, multi-copter drones have the ability to stay stationary in the air while hovering and making rapid changes to their direction of flight. In addition, several drones either come with or can be equipped with sensors for automation purposes, like cameras, systems for localization, and communication equipment[WCC$^+$07].

There is a call for more research into ways of reducing costs and resource demand relation to field tests of VANET protocols and applications. In thesis we investigate the possibility of utilizing drones as a replacement of vehicles in field test by designing a testbed in which vehicle mobility is simulated in a traffic simulator, but also duplicated by drones in the real world. We aim to get insight into whether using drones as a replacement for vehicles for VANET-testing purposes can be a feasible option to facilitate cheaper, accurate, and rapid testing of VANET-protocols.

1.2 Related Work

There are several examples of initiatives and consortiums that work to accelerate the adoption of ITS and VANET technology. One of them is C-Roads[1], a European initiative that conducts testing and implementation of ITS services in several countries in Europe, aiming to accelerate the implementation of ITS systems in our society. Another example is Virginia Smart Road, which is a state-of-the art, closed test-bed facility, and is one of the worlds most advanced test facilities for transportation research[Vir][Han18]. Launched in 2017, the Virginia Smart Road consist of more than eight kilometers of roadbed that can be used to conduct experiments for a vast variety of scenarios. Both of these initiatives are state sponsored, having large financial forces behind them.

We found that most testbeds developed for VANET purposes are designed as fully simulated testbeds. However, we found some testbeds that were created to integrate real world aspects with simulated aspects.

[1]https://www.c-roads.eu/platform.html

in [SVB18], the authors presented a Hardware-in-the-loop (HIL) testbed for co-operative applications related to V2V/V2I communication. The solution revolves around assigning real-life communication hardware to simulated vehicles and controlling their interactions with a developed software called an *orchestrator*. This enables not only testing of applications, but also real hardware used for communication. The downside is that the traffic environment is fully simulated.

In [BSK$^+$18], the authors proposed an approach to couple real-time HIL-components used in VANET with simulated vehicles. It aims to bridge the world between discrete simulations and real-time systems from the real world. To do this, they created an interface be called the *Ego Vehicle Interface* (EVI). Even though this approach does not directly deal with testing communication protocols in VANET, it revolves around connecting discrete vehicle simulation with real world hardware, which is what we will do in this thesis. However, in this proposed approach, vehicle mobility is also fully simulated.

As mentioned, most of the testbeds we found for VANET purposes was fully simulated testbeds, combining different types of simulators to include several aspects related to VANET. In [CMM$^+$16], the authors combined the traffic simulator VISSIM and the network simulator NS-3 to create a integrated simulation environment for V2V/V2I communication. The authors in [KKK$^+$14], developed V2XREF (V2X runtime emulation framework), a framework designed for simulated V2V/V2I testing. They altered the models for network simulation to fit the specifications of VANET. Finally [ALJN19] designed and developed a simulation framework to test different aspects of cooperative intelligent transportation systems. However, it mainly aimed to test platooning, i.e., to have a group of cars drive themselves simultaneously, by cooperating.

[WNF21] present a comprehensive review of several widespread simulation tools that can be used for evaluating novel technologies and communication protocols designed for VANET. The authors point out that with emerging technologies such as 5G, SDN and edge computing, there has been a new wave of research conducted on VANET and surrounding technologies to enable the future of autonomous vehicles. However, they also point out that many of these simulators designed for VANET purposes are outdated and lack the capability of modeling several aspects of real world scenarios.

From our literature review, we found limited research into new ways of testing communication protocols designed for VANET in a real world environment with real moving vehicles, at a lower cost. However one approach have been proposed by the authors in [BMM13b]. Their approach involves swapping standard VANET communication terminals with smart phones, acting as VANET nodes. The solution involves a central server which relays messages between these VANET-nodes, utilizing already existing cellular infrastructure. The downside of this is that no direct V2V communication is performed, and real cars are still needed to perform tests. This idea was not implemented but the authors listed it for future work. The idea of using smart phones as VANET terminals is somewhat adapted in this thesis.

We also found limited research surrounding the utilization of drones in relation to VANET. However, it is not unheard of. For instance, in [WFZ+16], the authors devised a infrastructure-less, UAV assisted VANET system called VDNet in which vehicles are equipped with on-board drones that assist the dissemination of data in the V2V network. The solution showed promising results in relation to end-to-end delay and message delivery. The authors utilized quad-copter drones in their solution. Although we will used drones in a different way than here, this shows that UAVs already have some usecase in relation to VANET.

We identify that there is a lack of research into ways of performing assessment of communication protocols designed for VANET in a real world environment, at a lower cost. We see it as an open challenge to design a testbed that simultaneously lowers costs, resource consumption and time consumption, while also being available and easy to use.

1.3 Scope

We first present the research question that will guide the work of this thesis. Then, we define the our scope.

RQ1: *How can assessment of VANET protocols benefit from replacing vehicles with UAVs?*
RQ2: *How can mobility simulators be used to project vehicle trajectories in the real world using UAVs?*
RQ3: *How can drones be integrated with simulation software to create a testbed for routing- and data dissemination protocols in VANET?*

To emphasize, the overall goal of this thesis is to utilize software for traffic simulation combined with platforms and software communicating with UAVs to investigate the potential for using drones in testing of VANET protocols. In particular, we approach this by designing and implementing a testbed for VANET protocols, aimed towards researchers and developers designing novel data dissemination schemes and routing protocols in need of a relatively low-cost solution. We limit the scope of the testbed to support data dissemination- and routing protocols. Reactive protocols, meaning protocols that influence the vehicle- or driver behavior, is not supported by the testbed. This is because of the large layers of complexity added if including this category of protocols. For the same reason, the testbed is intended for protocols designed for V2V communication only. Still, we argue that we can obtain good results and present a valuable discussion revolving the research questions.

1.4 Outline

The structure of this master thesis is as follows:

- Chapter 2: A description of the methodology used.
- Chapter 3: Background theory on related concepts, tools and hardware.
- Chapter 4: Overview and detailed architecture description of the implemented testbed.
- Chapter 5: Description of experimental setup and how experiments were executed.
- Chapter 6: Presentation of the the data collected from the experiments.
- Chapter 7: Discussion. Results are discussed in light of the research questions and answers to them are presented.

Chapter 2 Methodology

To be able to conduct research in a manner that is fruitful, it is important to provide a baseline as to why the research is being conducted, how it was conducted, and to what extent the end results coincide with the reason the research was conducted. Even though this is an oversimplification of a research method, it describes the overall idea of utilizing predefined methods when conducting research in any field. A main goal for any scientific research is to obtain new knowledge about the area of interest. This applies to every field where research is conducted, although the ways of getting there may vary based on the domain in which the research is conducted.

Research methodology is a systematic way to solve a problem. It is a science of studying how research is to be carried out. Essentially, the procedures by which researchers go about their work of describing, explaining and predicting phenomena are called research methodology. It is also defined as the study of methods by which knowledge is gained. Its aim is to give the work plan of research[RPV06].

Research methodologies aim to act as a framework to conduct research that is useful in some way, by helping the researcher ask the right questions before and while going through the research. There exist several methodologies and frameworks for this. In this thesis we have been taking use of the Design Science methodology[Wie14] in order to guide us during the course of the project.

Even though design science share similarities with the scientific method, design science is adapted to support fields like computer science, where a solution to problem may revolve designing and creating software or hardware to solve it. While natural sciences often aim to understand the world around us without changing it, areas like communication technology and computer science often involve addressing challenges by designing and implementing software applications or hardware appliances. Using the scientific method in such scenarios does not seem like a good fit because the premise of the research is fundamentally different. As the nature of this thesis involves a change in the real world by designing a and implementing a testbed to answer the presented research questions, it seems to be a better fit to utilize design science.

In the following sections, the engineering- and design cycle will be presented. How this approach was executed in relation to this thesis and testing will also be described. The theory behind this methodology is drawn from [Wie14].

2.1 The Engineering- and Design cycle

A design science project aims to first identify a problem, design a solution and then investigate how the solution can contribute to solving the identified problem[Wie14]. We define such a solution as a conjunction between an *artifact* and *treatment.* A treatment is the appliance of a created artifact to a given problem context. In the case of this thesis, the problem context is assessment of VANET protocols. The created artifact is a testbed that when applied to this context, aim to provide a treatment addressing the challenge of expensive and resource demanding field tests.

The design cycle mainly deals with problem investigation, design of a treatment, and validating the treatment. The process of validating a treatment involves predicting how the artifact would behave in its given context. In this thesis, this prediction is made through conducting simulated experiments of the testbed. To include the aspect of evaluation for the treatment in the real world, some additional steps are needed, namely *implementation* and *evaluation.* The engineering cycle is a larger cycle, which encompasses the steps of the design cycle, as well as these additional steps for evaluating the treatment. Presented in the next sections are descriptions of the steps in the engineering cycle and how they were performed in relation to this thesis. The illustration in Figure 2.1, taken from [Wie14] shows the steps of the design- and engineering cycles.

2.1.1 Problem Investigation

The problem investigation is the initial phase of any design science project and aims to identify a problem to be addressed in a certain context, without having designed any artifact or outlined requirements. This phase demands collecting information from the real world in order to substantiate the motivation for the treatment to be designed. In this thesis the problem investigation was done by first studying already published literature on the general domain of ITS and VANET. Then, we conducted a limited literature review on the topic of testing communication protocols for VANET, and identified the main challenges. The result of this process is presented in chapter 1.

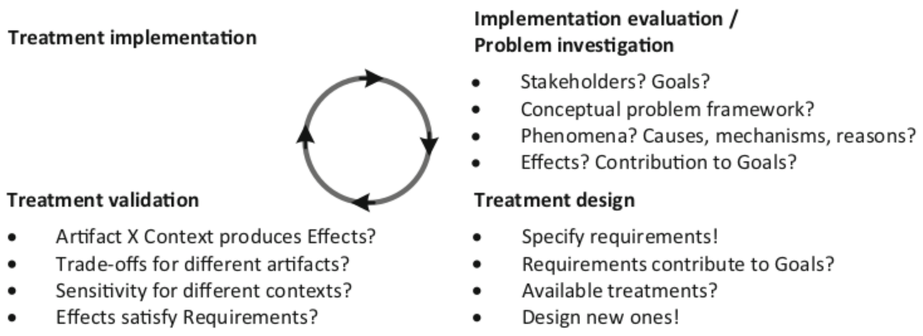

Treatment implementation
Implementation evaluation / Problem investigation
Stakeholders? Goals?
Conceptual problem framework?
Phenomena? Causes, mechanisms, reasons?
Effects? Contribution to Goals?
Treatment validation
Artifact X Context produces Effects?
Trade-offs for different artifacts?
Sensitivity for different contexts?
Effects satisfy Requirements?
Treatment design
Specify requirements!
Requirements contribute to Goals?
Available treatments?
Design new ones!

Figure 2.1: The design- and engineering cycle [Wie14]

We found that the amount of research around the general field of VANET is large and easily accessible. This also applies to the domain of testing protocols and applications for VANET protocols. However, we did not find many studies focusing on alternative ways to perform field tests. Therefore, we argue that the work done in this thesis is of a novel nature.

2.1.2 Treatment Design

A treatment in the context of design science is another, more precise formulation of a solution[Wie14]. The word *treatment* implies that there exists a problem that needs some form of solution. To successfully design an artifact that can be applied as a treatment to a problem, design science calls for specification of requirements.

List of Requirements

We define non-functional requirements for our testbed. When defining these requirements we take into account the properties of VANETs, UAVs, and the important aspects of cost- and resource reduction as identified in the problem investigation.

Firstly, VANETs may have a large number of vehicles connected at the same time. Thus, a testbed for VANET purposes needs to designed with scalability in mind, supporting a varying number of vehicles. In addition, we argue that an adaptable solution that supports different hardware and software for V2V communication is beneficial. With this, developers and researchers may implement their protocol in a way that they see fit. Even though there are few contenders with regards to the communication technology that will be adapted for V2V communication in the future, researchers may want to implement their protocol on other platforms for testing purposes. For simulated testing for instance, developers and researchers often need to implement their protocol specifically for the simulator that is used. Finally, since there are airborne UAVs involved in the proposed testbed, an important requirement is with regards to safety, that the solution is safe for people and property both involved and not involved in the testing.

On the basis of this we define a list nonfunctional requirements for the testbed:

- REQ1: The testbed must be scalable to varying number of vehicles.
- REQ2: The testbed should be adaptable to support protocol implementation on different types of hardware and software.
- REQ3: The testbed should be flexible in a way that facilitates further extension in the future to support aspects of VANET that were not included.
- REQ4: The testbed should be safe for people and property involved in the test as well as anyone not involved with testing.
- REQ5: The behavior of the testbed must be correct and predictable to avoid unwanted incidents.
- REQ6: The testbed should be relatively cheap and facilitate rapid setup and execution.

A testbed using real UAVs as well as simulated vehicles was designed based in these requirements. A detailed description of the design is presented in chapter 4. The design involves simulating vehicles in software, and duplicating their movement patterns in airborne UAVs in the real world. The testbed does not directly concern the communication done by the protocol under test, as hardware and software implementing the protocol should be attached to the drones.

To design the testbed, several aspects had to be addressed. First, we investigated different traffic simulators to be used for simulating vehicles. We ended up using SUMO [LBBW+18], a well established discrete, time driven mobility simulator. SUMO also supports an Application Programming Interface (API) for interacting with simulations, called TraCI, which was used to obtain information about the simulated vehicles.

We also had to choose which autopilot platform to install and utilize on the drones. The choice of autopilot affects the aerial behavior of the drone as well as capabilities and limitations. We ended up using ArduPilot as our autopilot of choice as it provides needed capabilities for our UAVs as well as relatively well documented Python-libraries for interacting with them.

Investigation into how to communicate with the drones while airborne was also conducted. We quickly realized that communication over telemetry, or radio, was the the only feasible option. A telemetry radio can be directly attached to our drones, and provide the needed range. Additionally, the Python-library used for interacting with the drones, DroneKit-py, supports telemetry link connections right out of the box. We ended up using 455Mhz telemetry radios for communication with the drones over the widely adopted MAVLink protocol.

Putting it all together, we designed a functional testbed capable of duplicating the movement of simulated vehicles in drones flying in the real world.

2.1.3 Treatment Validation

In the treatment validation phase, the goal is to be able to justify that the designed artifact will be able to treat the problem derived from the problem description[Wie14]. We use simulations as our validation model[Wie14].

To validate the performance of the testbed related to the requirements, a prototype was developed and tested through simulations. The setup and execution of the experiments are described in Chapter 5. The performance of the testbed is measured by looking at to what degree the requirements are fulfilled, and how well the drones were able to duplicate the behavior of simulated vehicles.

The actual treatment validation is presented in section 4.9. We iterate the requirements specified in section 2.1.2 and present our perspective on how the implemented testbed fulfill them. Out of specific approaches used in the validation we want to highlight Dynamic Time Warp (DTW), used to calculate the similarity between the trajectories of the simulated vehicles and the real life drones. DTW is presented in section 3.6. The results from the simulated experiments used as part of the validation are presented in chapter 6. The prototype was built using the the programming language Python 3.6.

2.1.4 Treatment Implementation

In design science terms, the *implementation* revolves transferring the the treatment to the original problem context[Wie14]. In this case the problem context is using drones for protocol testing in VANET.

Due to time constraints, we did not conduct a full treatment implementation of the testbed. In other words, we only conducted real life experiments to assess the performance of our testbed and not a routing- and data dissemination protocol for VANET. We did this by testing the same scenarios as in the simulated tests, but with real drones instead of simulated ones.

To be able to test a routing- or data dissemination protocol, we would have needed hardware on which a a protocol was implemented, and attached it to our drones. We were not able to obtain this. However, as will be explained in chapter 4, the step of attaching hardware that runs such a protocol is not a part of the implementation of the testbed itself, and we argue that this step is not crucial for evaluating performance in relation to the problem context.

The results gathered from the real world experiments are used to evaluate the performance of the testbed in real life scenarios. How these experiments were conducted and their results is presented in chapter 5 and chapter 6 respectively.

2.1.5 Treatment Evaluation

The treatment evaluation phase involves investigating how the implemented solution interact with the real-world problem context[Wie14].

After the implementation phase, we analyze the data gathered from the real world experiment and compare it to the simulated experiments done in the treatment validation phase. The evaluation step is integrated as a part of the discussion, in chapter 7. Here, we view the results in light of the problem context. In other words, we use these results to evaluate how we believe the testbed will perform in real life scenarios for testing VANET protocols. We also use these results to answer our research questions defined in section 1.3.

Chapter 3

Theoretical Background

This chapter aims to provide some background on different aspects related to this thesis. First, some more background on ITS, VANET and VANET-testing is provided. Then, we describe the hardware and software used in the development of the testbed, as well as some theory in relation to our methods used for data analysis.

3.1 ITS and VANET

3.1.1 VANET

An ad hoc network in general terms is made up of equal nodes communicating with each other in a decentralized manner. MANETs describe ad hoc networks where the nodes are mobile and communicate over wireless links. Characteristics of MANETs include dynamic network topology, higher loss rates than wired ad hoc networks, as well as the need for energy reservation due to nodes often running on batteries or other exhaustible energy supplies. [Sto02]

VANETs are a special category of MANETs. Instead of mobile phones, tablets or laptops, vehicles make up the nodes in the network. Even though VANETs and MANETs have some similar characteristics, there are some important differences. A summary of these differences are listed in Table 3.1, adapted from [KHK].

Due to these, and other notable differences between the two technologies, they also have different challenges and issues that need to be addressed. For instance, due to limited energy supply, a prominent challenge in MANETs is energy conservation of nodes [JJ11]. This issue is not as prominent in VANET as the energy supply is not limited to a battery or another exhaustible energy resource. Another example is the different characteristics of network topology in MANET and VANET. As the topology in VANETs changes more frequently and more rapidly compared to traditional MANETs, other challenges related to routing and data dissemination arise.[SA14]

Table 3.1 Characteristics of MANETs and VANETs [KHK]

Category	**MANET**	**VANET**
Mobility	Low(Walking speed)	High (Up to 200km/h)
Production cost	Cheap	Expensive
Change of network topology	Slow	Fast and frequent
Node life-span	Dependent of power resource	Not dependent on power resource
Reliability	Medium	Very high
Node movement pattern	Random	Systematic

Since the difference in characteristics between MANETs and VANETs are not considered in protocols designed for traditional MANETs, they cannot be directly applied to the VANET domain[KHK]. Consequently, new protocols and applications need to be designed and developed specifically for VANET. Figure 3.1, taken from [SMR+19], illustrates the VANET environment, including the two different types of communication.

3.1.2 V2V/V2I Communication

Two types of communication are defined in VANET networks; Vehicle-to-vehicle (V2V) and Vehicle-to-infrastructure (V2I)[MBOH14]. V2V communication describes direct, inter-vehicle communication in an ad-hoc way, through single- or multi-hop manner. V2V communication enables vehicles to directly exchange important data like traffic conditions and emergency messages among themselves without the need for any network infrastructure. V2I communication, on the other hand, describes the communication between vehicles and the roadside infrastructure. With this, vehicles can avail of the the already existing infrastructure to exchange messages with other vehicles over longer distances. V2I communication also facilitate connection to the internet. An On-Board Unit (OBU) is the name of the equipment that is installed in every vehicle and performs the actual VANET communication. The infrastructure nodes, usually fixed along the road are called Road-side Units (RSUs)[Ak12]. OBU especially, will be frequently referred to in the rest of this thesis.

3.1.3 VANET Testing

As mentioned in section 1.1 There are mainly two methods of assessing performance of applications and protocols developed for VANET, namely simulated experiments and field tests[LLZ+15]. To execute simulated experiments, two core modules are required, a traffic mobility simulator and a network simulator. The mobility simulator deals with realistic simulation of vehicle mobility. One of the more notable traffic

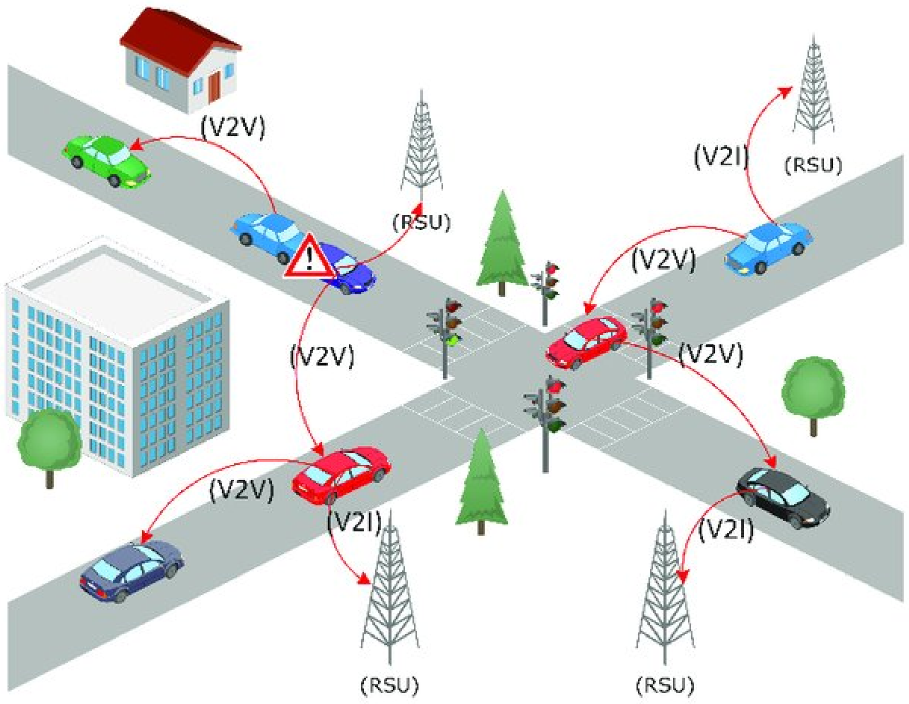


Figure 3.1: VANET Environment and Architecture[SMR+19].

simulators is Eclipse SUMO[LBBW+18]. In addition, to incorporate realistic signal propagation and network behavior, a network simulator is needed. In this domain, notable mentions are OMNet++[Var10], ns-2[ns2] and NS-3[ns3]. Combining these two types of simulators constitutes the main method of assessing VANET applications through a fully simulated environment.

Even though simulators over the years have become more and more realistic in terms of accurately representing environments in the real world, large scale field experiments with real vehicles still stand out as the most superior way of assessing performance of VANET applications and protocols[WSGY19]. In a field test, vehicles may be rented or bought. Each vehicle is then equipped with an OBU necessary to perform the communication in a VANET. Additionally, the vehicles are transported to a dedicated test site where experiments can be conducted. Since field tests incorporate real vehicles and standard V2V communication equipment, it is the most realistic form of VANET-protocol testing.

In addition to the above mentioned methodologies, another form of testing, a hybrid testing scheme may also be used. A hybrid testing scheme includes both simulated and real life aspects, and combines them into providing performance tests that aim to be closer to real life than pure simulations, but still lack the complete representation of real life environments like field experiments. Hardware-in-the-loop (HIL) testing is a way of performing hybrid testing of a VANET application, in which real hardware is integrated with simulated aspects.

The testbed designed in this thesis is of a hybrid nature in that the vehicle behavior is simulated. However, the vehicle behavior is also performed in the real world, but by drones instead of cars.

3.2 Hardware

3.2.1 Drone

Drones, or Uncrewed Aerial Vehicles (UAV)s, are aerial vehicles designed to be flown without a pilot onboard the aircraft. Originally designed for military purposes, drones have today found their way into the consumer market and are used for multiple purposes like aerial photography, mapping and area surveying[WFZ+16].

There are several types of drones[KJT21]. The most notable difference is that of fixed-wing drones and single- or multi-rotor drones. Fixed wing drones are similar to airplanes in that they are able to ride across the air without using energy, however they are not able to maintain a stationary position. Single- or multi-rotor drones however, can hold a stationary position, but uses energy to stay afloat in the air.

Drones usually contain some form of embedded hardware to control its motion, a Flight Controller (FC)[KJT21]. In addition, drones may have a companion computer attached. A companion computer is usually a small computer able to run more complex code or even operating systems. This enables them to have on board intelligence that can be used to achieve higher levels of automation without control from a pilot or Ground Control Station (GCS). For the rest of this thesis, we will use the terms *drone* and *UAV* interchangably. The drones used for experiments with the testbed created for this thesis are multi-copter type UAVs, specifically the Intel Aero RTF. Full specifications for this drone can be found on Intel's websites[1]

3.2.2 Telemetry Radio

In this project, we utilize transceiver telemetry radios for drone communication. These are small and inexpensive, and are capable of communication for ranges up to 300m. The specific brand of telemetry radio used in this thesis operate at 433Mhz[2],

[1]Intel Aero RTF drone specifications: https://www.intel.com/content/www/us/en/support/articles/000023271/dron
drones.html

[2]Telemetry radios: http://www.holybro.com/product/transceiver-telemetry-radio-v3-915mhz/

and are designed specifically to perform well using the MAVLink[3] protocol, presented later.

3.2.3 Flight Controller

The flight controller (FC) is an essential part of any drone. The FC is an embedded computer that takes various sensor data as input, and adapts motors accordingly. Examples of such sensors are Global Positioning System (GPS), accelerometer, gyroscope as well as pilot input through a wireless transmitter or telemetry radio. The FC gathers the data generated from these sources and adapts the power level of each rotor accordingly[KJT21].

In this thesis, a proprietary FC that comes with the Intel Aero RTF from the factory is used. This particular FC support installation of two popular autopilot softwares, ArduPilot and PX4.

3.3 Software

3.3.1 SUMO

Simulation of Urban MObility (SUMO)[LBBW⁺18] is a well established, time-driven and time discrete mobility simulator. The open source project is often used to simulate traffic scenarios in a multitude of applications, including testing of VANET applications and protocols. SUMO is also a microscopic simulator, which means that all simulated vehicles are controller separately. Each vehicle have their own physical properties and destination, and abide by defined traffic rules when interacting with other vehicles. The speed of each vehicles is determined by the individual properties of each vehicle as well as the traffic rules, i.e., the traffic model, that is implemented.

A simulation run in SUMO is time-driven, meaning that properties of each vehicle is calculated in *timesteps*. At each timestep, calculations for determining speed, position and other properties is made for each individual vehicle. A very useful property of SUMO is that it allows for granular access to information about each simulated vehicle after each timestep, through a Python interface called TraCI, presented later.

In addition to this, SUMO comes with many other tools for different purposes. One of these tools is called *netedit*. Together, netedit and SUMO can can be used to design, create and run a large variety of traffic simulations. Additionally, combining SUMO with a network simulator, like OMNeT++ or NS-2, creates a powerful combination of tools to mimic the mobility of vehicles and network behavior of

[3]MAVLink documentation: https://mavlink.io/

VANET. In the testbed developed in this thesis, SUMO is used to simulate cars in virtual traffic scenarios.

3.3.2 netedit

As mentioned, netedit is part of the SUMO library and is a visual program for creating virtual traffic *network files*, meaning file formats describing traffic scenarios. These can either be created manually from scratch, or imported and exported using the OpenStreetMap[Ope17] format, an open, standard format for describing traffic networks. netedit also support the addition of vehicles, traffic signs, crosswalks and other aspects of traffic into these files, which is taken into account by simulated vehicles when the traffic scenario simulated in for instance SUMO. netedit is flexible and supports multiple aspects of traffic, including varying driver behavior and different vehicle types. In this thesis, netedit is used to create traffic scenarios for our experiments.

To give a better understanding of later diagrams and descriptions, we provide a list of definitions related to traffic- and vehicle simulation, and network files. Some of these terms are used later in this thesis. Definitions marked with $*$ can be found at SUMOs glossary page[4]. The rest are created by us.

*edge: A single-directed street connection between two points (junctions/nodes). An edge contains at least one lane.

**junction*: The place an edge begins or ends at (same as node)

**node*: A node (junction) is a single point were at least one edge (road) starts or ends

**(street) network*: In our terms, a network is the combination of junctions (nodes) and edges (streets)

network file: a *network* stored in a file format readable by SUMO to be used for simulations

network geolocating: This describes the process of projecting a *network file* based on cartesian coordinates into a georeferenced network, where each cartesian point is converted into coordinates based on the WGS84 ellipsoid, which is the projection used in this thesis. In simpler, more informal terms, *network gelocating* can be described as converting a network file based on cartesian coordinates into a network file based on GPS-coordinates.

[4]SUMO glossary: https://sumo.dlr.de/docs/Other/Glossary.html

demand elements: These elements describe the virtual vehicles added to a network file, and their behavioral model.

mobility traces: A term for describing the movement of vehicles. mobility traces describe the location of a vehicle at a given time.

3.3.3 Traffic Control Interface (TraCI)

TraCI[5] is included in the SUMO package, and is a Python-interface that provides access to simulations run in SUMO. TraCI enables a flexible and dynamic simulation environment and can be used to modify different parts of a simulation while it is running. In this project, TraCI is used as part of an interface between simulated vehicles and drones, collecting mobility traces from the simulated vehicles and controlling airborne UAVs. TraCI provides granular information retrieval and modification for each simulated vehicle as well as for the simulation environment as a whole.

3.3.4 ArduPilot SITL

ArduPilot[ard] is a open autopilot software for drones and other autonomous systems, and is the autopilot of choice in this thesis.

ArduPilot provides a platform for simulating fully operational drones, and includes support for both fixed-wing and single- and multicopter drones, called ArduPilot SITL[6]. A Flight Dynamic Model (FDM) is used to simulate the physics involved in the movement of a drone[7]. The combination of the FDM and simulation of ArduPilot firmware is referred to as software in the loop (SITL) and provides a powerful solution for testing application and solutions in a simulated scenario before executing real world tests. Performing tests with simulated drones played an important part when implementing the testbed created in this thesis and was used to verify correct operation and to reduce the risks of crashes in the real world.

3.4 Drone Control

There are several aspects involved in controlling airborne UAVs. Drones can be controlled in a variety of ways. These include manual control with handheld transmitters, over Wi-Fi or over telemetry using telemetry radios. We present the technologies and solutions used in this thesis for drone control and monitoring.

[5]TraCI: https://sumo.dlr.de/docs/TraCI.html
[6]ArduPilot SITL: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
[7]Flight Dynamics Model: https://ardupilot.org/dev/docs/simulation-2.html

3.4.1 Ground Control Station (GCS)

A GCS enables remote control of aerial vehicles from a computer on the ground. Usually a software application, a GCS enable bidirectional communication with a UAV while it is flying. This communication link can be used both for monitoring purposes, as well as for issuing commands. GCSs may also include failsafes and other safety features. Figure 3.2 illustrates the look and feel of QGroundControl, the GCS used in this thesis. QGroundControl is used mainly to monitor drone mobility, like speed, location and altitude, but also as a safety measure.

3.4.2 MAVLink

MAVLink is a widespread protocol for bidirectional communication between drones and GCS. MAVLink can also be used for communication between internal components inside the UAV itself. As MAVLink is a protocol standard, several libraries have been created to implement it, in multiple programming languages. In this project, a MAVLink Python-library called DroneKit-Python is used for the communication between drones and the software developed for our testbed. DroneKit-Py is built on top of pymavlink, which is a low level library for communicating with MAVLink.

3.4.3 DroneKit-Python

DroneKit is a wrapper for a low level MAVLink library called pymavlink. We found that in our opinion, DroneKit have better documentation compared to pymavlink. DroneKit provides a Python API towards the MAVLink protocol called DroneKit-Python which provides a relatively simple way of communication with drones over MAVLink, through a number of channels, inlcuding Wi-Fi, USB, and telemetry radio. For our testbed, DroneKit is utilized to control the speed, motion and position of drones while airborne.

3.4.4 GNSS

Global Navigation Satellite Systems (GNSS) is a collective term used to describe navigation and position systems that utilize satellites. To be able to navigate based on Geographic Coordinate Systems, drones need to be equipped with hardware that supports GNSS technology. There are four major players in the GNSS domain. These are GPS, GLONASS, BeiDu and Galileo. The testbed created in this thesis, requires the drones to be equipped with such a that allows for easy access to coordinates as well as being able to navigate based on WGS84 coordinate system, a mathematical representation of the earth used for navigation purposes. In this thesis, the drones used in the experiments are equipped with GPS that is used for localization and navigation purposes. For experiments with simulated drones, each drone has a virtual GPS-module.

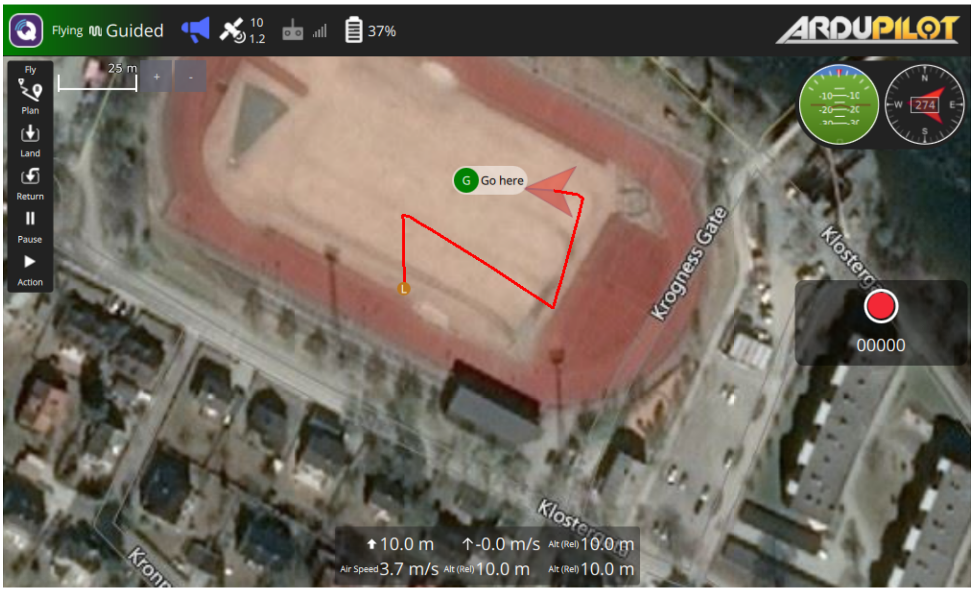


Figure 3.2: QGroundControl connected to a single airborne UAV

3.4.5 Flight Modes

Flight modes play an important role in controlling drones. Depending on what flight mode is currently enabled on a drone, the FC and autopilot software may adapt the behavior of the UAV. Flight modes can provide the pilot with assisted control in a variety of ways. Pilot input can come from either a handheld RC-transmitter or a GCS. In other words, a *pilot* is not necessarily a person manually controlling the drones, but can also be automatic commands issued programmatically through scripts, or a GCS such as QGroundControl. Flight modes can also be changed dynamically, meaning that a pilot can swap flight modes at any time while the drone is airborne.

Below is a list that describes some of the important flight modes that were used in the course of this project, together with their functionalities. Different FC software support different flight modes. As mentioned earlier, in this thesis we are using the FC software and autopilot system called ArduPilot.

– **Return To Launch (RTL)**: When a drone enters this mode, it will return to the home location, and land. the home location is defined as the location where the drone was last armed to fly.

– **Auto**: When in Auto mode, the drone will execute a predefined mission that is uploaded to the drone before flight. A predefined mission may contain navigation commands to fly to a specific waypoint, or other commands that does not affect the location of the vehicles i.e the camera shutter to take images.

- **Guided**: Guided mode is not a traditional flight mode and is specific to ArduPilot. This mode is designed to be used by a GCS to dynamically specify a target location over telemetry radio. Once a target location is reached, the drone will hover until the next waypoint is sent received from the GCS.
- **Loiter**: In Loiter mode, the drone will attempt to maintain the current speed, heading and altitude, leaving only the drone movement in the X/Y plane to be controlled by a pilot.
- **Brake**: When a drone is instructed to enter brake mode, it will attempt to stop movement and hold position in the air. As no pilot input is accepted in this mode, the flight mode needs to be changed after entering Brake mode to resume flying. Brake mode relies on a functional Global Navigation Satellite Systems (GNSS).

In the testbed created in this thesis, we have utilized Guided mode as our main mode of control. Compared to Auto, Guided mode allows for more interactive and dynamic control of a multi-copter. Manual intervention using Brake mode and Loiter mode was included as a backup in case of emergency. These flight modes were also used while practicing to fly the drones manually with a handheld transmitter.

3.5 Time Series

A time-series data set consist of a sequence of observations measured over time, and is the simplest form of temporal data[HKP12][GD01]. The frequency of measurements can be for instance every minute, hour, day, month or even decades. A time series can be expressed on the form $x_i(t); [i = 1, \cdots, n; t = 1, \cdots, m]$ [YS04]. Here, i denotes the index or indices of observation(s) at time step t. It is called a *univariate* time series when n is equal to 1 and a *multivariate* time series when n $>$ 1[YS04]. In this thesis we collect multivariate time series and perform similarity measurments between them. Each measurement, done at regular time intervals, contains location data on the coordinate form (*latitude*,*longitude*), i.e., $n = 2$.

3.6 Dynamic Time Warp (DTW)

To perform similarity measurements between multivariate time-series data sets in this thesis, we will utilize DTW. DTW is a spatio-temporal, nonmetric similarity measure for time series data[MSMEB15]. DTW is also a popular choice for measuring similarity between trajectories[TD15]. A strength of DTW is the ability to do

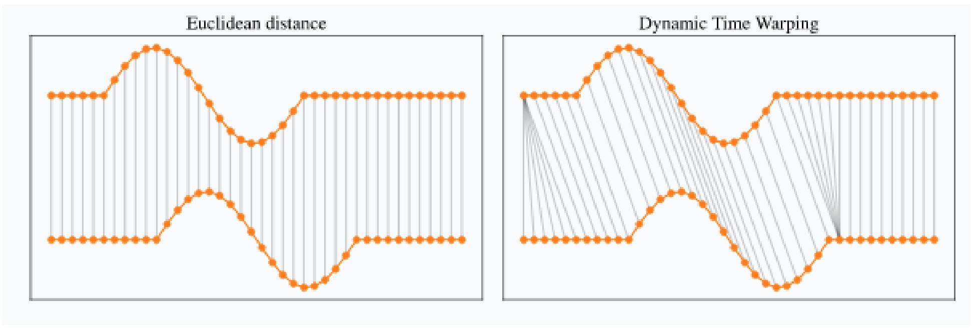


Figure 3.3: Comparison of distance measure between Euclidean distance and DTW. The two time series have similar measurements but are not aligned on the time axis. The illustration can be found in [Tav].

temporal alignment of time series. This means that it allows stretching a shrinking of time series in the time axis to create a better alignment, which can be used for measuring similarity. Figure 3.3, taken from [Tav], shows an example in which two timeseries have similar measurements but are shifted in the time axis. Regular euclidean distance fails to take these shifts into account.

Having two time-series P and Q of length m and k, a distance function is used to construct a m x k matrix where the (i, j) element represent the distance between points P_i ad Q_j. Using this matrix, the optimal alignment, i.e., optimal *warping path*, can be constructed by minimizing the total distance between all points. The DTW measure is the total of all distance measures contained in this path[TD15].

In the case of this thesis, we will use DTW as a way of measuring the similarity between trajectories. Specifically, how well a drone is able to duplicate the path of a simulated vehicle using our testbed. In our case, each point P_i and Q_j in our time series data is a geographical coordinate, a tuple on the form $(longitude, latitude)$. Using regular euclidean distance between geographical coordinates does not work and will result in incorrect distance measurements. We therefore use the Haversine formula as the distance function when calculating the distances between measurements in the time series.

3.7 Haversine Formula

The Haversine formula is used to calculate the great-circle distance between two geographical points on $(longitude, latitude)$ format[Nic13]. The formula can be expressed as shown in Equation 3.1.

$$d = 2r \sin^{-1}\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\psi_2 - \psi_1}{2}\right)}\right) \tag{3.1}$$

d is the distance between two geographical points, r is the radius of the earth. ϕ_n and ψ_n respectively represent the longitude and latitude of the two points.

3.8 Regulations

Rules and regulations are an important aspects to consider when flying drones. Regulations for automatic and autonomous drone flight may vary from location to location. the 1st of January 2021, new laws and regulations were passed in Europe, clarifying and regulating different aspect of drone flight[8]. Laws and regulation may impact the legality of executing experiments with the testbed presented in this thesis. For the experiments conducted in this project, the author obtained online training and a certificate from the Norwegian Civil Aviation Authority.

[8]New European drone laws: https://luftfartstilsynet.no/droner/nytt-eu-regelverk/

Chapter 4 Implementation

In this chapter we first present the design for the created testbed. Then, we attempt to validate the design in relation to the problem context.

4.1 Architecture

4.1.1 Overview

We will now first present a broad overview of the testbed before describing some aspects in more detail.

At the highest level, the functionality of the testbed can be summarized into two segments. Vehicles are simulated using SUMO and data from this simulation is sent to a software interface. After performing necessary calculations, this interface uses the retrieved data to control airborne UAVs to duplicate the movement of the simulated vehicles. When executing real world experiments, this interface and the SUMO-simulation run on a computer that is situated on the ground at the flight-area. From now on we will refer to this computer as the Ground Based Computer (GBC). Figure 4.1 illustrate a high level overview of the architecture.

Since drones are also vehicles, we clarify the important distinction between real drones and the simulated vehicles in SUMO. From now on will will refer to simulated vehicles (SVs), and drones or UAVs. We emphasize that we will also mention drones that are fully simulated, that are used for testing purposes.

The *interface*, depicted yellow in Figure 4.1, is a Python-program that performs the necessary logic and communication to duplicate the movement of SVs in airborne drones. It integrates TraCI and the Dronekit-py library to facilitate indirect communication between the SUMO-simulation, depicted blue in the figure, and the airborne UAVs, depicted red. TraCI is used to retrieve relevant information about each SV, like location, speed, and direction. DroneKit is used to retrieve information

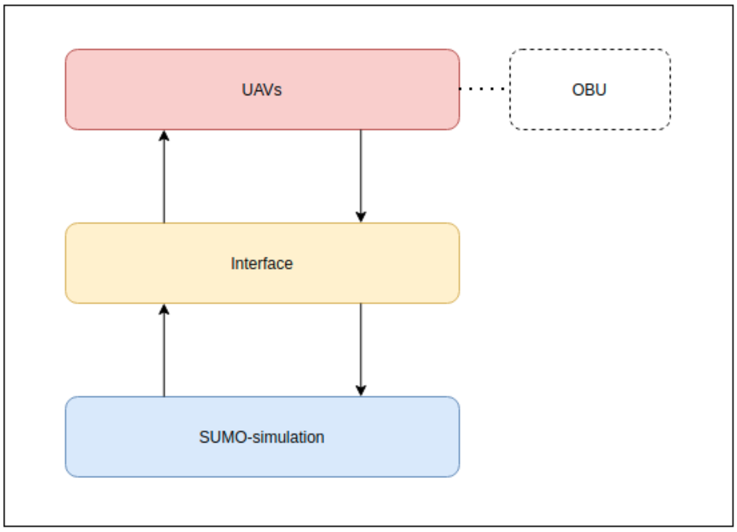


Figure 4.1: Overview of testbed architecture

and send navigation commands to each airborne UAV, based on the data collected from SUMO.

An important note is that the testbed created in this thesis, does not directly concern the aspects revolving V2V communication. The V2V communication component is left to the people assessing the performance of a VANET-protocol to implement. This allows for flexibility in choice of hardware and software when implementing a protocol to be tested. In other words, the OBU is not fixed and can be implemented as researchers see fit. The OBU (described in subsection 3.1.2) that is implementing the protocol under test should be attached to each individual UAV while running a test, depicted by the dashed lines in Figure 4.1. For instance, a routing- or data dissemination protocol can be implemented on smart phones that can be attached to the drones and communicate between each other.

Each component depicted in Figure 4.1 has a main role to fulfill to conduct a protocol assessment. These roles are summarized below, in Table 4.1.

Table 4.1 Main role of testbed componets

Component	Role
SUMO	Simulate mobility of vehicles, taking a network file as input
Interface	Query SUMO simulation for vehicle information. Use this information to control UAVs. The interface can also be used for data collection.
OBU	Attached to each drone, responsible for executing communication for protocol under test, as well as data collection for the protocol under test.
UAV	Receive and follow navigation commands from the interface.

4.1.2 Detailed Description

In this section we describe the tasks of each component in greater detail, and further break down the structure to give a better description of the architecture. Illustrated in Figure 4.2 is a more granular description of the testbed architecture.

(1) Ground based computer:

While conducting a test, a GBC runs the vehicle simulation in SUMO as well as the interface that connects the simulation with the UAVs. In case of a tests where the drones also are simulated, the GBC may also run the drone simulation software. In the case of a real world test, the ground computer is situated on the ground at the flight location, within communication range of the drones, using telemetry radios.

(2) SUMO

As described in Section 3.3.1, SUMO is a mobility simulator used to simulate vehicles and driver behavior in a virtual environment. In this testbed, SUMO takes a georeferenced network file as input and generates mobility traces for vehicles inside it. As SUMO is time driven, it operates in time steps. At each time step during the simulation, SUMO generates data about the current state of the simulation as well as for each individual vehicle, like position, speed and heading. Between every time step, this data can be accessed through TraCI, which is utilized in the interface (3) to obtain relevant data.

(3) Interface

The code developed in this thesis resides in the interface. This is the core component of the testbed and has several functions. As stated in Table 4.1, the main role of the interface is to control the UAVs so that they duplicate the movement of SVs. To do this, it gathers data from a simulation, processes it, does necessary calculations,

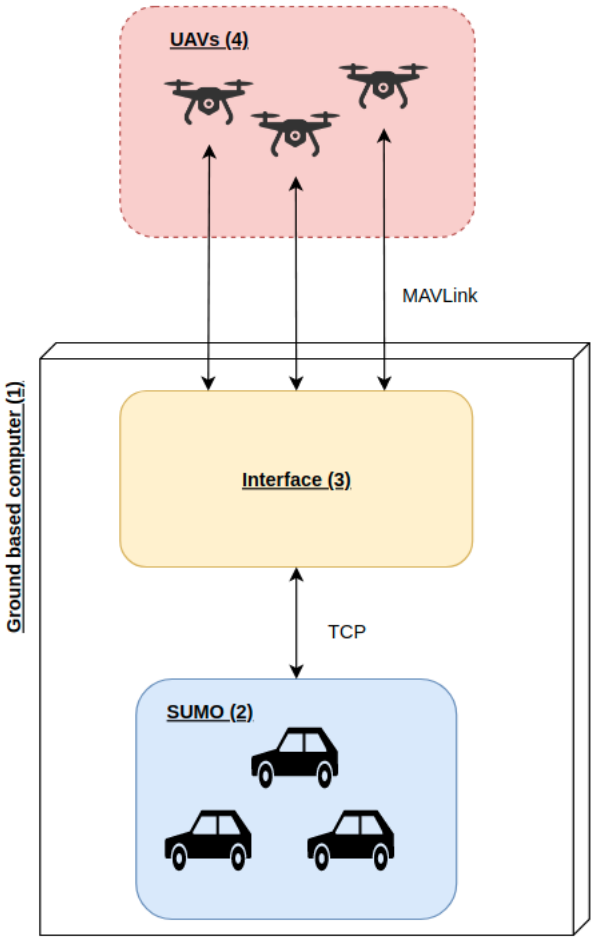


Figure 4.2: Descriptive architecture overview

and passes navigation commands to each UAV based on this data. Regarding communication with the UAVs, the interface utilizes DroneKit-Py, as described in subsection 3.4.3. The link used for UAV-communication can be seamlessly changed. This allows the interface to support among other things, communication with real drones over telemetry radio, as well as simulated drones through a network interface.

To present the functionality and inner workings of the interface, a more granular description of this component will be presented in the next section.

(4) UAVs

The UAVs receive control commands from the interface and passes them to the FC, which in turn adapts the throttle of each rotor to fulfill the command. An example of a control commands can be to tell the UAVs to travel to a certain geographical location.

For real world tests of a VANET-protocol, OBUs performing the communication with the protocol need to be attached to the drones.

Not depicted in this figure is the part that is used to monitor the state of the drones. This is done using QGroundControl which continuously queries the drones for information like location, speed, altitude and battery level. As this a is convenient, but optional and not part of the testbed itself, it is not included in the figure.

4.1.3 Interface

As the interface is the most central part of the testbed, we present its most important building blocks.

traci-script: The script that runs the vehicle simulation is adapted from the starting-examples provided by SUMO[1]. Its only task is to increment the simulation step and pass the data generated by SUMO for the current timestep to the Handler. At each simulation time step, SUMO calculates among other things the position, speed and direction of each SV. The TraCI-script passes all of this information to the Handler. The choice of separating this script and the rest of the interface, was to make this script easily modifiable. Developers can edit this script to modify vehicles behavior during simulation without interfering with the core logic of the testbed. The barebones TraCI-script used in the experiments conducted in this thesis can be found in appendix D.3.

Handler: The Handler keeps track of the SVs and drones that are active, in other words SVs that are still running in SUMO and drones that are still flying. In addition, the Handler is responsible for creating pairings between a UAV and a SV in a 1-to-1 relationship. Finally, the Handler uses information gathered from the SVs to queue navigation commands for the UAV they are paired with. The *Handler* is initialized by the TraCI-script and is provided with knowledge of a *traci*-object. The traci-object contains all information about the current state of the simulation, and the Handler receives updated information at each simulation time step. The Handler queries this object to extract the needed information about the simulation environment as well as for each individual SV. The code for the Handler-class can be found in appendix D.2.

Vehicle: Each instantiation of the Vehicle-class represents a pairing between a UAV and a SV. The paring is a 1-to-1 relationship, so each SV is only paired with

[1]TraCI examples: https://sumo.dlr.de/docs/TraCI.html

one drone. Vehicle-objects are instantiated by the Handler whenever a new SV begins its journey in SUMO. Every Vehicle-object contains a queue that represents positions that the corresponding SV has visited in SUMO. A Vehicle-object uses this queue to send sequential navigation commands to its drone, so that when there are no more elements in the queue, the drone has visited the same positions as the corresponding SV. The Handler is responsible for queuing correct positions for each Vehicle-object. The code for the Vehicle-class can be found in appendix D.1.

The Vehicle-objects have a bidirectional communication link with their respective drone and uses this link to send navigation commands as well as querying the UAV for necessary information. When a SV finishes its route in SUMO, the Vehicle-object responsible for that particular SV makes sure that the corresponding UAV finishes its route and commands it to enter RTL-mode, before concluding execution.

Figure 4.3 illustrates an example of a state of the interface where three vehicles are simulated in SUMO. For every vehicle, the Handler creates a Vehicle-object that represents the pairing between each of these vehicles with a UAV. The uni- and bidirectional arrows represent the direction in which the internal information flows inside the interface.

4.2 Diagrams and Data Flow

To further illustrate some of the behavior of the testbed, we present some flow diagrams. First we present an activity diagram showing how a typical protocol test would be conducted with the testbed. Then we present two sequence diagrams showing the behavior of the interface.

4.2.1 Activity Diagram

For using the testbed, some preliminary steps are needed. Presented in Figure 4.4 is an activity diagram that shows the main steps in order to perform a test. The diagram presents the different steps needed for simulated tests and for real life executions, and are explained in the next paragraphs.

A user starts by designing relevant traffic scenarios for the protocol under test, for instance by using netedit, presented in subsection 3.3.2. Relevant traffic scenarios may vary depending on what protocol is tested. For instance, for data dissemination protocols designed for rural areas, high speeds and few vehicles may be a important aspect of a traffic scenario. For urban areas and in big cities, the opposite may be the case.

After designing relevant traffic scenarios, they need to be transferred to a SUMO network file, which SUMO uses to execute simulations. Several tools can be used

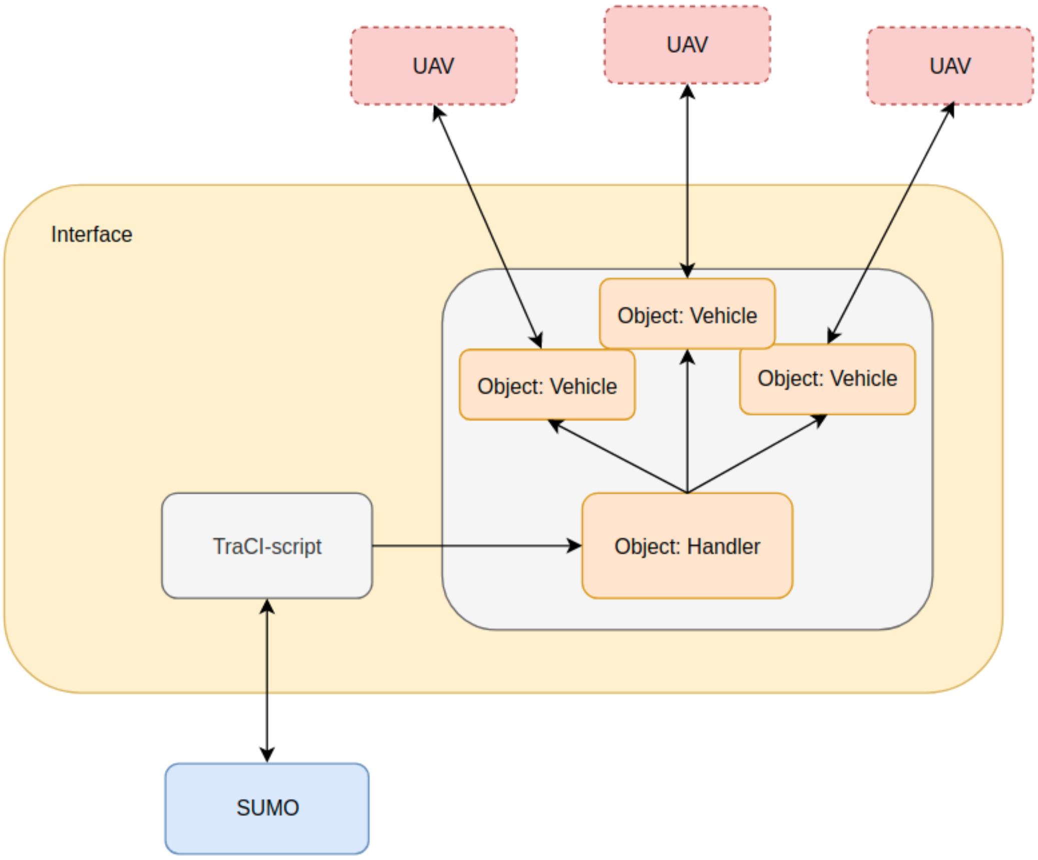


Figure 4.3: Components and communication flow in the interface.

for this, as long as a SUMO network file is generated in the end. SUMO provides a tool called *netconvert*[2] which has the ability to convert a number of file types used for representing road networks to the correct SUMO-network format. The network file also has to be georeferenced in order to be used in the testbed. To the best of our knowledge, there is no simple way of georeferencing networks files for SUMO that are created manually. For this reason, we have adapted and extended the functionality of a script included in SUMO that is originally used for fetching background images of an already georeferenced network file, called tileGet.py[3]. We extended the functionality of this script to include the ability of georeferencing a network file by only taking a origin coordinate and the UTM zone[4] as parameters. We named this script geoLocate.py. Additional details are presented in subsection 4.6.1.

[2]https://sumo.dlr.de/docs/netconvert.html
[3]https://sumo.dlr.de/docs/Tools/Misc.htmltilegetpy
[4]https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate$_system$

The decision revolving what kind of test should be executed affects the activity flow drastically. The main argument for conducting simulated testing before performing real life tests is that it can be used as a security measure to reveal any discrepancies in the behavior of the UAVs before conducting real world test. For real life tests, some additional steps are needed. Worth mentioning is the calibration of UAV sensors like GNSS, compass, gyroscope, accelerometer and other attached sensors.

Attaching OBUs to UAVs is illustrated as an optional step. Even though the intention of the testbed is to test protocols implemented on an OBU, it may also be useful to only test the movement of the drones, as a preparation step for instance. For that reason, the possibility to omit the step of attaching OBUs is included.

Finally, UAV parameters must be set. The testbed provides a configuration file called *drones.conf*, where a user must specify the altitude for each drone, as well as how the interface will connect to each drone. When these parameters are set, the testbed can be executed. An example of this configuration file is found in appendix D.4

4.3 Behavioral Diagrams

To highlight the most important behavior of the testbed, we present some behavioral diagrams from different phases of the execution. First we present the initialization phase were vehicles and drones are paired and initialized. We then present the main phase of the execution which includes the interface controlling the drones based on the data from the SVs. This phase also includes the finalizing and landing all drones back on the ground.

4.3.1 Initialization

We define the initialization phase as preliminary steps done by the testbed before and up until the point where UAVs starts to duplicate the path of their corresponding SV. This includes starting the SUMO-simulation, creating pairings between UAVs and SVs as well as UAV takeoff and navigation to their respective starting points. No drone start to duplicate the path of their SV until all active UAVs have arrived at their starting point, assuming all vehicles are started at the same simulation step.

A sequence diagram was choosen for illustrating the behavior of the interface during the initialization step. This diagram is presented in Figure 4.5. Despite being a simplified diagram, it includes the most important aspects of the general program flow. The example in the diagram is that of a scenario with two SVs and two drones.

The entire initialization phase occurs between the first and second simulation time steps. For simplicity and space constraits, the script running TraCI and the

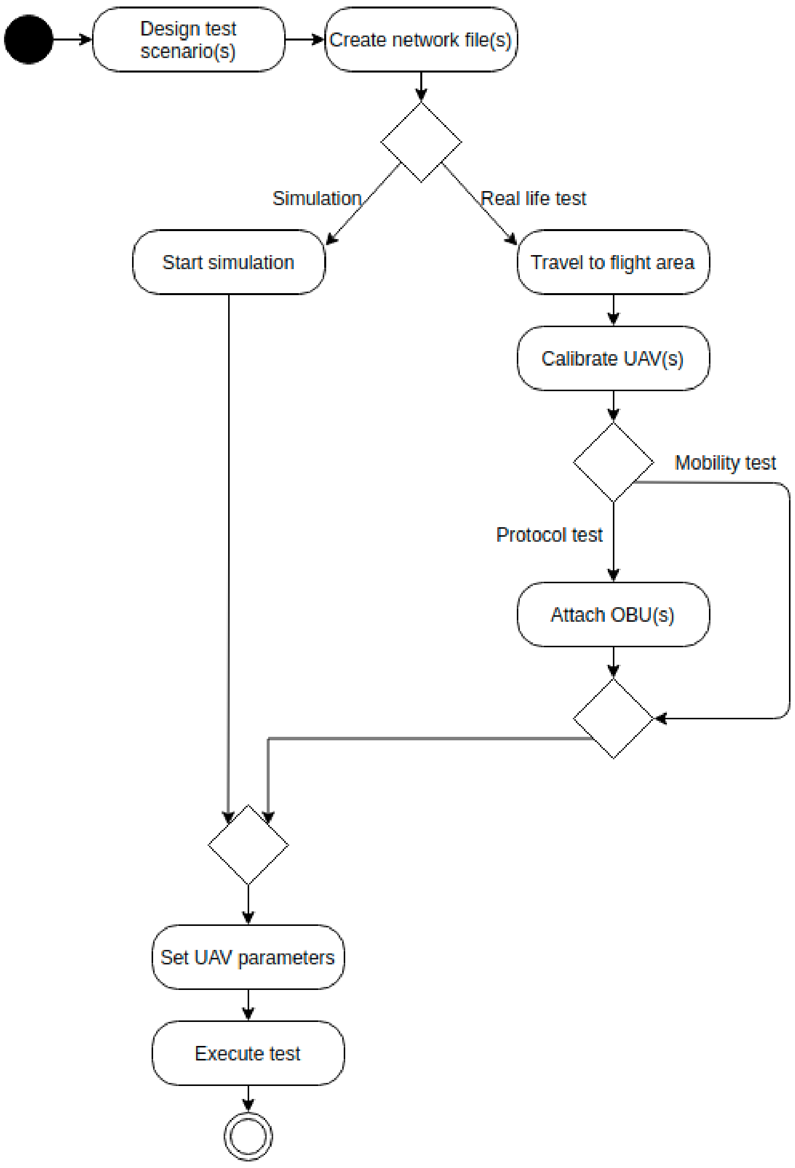


Figure 4.4: Activity diagram for simulated and real life tests

instantiation of the Handler is not included in the diagram. The black circle depicting the entry point, illustrates the point where the TraCI-script sends the traci-object to the Handler. As a reminder, the traci-object acts like a snapshot of the SUMO simulation at every timestep, and encompasses as mentioned earlier, information about every SV as well as the state of the simulation as a whole. The Handler identifies that two SVs were added in SUMO and instantiates a Vehicle-object for each SV. The Vehicle-objects are instantiated with a connection to a drone as well as knowledge about which simulated vehicle it corresponds to. Each Vehicle-object contains a FIFO-queue to which the Handler can add waypoints, reflecting the location of the vehicles in SUMO.

When instantiated, the Vehicle-objects immediately starts an interal thread which continuously monitors this queue for new waypoints. In the initialization phase, each thread only waits for the first element to be added to its queue, at which the threads will command their respective drone to go to the first location, the starting point. When all drones have reached their starting points, we move on to the main part of the execution.

4.3.2 Execution

After the initialization phase, and all UAVs have reached their starting point, the main execution begins. During this phase, the drones actively duplicate the movement of their respective SVs. To execute this, two processes run in parallel. The first process involves the Handler and the Vehicle-objects; At each simulation time step, the Handler is passed the traci-object containing the current state of the simulated scenario. The Handler extracts position information about each SV and sends it to their respective Vehicle-object. Each Vehicle-object stores these positions in the earlier mentioned queue, which is used by the process running in parallel.

As each Vehicle-object instantiates one thread running in the background, we specify that this second, parallel process includes the execution of all threads spawned by Vehicle-objects. They all run independently of each other and each thread is connected to a single Vehicle-object. The task of these threads is to continuously get the next element in their position-queue, create a MAVLink packet and send it to the UAV as a navigation command. After sending the command, a thread will wait for confirmation that the drone has reached the point. Then, it will get the next position from the queue and send it to the UAV. This happens continuously during the execution phase. Figure 4.6 show the overall data flow during this phase. Not all waypoits are sent to the drones during execution. Why this is and how it was done is described in subsection 4.6.2.

When a thread has no more positions in the queue, and the SUMO-simulation is finished, it will issue a command to put its drone in RTL mode, which tells the

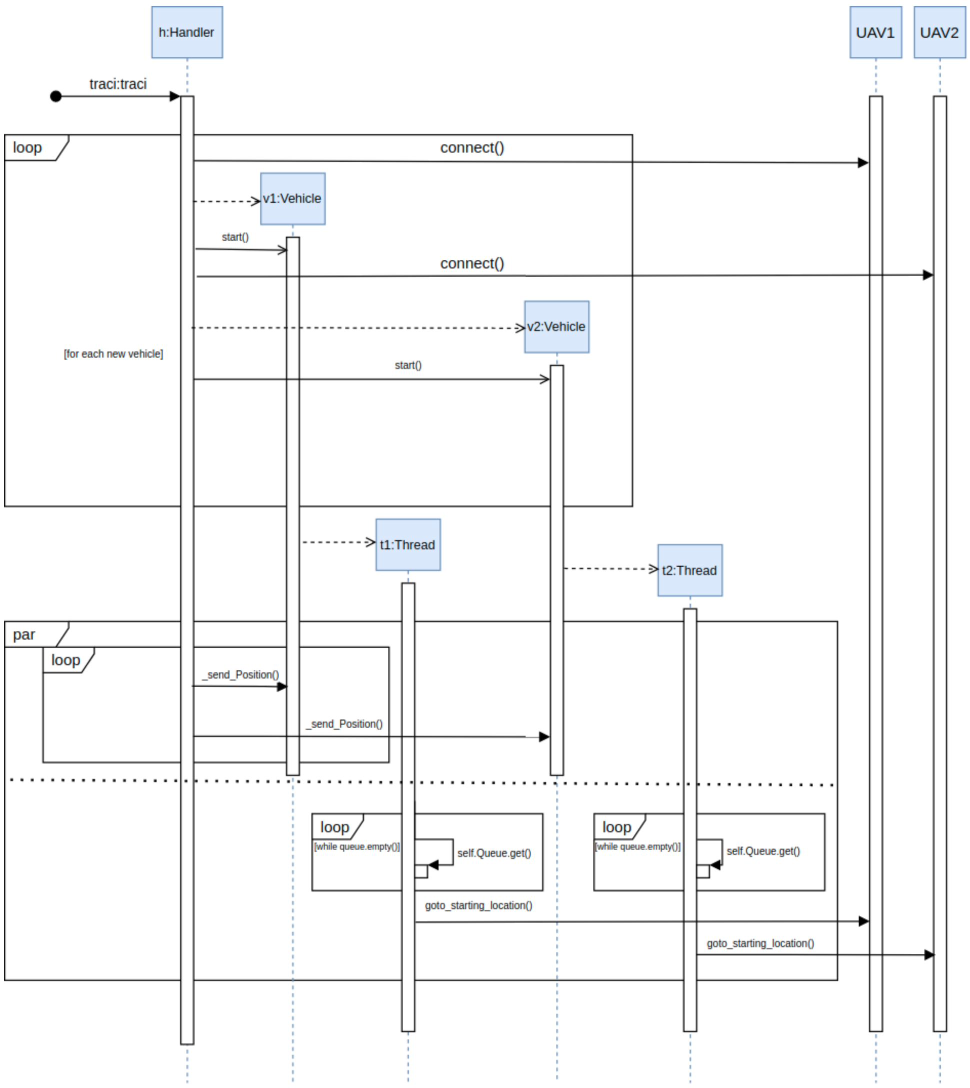


Figure 4.5: Simplified sequence diagram of initialization phase

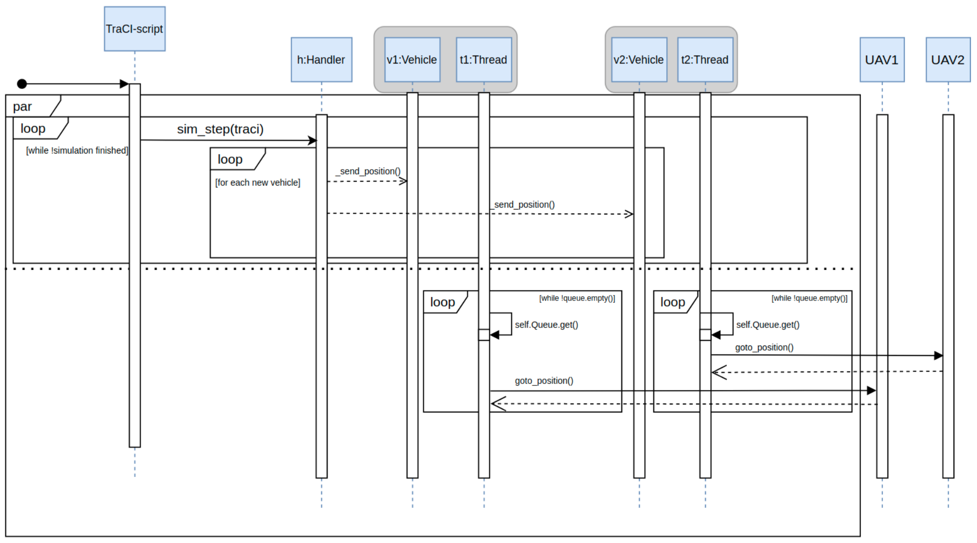


Figure 4.6: Simplified sequence diagram of the execution phase

drone to navigate to and land at takeoff location. This concludes the execution of the testbed.

4.4 Choice of Hardware and Software

4.4.1 Mobility Simulator

We chose SUMO to be used as our mobility simulator. SUMO is a well established, free and open-source traffic simulator with large capabilities. SUMO has granular data retrieval on both simulated vehicles and the simulated scenario as a whole. SUMO is heavily used as a mobility simulator in research by researchers in the field of VANET. Multiple well renowned VANET-simulators use SUMO as as the main mobility simulator, including Traffic and Network Simulation Environment (TraNS)[PRL$^+$08], Veins [SGD11], and MOVE [Lan]. SUMO also provides a Python-API for interacting with simulations. Finally, SUMO comes with many tools and scripts that can be used for a variety of purposes.

4.4.2 UAV Communication Protocol

As presented earlier, the protocol used for drone communication in this testbed is MAVLink[5]. MAVLink is lightweight, open-source and is used for communication between GCSs and UAVs[KJBN20]. MAVLink is widely adopted and is the most

[5]https://mavlink.io/en/

popular choice among the alternatives for drone communication. MAVLink also makes up the core of communication implemented by ArduPilot[6] and PX4[MHP15], both very common autopilot systems[KAA+19]. Although the choice of MAVLink came with the choice of the autopilot, ArduPilot, we still want to emphasize that MAVLink is widely adopted and thus empirically tested.

4.4.3 Autopilot

The autopilot is a collective term describing the solution used to control the drones, everything from the FC firmware, to the operating system facilitating automation. In this thesis, we chose ArduPilot as our autopilot platform.

We chose ArduPilot because of it having properties that fit very well with the way our testbed is operating. We also considered PX4 as our underlying autopilot, another widely used autopilot system. However, there are couple of reasons we did not choose this. Firstly, communication with a drones installed with PX4 through Python is less supported. DroneKit is currently working on getting full integration with PX4, but as of now, the integration is still limited. Additionally, PX4 does not support the Guided flight mode, or similar. Therefore, ArduPilot became the autopilot of choice. With better support to be used with Python, and having the Guided flight mode, ArduPilot has the necessary properties to be used for the drones in our testbed.

4.4.4 Drone

As stated in chapter 3, the drones used for the experiments in this thesis are the Intel Aero RTF. The primary reason these drones were chosen was that they were readily available from our faculty at NTNU, and had the features necessary to be used with the testbed. One of the most appealing feature of the Intel Aero RTF is that the drone is pre-buildt and ready to fly out of the box. Also, the FC that comes with Intel Aero RTF is flexible in terms of autopilot firmware and supports installation of both ArduPilot and PX4. However, we would ideally have used cheaper drones as the Intel Aero RTF is relatively expensive. Finally, as will be described in chapter 5, we had issues with one of our two drones and was not able to make it behave reliably.

4.4.5 Telemetry Radios

The telemetry radios used for UAV communication was decided to be the *Transceiver Telemetry Radio V3 433MHz* from HolyBro[7]. These radios are said to provide a range of 300m out of the box and are to the best of our knowledge, widely used in the drone milieu. It also has good support for the MAVLink protocol. However, as

[6]https://ardupilot.org/

[7]HolyBro telemetry radios: http://www.holybro.com/product/transceiver-telemetry-radio-v3/

will be presented later, we were not able to obtain more than 50m of range with these radios. This may be due to the placement of the radios or other disturbances. We would also like to emphasize that to control many drones at the same time, we believe a multi-point radio should be used instead. However, in the experiments conducted in this thesis only two drone were flown simultaneously, and thus, basic telemetry radios like these were sufficient.

4.4.6 Guided Flight Mode

The flight mode of choice for the UAVs was decided to be the Guided flight mode, described in subsection 3.4.5. The Guided flight mode was also part of the reason ArduPilot was chosen as the FC firmware. The Auto flight mode was considered as a candidate for quite some time. Auto mode lets a drone pilot create predefined mission, upload them to a small computer attached to a UAV, and instruct it to simply execute the mission. The benefit of Auto mode is that it minimizes the potential for packet loss. As the companion computer that stores the mission is connected through serial communication directly to the FC, the communication link does is not dependent on wireless signal strength or interference. The downside is that it does not allow for dynamic, continuous control in real time by a pilot through a GCS or other software.

Guided mode however, allows a pilot to continuously send navigation commands from a GCS or other software to the UAV. The benefit of this is that the UAV can be controlled in real time, giving a pilot the ability to change the movement of the vehicle based on events from external sources, such as SUMO running on a GBC. In addition, it facilitates indirect communication between each UAV, through the GCS. In the testbed implemented in this thesis, this is used for synchronization in the initialization phase (described in subsection 4.3.1). When all drones have reached their starting point, the GBC signals all UAVs to start duplicating the path of their respective, simulated vehicle. A downside with Guided mode is that the communication link depends on wireless signal strength and interference. Although the MAVLink protocol supports up to 255 simultaneous network nodes[8], signal interference may become an issue if many UAVs are airborne at the same time.

4.5 Component Requirements

The way the testbed was designed, raises some requirements to the drones to be used with the testbed. Table 4.2 summarizes the requirements to drones and related equipment to be used in a real world test.

[8]https://mavlink.io/en/about/faq.html

Table 4.2 System component requirements

Component	**Requirement**
UAV(s)	Multi-copter frame
UAV(s)	Have ArduPilot autopilot installed
UAV(s)	Ability to perform MAVLink communication
UAV(s)	Equipped with telemetry radio
UAV(s)	Equipped with GNSS
FC	Support installation of ArduPilot

There are remarks to some of these requirements. First of all, the testbed was developed and tested using multi-frame UAVs, installed with ArduPilot. Without running test on several drone types, we cannot guarantee the same functionality for every multi-copter drone running ArduPilot. ArduPilot becomes a requirement due to the Guided flight mode. As this mode is not a standard flight mode, we cannot insure interoperability with other autopilot- or FC software. There exists other widely used FC software, like PX4 for instance, which do not support Guided mode or similar modes.

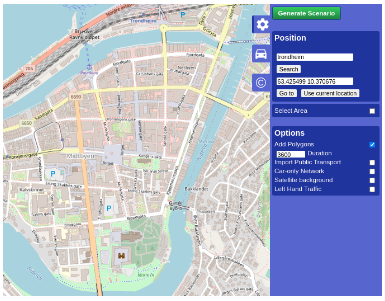
4.6 Challenges and Trade-offs

The creation of this testbed did not go without challenges. For the drones to duplicate the movement of the SVs, our interface collects mobility traces on each SV from SUMO and performs calculations and operations on the data, before sending navigation commands to the drones. We quickly realized that the mobility traces of the SVs could not be sent directly to the UAVs for several reason and thus had to mitigate this issue. We highlight two of main challenges during development and testing.

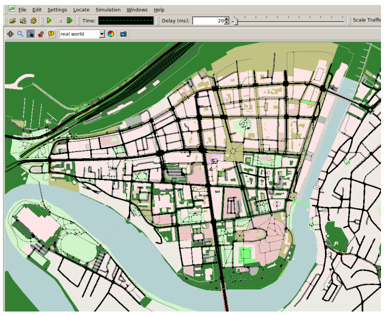
4.6.1 Network Georeferencing

To the best of our knowledge, there is no simple way of georeferencing a manually created SUMO network file. SUMO provides a full solution called osmWebWizard.py, which lets one export georeferenced traffic scenarios from a world map. This approach is feasible if someone would want to generate network files based on real roads and areas. However, when designing network files manually, we did not find a simple solution for georeferencing. This is needed in our case because the network files need to be georeferenced to the flight area.

The simplest way we found was to use osmWebWizard.py to export a full scenario of the flight area, open the generated file, copy the projection information and paste it



(a) osmWebWizard web interface



(b) Exported georeferenced network, as seen in the *netedit* GUI

Figure 4.7: osmWebWizard and Netedit

into our manually created network file. As this is a cumbersome and time consuming solution, we created a script that georeferences a network file only based on an origin coordinate and UTM-zone. As mentioned earlier, we extended an already existing script that is used by osmWebWizard, called *tileGet.py*. This script is used to fetch map images of an already georeferenced network file. With our modification, one can georeference a network file and at the same time, download the background images corresponding to the mapping. This made it quicker to create traffic scenarios in netedit and make them ready to be used with out testbed. The script created for this can be found on the GitHub repository that complements this thesis [Gro21].

4.6.2 Speed Control

The greatest issue we had while developing the testbed relates to the behavior of the ArduPilot autopilot. We quickly discovered that the default behavior when a UAV running ArduPilot in Guided mode receives a MAVLink navigation command e.g., *Command: Go to waypoint* $(longitude, latitude)$, was that the drone accelerated toward this waypoint until approximately halfway. Then, it would decelerate until it had reached the waypoint, coming to full stop. Additionally, MAVLink commands are non-blocking in the sense that if a UAV receives such a command while already navigating towards another waypoint, the new command overrides the original one and the drone will immediately start obeying the most recent command received.

From this, a challenge arise. Lets say that we want a UAV to follow the same path in the real world as a vehicle simulated in SUMO. Lets also say that for each simulation time step, we obtain the location of the SV and directly send it to the drone as a waypoint to visit. For this to be successful, the SV in the simulation must be at the exactly same place on the path as the drone at any given time. If the drone starts falling behind the SV on the route, the drone will start to cut corners as it will not have time to complete commands before new commands are received, overriding the current one. The result of this will be that, when finished, the drone will not have visited every point on the path of the SV. Additionally, a drone can never be in front of an SV on a path, because the path of the SV determines where the drone will fly. We deemed it infeasible to solve the challenge of having the drones and SV at the same location at all times, but we came up with another solution.

Our first thought was to solve this by utilizing a simple queue-like data structure. As the SV travels along a path in SUMO, we extract the location of the vehicle at every time step and add it to a queue. When a drone reaches the current waypoint, the next waypoint in the queue can be sent to the drone. The drone can then, at its own pace, grab the next element in queue and navigate to that location. This ensures that when the drone is finished, and the queue is empty, it has visited the same points as the SV. However, due to the mentioned default drone behavior in Guided mode, this solution resulted in the drone coming to a full stop at every waypoint, which does not coincide with the behavior of the SV, which acts as a continuously moving car, maintaining speed on straight roads and reducing speed while turning.

After several different attempts, we ended up creating a solution that largely enhanced the performance of the drones both in terms of speed and how well it duplicated the path of the SV, albeit more optimization can be done. Even though there are several details to the solution, we present the overall idea of the solution in two steps. The first step involves reducing the number of waypoints sent to the drone as much as possible, while also attempting to guarantee that the drone follow the correct path. The second step involves creating an *acceptance radius*, presented later.

To reduce the number of waypoints sent to the drone, we only force the drone to navigate to waypoints where the heading, i.e., the compass direction, changes. The waypoint W_1 in Figure 4.8 is an example of such a waypoint. At each simulation time step, we extract the location information for an SV. If the SV has changed its heading since the last timestep, the waypoint is put in the queue and marked as a crucial waypoint. If the heading is the same as the last waypoint, it means that the SV is moving in a straight line, and the waypoint is generally not queued. We say generally because some of the waypoints along a straight line is queued, but not marked as crucial. For simplicity we will not go into detail, but it has to do with ensuring that the drone follows the correct path in case of a packet loss of a crucial point.

The second step involves the mentioned *acceptance radius*, illustrated in Figure 4.8. In this figure, W_n denotes waypoints that a drone should visit in its journey to duplicate the path of an SV. The solid line with arrows denotes the ideal path for

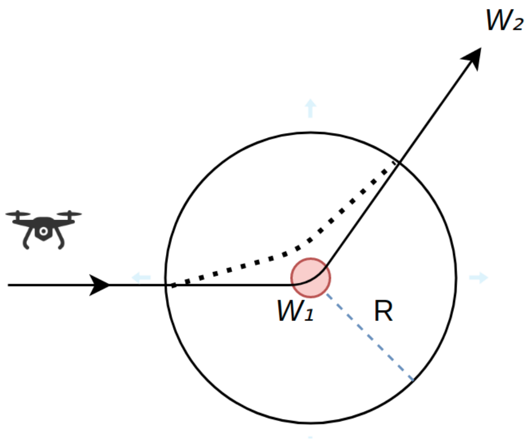


Figure 4.8: Acceptance radius for waypoint

the drone, i.e, the path of the SV, and the dotted line illustrates the actual flying path. A drone is traveling towards a crucial point W_1. Instead of waiting to reach W_1 before continuing to W_2, the next waypoint in the queue, W_2, will be sent to the drone as soon as it reaches the acceptance radius of W_1. As mentioned, the default behavior of the drone is to decelerate and come to a full stop when reaching a waypoint. With an acceptance radius enabled, we reduce this issue by allowing the drone to start accelerating towards W_2 before coming to a full stop at W_1. Increasing the acceptance radius for waypoints will allow the drone to maintain a higher speed, but also increase how much corners are cut. Decreasing this radius has the opposite effect. A lot of testing was done to find a good balance between these two properties.

We determined that the drones ability to follow the path of the SV was of higher importance than maintaining the correct speed while turning. From our results in chapter 6, one can see the effects of this decision. The drones are able to accurately duplicate the path of the SVs, but struggle to maintain speed when turning.

4.7 Limitations

4.7.1 Traffic Scenario Complexity

The mitigation for issues related to speed control presented above, in subsection 4.6.2, limits the complexity of the traffic scenarios that can be successfully executed by the testbed. The most significant limitation is that the testbed does not support rapid fluctuations in speed. If we introduce for instance crosswalks or stop signs into the traffic scenarios, drones will not behave the same as the SVs. Even though an SV may stop for a pedestrian crossing the road in SUMO, the real life drone will not. This is because the drones are only forced to travel to crucial points and will skip any speed reductions along a straight road. To overcome this limitation,more complex logic would need to added to how the drones are controlled.

4.7.2 Acceleration Control

The acceleration rate of the drones is different than for the simulated vehicles. For the drones, this rate is controlled by the FC firmware, in our case ArduPilot, to ensure stability. Through simulations, presented later in chapter 6, we observe that the acceleration rate is generally lower for the drones compared to the SVs, resulting in the UAVs flying at generally lower speeds compared to the SVs. The acceleration parameters can be changed, but not through the DroneKit-py library used in our interface. Additionally, changing the acceleration rate for the UAVs may lead to instability while airborne.

4.7.3 Signal Propagation

Another limitation has to do with signal blockage. A part of testing protocols to be used in VANET, is to determine how they will behave in scenarios when vehicles, other than the ones communication, is either blocking or interfering with the signal. In our testbed, this functionality becomes severely limited. As drones have smaller chassis, are relatively small, and flies in the air, emulating blockage that normal vehicles would have is not possible.

4.8 Safety

In our nonfunctional requirements in chapter 2 we have defined safety as an area of focus. We quickly understood that there are several things that can go wrong when flying drones, especially when automation is involved. This was learned the hard way after multiple crashes and broken propellers. Failsafes to mitigate this is important.

We started by creating some failsafes ourselves and implementing them in the testbed. The first failsafe was about how a drone should react if low on battery. The other failsafe was how the drone should respond if it was instructed to fly outside a geographical flight boundary. If a drone flew beyond a predefined radius from the take-off location, it would return and land at the launch site.

After implementing these failsafes we decided to depart from this approach and let a GCS handle it instead, being empirically tested over a longer period of time and much less prone to coding errors and software bugs. GCS's like QGroundControl, have several failsafes that can be easily configured before each flight. As such, when conducting experiments, in addition to connecting to drones through our interface, we also connected to each drone through QGroundControl. QGroundControl was used to monitor each drone and adapt their behavior in case something went wrong. Included in QGroundControl, failsafes in response to low battery, GPS glitching, EKF variance, maximum altitude and flight boundry radius are included. When any of these failsafes trigger, QGroundControl is able to override our testbed and instruct the drones to react accordingly. How this was set up and used is described in chapter 5.

4.9 Treatment Validation

We will now describe the design of our artifact in relation to the context of testing VANET protocols, and explain how the testbed correspond to the requirements specified in 2.1.2.

For **REQ1**, we designed the testbed to support a varying number of UAVs and SVs. As each pairing between a UAV and a SV is instatiated in their own thread, they can be instantiated and stopped without affecting the rest of the execution. **REQ2** is accomplished by separating the hardware and software implementing the protocol under test from the rest of the testbed. Researchers and developers stand free to choose the OBU to be attached to the drones, as well as the software running on the OBUs. In relation to **REQ3**, concerning flexibility for further extension, we highlight a few different aspects. The core of this testbed is designed so it can be imported into a program running a SUMO simulation using the TraCI Python-interface, and it will take of care of duplicating the movement of SVs in UAVs. With this, as well as being open-source, the testbed facilitate modification and extension by anyone that wants to use it. **REQ4** revolves around the safety of execution. even though the testbed in it self does not implement any safety features, we found that a more feasible solution was to connect a separate GCS of choice to the drones to take care of safety and failsafes. Further, for **REQ5**, we can not guarantee that the testbed will behave correctly in every scenario. However, when conducting simulated testing during and after development, we tested several different traffic scenarios, georeferenced to several different locations, together with a variety of number of vehicles. We revealed some software bugs arising from edge-case scenarios and fixed them. None of the bugs affected the safety of the testbed. After fixing these bugs, we found the testbed to behave predictably in further tests. Finally, in relation to **REQ6**, we believe that our testbed can facilitate rapid testing of VANET-protocols, and reduce the resources necessary to peform them. Drones are usually cheaper and lighter than cars, and more easily transported to the testing field. In addition, after setting up the testbed, initial calibration of drone sensors, and attachment of OBUs, it should be as simple as placing the drones on the ground, and start our interface.

For validating the performance of the testbed, we conducted fully simulated tests. Drones were simulated with ArduPilot SITL, and thus no OBUs were included. How the experiments were set up and executed, as well as the results from these tests can be found respectively in chapter 5 and chapter 6. The drones demonstrated good accuracy in duplicating the path of their SV at the costs of not being able to maintain speed in traffic scenarios involving frequent turning. However, in scenarios with longer distances and more straight roads, we were able to achieve relatively higher speeds as well as a high accuracy for the path tracing. Yet, the acceleration of the simulated drones were considerably lower than the SVs.

In the initial phase of this thesis, the problem investigation, we identified that a great issue with traditional field testing of VANET-protocols is that they are expensive and resource demanding. From our simulated testing, it seems like the design of our artifact can help addressing this issue. Even though our testbed is more of a hybrid nature, both vehicle mobility and communication is realized in the real world. Since drones are usually cheaper than vehicles traditionally used in field testing of VANET-protocols, we believe that our proposed design using drones can significantly reduce the costs involved with conducting tests that are fairly close to reality, even though real cars are used.

Additionally, we argue that the resources needed to conduct testing can be significantly reduced with the proposed solution. Due to drones being relatively small and lightweight, they can be transported more easily to the test location. Additionally, if the UAVs used in the test are configured correctly and ready to fly, one would only have to attach OBUs, turn them on and run the testbed. We argue that in addition to reducing resources needed, this also facilitates rapid testing.

With this, we predict that our design will succeed in treating the main issues that we aim to address, to a certain degree.

Chapter 5

Experiments: Setup and Execution

This chapter present the experimental setup and execution for simulated scenarios and real world scenarios. We begin by describing the aspects of the simulated experiments, then we present the details for the real world experiments. Experiments with simulated drones were conducted as part of the treatment validation, as well as to collect and analyze data regarding to what degree drones succeeded in duplicating the behavior of SVs. The real world experiments were conducted to observe the behavior of the testbed in the real world, and is part of the treatment implementation. Scenario 1 and Scenario 2, presented in the next section, were executed in simulated and real life environments for comparison purposes. The results are used in the validation and evaluation of the testbed.

The results gathered from the experiments are also used a baseline for answering our research questions.

5.1 Simulated Experiments

5.1.1 Setup

To conduct simulated experiments, we design two different traffic scenarios. As we would like to compare simulated results to real life tests, we design traffic scenarios that can be used for both experiments. Therefore, these scenarios are based on the limitation of the flight area, Udduvoll Airfield in Trondheim, which is why the traffic scenarios are not very large in size.

Scenario 1, shown in Figure 5.1, is a two-lane road that goes in a straight line, before curving at each end. The shape of the road in Scenario 2, shown in Figure 5.2, is an irregular circle. The yellow car in each figure illustrate the starting point for the SV and the drone. Both of these scenarios only involve one vehicle each. The illustrations are screenshots from the netedit Graphical User Interface (GUI). In the bottom left of each figure is a scale to give an impression of the size of the traffic

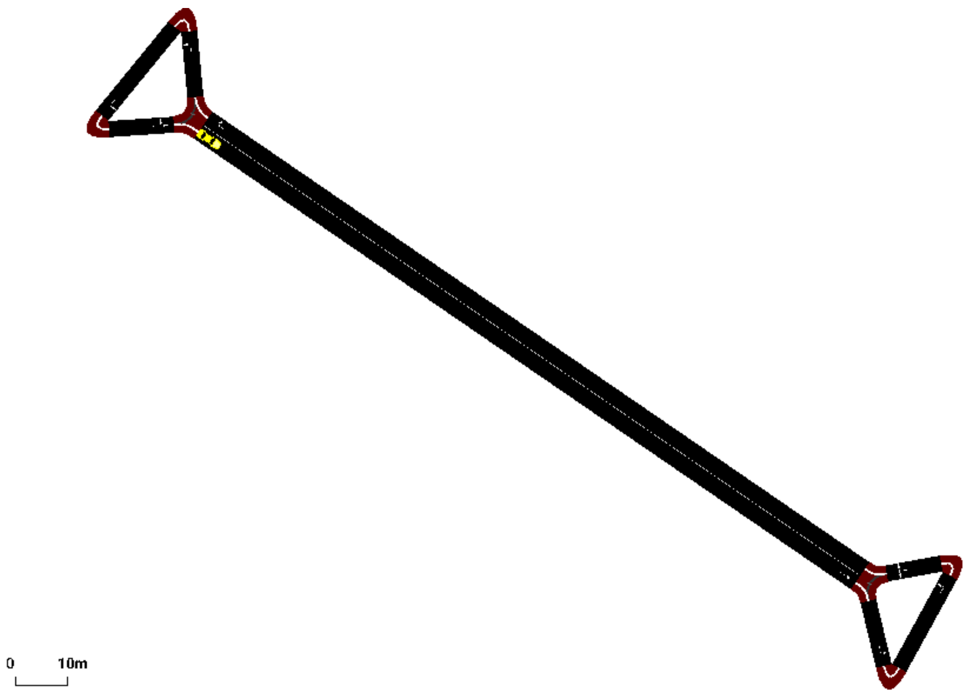


Figure 5.1: Scenario 1

scenarios. In both scenarios, the SV simulated in SUMO is moving in a continuous fashion, without stopping. The speed is regulated my a max speed indicator. This is set to an equal value for drones and simulated vehicles in each test. For each scenario we execute three tests with different maximum speeds.

In addition to these two main scenarios, we design one more to verify the scalability of the testbed. In this traffic scenario we conducted four experiments, increasing the number of vehicles in each one. The number of SVs and UAV used for these tests were 1, 3, 5 and 10. In the Appendix B.2 illustrate this scenario and the starting points for each vehicle.

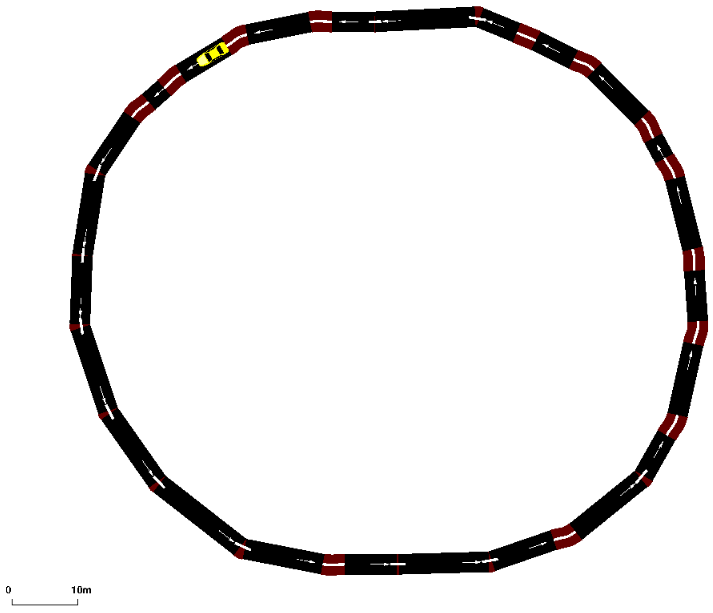


Figure 5.2: Scenario 2

Table 5.1 Software used for simulated testing

Tool	**Version**	**Role**
ArduPilot	3.6	UAV SITL simulation
SUMO	1.8.0	Vehicle simulation
QGroundControl	4.1	UAV monitoring and safety measures
netedit	1.8.0	Creation of network files
geoLocate.py	-	Georeferencing network files
collect.py	-	Data collection from UAV

Table 5.1 summarizes the tools and scripts utilized to execute the simulated experiments and to collect data. *collect.py* has not been mentioned before, and is a simple script we made which connects to a UAV and queries data on speed and location at a specified time interval.

QGroundControl is used to get a visual overview of UAVs. In addition, it lets us monitor the location of drones on a map, as well as their speed and altitude. However, the most important aspect QGroundControl is used for, is its built in safety features and failsafes. Even though these failsafes are not as crucial in simulated environments, we wanted to conduct simulated experiments as close to reality as possible.

5.1.2 Execution

For Scenario 1 and Scenario 2, we iterated the steps in the activity diagram from Figure 4.4 for conducting simulated experiments. First, we conducted the preliminary steps of creating the traffic scenarios in netedit, and using geoLocate.py to georeference the network files to our wanted location, in this case, Uddevoll Airfield in Trondheim. Next, we initialized the UAV with ArduPilot SITL. When simulating a drone through ArduPiot SITL a program called MAVProxy is automatically started. MAVProxy is used to merge outgoing MAVLink datastreams into one, and allows us to connect to simulated drones through multiple data streams simultaneously. Finally, we connected QgroundControl and collect.py to the simulated drone before configuring and initializing our implemented interface.

In our interface we added some code to collect position- and speed data from the SV. We gathered data from the SV at each simulation timestep. To obtain granular data, we specified a small simulation time step length of 0.1 seconds. collect.py was set to querying the drone for data at the same interval of 0.1 seconds. When decreasing this interval more, we experienced a lot of duplicate position data when querying the drones. With longer intervals, we observed the captured data was not detailed enough and did not capture important data points from the SVs or the UAVs. This would make the results from DTW difficult to interpret.

We conducted three experiments for Scenario 1 and three experiments for Scenario 2, changing the max speed for the drones and SV each time. We tested each scenario at max speeds of $5m/s$, $10m/s$ and $15m/s$. After each experiment, we had obtained multivariate time-series data from the SV and drone, with the same sample frequency. A simple illustration of the experimental setup is shown in Figure 5.3. The interface, depicted yellow is the only component actively controlling the movement of the drone. QGroundControl does not actively control the drones, but is able to take control if a safety-event is triggered. After executing the tests, we formatted the collected data to CSV-format to perform further analysis. To compare the trajectories of the simulated vehicles and the drones, we utilized DTW with some modifications to fit our data. The results from conducting the experiments as well as data analysis can be found in chapter 6

After conducting experiments for Scenario 1 and Scenario 2, we conducted four multi-vehicle experiments in a new traffic scenario. As mentioned, these four experiments were conducted with 1, 3, 5 and 10 vehicles. For each experiment, we initialized the same number of simulated drones with ArduPilot SITL as the number of

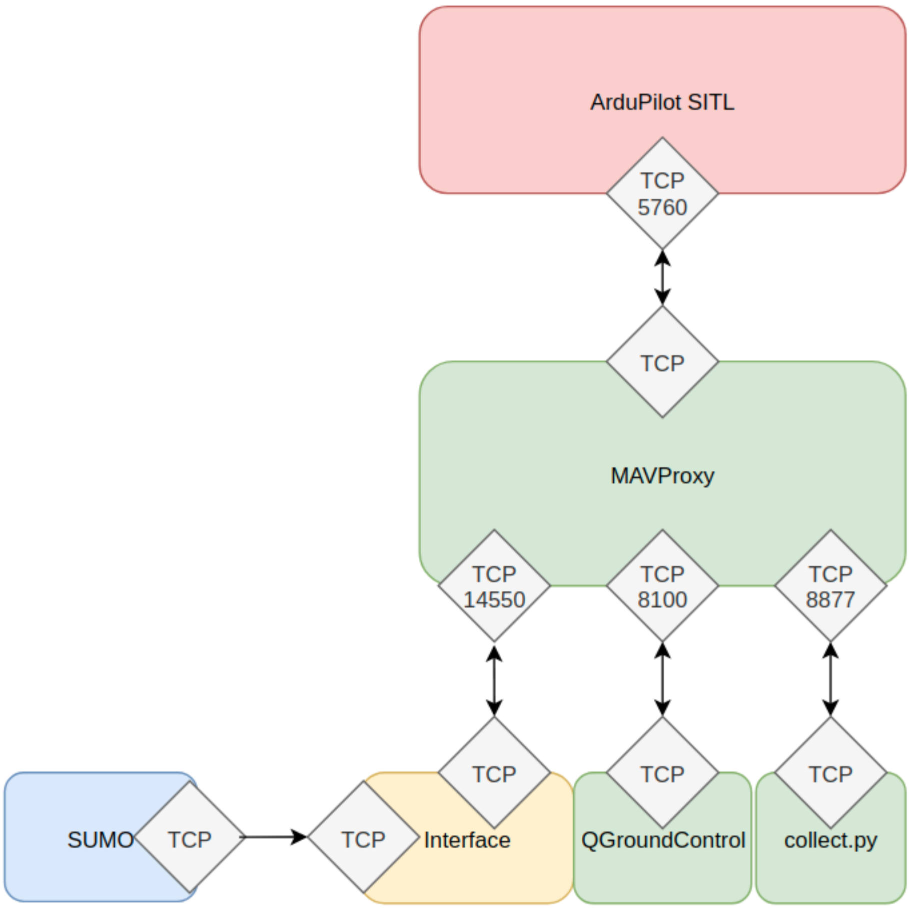


Figure 5.3: Setup for single-vehicles experiments. The numbers indicate TCP ports.

SVs. We again connected QGroundControl to the drones and configured our interface with the correct parameters to be able to communicate with all drones simultaneously. We did not collect data from these experiments as they were only intended to test the scalability of the testbed. A simple illustration of the experimental setup with multiple vehicles in a simulated environment can be found in Appendix B.1. This figure shows the architectural setup when conduction simulated experiments with two drones. However it is easily extended to more drones by just starting more instances of ArduPilot SITL and MAVProxy and connecting QGroundControl and out interface to them.

5.2 Real world Experiments

To iterate the implementation phase of the engineering cycle, we executed real life tests using our testbed. We did not conduct testing with OBUs attached to drones, as we were no able to obtain any VANET-protocols to test. Even though this means that the real world experiments were not conducted completely in the context of VANET protocol testing, we argue that the results are still sufficient concerning our research questions.

5.2.1 Setup

Three experiments were conducted in a real world environment. First, we conducted experiments with Scenario 1 and Scenario 2, one experiment for each scenario. We then attempted a multi-vehicle experiement, although we only had access to two drones.

Compared to the simulated tests presented in the last section, some architectural changes were needed. First of all, we swapped the drones originally simulated with ArduPilot SITL with real, Intel Aero RTF drones. In doing this, we also had to swap the communication link between the GBC and the drones. In the simulated scenario, our interface communicated with the drones over a link-local TCP connection. For the real life experiments, this communication was done over telemetry instead. For this, our Intel Aero RTF drones and the GBC were equipped with telemetry radios. On the drones, the telemetry radio was connected through a serial port, directly communicating with the FC and attached with adhesive tape on the top of the drone body, shown in Figure 5.4 . To the GBC they were connected through USB. On the GBC, the telemetry radio was connected through a USB port. Figure 5.5 illustrates the architecture for the real life tests for the single-vehicle scenarios, Scenario 1 and Scenario 2. The box colored in gray illustrates a handheld RC transmitter that was used as a backup link to regain to control of the drone in case of communication failure between it and the GBC.

For data collection in the experiments with Scenario 1 and Scenario 2 in the real world, the same approach that was used in the simulated experiments was utilized, with one additional step of monitoring packet loss. This additional step was done by keeping track of how many commands were sent from the interface and comparing it to how many commands were received by the UAV. To obtain the number of sent packets, we added some code in the interface that accumulated the number of packets sent. To obtain the number of received packets, we extracted the logs stored locally on the drones after each flight.

Before using the drones, we had to update their firmware. We followed a compre-

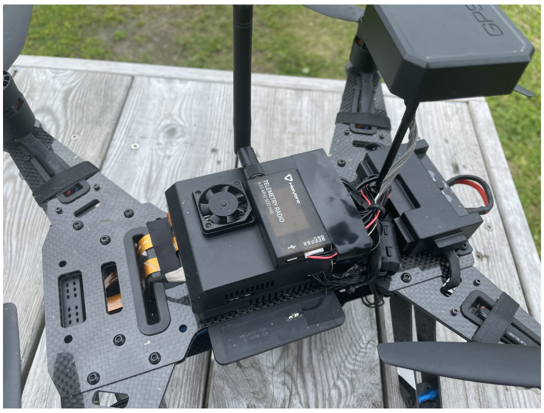


Figure 5.4: Telemetry radio is attached on top of a drone

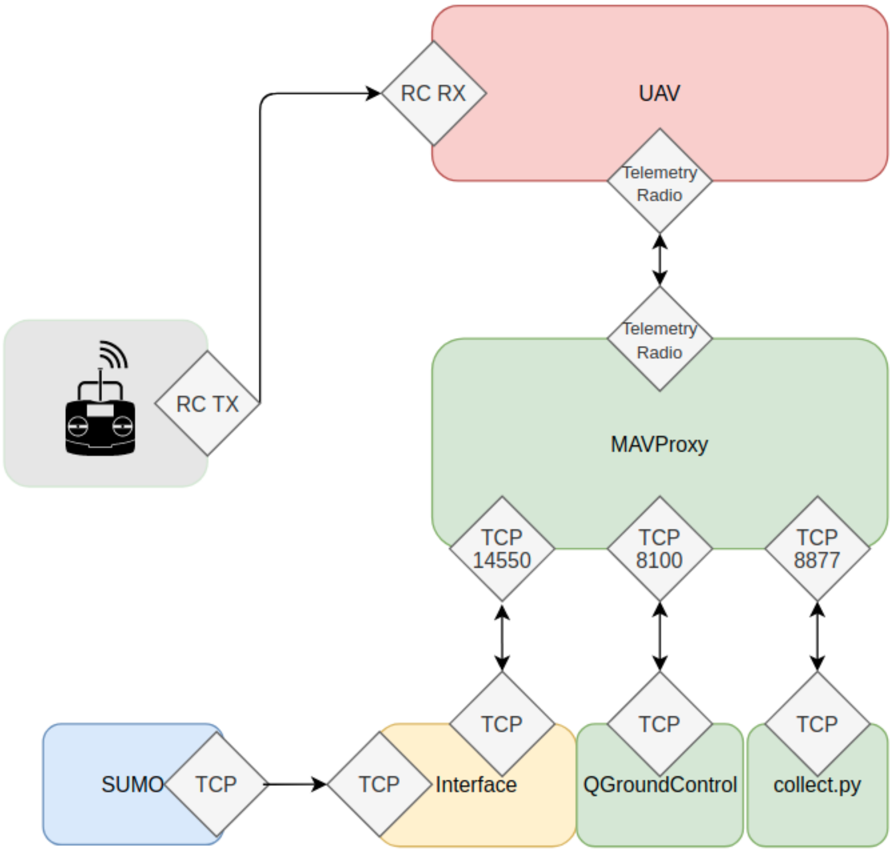
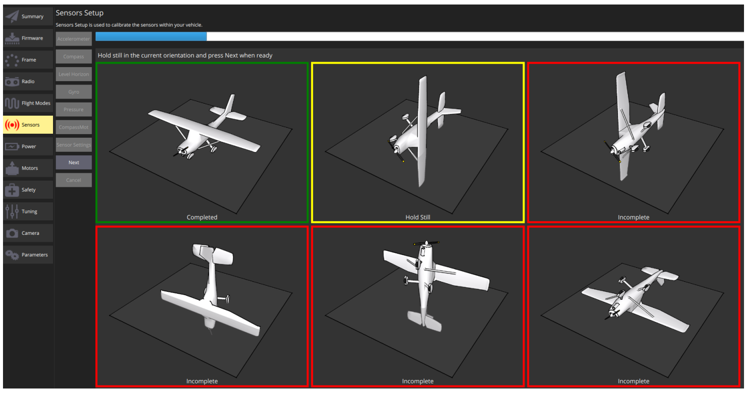


Figure 5.5: Setup for single-vehicle experiments in the real world. The numbers indicate TCP port numbers



Sensors Setup

Sensors Setup is used to calibrate the sensors within your vehicle.

Hold still in the current orientation and press Next when ready

Figure 5.6: Snapshot of accelerometer calibration process in QGroundControl

hensive guide by Intel[1] for the initial installation of ArduPilot on the Intel Aero RTF. After flashing the BIOS and FPGA with the correct firmware and installing ArduPilot, we did an initial calibration of different sensors like compass, accelerometer, and gyroscope using QGroundControl. Figure 5.6 shows a snapshot of the calibration process for the accelerometer. After calibration, we performed tests to verify correct operation of both drones. We did this by conducting test flights, both manually controlled with the RC transmitter and programatically with Guided flight mode. This did unfortunately not go as planned.

While conducting test flights, we found that only one of the drones was working as expected. The second drone had frequent, irregular failures in the GNSS-system and the magnetometer, i.e,. the compass. This resulted in aggressive behavior, fly-aways and several crashes. After some investigation and studying logs, we believe the most likely cause is due to hardware failure or compatibility issues. Several attempts were made to remove the seemingly arbitrary sensor failures. Among these were re-calibration, software re-installation and changing software used for calibration and many hours of debugging. We were not able to isolate the culprit causing the issues.

A lot of time was spent getting comfortable with flying the functioning drone manually in different flight modes. This was so that we would be able to retake manual control should something go wrong while flying.

We also had to adjust the center of gravity on the drones. The Intel Aero RTF drone has a open compartment in the middle of the chassis that is meant to hold the LiPo battery that powers it. The 3S LiPo battery used in our experiment was too long to keep the center of gravity balanced enough when inserted. This lead to the

[1]Installing ArduPilot on the Intel Aero RTF: https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-setup

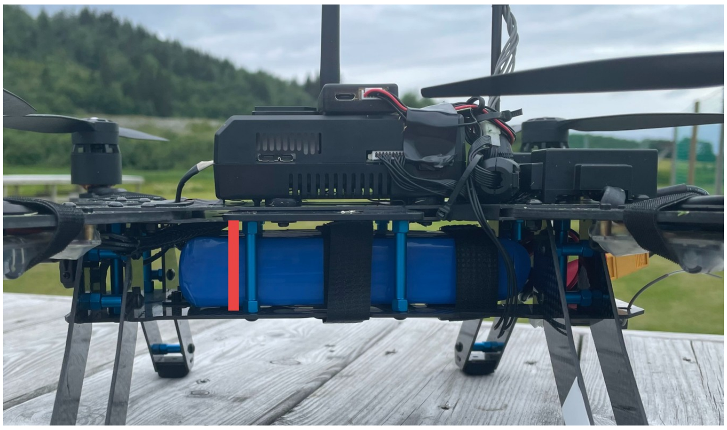


Figure 5.7: Battery position after adjustment

drones being imbalanced and not unable to keep a stable enough position. To adjust for this, we removed one of the pillars used to separate the upper and the lower part of the chassis. After this adjustment, the drones were able to maintain a more stable position while airborne. The result from this adjustment is shown in Figure 5.7.

Even though we only had one fully operation drone, we decided to attempt a multi-vehicle experiment using our testbed. For this, some additional setup was needed. We first designed a new traffic scenario in netedit, involving two vehicles. We also had to configure each drone and and the telemetry radios to correctly address traffic between the GBC and the drones. For configuring the drones, we changed the a system parameter called SYS_MAVID. No drones participating in a multi-vehicle test should have this parameter set to the same value. Additionally, each pair of telemetry radios were given a common *Net ID* to separate data streams. These configurations were done using a GCS called Mission Planner[2], due to lack of support from QGroundControl.

The architectural setup and traffic scenario for this experiment can be found in Appendix C. Most of the equipment used for this experiment can be seen in Figure 5.8.

[2]Description of Mission Planner: https://ardupilot.org/planner/

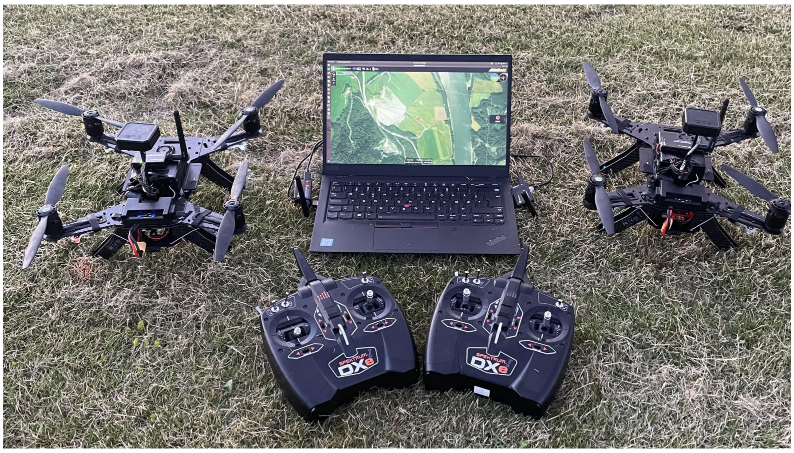


Figure 5.8: Equipment used for multi-vehicle experiment

Table 5.2 Software used for real-life testing

Tool	Version	Role
MAVProxy	1.8.32	Merge and split telemetry data
ArduPilot	3.6	UAV software
SUMO	1.8.0	Vehicle simulation
QGroundControl	4.1	Manual UAV monitoring
netedit	1.8.0	Creation of network files
geoLocate.py	-	Georeferencing the network file
collect.py	-	Data collection from UAV

Table 5.3 Hardware used in real-life testing

Tool	Role
Intel Aero Compute board	Linux board running ArduPilot
STM32 Microcontroller	FC running ArduPilot
Intel Aero RTF	UAV
HolyBro Tranceiver telemetry radio 433MHz	Telemetry Communication
Spektrum DXe RC Controller	Backup control link
3S 5000mAh LiPo battery	Power supply for UAVs
Laptop computer	GBC

5.2.2 Execution

After the initial steps of updating drones, getting comfortable flying manually, as well as making the architectural changes needed for drone communication, we were ready to perform real world experiments. Again, we iterated the steps from the activity diagram in Figure 4.4. We travelled to Udduvoll Airfield in Trondheim, a airfield for model airplanes, and calibrated the sensors on the drones again. To calibrate all sensors should not be necessary, but the compass should be re-calibrated each time the flight location is changed to account for variations in magnetic field. Even though the intention of the testbed is to attach OBUs to the drones, we did as mentioned not have access to any potential VANET-protocols implemented on OBUs that were small enough to be attached to our drones such as a smart phone. Therefore, no OBUs were included in these experiments.

First we performed one real world test with Scenario 1 and Scenario 2. For each test, the maximum speed of the SV and the drone was set to $5m/s$.

We placed the drone on the airfield and continued by starting MAVProxy manually. Then, we connected QGroundControl and collect.py to the drone through MAVProxy. After starting these processes we configured the testbed to also connect to the drone through MAVProxy. Finally we started our interface and closely monitored the drone as well as the SV in SUMO.

A couple of tries was needed to conduct a successful experiment. In the beginning, we experienced issues with the airborne drone suddenly stopping and returning to the launch site. With some investigation it was discovered the telemetry radio did not deliver the 300 meters range of communication as first thought. We were only able to reach a maximum range of approximately 50 meters. When the drone experienced loss of radio signal, QGroundControl would activate a safety measure where the default behavior is enter RTL flight mode. By keeping the telemetry radio connected to the GBC up in the air, we were able to extend the range a little but, but not nearly enough. We also tried changing the placement of the radio on the drone, with little success. In the end, although not an optimal solution, this issue was mitigated by walking after the flying drone while carrying the GBC, to keep it within range. With this, the tests were executed successfully, from launch to landing. The results from these experiments can be found in chapter 6.

Finally, we attempted the multi-vehicle experiment. We experienced both drones taking off successfully, going to their correct starting points and flying along the path of their respective SV. However, after a while, the unstable drone had a malfunction in the magnetometer and started to rapidly descend while flying away. We did not manage to regain manual control in time to avoid a crash. However, the drone did not suffer major damage. The fully functioning drone completed its route and landed safely back at the launch site. Even though this experiments was not fully successful, our testbed controlled both drones successfully until the malfunction. For safety reason, we did not conduct any more multi-vehicle experiments after this.

Chapter 6 Results

In this chapter we present results from our experiments. This includes results from fully simulated and real world tests. We first present the results from the simulated experiments. Then, the results from the real world tests are presented.

6.1 Simulated Experiments

The simulated experiments were mainly executed as part of the validation step of the design science cycle, and to predict the behavior of the drones in real life. In addition to the experiments mentioned here, simulated testing was conducted several times throughout development to verify correct operation and uncover bugs and other issues. We present the results from the single-vehicle experiments first. Then, in subsection 6.1.3 we present the results from the multi-vehicle experiments.

6.1.1 Travel Time

To compare the temporal similarity between movement of the SV and the UAV in Scenario 1 and Scenario 2, we recorded total travel time. Concerning the SV, the time was recorded from the timestep it was added to the simulation, until it finished its route. For the UAV, we recorded the time from when it had reached its starting point and until it entered the RTL flight mode. The times are listed in Table 6.1.

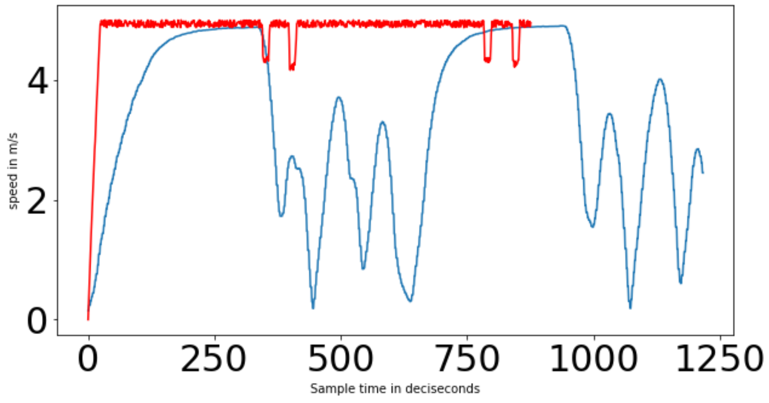
Table 6.1 Time spent to complete route, simulated experiments

Scenario	**speed**	**SV**	**UAV**
Scenario 1	5 mps	87.7 seconds	123.3 seconds
Scenario 1	10 mps	52.9 seconds	98.2 seconds
Scenario 1	15 mps	42.5 seconds	95.2 seconds
Scenario 2	5 mps	57.4 seconds	99.8 seconds
Scenario 2	10 mps	30.1 seconds	99.2 seconds
Scenario 2	15 mps	22.5 seconds	95.5 seconds

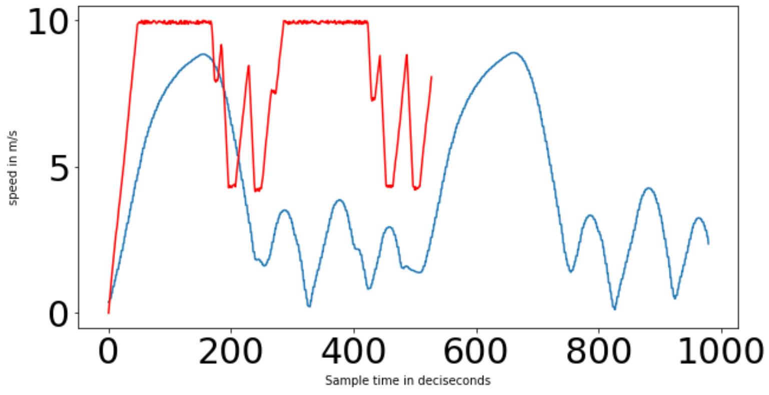
The collected data shows that the UAV spent more time than the SV to completing its route in all experiments. To determine what caused this, we plot the speed for both the SV and the UAV for every test. Figure 6.1 and Figure 6.2 illustrates the speed the SV and UAV for Scenario 1 and Scenario 2 respectively. The graphs colored red depict the speed of the SV in the given scenario. The graph colored blue depict the speed of the UAV.

Several observations can be made from these figures. The first observation is that in almost all scenarios, the SV reaches the maximum speed while the UAV does not. The exception to this is Scenario 1 with a maximum speed of $5m/s$ (Figure 6.1a), where the drone is able to reach the max speed twice while travelling on the straight segments of the route. The second observation is that the drone generally exhibits lower acceleration than the SV. The third observation is that the differences in behavior between the SV and UAV is most prominent while turning. This can especially be observed in the graphs related to Scenario 2 (Figure 6.2). In these experiments, the SV accelerates and maintains the max speed throughout the whole experiment, while the speed of the drone is more unstable and generally lower than the SV. The final observation is that when the maximum speed is decreased, the difference between the speed of the SV and the UAV also decreases, resulting in a lower difference in travel time.

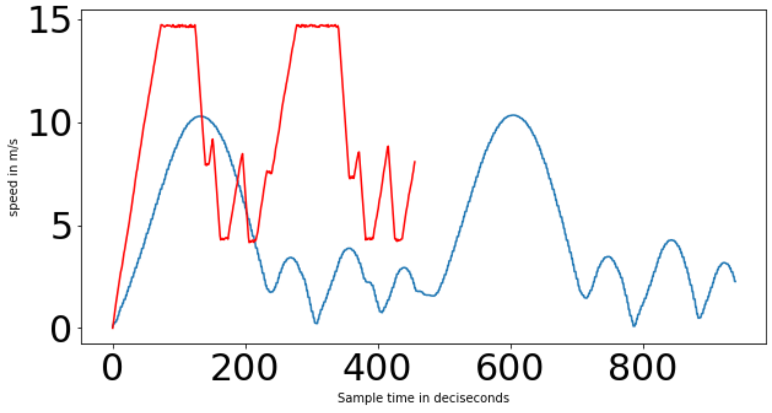
To summarize, these graphs show that the drone generally obtain lower speeds than the SV and thus spend more time completing a route. However, as the maximum speed is decreased, the drone gets closer to keep up with the speed of the SV. These observations will be further discussed in the next chapter.



(a) 5 m/s maximum speed

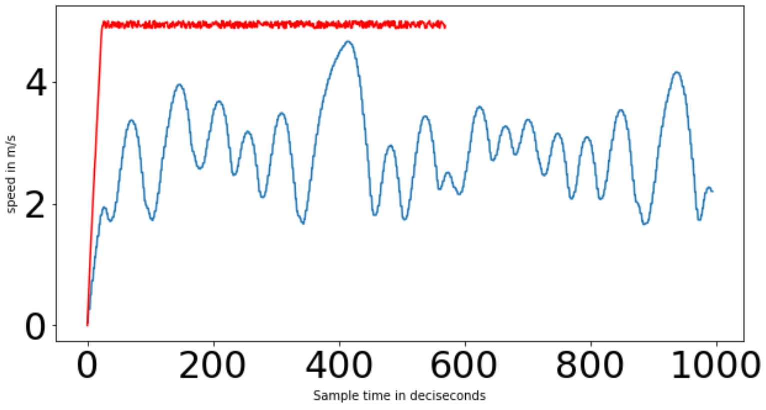


(b) 10 m/s maximum speed

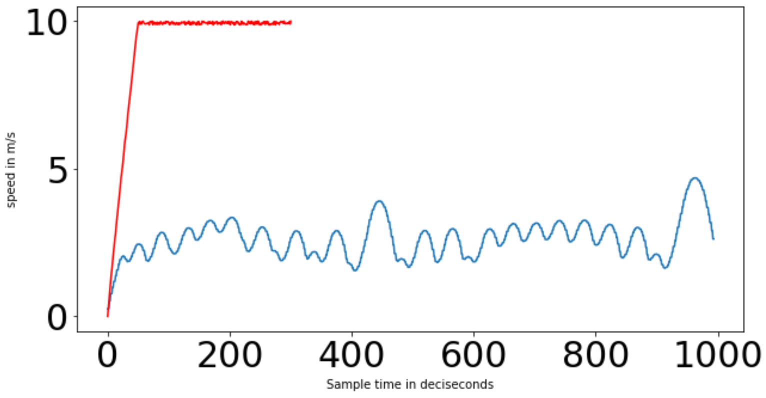


(c) 15 m/s maximum speed

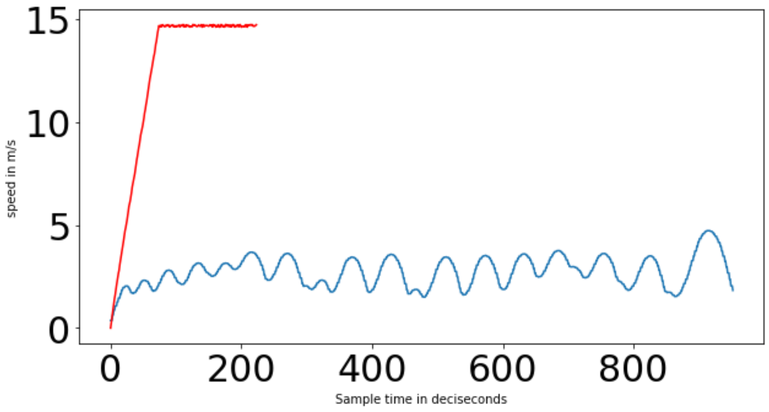
Figure 6.1: Graphs illustrating the speed of SV (red) and UAV (blue) in fully simulated experiments, Scenario 1.



(a) 5 m/s maximum speed



(b) 10 m/s maximum speed



(c) 15 m/s maximum speed

Figure 6.2: Graphs illustrating the speed of SV (red) and UAV (blue) for fully simulated experiments, Scenario 2

6.1.2 Path Tracing Accuracy

For measuring the spatial similarly between the trajectory of the SV and the UAV we utilize DTW, presented in section 3.6, with the Haversine formula (described in section 3.7) as the distance function for constructing distance matrices. Similarity measurements for the path of the UAV and SV are only presented for the experiments conducted at a maximum speed of $5m/s$. This is because we will compare the results with the experiments conducted in real world, which were conducted in at a maximum speed of $5m/s$.

Figure 6.3 and Figure 6.4 illustrate the results from DTW for Scenario 1 and Scenario 2 respectively. The red line illustrate the path of the SV in SUMO. The red dot illustrate the starting point for the drone and the SV. The plotted line with color mapping illustrate the path of the drone. The colormapping indicate the distance in meters between the path of the UAV and the SV at any given geographical location, according to the optimal warping path. The x-axis depict the longitude of a point, and the y-axis depict the latitude.

From these figures it can be observed that the drone in general follow the path of the simulated vehicle accurately. We see however, some segments, where the distance between trajectories exceeds 2 metres in Scenario 1 and 1.4 metres in Scenario 2.

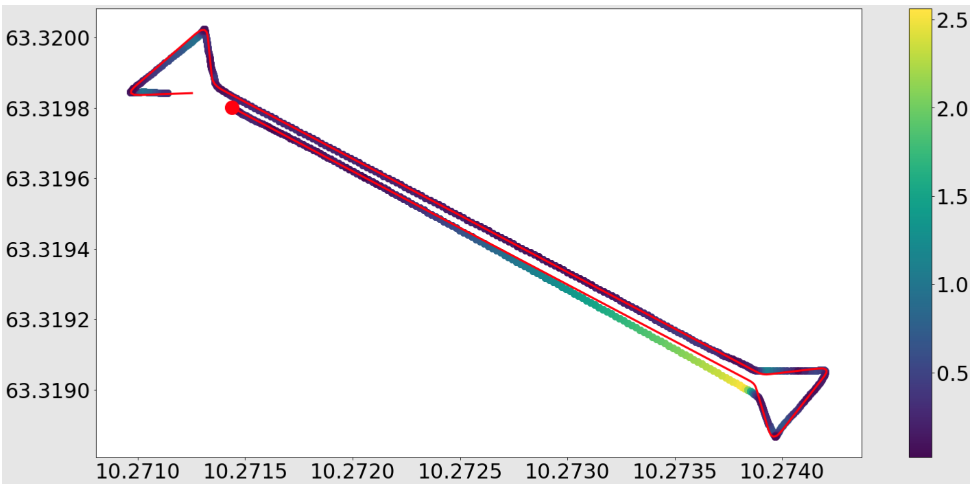


Figure 6.3: Distance measures for scenario 1

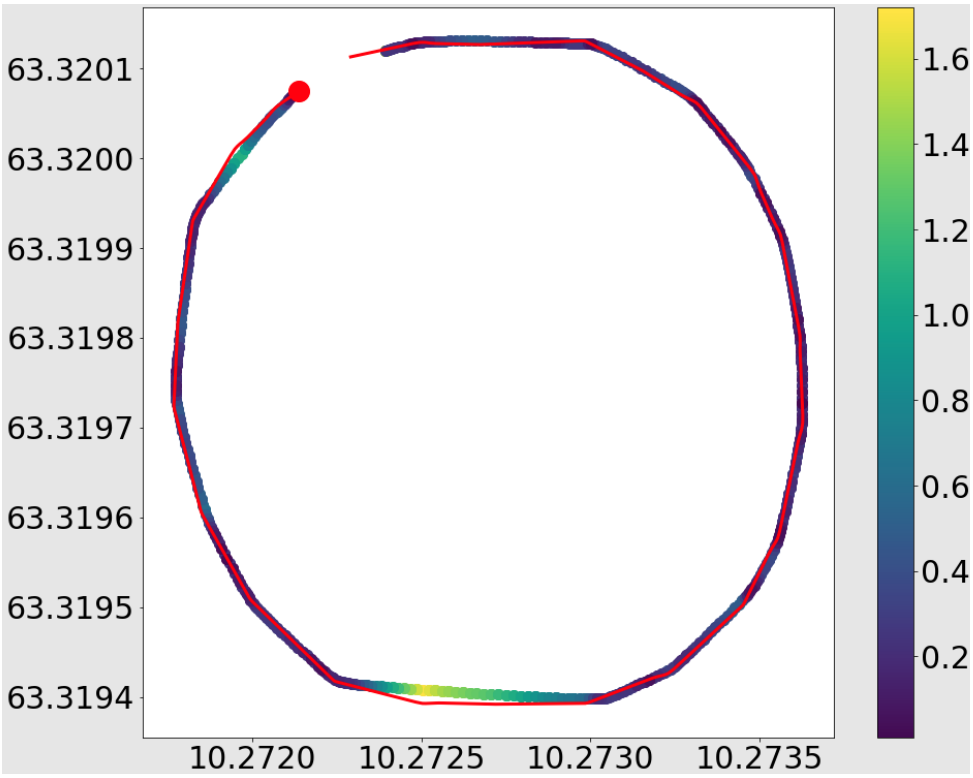
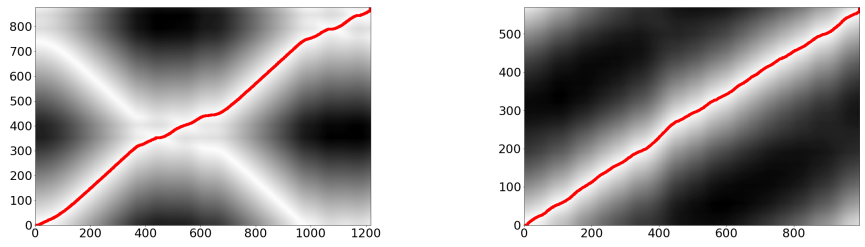


Figure 6.4: Distance measures for scenario 2



(a) Scenario 1

(b) Scenario 2

Figure 6.5: Optimal warping paths for Scenario 1 and Scenario 2, in simulated experiments

Figure 6.5 illustrates the distance matrices between the trajectories of the SV and the drone, in both scenarios. The red line depict the optimal warping path between the trajectories. The warping generally follows the diagonal, illustrating that there is generally good spatial alignment between the trajectories in both scenarios, even though they do not alight in time.

6.1.3 Multi-vehicle Experiments

As mentioned, experiments with multiple vehicles was conducted to test the scalability of the testbed. In all experiments, the testbed behaved as intended. First, all drones ascended to the correct height and travelled to their respective starting points. When all drones had reached their starting points, they all started duplicating the route of their respective SV. Every drone entered RTL mode when finished with its route, returned to the home location, and landed. Table 6.2 summarize the results. Figure 6.6 shows a snapshot from QGroundControl while the experiment with 10 drones was running. The red line shows the mobility trace for one of the UAVs.

Table 6.2 Results from multi-vehicle experiments

Number of vehicles	**Success**	**Remarks**
1	Yes	Experiment executed successfully
3	Yes	Experiment executed successfully
5	Yes	Experiment executed successfully
10	Yes	Experiment executed successfully

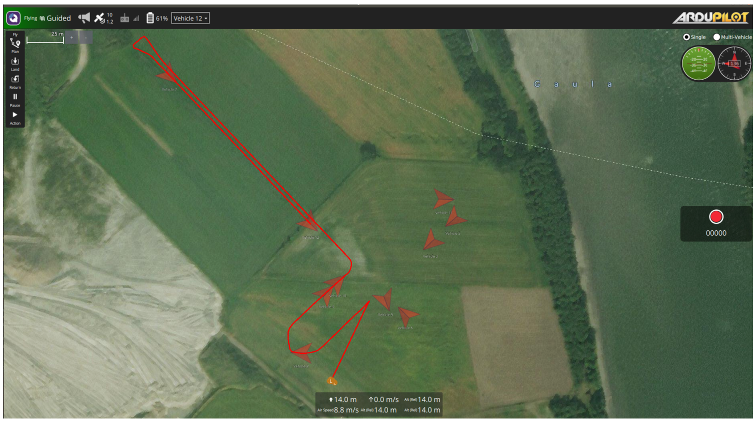


Figure 6.6: Snapshot of multi-vehicle experiment with 10 drones

6.2 Real World Experiments

The real world experiments were conducted as part of the implementation phase of the design science cycle, to evaluate the behavior of the testbed in a real world context. Due to time limitations, we did not conduct testing of an actual VANET-protocol, and thus did not attach OBUs do the drones.

In subsection 5.2.2, we described how the real world experiment with multiple drones unfolded. As it was not fully successful, and no valuable data beyond that was obtained, we will only present the results from the single-vehicle experiments.

6.2.1 Travel time

As with the simulated experiments, we also gathered data on travel time in the real world experiments. The results were gathered in the same way as in the simulated experiments. Table 6.3 shows the travel time of the SV and UAV in Scenario 1 and Scenario 2.

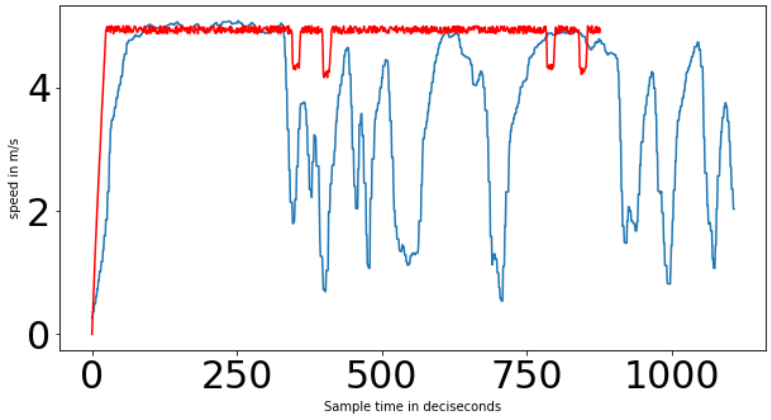
Table 6.3 Time spent to complete route, real world experiments

Scenario	**speed**	**SV**	**UAV**
Scenario 1	5 m/s	87.7 seconds	111.3 seconds
Scenario 2	5 m/s	57.4 seconds	83.4 seconds

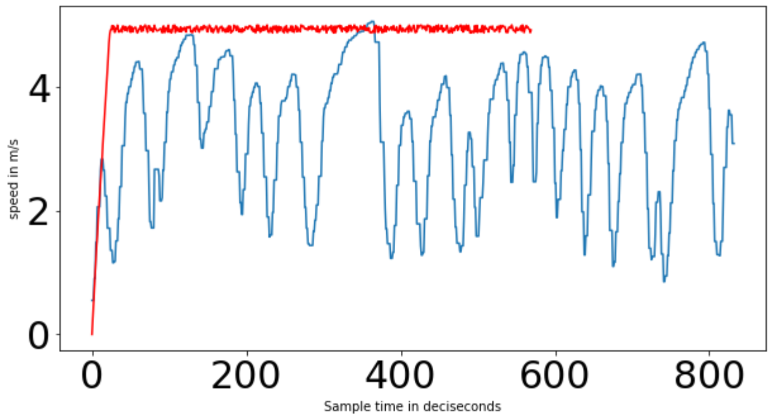
As expected, in both scenarios, the SV spends the same amount of time on its route as in the simulated experiments. Again, the drones spend more time completing their route. We plot the speed of the drone and the SV in both scenarios, shown in Figure 6.7.

Most of the observations we saw from the simulated experiments can also be seen here. In both scenarios, the SV maintains the maximum speed during most of the experiment, while the drones do not. Again, we see the the biggest speed difference between the SVs and the drones happens in the turns. Also, again, the acceleration of the drones is lower than for the SV.

In Figure 6.8, we have plotted the speed of the drones in the real world experiment, and the speed for the simulated drones in the same scenario, to illustrate the similarities. Although shifted in time, the graphs have a similar patterns in both scenarios. For Scenario 1, there is an anomaly between around 60 and 80 seconds in to the experiment, where the real drone experienced a rapid drop in speed which the simulated drone did not. This was caused by a sudden loss of altitude, in which the drone recovered its altitude before continuing its path.

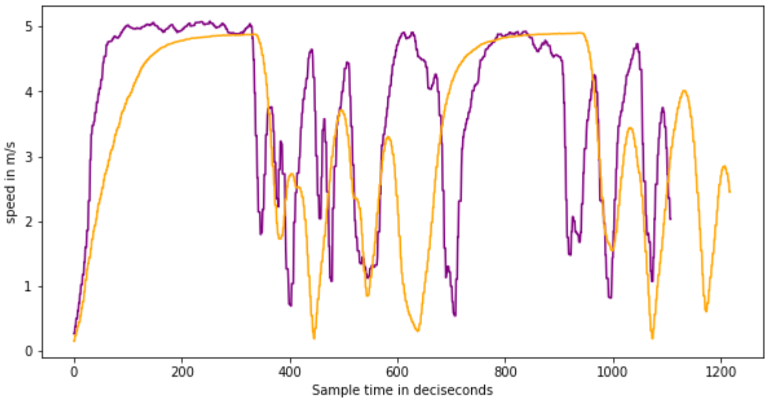


(a) Scenario 1: $5m/s$ maximum speed

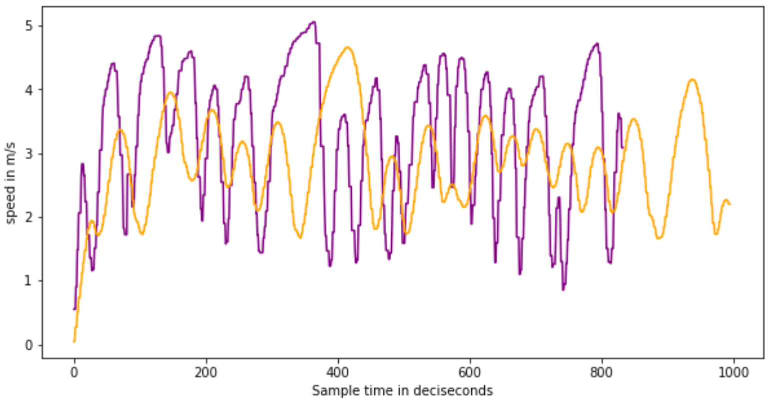


(b) Scenario 2: $5m/s$ maximum speed

Figure 6.7: Graphs illustrating the speed of SV (red) and UAV (blue) in real life experiments



(a) Scenario 1: $5m/s$ maximum speed



(b) Scenario 2: $5m/s$ maximum speed

Figure 6.8: Graphs illustrating the speed of the real drone (purple) and the simulated drone (orange) in identical scenarios

6.2.2 Path Tracing Accuracy

We use the same approach for calculating spatial similarity between the UAV and the SV, using DTW with the Haversine formula as the distance function.

Figure 6.9 and Figure 6.10 show the result of this operation for Scenario 1 and Scenario 2 respectively. Again, we observe that the drone generally follows the path of the SV well. However, as in the experiment with simulated drones , the distance between the trajectories in some segments are larger than in others. Furthermore, we observe that these are the same segments as in the simulated experiments.

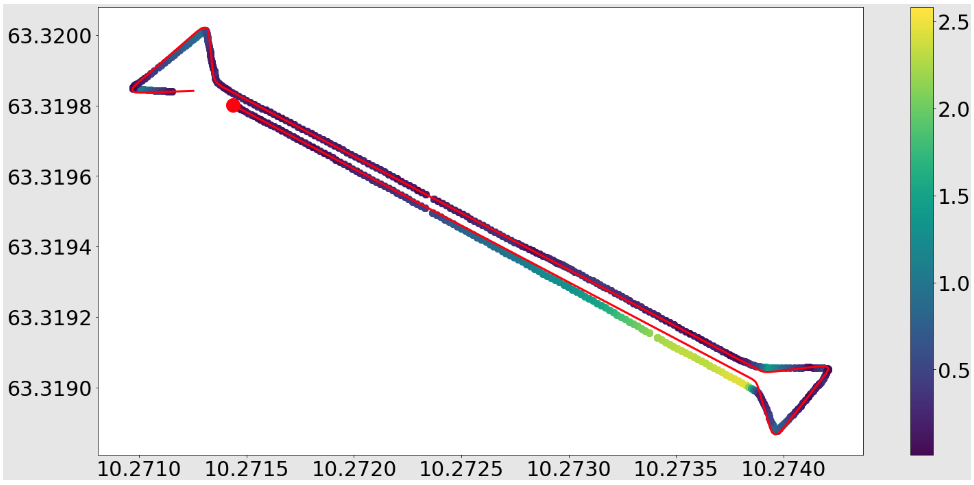


Figure 6.9: Distance measures for scenario 1

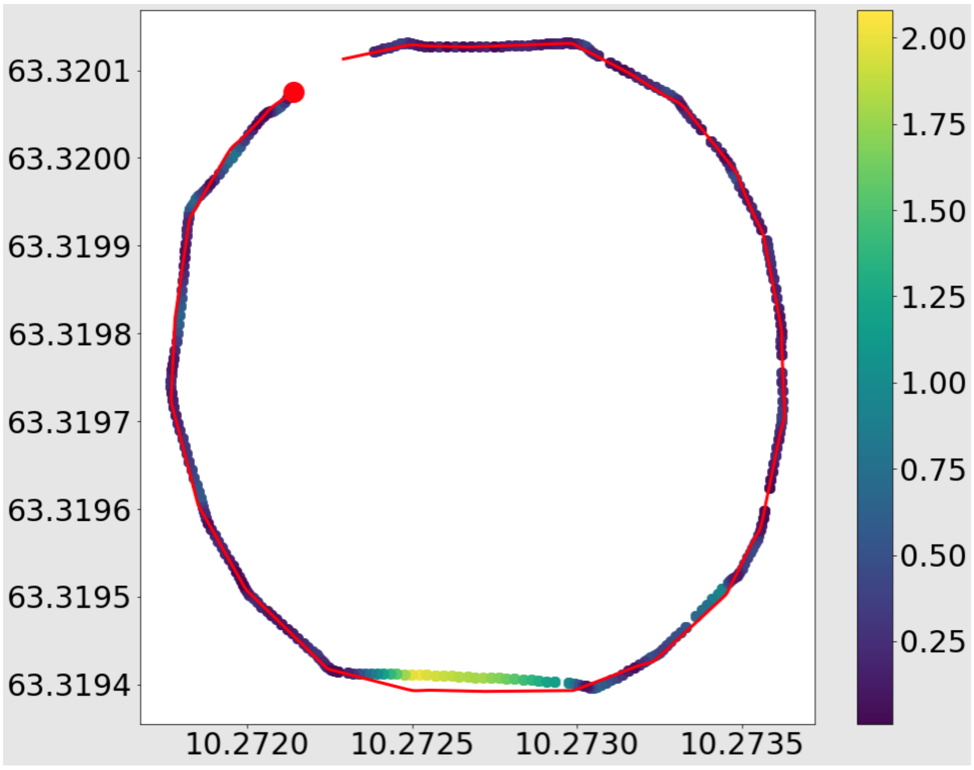
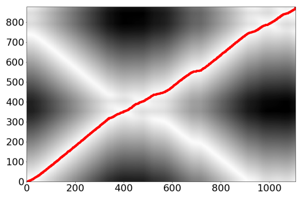
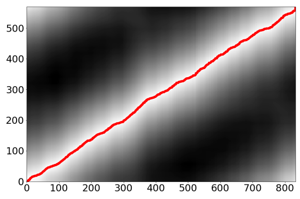


Figure 6.10: Distance measures for scenario 2



(a) Scenario 1



(b) Scenario 2

Figure 6.11: Optimal warping paths for Scenario 1 and Scenario 2, in real world experiments

Illustrated in Figure 6.11 are the distance matrices and optimal warping paths for the real world experiments. Again, we note that the optimal warping path lies along the diagonal of the distance matrix, indicating a generally good spatial alignment.

6.2.3 Packet Loss

As mentioned, for the real life experiments, we collected data on packets sent to the drone and packets received by and executed by the drone. We only counted packets containing navigation commands, that were generated by the interface in our testbed. Table 6.4 Show the packets sent and the packets received for each scenario. We experienced no packet loss for Scenario 1, and 3 lost packets for Scenario 2.

Table 6.4 Packet loss statistics

Scenario	**Packets sent by GBC**	**Packets received by UAV**
Scenario 1	116	116
Scenario 2	91	88

Chapter 7
Discussion & Conclusion

In this chapter we present our discussion, conclusion and proposal for future work. We begin by elaborating on the results obtained from the conducted experiments. Next, we relate the result to VANET and our research questions, and provide answers to them.

7.1 Experimental Results

7.1.1 Travel Time

From our experiments, we see that the drones spend more time completing a route compared to the SVs, in both simulated and real experiments. As explained in subsection 4.6.2, we had to make a trade off between speed and a drones' ability to accurately duplicate the path of the SV. This trade-of is mostly visible while turning, as the drones tend to slow down very much to be able to follow the correct path. However, we emphasize that without our solution presented in subsection 4.6.2, the performance would have been much more limited. By tuning the acceptance radius of each waypoint, we believe that we could have achieved higher speeds for the drones in all scenarios. However, this would be at the expense of how accurate they would duplicate the path of the SVs.

Another factor that affected the travel time of the vehicles has to do with acceleration. The SVs in SUMO has a higher acceleration than the drones. To make a drone accelerate at the same rate as an SV, we believe a modification of acceleration parameters in the drone would be sufficient. This can be done pre-flight, through a GCS that is connected to the drone. However, it may affect a drones ability to stay stable in the air.

Next, to make the drone be able to maintain the same speed as the SV while turning, we believe that there are two ways this can be achieved. The first one is a modification of our implementation of speed control. This may require complex changes and fine tuning for each experiment. Another way to go could to adapt the autopilot-software itself, to better mimic the behavior of cars. However this could also become rather complex.

Even though there is a notable difference in travel time and speed when comparing drones with SV, simulated and real drones behave quite similarly. The value of this is that a user of our testbed can get an accurate representation of how a real world experiment would unfold by running a simulated experiment first. However, the simulations does not account for unexpected incidents, like we saw in the real world experiment of Scenario 1. The unexpected drop in speed was caused by the drone losing a lot of altitude, and therefore reduced its speed. Instability or sensors faults may have been the cause of the altitude drop.

7.1.2 Path Tracing Accuracy

For the most part, the drones in the experiments are able to accurately duplicate the path of the SV. Having said that, in our experiments we observed a couple of segments where the drone drifted away from the path of the SV. These segments coincide in the real life tests and the simulated tests.

If we did not conduct simulated experiments, we would mention external factors like wind or GPS disturbance as a possible cause for this drift in the real world experiments. However, as the drones in the simulated scenario experienced drift in the same road segments, we consider this to be unlikely.

Another potential cause we investigated was the precision of the GPS-modules for the drones and mobility traces in SUMO. If the precision would have been low, this could be considered as a likely cause for these drifts. For the SV, the GPS-coordinates in the mobility traces was accurate to less than a millimeter. Furthermore, for the drones, both simulated and real ones, the GPS-measurements was accurate to around $1cm$. Thus, we also consider this as an unlikely cause for the drifts.

We believe that it is more likely that some part of the logic we implemented in the testbed caused the drift in these road segments.

7.1.3 Packet Loss

As presented in chapter 6, we experienced a small packet loss during the real life test of Scenario 2. We believe there may be two potential causes for this.

Firstly, it may have been caused by poor signal in the telemetry link between the GBC and UAV. Possible reasons for this include sub-optimal antenna placement on the drone, signal blockage by the drone chassis, or signal disturbance from transmitters used by other pilots in the area.

As described in subsection 5.2.2, we only experienced a maximum range of approximately 50m when using the radios, and mitigated this by walking after the drone while flying. This also includes the possibility that the drone temporarily flew out of range sometime during execution, and lost packets that way.

Although we had a limited number of real world experiments, we experienced little packet loss. However, we only logged lost packets for single-vehicle experiments. For multi-vehicle experiments, there may be more disturbance in the radio spectrum and may lead to more packet loss. We cannot say this for sure as we did not have the opportunity to test it, but consider it likely.

7.1.4 Multiple Vehicles

We were able to conduct a real world experiment with two drones, albeit it was only a partial success. However, in simulation, we conducted several experiments with up to 10 simultaneous drones, and did not encounter any issues. We argue that if we would have had more functioning drones, we would have been able to conduct successful, multi-vehicle experiments in the real world as well, because we experienced that the behavior of the drones in our simulated experiments generally coincided with drones in the real world experiments. Also, we also believe that the real world experiment with two drones would have executed successfully if the one drone did not encounter hardware errors.

In theory, our testbed should be able to support 255 simultaneous vehicles, limited by the specifications of the MAVLink protocol. In practice however, increasing the number of drones may lead to issues. In one of our real life experiments for instance, the drone experienced a unintended, rapid altitude drop. When controlling several drones at the same time, issues like this may lead to collisions and decreased the reliability of the testbed. Thus, we argue that in general, to conduct real world assessments of VANET protocols with several UAVs, with low risk of accidents and collisions, they need to be highly reliable and unlikely to encounter unforeseen events similar to the one we experienced.

On a final note we would like to emphasize that real world multi-vehicle experiments will not scale well when using the telemetry radios we used in this thesis. This is because two radios is needed for every drone, on attached the drone, and one plugged into the GBC. Therefore, we propose to use a multi-point antenna for large scale, real world experiments. In this way, all drones still need one radio each, but only one radio needs to be plugged in to the GBC.

In short, even though we have conducted successful experiments with several drones in a simulated scenario, we believe in that practice, improvements to our solution needs to be made to conduct safe and successful experiments at a large scale.

7.2 RQ1

To discuss how drones can benefit different areas surrounding testing of VANET protocols, we clarify a few things. Firsty, multi-copter drones come in all shapes and sizes, with varying capabilities. Drones can be cheap, expensive, small and large. Also, the quality of sensors and other hardware will also vary, depending on price. We do not believe that all drones are feasible in terms of replacing cars in field tests of VANET protocols. The drones used in our experiments are relatively large and quite slow. However we believe that with the correct type of drone, for the right price, they can be utilized to benefit testing of VANET-protocols in several ways.

We argue that drones can be used to reduce the costs of performing assessment of VANET-protocols in a real world environment. As ordinary field tests using cars may be expensive, currently, simulations are the most widely used method of perform such assessments. By utilizing drones, we have proposed a way for researchers developing novel rouing- and data dissemination protocols for VANET to perform assessments of them in an environment that is closer to reality than simulations, requiring relatively little equipment, and at a relatively low cost.

We also argue that drones can facilitate rapid testing. Drones being small in size and relatively light, they can be easily transported. Further, in theory, using a solution like our testbed, conducting a protocol assessment can be performed by few people and with little interaction during the execution. Still, preparation like installing firmware to drones, attaching OBUs and designing traffic scenarios can be a time consuming process.

Finally, drones can operate in three dimensions, enabling several opportunities. This can be utilized for collision avoidance by having each drone partaking in an experiment flying at different heights. It can also be used to emulate changes in road topology in the vertical space, which facilitates testing a range of different traffic scenarios. However, this property is not only positive. Adding this extra dimension means added complexity. It also introduces reliance on each drone, depending on them to maintain the correct height at all times. For tests with several drones, this may become a challenge.

However, there are also drawbacks related to using drones instead of cars for testing VANET protocols.

Firstly, the OBUs used in real field can be large and contain relatively heavy equipment. To lift heavy equipment, larger drones and larger batteries will be needed. This may drastically increase their price, making them infeasible to use. Using drones instead of cars heavily limits the maximum size of the OBU implementing a protocol to be tested. We believe that, as of today, using drones to test protocols with standard V2V communication equipment, drones may not be a feasible option. However for testing purposes, a protocol can be implemented on smaller equipment, like smart phones, to act like OBUs.

The second drawback is concerning flight time. The drones used for the experiments in this thesis had around 8 to 10 minutes of air time with a fully charged, 4000mAh LiPo battery, without any payload. This means that for long or repetitive experiments, the batteries would have to be charged or changed frequently. This may be both cumbersome and time consuming. Compared to cars, which are only limited by the size of a gas tank, drones have a long way to go in this area.

Finally in relation to safety. When flying drones with any form of automation, great care has to be taken in relation to safety. If a drone malfunctions, manual intervention may be needed to land it safely back on the ground. However, if a pilot is too slow to react, or several drones malfunction at the same time, there is little one can do to prevent hard landings, crashes or fly-aways. Not only can equipment be damaged or destroyed, but it can be dangerous to people and property that is not involved in the test. Also, if a drone is simply turned off during flight, it will fall and crash on the ground. For ground-based vehicles, simply shutting down the engine in case of malfunction will only cause the vehicles to come to a stop.

We argue that drones have the ability to mimic the behavior of vehicles. Many can travel at high speeds, rapidly turn and hover in the air while standing still. they are also relatively cheap and lightweight. These traits make them suitable for mimicking the behavior of cars and thus, be used for assessment of VANET-protocols in a real world environment. However, not being able to carry the weight of heavy equipment limits their ability to partake in tests with standard hardware used for communication in VANET. Also, batteries restrict the time that a drone can be airborne, limiting the duration of any performed test. Despite this, we argue that in this thesis, we have shown that drones can indeed be utilized as a mean of realizing simple traffic scenarios at a low costs and little intervention and that attaching OBUs turns our implementation into a testbed for VANET-protocols. We do, however, not believe that utilizing drones should, or can, replace ordinary field tests for conducting rigorous testing that is needed before commercialization of novel VANET communication protocols. Still, we believe that testbeds using drones can fill the gap between these field tests and simulated assessments, providing a more real environment than simulations while requiring less resources than ordinary field tests.

7.3 RQ2

In this thesis we have designed a testbed that projects trajectories from simulated vehicles in the real world using drones. Several ways of doing this was investigated while designing the testbed, however all candidates had three common properties.

Firstly, we argue that the mobility simulator needs to be time-discrete, time driven simulator and microscopic. time-drive and time-discrete, so we can obtain information from the simulation frequently, and microscopic, so that we can obtain granular mobility traces and other necessary data from each individual vehicle. For our testbed, necessary data include information on speed, position and direction of all simulated vehicles. Being able to extract this information from a simulation is from our experience enough to project their trajectories to drones in the real world.

The second common factor is that the drones used to project a trajectory of a simulated vehicle needs to support some level of automation. It does not need to be fully autonomous, but automatic, in the sense that it should support navigation commands from a software, and be able to execute them. We believe that apart from sensors needed to maintain balance, such as a gyroscope and accelerometer, some system for localization, like for instance GNSS, and sensors to measure altitude, like a barometer, is needed. GNSS can also provide altitude measurements, although it is less accurate.

The third and final common factor to the designs we came up with in the initial design process is that there needs to be some kind of middle-ware between the mobility simulator and drones. This is needed for mainly two tasks, the first one being to convert mobility traces into a format that can be interpreted by the drones. The second task is to control the drones and facilitate indirect communication between them. One approach for such middle-ware is the *interface* in our testbed.

We believe that as of now, these three factors are needed to project trajectories from simulated vehicles to the real world using drones. Our testbed show one approach using these three properties.

7.4 RQ3

Above, we discussed an overall approach for projecting the trajectories of simulated vehicles in the real life using drones. Now we will present and discuss how this integration can be realized in practice, using the design of our testbed as the example.

In the testbed created in this thesis, SUMO is used as the mobility simulator and TraCI provides easy access to information about each simulated vehicle. The Intel Aero RTF drones used for our experiment support a high degree of automation and can be relatively easily controlled by software. We also created a middle-ware that extracts data from the simulated environment in SUMO and uses this data to control the drones.

Because of time constraints, we did not evaluate the potential for other mobility simulators compared to SUMO. For the drones, the combination of Intel Aero RTF drones and the ArduPilot autopilot provide a relatively simple way of controlling and monitoring drones, and thus this combination was used in the experiments. Swapping the autopilot software and drone brand may lead to different results than ours.

We saw that due to the default behavior or the Intel Aero RTF drone with ArduPilot, we had to create logic in our interface, presented in subsection 4.6.2, to make the drones more accurately mimic the behavior of simulated vehicles. We see this as a workaround and not something that should be a sought after solution. We argue that a more viable, although complex, approach for handling this problem would be to create autopilot software for drones that better mimic the behavioral models of simulated vehicles. Since this changes how a drone maneuvers and accelerates, we also argue that this cannot be created at the expense of the stability of a drone while airborne.

To summarize, the testbed designed in this thesis serves as an example on how to integrate mobility simulators with drone technology to project the trajectories of simulated vehicles in the real world. Even though we argue that our testbed meets the requirements to be used for testing routing- and data dissemination protocols for VANET, we also believe that other variations of this testbed or whole new approaches can be proposed to largely enhance our results. Variations can include using another mobility simulator, other drones, or even implement a whole new middle-ware.

7.5 Summary

We argue that through our experimental results, we have shown that drones have the potential to facilitate a reduction in costs, resources and time for testing novel protocols designed for VANET and V2V communication. In our experiments, we have shown that through relatively simple software, we are able to control drones to mimic the behavior of simulated vehicles, to a certain degree. Our results show that with our solution, the drones perform well in duplicating the path of simulated vehicles, but not so well with regards to maintaining correct speed.

We have also shown that our testbed can be used to control multiple drones at the same time, an important feature of VANET. Although this was only performed successfully in a simulated environment, we believe that our real world experiment with two drones would have been successful, if none of the drones had malfunctioned.

Next, due to time constraints, we did not conduct any experiments with an OBU attached to drones, to assess the performance of an actual VANET protocol. However, we believe that it should not cause issues if the OBUs are small enough to be carried by the drones.

Finally, we argue that instead of relying on logic implemented in the middle-ware to adapt the behavior of drones to mimic cars, a more long term solution may be to create autopilot software for drones specifically designed to mimic the behavior of vehicles. This should not be at the expense of safety and drone stability while airborne.

7.6 Conclusion

In this thesis we have proposed and created a testbed for routing- and data dissemination protocols created for VANET. The solution integrates the mobility simulator SUMO and real drones to create a testbed where researchers can implement a protocol on an OBU of choice, attach them to drones, and test its performance in a variety of traffic scenarios, in a real world environment.

For **RQ1**, we believe that drones have the potential to be used as a replacement for cars when testing new routing- and data dissemination protocols for VANET in the real world, although with several limitations. we do not believe that drones have the potential to substitute field tests with standard equipment and procedures as a whole, but that they can be used as a complement to fully simulated testing, in a more realistic scenario and at a relatively low cost.

Concerning **RQ2**, we identified three elements that we believe are needed for projecting trajectories of simulated vehicles in the real world using drones. The first element is a time-driven traffic simulator with the ability to generate granular mobility traces of simulated vehicles. The second element is that the drones that are used are capable of automatic control from some middle-ware sitting between the drones and mobility simulator, which is the third element.

Finally, for **RQ3**, the testbed developed in this thesis shows one approach for realizing the three elements defined in the answer to **RQ2**. We believe that even though our integration using SUMO as a mobility simulator with drones running the ArduPilot autopilot software fulfilled our specified requirements, solutions involving other mobility simulators, other drones or another way of implementing the middle-ware could lead to better results.

7.7 Future Work

Several potential areas have been left open for future research. Future work could concern implementing a routing- or data dissemination protocol for VANET on an OBU, and perform experiments with the testbed. Then, fully simulated experiments using a mobility simulator like SUMO and a network simulator like ns-2 could be performed with the same protocol, and their results compared and analyzed. This can give insight into how well simulated experiment coincide with experiments in a real world environment.

Additionally, investigation into different approaches to make drones better mimic the speed of simulated vehicles can be of value to future expansions of our testbed, or completely new implementations. This can include deeper analysis into how drone firmware can be created or manipulated to accomplish this or how it can be done better in the middle-ware.

Unfortunately, as one of our drones malfunctioned and with no access to more drones, a real world test with multiple drones was not fully successful. Additionally, the short range of our telemetry radios limited the maximum speed on our drones. Performing real world experiments with multiple vehicles, involving higher speeds and several drones could provide even more insight into the feasibility of using drones for testing VANET-protocols in larger scenarios.

Finally, we would have liked to expand the functionality of the testbed to include support for reactive VANET-protcols. For this to be possible, OBUs would need to communicate with the drones, and the drones would have to be able to react to the input from the OBUs.

References

[Ak12] Mohammed Saeed Al-kahtani. Survey on security attacks in vehicular ad hoc networks (VANETs). In *2012 6th International Conference on Signal Processing and Communication Systems*. IEEE, dec 2012.

[ALJN19] Maytheewat Aramrattana, Tony Larsson, Jonas Jansson, and Arne Nåbo. A simulation framework for cooperative intelligent transport systems testing and evaluation. *Transportation Research Part F: Traffic Psychology and Behaviour*, 61:268–280, 2019. Special TRF issue: Driving simulation.

[ard] Ardupilot. https://ardupilot.org/ardupilot/index.html. (Accessed on 08/01/2021).

[BMM13a] Josip Balen, Josip Matijas, and Goran Martinovic. Simulation and testing of vanet protocols. *33rd Conference on Transportation Systems with International Participation Autiomation in transportation 2013*, page 4, 01 2013.

[BMM13b] Josip Balen, Josip Matijas, and Goran Martinovic. Simulation and testing of vanet protocols. *33rd Conference on Transportation Systems with International Participation Autiomation in transportation 2013*, page 4, 01 2013.

[BSK$^+$18] Dominik S. Buse, Max Schettler, Nils Kothe, Peter Reinold, Christoph Sommer, and Falko Dressler. Bridging worlds: Integrating hardware-in-the-loop testing with large-scale vanet simulation. In *2018 14th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 33–36, 2018.

[CJPZ14] Chen Chen, Yanan Jin, Qingqi Pei, and Ning Zhang. A connectivity-aware intersection-based routing in VANETs. *EURASIP Journal on Wireless Communications and Networking*, 2014(1), mar 2014.

[CMM$^+$16] Apratim Choudhury, Tomasz Maszczyk, Chetan B. Math, Hong Li, and Justin Dauwels. An integrated simulation environment for testing v2x protocols and applications. *Procedia Computer Science*, 80:2042–2052, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[CSAZ09] R. Costa, S. Sargento, R. Aguiar, and W. Zhang. Development of a hybrid simulation and emulation testbed for vanets. 2009.

[Dir] Directive 2010/40/eu of the european parliament and of the council of 7 july 2010 on the framework for the deployment of intelligent transport systems in the field of road transport and for interfaces with other modes of transporttext with eea relevance. https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2010:207:0001:0013:EN:PDF. (Accessed on 06/03/2021).

[EZL14] Elias C. Eze, Sijing Zhang, and Enjie Liu. Vehicular ad hoc networks (vanets): Current state, challenges, potentials and way forward. In *2014 20th International Conference on Automation and Computing*, pages 176–181, 2014.

[GD01] Dimitrios Gunopulos and Gautam Das. Time series similarity measures and time series indexing (abstract only). *ACM SIGMOD Record*, 30(2):624, jun 2001.

[GMSG12] George Giannopoulos, Evangelos Mitsakis, and Josep Maria Salanova Grau. Overview of intelligent transport systems (its) developments in and across transport modes. *JRC Scientific and Policy Reports*, 01 2012.

[Gro21] Halvor Groven. Master thesis VANET testbed. https://github.com/halvisg/Master_thesis_VANET_testbed, 2021. (Accessed on 08/01/2021).

[Han18] Ole Andreas Hansen. Developing a testbed for intelligent transportation systems. 6 2018.

[HKP12] Jiawei Han, Micheline Kamber, and Jian Pei. 13 - data mining trends and research frontiers. In Jiawei Han, Micheline Kamber, and Jian Pei, editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 585–631. Morgan Kaufmann, Boston, third edition edition, 2012.

[its] its-strat-2010.pdf. https://www.regjeringen.no/globalassets/upload/sd/vedlegg/rapporter_og_planer/its-strat-2010.pdf?id=2113950. (Accessed on 06/03/2021).

[JJ11] Nivedita N. Joshi and R. Joshi. Energy conservation in manet using variable range location aided routing protocol. *International Journal of Wireless Mobile Networks*, 3:261–276, 2011.

[KAA$^+$19] Anis Koubâa, Azza Allouch, Maram Alajlan, Yasir Javed, Abdelfettah Belghith, and Mohamed Khalgui. Micro air vehicle link (mavlink) in a nutshell: A survey. *IEEE Access*, 7:87658–87680, 2019.

[KHK] Tae-Hwan Kim, Won-Kee Hong, and Hie-Cheol Kim. An effective multi-hop broadcast in vehicular ad-hoc network. In *Lecture Notes in Computer Science*, pages 112–125. Springer Berlin Heidelberg.

[KJBN20] Navid Ali Khan, Noor Zaman Jhanjhi, Sarfraz Nawaz Brohi, and Anand Nayyar. Emerging use of UAV's: secure communication protocol issues and challenges. In *Drones in Smart-Cities*, pages 37–55. Elsevier, 2020.

[KJT21] Vemema Kangunde, Rodrigo S. Jamisola, and Emmanuel K. Theophilus. A review on drones controlled in real-time. *International Journal of Dynamics and Control*, jan 2021.

[KKK+14] Hyunmyung Kim, Taewon Kim, Seongjae Kang, Chongsei Yoon, and Jaeil Jung. Design of v2x runtime emulation framework for evaluation of vehicle safety applications. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*, pages 262–268, 2014.

[Lan] Kun-Chan Lan. MOVE. In *Telematics Communication Technologies and Vehicular Networks*, pages 355–368. IGI Global.

[LBBW+18] 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.

[LCQ+18] Lei Liu, Chen Chen, Tie Qiu, Mengyuan Zhang, Siyu Li, and Bin Zhou. A data dissemination scheme based on clustering and probabilistic broadcasting in vanets. *Vehicular Communications*, 13:78–88, 2018.

[LLZ+15] Wenshuang Liang, Zhuorong Li, Hongyang Zhang, Shenling Wang, and Rongfang Bie. Vehicular ad hoc networks: Architectures, research issues, methodologies, challenges, and trends. *International Journal of Distributed Sensor Networks*, 11(8):745303, aug 2015.

[MBOH14] Mohamed Nidhal Mejri, Jalel Ben-Othman, and Mohamed Hamdi. Survey on vanet security challenges and possible cryptographic solutions. *Vehicular Communications*, 1(2):53–66, 2014.

[MHP15] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6235–6240, 2015.

[MSMEB15] Nehal Magdy, Mahmoud A. Sakr, Tamer Mostafa, and Khaled El-Bahnasy. Review on trajectory similarity measures. In *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 613–619, 2015.

[MSS17] Barakat Pravin Maratha, T. Sheltami, and K. Salah. Performance study of manet routing protocols in vanet. *Arabian Journal for Science and Engineering*, 42:3115–3126, 2017.

[mul] What is a multicopter and how does it work? https://ardupilot.org/copter/docs/what-is-a-multicopter-and-how-does-it-work.html. (Accessed on 08/01/2021).

[Nic13] Mangesh Nichat. Landmark based shortest path detection by using a* algorithm and haversine formula. 04 2013.

[ns2] The network simulator - ns-2. https://www.isi.edu/nsnam/ns/. (Accessed on 08/01/2021).

[ns3] The network simulator - ns-2. https://www.nsnam.org/. (Accessed on 08/01/2021).

[Ope17] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[PRL⁺08] Michal Piórkowski, M. Raya, A. Lugo, Panos Papadimitratos, M. Grossglauser, and J. Hubaux. Trans: realistic joint traffic and network simulator for vanets. *ACM SIGMOBILE Mob. Comput. Commun. Rev.*, 12:31–33, 2008.

[RGFW15] Raphael Riebl, Hendrik-Jörn Günther, Christian Facchi, and Lars Wolf. Artery: Extending veins for vanet applications. In *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, pages 450–456, 2015.

[RPV06] S. Rajasekar, Philomi nathan Pitchai, and Chinnathambi Veerapadran. Research methodology. 01 2006.

[SA14] Surmukh Singh and Sunil Agrawal. Vanet routing protocols: Issues and challenges. In *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*, pages 1–5, 2014.

[SGD11] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing (TMC)*, 10(1):3–15, January 2011.

[SH18] Agachai Sumalee and Hung Wai Ho. Smarter and more connected: Future intelligent transportation system. *IATSS Research*, 42(2):67–71, 2018.

[SMR⁺19] Syed Sarmad Shah, Asad Waqar Malik, Anis U. Rahman, Sohail Iqbal, and Samee U. Khan. Time barrier-based emergency message dissemination in vehicular ad-hoc networks. *IEEE Access*, 7:16494–16503, 2019.

[Sto02] Ivan Stojmenović, editor. *Handbook of Wireless Networks and Mobile Computing.* John Wiley & Sons, Inc., feb 2002.

[SVB18] Zsolt Szendrei, Norbert Varga, and László Bokor. A sumo-based hardware-in-the-loop v2x simulation framework for testing and rapid prototyping of cooperative vehicular applications. In Károly Jármai and Betti Bolló, editors, *Vehicle and Automotive Engineering 2*, pages 426–440, Cham, 2018. Springer International Publishing.

[Tav] Romain Tavenard.

[TD15] Kevin Toohey and Matt Duckham. Trajectory similarity measures. *SIGSPATIAL Special*, 7(1):43–50, May 2015.

[Var10] Andras Varga. OMNeT. In *Modeling and Tools for Network Simulation*, pages 35–59. Springer Berlin Heidelberg, 2010.

[Vir] Virginia smart roads | virginia tech transportation institute. https://www.vtti.vt.edu/facilities/virginia-smart-roads.html. (Accessed on 08/28/2021).

[WCC+07] S. Y. Wang, C. L. Chou, Y. H. Chiu, Y. S. Tzeng, M. S. Hsu, Y. W. Cheng, W. L. Liu, and T. W. Ho. Nctuns 4.0: An integrated simulation platform for vehicular traffic, communication, and network researches. In *2007 IEEE 66th Vehicular Technology Conference*, pages 2081–2085, 2007.

[WFZ+16] Xiong Wang, Luoyi Fu, Yang Zhang, Xiaoying Gan, and Xinbing Wang. Vdnet: An infrastructure-less uav-assisted sparse vanet system with vehicle location prediction. *Wireless Communications and Mobile Computing*, 16, 12 2016.

[Wie14] Roelf J. Wieringa. *Design science methodology for information systems and software engineering.* Springer, Netherlands, 2014. 10.1007/978-3-662-43839-8.

[WNF21] Julia Silva Weber, Miguel Neves, and Tiago Ferreto. VANET simulators: an updated review. *Journal of the Brazilian Computer Society*, 27(1), may 2021.

[WSGY19] Jian Wang, Yameng Shao, Yuming Ge, and Rundong Yu. A survey of vehicle to everything (v2x) testing. *Sensors*, 19(2):334, jan 2019.

[YS04] Kiyoung Yang and Cyrus Shahabi. A PCA-based similarity measure for multivariate time series. In *Proceedings of the 2nd ACM international workshop on Multimedia databases - MMDB '04*. ACM Press, 2004.

[ZHC+10] Sherali Zeadally, Ray Hunt, Yuh-Shyan Chen, Angela Irwin, and Aamir Hassan. Vehicular ad hoc networks (VANETS): status, results, and challenges. *Telecommunication Systems*, 50(4):217–241, dec 2010.

Appendix A collect.py

This document contains the code for collect.py, the script used to collect data from a drone. This script can also be found in the GitHub repository that is complementary to this thesis, at [Gro21].

A.1 collect.py: Simple data collection script

```python
from dronekit import connect, VehicleMode, LocationGlobalRelative
import time
import datetime
import time
import os

veh = connect("udp:<drone_IP>:<drone_port>", wait_ready=True)

while not os.path.isfile('/tmp/start_collect'):
    time.sleep(0.2)
past_date = datetime.datetime.now()

while os.path.isfile('/tmp/start_collect'):

    loc = veh.location.global_frame
    speed = veh.groundspeed

    future_date = datetime.datetime.now()
    difference = (future_date - past_date)
    total_seconds = difference.total_seconds()

    print(str(loc) + "," + str(speed) + "," + str(total_seconds))

    time.sleep(<interval_in_seconds>)
```

Appendix B
Multi Vehicle Simulation

This document contains illustration related to simulated multi-vehicle experiments. Figure B.1 depict the architectural setup for the testbed when conducting experiments with two drones. This can easily be extended to several drones, by starting more instances of ArduPilot SITL. Then, connect QGroundControl and our interface to the MAVProxy instance that is automatically started and paired with the new ArduPilot SITL instance. Figure B.2 illustrate the traffic scenario we used when conducting simulated experiments with up to ten drones. The figure in this example illustrate a scenario with ten simulated vehicles. Each car is represented by a yellow car-shaped object.

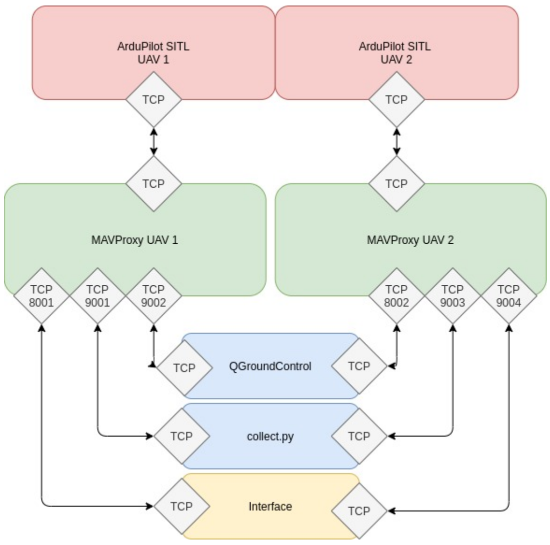


Figure B.1: Architecture for multi-vehicles experiment, two drones

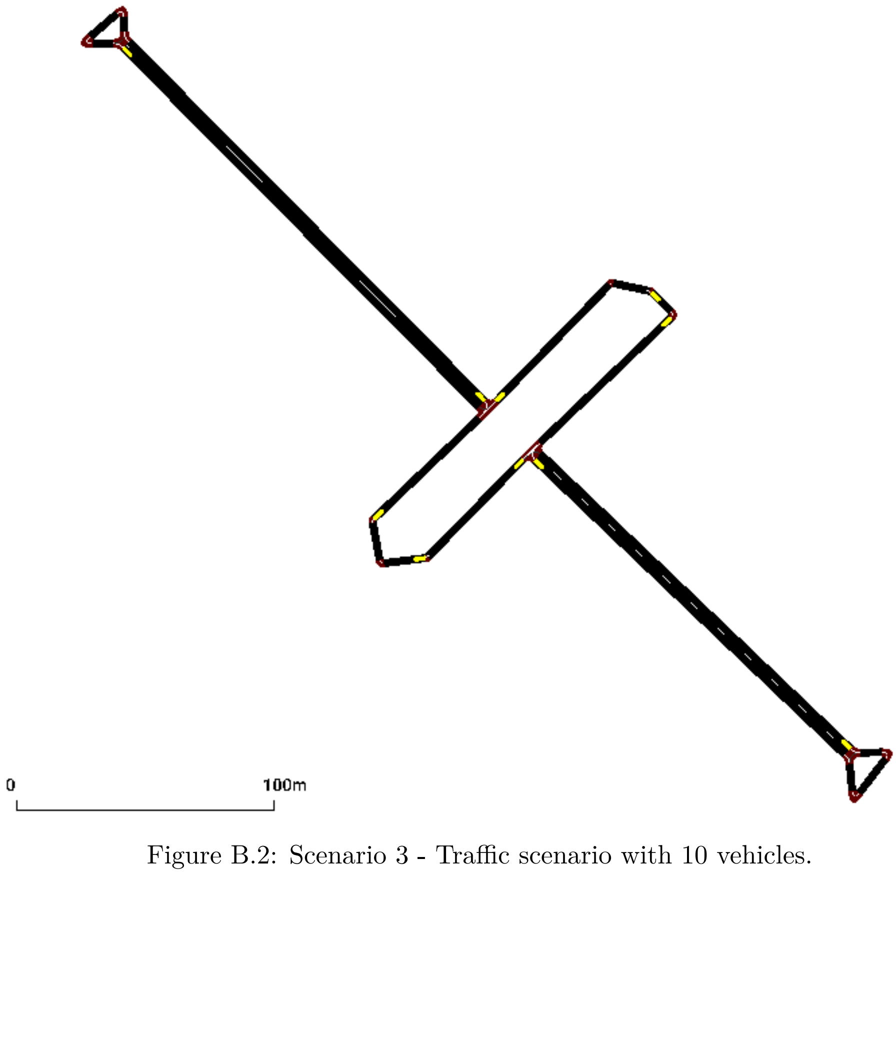


Figure B.2: Scenario 3 - Traffic scenario with 10 vehicles.

Appendix C

Real World Multi-vehicle Experiment

This document contains illustrations and images related to the real world experiment with two drones. Figure C.1 depict shows the traffic network projected onto a map of the flight area in the real world. Figure C.2 gives an overview of the equipement needed to conduct a multi-vehicle experiment with two drones, in the real world. Finally, Figure C.3 illustrate the architecture of the testbed when conducting experiments with two drones. Worth mentioning is that this architecture can be extended to more vehicles by starting more instances of MAVProxy and open more connections from QGroundControl, the interface, and collect.py to it.

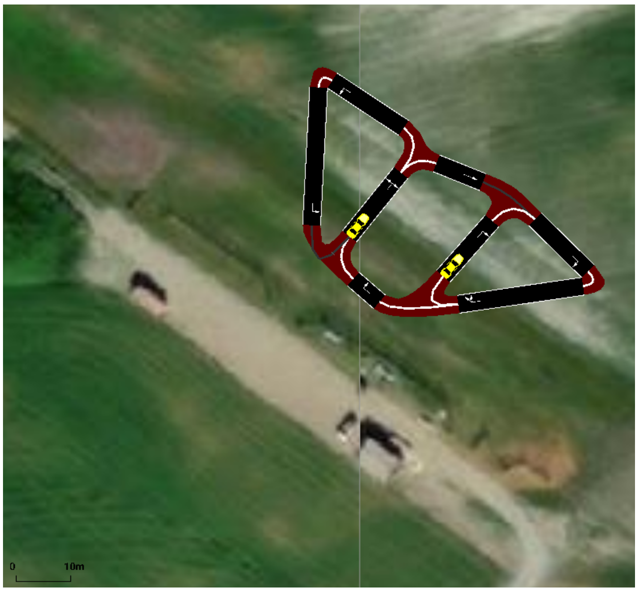


Figure C.1: Real world multi-vehicle traffic scenario

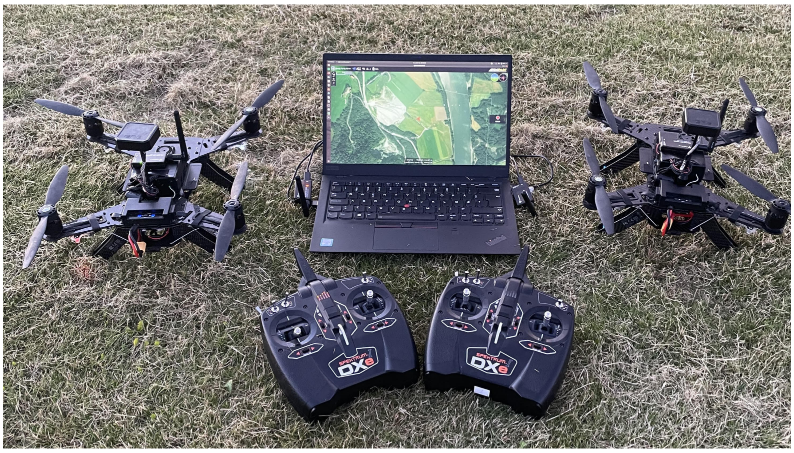


Figure C.2: Equipment used for multi-vehicle experiment

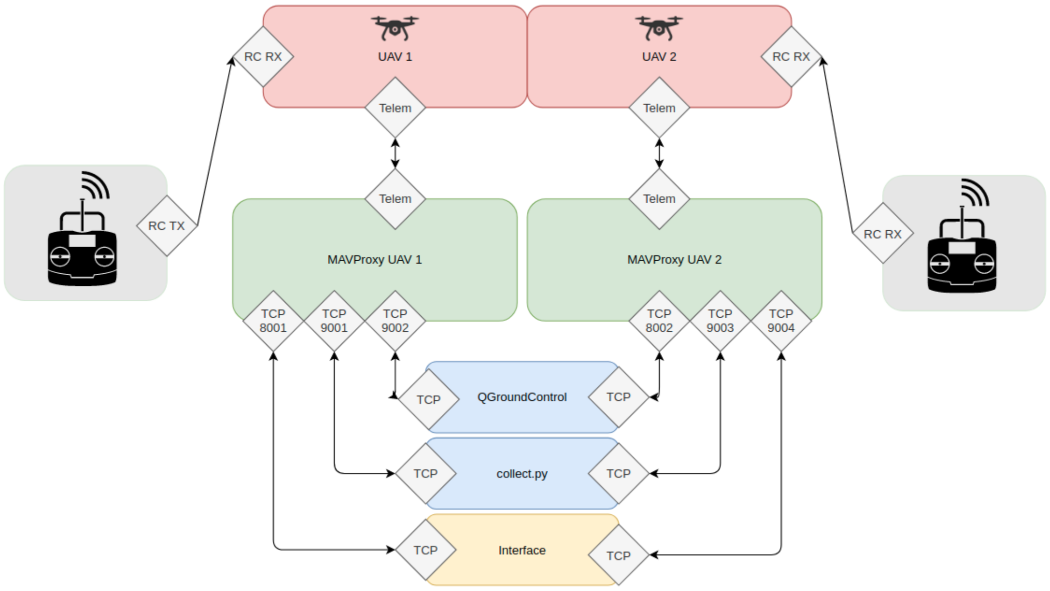


Figure C.3: Architecture for real world, multi-vehicle experiment with two drones

Appendix D Testbed Code

This document includes the code we developed for this thesis. The same code, but including documented methods, as well as installation instructions and example files to perform simulated tests with two drones have been published on GitHub and can be found on [Gro21].

D.1 drone.py

Below is the code for the Vehicle class in the testbed. The filename differs from the class name to avoid conflicts with libraries included in SUMO.

```
import math
import os
import queue
import threading
from dronekit import VehicleMode
import time


class Vehicle:

    def __init__(self, name, pairing):
        self.name = name
        self.pairing = pairing
        self.locationQueue = queue.Queue(maxsize=0)
        self.next_step = True
        self.prev_location = []
        self.set_prev_bearing = 0
        self.current_bearing = None
        self.drone_turning = False
        self.drone_default_alt = 0
        self.drone_straight_cycle = 0
        self.prev_bearing = 0


    def start(self, handler):

        thread = threading.Thread(target=self._runLocationQueue,
                                  args=(self.pairing, self.name,
    self.drone_default_alt, handler))
        handler.drone_threads.append(thread)
        thread.start()


    def _runLocationQueue(self, vehicle_obj, vehicle_id, alt, handler):

        num_packets_sent = 0
        start_point_reached = False
        self._armAndTakeoff(vehicle_obj, int(alt))
        starting_point = self.locationQueue.get()[0]
        print("Going to start")

        self._gotoStartingPoint(vehicle_obj, starting_point)

        handler.update_start_point_reached()
        while not handler.get_start_points_reached_indicator():
            time.sleep(0.1)

        start_point_reached = True

        vehicle_obj.airspeed = 10

        while not handler.simulation_end:
            if self.locationQueue.empty():
                pass

            else:
                if not start_point_reached:
                    starting_point = self.locationQueue.get()[0]
                    self._gotoStartingPoint(vehicle_obj,
    starting_point)
                    num_packets_sent += 1
                    start_point_reached = True

                prev = self.locationQueue.get()
                prev_a = prev[1]
                prev_loc = prev[0]
                if prev_a == 1:
                    vehicle_obj.simple_goto(prev_loc)
                    num_packets_sent += 1
                    a = 1
                    while a == 1 and handler.simulation_end is False:
                        cur = self.locationQueue.get()
                        a = cur[1]
                        loc = cur[0]
                        if a == 1:
                            vehicle_obj.simple_goto(loc)
                            num_packets_sent += 1
                        else:
                            self._leash(vehicle_obj, prev_loc)
                            num_packets_sent += 1
                        prev_loc = loc
                        time.sleep(0.1)
                else:
                    self._leash(vehicle_obj, loc)
                    num_packets_sent += 1

            time.sleep(0.1)

        while not self.locationQueue.empty():
            prev = self.locationQueue.get()
            prev_a = prev[1]
            prev_loc = prev[0]
            if prev_a == 1:
                vehicle_obj.simple_goto(prev_loc)
                num_packets_sent += 1
                a = 1
                while a == 1 and not self.locationQueue.empty():
                    cur = self.locationQueue.get()
                    a = cur[1]
                    loc = cur[0]
                    if a == 1:
                        vehicle_obj.simple_goto(loc)
                        num_packets_sent += 1
                    else:
                        self._leash(vehicle_obj, prev_loc)
                        num_packets_sent += 1
                    prev_loc = loc
                    time.sleep(0.1)
            else:
                self._leash(vehicle_obj, prev_loc)
                num_packets_sent += 1
                time.sleep(0.1)

        self._leash(vehicle_obj, prev_loc)
        num_packets_sent += 1

        print("Number of packets sent:" + str(num_packets_sent))

        while not vehicle_obj.mode == VehicleMode("RTL"):
            vehicle_obj.mode = VehicleMode("RTL")


    def _armAndTakeoff(self, vehicle_obj, aTargetAltitude):

        print("Basic pre-arm checks")

        while not vehicle_obj.is_armable:
            print(" Waiting for vehicle to initialise...")
            time.sleep(2)
```

```python

        print("Arming motors")

        while not vehicle_obj.armed:
            print(" Waiting for arming...")
            vehicle_obj.mode = VehicleMode("GUIDED")
            vehicle_obj.armed = True
            time.sleep(3)

        print("Taking off!")

        vehicle_obj.simple_takeoff(aTargetAltitude)

        while True:
            print(" Altitude: ", vehicle_obj.location.global_relative_frame.alt)
            if vehicle_obj.location.global_relative_frame.alt >= aTargetAltitude * 0.95:
                print("Reached target altitude")
                break
            time.sleep(1)


    def _gotoStartingPoint(self, vehicle_obj, target):

        current_location = vehicle_obj.location.global_relative_frame
        target_distance = self._get_distance_metres(current_location, target)
        vehicle_obj.simple_goto(target)

        while vehicle_obj.mode.name == "GUIDED":
            remaining_distance = self._get_distance_metres(vehicle_obj.location.global_frame, target)
            if remaining_distance <= 1:
                print(vehicle_obj.airspeed)
                break
            time.sleep(0.1)


    def _leash(self, vehicle_obj, target):

        current_location = vehicle_obj.location.global_relative_frame
        target_distance = self._get_distance_metres(current_location, target)
        vehicle_obj.simple_goto(target)

        while vehicle_obj.mode.name == "GUIDED":
            remaining_distance = self._get_distance_metres(vehicle_obj.location.global_frame, target)
            if remaining_distance <= 5:
                    break
                time.sleep(0.1)


    def _get_distance_metres(self, aLocation1, aLocation2):

        dlat = aLocation2.lat - aLocation1.lat
        dlong = aLocation2.lon - aLocation1.lon
        return math.sqrt((dlat * dlat) + (dlong * dlong)) * 1.113195e5


    def _straigth_run(self, vehicle_obj, handler):

        prev = self.locationQueue.get()
        prev_a = prev[1]
        prev_loc = prev[0]

        if prev_a == 1:
            vehicle_obj.simple_goto(prev_loc)
            a = 1
            while a == 1 and handler.simulation_end is False:
                cur = self.locationQueue.get()
                a = cur[1]
                loc = cur[0]
                if a == 1:
                    vehicle_obj.simple_goto(loc)
                else:
                    self._leash(vehicle_obj, prev_loc)
                prev_loc = loc
                time.sleep(0.1)

            self._leash(vehicle_obj, loc)
```

D.2 Handler.py

The code for the Handler is presented below.

```
import math
import queue
from dronekit import connect, LocationGlobalRelative
from numpy import arctan2
import drone


class Handler:

    start_points_reached = 0
    drones_list = queue.Queue(maxsize=0)
    sync_edges = []
    step_length = 1
    send_interval = 2
    vehicle_list = []
    drone_threads = []
    simulation_end = False
    step_length = -1
    drones_list_altitude = queue.Queue(maxsize=0)
    available_drones = 0


    def __init__(self, step_length, traci):

        self.traci = traci
        self.step_length = step_length
        self.start_points_reached = 0
        self.all_startpoints_reached = False
        with open('drones.conf', 'r') as drone_conf:
            for line in drone_conf:
                connection_string, altitude = line.split(" ")
                self.drones_list.put(connection_string.strip())
                self.drones_list_altitude.put(altitude.strip())
                self.available_drones += 1


    def update_start_point_reached(self):

        self.start_points_reached += 1
        if self.start_points_reached == len(self.vehicle_list):
            self.all_startpoints_reached = True

        else:
            self.all_startpoints_reached = False

    def get_start_points_reached_indicator(self):
        return self.all_startpoints_reached


    def _get_bearing(self, lat1, long1, lat2, long2):

        dLon = (long2 - long1)
        x = math.cos(math.radians(lat2)) * math.sin(math.radians(dLon))
        y = math.cos(math.radians(lat1)) *
    math.sin(math.radians(lat2)) - math.sin(math.radians(lat1)) *
    math.cos(
            math.radians(lat2)) * math.cos(math.radians(dLon))
        brng = arctan2(x, y)

        return math.degrees(brng)


    def _send_position(self, traci, sim_vehicle, cur_lon, cur_lat,
    straight_indicator):

        sim_vehicle.prev_location = [cur_lon, cur_lat]
```

```python
        x, y = traci.vehicle.getPosition(sim_vehicle.name)
        lon, lat = traci.simulation.convertGeo(x, y)
        current_location = LocationGlobalRelative(lat, lon,
    sim_vehicle.drone_default_alt)
        sim_vehicle.locationQueue.put([current_location,
    straight_indicator])


    def step(self, traci):

        self.traci = traci
        self.send_interval = round(float(1 / float(self.step_length)))

        arrived_vehicles = traci.simulation.getArrivedIDList()
        for arrived_vehicle in arrived_vehicles:
            for v in self.vehicle_list:
                if v.name == arrived_vehicle:
                    self.vehicle_list.remove(v)

        for sim_vehicle in self.vehicle_list:
            sim_vehicle.next_step = False
            x, y = traci.vehicle.getPosition(sim_vehicle.name)
            cur_lon, cur_lat = traci.simulation.convertGeo(x, y)
            if len(sim_vehicle.prev_location) != 0:
                prev_lon = sim_vehicle.prev_location[0]
                prev_lat = sim_vehicle.prev_location[1]

            if len(sim_vehicle.prev_location) == 0:
                sim_vehicle.prev_location = [cur_lon, cur_lat]
                sim_vehicle.next_step = True
                sim_vehicle.set_prev_bearing = True

            if not sim_vehicle.next_step:

                if sim_vehicle.prev_bearing != 0:
                    sim_vehicle.current_bearing =
    self._get_bearing(prev_lat, prev_lon, cur_lat, cur_lon)

                    if abs(sim_vehicle.prev_bearing -
    sim_vehicle.current_bearing) > 5:
                        sim_vehicle.drone_turning = True
                        traci.vehicle.setColor(sim_vehicle.name, (255,
    0, 0))
                        self._send_position(traci, sim_vehicle,
    cur_lon, cur_lat, 0)

                    else:
                        traci.vehicle.setColor(sim_vehicle.name, (255,
    0, 255))

                        if sim_vehicle.drone_turning:
```

```
                        self._send_position(traci, sim_vehicle, cur_lon, cur_lat, 1)
                        sim_vehicle.drone_straight_cycle = 0
                        sim_vehicle.drone_turning = False
                        sim_vehicle.drone_straight_cycle += 1

                    elif sim_vehicle.drone_straight_cycle % self.send_interval == 0:
                        self._send_position(traci, sim_vehicle, cur_lon, cur_lat, 1)
                        sim_vehicle.drone_straight_cycle += 1

                    else:
                        sim_vehicle.drone_straight_cycle += 1

                sim_vehicle.prev_bearing = self._get_bearing(prev_lat, prev_lon, cur_lat, cur_lon)

            else:
                sim_vehicle.prev_bearing = self._get_bearing(prev_lat, prev_lon, cur_lat, cur_lon)

    new_vehicles = traci.simulation.getLoadedIDList()

    for new_vehicle in new_vehicles:
        self.available_drones -= 1
        if self.available_drones < 0 or self.drones_list.empty():
            break
        if self.drones_list.empty():
            break

        traci.vehicle.setMaxSpeed(new_vehicle, 10)

        new_drone_connection_string = self.drones_list.get()
        new_drone_instance = connect(new_drone_connection_string, wait_ready=False)
        instance = drone.Vehicle(new_vehicle, new_drone_instance)
        self.vehicle_list.append(instance)
        instance.drone_default_alt = int(self.drones_list_altitude.get())
        instance.start(self)

        print(self.drone_threads)
```

D.3 traci-script.py

Below is the code for a bare-bones traci-script. The main job of this script is to increment the simulation step and provide the Handler with information about the simulated vehicles. This script can be extended with more functionality without affecting the logic in the testbed that control the drones.

```python
from __future__ import print_function
import time
import traci
import handler

step_length = "0.1"
sumoBinary = "<sumo-gui>" # <sumo-gui>: Path to sumo-gui binary
sumoCmd = [sumoBinary, "-c", "<sumocfg>>", "--step-length", \
    step_length] # <sumocfg>: Path to sumocfg file
traci.start(sumoCmd)
step = 0
simulation_end = False
handler = handler.Handler(step_length,traci)

while traci.simulation.getMinExpectedNumber() > 0 and step < 1000:
    handler.step(traci)
    traci.simulationStep()
    step += 1
    time.sleep(0.1)

handler.simulation_end = True
traci.close(False)
```

D.4 drones.conf

This is an example of the contents of drones.conf, the file providing the testbed with details on how to connect to each drone, and their altitude of flight. In this example, the testbed will connect to five drones, through five different, locally running MAVProxy instances.

```
#format: <connection string> <altitude>
udp:127.0.0.1:9001 5
udp:127.0.0.1:9003 10
udp:127.0.0.1:9004 3
udp:127.0.0.1:9005 6
udp:127.0.0.1:9006 12
```

Halvor Groven

Developing a Testbed for VANET Protocols using Drones

NTNU
Norwegian University of
Science and Technology