FACULTY OF ENGINEERING AND ARCHITECTURE

UAVaaS: Efficient Management of Network Flows and Cloud Infrastructures for Drone Applications

Jerico Moeyersons

Doctoral dissertation submitted to obtain the academic degree of
Doctor of Information Engineering Technology

Supervisors
Prof. Filip De Turck, PhD - Prof. Bruno Volckaert, PhD

Department of Information Technology
Faculty of Engineering and Architecture, Ghent University

March 2022

GHENT UNIVERSITY

ISBN 978-94-6355-582-1
NUR 988, 986
Wettelijk depot: D/2022/10.500/23

Members of the Examination Board

Chair

Prof. Em. Hendrik Van Landeghem, PhD, Ghent University

Other members entitled to vote

Steven Lauwereins, PhD, Televic
Prof. Hiep Luong, PhD, Ghent University
Siegfried Mercelis, PhD, Universiteit Antwerpen
Tim Wauters, PhD, Ghent University

Supervisors

Prof. Filip De Turck, PhD, Ghent University
Prof. Bruno Volckaert, PhD, Ghent University

Dankwoord

"Beste vrienden, 't is gebeurd".[1]
Met deze wijze woorden van de presentatoren van 2 van mijn favoriete programma's kan ik het laatste schrijfwerk, namelijk dit dankwoord, van mijn thesis afronden. Dit boek is het resultaat van een fantastische periode van meer dan vijf jaar bij IDLab, een periode die me voor de rest van mijn leven zal bijblijven. En dit alles kon enkel en alleen mogelijk zijn door de steun van verschillende mensen, waartoe ik me tijdens dit dankwoord graag even richt.

Eerst en vooral wil ik graag mijn promotoren, Prof. Filip De Turck en Prof. Bruno Volckaert, bedanken voor het aanreiken van de kans om dit doctoraat te mogen schrijven. Ook de vele adviezen, reviews, tips & tricks doorheen dit hele traject waren altijd zeer welkom en hebben bijgedragen tot dit mooie resultaat. Voor zowel formele, als soms ook iets minder formele meetings, kon ik steeds bij jullie terecht, op eender welk moment van de dag. Ook nog een speciale dankjewel aan dr. Pieter-Jan Maenhaut die me in het begin met raad en daad heeft bijgestaan en me de wondere wereld van cloud computing heeft doen ontdekken.

Naast mijn promotoren zijn er ook nog andere collega's die ik graag specifiek wil bedanken. Jono, bedankt om me wegwijs te maken in de wondere wereld van de drones. Tim W., bedankt voor de kritische gesprekken en brainstorms die we hadden. Alle andere collega's van IDLab, ook jullie bedankt voor de soms onverwachte en plotse babbels of de tips & tricks die uitgewisseld werden. Tot slot ook nog een welgemeende dankjewel aan alle collega's van de opleiding industrieel ingenieur informatica: Veerle, Helga, Marleen, Ann, Jan, Joris, Wim, Pieter, Sam, Miel, Leen en Brecht. De babbels en lunchpauzes tussen de lessen door waren absoluut een meerwaarde.

Nu ik toch ook over collega's bezig ben, kan er uiteraard één ding niet ontbreken. Mijn (voormalige) bureaugenoten van 200.012: Leandro, Vincent, Laurens, Cedric, Merlijn, Wim, Thijs, Sander, Tom, Stefano, Maxim en

[1] "Beste vrienden" - G. De Coster, De Mol.
"'t is gebeurd" - E. Van Looy, De slimste mens ter wereld.

Thomas. Bedankt voor de formele en informele gesprekken op kantoor en om me in het begin een beetje wegwijs te maken in de verschillende beschikbare tools en systemen. Ook de lunchpauzes, zeker als het coronabeestje nog niet bestond, waren fantastisch! Soms begonnen we tijdens deze lunchpauzes met z'n allen te twijfelen aan de realiteit van het dagdagelijkse leven, want stel je eens voor, misschien zijn wij gewoon onderdeel van een simulatie zelf?!

IDLab is een grote, geoliede machine dankzij het fantastische werk van vele mensen in de achtergrond. Ik wil dan ook graag deze mensen bedanken voor de vele ondersteuning die ik van hen ontvangen heb. Martine, Davinia en Karen, bedankt voor het helpen bij vele vragen en verzoeken die ik had gedurende de laatste vijf jaar. Ook bedank ik graag Joke, Nathalie en Mike voor alle adminstratieve hulp. Bernadette: dank je wel voor de steeds spoedige hulp bij het inbrengen van kosten. Vicent, Joeri, Sai en Brecht, bedankt voor de technische ondersteuning. Ook Sabrina wil ik graag bedanken om ons verdiep en bureau telkens kraaknet te houden en tot slot kan prof. Piet Demeester onmogelijk ontbreken in deze lijst. Vanwege zijn leiderschap loopt alles in goede banen bij IDLab.

Het onderzoek binnen dit doctoraat is vooral het resultaat van de verschillende nationale samewerkingen. Op deze manier wil ik ook graag iedereen bedanken die betrokken was de verschillende projecten zoals 5Guards, 3DSafeguard-Vl, Dynamo en Mozaik, maar zeker ook alle andere mensen met wie ik heb samengewerkt tijdens de afgelopen vijf jaar. Namen ga ik hier niet specifiek noemen om te mijden dat ik iemand zou kunnen vergeten.

De kwaliteit van dit boek is naar een hoger niveau getild dankzij de feedback van de jury-leden, die ik hierbij ook graag bedank: Prof. Hendrik Van Landeghem, Prof. Hiep Luong, dr. Siegfried Mercelis, dr. Steven Lauwereins en dr. Tim Wauters.

Tot slot wil ik ook enkele mensen uit mijn persoonlijke kring bedanken. Zonder deze mensen zou ik onmogelijk kunnen staan waar ik vandaag sta. Eerst en vooral mijn familie, en in het bijzonder mijn meter, Denise, en mijn ouders, Johan Moeyersons en Katleen Schelkens, voor de ondersteuning die ik op verschillende vlakken van jullie heb mogen ontvangen. Ook de vele leuke momenten die zorgden voor de nodige ontspanning hebben bijgedragen aan het resultaat van dit boek. Zeker niet te vergeten in dit lijstje, ook een dikke merci aan mijn broer Keoni! Nog een belangrijke toevoeging: Geert Haentjens en Ingrid Ingelrelst, mijn schoonouders in spé, dankjewel om me steeds thuis te doen voelen bij jullie, voor de support en voor de leuke babbels. Verder wil ik ook nog een aantal vrienden persoonlijk bedanken die een groot deel van dit project hebben meegemaakt: Kristof, Kiani en Emiel, merci!

Saving the best for last, als allerlaatste moet ik nog één iemand bijzonder bedanken. Mijn partner en vrouwtje in spé, Lynn Haentjens, die mijn leven zoveel meer kleur heeft gegeven. Tijdens de voorbije vijf jaar hebben we veel zaken meegemaakt, waarbij toch wel het hoogtepunt de bouw van ons huis was. Het was een zware periode, maar kijk, *we've made it!*. Daarom dus: dikke, dikke merci!
Lynn, Stella en Leonie: ik zie jullie doodgraag ♡. (Stella en Leonie zijn onze twee huiskatten, om eventuele verwarring te mijden ☺).

Gent, Maart 2022
Jerico

Table of Contents

List of Figures

List of Tables

List of Acronyms

0-9

3GPP	3rd Generation Partnership Project

A

API	Application Programming Interface
AI	Artificial Intelligence

B

BVLOS	Beyond-Visual-Line-of-Sight

C

CN	Core Network
CPU	Central Processing Unit
CLI	Command Line Interface
CRI	Container Runtime Interface

D

DSS	Digital Signage Server

G

gRPC	Google Remote Procedure Call
GUI	Graphical User Interface

H

HA	High Availability
HTTP	Hypertext Transfer Protocol

I

IaaS	Infrastructure-as-a-Service
ILP	Integer Lineair Programming
IoT	Internet-of-Things
IoD	Internet-of-Drones

J

JIM	Journey Information Manager

L

LiDAR	LIght Detection And Ranging or Laser Imaging Detection And Ranging
LP	Lineair Programming

M

MEF	Managed Extensibility Framework
MUC	Main Unit Controller
MPLS	Multiprotocol Label Switching

N

NTP	Network Time Protocol

NOS	Network Operating System
NFV	Network Function Virtualization

O

OS	Operating System
OCI	Open Container Interface
ONOS	Open Network Operating System
OXM	Openflow eXtensible Match

P

PaaS	Platform-as-a-Service
PoC	Proof-of-Concept

Q

QoS	Quality-of-Service

R

RAN	Radio Access Network
RDBMS	Relational Database Management System

S

SSE	Server-Sent Events
SaaS	Software-as-a-Service
SDN	Software-Defined Networking
SNMP	Simple Network Management Protocol
SOM	System-On-Module

T

TCMS	Train Control & Management System

TCP	Transmission Control Protocol

U

UDP	User Datagram Protocol
UART	Universal Asynchronous Receiver-Transmitter
UAV	Unmanned Aerial Vehicle
UAVaaS	UAVs-as-a-Service
URLLC	Ultra-Reliable and Low Latency Communication

V

VLAN	Virtual Local Area Network
VM	Virtual Machine
VMAMV	Version-based Microservice Analysis, Monitoring, and Visualization
VTOL	Vertical Take-Off and Landing

W

WMN	Wireless Mesh Network

Samenvatting
– Summary in Dutch –

Unmanned Aerial Vehicles (UAV's), beter bekend als drones, zijn een opkomende technologie met het potentieel om in verschillende industrieën te worden gebruikt om een breed scala aan toepassingen en diensten te bieden, zoals luchtinspecties, fotografie, pakketbezorging en zoek- en reddingsoperaties. Tijdens het laatste decennium is er een groeiende focus van het onderzoek naar de netwerk- en toepassingsparadigma's van UAV's, wat heeft geleid tot effectievere en betrouwbaardere verbindingen tussen de drone en de controller of het Internet. Het aanbod van verschillende drones met verschillende toepassingsmogelijkheden heeft echter geleid tot een langzaam en omslachtig ontwikkelingsproces voor drone-toepassingen. Omdat elke drone kleine verschillen heeft in de beschikbare hardware en de geïnstalleerde software, kan het ontwikkelproces per drone verschillen. Overstappen op nieuwe drones of het installeren van nieuwe randapparaten aan boord brengen veel complicaties met zich mee in het ontwikkelingsproces vanwege niet-compatibele hardware. Bovendien moet er veel op de drone zelf worden ontwikkeld, omdat implementatie op afstand doorgaans niet mogelijk is. Deze ontwikkeling op de drone is niet wenselijk omdat drones doorgaans maar een beperkte batterijduur hebben, wat leidt tot noodzakelijke en soms lange laadpauzes waarbij de toepassingen niet getest kunnen worden. Het gebruik van cloudtechnologieën in de ontwikkeling van drones zou het leven van ontwikkelaars van drone-applicaties gemakkelijker maken, maar een totaaloplossing die rekening houdt met het netwerk- en het applicatiebeheer van UAV's en de cloud is nog steeds een fundamentele tekortkoming.

Met de invoering van de nieuwe EASA-regelgeving voor het veilig opereren van UAV's in het Europese luchtruim, werden Beyond-Visual-Line-of-Sight (BVLOS) vluchten toegestaan, evenals andere mogelijkheden om met een UAV in de private of publieke ruimte te vliegen. Een BVLOS-vlucht vereist een stabiele, veilige en snelle netwerkverbinding met de controller, terwijl de op deze UAV ingezette applicaties soepel en zonder fouten moeten werken om een succesvolle vlucht te kunnen uitvoeren.

Cloud computing is een vast begrip in de IT-dienstverlening van vandaag, dat het optimale gebruik van datacenters, servers en edge devices (zoals computers, tablets, smartphones en smartwatches) mogelijk maakt met

behulp van virtualisatie- en containertechnologieën. Complexe berekeningen kunnen worden verspreid over meerdere apparaten en complexe toepassingen kunnen worden gebouwd en ingezet als microservices, waardoor elke dienst kan worden geschaald en responsief kan zijn in functie van het aantal verzoeken en de beschikbare middelen. Cloud computing-platformen vereisen ook complexe monitoringoplossingen om het hele systeem te observeren wat betreft de huidige toestand, de beschikbare middelen, eventuele fouten in het systeem, enz. Anderzijds zijn er 5G-netwerken die het gebruik van network slicing-concepten mogelijk maken om te voldoen aan Quality-of-Service-beperkingen, die kunnen worden geïmplementeerd door gebruik te maken van Software-Defined Networking (SDN)-oplossingen. Cloud en 5G concepten kunnen worden gebruikt om de huidige tekortkoming op te lossen om een totale UAV-bewuste cloudoplossing te bekomen.

Daarom stelt deze dissertatie het antwoord voor op de volgende vier onderzoeksuitdagingen of -doelstellingen: (i) hoe de Quality-of-Service op netwerkniveau te behouden in een UAV-bewuste context, (ii) hoe netwerkslicing met hoge prioriteit mogelijk te maken in geval van nood, (iii) hoe een beter applicatiebeheer op UAVs mogelijk te maken door gebruik te maken van cloud computing en (iv) hoe de ingezette applicaties in de cloud op een niet-indringende manier te monitoren. Deze vier doelstellingen worden beantwoord en gevalideerd doorheen dit proefschrift door middel van verschillende Proof-of-Concepts (PoCs) of prototypes.

In Hoofdstuk 2 wordt een inplugbare SDN-applicatie voorgesteld, genaamd Domino, die gedeeltelijk de uitdaging aangaat om te voldoen aan Quality-of-Service beperkingen op netwerkniveau (i). In dit hoofdstuk is Domino in staat om heterogene SDN-netwerken te beheren, waarbij de moeilijkheden van het consequent integreren van verschillende SDN-controllertypes worden opgelost. Een mogelijke standaard voor de northbound API van SDN controllers wordt voorgesteld, waardoor verschillende SDN netwerken aan elkaar gekoppeld kunnen worden. Domino is geïmplementeerd als een microservice plugin architectuur en geëvalueerd door middel van een kortste pad routing algoritme over heterogene SDN netwerken.

Hoofdstuk 3 pakt onderzoeksuitdagingen (i) en (ii) gelijktijdig aan door gebruik te maken van het framework uit Hoofdstuk 2 en een algoritme toe te voegen om de bandbreedtebeperkingen van noodnetwerkstromen te garanderen en tegelijkertijd de overige best-effort stromen te optimaliseren over de resterende beschikbare bandbreedte. Bestaande stromen in het netwerk worden geoptimaliseerd door het voorgestelde Lineair Programmeren (LP) algoritme of het Integer Lineair Programmeren (ILP) algoritme, terwijl een online aanpak nieuwe inkomende stromen in real-time behandelt tussen de berekeningen van het algoritme in. Zowel het LP- als het ILP-algoritme worden geëvalueerd door middel van simulaties en er wordt een kleiner prototype geïmplementeerd om het ILP-algoritme in combinatie met de online aanpak te demonstreren en te evalueren.

Zoals hierboven vermeld, zijn de cloud en cloud computing belangrijke concepten die in dit proefschrift zullen worden gebruikt. Daarom wordt in Hoofdstuk 4 een agent-gebaseerde, niet-intrusieve monitoring oplossing voorgesteld om complexe microservice omgevingen in de cloud te observeren, waarmee doelstelling (iv) wordt bereikt. Deze is in staat om zowel intern als extern ontwikkelde diensten te monitoren door het gebruik van sidecar containers, waarbij de status, de metrieken en het netwerkverkeer worden geobserveerd. Een prototype van de implementatie bewijst dat een complexe microservice omgeving kan worden gemonitord met een geringe impact op de gemonitorde diensten zelf.
Doelstelling (iii) wordt zowel in Hoofdstuk 5 als in Hoofdstuk 6 behandeld. Terwijl Hoofdstuk 5 de fundamenten van een UAVs-as-a-Service (UAVaaS) container platform beschrijft samen met een prototype implementatie, wordt in Hoofdstuk 6 het resultaat van dit UAVaaS platform gebruikt door het uit te breiden met een container-orchestratie platform gebaseerd op Kubernetes en beheerd met een prototype applicatie om gecontaineriseerde applicaties uit te rollen in de cloud of op de drone. Een evaluatie van een use-case met het SDN-framework dat in Hoofdstuk 3 wordt voorgesteld, bewijst dat toepassingen met succes kunnen worden ingezet op een drone of in de cloud en dat prioritering van bepaalde drones in noodgevallen kan worden toegepast.
Hoewel in de verschillende hoofdstukken de tekortkomingen van het ontbreken van een applicatie en netwerkbeheer van UAVs in de cloud grotendeels zijn opgelost, zijn er nog veel onopgeloste uitdagingen of verbeteringen. Deze uitdagingen, samen met conclusies en verdere onderzoeksrichtingen worden besproken in Hoofdstuk 7.

Summary

Unmanned Aerial Vehicles (UAVs), better known as drones, are an emerging technology with the potential to be used in industries and various sectors of human life to provide a wide range of applications and services, such as aerial inspections, photography, package delivery and search-and rescue operations. During the last decade, there has been a growing focus of research on the network and application paradigms of UAVs, resulting in more effective and trustworthy connections between the drone and the controller or the Internet. However, having all kinds of drone types with different application potential has led to a slow and cumbersome development process for drone applications. As each drone has small differences in available hardware and installed software, the development process can be different per drone. Switching to new drone types or installing new on-board add-on devices then brings many complications to the development process because of non compatible hardware. Moreover, a lot of development needs to happen on the drone itself as no remote deployment is typically possible. This on-drone development is undesirable because the drones typically have a limited battery life, leading to necessary and sometimes long charging breaks during which the applications cannot be tested. Bringing cloud technologies to drone development would ease the life of drone application developers but a total solution taking network and application management of UAVs and the cloud into account is still a fundamental shortcoming.
With the introduction of the new EASA regulations for safe operations of UAVs in the European skies, Beyond-Visual-Line-of-Sight (BVLOS) flights became allowed, as well as other opportunities for flying a UAV in private or public space. A BVLOS flight requires a stable, secure and fast network connection to the controller while the application deployed on this UAV must run smoothly and without errors in order to have a successful flight.
Cloud computing is a fixed term in the IT-services of today, allowing the optimal use of data centers, servers and edge devices (such as computers, tablets, smartphones and smartwatches) with the aid of virtualisation and container technologies. Complex calculations can be spread across multiple devices and complex applications can be built and deployed as microservices, allowing each service to be scaled and responsive in function of the number of requests and available resources. Cloud computing platforms also require complex monitoring solutions in order to observe the whole system in terms of the current state, the available resources, eventual errors in the system,

etc. On the other hand, there are 5G networks, allowing the use of network slicing concepts in order to satisfy Quality-of-Service constraints, which can be implemented by using Software-Defined Networking (SDN) solutions. Combined, cloud and 5G technology can be used to solve today's shortcoming to have a total UAV-aware cloud solution.
Therefore, this dissertation proposes to answer the following four research challenges or objectives: (i) how to maintain Quality-of-Service at network level in a UAV-aware context, (ii) how to enable high priority slicing in case of an emergency, (iii) how to allow better application management on UAVs by using cloud computing and (iv) how to monitor the deployed applications in the cloud in an unobtrusive manner. These four objectives are answered and validated throughout this dissertation by means of different Proof-of-Concepts (PoCs) or prototypes.
In Chapter 2, a pluggable SDN framework called Domino is proposed which partly tackles the challenge to satisfy Quality-of-Service constraints at network level (i). In this chapter, Domino is able to manage heterogeneous SDN networks, solving the difficulties of consistently integrating different SDN controller types. A possibility to standardize the northbound API of the SDN controllers is proposed, allowing different SDN networks to be linked together. Domino is implemented as a microservice plugin architecture and evaluated by means of a shortest path routing algorithm over heterogeneous SDN networks.
Chapter 3 tackles research challenge (i) and (ii) at the same time by using the framework provided in Chapter 2 and adding an algorithm to guarantee the bandwidth constraints of emergency network flows while optimizing the remaining best-effort flows over the remaining available bandwidth. Existing flows in the network are optimized by the proposed Linear Programming (LP) or Integer Linear Programming (ILP) algorithm while an online approach handles new incoming flows in real-time in between the calculations of the algorithm. Both the LP and ILP algorithms are evaluated by means of simulations and a smaller prototype is implemented to demonstrate and evaluate the ILP algorithm in combination with the online approach.
As stated above, the cloud and cloud computing are important concepts that will be used throughout this dissertation. Therefore Chapter 4 proposes an agent-based unobtrusive monitoring solution to observe complex microservice environments in the cloud which will address objective (iv). It is able to monitor both internally and externally developed services by the use of sidecar containers, observing state, metrics and network traffic. A prototyped implementation proves that a complex microservice environment can be monitored with a negligible impact on the monitored services itself.
Objective (iii) is tackled in both Chapter 5 and Chapter 6. While Chapter 5 describes the fundaments of a UAVs-as-a-Service (UAVaaS) container platform together with a prototype implementation. Chapter 6 uses the outcome of this UAVaaS platform by extending it with container orche-

stration based on Kubernetes and managed by a software prototype to deploy containerized applications in the cloud or on the drone. A use case evaluation with the SDN framework proposed in Chapter 3 proves that applications can be successfully deployed on a drone or in the cloud and that emergency prioritization of select drones can be applied.
Although the different chapters resolve most of the shortcomings of not having an application and network management of UAVs in the cloud, many unsolved challenges or improvements still remain. These challenges, together with conclusions and future research directions are discussed in Chapter 7.

1 Introduction

Drones, also known as Unmanned Aerial Vehicle (UAV)s have been on the rise for the last decades. Today, drones and the benefits that come with them can no longer be ignored. Their usage started in the military, then spiked with many hobbyists using them for photography, and currently, the industry adoption is going through the roof [1]. Drones can be used for wildlife surveys, fire mapping, forest health monitoring, and to monitor destructive activities, such as poaching and illegal logging [2]. In the more popular culture, drones appear in many applications such as aerial photography, express shipping and delivery, ambulance drone, thermal sensing for search and rescue operations, disaster management, and many more [3][4][5]. Drones are very useful in these use cases due to their inherent characteristics: their flying capacities let them come where machines or humans cannot, their size makes them compact to handle, they can be controlled very precisely and remotely, and they are widely available at a more and more fair price [6].

Because of their flexibility and ease of use, some issues and concerns come up as well. Often equipped with a camera, drones can easily invade people's privacy. They can be used to track license plates, follow people, look inside private homes, and more. Also, cases have been reported where drones were hacked, leading to privacy-sensitive data being leaked or malicious use of the drone [7]. Air safety concerns must be taken into account as well since drones share the airspace with other aircrafts such as airplanes, and collisions could have disastrous consequences, as shown in Figure 1.1, where a drone hit

Figure 1.1 Nose cone and radome damage to an airplane on final approach to a Mexican airport caused by drone. [8]

an airplane during his final approach to a Mexican airport [8]. To find a balance between the benefits and the issues, legislation was put into place in most countries [9][10], allowing Beyond-Visual-Line-of-Sight (BVLOS) flights, as also other opportunities for flying a UAV in private or public space. Although there has been some controversy about drones concerning privacy and safety, they have proven their value in multiple sectors.

Having all kinds of drones with different application possibilities has led to a slow and cumbersome development process for drone applications. As each drone has small differences in available hardware (e.g. regular camera, thermal camera, LIght Detection And Ranging or Laser Imaging Detection And Ranging (LiDAR)) and the installed software, the development process can be different per drone. Switching to new drones then brings many complications to the development process. Moreover, a lot of development needs to happen on the drone itself as no remote deployment is typically possible. This on-drone development is undesirable because the drones typically have a limited battery life, leading to required long charging breaks during which the applications cannot be tested. The installed software on a drone is generally rather limited, and for many applications, extra software needs to be installed. The process of installing software can be cumbersome, and it is hard to remove unnecessary software from the drone as well. Another common point of thought during drone development is the limited resources available on the drone in terms of processing power, memory and storage.

The limited resources on drones imply that one has to take into account how

applications can comply with these limitations. A solution can be to off-load certain tasks to the cloud. In the regular software industry, cloud usage and deployment have been around for quite some time. What started as simply using extra servers to strengthen and expand the available resources quickly turned into virtualizing monolithic applications in a Virtual Machine (VM) on cloud servers. More recently, a container-based approach with micro-services has proven beneficial in terms of scalability, platform portability and easy, fast and efficient application deployment. Container-based deployment systems are used in many large companies such as Netflix [11], Spotify [12], Google [13] and even The New York Times [14].

Drone communication also plays a critical role in different operations nowadays, making it important to understand the types of UAV communication. However, different wireless channels and network protocols are employed in drone communication, depending on the application and the scenario in which the drone is used. For example, in outdoor communication, it has been observed that a simple line of sight point-to-point communication link between the drone and the device can be utilized without any break in signal transmission. Another example is surveillance, where drones effectively communicate through satellite communication links. Satellite communication technique is a preferable choice for drone communication when they are used for security, defense, or more extensive outreach operations. On the other hand, for civil and personal applications, cellular communication technologies, such as 4G or 5G networks, are preferred [15].

This disseration mainly focusses on drone communication over the 5G cellular network. In comparison with the 4G network, the 5G network improves the network capacity and scale requirements (thousand-fold capacity increase with respect to 4G and the ability to provide connectivity to billions of devices) [16], but 5G also introduces tight constraints in terms of latency and reliability which are imposed by critical services, such as virtual reality office, teleprotection for smart grids, real-time remote computing for mobile terminals, and traffic safety and efficiency. These use cases are regarded as key use cases for 5G networks [17]. Together with Software-Defined Networking (SDN) technologies, the whole network architecture can be dynamically managed in terms of bandwidth allocation, delay constraints, and more [18].

This chapter is further organised as follow: in Section 1.1 a brief introduction about cloud computing is given, followed by the explenation of SDN and 5G networks in Section 1.2. Section 1.3 provides an overview of the different drone types and finally Section 1.4, 1.5, 1.6, 1.7, 1.8 and 1.9 propose the research challenges, the research questions with their hypotheses, the outline of this dissertation, the publications and the code repositories respectively.

1.1 Cloud Computing

Cloud computing implements the idea of utility computing, which was first coined by Professor John McCarthy in 1961, where computing was viewed as a public utility just as the telephone system. There is a plethora of definitions for cloud computing, from both academia and industry. Among them, Rimal et al. [19] defined cloud computing as a model of service delivery and access where dynamically scalable and virtualized resources are provided as a service over the Internet. Cloud computing provides a paradigm shift of business and IT, where computing power, data storage, and services are outsourced to third parties and made available as commodities to enterprises and customers. Cloud computing is a center point for the most highly impactful technologies such as mobile Internet, Artificial Intelligence (AI), Internet-of-Things (IoT), and Big Data. Further, cloud technology can offer tremendous economic benefits. For example, the total economic impact of cloud technology could be $1.7– $6.2 trillion annually in 2025, and the proliferation and further sophistication of cloud services could become a major driving force in making entrepreneurship more feasible in the coming decade [20].
Cloud service models can be classified into three groups: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS):

1. Software-as-a-Service (SaaS): SaaS, commonly referred to as the Application Service Provider model, is a way of delivering centrally hosted applications over the Internet as a service. It shares common resources and a single instance of both the object code of an application as well as the underlying database to support multiple customers simultaneously.

2. Platform-as-a-Service (PaaS): The idea behind PaaS is to provide developers with a platform including all the systems and environments comprising the end-to-end life cycle of developing, testing, deploying, and hosting of sophisticated web applications as a service delivered by a cloud. It provides an easier way to develop business applications and various services over the Internet. PaaS can slash development time and offer hundreds of readily available tools and services compared to conventional application development.

3. Infrastructure-as-a-Service (IaaS): IaaS is the delivery of resources (e.g., processing, storage, networks) as a service over the Internet. Aside from the higher flexibility, a key benefit of IaaS is the usage-based payment scheme. This allows IaaS customers to pay as they

grow.

This dissertation proposes a UAVaaS group, which is actually a subgroup of the IaaS group. It delivers the use of different drones as a service over the Internet, along with resources in the cloud itself.
The deployment modes can be categorized into three groups: Public Cloud, Private Cloud, and Hybrid Cloud:

1. Public Cloud: It describes the cloud computing in the traditional mainstream sense, whereby resources are dynamically provisioned on a fine-grained, self-service basis over the Internet, via web applications, from an off-site third-party provider who shares resources.

2. Private Cloud: Data and processes are managed within the organization without the restrictions of network bandwidth, security exposures, and legal requirements that using public cloud services across open, public networks might entail.

3. HybridCloud: The environment is consisting of a combination of private and public cloud provisions.

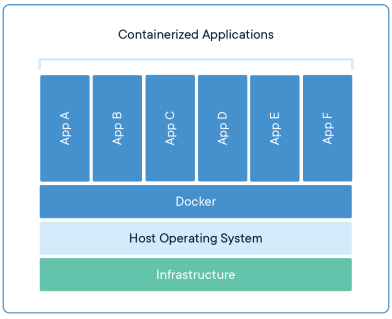
To summarize, in cloud computing, resources are provided to users over the Internet. These cloud resources act as a highly variable and scalable virtual environment for the user to deploy their applications to. The cloud's physical hardware or software can be located anywhere from a geographical point of view. Moreover, the used infrastructure is, to a certain extent, irrelevant to the user [21]. This can be achieved by using VMs or container technologies, with the accompanied orchestrators, which are explained below.

1.1.1 Containers vs. Virtual Machines

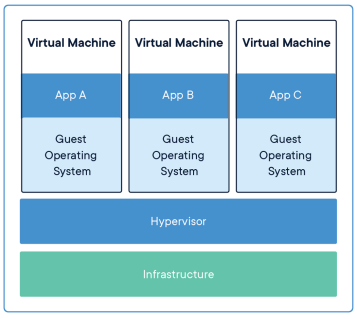
Following Red Hat's definition, a VM is a virtual environment that functions as a virtual computer system with its own Central Processing Unit (CPU), memory, network interface, and storage, created on a physical hardware system. The hypervisor is the piece of software responsible for separating the machine's resources from the hardware and provisions them appropriately so they can be used by the VM [23]. A server can have multiple VMs that use the server's hardware. This way, the capacity of a server farm can be optimally used. VMs can be initialized on a server until the server's capacity is fully occupied, and other servers should only be activated if more resources are necessary.
Containers are another form of virtualization. A container provides virtualization on the Operating System (OS) level. The host's kernel provides

Figure 1.2 Containers versus virtual machines [22]

(a) Container overview



(b) Virtual machine overview



the possibility to create multiple user-space instances that can be used to run containerized applications. This, in contrast to a VM that emulates a complete operating system (1 system per VM) where applications run on top of these emulated operating systems, as illustrated in Figure 1.2 [22]. Since the virtualization happens at the OS-level instead of the hardware level, there is less overhead involved. On top of that, containers do still provide many of the benefits that come with VMs such as security, decoupling from the hardware and separation of concerns.

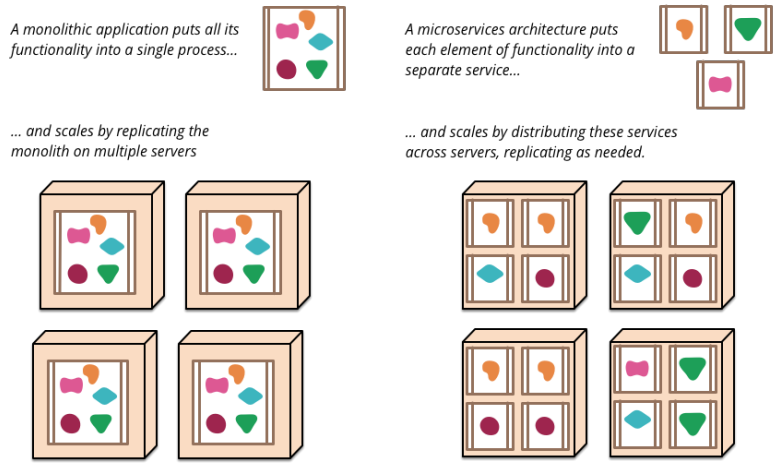
1.1.2 Container orchestration

Internet products or services that follow a container approach often exist out of many of these containers that interact with each other. One container can be replicated multiple times to ensure availability or to split the workload. For larger systems, the containers can be deployed on different hardware for multiple reasons:

- In case of hardware failure, the containers are moved to another server.
- In case of temporary high workload, the containers are replicated and started on multiple servers to ensure all requests are taken care of.
- The services provided by a container are used locally by users with different consumer hardware.
- Systems can easily be switched from cloud provider without worrying about hardware compatibility.

These features call for a way to manage the containers. Container orchestration is precisely that. It automates the deployment of containers to provide scaling possibilities and general container management. Well-known

Figure 1.3 Monoliths and Microservices[26]



examples of container orchestrators are Docker Swarm [24] and the industry de-facto standard Kubernetes [25].

Therefore, this dissertation proposes a container-based deployment system for drones, unlocking container technologies for drone application development orchestrated by a Kubernetes-compatible framework handling a range of (micro)service-based application components both on a drone fleet and in the cloud backend.

1.1.3 Microservices

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an Hypertext Transfer Protocol (HTTP) resource Application Programming Interface (API) [26]. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. Microservice based design is in sharp contrast to monolithic software design, where all application functionality resides in one big inseperable component. An illustration about monolithic applications and microservices is shown in Figure 1.3.

1.2 5G and SDN Networks

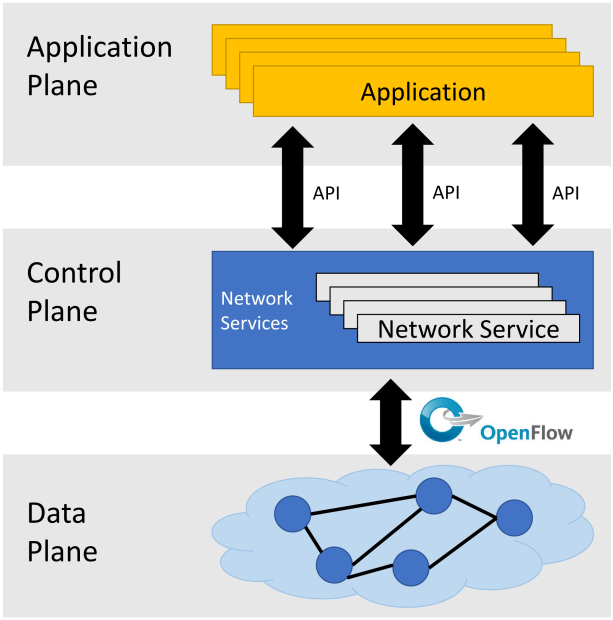
SDN is an important technology to realize dynamic and flexible network management which overcomes the current shortcomings of traditional networks, such as complex design and complex configuration [27, 28]. A schematic overview of SDN is shown in Figure 1.4 and further explained. With SDN, the control plane, that decides how to handle network traffic, and the data plane, that forwards traffic according to the decision made by the control plane, are separated. Due to the separation of the control and data planes, network switches in the data plane become simple packet forwarding devices instructed by the rules created by the SDN controller application in the control plane. The data plane always consists of the southbound API, which is further explained below, and the network infrastructure. The SDN controller (also called the Network Operating System (NOS)) and the corresponding northbound API are part of the control plane. A network hypervisor can eventually be added. Finally, the application plane bundles the different possible network applications who implement the control-logic that will be translated into commands to be installed in the data plane. These applications can perform traditional functions such as routing, load balancing and security but they can also explore novel approaches such as reducing power consumption. [29] The above-mentioned features of SDN are harder to achieve in traditional networks due to the complex network control and the different protocols. Also the limited testing environments and the long standardization process are avoided when using SDN.

Communication between the control plane and the data plane, the southbound API, follows the OpenFlow protocol in most cases [30, 31], giving the control plane access to network switches and routers in the data plane. OpenFlow allows switches from different vendors, each having their own interfaces, to be managed remotely using a single, open protocol. The OpenFlow protocol has evolved from version 1.0 which supported only 12 fixed match fields and a single flow table, to version 1.5.1 (released on March 26, 2015) which features multiple tables, more than 41 matching fields and several new functions [32] such as the addition of egress tables, enabling processing to be done in the context of the output port instead of in the context of the input port. The business applications on top of the control plane communicate over a custom northbound API.

The 5G network uses the principles of SDN networks for the management of the wired structure, while similar solutions are used for the wireless part (the so called Radio Access Network (RAN) slicing).

This dissertation proposes a solution for generalizing the nortbound API within different SDN networks (and 5G networks) together with the use of

Figure 1.4 Schematic overview of Software-Defined Networking planes[33]



these principles to create high priority flows in order to achieve Quality-of-Service (QoS) constraints at network level for emergency use cases where network traffic dealing with the incident at hand (e.g. emergency service communication) should be prioritised over other network traffic.

1.3 Drones

A drone or UAV is an aircraft without a human pilot on board. Drones exist in many shapes and sizes. Where the military UAVs are typically quite large and have a large range combined with top-notch technology, the hobbyist UAVs are typically smaller and come with a shorter range of use [1]. In the literature, drones are being classified in 4 categories: multi-rotor drones, fixed-wing drones, single-rotor drones and fixed-wing hybrid Vertical Take-Off and Landing (VTOL) drones [34], as illustrated in Figure 1.5.

In the context of this dissertation, one can think of the typical quadcopter multi-rotor drone with an onboard computer and autopilot. The most famous brand in drone retail is DJI [35] with, e.g. their Phantom and Mavic series as illustrated in Figure 1.6

Figure 1.5 Drone categories [34]

(a) Multi rotor drone

(b) Fixed wing drone

(c) Single rotor drone

(d) Fixed wing hybrid VTOL

Figure 1.6 DJI models [35]

(a) DJI Mavic series

(b) DJI Phantom series

1.4 Research Challenges

Based on the shortcomings of the current state of the art and the identified gaps discussed in the previous sections, we present four obstacles which we will tackle in this dissertation.

C1: Quality-of-Service at network level in a UAV-aware context. When flying a drone, a reliable network connection to the drone is required, especially during BVLOS drone flights. An objective of this dissseration is to find a solution in order to maintain the QoS constraints at network level.

C2: High priority slicing. In case of an emergency, the network connection to a drone handling this emergency use case must be prioritized above other network connections. Based on the first objective (**C1**), this disseration will prove such high priority slicing is possible and the other remaining network connections will be allocated on a best-effort basis.

C3: Drone application management. Application development and management for drones is nowadays a struggle for developers. Therefore, a solution is proposed that brings cloud technologies and concepts to drone application development and management, with the use of container technologies and container orchestrators.

C4: Unobtrusive container monitoring. When a container application is deployed in the cloud, monitoring can improve the reliability and allow users to fix new issues as soon as possible. This must be done in an unobtrusive manner in order to avoid impact on the observed services. This dissertation proposed such a new monitoring system to overcome this issue.

1.5 Research Questions & Hypotheses

In order to overcome the obstacles discussed in Section 1.4, we form several hypotheses which will be evaluated throughout this dissertation.

1.5.1 Bringing Quality-of-Service at network level with SDN technologies

As discussed in Section 1.2, SDN networks decouple the control plane from the network plane, allowing dynamic and flexible network management. Multiple SDN-controllers can be used in a network, avoiding a single point of failure, with the requirement that every controller must run the same software. When flying a drone, there is the possibility that a connection must be made with other networks not running the same SDN controller software. Therefore, we hypothesize that a solution can be found to manage heterogeneous SDN networks and apply the QoS constraints within these different networks.

RH1: Managing heterogeneous SDN networks while maintaining QoS constraints should be feasible with a negligible overhead on the performance of the network.

1.5.2 Enable high-priority slicing in UAV-aware SDN networks

In case of an emergency, the network traffic from and to certain devices must be prioritized over the other, remaining network traffic. In terms of UAVs, this means that a drone which is handling an emergency situation (e.g. streaming a video feed from the incident area) must have priority over the other network traffic. This other network traffic however cannot be ignored and should be divided and allocated over the remaining network resources.

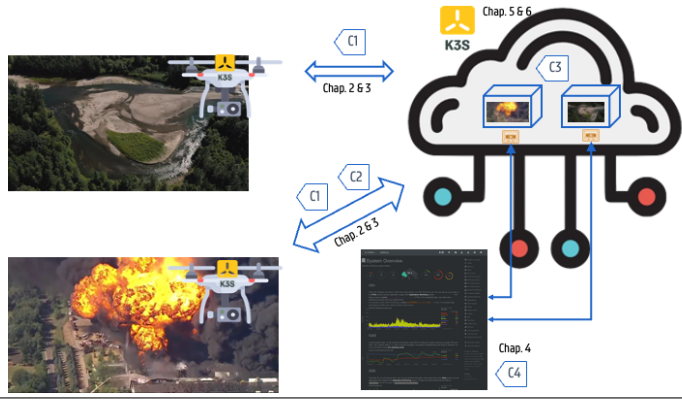
RH2: Ensure high-priority network traffic and optimize the remaining network traffic over the remaining resources.

1.5.3 Cloud-based drone application development and management

When having multiple drones in operation, each drone will be equipped with a specific application to run. This application can send a video feed to the cloud, capture sensor data from equipped sensors on the drone, execute on-board preprocessing (e.g. preliminary object detection on a video feed without relying on cloud-based analysis) and many other possibilities. At the moment, it is not possible to change these applications during flight and every drone can have slightly different on-board hardware making it difficult to change the application on each drone. Therefore, a container-based solution must be introduced allowing applications to be built and deployed as containerized applications, allowing easy management and development.
Orchestration of these containers is also possible in order to integrate drones with services running in a cloud backend. Therefore, a Kubernetes-like solution must be investigated that allows UAV-aware cloud applications.

RH3: Design and develop a UAV-aware cloud solution with minimal resource impact on the drones. This solution must be able to migrate containerized application components from the drone to the cloud and vice versa from an easy-to-use web platform.

Figure 1.7 Dissertation overview



1.5.4 Unobtrusive monitoring in cloud based environments

When a larger application is deployed in the cloud, this is mostly done as a microservice environment because of the many advantages this brings such as better scaling options. However, these microservice environments may also rely on third-party services that in turn may also introduce errors due to undocumented code changes or updates. Therefore, a monitoring solution must be introduced that can observe the microservice itself but also the integrations with any third-party services. This must be done with a negligible impact on the monitored services itself.

RH4: Design and development an unobtrusive monitoring solution that has a negligble impact on the monitored services.

1.6 Dissertation Outline

This doctoral dissertation is comprised of 7 chapters, including an introduction and conclusion. The other 5 chapters are research focused and some are inspired by research projects, summarized in Section 1.7. In what follows, we briefly introduce the contents of each chapter with an overview of how each of the chapters maps to the different provided objectives. An overview of the chapters with the mapping of the Research Challenges is provided in Figure 1.7.

Chapter 2 – Pluggable SDN Framework for Managing Heterogeneous SDN Networks

In Chapter 2, the Domino framework is proposed, a pluggable SDN framework for managing heterogeneous SDN networks, which allows research into SDN networks controlled by different SDN controllers. Domino is deployed as a microservice architecture and it tackles **C1** and **RH1**. With the outcome of this research, a heterogeneous SDN network can be controlled, allowing to control the QoS constraints within the network.

Chapter 3 – Towards Distributed Emergency Flow Prioritization in SDN Networks

With the outcome of Chapter 2, Chapter 3 and Appendix A adds an approach to guarantee the bandwidth allocation for emergency network flows while optimizing the allocation of the remaining bandwith over the other flows. This approach, designed as joint online-offline, consists of a Lineair Programming (LP) and Integer Lineair Programming (ILP) algorithm (the offline side) and an online algorithm. Where the offline approach handles all the known network flows as a batch, the online approach will handle all the new incoming flows, tackling **C1**, **C2** and **RH2**.

Chapter 4 – Towards cloud-based unobtrusive monitoring in remote multi-vendor environments

Objective **C4** and hypothesis **RH4** are both handled in chapter 4. This chapter proposes a monitoring framework to observe complex microservice environments and the corresponding third party services it relies on in an unobtrusive manner. Advanced service monitoring can offer a solution to quickly detect anomalies and possible "upcoming errors" in order to resolve these errors as quickly as possible.

Chapter 5 – UAVs-as-a-Service: Cloud-based Remote Application Management for drones

Chapter 5 presents the fundamental research required in Chapter 6 to solve objective **C3** and **RH3**. It presents a container-based deployment system for drones called UG-One, bringing cloud technologies to drone application development and management. By using containerized application workloads, drone development has become much easier and a new IaaS platform is thus created, called UAVs-as-a-Service (UAVaaS).

Chapter 6 – Towards a cloud-based drone application management platform in emergency situations

As stated above, Chapter 6 builds upon the outcome of Chapter 5, bringing also container orchestration capabilities to UAV-aware clouds. A Kubernetes based approach, called K3S, is used as container orchestrator due to the low impact it has on the edge device (e.g. drones). From this moment, every drone is fully integrated into the cloud and container-based applications can be deployed on either the cloud or on a drone. The networking side within K3S is implemented with the outcome of Chapter 2 and 3 and hereby concludes the work of this dissertation.

1.7 Research Projects

Parts of the research conducted in this dissertation are the result of different national research projects which are listed below:

5Guards (imec.icon): In this project, the aim was to investigate how the network slicing concept of future 5G networks could provide the means to meet the connectivity requirements of security services. Network slicing aims at providing the ability to allocate resources on demand by creating multiple isolated logical networks on top of a common shared physical infrastructure. The focus lied primarly on emergency or operational use cases for 5G network slicing in the public (e.g. fire departments) and industrial sectors. A major asset of the project is the involvement of multiple industrial and academic players who cover the complete 5G network – from the radio link over the radio access network to the core network.[1]

3DSafeGuard-VL (ITEA3): This project focused on research activities which resulted in the development of a Proof-of-Concept (PoC) of a new UAV-platform. This platform can be used by emergency workers to increase in real time the global situational awareness and support decision processes of an active emergency situation. Our main task in this project was to collect the on-board sensor data of the UAV which is used to create a map in real time and distributed an annotate information on that map towards creating information to advice and support decisions of the officers and hazard workers on site.[2]

[1] https://www.imec-int.com/en/what-we-offer/research-portfolio/5guards

[2] https://droneport.eu/flemish-consortium-demonstrates-unique-drone-technology-for-emergency-services/?lang=en

DynAMo (imec.icon): DynAMo seeks to reduce integration problems and achieve faster go-live of software deployments by giving companies more insights into their own IT service offering, and also into third-party services interacting with it. The project developed a dynamic architecture monitoring solution that learns normal service behavior from data and detects when services are not operating in line with the rules governing interfaces, protocols, performance and security.[3]

1.8 Publications

The results obtained during this PhD research have been published in scientific journals and presented at different international conferences and workshops. Moreover, different talks were given to different news outlets over the course of this PhD for the broader public. The following list provides an overview of these publications and talks.

1.8.1 Publications in A1 Journals

[1] **J. Moeyersons**, PJ. Maenhaut, F. De Turck and B. Volckaert *Pluggable SDN framework for managing heterogeneous SDN networks.* Published in the International Journal of Network Management, vol. 30, no. 2, e2087, 2020.

[2] **J. Moeyersons**, B. Farkiani, T. Wauters, B. Volckaert and F. De Turck *Towards distributed emergency flow prioritization in software-defined networks.* Published as special issue paper in the International Journal of Network Management, vol. 31, no. 1, e2127, 2021.

[3] **J. Moeyersons**, S. Kerkhove, T. Wauters, F. De Turck and B. Volckaert *Towards cloud-based unobtrusive monitoring in remote multi-vendor environments.* Published in Software: Practice and Experience, 2021.

[4] **J. Moeyersons**, M. De Schutter, F. De Turck and B. Volckaert *Towards a cloud-based drone application management platform in emergency situations.* Submitted for review, 2022.

1.8.2 Publications in International Conferences

[1] **J. Moeyersons**, PJ. Maenhaut, F. De Turck and B. Volckaert *Aiding first incident responders using a decision support system based on live*

[3] https://www.imec-int.com/en/what-we-offer/research-portfolio/dynamo

drone feeds. In Jian Chen, Y. Yamada, M. Ryoke, & X. Tang (Eds.), Knowledge and Systems Sciences (pp. 87-100), 2018, Singapore: Springer Singapore.

[2] **J. Moeyersons**, B. Verhoeve, PJ. Maenhaut, B. Volckaert and F. De Turck *Pluggable drone imaging analysis framework for mob detection during open-air events.* In ICPRAM2019, the 8th International Conference on Pattern Recognition Applications and Methods (pp. 64-72).

[3] **J. Moeyersons**, B. Farkiani, B. Bakhshi, S. A. Mirhassani, T. Wauters, B. Volckaert and F. De Turck *Enabling emergency flow prioritization in SDN Networks.* In 15th International Conference on Network and Service Management (CNSM). IEEE, 2019. p. 1-8.

[4] **J. Moeyersons**, M. Gevaert, K. Réculé, B. Volckaert and F. De Turck *UAVs-as-a-service : cloud-based remote application management for drones.* In Manage-ioT2021, part of IM2021, the IFIP/IEEE Symposium on Integrated Network and Service Management. 2021. p. 1-6.

1.8.3 Talks

[1] Belgian Drone Federation. *Webinar: Innovatie met drones.* Webinar. June 2021.
URL: `https://belgiandronefederation.be/nl/news/webinar-1-juni-2021-innovatie-met-drones`.

[2] Drone-Days 2021 - Fly to Adventure. *Digital Convention: Drones @ Work.* Conference. November 2021.
URL: `https://www.drone-days.be/en/drone-days`.

1.9 Code Repositories

We advocate for open and reproducible research. The following list provides an overview of all public code repositories.

[1] The Last Post Thermal Dataset
URL: `https://github.com/IBCNServices/Last-Post-Dataset`.

[2] UG-ONE
URL: `https://github.com/IBCNServices/UG-One`.

Bibliography

[1] Kashyap Vyas. A brief history of drones: The remote controlled unmanned aerial vehicles (uavs), Jul 2020. URL https://interestingengineering.com/a-brief-history-of-drones-the-remote-controlled-unmanned-aerial-vehicles-uavs.

[2] Kurt W Smith. Drone technology: Benefits, risks, and legal considerations. *Seattle J. Envtl. L.*, 5:i, 2015.

[3] Drone technology uses and applications for commercial, industrial and military drones in 2021 and the future, Jan 2021. URL https://www.businessinsider.com/drone-technology-uses-applications?international=true&r=US&IR=T.

[4] About the AuthorJosh Pozner, About the Author, and Josh Pozner. A comprehensive list of commercial drone use cases (128 and growing), Nov 2020. URL https://www.dronegenuity.com/commercial-drone-use-cases-comprehensive-list/.

[5] P Anand, P Arjun, N Bharath Kumar, and K Gowtham. Drone ambulance support system. *International Journal of Engineering and Techniques*, 4(2), 2018.

[6] Francesco Castellano. Commercial drones are revolutionizing business operations, Oct 2017. URL https://www.toptal.com/finance/market-research-analysts/drone-market.

[7] Nigel McKelvey, Cathal Diver, and Kevin Curran. Drones and privacy. *International Journal of Handheld Computing Research*, 6:44–57, 01 2015. doi: 10.4018/IJHCR.2015010104.

[8] Aeromexico boeing 737-800 hits drone during approach; damage to nose cone and radome - aviation24.be. URL https://www.aviation24.be/airlines/aeromexico/boeing-737-800-hits-drone-during-approach-damage-to-nose-cone-and-radome/.

[9] Easy access rules for unmanned aircraft systems (regulation (eu) 2019/947 and regulation (eu) 2019/945), Jan 2021. URL https://www.easa.europa.eu/document-library/easy-access-rules/easy-access-rules-unmanned-aircraft-systems-regulation-eu.

[10] Feb 2021. URL https://www.faa.gov/uas/.

[11] Netflix Technology Blog. Titus, the netflix container management platform, is now open source, Apr 2018. URL

https://netflixtechblog.com/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436.

[12] Spotify case study, Sep 2020. URL https://kubernetes.io/case-studies/spotify/.

[13] Kubernetes. URL https://cloud.google.com/kubernetes-engine.

[14] New york times case study, Nov 2020. URL https://kubernetes.io/case-studies/newyorktimes/.

[15] Abhishek Sharma, Pankhuri Vanjani, Nikhil Paliwal, Chathuranga M.Wijerathna Basnayaka, Dushantha Nalin K. Jayakody, Hwang-Cheng Wang, and P. Muthuchidambaranathan. Communication and networking technologies for uavs: A survey. *Journal of Network and Computer Applications*, 168:102739, 2020. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2020.102739. URL https://www.sciencedirect.com/science/article/pii/S1084804520302137.

[16] Qian Clara Li, Huaning Niu, Apostolos Tolis Papathanassiou, and Geng Wu. 5g network capacity: Key elements and technologies. *IEEE Vehicular Technology Magazine*, 9(1):71–78, 2014. doi: 10.1109/MVT.2013.2295070.

[17] Afif Osseiran, Federico Boccardi, Volker Braun, Katsutoshi Kusume, Patrick Marsch, Michal Maternia, Olav Queseth, Malte Schellmann, Hans Schotten, Hidekazu Taoka, et al. Scenarios for 5g mobile and wireless communications: the vision of the metis project. *IEEE communications magazine*, 52(5):26–35, 2014.

[18] Riccardo Trivisonno, Riccardo Guerzoni, Ishan Vaishnavi, and David Soldani. Sdn-based 5g mobile networks: architecture, functions, procedures and backward compatibility. *Transactions on Emerging Telecommunications Technologies*, 26(1):82–92, 2015.

[19] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51. Ieee, 2009.

[20] Nick Antonopoulos and Lee Gillam. *Cloud computing*. Springer, 2010.

[21] Brian Hayes. Cloud computing, 2008.

[22] What is a container? URL https://www.docker.com/resources/what-container.

[23] What is a virtual machine (vm)? URL https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine.

[24] Swarm mode overview, Feb 2021. URL https://docs.docker.com/engine/swarm/.

[25] Production-grade container orchestration. URL https://kubernetes.io/.

[26] Martin Fowler. *Microservices*. Martin Fowler, 2017.

[27] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2): 114–119, 2013. ISSN 01636804. doi: 10.1109/MCOM.2013.6461195. https://doi.org/10.1109/MCOM.2013.6461195.

[28] Jain Raj and Paul Subharthi. Network virtualization and software defined networking for cloud computing: A survey. *IEEE Communications Magazine*, 51(11):24–31, 2013. ISSN 01636804. doi: 10.1109/MCOM.2013.6658648. https://doi.org/10.1109/MCOM.2013.6658648.

[29] Diego Kreutz, Fernando Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *arXiv*, 2014. https://arxiv.org/abs/1406.0440.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746. http://doi.acm.org/10.1145/1355734.1355746.

[31] Keith Kirkpatrick. Software-defined networking. *Commun. ACM*, 56 (9):16–19, 2013. ISSN 0001-0782. doi: 10.1145/2500468.2500473. http://doi.acm.org/10.1145/2500468.2500473.

[32] Openflow switch specification version 1.5.1. Technical report, 2015. http://www.opennetworking.org.

[33] What's software-defined networking (sdn)? - sdxcentral, Sep 2019. https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn.

[34] Ethan Smith. Types of drones - explore the different types of uav's, Feb 2017. URL https://www.circuitstoday.com/types-of-drones.

[35] Dji - official website. URL https://www.dji.com/be.

2

Pluggable SDN Framework for Managing Heterogeneous SDN Networks

*In this chapter, the Domino framework is proposed, allowing to manage heterogeneous SDN networks from a single framework. In the context of this dissertation, it enables the management and allocation of network resources where multiple drones are connected by means of SDN technologies and provide an answer to Research Challenge 1 (**C1**) discussed in Chapter 1. We hypothesize that management of these networks while maintaining Quality-of-Service constraints can be done with a minimal overhead (**RH1**). The Domino framework is implemented using a plugin-based microservices approach and evaluated in terms of performance, interoperability and modifiability. A use case exemplifying a simple Dijkstra routing algorithm over multiple SDN networks showcases the possibilities and features of this framework. Results show that the most frequent commands within the Domino framework are executed within 1 second, making it usable for managing the network in a drone related context.*

⋆⋆⋆

J. Moeyersons, P-J Maenhaut, F. De Turck, and B. Volckaert.

Published in International Journal of Network Management, June 2020.

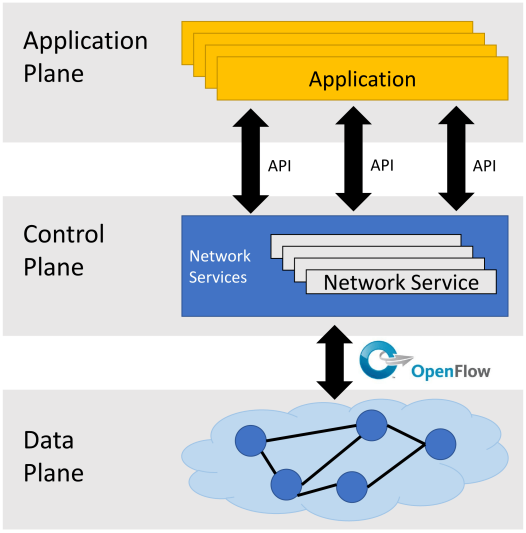
Abstract Software-Defined networking (SDN) is a new network paradigm that is separating the data plane and the control plane of the network, making one or more centralized controllers to supervise the behaviour of the entire network. Different types of SDN controller software exist and research dealing with the difficulties of consistently integrating these different controller types has mostly been declared future work. In this chapter the Domino framework is proposed, a pluggable SDN framework for managing heterogeneous SDN networks. In contrast to related work the proposed framework allows research into SDN networks controlled by different types of SDN controllers attempting to standardize the northbound API of them. Domino implements a microservice plugin architecture where users can link different SDN networks to a processing algorithm. Such an algorithm allows for e.g. adapting the flows by building a pipeline using plugins that either invoke other SDN operations or generic data processing algorithms. The Domino framework is evaluated by implementing a Proof-of-Concept implementation which is tested on the initial requirements. It achieves the modifiability and the interoperability with an average successful exchange ratio of 99.99 %. The performance requirements are met for the frequently used commands with an average response time of 0.26 seconds and the framework can handle at least 72 plugins simultaneously depending on the available amount of RAM. The proposed framework is evaluated by means of the implementation of a shortest path routing algorithm between heterogeneous SDN networks.

2.1 Introduction

The Internet is essential in today's society, needed for applications such as social networks, cloud computing and people's working, studying and living styles. However, traditional network technology cannot meet the requirements of the current network demands because of the high complexity for both design and configuration [1]. A new network design, Software-Defined Networking (SDN) has been proposed that can manage networks dynamically and flexibly.

SDN is an important technology to realize dynamic and flexible network management which overcomes the aforementioned weaknesses of traditional networks [3, 4]. A schematic overview of SDN is shown in Figure 2.1

Figure 2.1 Schematic overview of Software-Defined Networking planes[2]



and further explained. With SDN, the control plane, that decides how to handle network traffic, and the data plane, that forwards traffic according to the decision made by the control plane, are separated. Due to the separation of the control and data planes, network switches in the data plane become simple packet forwarding devices instructed by the rules created by the SDN controller application in the control plane. The data plane always consists of the southbound Application Programming Interface (API), which is further explained below, and the network infrastructure. The SDN controller (also called the Network Operating System (NOS)) and the corresponding northbound API are part of the control plane. A network hypervisor can eventually be added. Finally, the application plane bundles the different possible network applications who implement the control-logic that will be translated into commands to be installed in the data plane. These applications can perform traditional functions such as routing, load balancing and security but they can also explore novel approaches such as reducing power consumption. [5] The above-mentioned features of SDN are harder to achieve in the traditional networks due to the complex network control and the different protocols per problem. Also the limited testing environments and the long standardization process are avoided when using SDN. A brief comparison between SDN and conventional networking is provided in Table 2.1.

Communication between the control plane and the data plane, the southbound API, follows the OpenFlow protocol in most cases [6, 7], giving the

Table 2.1: Comparison between SDN and traditional networking [12]

	SDN	**Traditional Networking**
Features	decoupled data and control plane with application plane on top	a new protocol per problem, network control complex [13]
Configuration	automated configuration with centralized validation	error prone manual configuration
Performance	dynamic global control with cross layer information	limited information and relatively static configuration
Innovation	Easy software implementation for new ideas	Difficult hardware implementation for new ideas
	Sufficient test environment with isolation	Limited testing environment
	Quick deployment using software upgrades	Long standardization process

control plane access to network switches and routers in the data plane. OpenFlow allows switches from different vendors, each having their own interfaces, to be managed remotely using a single, open protocol. The OpenFlow protocol has evolved from version 1.0 which supported only 12 fixed match fields and a single flow table, to the latest version 1.5 which features multiple tables, more than 41 matching fields and several new functions [8] such as the addition of egress tables, enabling processing to be done in the context of the output port instead of in the context of the input port. The major differences between the different OpenFlow versions are summarized in Table 2.2. The business applications on top of the control plane communicate over a custom northbound API. For now, the northbound API has no well-defined standards to clarify what it is and can do [9, 10]. Zhou et al. [11] for example proposed a possible design for the northbound API in a RESTful manner through a set of REST API design patterns.

The original design of SDN consists of a single centralized controller for managing the whole network, resulting in a potential single point of failure. A single centralized controller also cannot handle the day-to-day increasing network demands due to performance limitations of the controller itself. To overcome these issues, a multi-controller architecture is proposed in several works [14–16], in which multiple controllers together fulfill the tasks of a logically centralized controller.

In multi-controller environments, the same SDN controllers such as Ryu [17], Floodlight [18] or Open Network Operating System (ONOS) [19] are used to manage the whole network. In these multi-controller environments, east- and westbound APIs, similar to the north- and southbound APIs, are essential components to enable clear communication between the different distributed controllers. [5] To enable and provide common compatibility and interoperability between the different controllers, it is necessary to have standard east/westbound interfaces such as SDNi [20], which defines common requirements to coordinate flow setup and exchange reachability information across multiple domains. Many solutions have already been proposed for creating and managing these homogeneous SDN networks. [21–25]

This chapter however focuses on the requirements for and the management of heterogeneous SDN networks by creating a pluggable framework that allows users to control the heterogeneous network, proposing a standard northbound API. The framework allows for research over heterogeneous SDN networks by linking those networks together in a standardized manner. To test the viability of the architecture, a proof-of-concept prototype is implemented and evaluated in terms of performance, modifiability, interop-

Table 2.2: Major version changes between main OpenFlow Versions [26]

Version	Major Feature	Reason	Use Cases
1.0 - 1.1	Multiple table	Avoid flow entry explosion	
	Group table	Enable applying action sets to group of flows	Load balancing, failover, link aggregation
	Full Virtual Local Area Network (VLAN) and Multiprotocol Label Switching (MPLS) support		
1.1 - 1.2	Openflow eXtensible Match (OXM) Match	Extending matching flexibility	
	Multiple controller	High Availability (HA)/load balancing/scalability	Controller failover, controller load balancing
1.2 - 1.3	Meter table	Add Quality-of-Service (QoS) and DiffServ capability	
	Table miss entry	Provide flexibility	
1.3 - 1.4	Synchronized table	Enhance table scalability	MAC learning/forwarding
	Bundle	Enhance switch synchronization	Multiple switch configuration
1.4 - 1.5	Egress table	Enabling processing to be done in output port	
	Scheduled bundle	Further enhance switch synchronization	

erability and security. As a use case, a shortest path routing algorithm is implemented and evaluated within the framework, which aims to identify the shortest path between different heterogeneous SDN networks. In case of an emergency case, regular communication and video streams from e.g. CCTV cameras and drones used by the public services should not be interrupted, delayed or down scaled while other, unimportant network communication such as live streams from Facebook, must be down scaled to a minimum or even blocked completely. This can be achieved by calculating the shortest path from source (the cameras) to destination (the public services) and there is no guarantee that this will happen between homogeneous SDN controllers. As such, we investigated that our proposed solution is capable of creating and managing new emergency (high priority) flows over different heterogeneous SDN networks. The remainder of this chapter is organized as follows. In Section 2.2 related work is discussed. Section 2.3 presents the requirements for the management of heterogeneous SDN networks and the corresponding framework design is discussed in Section 2.4. Section 2.5 presents the implementation of the framework prototype; the use case for setting up shortest path flows across heterogeneous SDN networks implemented on top of the framework is discussed in Section 2.6. Evaluation of the framework and the results for the shortest path algorithm are presented in Section 2.7. Finally, Section 2.8 draws conclusions from this research and indicates potential avenues for future research.

2.2 Related Work

Recent research on SDN has shown that many challenges in a multi-controller network have already been tackled [27]. These challenges include scalability, consistency, reliability and load-balancing. Solutions for the scalability problem are categorized into two aspects: the placement and number of controllers [28] and how to divide the network into multiple domains for a multi-controller SDN network [14]. Consistency between different controllers [22] and between the applied strategies [14] is required to have a correctly working multi-controller SDN network. A key feature of SDN is that the functionality of the network highly depends on the controller software. Misbehaving software, hardware errors or failing physical links among switches and controllers can lead to an inconsistent network state. There are solutions for both guaranteeing reliable controller nodes [21] and reliable connection links between switches and controllers [23]. Another challenge, load-balancing, is achieved by clustering controllers together [29] or migrating switches from one controller to another [24, 25, 30, 31], resulting in load-balanced SDN controllers. Some existing solutions will

be explained to determine the requirements imposed on the framework proposed in this chapter.

Dixit et al. [24] proposed ElastiCon, the first switch migration framework based on a dynamic multi-controller architecture. ElastiCon contains three main modules namely a load measurement module, load adaptation decision module and an action module. The load measurement module collects the load of each controller and sends the information to the load adaptation decision module, which decides on load allocation between the different controllers. The action module is responsible for migrating switches and adding or removing controllers from the network to achieve an elastic network. One of the main drawbacks of ElastiCon is the decision making in setting load balancing issues among controllers. This can settle uneven load distribution among controllers due to the extra time needed to collect information from all controllers and to send appropriate load balancing commands to the overloaded controller. As a result, Yu et al. [25] proposed a load balancing mechanism based on a load information strategy for controllers whereby the mechanism is running as an extra module on each controller. The load balancing mechanism is now responsible for measuring the load metrics of a controller, sharing the load information to other controllers and making decisions about load balancing and switch migration. Both Elasticon and the proposed load balancing mechanism of Yu et al. are evaluated on a simulated homogeneous multi-controller SDN network. In the work proposed in this chapter, similar architectures can be deployed and evaluated on both real- and simulated- heterogeneous SDN networks instead of homogeneous SDN networks.

A problem that can occur in both homogeneous and heterogeneous SDN networks is a connection failure between the controllers and the switches. Behesthi and Zhang [23] created a protection metric for the connections between the controllers and the switches, resulting in fast failover for the control traffic. Cascone et al. [32] propose SPIDER, a failure recovery in SDN that provides a fully programmable abstraction to application developers for defining the rerouting policies and for management of the failure detection mechanism. The results from their work can be integrated in our proposed framework to achieve improved reliability.

A lot of research on homogeneous multi-controller SDN networks has already been realized. However, research concerning heterogeneous SDN networks has to the best of our knowledge mostly been declared future work. In this chapter, the focus is on this type of SDN network by designing, prototyping and evaluating a management framework that enables research into SDN networks controlled by different SDN controller types.

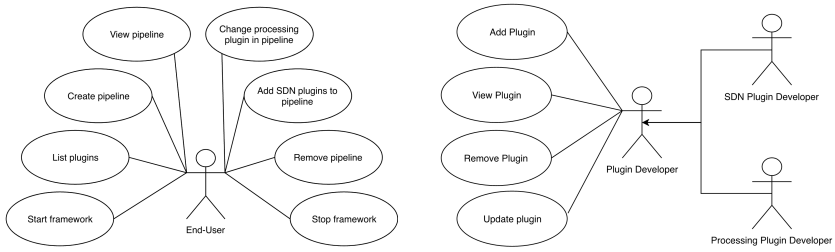
2.3 Requirements

In this section, both functional and non-functional requirements for the proposed framework will be discussed. The functional requirements will determine which actors can use the framework and their specific tasks. The non-functional requirements describe framework operational constraints that need to be taken into account.

2.3.1 Functional requirements

Three general actors are identified for the framework: an end-user that wants to manage different SDN networks for a specific use case, SDN experts implementing APIs for different SDN controller software (further called SDN plugins) and network planning/managing module developers integrating new processing algorithms or methods (further called processing plugins) into the framework so that end-users can utilize them to build their applications. The SDN experts and the processing software developers are generalized to an actor called plugin developer, who develops plugins to extend the functionality of the framework. These plugins are the building blocks with which the end-user can build SDN network managing applications. An end-user can choose from the different provided plugins to create and adapt his application with the framework. Plugin developers should be able to add new SDN network APIs and processing methods or adapt existing plugins. The tasks of both end-users and plugin developers are explained below and summarized in Figure 2.2.
A plugin developer is responsible for creating and maintaining new plugins. A plugin should implement a REST API that allows communication with other plugins. Each plugin should a least have an endpoint to link two plugins to each other and an endpoint to remove the link between two connected plugins. Each plugin should be uploaded to a distribution service where it will be available for use in the framework. An overview of the use cases for the plugin developers is illustrated in Figure 2.2b.
An end-user can use plugins from the different plugin distribution services to create custom SDN network managing networks. The SDN plugins are available from the SDN distribution and the processing plugins are available from the Processing distribution. Through the framework's Command Line Interface (CLI), the end-user can enlist the different available plugins and create a custom application (further called a pipeline) by linking different SDN plugins to a single processing plugin. Afterwards he can add or remove SDN plugins from the pipeline or swap the processing plugin. The end-user can also start and stop the entire framework through the CLI. The different use cases for the end-users are illustrated in Figure 2.2a.

Figure 2.2 Use case diagram for Dominos end-users and plugin developers



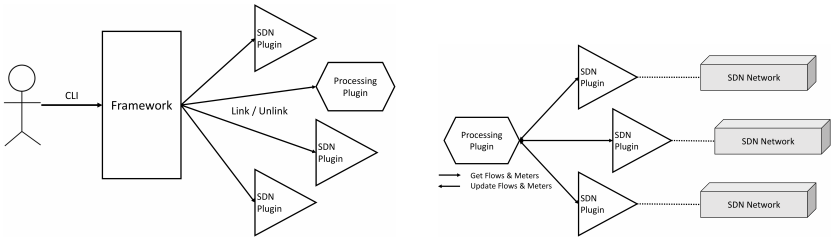
(a) End-user use cases **(b)** Plugin developer use cases

2.3.2 Non-functional requirements

The non-functional requirements specify how the framework is supposed to operate, or in what manner it should execute its functionality. Requirements that will be discussed are performance, interoperability, modifiablity and security. In terms of performance, the framework should be able to meet timing requirements such as maximum allowed latency and scalability. The software industry has not defined a quantified 'acceptable latency' for end-users, but a 4 second latency rule is often used as a rule-of-thumb [33]. The average response time for general framework commands should therefore be less than 2 seconds, with a standard deviation of 1 second. The latency introduced by the framework is caused by both the management of all plugins and the translation of the SDN control calls but it will not introduce any delay on the ongoing network operations in the underlying SDN networks. The number of users that can use the framework simultaneously is assumed to be low, so the assumption is made that a maximum of five users can use the framework at the same time based on the use case where one end-user, three SDN plugin developers and one processing plugin developer use the framework simultaneously. In terms of maximum number of active plugins, the framework should at least work with several active plugins without any remarkable performance downgrade. The number of active plugins that the framework is able to handle is evaluated in Section 2.7.

Interoperability is the degree to which two or more independent systems can usefully exchange meaningful information via interfaces in a particular context. The framework will interoperate with SDN controllers and processing methods via framework plugins as shown in Figure 2.3a. An SDN plugin is a plugin that represents an API of a specific SDN Controller software for e.g. getting the current flows and meters and set improved flows and meters with a higher priority. A processing plugin retrieves the current flows and meters from specific SDN plugins and processes them as illustrated in

Figure 2.3 Interoperability diagrams



(a) Framework interoperability **(b)** Plugin interoperability

Figure 2.3b. In some cases it is possible that a process plugin will forward improved flows and/or meters to the different SDN plugins. The framework will thus interact with the SDN and processing plugins, with which the framework exchanges requests to link them together, control their processing, etc. The more correct exchanges there are between them, the better the user can use the plugin for building applications with the framework.

Modifiability is the cost and risk of changing the functionality of the system. One of the most important values of the framework is modifiability of the supported SDN and processing plugins. The framework needs to be extendable for new functionalities by enabling developers to add new plugins. End-users should be able to modify the components that they use for their applications easily and quickly to allow for interchangeable SDN controller software and processing methods. To enable end-users to choose the extensions they need, the framework will need a distribution service that contains all possible plugins available for the framework. When a user adds a plugin from the distribution to his version of the framework, the framework should only reload once before making the plugin useable. Another part of modifiability is deployability, defined as the different device configurations that specify how the framework can be deployed. If the framework can be deployed in different fashions such as completely local, completely distributed or a combination of both where some components are deployed locally while others are deployed distributed, this can increase the value for the end-user.

Security has three main characteristics. Confidentiality is the property that data or services are protected from unauthorized access. Integrity is the property that data or services are protected from unauthorized manipulation. Availability is the property of the system maintaining its functionality during a possible attack. Security is important for the framework if it is deployed on multiple devices that use a public network to communicate.

2.4 Design of a heterogeneous SDN controller framework

In this section, the proposed software architecture will be discussed. First, an overview of the required design patterns to meet requirements from Section 2.3 is explained. Next, the static overview is shown with extra information about the architecture of the plugins. Finally, different deployment possibilities will be summarized.

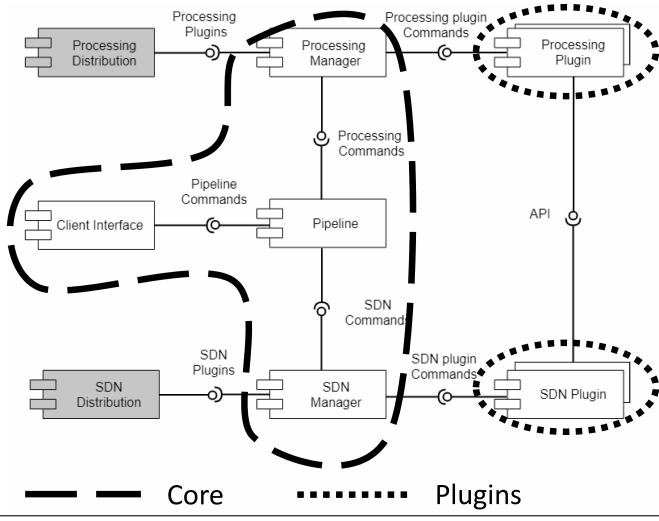
2.4.1 Application of design patterns

To meet the requirements listed in Section 2.3, several design patterns have been applied to the proposed framework. To meet the non-functional architectural requirements, the microkernel pattern in combination with the microservices pattern is proposed. The microkernel pattern [34] allows the addition of application features as plugins to the core application, providing extensibility as well as feature separation and isolation. The pattern consists of two components: a core system called the kernel and plugins. The kernel only contains the minimal functionality required to make the system operational. The plugins are standalone independent components that contain specialized processing, additional features and custom code.
The microservices pattern [35] structures the application as a collection of loosely coupled services that implement business capabilities. Each component of the pattern is deployed as a separate unit that can be deployed on one or multiple devices. Interoperability and deployability of these components can be ensured by the microservices pattern, as it designs the microservices to have well defined interfaces for interoperability and allows for the framework to be deployed in a distributed fashion.

2.4.2 Static view

Figure 2.4 presents an overview of the architecture as a component-connector diagram. Components are the boxes that represent different software entities. The components have interfaces through which they interact with other components. The type of data exchanged is noted next to the interface. Multiple boxes indicate that multiple components of the same kind can exist at runtime.
The two used patterns, described in Section 2.4.1, are also shown in Figure 2.4. Each component is a microservice that implements its own interface that can interact with other components. As will be explained in Section 2.4.4, different components can be deployed on either a single machine or spread across different machines. The microkernel pattern is implemented

Figure 2.4 Component-connector overview of the framework. The clear components are the core components of the framework that each user needs to install to use the framework. The colored components are used for the distribution of plugins.

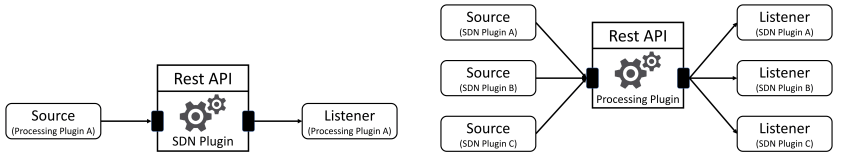


as different core components namely the client interface, the pipeline, the SDN manager, the processing manager and the plugins for both SDN and processing modules.

The architecture of the framework includes, next to the core components and the plugins, also distribution services (colored components in Figure 2.4) for framework plugins to extend the functionality. These are not installed with the core framework but can run as remote instances with which the user can interact to extend the core framework. An end-user can use the framework via the CLI, building pipelines that are maintained in the Pipeline component. The Pipeline component makes requests to the Processing Manager and SDN Manager components to activate and control the selected plugins to build the pipeline. Additional plugins can be added to the framework and are distributed via the SDN and Processing distribution components. The pipeline and the different plugins are further discussed in Section 2.4.3.

The communication between the microservices must follow a communication protocol. There are two types of traffic exchanged between the microservices. First, there are command requests that are exchanged between microservices to edit resources or change state. Second, there are API calls that are exchanged between the Processing and SDN plugins. For both types of traffic, the requests must be reliable and executed only once. HTTP can be used when communication reliability is of concern, while UDP can be

Figure 2.5 Plugin Diagram



(a) SDN Plugin **(b)** Processing Plugin

used if latency is deemed more important. For our prototype we prefer HTTP communication to obtain the reliability and to guarantee that a request is executed only once.

2.4.3 A pipeline: linked Processing and SDN plugins

A plugin represents an independent element, either of the Processing type (e.g. a flow viewer application) or the SDN type (e.g. a Ryu API application). Plugins are deployed as a standalone microservice providing a REST API interface that the framework uses to control the plugin. Figure 2.5 represents the plugin diagram for both SDN plugins (Figure 2.5a) and Processing plugins (Figure 2.5b). A plugin receives data from other plugins called sources, processes this data and forwards it to other plugins called listeners. An SDN plugin has only one source and one listener, which is in both cases a Processing plugin. A Processing plugin has both sources and listeners, which are different SDN plugins whereby the SDN plugin functions both as source and listener. For example, different SDN plugins will send flow information to one Processing plugin, who will process the flows to create better paths and afterwards will push the new flows to each SDN plugin. When different plugins are activated through the framework's CLI, they are linked to each other which is further called a pipeline. A pipeline is thus the collection of different SDN plugins who are linked to one specific processing plugin. An example is illustrated in Figure 2.7. In this example, three heterogeneous SDN networks are controlled by three SDN plugins which are sending flow information (source) to a Processing plugin called 'flow viewer'. Through a custom defined web interface in the processing plugin, a user can update the flows and these updated flows will then be pushed back to the different SDN plugins (listeners).
The plugin REST API should a least provide three endpoints: a `/state` resource representing the state of the plugin, a `/sources` resource that represent the sources from which the plugin receives data to process and a `/listeners` resource which represent the listeners to which the plugin transmits the processed data.

Figure 2.6 Plugin state transition diagram

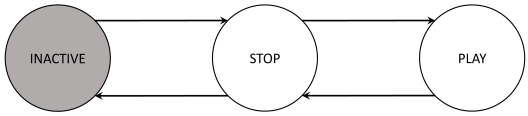
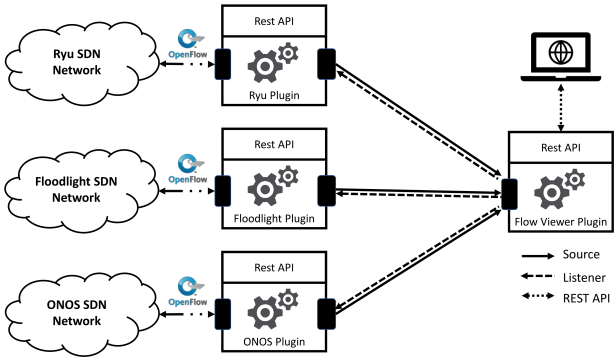


Figure 2.7 Example of a pipeline whereby three different SDN networks are connected to three SDN plugins which are connected to one processing plugin called 'flow viewer'. The flow viewer application can be reached through his custom made API which visualizes the different flows in the three SDN networks.



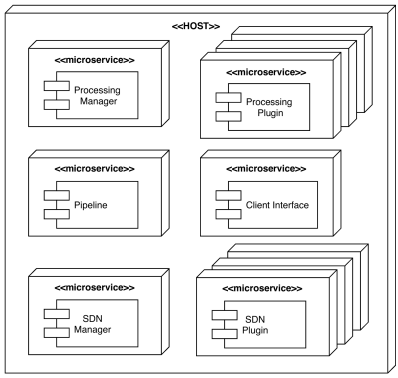
A plugin can be in three different states: **INACTIVE**, **STOP** and **PLAY**. When a plugin is in the **INACTIVE** state, no active microservice is running. This is the initial state of a plugin and only visible to the framework. When a plugin is in the **STOP** state, the framework has instantiated a microservice running the plugin that now is waiting to have listeners and or sources. Once the plugin receives a listener or a source, the state of the plugin is **PLAY**. Figure 2.6 shows the possible state transitions for each plugin. If a plugin needs to be put in the **INACTIVE** state from the **PLAY** state, the plugin will first make the transition from **PLAY** to **STOP** to make sure that all sources and listeners are released from the plugin before shutting it down and putting it in the **INACTIVE** state.

2.4.4 Deployment view

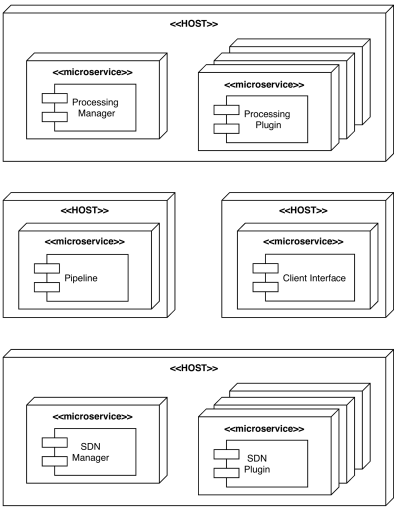
The different deployment options are illustrated via deployment diagrams in Figure 2.8. 'Host' specifies the device on which components are deployed and the 'microservice' indicates the isolated environment in which components are running. These isolated environments on the host are realized as software containers that enable portability of the components to other de-

Figure 2.8 Deployment diagrams

(a) Local configuration



(b) Distributed configuration



ployment configurations. The Processing and SDN distribution components were left out of the diagrams. Two deployment configurations are presented: a local configuration in which all components are deployed on a single device and the distributed configuration in which different components can be distributed across multiple devices. The proposed microservice architecture enables different deployment options because of the use of software containers which can be deployed in different manners.

The local configuration is shown in Figure 2.8a. With this configuration, the framework can work offline if the required plugins are already available on the host. The components are still deployed as separate microservices due to the architecture of the framework. This has an impact on the performance because for every interaction between components the HTTP protocol is used which introduces an extra overhead compared to direct invocation of commands.

Figure 2.8b illustrates the distributed configuration, deploying the framework on multiple devices. The components are distributed over these devices, made possible by the microservice isolation and communication protocols. Due to the distributed nature, performance will be worse when compared to the local configuration, because each request between components travels over a network connection that is subject to potential delays.

2.5 Prototype Implementation

To prove the concept of the architecture discussed in the previous chapters, a prototype is implemented. First the different possible microservice frameworks are discussed. Afterwards the different deployment options are explained followed by implementation of the different SDN and Processing plugins. Next an overview of the CLI is provided which will discuss the possible commands that can be used to manage the framework and finally the used hardware is illustrated.

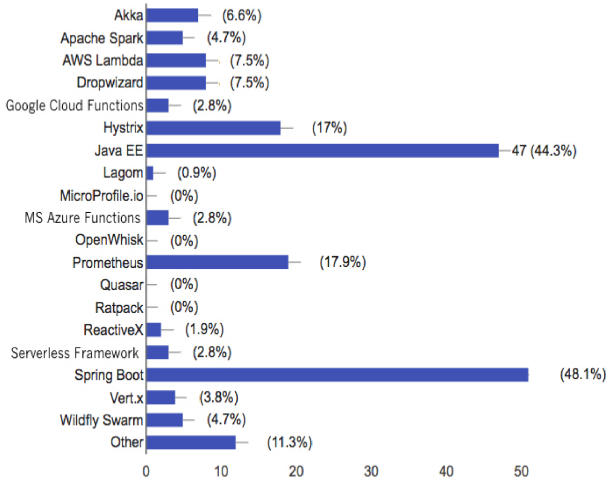
2.5.1 Microservice frameworks

The architecture presented in Section 2.4 relies heavily on the microservices pattern. Therefore this section aims to present the chosen microservice framework. Figure 2.9 depicts the results of the Rethink IT survey querying the most used frameworks for microservices by developers [36]. The most popular frameworks, Java EE and Spring Boot are written in Java. The Java EE framework is more of a one-stop-shop framework instead of a specialised lightweight microservice framework and is therefore not considered. Spring Boot is clearly a very popular and mature framework, more streamlined for microservices. Vert.x is a more recent framework gaining in popularity, renowned for its performance, making it worthwhile to explore. Python is a language for web development and because it is excellent for prototyping, several microservice frameworks for this language exist.
For the prototype implementation, Flask [37] is selected. Flask is a micro web development framework for Python, providing a glue layer to build a REST API around the application and uses the concept of Python decorators to bind Python functions to a REST API. Because it is a microframework, the memory footprint is small with the binary file only being 535KB large. Flask is in use by several large companies such as Netflix and Reddit [38]. In a production environment the default Flask web server is not sufficient, as it only serves one user at the time, and a production WSGI server such as Waitress [39] needs to be used. However for prototyping and evaluation purposes the Flask web server suffices.

2.5.2 Deployment framework

To allow for the modifiability and interoperability requirements discussed in Section 2.3 and the different deployment configurations in Section 2.4.4 Docker and Kubernetes are used.
Docker started as an open-source project at dotCloud in early 2013. It was an extension of the technology the company had developed to run its

Figure 2.9 Rethink IT: Most used tools and frameworks in 2017 based on an online survey for microservices [36]. The X-axis shows the usage percentage for the most used frameworks/tools (shown on the Y-axis) to built microservices.



cloud applications on thousands of servers [40]. Now, Docker is a standalone mature company providing a software container platform for the deployment of applications [41]. Docker provides two main services: a simple tool set and API for managing Linux containers and a cloud platform which provides easy access to images for software containers created by other developers [42]. Docker is the container technology with most public traction and is becoming the container standard at the time of writing, due to its offered functionality and responsive community [43]. It offers to easily build and run containers but also manages them in large clusters. A design decision that limits Docker is that each container can only run one process at a time in combination with the Docker client. Docker consists of a daemon that manages the containers and the API Engine, a REST client. Should this client fail, dangling containers can arise [44]. A dangling container can be compared to a zombie process. Docker Compose [45] is a tool for defining and running multi-container Docker applications. With Compose, a YAML file is used to configure the different application services and these can be started or stopped with a single command.

Kubernetes [46] is an open source cluster manager for Docker containers developed by Google and later on by the open source community. It allows for decoupling of application containers from the details of the systems on

which they run. This decoupling simplifies application development since users only ask for abstract resources like cores and memory, and it also simplifies data center operations. One or more containers deployed in Kubernetes are grouped together inside a pod, which will have its own unique IP address inside the Kubernetes cluster allowing communication with other pods whether they are co-located on the same physical machine or not. The concept of a pod [47] in Kubernetes makes it easy to tag multiple containers that are treated as a single unit of deployment. They are co-located on the same host and share the same resources. Each container running within the same pod gets the same hostname, so they can be addressed as one unit. When a pod is scaled out, all containers within this pod are scaled out automatically. Services in Kubernetes maintain a well-defined endpoint for each pod. As a pod could be relocated from one node to another, the endpoint of the pod will be changed but the endpoint of the service remains the same. Multiple pods running across multiple nodes of the Kubernetes cluster can be exposed as a service, which is an essential building block of microservices.

2.5.3 SDN and processing plugins

The prototype implementation includes three SDN plugins and two processing plugins. The processing plugins that are implemented are a flow viewer API, which simply retrieves the current active flows of the linked SDN plugins and a shortest path algorithm plugin which will be explained in Section 2.6. The three included SDN plugins are explained below.
The three SDN plugins are APIs built on top of three well-known SDN controllers. Each SDN plugin is then distributed as an Docker image, ready to be used by the framework. The first SDN plugin is an API on top of the Ryu SDN controller. Ryu [17] is a framework written in Python, providing several components useful for SDN applications such as a simple switch and a topology discoverer and viewer. It allows the modification of existing components or the creation of new components to combining them to form a Ryu application. Ryu supports OpenFlow version up to and including 1.3 and Nicira [48] extensions. The created SDN plugin is built on top of the Ryu base application and the endpoints are implemented with the Flask package [37]. The second SDN plugin is built on top of the Floodlight SDN controller software [18], an open-source enterprise level controller implemented in Java that declares itself to be designed for high performance and easy to use and deploy. It offers a module loading system that allows user extension and enhancements. Floodlight supports OpenFlow versions up to and including 1.5 and several extensions. The custom API for the SDN plugin is implemented using the Java Restlet framework [49]. The

Listing 1 Info.yml file for the Ryu SDN plugin

```
---
meta-information:
  version: 0.1
  name: Ryu API
  description: |
    API for Ryu based Controllers
  author: Jerico Moeyersons
build:
  context: .
  dockerfile: Dockerfile
ports:
  api_port: 5000
  OF_port: 6633
---
```

ONOS [19] SDN controller framework is used to implement the last SDN plugin. It declares itself as scalable, highly performant and highly available thanks to the modular software that is based on Floodlight and implemented in Java. ONOS also supports OpenFlow versions up to and including 1.5. The provided API for the SDN plugin is implemented using the Java Restlet framework.

Each SDN plugin should be accompanied by an **info.yml** file, describing the meta-information such as description, author, name and version, the build and the ports used for both the API endpoint and the OpenFlow port. Each processing plugin should also be accompanied by an **info.yml** file, which is similar to the file used for the SDN plugins but with no OpenFlow port entry. An example **info.yml** file for the Ryu SDN plugin is illustrated in Listing 1.

2.5.4 The command line interface

The CLI is the main entry point for an end-user to use and manage the entire framework. It can be installed with Python pip and referred to as Domino. The name comes from both the game where different dots needs to be interconnected to eachother (reffering to the pluggable feature of the framework) and from the Latin *dominus*, meaning the master of its slaves and servants which are in this case the plugins. It is built in Python with the Click package by Armin Ronacher [50]. Click is CLI creation kit. It resembles the Flask framework that is used to create the microservices and it also leverages Python decorators for most of its functionality. With Domino,

the end-user can start or stop the framework, get a list of the installed plugins, activate plugins and link plugins to each other. The following commands are implemented:

- `domino`: Displays a help page listing command groups
- `domino on`: Starts the application
- `domino off`: Stops the activated plugins and the application
- `domino plugins`: Groups all commands to manage plugins. In the current prototype, a list command has been implemented.
- `domino plugins ls`: Lists all locally installed plugins
- `domino pipeline`: Groups all commands to manage the current pipeline
- `domino pipeline add <type> <name>`: Adds a SDN or processing plugin to the pipeline. The type can be a *processing_plugin* or an *sdn_plugin* and the name is the name of the plugin as shown in the `domino plugins ls` command. When a new plugin is added to the pipeline, it will be linked automatically to the other active plugins in the pipeline.
- `domino pipeline delete <name>`: Deletes an SDN or processing plugin with the provided name. If *all* is filled in, all plugins will be removed from the pipeline.
- `domino pipeline elements`: Lists all SDN and processing plugins that are added to the pipeline

A typical use of the application would be to first start the application using the `domino on` command. Then plugins are added to the pipeline using `domino pipeline add <type> <name>`, which will instantiate the corresponding plugins in the SDN Manager or Processing Manager component. Note that only one processing plugin can be added to the pipeline due to design choices. As mentioned in the commands above, linking between the plugins inside a pipeline will happen automatically when a plugin is added to it.

2.5.5 Used hardware

To implement the described framework different hardware is used. The SDN networks are built with switches from Northbound Networks [51] namely

Zodiac ZX [52] and the Zodiac GX [53], both OpenFlow enabled switches with 10/100 Mbps Ethernet ports and 1Gbps Ethernet ports respectively. The Zodiax FX supports OpenFlow version up to and including 1.3 and the Zodiac GX supports the same OpenFlow versions but also version 1.4 and 1.5. OpenFlow version 1.3 is chosen to be used in all SDN networks to have consistency between the two different types of SDN switches.
The framework is deployed in two ways, a local deployment on a server and a distributed deployment on both the same server as on a Kubernetes development cluster. The server consists of 4 Intel(R) Xeon(R) CPU E5645 @ 2.40GHz and 4GB RAM, which is running Ubuntu 16.04.03 LTS with Docker CE version 18.09.3. The Kubernetes cluster has one master node with 2 Intel(R) Xeon(R) CPU E5645 @ 2.40GHz and 4GB RAM and one worker node with 4 Intel(R) Xeon(R) CPU E5645 @ 2.40GHz and 4GB RAM.

2.6 Case Study: SDN Routing algorithm implementation in DOMINO

In case of an emergency, finding the shortest path from source A to destination B is useful for setting up high priority routes for e.g. streaming purposes. In this case, a camera stream that is filming the emergency incident uses the created high priority route with eventual higher bandwidth possibilities while other cameras use the normal route. In case source A and destination B are not part of the same SDN network, the proposed framework is able to find the shortest path between A and B by adding the SDN plugins of the different SDN networks between A and B and linking these plugins to the shortest path algorithm processing plugin, which is summarized in Figure 2.10. In this section the shortest path processing plugin will first be discussed, followed by the implementation and the use of it in the framework.
The shortest path processing plugin is implemented in Python and based on Dijkstra's algorithm [54]. The algorithm is used for finding the shortest path between two nodes in a graph, in this case a network topology. The more common variant is used in this plugin, where one node is chosen as the source node and the shortest path to all other nodes in the graph are calculated. Once the requested destination node is reached, the algorithm will stop and the shortest path from the source to the destination is determined. The algorithm runs in time $O|V^2|$ where $|V|$ is the number of nodes.
To use Dijkstra's algorithm as processing plugin, some small changes are introduced. First of all, a Flask wrapper is created that enables the microservices pattern. As required by the framework, the plugin will have at

Listing 2 Activate the Dijkstra algorithm in Domino

```
$ domino on
$ domino pipeline add sdn_plugin ryuapi
$ domino pipeline add sdn_plugin floodlightapi
$ domino pipeline add sdn_plugin onosapi
$ domino pipeline add processing_plugin dijkstra
$ domino pipeline elements # returns:
# elements:
# sdn_plugin: ryuapi
# sdn_plugin: floodlightapi
# sdn_plugin: onosapi
# processing_plugin: dijkstra
```

least three endpoints as discussed in Section 2.4.3. An extra endpoint is added that will start the calculations for getting the shortest path from the given source to the given destination. Next, the feature to collect the needed data from the linked sources is implemented. The plugin will query the different switches, links and hosts from each SDN plugin for further processing. Finally, Dijkstra's algorithm is executed and eventual new flows are installed on the SDN plugins (where they act now also as listener).

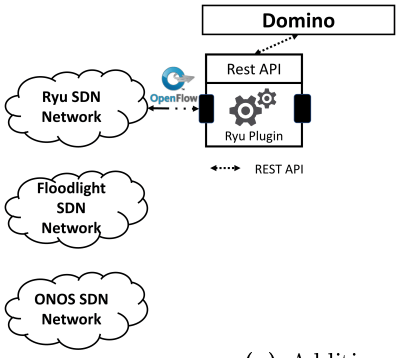
When the plugin is uploaded to the processing distribution, it is available for use in the framework. To activate and link the Dijkstra plugin, the framework is used with required steps visualized in Listing 2. Figure 2.10a demonstrates the step when the ryuapi SDN plugin is added to the empty pipeline. Next, the floodlightapi and the onosapi SDN plugins are being added which is illustrated in Figure 2.10b. Finally the Dijkstra Processing plugin is added to the pipeline. In this final step, the framework automatically creates the links between the SDN plugins and the Processing plugin, summarized in Figure 2.10c. Now, the shortest path from a source A to destination B can be requested from http://<docker-ip>:5000/emergency_path/<src>/<dst> which will return the shortest path in the form $[(< nexthop >, < in\_port >, < out\_port >), ...]$.

2.7 Evaluation Results

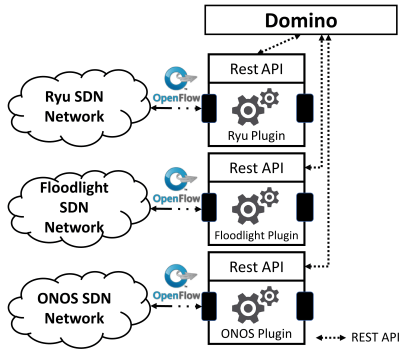
The goal of this section is to present the results of the framework and the use case experiment. The results of the framework evaluation are explained in Section 2.7.1 and the results of the use case experiment are presented in Section 2.7.2.

Figure 2.10 Building the use case Pipeline

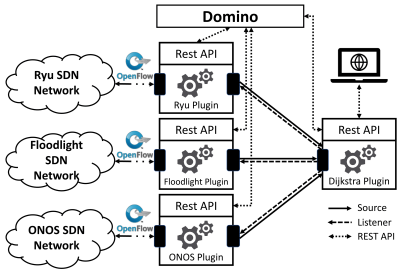
(a) Add the Ryu SDN plugin



(b) Add the Floodlight and ONOS SDN plugin



(c) Addition of Dijkstra Processing plugin, the link with the SDN plugins is automatically created by Domino



2.7.1 Framework results

To evaluate the framework, different tests are conducted that evaluate if the framework meets the requirements described in Section 2.3. First the performance is evaluated followed by the evaluation of the interoperability and the modifiability.

2.7.1.1 Performance evaluation

To evaluate the performance of the framework, the resource usage impact is monitored and the storage footprint is measured, giving an overview of the eventual overhead the framework will create. The framework execution times of each command executed with the CLI are measured. Each command is executed 200 times except for the **on** and **off** commands, which are measured manually 10 times because these commands launched system threads and their finish signal could not be captured, they had to be measured manually. As explained in Section 2.5.5, this is evaluated on both the local deployment and the distributed deployment. The summarized results of this evaluation are visualized in Table 2.3. The average execution times for listing the plugins, adding a plugin to the pipeline and listing the elements of the current pipeline do not exceed the 2 second bound specified in Section 2.3.2, while the average execution times of on, off and the delete plugin from the current pipeline commands do exceed this bound. Especially the delete and off command exceed the requirements. The delete plugin command shuts down a plugin and removes the Docker container or Kubernetes Pod from the host. This action is costly in time due to security reasons of Docker (safely shutdown of the container and then remove it). The Off command removes all the plugins and all the microservices of the framework and thus suffers from the same costly action. This could be ameliorated by having the framework not remove the containers but stopping them instead, which requires less resources as it only stops the process running in the container but does not purge the container from the system. The time to start the framework is measured in two ways, with and without cache. Without cache, the framework needs an average of 66 seconds to start. This is due to downloading different docker base images, building the different containers and finally starting them. With cache it only takes an average of 2.356 seconds to start the framework. The On command is left out of the 2 second bound delay, because it acts more like an installation, which requires more time.
The scalability requirement can be met if the Flask Werkzeug server is changed by a production WSGI server such as Waitress. However, in the proposed prototype Flask is used and the Flask Werkzeug server is only

Table 2.3: Performance test for both local and distributed setup, measured in seconds

Metric	On	On (cache)	Off	plugins ls
Avg	66.176	2.356	11.453	0.026
Std deviation	0.347	0.530	0.053	0.005
25 Percentile	66.054	1.902	11.405	0.024
Median	66.176	2.345	11.441	0.005
75 Percentile	66.299	2.799	11.495	0.028
90 Percentile	66.372	2.826	11.510	0.031
95 Percentile	66.397	2.836	11.515	0.034

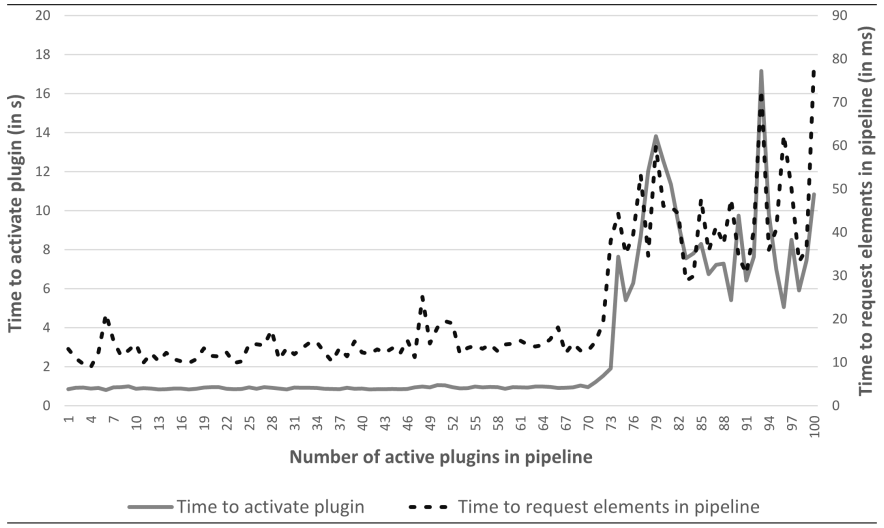
Metric	add plugin	delete plugin	elements
Avg	0.745	10.466	0.008
Std deviation	0.216	0.025	0.001
25 Percentile	0.770	10.451	0.008
Median	0.778	10.457	0.008
75 Percentile	0.839	10.487	0.009
90 Percentile	0.861	10.496	0.010
95 Percentile	0.885	10.497	0.010

able to process one request at a time.

As suggested in Section 2.3.2, the framework should be able to handle several active plugins at the same time without any remarkable performance downgrade. To evaluate this, new plugins are incrementally added to the pipeline while measuring the time needed to activate this plugin. Afterwards the time needed to request the elements in the pipeline is measured. The results are summarized in Figure 2.11 and explained further. The framework can handle 73 plugins in one pipeline without any remarkable performance downgrade. Namely, the average time to add a plugin is 0.9358 seconds and the time to request the active elements measures 0.0137 seconds. As from plugin 74, a major increase in time for both commands can be observed. The reason for this increase in time from an average 0.9358 seconds to 7 seconds or even 12 seconds is due to the memory constraints of the server. Monitoring showed that the available memory of the server was too low to start a new container and therefore Docker shut down another container. Adding extra memory to the system would increase the maximum number of active plugins in a pipeline, but this is not further evaluated as 73 active plugins at the same time is deemed sufficient.

Our evaluations show that the three performance requirements are not completely met by the prototype. However this is mostly due to some actions being very slow such as shutting down the framework or removing a

Figure 2.11 Time required to add and activate a new plugin to the pipeline and time required to request active plugins in the pipeline as a function of the number plugins in the pipeline



plugin. As these actions occur less frequently when a user is using the framework, they are deemed less important in terms of perceived user quality. Frequent actions such as adding and listing elements perform well. The number of active plugins that the framework can handle simultaneously is sufficient and the scalability requirement can be met if the different components are deployed through a production GI server instead of the provided Flask Werkzeug server.

The total size of all the Docker images of the components of the framework is illustrated in Table 2.4. The core components of the framework have an average size of 105 MB, while the plugins have sizes that range from 122MB to 775MB. The size can be explained due to the used base images and additionally installed software in the images. The images of the core framework are optimized by using lightweight UNIX base images, but as the plugins can be added by other parties (such as SDN experts), the size of different plugin images may vary.

2.7.1.2 Interoperability evaluation

The systems with which the framework exchanges data are the plugins. These plugins must follow the plugin model presented in Section 2.4.3, implement the presented resources using a REST API, the state machine and the protocols. If these specifications are met by a plugin, the frame-

Table 2.4: Total size of framework components.

Image	Size [MB]
sp-manager	95.9
sdn-manager	96.7
processing-manager	122
ryuapi	311
floodlightapi	195
onosapi	775
flow-viewer	183
shortestpathalgorithm	92.9

Table 2.5: Interoperability test results (p.= Plugin)

Value	Play	Stop	Link p.	Unlink p.	List linked p.
Correct	49985	50000	50000	49999	50000
Incorrect	15	0	0	1	0
Ratio (%)	99.97	100	100	99.998	100

work will be able to exchange information with the plugin. To evaluate this, a new mock plugin is implemented. For each resource of the plugin, the framework is given random mock input data to exchange with the different plugins. When the exchange is complete, the values in the plugin are requested and compared to the given input. If the input matches the value in the plugin, the exchange was successful. These tests were executed 50000 times and the results are summarized in Table 2.5.

Play and Stop are the requests to change the state of the plugin, the link and unlink plugin manipulate the sources and listeners of the plugin and the list linked plugins shows the status of the pipeline in the framework. Overall there were 16 errors made when exchanging information, only when changing the state of a plugin there were 15 incorrect exchanges and there was 1 incorrect exchange when a plugin was removed from the pipeline (unlinked). The cause of these incorrect exchanges is due to Kubernetes that cannot handle names containing an underscore and Flask's inability to handle special characters. The ratios achieved are always 100 % correct exchanges except for changing the state and deleting an element from the pipeline which are 99.97 % and 99.998 % respectively, so this requirement is met.

2.7.1.3 Modifiability evaluation

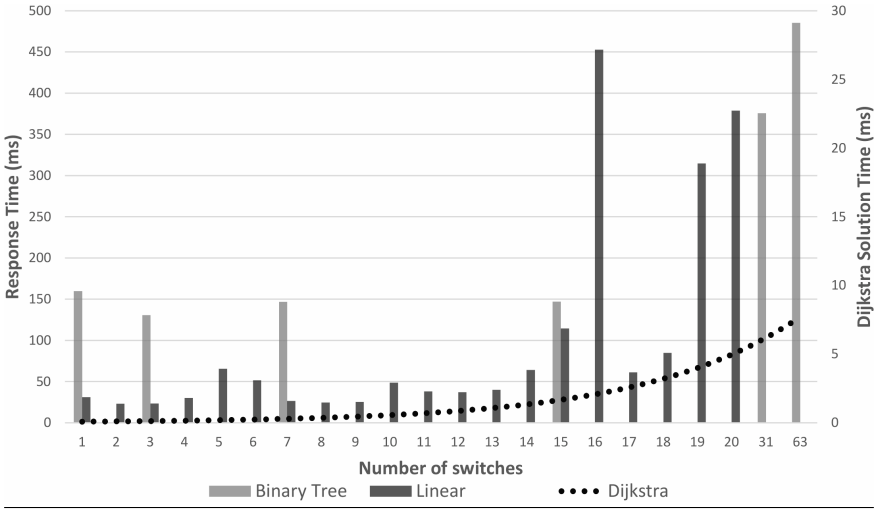
Plugins are installed for the prototype by building and adding their image to the image repository of the Docker host. The framework does not need to restart to install these images and the framework can detect newly installed plugins when starting up. End-users can extend the framework with new plugins by installing them by building the respective plugin images. Pipelines can be modified by linking different plugins by design. The prototype is deployed in both a local and distributed fashion without major changes to the code base.
In general the framework is designed to be modifiable for different SDN networks and processing plugins. The hybrid microkernel/microservices architecture enables this modifiability. The microkernel plugin architecture allows a user to modify a pipeline during framework use and the microservices architecture allows for a modifiable deployment configuration.

2.7.2 Routing evaluation results

To evaluate the shortest path use case based on Dijkstra's algorithm, a pipeline is implemented with the prototyped framework. This pipeline, consisting of three different SDN plugins, namely the ryuapi, the onosapi and the floodlightapi, and one processing plugin namely the dijkstra plugin is evaluated on both a real SDN network and on a simulated SDN network with Mininet [55]. To attach multiple SDN controllers to Mininet, a custom `Switch()` class needs to be created whereby the different Mininet switches can be mapped onto the corresponding remote SDN controllers. An example is available on the Mininet GitHub repository[1]. To evaluate the use case, the average response time of 100 requests to determine the shortest path is measured as a function of the number of SDN switches. The results are illustrated in Figure 2.12 and further explained.
Two different topologies are used to evaluate the algorithm namely a linear and a binary tree topology. The linear topology allows to evaluate the performance when the shortest path covers multiple switches compared to the binary tree topology, where the shortest path algorithm is essential to find the correct path from the root node to a specific leaf node (or any other node in the topology). The first topologies which require less than 6 different switches are created using the hardware described in Section 2.5.5 and from 6 switches onwards Mininet is used. It is clear that the response time for the tree topology is higher than for the linear topology probably because of there are more links between the different switches in the tree topology. As described in Section 2.6, the algorithm runs in time $O|V^2|$

[1] https://github.com/mininet/mininet/blob/master/examples/controllers.py

Figure 2.12 Average response time to calculate the shortest path as a function of the number of SDN switches in a linear or tree topology. From 6 switches onwards, Mininet [55] is used to simulate the network topology. The dotted line illustrates the calculation time for the Dijkstra's algorithm without the framework.



where $|V|$ is the number of switches and this can also be seen in the Figure (line plot). It is clear that the use of the framework has an impact on the solution time for the Dijkstra's algorithm. This is due to the different HTTP requests that are sent between the different plugins to obtain the status of the different SDN networks before the algorithm can be executed. But, the framework requirements specify a 2 second bound which is not exceeded in this evaluation.

2.8 Conclusion

Software-Defined networking separates the data plane and control plane of the network, making one or more centralized controllers control the behaviour of the entire network. Communication between these controllers and the switches follows a standardized southbound API which is in most cases implemented using the OpenFlow protocol. The applications on top of the controllers to manage the networks communicate through the Northbound API of these controllers, which has no clear standards to clarify what it can and cannot do. In most cases, a REST API is chosen as northbound API. Therefore the goal of this chapter was to design, build and evaluate a pluggable framework that allows building applications in a modifiable way

over heterogeneous SDN networks, in an attempt to standardize the northbound API. The specific use case of determining the shortest path between different heterogeneous SDN networks in case of an emergency situation was investigated as a sample use case for the framework. Hereby, high priority flows needed to be implemented over the heterogeneous SDN network, guaranteeing the network traffic from source (e.g. a drone) to destination (e.g. dispatch centre).

A hybrid combination of the microkernel pattern and the microservices pattern is used to meet the performance, interoperability and modifiability requirements as the microkernel pattern enables interchanging the SDN and Processing plugins via a plugin system and the microservices pattern enables different deployment configurations for the framework. To build and evaluate the framework, several technologies were needed: container and container orchestration technologies for the software architecture, SDN controller software and a processing plugin for the shortest path use case.

The microservices frameworks know a lot of variety, depending a lot on the use case for the application. Some aim at quick prototyping while others focus on performance etc. Flask was selected as microservice framework as it is easy to use and designed for prototyping, however, this comes with a performance trade-off. To deploy the microservices, container technology was chosen, utilising Docker containers on top of Kubernetes orchestration.

The implementation of the prototype includes different sample plugins such as the ryuapi, floodlightapi, onosapi, flow viewer and the dijkstra algorithm. The framework is limited to one processing plugin for different SDN plugins and is deployed in both a local and distributed fashion. The shortest path algorithm based on Dijkstra is also implemented using a pipeline created by the framework consisting of three SDN plugins and one processing plugin.

Afterwards, different aspects of the framework are evaluated. The performance requirements of the framework for the frequently used commands are met with an average response time of 0.26 seconds. Other commands such as deleting plugins, stopping or starting the framework do not meet the performance requirements since Docker requires significant time to start, stop and remove containers. The size of the core components of the framework is small with an average size of 105 MB per image. The size of the plugins depends on the images created and can vary between 183 MB to 775 MB. The framework can handle at least 73 plugins simultaneously without any performance downgrade on a server with 4GB of RAM. The interoperability requirements are all met by the framework with most exchanges having a 100 % successful exchange ratio. The modifiability requirements regarding the plugins and the deployment options are met by the framework. Finally the use case is evaluated regarding the response time in function of the

number of connected switches, which in this case resulted in an average response time between 0.02 and 0.45 seconds for a linear topology and 0.01 and 0.5 seconds for a tree topology.
In terms of future work, the framework should add security measures. In the distributed configuration, communications rely on an external network and additional security is therefore recommended. The creation of multiple pipelines with the framework will also be investigated in future work.

Acknowledgment

The authors would like to thank all partners namely, Orange, Ericsson, Accelleran, Rombit, imec and KU Leuven in the 5Guards project and the Agency for Innovation by Science and Technology (Vlaio) for funding and support.

Addendum

Note on the requirements in a drone use case: In this chapter, different requirements are made and evaluated without the notification of the drone use case. We stated that a 4 second latency is acceptable and that the average response time for the general framework commands should be less than 2 seconds, with a standard deviation of 1 second. Evaluations proved that these requirements are met (for the most common commands in the Domino framework). In a drone use case where the focus lies on the installation of emergency flows, these requirements are also acceptable, knowing that the moment an emergency occurs, the new plugin to handle this emergency can be installed within 0.745 seconds (as shown in Table 2.3).

References

[1] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. *6th USENIX Symposium on Networked Systems Design and Implementation*, pages 335–348, 2009. ISSN 1064-3745. doi: 10.1007/978-1-59745-177-2_17. https://doi.org/10.1007/978-1-59745-177-2_17.

[2] What's software-defined networking (sdn)? - sdxcentral, Sep 2019. https://www.sdxcentral.com/networking/sdn/definitions/what-the-definition-of-software-defined-networking-sdn.

[3] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2): 114–119, 2013. ISSN 01636804. doi: 10.1109/MCOM.2013.6461195. https://doi.org/10.1109/MCOM.2013.6461195.

[4] Jain Raj and Paul Subharthi. Network virtualization and software defined networking for cloud computing: A survey. *IEEE Communications Magazine*, 51(11):24–31, 2013. ISSN 01636804. doi: 10.1109/MCOM.2013.6658648. https://doi.org/10.1109/MCOM.2013.6658648.

[5] Diego Kreutz, Fernando Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *arXiv*, 2014. https://arxiv.org/abs/1406.0440.

[6] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746. http://doi.acm.org/10.1145/1355734.1355746.

[7] Keith Kirkpatrick. Software-defined networking. *Commun. ACM*, 56 (9):16–19, 2013. ISSN 0001-0782. doi: 10.1145/2500468.2500473. http://doi.acm.org/10.1145/2500468.2500473.

[8] Openflow switch specification version 1.5.1. Technical report, 2015. http://www.opennetworking.org.

[9] Jean Tourrilhes, Puneet Sharma, Sujata Banerjee, and Justin Pettit. Sdn and openflow evolution: A standards perspective. *Computer*, 47

(11):22–29, 2014. doi: 10.1109/MC.2014.326. https://doi.org/10.1109/MC.2014.326.

[10] Marta Weissenborn. Northbound intent, 2015. https://www.opennetworking.org/news-and-events/blog/northbound-intent/.

[11] Wei Zhou, Li Li, Min Luo, and Wu Chou. Rest api design patterns for sdn northbound api. *Proceedings - 2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, IEEE WAINA 2014*, pages 358–365, 2014. ISSN 1070986X. doi: 10.1109/WAINA.2014.153. http://doi.org/10.1109/WAINA.2014.153.

[12] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. A survey on software-defined networking. *IEEE Communications Surveys and Tutorials*, 17(1):27–51, 2015. ISSN 1553-877X. doi: 10.1109/COMST.2014.2330903. https://doi.org/10.1109/COMST.2014.2330903.

[13] Scott Shenker, Martin Casado, Teemu Koponen, Nick McKeown, et al. The future of networking, and the past of protocols. *Open Networking Summit*, 20:1–30, 2011.

[14] Kevin Phemius, Mathieu Bouet, and Jeremie Leguay. Disco: Distributed multi-domain sdn controllers. *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, pages 1–4, 2014. ISSN 1542-1201. doi: 10.1109/NOMS.2014.6838330. https://doi.org/10.1109/NOMS.2014.6838330.

[15] Yonghong Fu, Jun Bi, Jianping Wu, Ze Chen, Ke Wang, and Min Luo. A dormant multi-controller model for software defined networking. *China Communications*, 11(3):45–55, 2014. ISSN 16735447. doi: 10.1109/CC.2014.6825258. https://doi.org/10.1109/CC.2014.6825258.

[16] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane. *ACM SIGCOMM Computer Communication Review*, 37(4):1, 2007. ISSN 01464833. doi: 10.1145/1282427.1282382. URL http://portal.acm.org/citation.cfm?doid=1282427.1282382. https://doi.org/10.1145/1282427.1282382.

[17] FUJITA Tomonori. Introduction to ryu sdn framework. *Open Networking Summit*, 2013.

[18] N Solomon, Y Francis, and L Eitan. Floodlight openflow ddos. *Slideshare. net*, 2013.

[19] Pankaj Berde, William Snow, Guru Parulkar, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, and Pavlin Radoslavov. Onos: Towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, 2014. ISBN 9781450329897. doi: 10.1145/2620728.2620744. https://doi.org/10.1145/2620728.2620744.

[20] H Yin, H Xie, T Tsou, D Lopez, P Aranda, and R Sidi. Sdni: A message exchange protocol for software defined networks (sdns) across multiple domains. *IETF draft, work in progress*, 2012. https://tools.ietf.org/id/draft-yin-sdn-sdni-00.txt.

[21] Kshira Sagar Sahoo, Bibhudatta Sahoo, Ratnakar Dash, and Nachiketa Jena. Optimal controller selection in software defined network using a greedy-sa algorithm. *International Conference on Computing for Sustainable Global Development (INDIACom), 2016 3rd*, pages 2342–2346, 2016.

[22] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. pages 3–3, 2010. https://pdfs.semanticscholar.org/f7bd/dc08b9d9e2993b363972b89e08e67dd8518b.pdf.

[23] Neda Beheshti and Ying Zhang. Fast failover for control traffic in software-defined networks. *GLOBECOM - IEEE Global Telecommunications Conference*, pages 2665–2670, 2012. ISSN 1930-529X. doi: 10.1109/GLOCOM.2012.6503519. https://doi.org/10.1109/GLOCOM.2012.6503519.

[24] Advait Dixit, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Ramana Rao Kompella. Elasticon; an elastic distributed sdn controller. *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 17–27, 2014. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7856398.

[25] Jinke Yu, Ying Wang, Keke Pei, Shujuan Zhang, and Jiacong Li. A load balancing mechanism for multiple sdn controllers based on load informing strategy. *18th Asia-Pacific Network Operations and Management Symposium, APNOMS 2016: Management of Softwarized Infrastructure - Proceedings*, (61501044):1–4, 2016. ISSN 0167-8140. doi: 10.1109/APNOMS.2016.7737283. https://doi.org/10.1109/APNOMS.2016.7737283.

[26] Chang Ching-Hao and Ying-Dar Lin. Openflow version roadmap. Technical report, 2015. http://speed.cis.nctu.edu.tw/$\sim$ydlin/miscpub/indep_frank.pdf.

[27] Tao Hu, Zehua Guo, Peng Yi, Thar Baker, and Julong Lan. Multi-controller based software-defined networking: A survey. *IEEE Access*, 6:15980–15996, 2018. ISSN 21693536. doi: 10.1109/ACCESS.2018.2814738. https://doi.org/10.1109/ACCESS.2018.2814738.

[28] Ashutosh Kumar Singh and Shashank Srivastava. A survey and classification of controller placement problem in sdn. *International Journal of Network Management*, 28:e2018, 2018. ISSN 10991190. doi: 10.1002/nem.2018. https://doi.org/10.1002/nem.2018.

[29] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. Balanceflow: Controller load balancing for openflow networks. *Proceedings - 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems, IEEE CCIS 2012*, 2:780–785, 2013. doi: 10.1109/CCIS.2012.6664282. https://doi.org/10.1109/CCIS.2012.6664282.

[30] Jie Cui, Qinghe Lu, Hong Zhong, Miaomiao Tian, and Lu Liu. A load-balancing mechanism for distributed sdn control plane using response time. *IEEE Transactions on Network and Service Management*, 15(4):1197–1206, 2018. ISSN 19324537. doi: 10.1109/TNSM.2018.2876369. https://doi.org/10.1109/TNSM.2018.2876369.

[31] Hongchang Chen, Guozhen Cheng, and Zhiming Wang. A game-theoretic approach to elastic control in software-defined networking. *China Communications*, 13(5):103–109, 2016. ISSN 16735447. doi: 10.1109/CC.2016.7489978. https://doi.org/10.1109/CC.2016.7489978.

[32] Carmelo Cascone, Davide Sanvito, Luca Pollini, Antonio Capone, and Brunilde Sanso. Fast failure detection and recovery in sdn with stateful data plane. *International Journal of Network Management*, 27(2):1–14, 2017. ISSN 10991190. doi: 10.1002/nem.1957. https://doi.org/10.1002/nem.1957.

[33] Scott Barber. Acceptable application response times vs. industry standard, 2018. https://searchsoftwarequality.techtarget.com/tip/Acceptable-application-response-times-vs-industry-standard.

[34] Mark Richards. *Software architecture patterns*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated, 2015.

[35] Sam Newman. *Building microservices: designing fine-grained systems*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2015.

[36] Hartmut Schlosser. Microservices trends 2017: Strategies, tools and frameworks - jaxenter, 2017. https://jaxenter.com/microservices-trends-2017-survey-133265.html.

[37] Armin Ronacher. Welcome to flask — flask documentation (0.12), 2017. http://flask.pocoo.org/docs/0.12/.

[38] Stackshare. Companies that use flask and flask integrations, 2018. https://stackshare.io/flask.

[39] Waitress - a wsgi server for python 2 and 3, 2019. https://github.com/Pylons/waitress.

[40] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment, March 2014. ISSN 1075-3583. http://dl.acm.org/citation.cfm?id=2600239.2600241.

[41] Docker Inc. What is a container, 2018. https://www.docker.com/what-container.

[42] John Fink. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25:29, 2014.

[43] Matthew Heusser. 30 essential container technology tools and resources. https://techbeacon.com/enterprise-it/30-essential-container-technology-tools-resources.

[44] Chenxi Wang. What is docker? linux containers explained, 2017. https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html.

[45] Docker toolbox overview | docker documentation. https://docs.docker.com/toolbox/overview/.

[46] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes - lessons learned from three container-management systems over a decade. *ACM Queue*, 14(February):70–93, 2016. ISSN 15427730. doi: 10.1145/2898442.2898444. https://doi.org/10.1145/2898442.2898444.

[47] Janakiram MSV. The kubernetes way: Pods and services - the new stack, 2016. https://thenewstack.io/kubernetes-way-part-one/.

[48] Vmware buys nicira for 1.26 billion dollar and gives more clues about cloud strategy, 2012. http://tcrn.ch/MB7EeR.

[49] Jerome Louvel, Thierry Templier, and Thierry Boileau. *Restlet in action: developing restful web apis in Java*. United States: Manning Publications Co., 2012.

[50] Armin Ronacher. Click documentation (5.0), 2017. http://click.pocoo.org/5/.

[51] Paul Zanna. Northbound networks, 2018. https://northboundnetworks.com/.

[52] Paul Zanna. Zodiac fx openflow switch hardware - northbound networks, 2016. https://northboundnetworks.com/collections/zodiac-fx/products/zodiac-fx.

[53] Paul Zanna. Zodiac gx open vswitch hardware - northbound networks, 2018. https://northboundnetworks.com/products/zodiac-gx.

[54] E. W. Dijkstra. A note on two probles in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. https://doi.org/10.1007/BF01386390.

[55] Rogerio Leao Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6. IEEE, 2014. doi: 10.1109/ColComCon.2014.6860404. https://doi.org/10.1109/ColComCon.2014.6860404.

3

Towards Distributed Emergency Flow Prioritization in SDN Networks

*With the outcome of the previous chapter, the next step is to look into emergency flow prioritization in SDN networks. Therefore, this chapter along with Appendix A proposes an offline and online approach to guarantee emergency flows in SDN networks while optimizing the best effort flows over the remaining network resources, corresponding to Research Challenge 2 (**C2**) and hypothesis 2 (**RH2**). Evaluations prove that such an approach is feasible and that network traffic from specific drones, who are part of handling the emergency situation, can be prioritized over network traffic from other drones and devices while these still receive some network resources depending on the overall available resources in the network. It is also clarified how multiple heterogeneous SDN controllers can collaborate to distribute network management load, which otherwise would be limited by resource constraints alike controller memory limitations.*

⋆⋆⋆⋆⋆

J. Moeyersons, B. Farkiani, T. Wauters, B. Volckaert. and F. De Turck

Published in International Journal of Network Management, June 2020.

Abstract Emergency services must be able to transfer data with high priority over different networks. With 5G, slicing concepts at mobile network connections are introduced, allowing operators to divide portions of their network for specific use cases. In addition, Software-Defined Networking (SDN) principles allow to assign different Quality-of-Service (QoS) levels to different network slices.
This chapter proposes a microservices-based framework, able to run both centralized and distributed, that guarantees the required bandwidth for the emergency flows and maximizes the best-effort flows over the remaining bandwidth based on their priority. The proposed framework consists of an offline linear model, allowing to optimize the problem for a batch of flow requests. For dynamic situations, an online approach is also required in the framework to handle new incoming flows by calculating the path with a shortest path algorithm and utilising a greedy approach in assigning bandwidth to the intermediate flows.
In this chapter, the linear model is evaluated through simulation, the distributed architecture is evaluated through emulation while the online approach is validated through physical experiments with SDN switches. The results show that the linear model is able to guarantee the resource allocation for the emergency flows while optimizing the best-effort flows with a sub-second execution time. The distributed architecture is able to split up the managed network into different parts, allowing division of work between controllers. As a proof-of-concept, a prototype with Zodiac switches validates the feasibility of the centralized framework.

3.1 Introduction

During an emergency event, it is required to prioritize the network traffic that is coming from and going to the emergency services in the presence of large civilian crowds in order to coordinate the relief and response. The enabler for this statement were the terror attacks at Brussels airport and the metro station in Maalbeek on March 22, 2016 [1]. Right after the two explosions, the phone networks in Belgium had broken down and saturated as a lot of people were looking to contact the emergency services, friends and family. This also caused communication problems within the emergency services itself. To avoid similar cases in the near future, ASTRID, the specialist telecoms operator for Belgium's emergency and security services, launched priority SIM cards for specific persons. This will allow these persons (such as the minister of defense, first aid commanders, etc) to have secure and priority access to the mobile network [2]. However, these priority SIM cards cannot be shared with other persons that may need it during a

specific emergency situation, because every situation can be different and will require other capabilities. Therefore, a more generalized solution is required that can guarantee the bandwidth of emergency network traffic and can optimize the other, non-priority traffic, over the remaining bandwidth in the network. The solution can be found by first looking into the next generation mobile networks, called 5G.

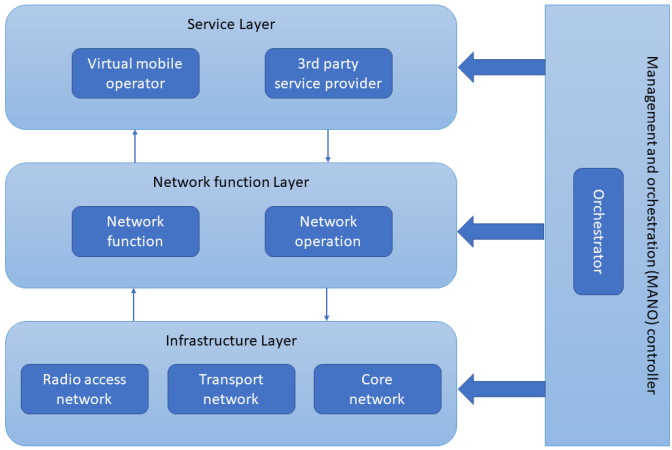
Since Release Document 15 of 3rd Generation Partnership Project (3GPP) [3], the 5G system is introduced and explained. In Release 16, expected to be formally released in June 2020, the completion of the 5G specifications as well as enhancements to many early capabilities for 5G standalone mode, including Ultra-Reliable and Low Latency Communication (URLLC), V2X Phase 3 and more are described. While some parts are thus already implemented and rolled out by the industry, the formal release of 5G is expected be sometime in 2021 [4]. One of the future aspects of 5G systems is to cater a wide range of services differing in their requirements and types of devices, going further than the traditional human-type communications and thus includes machine-type communications. In that case, the network must be able to take different forms depending on the required service, leading to the slicing of the network on a per-service basis. Technologies such as Software-Defined Networking (SDN) and Network Function Virtualization (NFV) can be used to provide these network slicing concepts, simultaneously providing a multitude of diverse services over a common underlying physical infrastructure [5]. This will allow network operators to provide portions of their networks for specific use cases such as Internet-of-Things (IoT), streaming videos and smart energy grids.

There are three layers needed that enable network slicing in the future 5G networks, namely the infrastructure layer, the network function layer and a service layer [5], all containing the necessary tools for the operators, enterprises, etc. These three layers are managed by a management and orchestration (MANO) controller. The architecture is illustrated in
Figure 3.1 and the infrastructure and network function layer will be further explained. The infrastructure layer refers to the physical network infrastructure including both the Radio Access Network (RAN) and the Core Network (CN) but also the deployment, control and management of the infrastructure. The allocation of resources to slices will also happen in this layer where the resources of each slice can be revealed and managed by the network function layer and eventual extra layers such as the service layer. The network function layer encapsulates the operations that are related to the configuration and life cycle management of the network functions that offer end-to-end service in the network slice. These network functions must however be placed optimally over the virtual infrastructure and chained together

to work optimally. In this layer, the industry and researches have already found a consensus about the role of SDN and NFV. [6–8] NFV separates network functions from the underlying proprietary hardware appliances, [9] enabling the life cycle management and orchestration of the network functions. The network functions running on dedicated hardware are thus transferred to software-based applications running in datacenters, network nodes, end-user premises etc. SDN on the other hand is an important technology to implement dynamic and flexible network management by separating the data plane from the control plane in networks [10]. Every SDN switch (further called switch) within an SDN network operates as a simple packet forwarding device that is controlled by a logically centralized software program, the SDN controller. An SDN controller performs all complex functions such as routing, naming and security checks. The controller defines the data flows that occur in the network, considering that e.g. the communication is permissible by the network policy. If the controller allows a flow, it computes a route for the flow and adds an entry for that flow in each of the switches along the path. Switches are now responsible for managing their own flow tables whose entries are thus populated by the controller. The switch also performs certain functions in an SDN network. When a new packet of a flow arrives at the switch, the switch forwards and encapsulates it to the controller. The controller can thus decide to add the flow to the flow table of the switch or always drop the packets in that flow. Other packets are forwarded to the specific output port of the switch, based on the entries in the flow table. Some flow tables may include priority information set by the controller. The controller can also decide to drop specific packets, apply bandwidth meters to limit the maximum bandwidth available to certain flows, etc. The communication between the controller and the switches uses a standardized protocol API, most commonly with the OpenFlow specification. [11] In this chapter we will focus on the network function layer within the network slicing concept in order to provide a generalized solution for the above described problem.

Therefore, this chapter presents a generalized and containerized framework to guarantee the bandwidth of emergency network traffic by generating SDN high-priority flows while other, non-priority traffic, will receive best-effort resources. The proposed framework is thus a use case within the slicing concept of 5G networks. Because different operators can collaborate in one network topology or because multiple controllers can manage the SDN-based network topology, a distributed approach is necessary. This allows for better management of the topology while guaranteeing the emergency flows and optimizing the best-effort flows. Fog computing and smart cities are other use cases that can benefit from the proposed framework. [12] A simulation,

Figure 3.1 Generic framework representing various 5G architectural proposals [5].



emulation and prototype has been implemented in order to evaluate our proposed framework in terms of speed, scalability and accuracy. In real cases, network operators deploying software-defined technologies can allow emergency flows to reserve specific slices of bandwidth for a specific amount of time.

This chapter contributes to four main topics: *(i)* design of models for guaranteed bandwidth allocation for emergency flows while optimizing best-effort flows over the remaining network resources, *(ii)* design of a joint online-offline approach to practically implement the model, *(iii)* design and implementation of a centralized and distributed microservices-based framework and *(iv)* the validation of the model through simulations, emulations and practical evaluation. The remainder of this chapter is organized as follows: Section 3.2 presents related work. In Section 3.3, the problem description is given followed by the problem formulation as linear model. Section 3.4 presents the architectural design and implementation of our framework followed by the evaluation methods and the corresponding result in Section 3.5. Finally, Section 3.6 discusses conclusions and future avenues of research.

3.2 Related work

The network slicing concept introduces the possibility to enable new features such as more fine-grained Quality-of-Service (QoS), and a lot of research is

done on SDN over the past few years. [13, 14] Different algorithms to provide QoS, but without considering bandwidth guarantees, are presented. [15, 16] Yan et al. [15] proposed a QoS solution based on SDN technology. They first defined a cost function which assigns a positive value to each link based on bandwidth, length and congestion of the link. Afterwards, they utilized a weighted shortest path algorithm [17] to find multiple paths for each source and destination pair in the network. When a new flow arrives, the path with the lowest cost is selected as the routing path for the flow. Zhang et al. [16] proposed a QoS framework based on the OpenFlow protocol which dynamically calculates a path for each flow. If the flow is a QoS-required flow, an algorithm based on Dijkstra is used to find the path with minimum delay and cost values.

Akella et al. [18] presented an approach to allocate bandwidth and satisfy QoS requirements. They categorized flows into QoS and best effort flows and defined a metric, used in path selection, that considers the requested rates. Shaohua et al. [19] categorized cloud applications into three levels based on the sensitivity to delay and bandwidth. A flow-based adaptive routing algorithm which utilizes Dijkstra and K-shortest path [20] algorithms with the aim of maximizing the utilization of network resources is proposed and evaluated through simulation by Tomovic et al. [21] Pinto et al. [22] defined four service classes including best effort and bandwidth guaranteed classes. Each new flow is first assigned to the probing class and its behavior is monitored. After some time, if the network can support its bandwidth along the path it will be reassigned to the bandwidth guaranteed class or otherwise the best effort class. A method to provide bandwidth guarantees by using OpenFlow meters and queues is presented by Krishna et al. [23] The authors categorized flows into QoS flows which have minimum guaranteed bandwidth and best effort flows with no requirements. For each QoS flow, first, an admission control process checks whether there is a path that can accommodate the flow rate. After that, by using a meter at the ingress switch, the input rate of the flow is monitored and if it exceeds the defined rate, the packets will be marked. Using three different queues at the egress port of each switch along the path for marked and unmarked QoS and best effort flows, traffic prioritization is made possible. Morin et al. [24] used Multiprotocol Label Switching (MPLS) tunnels to provide end-to-end bandwidth guarantees, which is similar to the work of Krishna et al. [23] where they used OpenFlow meters at the ingress switches. For each flow the input rate of the flow is monitored and based on that, a priority value is set in the header of each packet. Then, an MPLS tunnel is used to route the packets toward the egress switch and the priority of each packet specifies its output queue. Lu et al. [25] utilized preplanned network slices to

both satisfy QoS requirements and maximize the overall throughput of the network. The authors used the traffic history to create network slices which have fixed configurations during the network lifetime. When a flow arrives, it is assigned to a slice by using the VLAN ID of the slice. The MaxStream framework [26] is proposed in order to maximize the number of streaming sessions and bandwidth provisioning. The authors formulated two Integer Lineair Programming (ILP) problems. The first problem maximizes the number of accepted flows by considering the requested rate of the flows. Then, the set of accepted flows is used in the second problem to maximize the total rate of the accepted flows. Since the authors focused on multimedia streams, they ignored best effort flows with no QoS requirements.
More recent work proposes auction-based resource allocation in multi-tenant networks [27] and an SDN-based architecture for providing QoS to high-performance distributed applications. [28] The auction-based resource management scheme provides an online approach by means of a non-cooperative game theory. It achieves gains of up to 5 x reduction in transmission delays, but it does not focus on cases where different types of flows are active in the network. The architecture to develop a QoS provisioning, presented by Oliveira et al., [28] assumes that network operators implement specific QoS levels in the network topology whereby in our case, we optimize the existing best-effort flows (and the corresponding QoS levels) over the network without pre-configured QoS levels.
The most relevant studies are summarized in Table 3.1. In our previous article, [29] we have utilized both online and offline approaches to provide bandwidth guarantees for emergency flows and maximize the total rate of best effort flows. The offline approach optimized all existing emergency and best-effort flows while the online approach routed, based on a weighted shortest path algorithm, and allocated sub-optimally new incoming flows through a greedy heuristic in between offline batches. In this chapter however, we recreated the previous solution as a containerized framework, allowing us to also evaluate the distributed behaviour of our framework in different network topologies. As in the previous solution, only drop meter policies are used in this chapter because the current OpenFlow versions [30] do not support other policies such as 2-color-marking [31] and 3-color-marking. [32].

3.3 Problem Description and Formulation

In this section, the problem described in Section 3.1 is analyzed in detail. Afterwards, a linear model to solve this problem is presented, aiming to guarantee emergency flows while the best-effort flows are optimized over

Table 3.1: Summary of related research

Reference	Offline/Online	Objective	Path Selection	Evaluation
Akella et al.[18]	Online	Satisfy the QoS requirements of the QoS flows	Greedy	Geni Testbed [33]
Pinto et al.[22]	Online	Admission control and traffic management	Sink tree	Mininet [34]
Krishna et al.[23]	Online	Satisfy the minimum bandwidth requirements of QoS flows	Widest shortest path	Open vSwitch [35] and physical switches
Morin et al.[24]	Online	Guarantee bandwidth of QoS flows	SAMCRA [36] and Dijkstra	Mininet and physical switches
Samani et al.[26]	Online	Maximize the number of streaming sessions and bandwidth provisioning	Two ILP problems	Mininet
Previous Paper[29]	Online & Offline	Maximize the total rate of the best effort flows and guarantee bandwidth of high priority flows	Dijkstra (online), ILP problem (offline)	Mininet & Physical switches
This Paper	Online & Offline	Maximize the total rate of the best effort flows and guarantee bandwidth of high priority flows in a distributed manner	Dijkstra (online), ILP problem (offline)	Mininet

the remaining bandwidth in the network. This approach is designed for topologies where all the network flows are gathered based on prior knowledge or predictions and no new flows will be created. In a more realistic case, where new flows arrive dynamically, a second approach, further called the online approach, is described. It combines the solutions from the linear model, further called the offline approach, with a sub-optimal solution to handle new incoming flows.

3.3.1 Problem description

Within an SDN network, OpenFlow-enabled switches are connected to one or more SDN controllers and different best-effort flows in each of the switch flow tables are responsible for the correct routing of the network traffic. A flow is described using a tuple `<source, destination, class>` whereby the class describes the traffic class of the flow based on a priority value and the corresponding lower and upper bound bandwidth rates. In an emergency situation where e.g. a video feed must be transferred over the network, emergency flows will be requested for prioritizing this traffic. These emergency flows need to be satisfied by guaranteeing the requested bandwidth while the remaining bandwidth of the network should be allocated to the other best-effort flows. The priority of the traffic classes will be used to optimize the best-effort flows where a higher priority requires a larger share of the available network bandwidth.
This chapter proposes a solution to maximize the total input rate of the best-effort flows in the network while the requested rates of the emergency flows are satisfied and the bandwidth capacity constraints of the network are respected. The assumption is made that the requested rate for the emergency flows is not higher than the total available rate in the network. A linear model, LP, is defined aiming to solve this problem. This Lineair Programming (LP) formulation uses the principles of flow splitting, allowing flows to be separated over different links which optimize the bandwidth resource allocation. [37] The packet reordering effect that can occur when using flow splitting, can be mitigated using hash-based splitting and packet tagging. [38] However, flow splitting is not supported by every OpenFlow-enabled switch, and a more generalized linear model is needed. Therefore, a second formulation, ILP, is defined where flows cannot be split up and each flow needs to be assigned to a single path from source to destination. Both offline models are evaluated afterwards.
The formulations of these two offline models are provided in Section 3.3.2 and Section 3.3.3. Notations used in the formulations are summarized in Table 3.2. Some described constraints contain a multiplication of a continuous and a binary variable and because this cannot be directly solved

by state-of-the-art solvers, they need to be linearized first. These formulations will optimize the best-effort flows over the remaining bandwidth that is not used by emergency flows. In case the offline models are not able to run in real-time, the online approach manages new incoming flows in between offline batches, providing the shortest (but possible sub-optimal) path with a greedy-based solution to allocate bandwidth to these new flows.

3.3.2 The ILP formulation

The ILP formulation will maximize the sum of the traffic rates of the best-effort flows, multiplied by their assigned weights over the remaining bandwidth after allocating the emergency flows. This is illustrated in (3.1), subjected to the constraints [(3.2) - (3.5)] and explained further below.

$$\max \sum_{i \in B | \{u = Source(i), (u,v) \in E\}} W_i \times Z^i_{u,v} \tag{3.1}$$

Subject to:

$$\sum_{(u,v) \in E} y^i_{u,v} - \sum_{(v,u) \in E} y^i_{v,u} = \begin{cases} 1 & u = Source\,(i) \\ 0 & \text{otherwise} \\ \text{-1} & u = Destination\,(i) \end{cases} \quad \forall u \in V,\ i \in F \tag{3.2}$$

$$\sum_{i \in B} y^i_{u,v} \times R^i + \sum_{i \in M} y^i_{u,v} \times \tau_i \leqslant Cap^{(u,v)} \tag{3.3}$$

$$\sum_{(u,v) \in E} y^i_{u,v} \leqslant 1 \ \ \forall u \in V,\ i \in F \tag{3.4}$$

$$\sum_{(v,u) \in E} y^i_{v,u} \leqslant 1 \ \ \forall u \in V,\ i \in F \tag{3.5}$$

$$R^i \in \left[minRate^i,\ maxRate^i\right] \tag{3.6}$$

$$y^i_{u,v} \in \{0, 1\} \tag{3.7}$$

$$Cap\,(u, v) \geqslant \tau_i \ \ \forall i \in M \tag{3.8}$$

(3.2) is the flow conservation constraint, guaranteeing a path from source to destination. (A.3) enforces the capacity limit of each physical link and (3.4) and (3.5) are used to prevent loops as much as possible. (3.6) and (3.7) specify the bounds for the assigned rate and whether traffic is passing through link `(u, v)`. Finally in (3.8), the assumption is made that the network is at least able to handle all requested emergency flows.

Table 3.2: Notations summary

Variables	
$y^i_{u,v}$	Equals 1 if the traffic for flow i passes through link (u, v)
R^i	The rate assigned to best effort flow i
Parameters	
$F \equiv M \cup B$	Set of all flows
M	Set of all emergency flows
B	Set of all best effort flows
$G = (V, E)$	The graph of the network. V is the set of nodes and E is the set of physical links. All links are bidirectional with different capacity in each direction
$Z^i_{u,v}$	The rate of flow i on link (u, v)
$Cap^{(u,v)}$	The bandwidth of the link between u and v, in the direction from u to v
W_i	The weight assigned to flow i based on the traffic class it belongs to
τ_i	The requested rate for emergency flow i
$minRate^i$ $maxRate^i$	The lower bound and upper bound for the rate of flow i based on the traffic class it belongs to.
$Source\,(i)$ $Destination\,(i)$	The source and the destination of flow i

The second constraint contains a multiplication of a continuous and a binary variable as in $\sum_{i \in B} y^i_{u,v} \times R^i$. The constraint can be linearized as follows:

$$Z^i_{u,v} \leqslant Cap^{(u,v)} \times y^i_{u,v} \tag{3.9}$$

$$Z^i_{u,v} \leqslant R^i \tag{3.10}$$

$$R^i + Cap\,(u,v) \times y^i_{u,v} - Z^i_{u,v} \leqslant Cap^{(u,v)} \tag{3.11}$$

$$Z^i_{u,v} \in \left[0,\ \ maxRate^i\right] \tag{3.12}$$

3.3.3 The LP formulation

The LP formulation will use the principles of flow splitting to solve the described problem. The main objective is again to maximize the sum of the traffic rates of the best-effort flows multiplied by their assigned weights over the remaining bandwidth after allocating the emergency flows. This is illustrated in (3.13), subjected to constraints [(3.14) - (3.16)] and explained further below.

$$\max \sum_{i \in B} W_i \times R^i \tag{3.13}$$

Subject to:

$$\sum_{(u,v) \in E} y^i_{u,v} - \sum_{(v,u) \in E} y^i_{v,u} = \begin{cases} -R^i & u = Source\,(i) \\ 0 & \text{otherwise} \\ -R^i & u = Destination\,(i) \end{cases} \qquad \forall u \in V,\ i \in B \tag{3.14}$$

$$\sum_{(u,v) \in E} y^i_{u,v} - \sum_{(v,u) \in E} y^i_{v,u} = \begin{cases} \tau & u = Source\,(i) \\ 0 & \text{otherwise} \\ -\tau_i & u = Destination\,(i) \end{cases} \qquad \forall u \in V,\ i \in M \tag{3.15}$$

$$\sum_{i \in F} y^i_{u,v} \leqslant Cap^{(u,v)} \tag{3.16}$$

$$R^i \in \left[minRate^i,\ maxRate^i\right] \tag{3.17}$$

$$y^i_{u,v} \in \mathbb{R}_{\geqslant 0} \tag{3.18}$$

$$Cap\,(u,v) \geqslant \tau_i \ \ \forall i \in M \tag{3.19}$$

(3.14) and (3.15) are the flow conservation constraints for the best effort and emergency flows respectively. (3.16) enforces the bandwidth capacity limits of physical links. The LP formulation is solvable in polynomial time. [39, 40]

3.3.4 Online approach

In a realistic scenario, the LP or ILP model runs in batches in order to continuously optimize the network topology. When a new flow arrives in between this batch of the linear model, it should be handled appropriately in order to avoid long delays in assigning this new flow. Therefore, the online approach handles new incoming flows through a sub-optimal solution in between the offline batches. The shortest path between source and destination is determined by using a weighted shortest path algorithm based on Dijkstra's algorithm. [17] A new incoming emergency flow will obtain its requested bandwidth while newly arriving best-effort flows will be temporary assigned to the average best-effort traffic class. In case there is no bandwidth available for the new flow, a greedy heuristic determines which other best-effort flows should be decreased in bandwidth until the offline batches optimizes this again. This is illustrated in Algorithm 1 and the complete online approach is described in Algorithm 2. Note that the solution from the online approach is only temporary because it will be replaced with the optimal results from the linear model.

Algorithm 1 Greedy heuristic

```
τ ← requested bandwidth
C ← traffic classes sorted by priority (low to high)
for i in count(C) − 1 do
    a ← τ/2
    τ ← τ/2
    band[i] ← a
end for
band[count(C) − 1] ← τ
for each traffic_class in C do
    i ← index
    s ← number of meters in traffic_class
    for each meter in traffic_class do
        meter ← meter − band[i]/s
    end for
end for
```

3.4 Framework Design

In this section, the microservice based framework design is explained. First, the different architectural components are identified. Afterwards the different components are prototyped in order to create the proposed framework.

Algorithm 2 Online approach

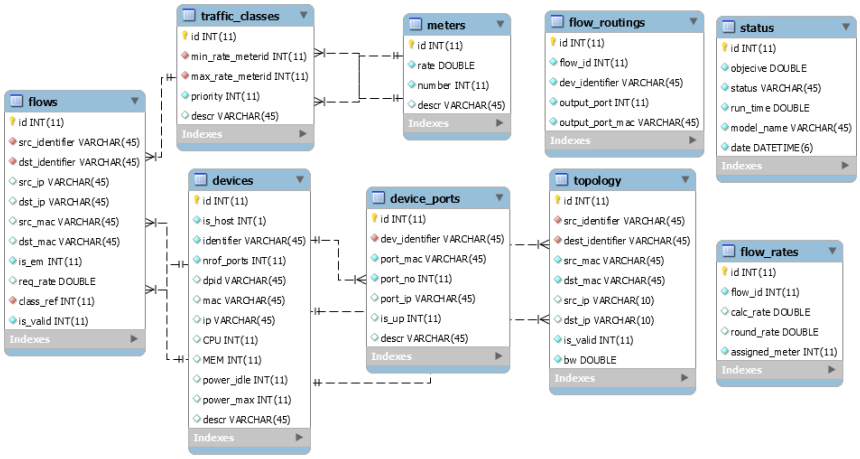
```
R ← average best-effort traffic rate
B ← best-effort flows
while batch is running do
    X ← new incoming flow
    if X is emergency then
        τ ← requested bandwidth by X
    else
        τ ← R
    end if
    if τ is not available then
        apply_greedy_heuristic()
    end if
    apply_flows()
end while
run_batch()
```

3.4.1 Architectural components and tasks

To be able to identify the microservices-based architecture of our proposed framework, different components need to be identified based on the problem described in Section 3.1 and Section 3.3.1. The main goal is to manage SDN network topologies, so an SDN controller is the first component of our architecture. Next, the offline and online problem should be able to manage existing and new incoming flows, and will be called the solver. The offline problem requires network topology information in order to calculate the optimal solution for our problem, a data store is thus needed. Finally, a REST API is required in order to communicate with and manage the different components. An overview of the different components is depicted in Figure 3.3.

Every component is responsible for a specific set of tasks. The controller component is responsible for the management of the corresponding network topology. It will handle new incoming flows by forwarding them to the solver in order to receive a correct path and traffic class. The controller will also update the database with the discovered topology and when the solver notifies the controller that a new solution is available, the controller pulls this new information from the database. The main task of the solver is thus to optimize the current flows in the current operational network topology, both with the online and offline method described in Section 3.3. Finally, the REST API component creates an API in order to manage and retrieve information from the solver and the controller components. It is also responsible for managing multiple instantiations of this architecture, as

Figure 3.2 Topology of the MySQL database. The tables devices, device_-ports, meters, topology and traffic_classes are filled in based on the network topology. The table flows contains the required flows in the topology and flow_rates and flow_routings contain the optimized best-effort and emergency flows after solving the offline problem.



will be explained in the next section.

3.4.2 Framework prototype

Now that the different architectural components are identified, each component has been prototyped. The different components are containerized using Docker CE Version 19.03. [41] The controller component is instantiated with a custom Ryu SDN Controller [42] and an API implemented with the Python Flask framework [43] in order to communicate with the other components. The solver component is split up into two containers, the online problem is responsible for the communication with the controller, calculating the shortest path from source to destination for new incoming flows and allocating bandwidth based on the greedy heuristic, both explained in Section 3.3.4, and finally it runs the offline model in batches. The offline model pulls the topology information from the database and stores the results. The offline model is based on openJDK [44] version 8 update 181 and IBM ILOG CPLEX [45] v12.7. The database component is instantiated as a MySQL version 8.0.18 database [46] and the table design is illustrated in Figure 3.2 and further explained.
The *flows* table contains the different flows in the network topology. Each flow is connected to a specific traffic class in the *traffic_classes* table and

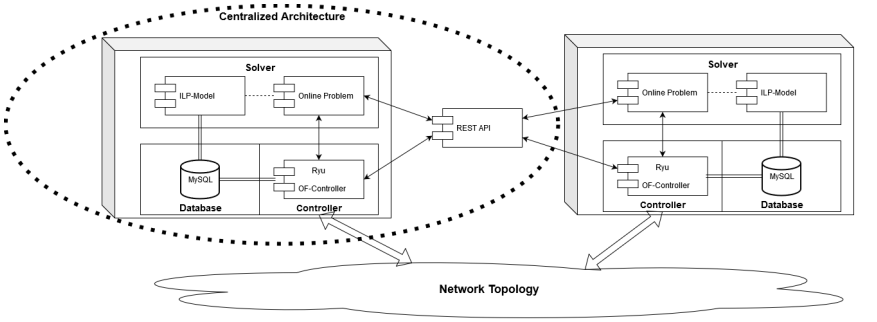
a traffic class is connected to two bandwidth meters, one for specifying the minimum rate and one for the maximum rate. Each flow is also twice connected to the *devices* table, one for specifying the source and one for the destination. This devices table also contains the different switches in the network topology. Each device has one or more ports, enlisted in the *device_ports* table and the links between the different devices, forming the actual network topology, is stored in the *topology* table. The *flow_routings* table contains the different hops per flow and the *flow_rates* table contains the assigned meters per flow. These two tables are filled in by the offline model. The *status* table contains the practical information about the different calculations performed during the processing of the offline model.

The proposed framework can be used both centrally and distributed. In case of a centralized architecture, illustrated in Figure 3.3 within the dotted circle, only one controller is responsible for the whole network and the network topology is thus added completely to the MySQL database. This approach is similar to the approach used in our previous paper [29]. A distributed approach enables the network topology to be managed by different controllers and is also a more realistic case, as for example different operators are responsible for the entire network. This is shown in Figure 3.3 and further explained. The two controllers will add a special virtual switch called `xx` to connect it with the border switches[1] of their part of the network. When the online or offline problem calculates flow routes going over multiple network providers, the controllers detect that the `xx` switch is part of the route, and ask the other controller(s) to check which of their border switches are connected with the `xx` switches in their part. Once the next part is found, the controller constructs a path from the border switch under its' control to the border switch in the next part, allowing the network traffic to be sent over different parts in the topology. The connections between the border switches in the different parts of the network topology are based on prior knowledge.

This distributed architecture allows to calculate optimal paths for best-effort flows after guaranteeing the bandwidth for emergency flows. In case when the network topology is great, or the hardware resources to run the framework are limited, the framework offers a division of the topology in several smaller parts in order to calculate the necessary results.

[1] A border switch is a switch in the current part of the topology that is connected to a switch in another part of the topology.

Figure 3.3 The architectural components and the instantiation of our proposed framework. The dotted circle illustrates a centralized architecture while the whole figure illustrates a distributed approach where two network operators collaborate, both having their own solvers, controllers and data stores.



3.5 Implementation, Simulation and evaluation

In this section, the proposed framework is evaluated in three ways. First, the offline models described in Section 3.3.2 and Section 3.3.3 are evaluated by simulation. Next, the framework is implemented and deployed on different systems in order to evaluate the distributed behaviour on an emulated network topology. Finally, the framework is evaluated on a smaller scale together with a practical environment.

3.5.1 Simulation Environment

The proposed offline models are first validated using simulations. The evaluated topology, as shown in Figure 3.4, consists of 16 ingress/egress points of traffic and 32 switches. Switches 16-30 are backbone switches and the backbone network has the same topology as the Internet2 network. [47] This topology is used to simulate a provider network catering to about 2050 flows. Switches 31-38 are mobile base stations and the ingress/egress points attached to them represent mobile users. Switches 39-46 are DSL switches and the ingress/egress points attached to them represent DSL users. The bandwidth of each the backbone network link is 40 Gbps bidirectional. The specifications of the network scenario are summarized in Table 3.3. IBM ILOG CPLEX v12.7 is used to implement the models and the simulations are executed on a server with 2 Xeon E5-2690 v4 CPUs operating at 2.6GHz with 16GB of memory.
We defined 3 traffic classes with ranges [0, 25000], [0, 10000] and
[0, 5000] Kbps with priorities of 100, 50 and 10, respectively for best effort

Figure 3.4 The simulation topology based on the Internet2 network.

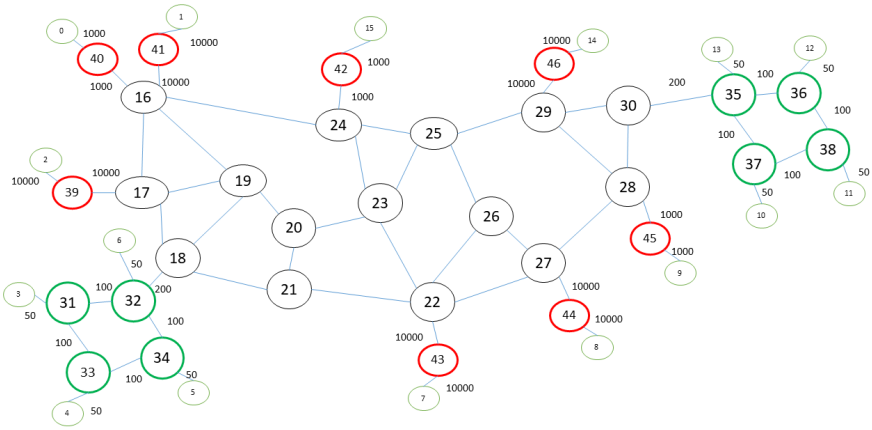


Table 3.3: Specification of the network scenario

Source/Destination of All Flows	0 - 15
Backbone Network (40 Gbps)	16 - 30
Source/Destination Emergency Flows	{0, 2, 3, 5, 7, 8, 11, 13, 14}
DSL Network	39 - 46
Mobile Network	31 - 38

flows. Moreover, the requested rate of each emergency flow was randomly chosen from set $\{25000, 10000, 5000\}$ Kbps because of the variation in types of emergency network traffic. Each best effort flow was randomly assigned to a class. Each evaluation result is the average of 30 simulation runs.

3.5.2 Simulation Evaluation - Results

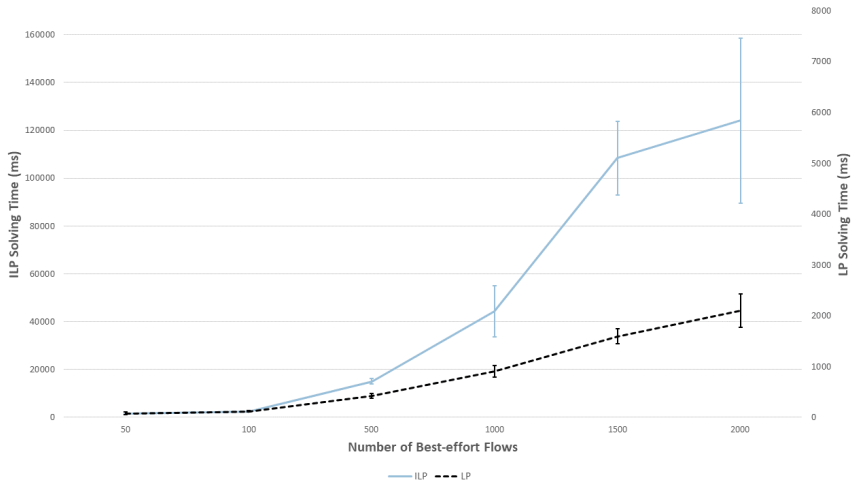
The performance of the two models is compared in Figure 3.5. By increasing the number of best effort flows, the solving time increases in both models. However, the increase rate of the ILP model is exponentially higher than for the LP model. For 2000 best effort flows along with 50 emergency flows, the ILP model solves the problem in almost two minutes. It is worthwhile to mention that the solving time of the ILP model can further be decreased by up to one order of magnitude when using acceleration methods such as the novel algorithm based on the Benders decomposition method as described in Behrooz et al. [48]

To investigate the operational details of the models, we first generated 500

Table 3.4: Comparison of the LP and ILP model simulation results

	LP Model	ILP Model
Solving Time-Before(ms)	484	18408
Solving Time-After(ms)	484	15210

Figure 3.5 The solving time of the ILP and LP models. Standard deviations are shown in the form of error bars

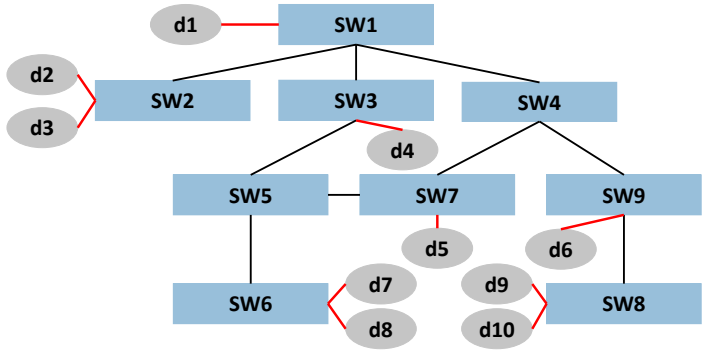


best effort flows and solved both the ILP and LP models. After that, we added 50 emergency flows and solved the problems again. Both models reported the same optimal values before and after adding the emergency flows which means that the same result is achieved by both models and the LP model solved the problem 30 times faster than the ILP model. After adding the emergency flows, the models decreased the rate of best effort flows to allocate the requested bandwidth of the emergency flows which resulted in a lower optimal value. A summary of the results is shown in Table 3.4.

3.5.3 Prototype Implementation

To implement the proposed framework, both an emulated network topology (as illustrated in Figure 3.4) and a practical network topology consisting of Zodiac SDN switches, [49, 50] depicted in Figure 3.6 are used. The emulated network topology, used to evaluate the distributed approach, is implemented using Mininet version 2.3.0d6 [51] and contains 420 flows. The default installation of Mininet only supports OpenFlow version 1.0, but

Figure 3.6 Topology of the evaluation environment. Sw1 is a Zodiac GX switch, sw2 - sw9 are Zodiac FX switches and d1 - d10 are Raspberry Pi's 3.



because our framework will use OpenFlow meters to implement the different traffic classes, OpenFlow version 1.3 or higher is required. In order to let Mininet use OF 1.3, the CPqD switch [52] must be installed together with Mininet.[2] The distributed approach is evaluated using 2 systems, one with 24GB RAM and 2 CPUs, each with 6 cores and hyper-threading enabled running at 2.4GHz[3] and one with only 2GB RAM and 1 CPU and 1 core (without hyper-threading) running at 1.12GHz. The practical network topology is built with 1 Zodiac GX switch [50] (sw1), 8 Zodiac FX switches [49] (sw2 - sw9) and 10 Raspberry Pi's model 3B (d1 - d10). The Zodiac GX has an uplink of 1 gbps while the Zodiac FX switches have an uplink of 100 mbps. The used traffic classes are illustrated in Table 3.5 and the requested bandwidth rates based on destination are summarized in Table 3.6. OpenFlow v1.3 meters were used to specify the upper bound and lower bound rates of each traffic class. Note that with the provided meters in this prototype, the offline model will rather assign the meter with 0 kbps bandwidth to flows with a lower priority in case there is a shortage. When more meters per traffic class are allocated, a downgrade is possible, but this is currently not implemented. The used system to run the framework has 16GB RAM and 4 cores running at 2.8GHz. Because the same framework is used for both the emulated and the practical topology and because the Zodiac switches do not support flow splitting, the offline model is implemented with the slower ILP model.
Assume $\{ m_1, m_2, \ldots, m_n\}$ are n defined meter rates and $m_i \leqslant m_{i+1}\ \forall i$.

[2] To install mininet with the CPqD switch, use the following command: `mininet/util/install.sh -n3f`

[3] The command `nproc -all` outputs 24 (2 CPUs x 6 cores x 2 threads per core). We will use 24 processing units in the remainder of this chapter to refer to this server.

Table 3.5: Traffic classes (all in kbps)

Id	Name	Minimum Rate	Maximum Rate
1	High Priority	0	25000
2	Normal Priority	0	10000
3	Low Priority	0	5000

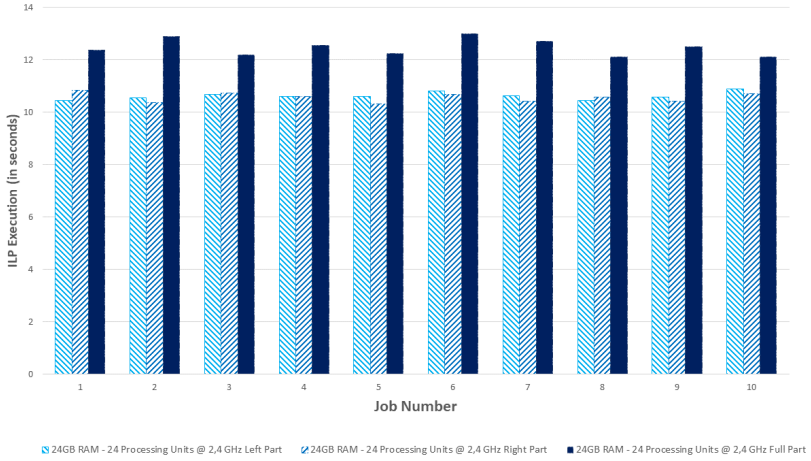
Table 3.6: Requested rates per flow based on the destination. Rates between 0 and 4999 kbps are part of traffic class 3, rates between 5000 and 9999 kbps are part of traffic class 2 and rates higher dan 10000 kbps are part of traffic class 1.

Destination	Traffic class	Destination	Traffic class
d1	3	d2	3
d3	3	d4	3
d5	3	d6	2
d7	2	d8	2
d9	2	d10	1

The weight of best effort flow i is calculated by $\left\lceil P_i \times \frac{m_n}{m_1+1} \right\rceil$ in which P_i is the priority of the class that the flow belongs to and $\lceil x \rceil$ is the ceiling function.

When a new flow arrives, it is added to the database by the controller. When the previous batch of the offline model is finished, the online model runs it again after a certain specified amount of time or when the previous calculation is done. The ILP model reads the database information, solves the problem and stores the results in the database. The output of the ILP is the assignment of each flow to one meter and the routing of flows over the network. To assign a flow to a meter, whether it be best effort flows or emergency flows, the implementation rounds down the calculated rate to the nearest defined meter rate. Based on the simulation results summarized in Section 3.5.2, the ILP model provides optimal results with a high number of flows but not in real-time. To combat this, we run the offline model consecutively while the online approach is used to route and to apply the corresponding meter to new incoming flows. Best-effort flows will be assigned to the average meter with 10000 kbps while emergency traffic will be assigned to their requested rate. To decrease the impact on the current best-effort and emergency flows, a greedy heuristic is applied to reassign available bandwidth from other best-effort flows, based on their priority.

Figure 3.7 The ILP execution time for the left part, right part and full part on a server with 24GB RAM and 24 processing units, each running at 2.4GHz.

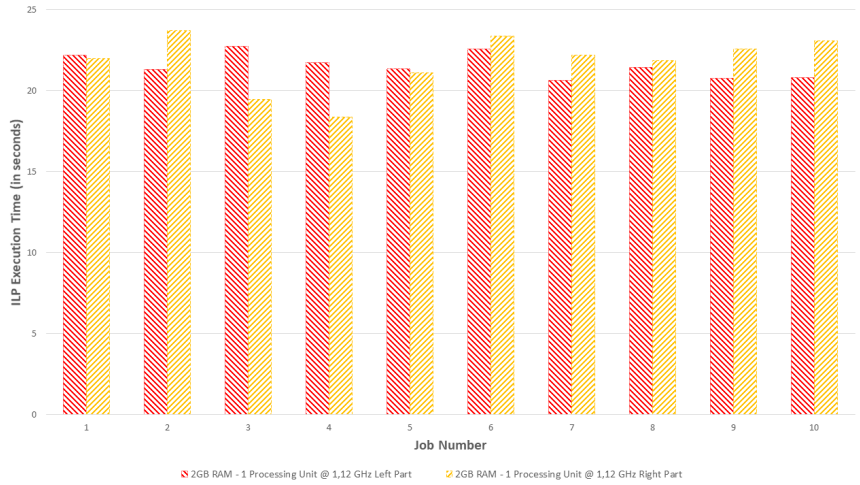


3.5.4 Distributed architecture - evaluation

The distributed architecture with the emulated network topology is first evaluated on the system with a lot of resources (24GB RAM and 24 processing units in total). The network topology is split up into two parts, further called the left part and the right part. The left part contains switches *16-21, 23, 24, 31-34, 39-42* and the right part contains switches *22, 25-30, 35-38, 43-46*. The ILP model is executed 10 times for both parts and the results are visualized in Figure 3.7. The ILP execution time for the whole network topology (full part) is also added in this figure. It is clear that the division of the network results in a speed-up of about 15%. However, the flow routing results of the two smaller parts differ from the full part but the objective from the ILP model is the same. This means that there are different routes in both cases, and with the distributed architecture it is possible that a flow is not routed along its shortest path, but this is without any noticeable delay. Because the objective from the ILP model is the same, the same optimization is achieved in both cases, meaning that the flows received the same traffic classes.

Next, the distributed architecture is evaluated on a system with fewer resources (2GB RAM and 1 processing unit), whereby the network topology is again split up into the same two parts. The results of this evaluation are illustrated in Figure 3.8. In comparison with the other server, the results of the full part are not included in this figure, because the calculation

Figure 3.8 The ILP execution time for the left part and right part on a server with 2GB RAM and 1 processing unit, running at 1.12GHz. Note that the ILP execution for the full part had not enough memory and is thus not visualized.

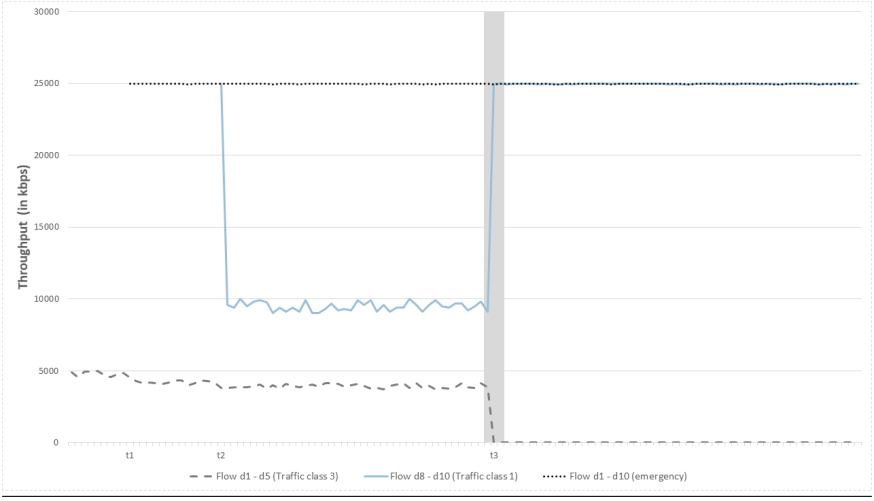


was not possible due to the lack of memory. This already illustrates the importance of the distributed architecture, because most SDN controllers have limited resources in practice. Both parts achieve the same objective as in the evaluation on the other server, but it takes about 51% more time to come to a solution.

3.5.5 Evaluation of the practical network topology - Example scenario

The framework with the centralized architecture is also evaluated on a practical network topology to study the behavior of the online approach and the findings are illustrated in Figure 3.9. When the batch calculation is running, the online approach will handle the new incoming flows. The flow responsible for the traffic going from d1 to d5, which is part of traffic class 3, is already allocated together with 87 other flows. Next at time t1, a new incoming emergency flow going from d1 to d10 is added to the network, with a requested bandwidth of 25,000 kbps. Because of its priority, the requested bandwidth is allocated and the greedy heuristic reduced the bandwidth from the other best-effort flows. The flows part of traffic class 3 have an average decrease of 284 kbps. Afterwards at time t2, a flow going from d8 to d10 is added, which is part of traffic class 1. As this is a best-effort flow, the average best-effort meter with a bandwidth of 10,000 kbps

Figure 3.9 Throughput of 2 best-effort flows and 1 one emergency flow. At time t1, the emergency flow is added and assigned by the online approach. At time t2, another best-effort flow is added and assigned by the online approach. At time t3, the offline batch has calculated and applied the optimal solution.



is allocated. The greedy heuristic again determines the bandwidth for each best-effort flow without impacting the current emergency flows. Finally, the batch calculations (visualized by the gray vertical line at time t3 in Figure 3.9) optimizes the flows of the whole network again.

It is clear that the online approach is guaranteeing the bandwidth of the emergency flows and creates a sub-optimal solution for the new incoming best-effort flows. The sub-optimal solution has 2.76% difference per flow compared to the result of the offline batches in the whole example scenario. In some cases, this difference is 100% because the online approach does not drop any new incoming flows, while the offline batches can decide to drop a flow much faster as explained in Section 3.5.3. Afterwards, the batch calculations optimize the best-effort flows over the remaining available bandwidth not used by emergency flows. The solving time of the batch calculations before and after adding 50 emergency flows is illustrated in Table 3.7 and shows that the proposed offline model can solve small-sized networks efficiently.

Table 3.7: The ILP model results

	Before	After
Solving Time (ms)	20520.234	17489.531

3.6 Conclusion

Emergency network traffic needs to have priority over best-effort traffic during emergency situations. With the expected release of 5G, slicing concepts at network level will enable prioritization of the emergency network traffic over mobile connections. In addition, SDN principles allow to assign different QoS levels to different network slices.

In this chapter, we therefore first propose two mathematical linear models that guarantee the requested rate of emergency flows and maximize the best-effort flows over the remaining available bandwidth. The LP model uses the principles of flow splitting, which is not supported by every OpenFlow-enabled switch. Therefore, a second linear model, ILP, is proposed that is supported by most of the OpenFlow-enabled switches running version 1.3 or higher. Afterwards, an online approach is explained, handling new incoming flows in between batches of the linear model. The shortest path, based on Dijkstra's algorithm, is calculated and a greedy heuristic is applied to obtain bandwidth from the best-effort flows. When the new incoming flow is an emergency flow, the requested bandwidth will be allocated by any means, a new incoming best effort flow will be allocated with the average bandwidth of all the active best-effort flows. Finally, a microservices-based framework is discussed and prototyped. This framework is able to run both in a centralized and a distributed manner, enabling scalability over larger network topologies. The distributed approach is necessary as different network operators can collaborate in managing cross-operator flows in the network topology or when the hardware resources are limited.

The two offline models are first evaluated by simulations and the results show that both the ILP and LP mathematical problems can be used with the ILP model exhibiting plus-second execution time while the LP model works 30 times faster for 500 best-effort flows and 50 emergency flows. Next, the distributed approach is evaluated by using an emulated network. Results show that the distributed architecture is a solution in case there is a lack of resources, allowing to split up the network topology in multiple parts in order to calculate and optimize the emergency and best-effort flows. When enough resources are available, a split up of the network in two parts results in speed up around 15%. When the results of the distributed architecture are compared with the centralized architecture, it shows that different paths

are chosen for some flows, but the allocation of resources remain the same. Afterwards, the centralized framework is evaluated on an SDN network consisting of Zodiac Switches and Raspberry pi's. The Zodiac switches do not support flow splitting, so the use of the slower ILP model is obliged. The practical evaluation shows that the online problem efficiently handles new incoming flows while guaranteeing the bandwidth for all the emergency flows and providing a sub-optimal temporary solution for the best-effort flows.

Research concerning the distributed approach when three or more controllers are connected is envisaged as future work. This is because the overhead of adding a virtual switch that will connect the border switches of a specific part can be greater, possibly resulting in longer calculation times for the flow allocation. An improved network topology discovery service that is able to optimally divide the network into different parts along with better handling of inter-part flows can offer a solution.

Addendum

Note on the distributed approach: In this chapter we stated that the connection between the different network segments are based on prior knowledge. It is thus known how a network packet can go from one network segment to another, resulting in parallel computations for each network segment. In case there is no prior knowledge of the network toplogy between the different segments or when it is not known on which border switch the packet will arrive from the previous segment, a slower sequential calculation is required. This can be improved by letting all the network segments calculate multiple solutions in advance, each starting with a different border switch.

Note on the LP, ILP and online approach: Evaluations proved that the LP model is much faster (2 seconds) than the ILP model (2 minutes) but the LP cannot be used in our prototyped solution. It is important to note that the outcome of both the models are the same. Due to the longer calculation time of the used ILP model, an online approach with a greedy heuristic is introduced. The bandwidth guarantee for the emergency flows in the online approach is the same as in the offline approach with the LP and ILP models. The difference between the online and offline approach is the slightly different allocation of the best-effort flows in between the calculation of the offline approach.

References

[1] Andrew Griffin. Brussels attacks: Phone networks down and saturated after explosions at zaventem airport and metro station. *Independent*, Mar 2016. https://www.independent.co.uk/life-style/gadgets-and-tech/news/brussels-attacks-phone-networks-zaventem-airport-explosion-maelbeek-metro-live-updates-a6945571.html.

[2] Belgium: Astrid launches the next generation of its blue light mobile service - critical communications today, Oct 2017. http://www.criticalcomms.com/news/belgium-astrid-launches-the-next-generation-of-its-blue-light-mobile-service.

[3] Release 15 - 3gpp. https://www.3gpp.org/release-15.

[4] Bob O'Donnell. The evolution of 5g. *Forbes*, Nov 2019. https://www.forbes.com/sites/bobodonnell/2019/11/12/the-evolution-of-5g.

[5] Xenofon Foukas, Georgios Patounas, Ahmed Elmokashfi, and Mahesh K. Marina. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine*, 55(5):94–100, 2017. ISSN 01636804. doi: 10.1109/MCOM.2017.1600951. https://doi.org/10.1109/MCOM.2017.1600951.

[6] Peter Rost, Albert Banchs, Ignacio Berberana, Markus Breitbach, Mark Doll, Heinz Droste, Christian Mannweiler, Miguel A Puente, Konstantinos Samdanis, and Bessem Sayadi. Mobile network architecture evolution toward 5g. *IEEE Communications Magazine*, 54(5):84–91, 2016.

[7] Xuan Zhou, Rongpeng Li, Tao Chen, and Honggang Zhang. Network slicing as a service: enabling enterprises' own software-defined cellular networks. *IEEE Communications Magazine*, 54(7):146–153, 2016.

[8] Albert Banchs, Markus Breitbach, Xavier Costa, Uwe Doetsch, Simone Redana, Cinzia Sartori, and Hans Schotten. A novel radio multiservice adaptive network architecture for 5g networks. *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pages 1–5, 2015.

[9] Hassan Hawilo, Abdallah Shami, Maysam Mirahmadi, and Rasool Asal. Nfv: State of the art, challenges, and implementation in next generation mobile networks (vepc). *IEEE Network*, 28(6):18–26, 2014. ISSN 08908044. doi: 10.1109/MNET.2014.6963800.

[10] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2): 114–119, 2013. ISSN 01636804. doi: 10.1109/MCOM.2013.6461195.

[11] William Stallings. The internet protocol journal software-defined networks and openflow. *The Internet Protocol Journal*, 16(1):1–40, 2013. ISSN 0890-8567. doi: 10.1097/00004583-201007000-00001.

[12] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Fog computing: Enabling the management and orchestration of smart city applications in 5g networks. *Entropy*, 20(1):4, 2018.

[13] Murat Karakus and Arjan Durresi. Quality of service (qos) in software defined networking (sdn): A survey. *Journal of Network and Computer Applications*, 80:200–218, 2017. ISSN 1084-8045.

[14] B. Oh, S. Vural, N. Wang, and R. Tafazolli. Priority-based flow control for dynamic and reliable flow management in sdn. *IEEE Transactions on Network and Service Management*, 15(4):1720–1732, Dec 2018. ISSN 2373-7379. doi: 10.1109/TNSM.2018.2880517.

[15] J. Yan, H. Zhang, Q. Shuai, B. Liu, and X. Guo. Hiqos: An sdn-based multipath qos solution. *China Communications*, 12(5):123–133, 2015. ISSN 1673-5447.

[16] Y. Zhang, Y. Tang, D. Tang, and W. Wang. Qof: Qos framework based on openflow. *2015 2nd International Conference on Information Science and Control Engineering*, pages 380–387, 2015.

[17] E. W. Dijkstra. A note on two probles in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390.

[18] A. V. Akella and K. Xiong. Quality of service (qos)-guaranteed network resource allocation via software defined networking (sdn). *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 7–13, 2014.

[19] S. Cao, M. Tong, Z. Lv, and D. Jiang. A study on application-towards bandwidth guarantee based on sdn. *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, 2016.

[20] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971. ISSN 0025-1909.

[21] S. Tomovic and I. Radusinovic. Fast and efficient bandwidth-delay constrained routing algorithm for sdn networks. *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 303–311, 2016.

[22] P. Pinto, R. Cardoso, P. Amaral, and L. Bernardo. Lightweight admission control and traffic management with SDN. *2016 IEEE International Conference on Communications (ICC)*, pages 1–7, 2016.

[23] H. Krishna, N. L. M. van Adrichem, and F. A. Kuipers. Providing bandwidth guarantees with openflow. *2016 Symposium on Communications and Vehicular Technologies (SCVT)*, pages 1–6, 2016.

[24] C. Morin, G. Texier, and C. Phan. On demand qos with a sdn traffic engineering management (stem) module. *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–6, 2017.

[25] You Lu, Baochuan Fu, Xuefeng Xi, Zhancheng Zhang, and Hongjie Wu. An sdn-based flow control mechanism for guaranteeing qos and maximizing throughput. *Wireless Pers Commun*, 97(1):417–442, 2017. ISSN 1572-834X.

[26] A. Samani and M. Wang. Maxstream: Sdn-based flow maximization for video streaming with qos enhancement. *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pages 287–290, 2018.

[27] Salvatore D'Oro, Laura Galluccio, Panayotis Mertikopoulos, Giacomo Morabito, and Sergio Palazzo. Auction-based resource allocation in openflow multi-tenant networks. *Computer Networks*, 115(318306):29–41, 2017. ISSN 13891286. doi: 10.1016/j.comnet.2017.01.010. http://dx.doi.org/10.1016/j.comnet.2017.01.010.

[28] Alexandre T. Oliveira, Bruno Jose C.A. Martins, Marcelo F. Moreno, Antonio Tadeu A. Gomes, Artur Ziviani, and Alex Borges Vieira. Sdn-based architecture for providing quality of service to high-performance distributed applications. *International Journal of Network Management*, (March):1–21, 2019. ISSN 10991190. doi: 10.1002/nem.2078.

[29] Jerico Moeyersons, Behrooz Farkiani, Bahador Bakhshi, Seyyed Ali Mirhassani, Tim Wauters, Bruno Volckaert, and Filip De Turck. Enabling emergency flow prioritization in sdn networks. *CNSM2019, the 15th International Conference on Network and Service Management*, pages 1–8, 2019.

[30] OpenFlow Switch Specifications. 1.5. 1. *Open Networking Foundation*, 3, 2015.

[31] Understanding cos two-color marking - techlibrary - juniper networks, Jun 2019. https://www.juniper.net/documentation/en_US/junos/topics/concept/cos-ex-series-two-color-marking-understanding.html.

[32] Stephen Haddock. Frame metering in 802.1 q, 2013.

[33] Geni, Apr 2019. https://www.geni.net/.

[34] Mininet overview - mininet, . http://mininet.org/overview/.

[35] Open vSwitch, . https://www.openvswitch.org/.

[36] P. Van Mieghem and F. A. Kuipers. Concepts of exact qos routing algorithms. *IEEE/ACM Transactions on Networking*, 12(5):851–864, 2004. ISSN 1063-6692.

[37] Daphne Tuncer, Marinos Charalambides, Stuart Clayman, and George Pavlou. Flexible traffic splitting in openflow networks. *IEEE Transactions on Network and Service Management*, 13(3):407–420, 2016. ISSN 19324537. doi: 10.1109/TNSM.2016.2580666.

[38] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2): 17–29, 2008.

[39] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980. ISSN 00415553. doi: 10.1016/0041-5553(80)90061-0.

[40] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[41] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[42] FUJITA Tomonori. Introduction to ryu sdn framework. *Open Networking Summit*, 2013.

[43] Miguel Grinberg. *Flask web development: developing web applications with python*. O'Reilly Media, Inc., 2018.

[44] Openjdk. https://openjdk.java.net/.

[45] Ibm ilog cplex optimization studio. https://www.ibm.com/be-en/marketplace/ibm-ilog-cplex.

[46] Paul DuBois. *MySQL*. New riders publishing, 1999.

[47] Internet2 network infrastructure topology, 2018. https://www.internet2.edu/media/medialibrary/2019/04/10/I2-Network-Infrastructure-Topology-All-legendtitle.pdf.

[48] Behrooz Farkiani, Bahador Bakhshi, and S. Ali MirHassani. Stochastic virtual network embedding via accelerated benders decomposition. *Future Generation Computer Systems*, 94:199–213, 2019. ISSN 0167-739X.

[49] Paul Zanna. Zodiac fx, 2019. https://northboundnetworks.com/collections/zodiac-fx/products/zodiac-fx.

[50] Paul Zanna. Zodiac gx, 2019. https://northboundnetworks.com/collections/zodiac-gx.

[51] Rogerio Leao Santos De Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6. IEEE, 2014.

[52] Eder Leao Fernandes, Elisa Rojas, Joaquin Alvarez-Horcajo, Zoltan Lajos Kis, Davide Sanvito, Nicola Bonelli, Carmelo Cascone, and Christian Esteve Rothenberg. The road to bofuss: The basic openflow user-space software switch. *CoR*, abs/1901.06699, 2019. http://arxiv.org/abs/1901.06699.

4

Towards cloud-based unobtrusive monitoring in remote multi-vendor environments

*In this dissertation, a UAV-as-a-Service (UAVaaS) platform is proposed, allowing to extend drone application development and management into the cloud. Because many different applications can be developed and deployed on drones, and each application can be complex, monitoring is required to keep everything running smoothly. Research Objective 4 (**C4**) describes that unobtrusive monitoring for container-based applications is necessary and therefore this chapter proposes such a system. Based on the sidecar container pattern, an unobtrusive monitoring solution is designed, prototyped and evaluated on a simulated railway passenger information system, showing an average impact of 0.02 % in terms of CPU usage on the application performance and hereby confirming Hypothesis 4 (**RH4**). Note that the prototyped solution can be used in every microservice based environment and thus afterwards can be used in our UAVaaS platform.*

⋆⋆⋆⋆⋆

J. Moeyersons, S. Kerkhove, T. Wauters, F. De Turck and B. Volckaert

Published in Journal of Software: Practice and Experience, September 2021.

Abstract Nowadays, many complex multi-vendor production environments, such as telecom infrastructures in smart cities or on-board passenger information systems in trains, are based on microservices and deployed in the cloud. From a service integrator point-of-view, building new solutions for these environments, which can host a large number of externally designed and developed microservices, is often complex and error-prone. This is in part due to undocumented behaviour or undocumented architectural specifications of such systems. Advanced service monitoring can offer a solution to quickly detect anomalies or unexpected service interaction behaviour during on-site integration. However, the monitoring service should not have an impact on the production environment itself. Therefore, this chapter proposes an agent-based unobtrusive monitoring platform, capable of monitoring both internally developed and externally developed services through the use of sidecar containers. It monitors state, metrics and network traffic at microservice level and the research was conducted as part of the DynAMo research project, a collaboration with various industry partners. Prototype evaluation proves that our solution has a negligible impact (below 0.02% CPU usage on average) on an existing microservice environment just as other monitoring systems like Prometheus while offering additional functionality focused on multi-vendor service integration. This makes it suitable to be deployed in complex production domains to further aid on-site integration and quickly find potential new anomalies.

4.1 Introduction

On-site integration of software offerings in an operational environment can be complex and error-prone. The runtime dependencies and services in these environments at times do not comply with the interface behaviour agreed at design time. Undocumented service behavior can give rise to unstable operational software systems which are difficult to debug. Such breaches can be observed at the level of interfaces, order of protocol interactions, performance and timing properties as well as security aspects such as confidentiality, authentication and authorisation.
In the case of cyber-physical systems, such as information systems onboard trains [1], infrastructure in a smart home and smart city [2, 3] or Internet-of-Things (IoT) systems [4, 5], some hardware and software components

are under control of different external parties. These ecosystems are highly dynamic (updates and functional changes can be installed at any time for any service), can contain software solutions from different companies and have a lot of third-party dependencies and service interactions.

These cyber-physical systems nowadays require integration of software and updates in a dev-ops manner with continuous operation of devices and applications, at times in mission critical environments (e.g. train passenger information systems which need to convey the right information to passengers at all times). This results in three main challenges. The first challenge is that this continuous integration process encounters problems at many levels, such as inside the cyber-physical system between the services operating on the devices, between the on-site platform and the self-managed back-end services, between the services in the back-end and between external parties and the self-managed services. The second challenge is that current development practices often lead to a lack of explicit system architectures describing subsystems, interfaces and protocols. System models are often out of date compared to the actually deployed subsystem, while interface and protocol descriptions often lack non-functional properties such as performance guarantees or security features and functions properties like authentication, authorisation and constraints. A third and final challenge is the mission-critical nature of the cyber-physical system (e.g., a train where passenger information systems are interconnected with main train operating software services signaling location, door status and occupancy), which does not allow intrusive interactions for inspecting or even halting or restarting the system or some of its constituent services. Interactive inspection and analysis of system performance, adherence to protocols and security should thus occur in a non-intrusive way, but preferably also on a runtime copy of the actual system.

In-depth monitoring of different, potentially multi-vendor services in production environments allows to quickly detect anomalies or unwanted behaviour. However, this monitoring must be performed in an unobtrusive manner, in order to avoid having an impact on the production environment itself.

A production environment typically consists of different services, some of which are potentially developed and managed by third parties, deployed over one or more servers. When such a third-party service is updated, errors can occur in the production domain. Therefore, this chapter proposes an unobtrusive monitoring and analysis platform to efficiently monitor remote (multi-vendor) service environments from a cloud-based backend. Different remotely configurable traffic probes, from here on called agents, are added at each link in the environment capturing interactions between the differ-

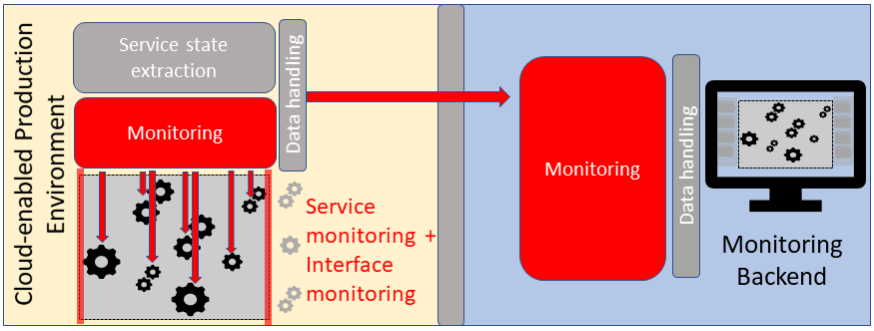
ent components, even when network interruptions and time drifts occur. The deployed agents are able to capture the state, processing, storage and memory metrics and network traffic from these services, where the analysis of the network traffic can aid in monitoring the behaviour of third-party services that communicate with the services under control of the company monitoring the system. An agent runs fully alongside an existing service imposing no integration requirements. No changes need to be made to the actual service, which is useful when the service is a black box or cannot be modified or recompiled (e.g. due to not having access to the source code). If an agent would fail for some reason, it must not have an impact on the deployed production system. As such an important design decision was to avoid the use of proxies that intercept messages and forward them to their destination.

When different services need to be monitored, a management system is needed in order to orchestrate and manage the different deployed agents. Therefore, every agent needs to have an endpoint in order to communicate with this orchestration service. An agent should also be remotely (re)-configurable in order to tune the monitoring behaviour for the specific production service at hand. The agent orchestrator is responsible for this and keeps track of all deployed agents and their configuration. An essential component in the architecture is the master clock because time synchronization between agents is of critical importance. Time between various servers and edge hardware can differ significantly. If there would be a deviation in the local time of the various recipients in a flow, it becomes very hard to reconstruct the flow of events (timestamped with local time) in a service trace. The agent orchestrator therefore time-syncs agents to the master clock allowing to accurately timestamp service interactions.

This chapter provides a solution to monitoring complex systems consisting of a mixture of controlled and uncontrolled entities (services or hardware) by focusing on the following novel contributions: (i) architectural design of a monitoring system for monitoring complex third party software ecosystems, (ii) research and prototyping of an unobtrusive and widely applicable monitoring component by using a sidecar container pattern and (iii) evaluation and a comparison to Prometheus[6] on a real-world use case. The concept of this monitoring system is illustrated in Figure 4.1.

The remainder of this chapter is organized as follows: Section 4.2 provides an overview of existing technologies, including their benefits and drawbacks. Next, in Section 4.3, the different components of the proposed platform are discussed and in Section 4.4, technological choices for the prototype are explained. Afterwards, Section 4.5 evaluates the impact of the proposed monitoring system together with a comparison to the Prometheus monitor-

Figure 4.1 Concept of an unobtrusive monitoring and analysis platform. Every gear wheel represents a microservice that needs to be monitored.



ing system while conclusions and avenues for future research are summarized in Section 4.6.

4.2 Related Work

Different service monitoring solutions for complex service deployments exist today (Telegraf [7], Prometheus [6], etc.). These solutions focus on services deployed on controlled backend environments. At the service interface side, different API monitoring solutions offer metrics about deployed APIs (Rigor [8], APImetrics [9], etc.) and operational monitoring can be baked into API gateways (Kong [10], Traefik [11], etc.). Monitoring a full cross-technology service offering integrated with third party services that cannot be altered or recompiled remains a research challenge [12].

Such type of service deployments also suffer from difficulties to assess adherence to security constraints. Current security monitoring solutions are often limited to network-level intrusion detection systems or monitoring authentication attempts. Deep inspection of distributed application-level authentication and authorisation is often missing.

Vaculin et al. [13] add an event model on top of the OWL-S standard, including error handling, which allows the description of services and their components, as well as interactions with them. This approach (like OWL-S) stems from a time in which services were relatively heavy components, contrary to current highly agile microservice approaches, and is therefore considered to be dated.

Calero et al. [14] mention in their publication on MonPaaS (a solution for the cloud consumer, not the service provider) that, at the time of writing, no real solution exists for the monitoring of rapidly changing, virtual infrastructures (cloud computing). The problems of traditional mon-

itoring they mention are twofold: a rapid life cycle of virtual assets and the need for custom monitor software. Their attempt to counter these problems, by implementing MonPaaS, is realized by attaching their solution to the communications middleware that handles all traffic between the respective cloud and consumers' and administrators' monitoring solution. This way, they can automatically respond to changes in the topology.

Pina et al. [12] proposes a nonintrusive monitoring system of microservice-based systems based on log gateway activity to register all calls to and between microservices, as well as their responses, thus enabling the extraction of topology and performance metrics, without changing source code. The authors resorted to three Netflix components to gather metrics outside of the critical path while our proposed solution uses an agent-based approach to gather metrics. Another example is presented by Noor et al. [15], which has the same architecture as Pina et al. The authors present a generic agent-based monitoring framework, where the agents are deployed as start-up packages in different VMs, making it an intrusive solution. Our proposed solution can be deployed at any time, making it more unobtrusive than the solution proposed by Noor et al.

Alhamazani et al. [16] proposes, develops and validates CLAMBS, Cross-Layer Multi-Cloud Application Monitoring and Benchmarking as-a-Service for efficient QoS monitoring and benchmarking of cloud applications hosted on multi-clouds environments. Souza et al. [17] presents an osmotic monitoring model, based on CLAMBS, for monitoring IoT systems applications decomposed as microservices and executed in an Osmotic computing environment. The proposed solution in this chapter monitors cloud and/or edge systems, as both works, but avoids eventual vendor lock-ins and adds monitoring of external third-party services.

Cinque et al. [18] presents a non-intrusive novel monitoring approach to accompany microservices logs with black box tracing to help practitioners in making informed decisions for troubleshooting. It is thus a passive tracing solution and it aims to cope with the flexibility requirements of microservices systems, but only on log level.

Version-based Microservice Analysis, Monitoring, and Visualization (VMAMV) [19] is a system that automatically detects potential design problems for microservice with multiple versions in design time, discover service anomalies for all service versions in runtime, and immediately notifies users of problems shortly after they occur. This solution focuses on the different dependencies and their corresponding version of microservices deployed in a production environment, making it able to early detect anomalies triggered by version mismatches. The monitoring of microservices itself is not handled in this chapter.

Nagios [20], a current industrial player, focuses on network analysis, alongside automated alerts. Additionally, they offer application and server monitoring with their (closed-source) Nagios XI [21]. Nagios runs on Linux and is mostly tailored to host or network monitoring. Various forks exist of the Nagios core open-source project, which may be of interest [22, 23].

Zabbix [24] is an open-source solution, offering real-time monitoring of metrics (both host and application metrics). These are collected by agents in an active or passive manner. In the passive mode, the backend server connects to the agent and polls for a certain value (e.g., host CPU load). The backend waits until the agent on the host responds with the value. Then the server gets the value back, and the connection closes. In the active mode, all data processing is performed on the agent, without the interference of the backend. However, the agent must know what metrics should be monitored. Their service monitoring requires custom written plugins.

Zenoss [25] is a SaaS solution that offers monitoring for IT infrastructure (services, network, storage, etc.). They offer root cause analysis for degraded services as well as discovering trends and dependency issues between services that may cause future problems. Their proposed solution is aided by Artificial Intelligence and can be labeled AIOps (AI for IT operations), combining DevOps with advances in AI research. They require custom monitors that connect to services through, e.g., Simple Network Management Protocol (SNMP) or Hypertext Transfer Protocol (HTTP), rendering it less suitable for use here because this cannot be deployed in an unobtrusive way, especially when dealing with third party services.

Siemens provides the MindSphere [26] platform which is an open cloud platform for IoT applications. It stores operational data of various producers, be it hardware or software. Hardware applications such as Siemens' MindConnect can be used to collect and transfer data from the field, either real-time or buffered, to MindSphere. The platform offers outlier detection, anomaly detection and can perform trend prediction. The Siemens platform is specifically aimed at industrial IoT. Similar functionality is offered by Thingworx [27], who employ AI and machine learning technologies for IoT systems in their ThingWorx analytics package.

Acronis [28] offers detailed monitoring as a service. While most targets are predefined, such as common web services or operating systems, it is possible to define own monitors through custom agents. While their offering can group targets and show meaningful information through a Graphical User Interface (GUI), it seems that it is not possible to approach black box services or include deep state analysis or replays.

Dynatrace [29] attempts to inspect the runtime of processes. It includes resource utilisation on process level, which is further than what most monit-

oring solutions offer but requires code changes in production services, which the solution proposed in this chapter aims to avoid (as some of the services are third party and cannot be adapted).
Google Cloud's operations suite [30] is a systems management platform that has a couple of properties relevant to this research. It operates on top of AWS (Amazon) and/or Google Cloud, while the solution proposed here aims to be platform agnostic (running on any platform that Docker can be installed on). Relevant properties are real-time debugging and state inspection. More commonly, Google Cloud's operations suite also allows monitoring, logging, and diagnostics of the configured services. Predefined profiles are available for commonly used platforms and programs, allowing detailed monitoring of, e.g., an Nginx web server or PostgreSQL database. Also of note, auto discovery of such services is supported.
It is clear that different technologies exist but most technologies assume full control over the services or operational platform at hand, which is not the case when developing for or integrating in existing multi-vendor service solutions. These shortcomings are for example vendor lock-in, the potential to impact operational systems when the monitoring system fails (e.g. by means of solutions proxying communications) and requiring changes (e.g. recompilation or linking external libraries) in the production environment to allow monitoring. Table 4.1 provides a comparison of all the monitoring solutions discussed above from which we can conclude that this chapter provides an unobtrusive monitoring platform that can operate in a platform agnostic manner, can be integrated in an existing production environment and which is capable of monitoring incoming network traffic from third-party services which are not in control of the production environment owner.

4.3 Architectural design

This section will explain the different components of the proposed solution. First, the functionality of a single agent are listed. Afterwards, the agent management system, responsible for the orchestration of the different agents and the collection of data, is discussed. Finally the overall architecture is presented.

4.3.1 Agent functionality

The main goal of the agent is to provide insight in the behaviour of each service. Primarily, this is executed in a non-intrusive way, by extracting or observing the desired information. Network traffic is extracted in a passive manner, as well as generic system state (CPU usage, memory usage, etc.)

Table 4.1: Related work compared to our proposed solution in terms of our main requirements: avoid vendor lock-in, unobtrusive, scalable, log-level monitoring, system monitoring, network monitoring, open-source and support for third-party services.

	Vendor lock-in	Unobtrusive	Scalable	Log-level monitoring	System monitoring	Network monitoring	Open-Source	Third-party services support	Remarks
Vaculin et al.[13]	✓	✗	✗	✓	✓	✓	✗	✗	Based on OWL-S
Calero et al.[14]	✓	✗	✓	✓	✓	✗	✓	✗	/
Pina et al.[12]	✗	✓	✓	✗	✗	✓	✓	✗	/
Noor et al.[15]	✗	✗	✓	✓	✓	✓	n.a.	✗	Source code not available
Alhamazi et al.[16]	✗	✗	✓	✗	✓	✓	n.a.	✗	Focus on QoS metrics
Souza et al.[17]	✗	✗	✓	✗	✓	✓	n.a.	✗	Extension of Alhamazi et al.
Cinque et al.[18]	✗	✓	✓	✓	✗	✗	✓	✓	/
Ma et al.[19]	✗	✓	✗	✗	✗	✓	n.a.	✗	Focus on version control
Nagios[20]	✗	✗	✓	✓	✓	✓	✓/✗	✗	Both open- and closed-source solutions
Zabbix[24]	✗	✗	✓	✓	✓	✓	✓	✗	Plugin-based
Zenoss[25]	✗	✗	✓	✓	✓	✓	✗	✗	/
MindSphere[26]	✓	✗	✗	✓	✓	✓	✓	✗	Focussed on IoT
Acronis[28]	✗	✗	✓	n.a.	n.a.	n.a.	✗	✓	Focussed on cyber-security monitoring
Dynatrace[29]	✗	✗	✓	✗	✓	✓	✗	✗	/
Google's Cloud ops. suite[30]	✓	✓	✓	✓	✓	✓	✗	✗	/
Our solution	✗	✓	✓	✓	✓	✓	✓	✓	/

which can be easily retrieved from various systems when rudimentary access is available. Secondly, when permitted (e.g. service is self-developed or is third party but open sourced), additional information can be extracted by modifying the service. This entails having the service to actively send information to the agent, or making data available in a way that would not be possible in its default state. Detailed state extraction will be intrusive (changes are required to the service in production) in order to extract insights into the inner workings of the service.

Service Interaction Monitoring: To monitor service interaction, agents capture incoming and outgoing traffic for each service. The ideal case would be a containerised service, where network traffic is relevant for the service (as most if not all communication with the containerised environment is aimed at that service or stemming from that service). However the default approach should take into account a generic network device that is located in a LAN. When faced with raw data streams, it is not trivial to reconstruct the intended data flow in the actual messages. Sensors, actuators and other low level hardware can use raw encrypted Transmission Control Protocol (TCP) streams or User Datagram Protocol (UDP) diagrams to communicate with a central service (e.g. hardware installed in trains). In this case, access to or knowledge about proprietary code and components is not trivial and custom solutions needs to be added to the agent in order to interpret the raw stream of data. In other cases when a more standardized protocol such as HTTP or Google Remote Procedure Call (gRPC) is used, the agent can automatically detect the communication content and transform it to message requests and responses.

Another problem is to identify if a certain packet is intended for the monitored service or not. In the ideal case, a *traffic flow ID* would be available in order to identify a data flow through several different services. E.g. an event is generated in a sensor which is then picked up by a service that performs a transformation and sends it on to another service or a database. Having this data flow tagged with an ID will greatly ease analysis of the network traffic. The agent needs to reassemble the various network packets. These packets can then be sent to the agent endpoint, potentially disregarding message bodies. Messages can be stored on the agent in a buffer with limited size, which can be fetched later by the agent endpoint in case more details about a specific data flow are needed. However, doing this in a reliable way may require intrusive instrumentation of the monitored service.

System State Monitoring: System state envelops the concepts of, e.g., hardware state (memory, CPU, secondary storage, temperature) or container state (replication level, disk use). An agent can collect metrics about the health of its observing container. These metrics can also be recorded

alongside the captured messages and can provide valuable insights as to why a service is unable to handle an incoming request properly, e.g., when disk storage is full or the system is memory-starved.

Runtime State Monitoring: Runtime state envelops concepts such as extracting information from a running system and using it to detect and possibly react to observed behaviors satisfying or violating certain properties. Such information is crucial to do highly contextual anomaly detection and almost always imply implementations or changes to the services running in production. These metrics can however easily be exposed through an interface such as the Promotheus exporter [31]. This type of monitoring is not always available: only when the service itself offers external access to these parameters, the agent can scrape them. An extra feature may be to analyze and parse the service container logs and send those to the agent management system as well.

Time Synchronization: Time synchronization between agents is a critical component as discussed in Section 4.1. The local time between various servers and edge hardware can differ which can make it difficult to reconstruct the flow of events in a trace, especially if there is no correlation ID. Other issues such as jumps in time when the Network Time Protocol (NTP) client has performed a resynchronization on the host would also hamper correct reconstruction of a flow, as it is entirely possible to suddenly have messages out of order or tagged with future timestamps. To mitigate this issue, the proposed system hosts its own master wall clock to which all agents periodically sync to, preventing clock drift. There are a number of protocols that can be used with varying precision from a holdover stability of 250 milliseconds per year to 3 seconds per year [32].

Offline Support: It is possible that an agent cannot reach the system endpoint to report recorded messages. The agent then needs to store all pending upstream data in a locally persisted queue that is processed when the endpoint becomes reachable again.

Remote (re)configuration: Agents should be able to fetch their configuration from the management system and periodically synchronize to ensure they are properly reconfigured. Operators might not always want to be able to access the agents remotely, so having a management system that pushes the configuration to the agents is key. This configuration also entails a set of filters on considered traffic and messages.

This remote reconfiguration encompasses the scheduling of a specific time window that needs to be recorded in full detail i.e. all data exchanges must be captured instead of sampled data exchanges. This detailed recording can be used in a later stage to reproduce service interactions in order to identify issues in a controlled environment.

4.3.2 Agent Management System

When different services need to be monitored, an agent management system is required in order to orchestrate and manage the various deployed agents. Therefore, every agent needs to have an endpoint in order to communicate with this orchestration service. These two components are further explained below.

Agent endpoint: The agent endpoint is the ingest point for all messages captured and is the (only) access point for all the deployed agents. An agent establishes a bidirectional communication channel with the endpoint to stream messages, synchronize the master clock and fulfill requests for full body messages.

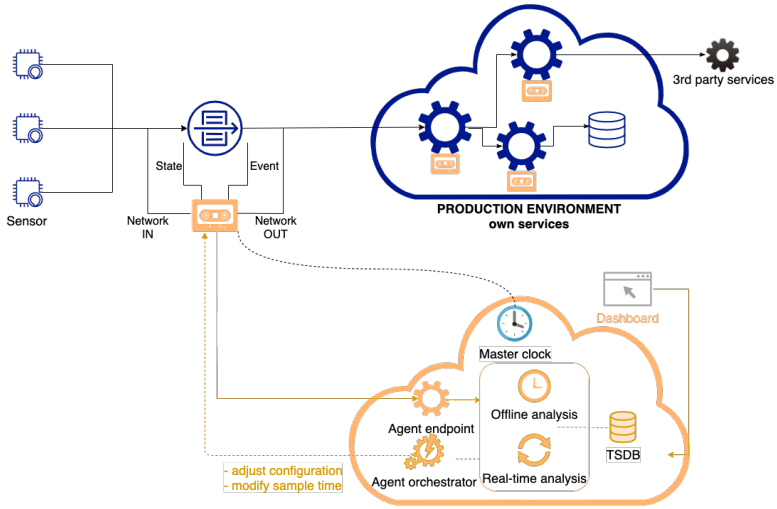
If all links within the production system contain agents, most messages are received twice: once from the agent monitoring the service that sent the message and once from the agent monitoring the service that received the message. The only exception would be messages going through 3rd party external services. To prevent storing duplicate data, the endpoint needs to match these messages, based on 1) from, to, message body hash, 2) correlation id, if available and 3) acceptable time window. While not strictly required, removing duplicates from the system early on reduces load and makes it easier to follow and replay traces. It can also give insight into the actual latency between sending and receiving messages.

The endpoint also needs to stream the received messages to the other components of the system, either through a shared buffer (e.g. Kafka) or directly through Server-Sent Events (SSE), where the latter is a standard describing how servers can initiate data transmission towards clients once an initial client connection has been established, providing a memory-efficient implementation of XHR streaming [33]. To keep the endpoint scalable so that multiple instances can be deployed, a shared buffer is probably the better option, even though it introduces latency.

Agent orchestrator: A centralized system is required to (re)configure deployed agents remotely. From the previous section we can deduce that an agent has many configurable features or parameters: which modules to load, the local transmission queue, which metrics to capture, at which rate the metrics are captured and so on. A consistent reliable connection is not always feasible, especially on edge networks. To account for connection properties, an agent could be configured when it is first installed or on-the-fly when a stable internet connection is available. For the latter, it is possible that due to changing production load the agent needs to be reconfigured to change its impact on the production environment.

The agent orchestrator keeps track of all deployed agents and their configuration when they periodically synchronize with the system. The system can

Figure 4.2 Detailed agent-based architecture of the proposed monitoring platform

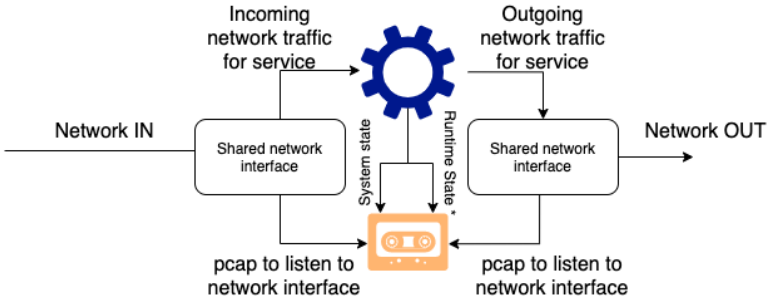


define profiles, with reduced or full tracing capabilities, that determine a set of configurations to use. This allows operators to quickly reduce potential load issues by reducing the impact agents have.
The orchestrator can also be in charge of the aforementioned master clock to ensure all agents use timestamps relative to a single clock.

4.3.3 Overall architecture

Based on the architecture described in Section 4.3.1, 4.3.2 and the *unobtrusive* requirement described in Section 4.1, it is now clear which components an agent can support, how an agent and the full prototyped architecture needs to be deployed and how every agent is managed in a complex multi-service environment. A more detailed architecture is proposed and explained in Figure 4.2.
The blue components are the services in a production environment that need to be monitored, the orange components are the agents of our proposed system, deployed as sidecar containers. An agent is deployed to monitor an existing service and capture the "Network IN" and "Network OUT" traffic, process this traffic and eventually send it to the agent orchestration service through the agent endpoint. This is visualized on the left hand side of Figure 4.2. Each microservice in the production environment, under control of the integrating company, (blue gears in Figure 4.2) will be monitored in this way. The orchestration service stores the captured information in a time

Figure 4.3 Detailed architecture of our sidecar container implementation. A monitoring agent is attached to the monitored service through the network interface, allowing to capture and monitor all incoming and outgoing network traffic.



series based data store for potential further use, such as anomaly detection (e.g. identifying faulty service interactions in specific context or state), and is also responsible for clock synchronization between agents.

Figure 4.3 illustrates the employed sidecar container pattern. An agent (displayed in orange) is attached to a microservice in the production environment through the network interface. This allows the agent to monitor the incoming and outgoing network traffic. System state monitoring data is captured by reading out the kernel data of the monitored microservice. Runtime state monitoring is only possible when the service itself offers external access to these parameters over e.g. a REST API.

4.4 Proof-of-Concept implementation

For evaluation purposes a Proof-of-Concept (PoC) in dotnetcore 3.0 [34] is implemented, which can run cross-platform and self-contained in a Docker container [35]. The proposed solution can also run on other cloud solutions such as VMs or bare metal servers but this is outside the scope of this chapter. The agent is modularized, each component is seen as a module with specific properties to configure. Modules can be built in or loaded from additional external plugins with the help of Managed Extensibility Framework (MEF) [36]. This modularization makes it easy to enable or disable certain modules of the agent when desired or when the impact of the agent is too large on the production environment hardware.

The agent is typically deployed alongside a service and its main task is to unobtrusively collect messages that are sent from and received at the service. With the help of .NET libraries such as Pcap.NET and

PacketDotNet, packets are extracted from the network interface and (in the case of TCP/IP) reconstructed into binary streams. Many protocols exist on top of this, however, our implementation limits itself to HTTP request/responses that form a message pair.
Each message consists of:

- **Local timestamp:** the timestamp of the first packet of the message.
- **Master clock timestamp:** the converted master clock timestamp equivalent.
- **From:** the IP (or hostname) and port where the message came from.
- **To:** the IP (or hostname) and port where the message was sent to.
- **Flow correlation ID:** can be filled in by custom modules into the agent processing pipeline to chain messages together.
- **Tags:** a key value pair list to add extra information to a message, that makes it easier to filter on (e.g. HTTP method, HTTP status code or message hash of the corresponding HTTP request).
- **Is anomalous:** a flag indicating whether the message is anomalous. This can be filled in by a simple anomaly detector (e.g. HTTP 500 status code is an anomaly) or can be filled in by a custom, more complex anomaly detector.
- **Captured on:** the name of the agent the message was captured on.
- **Fingerprint:** a SHA256 hash of the entire binary stream, used to match messages.
- **Agent on receiving end:** flag indicating whether the message was received at the agent's service or not.

The messages are conceived to be lightweight, holding a bare minimum of data required. The full payload of the message is also obtained but not sent through to the agent management system by default. Therefore, it is first retained in memory but with a configured limit in terms of number of messages and total size. Persisting this information to disk is an option but undesirable when the hardware is using flash storage with limited writing cycles.
Messages that are captured are sent to a processing pipeline before being sent to the agent management system. Extra modules can add additional processors to alter the message, add tags to perform actions such as early anomaly detection.

The agent also reports its status periodically to the agent management system to notify that it is still running. With this interaction, the management system also reports back any tasks that the agent has to do, such as:

- If the configuration has changed, request the newer configuration.
- Push the initial configuration if the agent did not connect to the agent management system before.
- Send the full payload for specific message hashes, if they are still available. The management system can run a larger, more resource intensive anomaly detector that can flag certain messages as anomalous. For later root cause analysis it is desirable that the full payload is also stored in a data store.

Agents are also tasked with collecting monitoring metrics of the host it is running on (system state monitoring) and metrics from the service (runtime state monitoring). To obtain metrics from the host, agents can be bundled with a separate Netdata worker process [37]. This process exposes a large amount of information through a Prometheus endpoint. This endpoint is scraped regularly by the agent and the obtained metrics are sent to the management system. The Prometheus format is chosen because of its wide availability and easily parsable format. Many languages and frameworks provide ready-made libraries to collect and expose metrics with a Prometheus endpoint.
To prevent clock drift between various hosts, the agent periodically syncs an internal clock with the masterclock in the management system and converts the timestamps of data that it sends to the master clock timestamp. The roundtrip time of this synchronization is halved and added to be more accurate.
Agents are typically deployed as sidecar containers[38] to the service containers they are observing. By tapping into the network namespace of the service, it can unobtrusively monitor packets specific for that service. Running the agent as a sidecar container prevents the agent from having an impact on the monitored service. If the agent were to crash, the service remains unaffected. One exception to this sidecar pattern is the agent that captures monitoring data from the host, which is running in a standalone container on the host network namespace.
The agent management backend consists of a gRPC endpoint, a binary format making it more efficient in terms of bandwidth and CPU usage, that agents connect to. Agent metadata, such as its known configuration state, are stored in a MongoDB database [39].

For actual data such as messages, full payload of messages and monitoring data, Clickhouse [40] is used. Clickhouse is a classic Relational Database Management System (RDBMS) instead of a time series database but can handle large amounts of data, in its default configuration, with ease.
The agent management system also provides a simple GUI for changing the configuration parameters of specific agents. Once the agent reports its status, the management system checks and schedules the configuration update as a task back to the agent.
The PoC implementation can be deployed at any time by running a simple startup script, which will spin up the management containers and will deploy and couple agents with the network interface of each other service or container that needs to be monitored.

4.5 PoC evaluation results

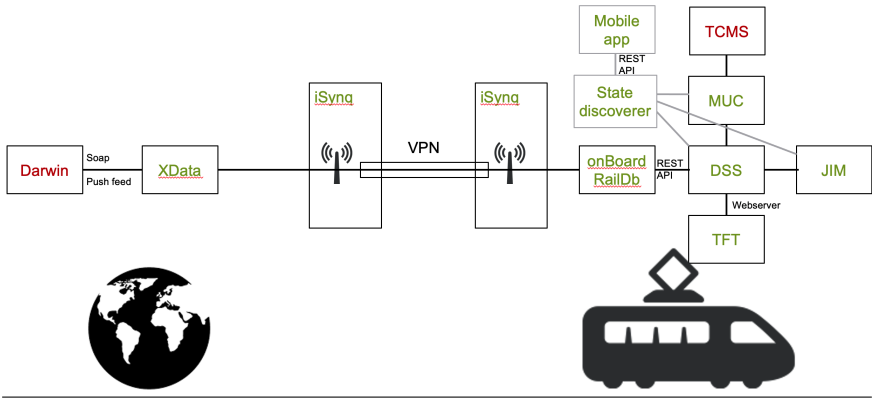
In this section, the impact of the proposed framework in terms of CPU and memory usage is evaluated, discussed and compared to another monitoring solution, namely Prometheus. First, the evaluation setup is illustrated and explained, followed by the different metrics that are important in this evaluation. Finally, the obtained results are compared and conclusions are made.

4.5.1 Evaluation setup and metrics

The proposed platform is evaluated in terms of unobtrusiveness. Our goal is to evaluate the impact of our proposed agent on a service in a complex production environment, whereby the service is first operating without monitoring additions and afterwards running with our agent monitoring the service. The evaluation setup is illustrated in Figure 4.4 and is a simulation railway passenger information system from the Televic Group[1], a company responsible for developing, manufacturing and installing top end communication systems for niche markets such as trains. This simulator is very close to the real case scenario, as it combines containers from different vendors and it takes into account the delays and struggles in the communication process when a train is continuing its journey. The services simulate a journey of one train with all necessary stops and passenger and personnel communication. This use case is an example of a complex multi vendor integration consisting of Darwin systems, systems from the train manufacturer, systems from integrator companies and multi vendor hardware. The whole simulation environment consists of services deployed in Docker containers (with a single

[1] https://www.televic.com

Figure 4.4 Simulation and evaluation setup. The items in red are third-party services.



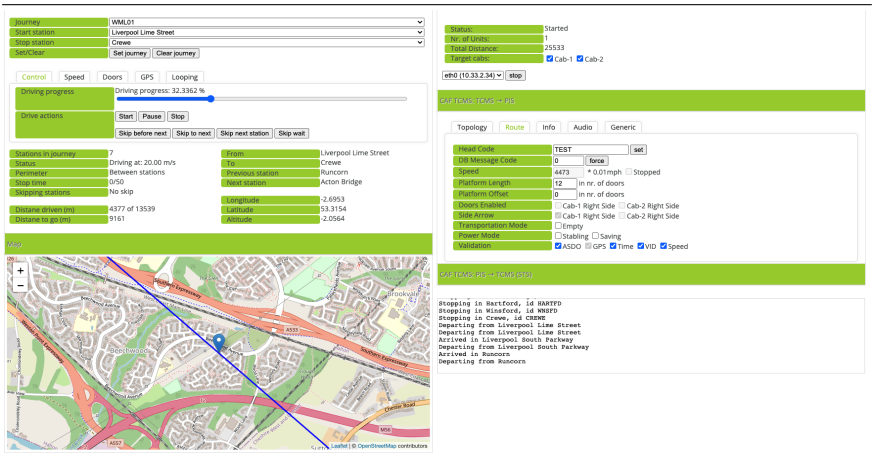
service instantiated per container). One other can conclude that other production microservice environments are working similarly.
The train part consist of following items:

- The isync server is responsible for communication with the wayside
- An onboard rail database for storing information that needs to be visualised on the train information systems
- The tcms-simulator simulates the third-party (! (()TCMS)
- The Main Unit Controller (MUC) is responsible for translating the TSMC messages.
- The Digital Signage Server (DSS) is responsible for broadcasting messages to different screens in the train
- The diagnostics client (TFT) is responsible for collecting diagnostics on the train
- The MongoDB database stores general information
- The Journey Information Manager (JIM) is the journey message listener

The wayside part (wayside is a term used extensively in railway systems to denote any system that is not located on the trains e.g. trackside systems, cloud backend infrastructure, etc.) consist of the following items:

- The isync UI is the user interface for the isync service

Figure 4.5 The railway communication application, provided by the industry partner in the DynAMo research project, to conduct the detailed performance evaluations.



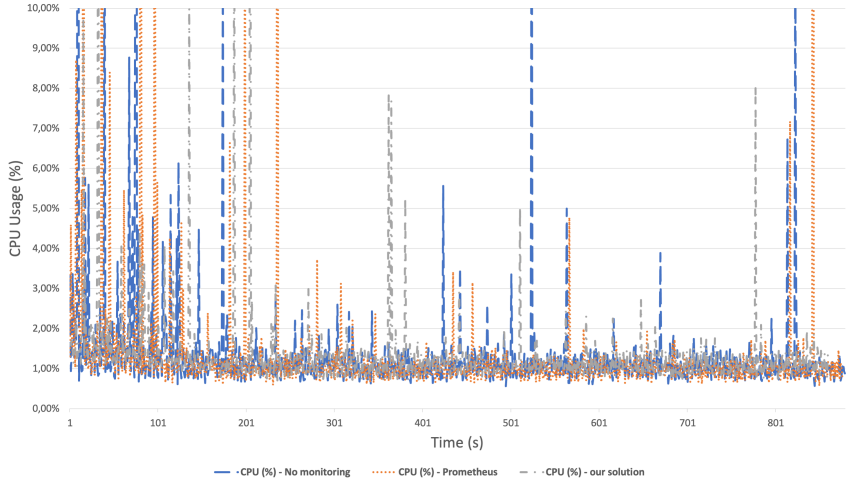
- The isync server is the management server for the isync services on the wayside and on the trains
- The MongoDB database stores general information on the wayside
- The xdata service is the message broker

All these services act like black boxes and are running as individual containers on a server with 2 Hexacore Intel E5645 (2.4GHz) CPUs, 24GB of RAM and Ubuntu 18.04 LTS installed. As already mentioned, the service is first monitored without monitoring active, afterwards with the Prometheus solution monitoring the Docker containers and finally with the agent and the agent management system active.

4.5.2 Obtained results

To evaluate the platform, a simulation of a train journey is executed. This is a train journey covering 6 stops from Liverpool to Crewe, with an average train speed of 20m/s. A screenshot of the simulation is provided in Figure 4.5. After each simulation, the system is restarted completely in order to avoid interference by e.g. a garbage collection system. During the simulation, the statistics of each docker container are stored every second using the *docker stats* command. Per service, the CPU usage and memory usage in percentages are plotted, without the monitoring active, with Prometheus monitoring active and with our proposed solution active.

Figure 4.6 Evaluation of the CPU usage (%) of the train isync service without any monitoring system, with Prometheus monitoring and with our proposed solution active



The train isync service has an average CPU usage of 1.33% with a standard deviation of 2.02% (afterwards mentioned as +/- 2.02%) and memory usage of 1.58% (+/- 0.03%) without any monitoring solution active. When Prometheus is monitoring the whole setup, the total average CPU usage of this service is 1.42% (+/- 3.86%) and the total average memory usage is 1.46% (+/- 0.04%). With our monitoring system, the total average CPU usage of this service is 1,31% (+/- 1.23%) and the total average memory usage is 1.40% (+/- 0.03%). The graphical results of this service are illustrated in Figure 4.6 and Figure 4.7. The statistical results for all the other services are illustrated in Table 4.2.

Taking the standard deviation of each result into account, one can see that the proposed system has a negligible impact on the simulation environment. The evaluation of the Prometheus system shows that these results are also similar to the results of our proposed system. This means that the proposed system works while portraying similar performance to Prometheus despite offering additional functionality such as remote reconfiguration, offline support and anomaly detection - all while being tailored to complex multi-vendor service integration environments. The major requirement to create an unobtrusive monitoring system is thus proven.

The agents monitoring the production services are also evaluated in terms of CPU-usage and memory usage. Figure 4.8 illustrates the CPU usage of each agent and Figure 4.9 illustrates the memory usage of each agent. In

Figure 4.7 Evaluation of the Memory usage (%) of the train isync service without any monitoring system, with Prometheus monitoring and with our proposed solution active

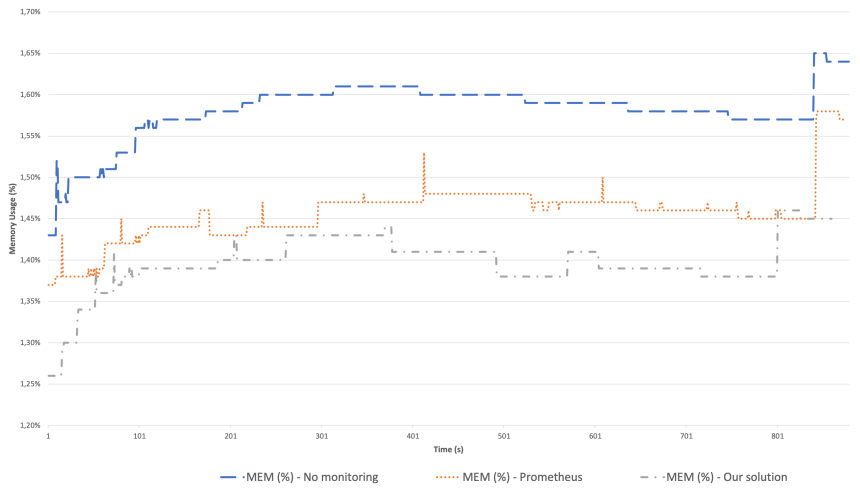
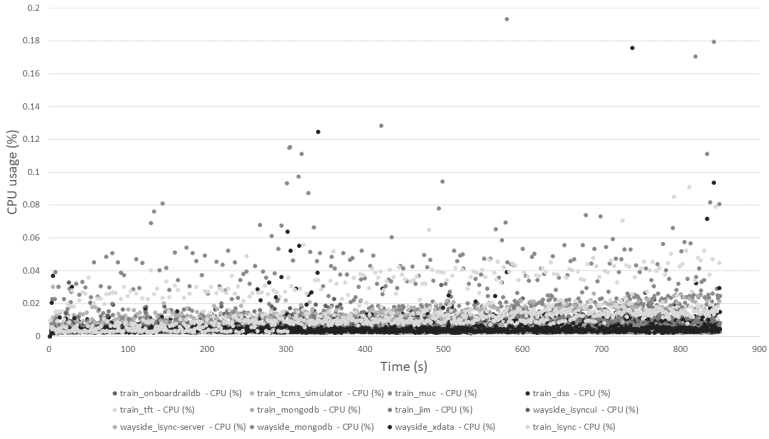


Table 4.2: Monitored services in the PoC environment. The CPU usage and memory usage of each service are captured during three train journey simulations, one without the proposed monitoring solution, one with Prometheus monitoring active and one with our proposed monitoring solution. Every number is illustrated in %. The first value is the average CPU or memory usage during the whole simulation and the values between brackets are the standard deviations.

	CPU (%) No monitoring	**CPU (%) Prometheus**	**CPU (%) Our solution**
Train dss	1.28 (+/- 1.90)	1.29 (+/- 1.28)	1.33 (+/- 1.37)
Train isync	1.33 (+/- 2.02)	1.42 (+/- 3.86)	1.31 (+/- 1.23)
Train jim	1.60 (+/- 4.41)	1.58 (+/- 5.16)	1.50 (+/- 2.31)
Train mongodb	0.20 (+/- 0.02)	0.20 (+/- 0.01)	0.22 (+/- 0.02)
Train muc	1.03 (+/- 0.18)	1.09 (+/- 0.25)	1.21 (+/- 0.26)
Train onboardraildb	1.84 (+/- 6.86)	1.41 (+/- 4.21)	2.07 (+/- 6.38)
Train tcms-simulator	5.36 (+/- 0.51)	5.37 (+/- 0.51)	5.57 (+/- 0.50)
Train tft	0.86 (+/- 2.00)	0.76 (+/- 1.90)	0.84 (+/- 1.97)
Wayside isync-server	0.86 (+/- 0.91)	1.00 (+/- 2.98)	0.98 (+/- 1.73)
Wayside isync-ui	0.22 (+/- 2.07)	0.16 (+/- 1.06)	0.13 (+/- 0.31)
Wayside monogdb	0.20 (+/- 0.02)	0.20 (+/- 0.01)	0.22 (+/- 0.02)
Wayside xdata	0.18 (+/- 0.16)	0.25 (+/- 0.72)	0.28 (+/- 0.41)
	MEM (%) No monitoring	**MEM (%) Prometheus**	**MEM (%) Our solution**
Train dss	4.35 (+/- 0.67)	4.67 (+/- 0.72)	6.11 (+/- 1.15)
Train isync	1.58 (+/- 0.03)	1.46 (+/- 0.04)	1.40 (+/- 0.03)
Train jim	1.06 (+/- 0.01)	1.02 (+/- 0.01)	1.04 (+/- 0.01)
Train mongodb	0.14 (+/- 0.00)	0.14 (+/- 0.00)	0.14 (+/- 0.00)
Train muc	0.04 (+/- 0.00)	0.04 (+/- 0.00)	0.04 (+/- 0.00)
Train onboardraildb	3.40 (+/- 0.12)	3.79 (+/- 0.03)	3.92 (+/- 0.03)
Train tcms-simulator	0.41 (+/- 0.00)	0.37 (+/- 0.00)	0.37 (+/- 0.00)
Train tft	1.27 (+/- 0.04)	1.16 (+/- 0.04)	1.07 (+/- 0.04)
Wayside isync-server	1.41 (+/- 0.05)	1.36 (+/- 0.05)	1.68 (+/- 0.15)
Wayside isync-ui	8.56 (+/- 0.02)	10.57 (+/- 0.01)	10.45 (+/- 0.02)
Wayside monogdb	0.21 (+/- 0.00)	0.14 (+/- 0.00)	0.13 (+/- 0.00)
Wayside xdata	2.45 (+/- 0.00)	3.56 (+/- 0.00)	2.96 (+/- 0.00)

Figure 4.8 CPU usage of deployed agents monitoring the different services in the train passenger information system simulator. The CPU usage of every agent remains low with some with average CPU usage at 0.01% and outliers up to 0.19%



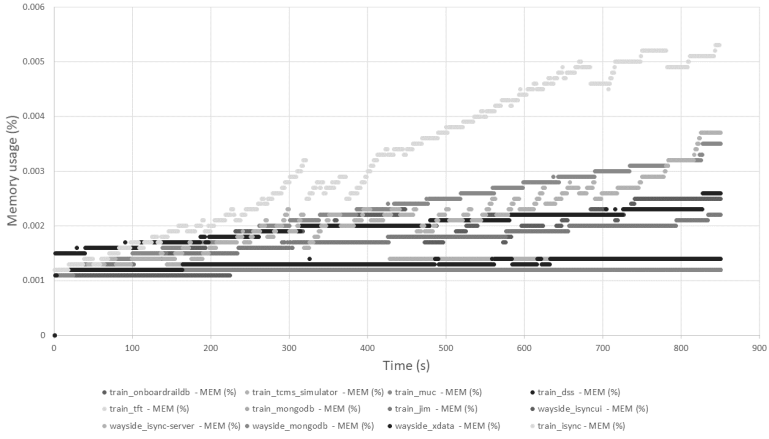
general, the CPU and memory usage of each agent remains low, supporting our conclusion that our monitoring system has a negligible impact. On the other hand, the CPU and memory usage is getting larger in time, due to the garbage storage in each agent. When the system limits are achieved, this is resolved by the automatic garbage collection in .NET. The CPU and memory usage of each agent itself can also be limited with the corresponding docker commands when starting the proposed system, ensuring the unobtrusive requirement.

4.6 Conclusion

Monitoring services in complex cloud-enabled production environments, hosting services from different vendors or developers, is mostly complex and error-prone. Therefore, this chapter proposed the design and inception of a monitoring system that does not have significant impact on the production environment. The required components are discussed together with the management system responsible for orchestrating different agents in such a production environment. Every agent is unobtrusive, remotely configurable and can handle network interruptions and time drifts.

To evaluate the impact of our proposed platform, a PoC simulation environment is deployed and evaluated in terms of CPU usage and memory usage without any monitoring system, with Prometheus monitoring and with our

Figure 4.9 Memory usage of deployed agents monitoring the different services in the train passenger information system simulator. The memory usage is increasing over time due to the garbage storage in the agent, but is resolved automatically by means of a garbage collection process in the proposed PoC. Hence, the conclusion can be made that the memory usage remains low.



proposed solution active. It was clear that our approach has a negligible impact on the production environment, works with similar performance as Prometheus while offering additional functionality focused on multi-vendor service integration, and that the required resources to deploy our agents are small (an average of 0.02 % in terms of CPU usage).
The evaluation of the monitoring metrics in a (simulation) production environment is foreseen as future work. Also, further processing of the monitoring metrics stored by the agent orchestrator is planned to allow coupling with error and anomaly detection, error prediction and AI solutions. All this will eventually lead to the creation of a digital twin environment that can provide a duplicate (including stubs imitating external service interaction behavior) of software services running in the production environment. This digital twin should allow controlled study of software interactions causing errors and the impact of proposed solutions or patches to remediate these faults.

Acknowledgement

Vlaio (Flemish Agency for Innovation and Entrepreneurship) in the context of the imec ICON project DynAMo (https://www.imec-int.com/en/what-we-offer/research-portfolio/dynamo)

References

[1] Televic rail: Passenger information systems & condition based monitoring | televic rail. https://www.televic-rail.com/en. (Accessed on 26 Jan. 2021).

[2] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Fog computing: Enabling the management and orchestration of smart city applications in 5g networks. *Entropy*, 20(1):4, 2018.

[3] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. Resource provisioning for iot application services in smart cities. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2017.

[4] Ameni Kallel, Molka Rekik, and Mahdi Khemakhem. Iot-fog-cloud based architecture for smart systems: Prototypes of autism and covid-19 monitoring systems. *Software: Practice and Experience*, 51(1): 91–116, 2021. doi: https://doi.org/10.1002/spe.2924. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2924.

[5] Mohit Taneja, Nikita Jalodia, John Byabazaire, Alan Davy, and Cristian Olariu. Smartherd management: A microservices-based fog computing–assisted iot platform towards data-driven smart dairy farming. *Software: Practice and Experience*, 49(7):1055–1078, 2019. doi: https://doi.org/10.1002/spe.2704. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2704.

[6] Prometheus - monitoring system & time series database. https://prometheus.io/. (Accessed on 26 Jan. 2021).

[7] Telegraf open source server agent | influxdata. https://www.influxdata.com/time-series-platform/telegraf/. (Accessed on 26 Jan. 2021).

[8] Rigor | digital experience monitoring and optimization. https://rigor.com/. (Accessed on 26 Jan. 2021).

[9] Apimetrics software api testing monitoring compliance performance and security. https://apimetrics.io/. (Accessed on 16 Jan. 2021).

[10] Open source api gateway | kong microservices api gateway - konghq. https://konghq.com/kong/. (Accessed on 26 Jan. 2021).

[11] Traefik, the cloud native application proxy | traefik labs. https://traefik.io/traefik/. (Accessed 26 Jan. 2021).

[12] F. Pina, J. Correia, R. Filipe, F. Araujo, and J. Cardroom. Nonintrusive monitoring of microservice-based systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018. doi: 10.1109/NCA.2018.8548311.

[13] Roman Vaculin and Katia Sycara. Semantic web services monitoring: An owl-s based approach. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, pages 313–313. IEEE, 2008.

[14] Jose M Alcaraz Calero and Juan Gutierrez Aguado. Monpaas: an adaptive monitoring platform as a service for cloud computing infrastructures and services. *IEEE Transactions on Services Computing*, 8(1):65–78, 2014.

[15] A. Noor, D. N. Jha, K. Mitra, P. P. Jayaraman, A. Souza, R. Ranjan, and S. Dustdar. A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 156–163, 2019. doi: 10.1109/CLOUD.2019.00035.

[16] K. Alhamazani, R. Ranjan, P. Prakash Jayaraman, K. Mitra, C. Liu, F. Rabhi, D. Georgakopoulos, and L. Wang. Cross-layer multi-cloud real-time application qos monitoring and benchmarking as-a-service framework. *IEEE Transactions on Cloud Computing*, 7(1):48–61, 2019. doi: 10.1109/TCC.2015.2441715.

[17] A. Souza, N. Cacho, A. Noor, P. P. Jayaraman, A. Romanovsky, and R. Ranjan. Osmotic monitoring of microservices between the edge and cloud. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 758–765, 2018. doi: 10.1109/HPCC/SmartCity/DSS.2018.00129.

[18] M. Cinque, R. Della Corte, and A. Pecchia. Microservices monitoring with event logs and black box execution tracing. *IEEE Transactions on Services Computing*, pages 1–1, 2019. doi: 10.1109/TSC.2019.2940009.

[19] S. Ma, I. Liu, C. Chen, J. Lin, and N. Hsueh. Version-based microservice analysis, monitoring, and visualization. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 165–172, 2019. doi: 10.1109/APSEC48747.2019.00031.

[20] Nagios - network, server and log monitoring software. https://www.nagios.com/. (Accessed on 26 Jan. 2021).

[21] Nagios xi - easy network, server monitoring and alerting. https://www.nagios.com/products/nagios-xi/. (Accessed on 16 Jan. 2021).

[22] Piotr Gawkowski. Extending icinga monitoring capabilities. *Information Systems in Management*, 5, 2016.

[23] Christophe Haen, Enrico Bonaccorsi, and Niko Neufeld. Distributed monitoring system based on icinga. *WEPMU035 ICALEPCS*, 2011.

[24] Andrea Dalle Vacche and Stefano Kewan Lee. *Zabbix Network Monitoring Essentials*. Packt Publishing, 2015. ISBN 1784399760.

[25] Michael Badger. *Zenoss core network and system monitoring*. Packt Publishing Ltd, 2008.

[26] Dimitri Petrik and Georg Herzwurm. iiot ecosystem development through boundary resources: a siemens mindsphere case study. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software-Intensive Business: Start-ups, Platforms, and Ecosystems*, pages 1–6, 2019.

[27] Thingworx: Industrial iot software | iiot platform | ptc. https://www.ptc.com/en/products/thingworx. (Accessed on 26 Jan. 2021).

[28] Yuri S Per, Maxim V Tsypliaev, Maxim V Lyadvinsky, Alexander G Tormasov, and Serguei M Beloussov. Fast incremental backup method and system, 2008. US Patent 7,366,859.

[29] Bernd Greifeneder, Helmut Spiegl, Markus Gaisbauer, and Clemens Fuchs. Method and system for tracing end-to-end transaction which accounts for content update requests, 2017. US Patent 9,571,591.

[30] Operations: Cloud monitoring & logging | google cloud. https://cloud.google.com/products/operations. (Accessed on 26 Jan. 2021).

[31] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[32] R Wysor, K Michelle Baldwin, and Bobby R Whitus. Low cost time synchronization for remote measurement systems. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2018.

[33] Steve Vinoski. Server-sent events with yaws. *IEEE internet computing*, 16(5):98–102, 2012.

[34] Dirk Strauss. Getting started with. net core 3.0. In *Exploring Advanced Features in C#*, pages 189–220. Springer, 2019.

[35] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[36] Chris Anderson. The managed extensibility framework. In *Pro Business Applications with Silverlight 5*, pages 501–520. Springer, 2012.

[37] Agent reference docs · netdata agent | learn netdata. https://learn.netdata.cloud/docs/agent. (Accessed on 26 Jan. 2021).

[38] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[39] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011.

[40] Clickhouse - fast open-source olap dbms. https://clickhouse.tech/. (Accessed on 26 Jan. 2021).

5

UAVs-as-a-Service: Cloud-based Remote Application Management for drones

*In this chapter, Research Challenge 3 (**C3**) is partly addressed. It proposes a container-based application platform for drone development, enabling cloud-to-drone development. In a first stage, it is investigated how to deploy containerized applications on drones, followed by the creation of the prototype UAVaaS platform, allowing to deploy application components either on drones from a drone fleet, or connect them to components deployed and running in the cloud. A drone also requires specific application components to be deployed at startup to enable remote drone management and piloting from an application or from a ground control station. The proposed platform enables these application components when a drone is connected to the system. Three reference implementations are deployed by means of this system to demonstrate its capabilities.*

⋆⋆⋆⋆⋆

J. Moeyersons, M. Gevaert, K. Réculé, B. Volckaert and F. De Turck

Published in "2021 IFIP/IEEE International Symposium on Integrated Network and Service Management, Proceedings"

Abstract In recent years, the Internet-of-Drones (IoD) became an important research topic for both industry and academy. An IoD environment consist of different drones, called Unmanned Aerial Vehicles (UAVs), flying in different zones whereby communication is important. Therefore, drones are becoming increasingly ambiguous, capable and more cost effective than ever before. These drones have been equipped with different sensors, making it IoT-enabled drones, capable of capturing multiple data sources and send them to the cloud for further research, but the continuous advance in drone technology has not necessarily made drone application development easier. While mature Infrastructure-as-a-Service (IaaS) platforms offer features such as hardware abstraction, resource allocation and tools to manage applications remotely, commercial drones often offer a restrictive software environment instead. Inspired by the technical success and convenience of IaaS platforms, this chapter sets out to bring that experience to drones, resulting in the creation of the UG-One UAVaaS platform. Many of the technologies used in the UAVaaS platform can be found in the world of Cloud computing as well. Applications created for drones are containerized using Docker and application management can be done through a web interface. The drones host a REST API for platform management and they have a Linux onboard computer. Developers can deploy applications on the drones or forward the required data and deploy their applications on a remote server instead. This approach has delivered promising results when evaluated using several reference applications that either represent real world applications such as video streaming and movement control or instead just stress tests to check for resource availability and reliability. In the end, the UG-One platform is shown to succeed in simplifying drone application development and management while maintaining the reliability and versatility required from any drone platform.

5.1 Introduction

The last decade, Internet of Things (IoT), where various objects (or things) are connected through the Internet, has made a technological advancement [24]. Drones are one of these things [27], consisting of several IoT smart devices, such as LIDAR, thermal sensors, cameras, chemical sensors and many more [28]. They have become increasingly ambiguous and drone manufacturers have been able to create cost effective devices that can fly longer, carry heavier loads and have better on-board hardware. Even drone management systems have been developed to maintain a good overview of several deployed drones. [26]. The continuous advance in drone technology

certainly has improved the capabilities of drones, but it has not necessarily made drone application development easier.

When building applications that need to run in the cloud, developers create containerized applications which are remotely deployed and managed, have guarantees when it comes to resource availability and developers certainly do not need to worry about the vendor of the hardware on which their applications are deployed. This is not the case when developing an application for a drone, where a developer needs to take into account different aspects such as specific hardware constraints, power constraints and load constraints. Therefore, this chapter proposes a platform, further called the UG-One platform, unlocking the potential of these drone innovations towards a larger number of less drone-savvy developers.

By creating the UG-One UAVaaS platform based on cloud technologies, developers can now work in a familiar environment were applications are containerized, platform components communicate using REST APIs, drones run Linux and application deployment happens through a web interface.

However drones are not cloud servers, they need to make real-time decisions and run applications that often need access to on-board hardware such as cameras and actuators, or even require the ability to plot a new course for the drone. Instead of trying to create an all-in-one system that provides all the required functionality, the UG-One drone design is based on separate non-proprietary components with each component optimized for specific tasks. On a hardware level, the autopilot controls the drone while a Linux-based on-board computer runs applications. On a software level, each part of the UG-One drone system is containerized and can work independently without influencing other applications or components.

While a UG-One drone can be used on its own, deploying applications through a REST API or Swagger-UI is far from user-friendly and application management quickly becomes complex when dealing with multiple drones. For this purpose, the UG-One back-end has been created. This cloud system allows users to manage the applications running on a fleet of drones and simplifies things such as deploying applications that remotely control the drones, or even individually control drones through ground control software.

The remainder of this chapter is organised as follows: Section 5.2 covers how drones can fly reliably and the need for an on-board generic computing platform while Section 5.3 explains the use of Docker on this on-board computer. In Section 5.4, the proposed system is illustrated en explained in detail followed by some reference implementations and validation in Section 5.5. Section 5.6 explains how a drone fleet can be managed and conclusions are summarized in Section 5.7

5.2 Reliable drone flight and on-board computer

As with most aviation systems, drones need to be reliable and thoroughly tested, which can slow down development. Each time software or hardware that controls the drone is modified, all aspects needs to be evaluated again. To prevent this kind of slowdown, open-source autopilots such as Pixhawk [20] and ArduPilot [16] can be used. These autopilots are good at following precise instructions such as follow a specific path, fly to these coordinates and fly at this speed. These instructions are communicated through the open-source MAVLink protocol [18], which is utilized by the two most popular and advanced open-source autopilot software projects, PX4 [2] and ArduPilot. Since most developers do not have any experience with MAVLink, the open-source MAVSDK library has been created by the team behind Pixhawk [19]. With these thoroughly tested, maintained and documented autopilots, developers do not need the time or skills required to create an autonomous drone themselves, instead they can focus on their applications which are deployed on a separate Linux-based on-board computer. If the on-board computer fails, the autopilot simply pilots the drone back home. The on-board computer hardware and software can as such be optimized for application deployment, performance and efficiency instead of redundancy and reliability at all cost. This allows the usage of mainstream low-energy consumption embedded computers such as a Raspberry Pi. Since most of the drone specific challenges are handled by the autopilot, the on-board computer can now be interpreted as a remote deployment server. The remote connection between the drone and the ground has been implemented using a simple network connection over Wi-Fi during development, but this can be done over cellular as well.

While MAVSDK can be used to "Talk MAVLink", a communication pipeline between the autopilot and the applications that use MAVSDK is still needed. Between the autopilot and the onboard computer, this happens over Universal Asynchronous Receiver-Transmitter (UART). On the onboard computer itself, a program called MAVLink-Router [1] then forwards these messages over UDP and routes them to applications on the drone or anywhere else, such as a cloud back end or possibly even another drone. Since multiple autopilots utilize the MAVLink protocol, it should now also be possible to simply switch out one autopilot with another, or in other words, deploy the same application on multiple drones with different autopilots.

5.3 Dockerised Drone

Because the on-board computer now acts as a remote deployment server, the software running on it can be developed in a way that takes full advantage of container technologies such as Docker [17]. In fact, each platform component or application that runs on the drones is containerized. By making use of container virtualisation technology, the system becomes inherently more reliable, it has improved hardware compatibility and it is easier to deploy (e.g. new hardware requires only that docker is supported).
It is also much easier to manage the resources available to containerized applications. This is critical as without resource management, any application can utilize as much of the available resources as it wants to use, resulting in applications slowing down or even crashing.

5.3.1 Resource management

Resource management is done by a containerized component running on each UG-One drone called the UG-One API. This application, among other responsibilities, exposes a REST API which can be used to easily integrate the drones into a cloud back end. The API is designed to provide all of the Docker functionality that is needed to manage applications on a drone, but with the required limitations to guarantee resources. It also exposes endpoints that provide information on the system such as connected USB devices, available disk space and current memory usage.
While the UG-One API can guarantee a fixed amount of memory available to applications, using underlying Docker functionality, it does not guarantee a fixed percentage of CPU time. Instead, it works by allowing developers to assign a CPU priority weight. As long as the CPU is not being used 100%, any application can utilize as much of the CPU as it wants. However, as soon as applications start competing for processing power, the distribution of CPU time will be made based on the weight of each application that is making calculations at that time.

5.3.2 Unmanaged resources

While CPU and memory resources are handled by the system, the system still lacks network bandwidth prioritization and disk space limiting. Network bandwidth prioritization is not yet part of Docker and while it is possible to limit the size of a Docker container, it is not yet possible to limit the volume on the host that users can mount.
Despite these shortcomings, the UG-One API is still capable of reliably managing containers and guaranteeing two critical resources. Because of

this, it is used to manage all the components on a UG-One drone, including itself.

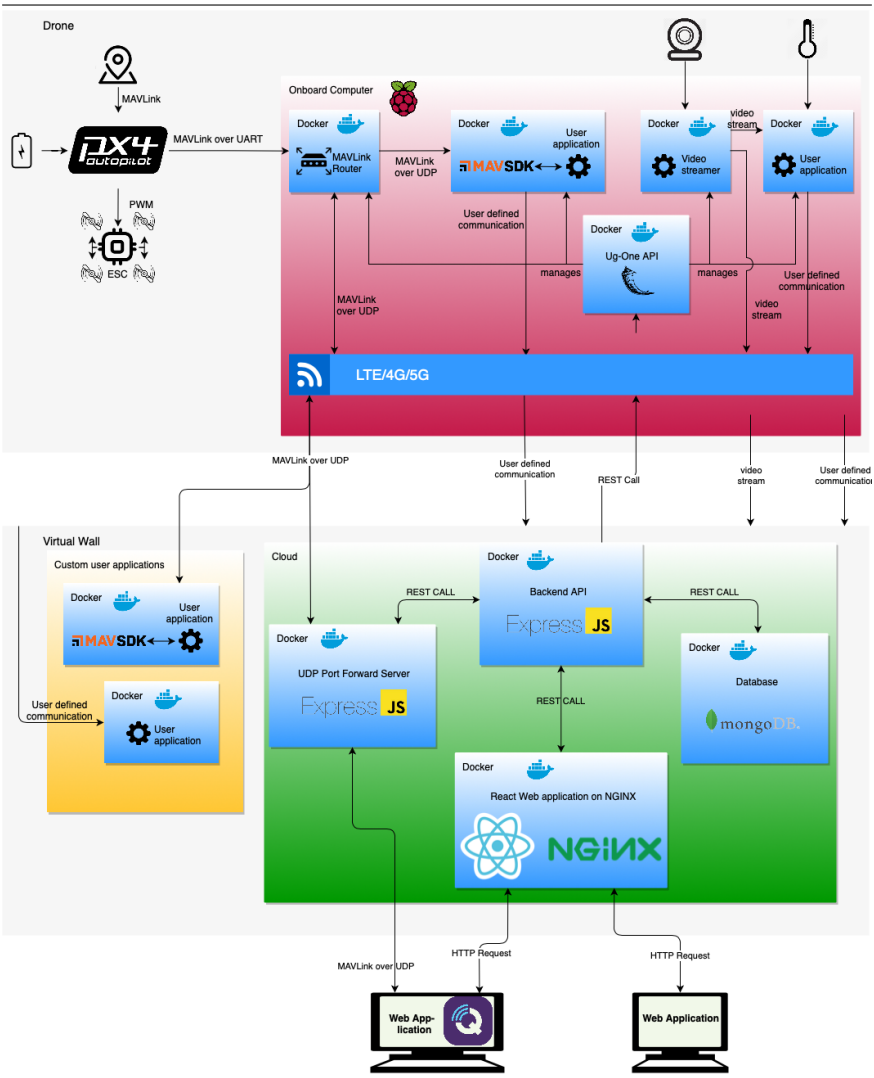
5.4 System overview and implementation details

This section covers the system overview, illustrated in Figure 5.1. The upper part covers the autopilot and the on-board computer on the drone, which was covered in Section 5.2 and Section 5.3. The other part visualizes the UG-One back-end and is explained below.

5.4.1 Back-end API

To ensure that the front-end can obtain dynamic content from the database, the API servers and from the drones, there is a need for a central unit that will retrieve this information. This unit also needs to ensure that the drones can extract data from the database and make it possible that drones can retrieve their configuration. This central unit is the Back-end API which is created by a Node.js [40] server and the npm module Express [4].
To make a connection with the database, drones and API servers, several node modules are used. The first node module is Axios [5] which is a promise-based HTTP client npm module for the browser and Node.js.
Through Axios, it is possible to retrieve information from the drones, manage the applications and request resources by using their UG-One API. Via the node module docker-hub-api [6], information of Docker Hub images from a public registry can be retrieved. To retrieve information from a Docker Image on GitLab, an attempt was first made to access it via a node module. These node modules were node-gitlab [9] and gitlab [7]. Due to lack of documentation and the absence of features which can retrieve Docker images, the GitLab API is chosen. With the GitLab API server [3], the different Docker images as their meta-information can be retrieved. The Mongoose node module [8] is used to access the MongoDB [23] database in the cloud to manage the data and modify the database structure.
For the development of a cloud platform, a data store is not to be missed out. For the cloud infrastructure, only one database was chosen for simplicity, but multiple types of databases can be used. Before a database can be deployed, research was done to determine if a SQL or a NoSQL data store is best suited for the cloud platform [37]. By weighing up the advantages and disadvantages, a choice was made to choose for a NoSQL database. NoSQL databases are easier to scale horizontally than SQL databases [31], because a NoSQL database ensures that if the data that needs to be stored grows,

Figure 5.1 UG-One system components overview



the database can easily be expanded. Also, there is no need to create a predefined schema and NoSQL databases are more suited to handle big data then SQL databases [37]. This is important because different applications on the drones, can store many different types of data in different structures. By comparing the different NoSQL databases and taking the CAP theorem [29] into account, the MongoDB database has been chosen. The CAP theorem states that it is difficult for a distributed datastore to simultaneously provide more than two out of the following elements: consistency, availability and partition tolerance. For the cloud infrastructure of the UG-One platform, consistency and partition tolerance are the two most important elements. Therefore, MongoDB was chosen because it offers consistency (all nodes see the same data at the same time) and partition tolerance (the system must work continuously without the loss of messages or partition failure).

5.4.2 UDP Port Forward Server

When the drone is configured and starts up, it is able to send MAVLink messages through the UDP Protocol to an application on a client's device such as a ground control station. A ground control station is a typical software application that runs on a computer on the ground. Through wireless telemetry, the computer on the ground communicates with the UAV. A ground control station can display real-time data about the performance, position, altitude, etc. of the UAVs. It can also be used to control an in-flight UAV, upload new mission commands and set parameters like altitude [21]. When the drone is ready to fly, one possibility is to send MAVLink messages directly to the client's device. However this has a couple of disadvantages. When the drone boots up, an IP address must be given to the drone to send MAVLink messages to. This IP address cannot be changed when the drone is active so when the IP address of the user changes, the connection with the drone will be lost. Also, when another user wants to take over the drone, this architecture prevents the user from controlling the drone. Another drawback is when multiple drones are sending their MAVLink messages to the same endpoint of an application or on the back-end. One can choose to try to distinguish the MAVLink messages from the different drones in that endpoint, but this can lead to complex routing systems within the endpoint. To solve these problems of the direct drone-client connection architecture, a protocol between the drone and the Back-end API has been set up and an UDP Port forward server was developed.
To make sure that not all drones send their MAVLink messages to the same port on the back-end, a protocol between the drone and the back-end is created. At drone startup time, the IP address of the back-end is given,

which always has the same IP address. However, the drone needs to be given a specific port to which it can send its data. Not all drones can send their data to the same port because then the back-end would have issues in knowing which data stems from which drone. The solution used in this chapter is to tie each drone to one specific port on the back-end. When a drone boots up, one of the first actions it performs is to send a request to the Back-end API to negotiate for a MavlinkPort. When the drone is registered to the platform, it receives a MavlinkPort. This Mavlinkport is the port on the UDP Port Forward Server to which the drone needs to send his MAVLink messages to. This MavlinkPort makes sure that only that specific drone is allowed to send to that specific port. This solves the problem that different drones would send their MAVLink messages to the same endpoint.

To solve the issues when a client IP address changes or another client wants to take over the drone, an UDP Port Forward Server is created. As the name implies, this server will forward UDP messages to another port on a different IP address. To configure the forwarding of the UDP messages, a REST request must be sent to the UDP Port Forward Server. This REST request will dynamically open or close the ports and configure the forwarding mechanism. Through this UDP Port Forward Server, multiple drones can send their MAVLink messages to the server and it forwards them to the right device of the client. Through the web application, the user can make sure that these MAVLink messages arrive on the application on the client's device or on an application on the back-end. The web application sends a request to the back-end API with the configuration data. The back-end API will in turn extract additional data from the database and send it to the UDP Port Forward Server. Using this configuration data, the UDP Port Forward Server can open the MavlinkPort as well as a ClientPort on the UDP Port Forward Server. The ClientPort is a port that is used to send data from the client back to the MavlinkPort on the drone's IP Address. This way, multiple drones can connect to the cloud and users can monitor all their drones with a ground control station.

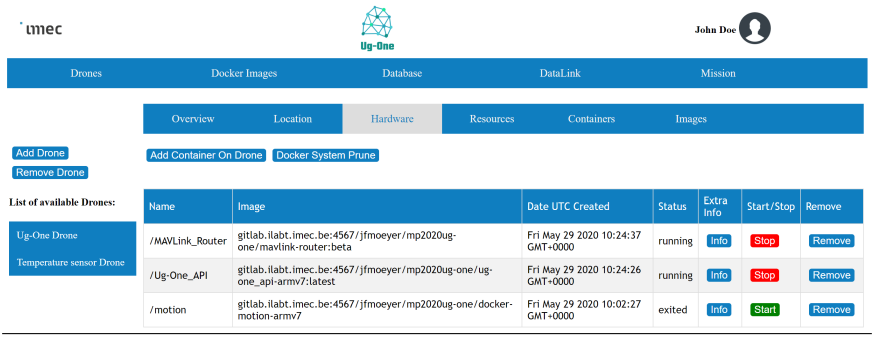
When the IP address of the user changes or when another user wants to monitor or control the drone, this is no problem anymore. By reconfiguring the ports and IP addresses of the UDP Port Forward Server, a user can reconnect with their drone or another user can take control of a drone. We investigated how such an UDP Port Forward Server could be properly incepted. The first solution was to use a Router of which the Routing- and IPTables could be modified [32, 36]. This solution had one drawback, the router had to be rebooted every time the UDP Port Forward Server would set up new forwarding connections. The solution that therefore was used

here is an API Server. On this server, UDP sockets (ports) can dynamically be opened and closed. Also, incoming requests with configuration details can be handled and ports can easily be set up. For simplicity, Express.js was used to set up an API Server. To forward the UDP Messages, several possibilities were evaluated. Eventually the node module dgram [22] was chosen to forward UDP messages. It has extensive documentation and can be configured easily.

5.4.3 Front-end

To ensure that users can easily manage their drones and their applications, a web application is created as illustrated in Figure 5.2. Through the web application, different Docker applications can be deployed on the drone. It is also possible to retrieve information, stop, restart and delete these applications. Through the web application, Docker images, which can be stored on Docker Hub or GitLab, can be linked to our solution. Those linked images can then be used for deployment of the Docker applications on the drone. To create this web-based front-end, several front-end frameworks were studied and the choice was made to use the framework React. React [30] is an UI library that uses a component-based architecture just like Angular. React is characterized by several features such as one-way-databinding [13], JSX [10] and the Virtual DOM [15]. Unlike the Angular framework, React does not have an app-level state. All the states are stored in all the different components of the web application. When a component wants to use information from another component, different components must pass their state to each other so that these components in their turn will pass their state to other components and so on. For larger projects, this can lead to errors and difficulties in managing the application. This problem can be solved by using Redux [14, 25] or Context API [12]. Redux and Context API [38] add structures and components to manage the app-level state. They also offer the possibility to work more conveniently by using this one central unit to store the state. When creating the web application, a combination of Redux and Context API was used [33, 34]. The Context API methods are used to recreate the structure and components of Redux. Redux itself is not easy to understand and to use. The combination of both app-level state managers makes it easier and faster for a developer to create a web application with an app-level state. But when creating a bigger more complex web application, Redux is recommended because it can manage the data better and clearer and offers tools to help manage the app-level sate.
For the deployment of the web server, NGINX was chosen [11]. This web server has a low memory load, is lightweight and offers the basic

Figure 5.2 The UG-One front-end showing the active containers on a specific drone



functionalities to host a web server.

5.5 Reference implementations and validation

In order to validate and demonstrate some of the capabilities of the UG-One platform, several component reference implementations have been created. These have the added benefit of demonstrating how certain applications can be implemented by developers that want to start developing for the UG-One platform. The source code is publicly available through https://github.com/IBCNServices/UG-One.

5.5.1 Video streaming

First of, a reference implementation [39] is created that connects to an on-board drone camera and streams video to both on-board and off-board applications. While this implementation can still be optimized significantly, it was already capable of providing a high-resolution stream to on-board and low-resolution stream to off-board applications, reducing bandwidth and battery usage.

5.5.2 Stress testing

The second application was created to simply run a stress test and use as much resources as possible. This application was used to validate the resource management capabilities of the UG-One drones. As expected, these applications could never exceed their memory limit and could only claim CPU time in relation to their weight. This means that no critical container

with a significantly higher CPU share value, or weight, ever experienced any real slowdowns while a stress test was running.

5.5.3 MAVLink application

Finally, as mentioned before, one of the reference applications was a container that reads out telemetry data provided by the autopilot over MAVLink.

5.6 Managing a drone fleet

Users can deploy and manage their applications, request system resources and more through the REST API on the drone. But this way of working is not user-friendly and certainly not when multiple applications need to be managed on multiple drones. To be able to manage those applications in a fast and easy way, a back-end cloud infrastructure is needed. The proposed cloud infrastructure that was created offers a web application that enables users to easily deploy applications, keep an overview of their drone fleet, manage resources on the drones, etc. Because the web interface shows dynamic content to the users, there is also a need for a back-end API and a database. This back-end API is the central unit of the cloud infrastructure which sends and receives requests to and from the web server, database, drones, API servers, etc. Next to the web server, back-end API and database, there is one last component present on the cloud, the UDP Port Forward Server.

5.7 Conclusion

In line with the established cloud service models, a new UAVaaS model and platform has been created, allowing developers with no previous drone experience to develop and deploy applications on drones. Container technologies are used to offer the convenience of application and drone management through a web interface while it also continues to be modular and very capable as well. Applications can run on any drone that has the required hardware components and often can run on the back-end as well with little to no modification.

As resources on the drone on-board computing unit are considered scarce, in future work we will investigate the use of more secure container technology alike Kata containers [35] and on using uni-kernels, stripping out functionality not required by the applications running on the drone.

References

[1] Mavlink router. URL https://github.com/intel/mavlink-router.

[2] PX4 documentation v1.10.1, 2019. URL https://docs.px4.io/v1.10/en.

[3] Projects api, 2020. URL https://docs.gitlab.com/ee/api/projects.html.

[4] Express, 2020. URL https://expressjs.com/.

[5] axios, 2020. URL https://www.npmjs.com/package/axios.

[6] Docker-hub-api, 2020. URL https://www.npmjs.com/package/docker-hub-api.

[7] node module gitlab, 2020. URL https://www.npmjs.com/package/gitlab.

[8] npm mongoose, 2020. URL https://www.npmjs.com/package/mongoose.

[9] node-gitlab, 2020. URL https://www.npmjs.com/package/node-gitlab.

[10] What is jsx, 2020. URL https://www.reactenlightenment.com/.

[11] Nginx web server, 2020. URL https://www.nginx.com/.

[12] Context in react, 2020. URL https://reactjs.org/docs/context.html.

[13] Data binding, 2020. URL https://docs.angularjs.org/guide/databinding.

[14] Redux in react, 2020. URL https://react-redux.js.org/introduction/why-use-react-redux.

[15] Understanding the virtual dom, 2020. URL https://bitsofco.de/understanding-the-virtual-dom/.

[16] Ardupilot, 2020. URL https://ardupilot.org/.

[17] 2020. URL https://www.docker.com/.

[18] Mavlink developer guide, 2020. URL https://mavlink.io/en/.

[19] Mavsdk documentation, 2020. URL https://mavsdk.mavlink.io/develop/en/.

[20] Pixhawk, 2020. URL https://pixhawk.org/.

[21] Uav ground control station, 2020. URL https://www.unmannedsystemstechnology.com/category/supplier-directory/ground-control-systems/ground-control-stations-gcs/.

[22] Udp/datagram sockets, 2020. URL https://nodejs.org/api/dgram.html.

[23] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*, pages 14–22, 2013.

[24] Basudeb Bera, Durbadal Chattaraj, and Ashok Kumar Das. Designing secure blockchain-based access control scheme in iot-enabled internet of drones deployment. *Computer Communications*, 153:229–249, 2020.

[25] Matthias Kevin Caspers. React and redux. *Rich Internet Applications w/HTML and Javascript*, page 11, 2017.

[26] Sung-Chan Choi, Nak-Myung Sung, Jong-Hong Park, Il-Yeop Ahn, and Jaeho Kim. Enabling drone as a service: Onem2m-based uav/drone management system. In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 18–20. IEEE, 2017.

[27] Gaurav Choudhary, Vishal Sharma, and Ilsun You. Sustainable and secure trajectories for the military internet of drones (iod) through an efficient medium access control (mac) protocol. *Computers & Electrical Engineering*, 74:59–73, 2019.

[28] Mahdi Ben Ghorbel, David Rodriguez-Duarte, Hakim Ghazzai, Md Jahangir Hossain, and Hamid Menouar. Energy efficient data collection for wireless sensors using drones. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–5. IEEE, 2018.

[29] Robert Greiner. Cap theorem: Explained, 2014. URL https://robertgreiner.com/cap-theorem-explained/.

[30] Madhuri A Jadhav, Balkrishna R Sawant, and Anushree Deshmukh. Single page application using angularjs. *International Journal of Computer Science and Information Technologies*, 6(3):2876–2879, 2015.

[31] Abhinav Korpal. Scaling horizontally and vertically for databases, 2019. URL https://bit.ly/2Bas3sJ.

[32] Eric Ma. Port forwarding using iptables, 2019. URL https://www.systutorials.com/port-forwarding-using-iptables/.

[33] Traversy Media. Node.js & express api | expense tracker, 2020. URL https://youtu.be/KyWaXA_NvT0.

[34] Pubudu Ranasinghe. Build a redux-like store with react context and hooks, 2019. URL https://bit.ly/36pGxAJ.

[35] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214. IEEE, 2019.

[36] Prithviraj S. Iptables tutorial, 2019. URL https://www.hostinger.com/tutorials/iptables-tutorial.

[37] Vatika Sharma and Meenu Dave. Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8), 2012.

[38] Ayushman Bilas Thakur. Redux vs context api, 2020. URL https://dev.to/ayushmanbthakur/redux-vs-context-api-3182.

[39] Nils Tijtgat, Wiebe Van Ranst, Toon Goedeme, Bruno Volckaert, and Filip De Turck. Embedded real-time object detection for a uav warning system. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 2110–2118, 2017.

[40] Stefan Tilkov and Steve Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6): 80–83, 2010.

6

Towards a cloud-based drone application management platform in emergency situations

*This chapter is an extension of the work explained in the previous chapter, along with a case study that used the insights of Chapter 2 and Chapter 3 to create a full drone-aware cloud platform. A lightweight Kubernetes based approach, K3S, is used as container orchestrator and an extra application layer allows developers to choose where to deploy a new containerized application: either on the drone or in the cloud. With the creation of this platform, on top of a K3S cluster with integrated drones, Research Challenge 3 (**C3**) and Hypothesis 3 (**RH3**) are confirmed. Moreover, the use case evaluation in this chapter proves that a drone can receive a high priority network traffic slice, which also confirms Research Challenge 1 (**C1**) and Research Challenge 2 (**C2**) resulting in the creation of the new UAVaaS platform.*

⋆⋆⋆⋆⋆

J. Moeyersons, M. De Schutter, F. De Turck and B. Volckaert

Submitted for review, 2022

Abstract Drones, also known as Unmanned Aerial Vehicles (UAVs), have been on the rise for the last decades. As drones are aerial objects, they are compact in size and shape, enforcing restrictions on the resources available to the drone. The limited resources on drones imply that careful thought must be put in place on how applications comply with these limitations.
In the regular software industry, an evolution to cloud usage accompanied by arising virtualization techniques such as containerization and container orchestration has transformed the industry. Inspired by this evolution, this chapter presents an application management platform for drones that enables offloading resource usage to the cloud. Via a proposed communication system, applications can be deployed to either a cloud or a drone without modifying the application configuration. K3S, a lightweight Kubernetes container orchestrator, bundles the drones and the cloud in a single cluster while NFV network slicing concepts are used to maintain the Quality-of-Service and allows the use of high priority slices.
Evaluating the presented platform through three different demo applications and an emergency use case scenario demonstrates the scalability, ease-of-use, low-resource usage, solidness and the network slicing concepts of the proposed solution. It is further shown that deploying or rearranging applications via the platform takes a matter of seconds and the overhead by K3S on the drone is limited to 120 MiB of memory and 4% of a CPU core.
In the end, the created platform succeeds in managing containerized drone applications with the possibility to offload tasks to a cloud, all while giving an overview of the cluster and its resource usage.

6.1 Introduction

Today, drones and the benefits that come with them can no longer be ignored. Drones can be used for wildlife surveys, fire mapping, forest health monitoring, and to monitor destructive activities, such as poaching and illegal logging [1]. In more popular culture, drones appear in many applications such as aerial photography, express shipping and delivery, thermal sensing for search and rescue operations, disaster management, and many more [2][3]. As drones are aerial objects, they are compact in size and shape, enforcing restrictions on the resources available to the drone. On top of that, a limited battery life with according long loading times also fortify the issues that arise when developing for drones. Using the cloud to offload certain resource-intensive tasks can prove beneficial.
In the industry, with the arrival of cloud computing, a trend is perceived towards virtualization. What started as simply using extra servers to strengthen and expand the available resources quickly turned into virtualizing

monolithic applications in a Virtual Machine (VM) on cloud servers. More recently, a container-based approach with microservices has proven beneficial in terms of scalability, platform portability, and easy, fast, and efficient application deployment. Container-based deployment systems are used in many large companies such as Netflix [4], Spotify [5], Google [6], and even The New York Times [7]. Also in drone development, a container-based deployment system has his benefits in terms of developer friendly (a developer only needs to implement the program once for multiple platforms) and homogeneous development on the cloud and on the drone (the edge). Container orchestrators are often used to manage the abundance of containers that can act together on different computational machines, so-called nodes, resulting in a cluster. Kubernetes [8] is currently the de-facto container orchestrator with several features such as orchestrating containers in both the cloud and on the edge, load balancing, resource allocation taking into account the available resources on every node, especially drones and application updates.

Inspired by these software industry trends, this approach can be translated to drone deployment, and that is where a container-based deployment system for drones comes into play. This chapter aims to design and implement a container-based deployment system that allows for container orchestration and aids in scheduling applications to the cloud or drones whilst keeping in mind that resources on the drone are rather limited. Kubernetes however, is not developed for low-resource devices. Alternatives exist, such as K3S [9], K0S [10] and KubeEdge [11], which focus specifically on low-resource devices. After investigating the possibilities, K3S turns out to have the largest community, along with being the lightweight version of Kubernetes and having a compatible API with the default Kubernetes. Therefore, all developers familiar with Kubernetes are able to use K3S without much effort [12]. Because of its lightweight character, K3S makes an excellent candidate as a container manager for drones besides the cloud. The architecture is designed with the concept of workers and servers where a server node is the manager of the cluster, deployed in the cloud, and worker nodes can then join the cluster. When drones join, it makes the drones accessible via the cluster, while if cloud servers join the cluster as worker nodes, they provide extra computational resources for the tasks that are offloaded to the cloud. In order to use a drone as a Kubernetes worker node, the drone hardware must support container technologies. This is already proven in one of our previous works [13].

In this chapter, a use case where drones are used for handling emergency situations, is introduced. It therefore requires high priority network traffic which can be fulfilled by using Network Function Virtualization (NFV)

network slicing concepts in 5G or Software-Defined Networking (SDN)-networks, allowing the use of e.g. high priority network slicing and the Quality-of-Service guarantee. The prioritization of the network traffic for this use case is already described in our previous research [14], but this chapter will combine it with the container-orchestration approach explained in this chapter.
This chapter starts with an overview of the current state-of-the-art in Section 6.2 and continues with an overview of the developed platform in Section 6.3. The emergency use case is explained in Section 6.4 followed by the evaluation of the platform in terms of ease of use, resource overhead, scalability, resource overhead and network prioritization in Section 6.5. Finally in Section 6.6, conclusions and avenues for future research are summarized.

6.2 Related work

Deployment systems for drones are a specialized topic for research. In this section, relevant research is discussed concerning the usage of containers for drones and the combination of drones with cloud back-ends.
Mehrooz et al. [15] describe an open-source framework for Internet-of-Drones (IoD) applications. Either on a drone or a simulated drone, ROS [16] is used as a meta-operating system. A Kubernetes-powered cloud provides the necessary cloud services. This setup makes it possible to have a scalable cloud solution to take care of the resource-intensive tasks. Communication is done via a REST API. In combination with ROS, these technologies provide an excellent stack for drone development. Whereas Mehrooz et al. describe a solid open-source framework for drone development, the framework is still composed of the typical development in ROS on the one side and development in the cloud on Kubernetes on the other side. In this chapter, a more unified platform is developed where the scalability of drones and their applications come into play. Another remark is the lack of objective measurements. Experiments to stipulate the trade-off between upload and download bandwidth usage and resource usage on the drone are out of the scope of the work by Mehrooz et al. These items are therefore tackled in this chapter.
A drone-as-a-service solution is presented by Hof et al. [17], making a system where drones are accessible in the cloud. The idea is that drones are often used for a single task, and via AnDrone, the drones are made accessible in the cloud. This allows third parties to run their applications on a drone that is flying without interfering with its flight path. Since the third-party users don't need to obtain hardware for their tasks but can reuse the hardware

that is used for another task, the authors refer to this concept as using a virtual drone. Multiple third-party users may run their virtual drone with its application on the hardware of a single physical drone. The authors implemented an AnDrone prototype that resulted in a low overhead of 1.5% for a single virtual drone and the possibility of running multiple virtual drones where the performance scales linearly with the workload. Moreover, these virtual drones can run without compromising the stability and safety of the drone. The concept proposed by Hof et al. is a source of inspiration for this chapter. Just like in this chapter, the idea of having a platform to deploy applications easily is combined with the usage of containers and container orchestration to virtualize the applications. In AnDrone, the Linux virtualization containers via Android Things are used. However, Google will stop supporting new Android Things projects for non-commercial use as of January 2022 [18]. The use of other container orchestrators will be discussed in this chapter, possibly resulting in an overhead that is greater than 1.5%. On another note, AnDrone uses the cloud to manage the virtual drones but does not deploy applications to the cloud, an aspect that will be a fundamental part of the research in this chapter.

Whilst not being related to drones specifically but to low-resource edge devices in general, the paper by Goethals et al. [19] proposes FLEDGE, a Kubernetes compatible container orchestrator for low-resource edge devices. In this chapter, different ways to achieve low-resource container orchestration are examined. Finally, the proposed FLEDGE system is evaluated and compared with other container orchestrators such as Kubernetes and K3S. Although the resource usage is significantly lower, FLEDGE is still highly experimental, and further improvements can be investigated. Because of this, FLEDGE will not be used as a container orchestrator in this chapter.

In another related work by Koubâa et al. [20], a cloud-based management system is proposed for the Internet-of-Drones. The system called Dronemap Planner provides communication between the cloud, users and drones through the Mavlink protocol. In their paper, they describe two main benefits of integrating drones with the cloud. A first one is virtualization since the cloud virtualizes access to drone resources through abstract interfaces. A second benefit is computational offloading. The drone can provide data to the cloud via its sensors and then act based on the results received from the cloud. Koubâa et al. focus mainly on the first benefit, virtualization. The research of this chapter specifically tackles the computation offloading aspect as well. In that way, this chapter can be seen as an addition to their research.

Finally, a last cloud-based system architecture for drones was proposed by Hong et al. [21]. Whereas the paper described above by Koubâa et al.

only supports one drone, this system provides multi-UAV support. This is a significant improvement since one can think of many applications where multiple drones are necessary and might have to cooperate. In this chapter, the idea of having multiple drones that need to be managed is taken into account as well. A significant difference between the last two papers and this chapter is the choice for container technology and container orchestrators to build the cloud-based deployment system for drones.

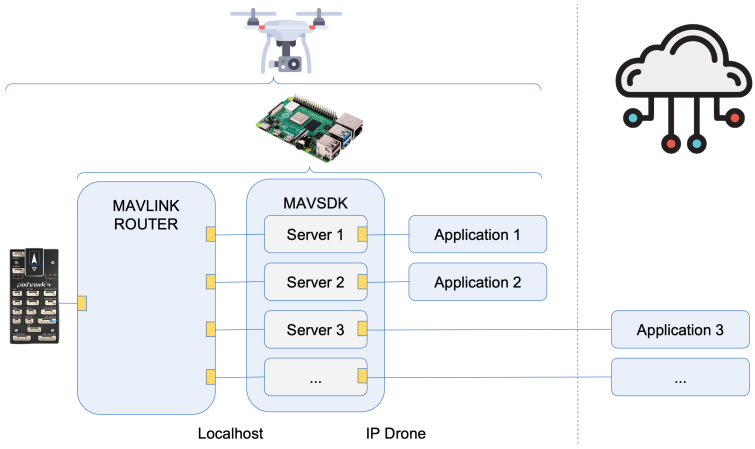
6.3 Platform overview

The solution in this chapter is twofold. Foremost, a drone communication system allows applications to be developed that communicate to the drone fleet from the cloud. Secondly, a platform is built to manage the cluster with its drones and applications. The platform also takes care of the correct deployment of the drone communication system.

6.3.1 Drone Communication System

Flight controllers are used to pilot the drone. These are connected to the hardware of the drone and can control the drone via provided SDK-software. The included autopilot provides flight control software to pilot the drone. The two most used open-source autopilots are ArduPilot [22] and Pixhawk [23]. In this chapter the flight controller is the Pixhawk 4 [24], resulting in Pixhawk being used as an autopilot. To deploy applications to the drone, a Raspberry Pi 4 [25] is used as an onboard computer, providing extra resources to the drone, as well as sidestepping any battery dependency during development.
To communicate with the autopilot, the Mavlink [26] protocol is used. A Mavlink-router [27], combined with MavSDK [28] servers allow for communication with a drone. Via this system, an application can reach and control the drone, be it from the drone itself or from the cloud.
In Figure 6.1, an overview is given of the drone communication system. Via a serial Telem2 connection, the onboard computer connects to the Pixhawk autopilot. A containerized Mavlink Router then opens up access to the flight controller, which allows multiple applications to control the drone. To talk with the drone from anywhere, a MavSDK Server is required. A MavSDK container can start multiple servers that connect to the Mavlink Router and serve the connection to the drone on a configurable port number at the IP address of the drone. Other applications can connect to this server using the aforementioned IP address and port number combination. Crucial is the ability for containerized applications to connect from both

Figure 6.1 Drone communication system

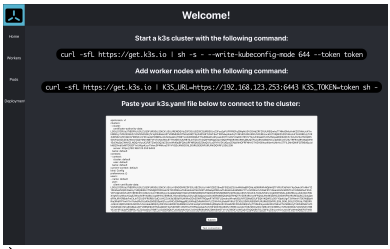


the cloud and the drone, making it possible to switch applications with the same configuration from drone to cloud and otherwise without any necessary configuration changes.

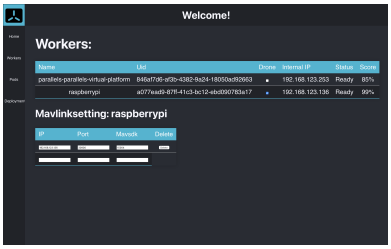
6.3.2 Management platform

The drone communication system allows applications to control the drone from the drone itself or from the cloud. To manage this system, a management platform is built, consisting of a backend, a frontend and a database to store the user configuration. The platform can be connected to a cluster by providing the platform with the KUBECONFIG file created by K3S called *k3s.yaml*. After setting up the connection with the cluster, the platform gives an overview of the nodes, deployments, and pods in the cluster.
In the nodes overview, all computers that joined the cluster are listed, along with some metadata. Per node, a score is given as well, indicating the health of the node by taking into account its CPU, memory, and bandwidth usage. On top of that, the platform allows the end-user to indicate if a node is a drone or not. This is required in order to allow the cluster to deploy applications either in the cloud or on the drone because some workloads must be deployed on a drone or vice versa. Upon indication, the Mavlink Router and MavSDK containers are instantly and automatically deployed to the drone, while an interface appears in the platform to manage the amount of MavSDK servers along with the ports used to serve on.
In a deployments overview, all application deployments are grouped by namespace. With a dropdown menu, it is possible to pin deployments to a drone, to the cloud or don't interfere with the deployment and let K3S take

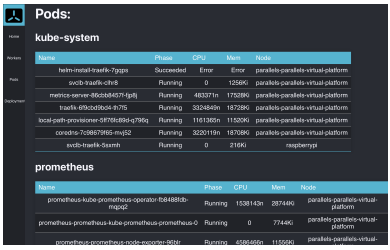
Figure 6.2 Overview of the management platform



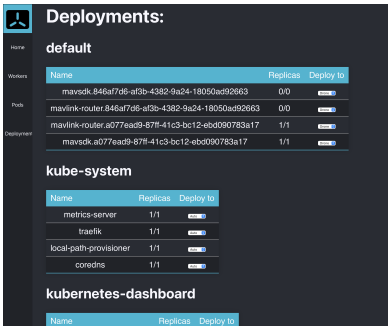
(a) Home page to connect to Kubernetes cluster



(b) Nodes overview



(c) Pods overview



(d) Deployments overview

care of the decision.

Finally, the pods' overview displays all pods, which are typically part of a deployment. Therefore, by pinning a deployment to a drone or a cloud, the platform will make sure all pods are deployed to the correct node type. The pods are again grouped by namespace and show other metadata such as the node it is deployed to, the pod's status, and to what node a pod is deployed.

An overview of these web pages are illustrated in Figure 6.2.

6.4 Emergency Use Case

In case of an emergency situation, for example a large building fire, a drone could be used to gain rapid insights into such a dangerous situation. In [29], a decision support system is introduced to aid first incident responders on scene based on live drone feeds. A custom-designed drone can fly autonomously to the incident area to capture important video feeds from the incident and these video feeds are afterwards processed with object detection and decision support software in order to gain more detailed insights. An

example of a possible outcome is when a victim is laying unconscious close to a barrel with possible explosive goods and a fire is nearby. The decision support system will then trigger an alert so the commander at scene can give priority to this situation.

When thinking about such a system, multiple questions must be resolved: (i) How will the video feed be delivered to the cloud, (ii) Can the drone be piloted securely when flying Beyond-Visual-Line-of-Sight (BVLOS), (iii) Can the cloud handle and process the video feeds in (near)-realtime and (iv) can multiple drones be orchestrated in such a use case. An answer to all these questions is provided in the following subsection. The first subsection handles the network connection between the drone and the cloud, answering questions (i) and (ii) while the second subsection answers questions related to (iii) and (iv) by the use of cloud technologies and the proposed platform in this chapter.

6.4.1 Network connection between drones and the cloud

With the advent of 5G-networks, network slicing concepts are introduced. These network slicing concepts can be implemented with the aid of SDN networks, as stated in [14], Section 1. In case of an emergency situation, a high-priority slice will receive the necessary requirements in terms of bandwidth, delay and jitter based on the previous set requirements while another slice for non-priority traffic will obtain any remaining resources. This concept allows the high-priority slice to maintain its Quality-of-Service (QoS) constraints.

In our previous research [14], Section 4, we introduced a distributed emergency flow prioritization model, which will calculate and guarantee the required bandwidth for the emergency slices (or emergency flows) and will maximize the best-effort flows over the remaining bandwidth. The proposed ILP-model (Integer-Linear Programming) to optimize the flows within the network together with the online approach to allocate new incoming slices in between the ILP calculations are used in the context of this chapter.

6.4.2 Cloud-based drone application management

By deploying a k3s cluster to the cloud and deploy k3s on the drones so they can function as worker nodes. The network connection within the cloud itself and between the cloud and the connected drones is done via the OpenFlow protocol, enabling the use of the network slicing concepts within SDN as stated in Section 6.4.1.

Kubernetes already supports the use of *PriorityClasses* for allocating and scheduling pods inside its cluster. In case of an emergency, the Pod responsible for analyzing the incoming data (e.g. video feeds and other drone data) can by prioritized above the other running pods, making it possible to analyze the data with the necessary amount of resources.

6.5 Evaluation

In this section, the evaluation of the platform and the emergency use case is conducted. First, in Section 6.5.1, the prototyped drone-cloud platform is evaluated in terms of ease of use, resource overhead, scalability, deployment times and node disconnection and secondly, the evaluation of the emergency use case in terms of Quality-of-Service on network level is explained in Section 6.5.2

6.5.1 Platform Evaluation

The proposed communication system and platform are created to focus on a low-resource environment and the possibility to offload resource-intensive tasks to the cloud. To evaluate these goals, several aspects of the platform are considered and evaluated by using three demo applications, namely a telemetry, a video streaming and a resource-intensive application, explained briefly in Section 6.5.1.1. A drone is simulated by means of a Raspberry Pi 4, with a Quad core Cortex-A72 (1.5GHz) processor and 4GB of RAM memory, connected to the Pixhawk 4 autopilot while the cloud is simulated by means of a single Ubuntu VM with a Dual core i7 (2.3GHz) processor and 8GB of RAM memory. Scalability is not a research purpose of this evaluation, hence a single node cluster will suffice. Monitoring was set up using Prometheus [30] and Grafana [31] to conduct these evaluations. A schematic overview of this setup is shown in Figure 6.3.

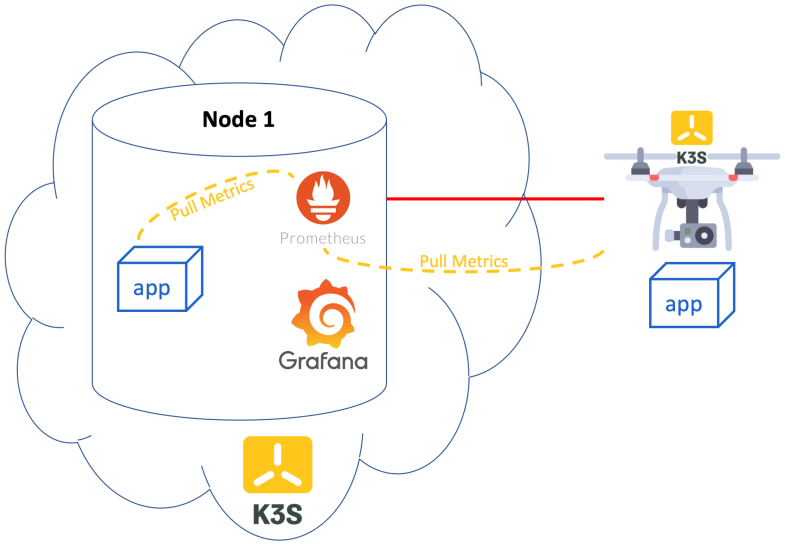
6.5.1.1 Demo applications

Three demo applications are used to showcase the usability of the platform. These applications were deliberately chosen to evaluate different aspects of the proposed platform and its communication system:

Telemetry application

A simple demo application showcases the Mavlink bidirectional communication from drone to cloud and back. The Python application running in the cloud only needs the IP address of the drone it should connect to along with the port of the running MavSDK server. After

Figure 6.3 Evaluation setup: The top side shows the cloud containing, simulated by an Ubuntu Server VM, and running the monitoring services Prometheus and Grafana. The bottom side shows the drone simulated by a Raspberry Pi 4 connected to a PixHawk 4 Autopilot.



connecting to the MavSDK server, the application will continuously read the battery level, GPS info, if the drone is flying, and the drone's position. By building a container image for both the drone and the cloud's architectures, the application can be easily switched from drone to cloud and vice versa by using the platform. This is required due to the different processor architecture: the drone has an arm64 architecture while the cloud has a x86-x64 processor architecture.

Video streaming

The video streaming application consists of two parts. The first part resides on the drone and streams an onboard Logitech C920 camera using cVLC, the headless version of VLC [32]. The second part resides in the cloud and captures the stream. It then applies face detection on the stream to mimic video streaming / object detection drone applications. This way, the application showcases the possibility to split applications functionality partly on the drone (camera feed capture and stream communication) and partly in the cloud (facial recognition algorithms) and deploy them via the platform, allowing more resource-intensive tasks to be deployed to the cloud.

Listing 3 Command to start a K3S cluster

```
$ curl -sfL https://get.k3s.io  | \
  sh -s - --write-kubeconfig-mode 644 --token token
```

Listing 4 Command to add a worker node to the cluster. Use the token provided or generated in Listing 3.

```
$ curl -sfL https://get.k3s.io | \
  K3S_URL=https://192.168.123.235:6443 K3S_TOKEN=token sh -
```

Resource-intensive application

The third and last demo application is designed to be heavy in terms of CPU usage. It is based on a Mandelbrot application created to enable high CPU loads on a computer. The application consists of an API to create workers used for Mandelbrot calculations and to let the workers start calculating. This application was already Dockerized. Upon creation of the workers, containers were created on the same Docker engine to start the calculations there. In this chapter, this application was adapted to be used with Kubernetes, allowing to deploy the application to a K3S cluster. Upon the creation of workers, pods are now wrapped around the resulting containers to be deployed to either drones or the cloud, depending on the configuration.

6.5.1.2 Ease of use

In terms of cluster setup, K3S is significantly easier to set up than a full-fledged Kubernetes distribution as only a single command is necessary to get a K3S cluster up and running, as shown in Listing 3. Adding new worker nodes to the cluster, be it drones or additional compute resources for the cloud, is remarkably simple as well. A single command will trigger the installation script on the worker node, connecting it to the cluster as provided in Listing 4.
Regarding the platform setup, all components (frontend, backend, database) were containerized and combined by means of Docker Compose [33]. A single command spins up all the platform's components.
Ease of use is an important design principle for the user experience of this platform. The platform gives a clear overview of the existing cloud and drone workers, pods, and deployments. Secondarily, it is straightforward to assign a newly added node as a drone, and configuring the Mavlink settings to be used by new applications. Finally, the platform eases pinning applications to the drone or the cloud and makes it possible to redeploy

Figure 6.4 CPU usage on Raspberry Pi with k3s cluster and monitoring (Prometheus and Grafana) pods running

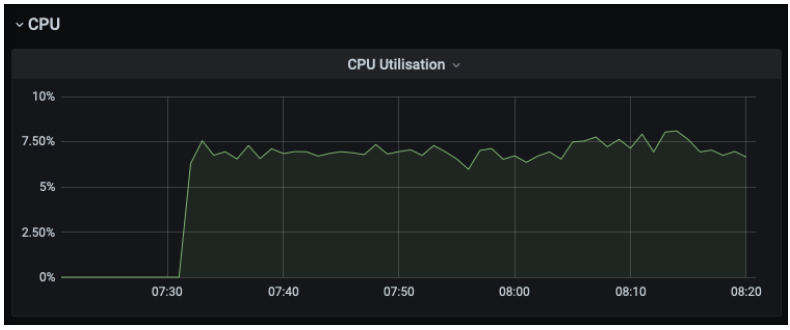
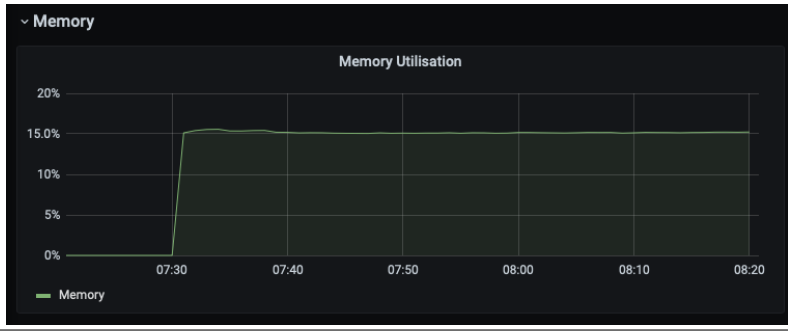


Figure 6.5 Memory usage on Raspberry Pi with k3s cluster and monitoring (Prometheus and Grafana) pods running



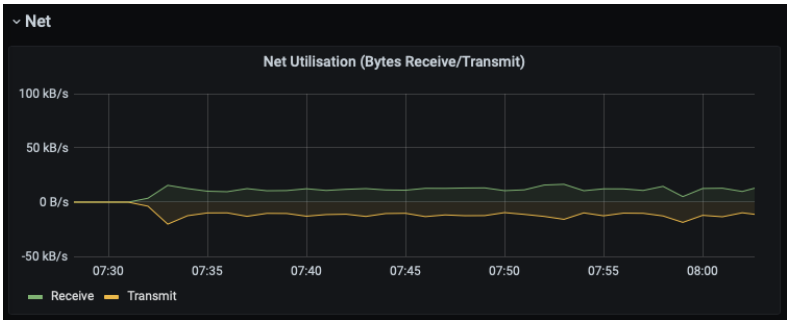
them easily.

6.5.1.3 Resource overhead

Having a system that orchestrates all application containers has its benefits but comes at a cost too, since the system imposes an overhead to work correctly. The developed platform to control the cluster is created in such a way that it is not part of the cluster and it can run on any other machine be it in the cloud or a local setup (but not on a drone). The k3s cluster does however consume resources which are discussed below along with the measurements on the resource usage of the demo applications.

Idle resource overhead: When running the k3s cluster in an idle state with Prometheus and Grafana deployed using the Ubuntu VM serving as the cloud, and the Raspberry Pi as the drone, the CPU usage on the drone is shown in Figure 6.4. An average of 7% of the CPU of a single core is used, keeping in mind that the Raspberry Pi has 4 cores, which can

Figure 6.6 Bandwidth usage on Raspberry Pi with k3s cluster and monitoring (Prometheus and Grafana) pods running



total to a maximum of 400%. In terms of memory, Figure 6.5 shows that approximately 15% of the 4 GBs of RAM is occupied, resulting in a memory usage of 600 MiB. Considering the bandwidth usage of the drone shown in Figure 6.6, a steady 12 kB/s for both the transmission and reception can be observed. Of course, these results also include resources occupied by the operating system and other applications that might be running by default, such as serving the SSH server and having Docker enabled. More advanced metrics are used to better understand the resource usage of the cluster itself. In Figure 6.7, one can observe that out of the 7% CPU usage earlier, 3% can be assigned to the monitoring through Prometheus and Grafana. Concerning the memory usage, Figure 6.8 clarifies that approximately 100 MiB out of the 600 MiB was used by Prometheus and Grafana related pods. The kube-system pods that foresee cluster management were responsible for only 1,5 MiB of memory usage. Further investigating the resource usage reveals that the remaining 4% CPU usage of one core can be assigned to the k3s-agent service that needs to be running on the drone. In terms of memory usage, the k3s-agent takes up approximately only 120 MiB. Furthermore, another 100 MiB of memory could be assigned to Docker, which is not strictly necessary to have the cluster running as other, more lightweight, container runtimes can be used.

Mavlink-router and MavSDK resource usage: In this evaluation, the Raspberry Pi was indicated as a drone node which triggers the deployment of the Mavlink-router and MavSDK applications wherefore the results can be seen in Figure 6.9, 6.10 and 6.11. The Mavlink-router application has a negligible amount of CPU usage and only 1 MiB of memory in use. Regarding the pod bandwidth usage, an initial speed of 10 kB/s can be perceived which settles down to approximately 3 kB/s for both the reception and transmission of data from that pod. When looking at the

Figure 6.7 CPU usage of the monitoring pods (Prometheus and Grafana) on the Raspberry Pi

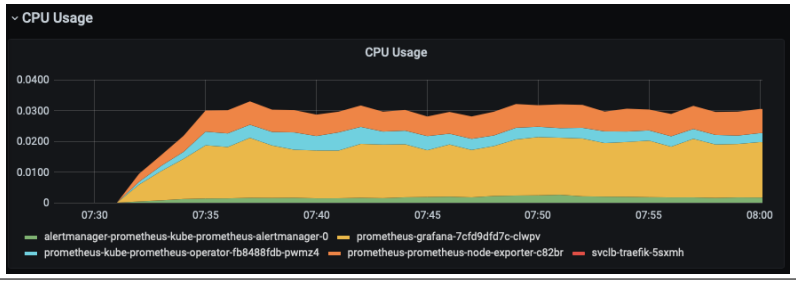


Figure 6.8 Memory usage of the monitoring pods (Prometheus and Grafana) on the Raspberry Pi

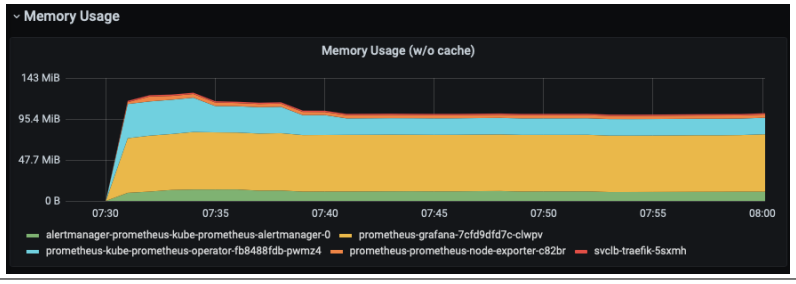


Figure 6.9 CPU usage on Raspberry Pi by the Mavlink-router and Mavsdk containers

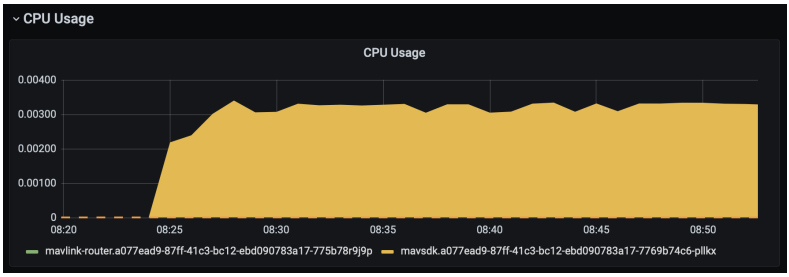
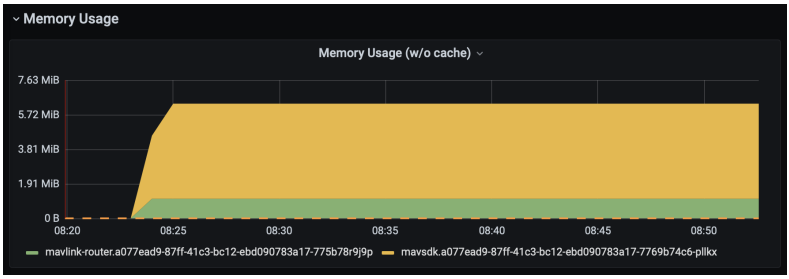


Figure 6.10 Memory usage on Raspberry Pi by the Mavlink-router and Mavsdk containers



usage of MavSDK, very similar results are perceived: a CPU usage of 0,3% of a core with 5 MiB of memory and the same network usage starting off at 10 kiB/s, which then drops down to 3 approximately kiB/s.

Telemetry resource usage: After the deployment of the telemetry application, only 0.4% of a core of CPU is used, combined with 19 MiB of memory. When looking at the network usage, a sole 15 kB/s is transmitted and received.

Video streaming and processing resource usage: As explained in Section 6.5.1.1, the streaming application consists of two parts. One part is deployed to the drone and serves as a streamer, the other part of the application is deployed to the cloud and processes the received stream. With the VLCstreamer running on the drone, 1% of a core of CPU and 10 MiB of memory are used. The network speed levels at 12 kB/s reception speed, but a speed of 150 kB/s for transmission is perceived. Of course, this corresponds to the data that is streamed for the video. The streamprocessor on the drone has a significantly higher CPU usage at 75% of a core, keeping in mind that the whole VM only has two cores available. Likewise, it received data at a speed of 150 kB/s and only transmitted 0,75 kB/s.

High-resource benchmark test resource usage: The Mandelbrot

Figure 6.11 Bandwidth usage on Raspberry Pi by the Mavlink-router and Mavsdk containers

application is used to showcase the possibility of splitting application loads between the drone and the cloud, and to evaluate how the system responds to high loads. When deploying the application, the main application is deployed to the cloud first. This application could then start worker nodes when demanded via an API call. Configuration can be set to only deploy workers to the drone, only workers to the cloud, or to let the system decide. In both cases where the workers are specifically deployed to the drone or the cloud, all the containers keep running without crashing. On top of that, the containers tried to claim as many resources as possible to finish the task as quickly as possible. When the containers increased their resource usage, this did not get out of control since the other pods are still using some parts of the total resources for their applications. Moreover, both the drone and the cloud remained responsive.

6.5.1.4 Scalability

Kubernetes, and therefore K3S, are built to be highly scalable. The limitations towards scalability in terms of how many drones can join the cluster and how many applications are allowed are therefore dictated by K3S.
Kubernetes provides the following guidelines regarding scalability [34]:

- No more than 100 pods per node

- No more than 5000 nodes
- No more than 150000 total pods
- No more than 300000 total containers

The above guidelines are unlikely to be reached in a cluster with drones, even for a whole fleet of drones. If, in extraordinary circumstances, the cluster would come close to these guidelines, the tasks can be split up into multiple clusters.

6.5.1.5 Resource overload

To evaluate how the system responds when deployed applications are too heavy for the given drone, multiple factors need to be taken into account as it highly depends on the situation. Given a new application in a single pod that is not pinned to the cloud or a drone, the system chooses where to deploy the application based on the available resources. It is possible to fill in the required resources for the application in a deployment file, as the system will take these into account. However, once the K3S container orchestrator has deployed the container, the container will typically keep running on that machine, even if the resource usage increases. If the resource needs become unsatisfiable, the performance of the application will go down since it does not have enough resources available to keep up with the demand.
If the application needs to be switched, this can happen via the platform, and the container orchestrator will start a new container. By default, the old container is not removed until the new container is running. This behavior can be tweaked in the deployment file, if necessary.
When dealing with applications that come with multiple replicas of the same pods, these replicas might appear on different nodes if the desired nodes are not specified. This way, the system will try to schedule new pods smartly.
In conclusion, it is important to notice that deployed applications will typically not crash if their resource usage increases but require manual intervention to move them from drone to cloud or vice versa.

6.5.1.6 Platform deployment times

To evaluate deployment times for applications to the cluster via the platform, tests are conducted for which the results are shown in Table 6.1. Each test is executed three times. For every test, all container images are pre-pulled, so the size of the images does not affect the test.
A first test, referred to as tele-cloud in the table, is conducted by deploying the telemetry application to the cloud. On average, this took 4 seconds, with a maximum of 7 seconds before the container starts running. Deploying the

Table 6.1: Deployment times for the platform

	tele-cloud	tele-drone	deploy MCS	switch drone-cloud
min	00:00:02	00:00:03	00:00:02	00:00:02
avg	00:00:04	00:00:04	00:00:03	00:00:02
max	00:00:07	00:00:06	00:00:05	00:00:03

telemetry application to the drone, referred to as tele-drone in the table, similarly takes 4 seconds on average and a maximum of 6 seconds. As a third test, the Raspberry Pi is indicated as a drone on the platform, which triggers the Mavlink-router and MavSDK applications to be deployed to the drone. The deployment of the Mavlink communication system, referred to as deploy MCS (Mavlink Communication System) in the table, takes 3 seconds on average and a maximum of 5 seconds. Finally, switching applications from drone to cloud or from cloud to drone is evaluated. Both cases have the same test results so they are grouped in the same column (switch drone-cloud). Moving a deployment from the drone to the cloud or vice versa takes 2 seconds on average and 3 seconds maximum for the container to be running.

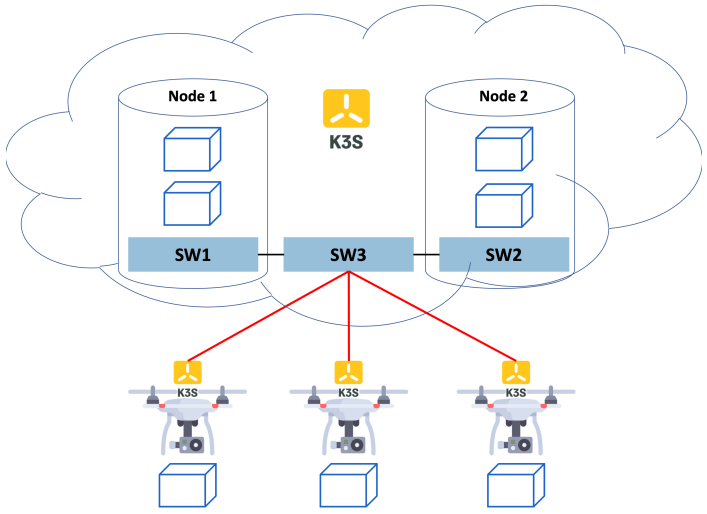
6.5.1.7 Node disconnection

During this research, a stable internet connection is assumed for the drone. However, when this is not guaranteed, all applications on the drone will keep running but without functionalities that require connectivity. Reconnecting the drone results in the node automatically rejoining the cluster and resuming its activities. It does however require applications to be developed to be resistant against network failures. The cluster's behavior is tested by manually disconnecting the Raspberry Pi 4 from the network.
In case a drone chrased or is unable to reconnect at all, an extra solution must be added to the platform in order to tag these drones so that all the running applications on these drones can be transferred automatically to another drone or the the cloud. This can be done with e.g. using deployments within Kubernetes.

6.5.2 Emergency use case evaluation

In this section, the emergency use case setup is evaluated in terms of Quality-of-Service on network level, allowing to use the emergency network slicing concepts. A schematic overview of this setup is illustrated in Figure 6.12. For this evaluation, 3 drones are emulated using 3 Raspberry Pi's 4

Figure 6.12 Setup for the evaluation of the created platform. The left side shows the cloud which is simulated by one Ubuntu Server VM and the right side shows the drone simulated by a Raspberry Pi 4 connected to a PixHawk 4 Autopilot.



connected over a WiFi connection to the cloud which are 2 laptops (nodes) running Ubuntu Server 20.04 LTS as operating system. The WiFi connection is established using a Zodiac WX switch (sw3 in the Figure 6.12), which is an SDN enabled switch, allowing us to simulate network slices. The Zodiac WX is connected with two Zodiac FX SDN switches (sw1 and sw2) and each switch is connected to a node (laptop) of our k3s cluster. The total available bandwidth is 100 MBit/s and the OpenFlow management of the SDN switches is done through the control plane of the k3s cluster, containing the ILP algorithm together with the online approach.

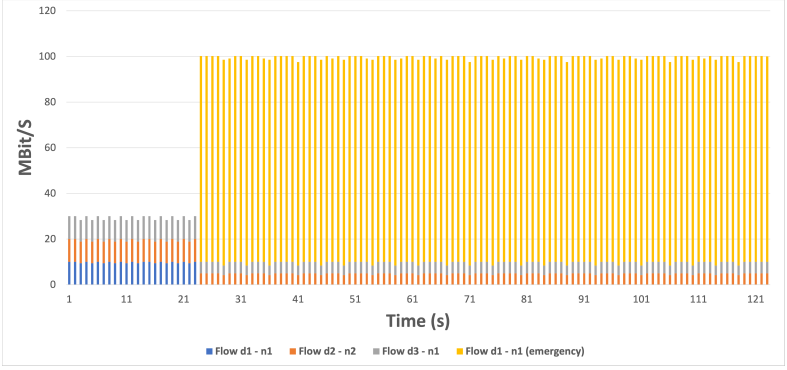
6.5.2.1 Evaluation at network level

Each drone in the evaluation setup runs the IPerf command as a client, where two drones connect to a Pod on node 1 and the third drone connects to a Pod on node 2. Three traffic classes are defined and shown in Table 6.2, the results of the experiment are illustrated in Figure 6.13 and further explained. From time 1 until 24, the three drones requested the normal priority traffic class with 10 MBit/s and they were allocated accordingly by the OpenFlow management. At time 25, an emergency flow, going from drone 1 to a Pod on node 1, is requested and allocated by the online approach with a bandwidth of 90 MBit/s. The online evaluation also downgrades the other

Table 6.2: Traffic classes (all in MBit/s)

Id	Name	Minimum Rate	Maximum Rate
1	High Priority	0	90
2	Normal Priority	0	10
3	Low Priority	0	5

Figure 6.13 Network speed between the drones, running the IPerf client, and the nodes, running the IPerf server. Between time 1 and 24, the three drones are using traffic class 2 and as from time 25, the communication from drone 1 to node 1 has become emergency traffic with a requested speed of 90 MBit/s. This results in the reduction of the traffic class to 5 MBit/s of the two remaining best effort flows.



flows to traffic class 3 so the total bandwidth of 100 MBit/s is not exceeded. At time 40, the ILP had completed a new calculation cycle and came with the same outcome as the online approach.

It is clear that the emergency flow received all the requested bandwidth and the other best-effort flows, drone 2 to node 2 and drone 3 to node 1, were downgraded to a lower priority in order to not exceeded to total bandwidth.

6.6 Conclusion and Future Work

This chapter aims to design and implement a container-based deployment system that allows for container orchestration and aids in scheduling applications to the cloud or drones whilst keeping in mind that resources on the drone are rather limited. This in combination with NFV network slicing concepts resulted in a emergency use case whereas a drone can have the highest priority in the network while the other network flows are allocated

on the remaining available bandwidth.
After research is conducted about the current state-of-the-art, a system is developed to allow applications to communicate with a drone from the cloud and the drone itself with no extra configuration. This system is incorporated in a cloud platform to manage drones and their applications. Finally, three demo applications are developed to evaluate the platform. As discussed during the evaluation, this system and platform proved promising, showing that it is possible to use drones and a cloud both as part of a single cluster. Moreover, this system provides tremendous scaling possibilities, along with the potential to manage a fleet of drones with a resource usage overhead of about 4 % CPU and 12O MiB RAM that is acceptable for a drone. In creating this system, a step is taken in the right direction to facilitate even higher resource-demanding applications for drones to be made as they can now be more conveniently managed. On top of that, the possibility to manage drone fleets could translate into industrial applications such as warehouse operations, cooperating drones, or package deliveries.
This chapter shows an implementation of using a container orchestrator to manage drones with resource offloading capabilities to the cloud. However, some aspects could use further research in this field that relate to this chapter. A logical continuation is further development of the platform. As it is a Proof-of-Concept, the platform can be made more stable, and production-ready by adding authorization and authentication so it can support multiple clusters simultaneously. The platform can also be evaluated using different hardware than the Raspberry Pi 4 that was used. The container runtime used in this chapter is containerd [35], as it is the default in K3S. Other container runtimes such as Kata [36], CRI-O [37], crun [38] or gvisor [39] could be subject of future research as they might provide even less resource overhead. Further, KubeEdge [11], or other container orchestrators can be evaluated as to how suitable they are for drones. Finally, bringing edge computing to drones for deep learning or AI tasks or AI based UAV-network are promising venues for future studies [40, 41].

References

[1] Kurt W Smith. Drone technology: Benefits, risks, and legal considerations. *Seattle J. Envtl. L.*, 5:i, 2015.

[2] Drone technology uses and applications for commercial, industrial and military drones in 2021 and the future, Jan 2021. URL https://www.businessinsider.com/drone-technology-uses-applications?international=true&r=US&IR=T.

[3] About the AuthorJosh Pozner, About the Author, and Josh Pozner. A comprehensive list of commercial drone use cases (128 and growing), Nov 2020. URL https://www.dronegenuity.com/commercial-drone-use-cases-comprehensive-list/.

[4] Netflix Technology Blog. Titus, the netflix container management platform, is now open source, Apr 2018. URL https://netflixtechblog.com/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436.

[5] Spotify case study, Sep 2020. URL https://kubernetes.io/case-studies/spotify/.

[6] Google kubernetes engine, 2021. URL https://cloud.google.com/kubernetes-engine.

[7] New york times case study, Nov 2020. URL https://kubernetes.io/case-studies/newyorktimes/.

[8] Production-grade container orchestration. URL https://kubernetes.io/.

[9] K3s: Lightweight kubernetes, 2021. URL https://k3s.io/.

[10] k0s | kubernetes distribution for bare-metal, on-prem, edge, iot, 2021. URL https://k0sproject.io.

[11] Kubeedge, Oct 2021. URL https://kubeedge.io/en/.

[12] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*, pages 65–73, 2021.

[13] Jerico Moeyersons, Martijn Gevaert, Karl-Erik Réculé, Bruno Volckaert, and Filip De Turck. Uavs-as-a-service: Cloud-based remote application management for drones. In *Manage-ioT2021, part of IM2021, the IFIP/IEEE Symposium on Integrated Network and Service Management*, pages 1–6, 2021.

[14] Jerico Moeyersons, Behrooz Farkiani, Tim Wauters, Bruno Volckaert, and Filip De Turck. Towards distributed emergency flow prioritization in software-defined networks. *International Journal of Network Management*, 31(1):e2127, 2021.

[15] G. Mehrooz, E. Ebeid, and P. Schneider-Kamp. System design of an open-source cloud-based framework for internet of drones application.

In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 572–579, Aug 2019. doi: 10.1109/DSD.2019.00087.

[16] Powering the world's robots. URL https://www.ros.org/.

[17] Alexander Hof and Jason Nieh. Androne: Virtual drone computing in the cloud. In *EuroSys '19: Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 03 2019. ISBN 9781450362818. doi: 10.1145/3302424.3303969.

[18] Android things : Android developers. URL https://developer.android.com/things.

[19] Tom Goethals, Filip De Turck, and Bruno Volckaert. Fledge: Kubernetes compatible container orchestration on low-resource edge devices. In *International Conference on Internet of Vehicles*, pages 174–189. Springer, 2019.

[20] A. Koubâa, B. Qureshi, M. Sriti, Y. Javed, and E. Tovar. A service-oriented cloud-based management system for the internet-of-drones. In *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 329–335, 2017. doi: 10.1109/ICARSC.2017.7964096.

[21] Chen Hong and Dianxi Shi. A cloud-based control system architecture for multi-uav. In *Proceedings of the 3rd International Conference on Robotics, Control and Automation*, ICRCA '18, page 25–30, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365307. doi: 10.1145/3265639.3265652. URL https://doi.org/10.1145/3265639.3265652.

[22] Ardupilot, 2021. URL https://ardupilot.org/.

[23] Pixhawk, 2021. URL https://pixhawk.org/.

[24] PX4 Dev Team. Pixhawk 4 autopilot, 2021. URL https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html.

[25] Raspberry pi website, 2021. URL https://www.raspberrypi.org/.

[26] . URL https://mavlink.io/.

[27] Mavlink-Router. mavlink-router/mavlink-router. URL https://github.com/mavlink-router/mavlink-router.

[28] Mavsdk, . URL https://mavsdk.mavlink.io/main/en/index.html.

[29] Jerico Moeyersons, Pieter-Jan Maenhaut, Filip De Turck, and Bruno Volckaert. Aiding first incident responders using a decision support system based on live drone feeds. In *International Symposium on Knowledge and Systems Sciences*, pages 87–100. Springer, 2018.

[30] Prometheus. Prometheus - monitoring system & time series database. URL https://prometheus.io/.

[31] Grafana: The open observability platform. URL https://grafana.com/.

[32] VideoLAN. Vlc media player. URL https://www.videolan.org/.

[33] Overview of docker compose, Apr 2021. URL https://docs.docker.com/compose/.

[34] Considerations for large clusters, Feb 2021. URL https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[35] URL https://containerd.io/.

[36] Kata containers - open source container runtime software. URL https://katacontainers.io/.

[37] Cri-o. URL https://cri-o.io/.

[38] Containers. containers/crun. URL https://github.com/containers/crun.

[39] Google. google/gvisor. URL https://github.com/google/gvisor.

[40] Mohamed-Amine Lahmeri, Mustafa A. Kishk, and Mohamed-Slim Alouini. Artificial intelligence for uav-enabled wireless networks: A survey. *IEEE Open Journal of the Communications Society*, 2:1015–1040, 2021. doi: 10.1109/OJCOMS.2021.3075201.

[41] Tom Goethals, Bruno Volckaert, and Filip De Turck. Enabling and leveraging ai in the intelligent edge: A review of current trends and future directions. *IEEE Open Journal of the Communications Society*, 2:2311–2341, 2021. doi: 10.1109/OJCOMS.2021.3116437.

7

Conclusions and Future Research Directions

Drone technology choices are increasing every year, and even though it is still a relatively new technology, the research is already spreading its wings. [1] Today, drones are used in industry for different purposes such as inspections and surveillance, but also used in private (recreational) space for e.g. aerial photography and in the public sector as support for emergency services. Therefore, many drone applications already exist but the management of these applications together with the management of the network is still a fundamental shortcoming. This dissertation investigates several challenges concerning the improvement of managing containerized drone applications together with ameliorating the network management based on SDN technologies, resulting in the creation of a new UAVaaS platform, handling the tasks of container and network orchestrator. Four hypotheses shape the direction and outcome of the research conducted in this dissertation, as listed below:

RH1: Managing heterogeneous SDN networks while maintaining QoS constraints should be feasible with a negligible overhead on the performance of the network.

RH2: Ensure high-priority network traffic and optimize the remaining network traffic over the remaining resources.

RH3: Design and develop a UAV-aware cloud solution with minimal resource impact on the drones. This solution must be able to migrate containerised application components from the drone to the cloud and vice versa from an easy-to-use web platform.

RH4: Design and development an unobtrusive monitoring solution that has a negligble impact on the monitored services.

This dissertation introduces a number of solutions that contribute to addressing the formulated questions and hypotheses in various application domains including cloud computing, network management, smart cities, and drone development. The sections below summarize these contributions.

7.1 Ensure high-priority network traffic in managed heterogeneous SDN networks

In line with the first two research questions and hypotheses, the first step towards the new UAVaaS platform was to design and implement a network management system. Based on the recent mobile trends in terms of 5G, SDN was chosen where a controller or controllers deployed in the cloud can handle and manage the whole network. With SDN, it is possible to create network flows and apply QoS constraints in terms of bandwidth, delay and jitter. Managing homogeneous SDN networks with multiple SDN controllers is already possible, but in Chapter 2, this dissertation proposed a plugin-based microservice framework called Domino to manage heterogeneous SDN networks with multiple SDN controllers. The architecture allows developers to design and implement new algorithms to manage the whole network without taking into account which type of SDN controllers are used, because this is handled by Domino itself. When new SDN controllers are released, these can be also be added to Domino as a plugin. A prototyped framework is evaluated in terms of performance, interoperability and modifiability and shows that:

1. The frequently used commands are executed with an average response time of 0.26 seconds.
2. The size of the core components of the framework is small with an average size of 105 MB per image.
3. The framework can handle at least 73 plugins simultaneously without any performance downgrade on a server with 4GB of RAM.
4. The interoperability requirements are all met by the framework with most exchanges having a 100% successful exchange ratio.

5. The modifiability requirements regarding the plugins and the deployment options are met by the framework.

The next step, explained in Chapter 3, was to ensure bandwidth allocation for emergency flows while other best-effort flows are allocated over the remaining bandwidth. An offline approach is designed based on a LP and an ILP model, which take into account the current state of the network together with the available network traffic classes, and calculate the bandwidth allocation for the emergency and best-effort flows. These two models are first evaluated by means of a simulation where 500 best-effort flows and 50 emergency flows have to be allocated. The LP model works faster than the ILP model (500ms and 17.000ms respectively), but the technologies used in the LP model are not compatible with our available hardware. Therefore, the ILP model is used, but new incoming flows have to be handled in between calculations of the ILP model. An online approach based on a greedy heuristic solves this issue and prototype evaluations show that the online problem efficiently handles new incoming flows while guaranteeing the bandwidth for all the emergency flows and providing a sub-optimal temporary solution for the best-effort flows. In case of very large networks, or when a SDN controller has a lack of memory, the management can be distributed over multiple heterogeneous SDN controllers.

7.2 Design and development of a UAVaaS platform

Research Hypothesis 3 (**RH3**) states that a UAV-aware cloud solution can be developed with a minimal resource impact on the drones. In Chapter 5, the UG-One platform is proposed, allowing to deploy containerized applications on a drone or in the cloud allowing developers with no drone experience to design, develop and deploy applications on a drone. With UG-one, the required containers to control the drone with software is automatically deployed when a drone is connected to the platform. This allows applications on the drone itself, in the cloud or even ground control stations to control a specific drone. A video streaming, stress testing and drone control application are developed and deployed on the prototyped platform to validate and demonstrate its capabilities.
One of shortcomings of the UG-one platform was the container orchestrator itself, adding extra features such as allowing to automatically redeploy running containers in case of fatal errors. Therefore, in Chapter 6, K3S is used as container orchestrator for the UAVaaS platform. K3S is based on the industry de-facto standard Kubernetes, but optimized for use on

edge devices, such as drones. An extra management layer, consisting of a backend, a frontend and a database to store user configuration, is developed and deployed in the same K3S cluster for ease of management. Every node in the cluster, either in the cloud or as a drone, is visualised through the management application, together with the resource usage of each node. Evaluation of the new platform shows that it is possible to use drones and in combination with a cloud backend as part of a single cluster. Moreover, the system provides good scaling possibilities, along with the potential to manage a fleet of drones with a resource usage overhead of about 4 % CPU and 12O MiB RAM, which is acceptable for a drone. Also, managerial control over the network resources within the cluster is possible with the framework. The outcome of Chapter 2, Chapter 3 and Appendix A are used to show that the SDN network slicing concepts can be used for configuring emergency network slices. On top of that, the possibility to manage drone fleets could translate into industrial applications such as warehouse operations, cooperating drones, or package deliveries.

7.3 Monitoring of complex microservice offerings in an unobtrusive manner

The last Research Hypothesis 4 (**RH4**) states that an unobtrusive monitoring solution can be developed to observe containerized services in the cloud. Chapter 4 describes such a system based on the sidecar container pattern. When applications are deployed to the cloud, these applications can have a complex microservice architecture relying on third-party services, making it difficult to keep the system error free. Typical problems are for example undocumented changes or undocumented architectural specifications of third-party services and bugs in the applications itself. Service monitoring at application and network level can quickly detect such errors and anomalies, resulting in faster and more reliable fixes for these problems. The proposed agent-based system consists of multiple agents, one per service container that needs to be observed, and an agent management system to orchestrate the agents. The main goal of the agents is to provide insights in the behavior of each service in a non-intrusive way and evaluations of the prototyped system shows that this is done with a minimal overhead of 0.02 % in terms of CPU usage, proving that the agents have a negligible impact on the monitored system. The agent management system collects all the monitored data which can be used to visualize the results afterwards.

7.4 Future Perspectives

This dissertation presented several contributions in the domain of cloud computing and drone application development. Of course, it would not be feasible to exhaust the subject matter within the scope of the research conducted, and therefore some challenges remain open, and promising venues for future studies have emerged as a result of the work documented herein. Below, these future research directions are briefly discussed.

Secure container runtimes

With the creation of the UAVaaS platform, Docker containers are commonly used to deploy and manage the containers in the cloud or on the drone through the Docker CLI or via Kubernetes. Containers are the way to go because they use the kernel of the host, resulting in a small footprint. However, one single vulnerability could jeopardize the isolation of the containers, meaning that one could escape from the container into to host and possibly into other containers. In case the UAVaaS platform will be used by many users (e.g. as a multi tenancy environment), the container approach is less desirable and a more secure solution needs to be investigated [2].

Secure container runtimes can solve this problem as stated by the Open Container Interface (OCI) and come with two implementations [3]. The sandboxed secure containers provide increased isolation between the containerized process and the host, as they do not share a kernel. The process runs on a unikernel or kernel proxy layer, which interacts with the host kernel, thus reducing the attack surface. Examples include gVisor[1] and nabla-containers[2]. The other implementation is through a virtualized runtime, providing increased host isolation by running the containerized process in a (lightweight) virtual machine (through a VM interface) rather than a host kernel. This can make the process slower compared to a native runtime. Examples include kata-containers[3] and the now deprecated clearcontainers[4] and runV[5] which are both migrated to kata-containers.

These secure containers can be easily used afterwards in a cloud orchestrator platform, such as Kubernetes or K3S, because they all rely on the standard Container Runtime Interface (CRI).

[1] https://gvisor.dev
[2] https://nabla-containers.github.io
[3] https://katacontainers.io
[4] https://github.com/clearcontainers
[5] https://github.com/hyperhq/runv

Drone wireless mesh networking

When talking about drones and autonomous flights, the assumption is made that a network connection is available. However, in some regions or places, there are areas where the cellular network is not available or cannot be used. This absence of communication can cause fatal implications [4]. A Wireless Mesh Network (WMN) can solve this problem as the communication can be done over radio nodes (in this case the drones) which are organized in a mesh topology. The main benefits of using a wireless mesh network are:

- All drones are able to communicate with each other without external network dependencies.
- Self-organization and configuration of the network.
- Fault tolerance: the mesh network will continue operating when a node fails.
- Use one or more nodes that are in range of public network connectivity as a gateway for meshed nodes.

However, there are also some downsides of using a mesh network for drones because in a traditional WMN mobility of the nodes is expected to be rather infrequent or otherwise, the network will spend most of its time updating routes. When using drones, mobility will change a lot and have a huge impact on the system. Therefore, a solution must be found in order to successfully integrate mesh networking in a drone context where a trade-off between finding the optimal path with LP and ILP solutions or with online optimizers which could work faster but are less optimal.

Enabling heavy computing tasks on edge devices

Drones are often used for capturing live video feeds which are analysed afterwards by e.g. a deep learning model or AI. This requires high computational and memory requirements for both training and inference phases. Training a deep learning model is space and computationally expensive due to millions of parameters that need to be iteratively refined over multiple time periods. Inference is computationally expensive due to potentially high dimensionality of the input data (e.g., a high-resolution image), and millions of computations that need to be performed on the input data. To meet the computational requirements of deep learning, a common approach is to leverage cloud computing. To use these cloud resources, the data must be moved from the drone at the network edge to a centralized location in the cloud, introducing several challenges such as latency, scalability

and privacy [5]. To overcome these challenges edge computing can be introduced by providing computational abilities close to the drones or on the drones itself. A possible solution is to equip drones with Nvidia Jetson hardware[6], bringing supercomputer performance to the edge in a compact System-On-Module (SOM) with energy efficiency in mind, optimal for installation on drones [6]. Drone movements can also require the dynamic migration or handover to/between edge cloud components on different edge infrastructures. Learning-based algorithms can be designed to facilitate these dynamic migration or handovers.

References

[1] Beth Ewoldsen. Drones are still a new technology, but the research is spreading its wings | national academies. URL https://www.nationalacademies.org/trb/blog/drones-are-still-a-new-technology-but-the-research-is-spreading-its-wings.

[2] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214, 2019. doi: 10.1109/IOTSMS48152.2019.8939164.

[3] Lennart Espe, Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance evaluation of container runtimes. In *CLOSER*, pages 273–281, 2020.

[4] Gudi Chand, Manhee Lee, and Soo Shin. Drone based wireless mesh network for disaster/military environment. *Journal of Computer and Communications*, 06:44–52, 01 2018. doi: 10.4236/jcc.2018.64004.

[5] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proc. IEEE*, 107(8):1655–1674, 2019.

[6] Sparsh Mittal. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, 97:428–442, 2019.

[6] https://www.nvidia.com/en-us/autonomous-machines/jetson-store/

A

Enabling Emergency Flow Prioritization in SDN Networks

⋆⋆⋆

J. Moeyersons, B. Farkiani, B. Bakhshi, S.A. Mirhassani, T. Wauters, B. Volckaert, and F. De Turck.

Published in International Conference on Network and Service Management (CNSM). IEEE, 2019. p. 1-8.

Abstract Emergency services must be able to transfer data with high priority over different networks. With 5G, slicing concepts at mobile network connections are introduced, allowing operators to divide portions of their network for specific use cases. In addition, Software-Defined Networking (SDN) principles allow to assign different Quality-of-Service (QoS) levels to different network slices.

This paper proposes an SDN-based solution, executable both offline and online, that guarantees the required bandwidth for the emergency flows and maximizes the best-effort flows over the remaining bandwidth based on their priority. The offline model allows to optimize the problem for a batch of flow requests, but is computationally expensive, especially the variant where flows can be split up over parallel paths. For practical, dynamic situations, an online approach is proposed that periodically recalculates the optimal solution for all requested flows, while using shortest path routing

and a greedy heuristic for bandwidth allocation for the intermediate flows. Afterwards, the offline approaches are evaluated through simulations while the online approach is validated through physical experiments with SDN switches, both in a scenario with 500 best-effort and 50 emergency flows. The results show that the offline algorithm is able to guarantee the resource allocation for the emergency flows while optimizing the best-effort flows with a sub-second execution time. As a proof-of-concept, a physical setup with Zodiac switches effectively validates the feasibility of the online approach in a realistic setup.

A.1 Introduction

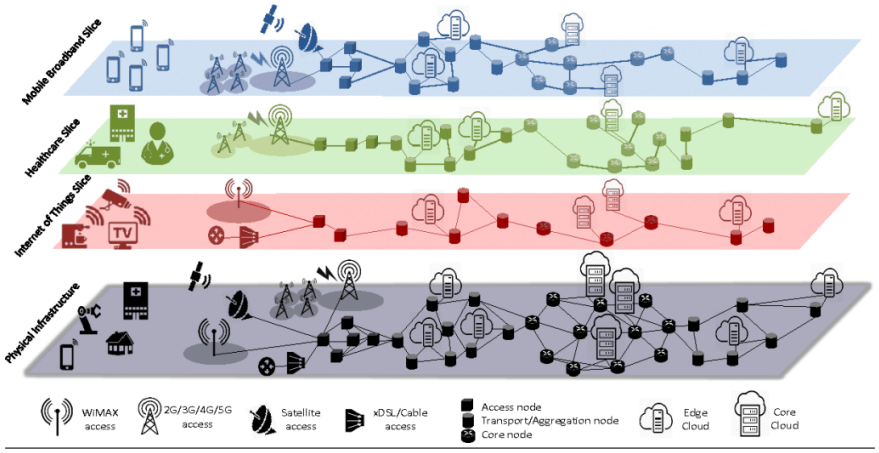
With the expected release of 5G by the end of 2019 [1], slicing concepts at network level will be introduced [2, 3], to allow network operators to provide portions of their networks for specific use cases such as IoT, streaming videos and smart energy grids.

Network slicing is a virtual networking mechanism that is part of the same family as SDN and Network Function Virtualization (NFV), two closely related network virtualization technologies that are moving networks towards automation through software. SDN is an important technology to implement dynamic and flexible network management by separating the data plane from the control plane in networks [4]. Every SDN switch (further called switch) within an SDN network acts like a simple packet forwarding device that is controlled by a logically centralized software program, the SDN controller. NFV on the other hand separates network functions from the underlying proprietary hardware appliances [5]. The network functions running on dedicated hardware are thus transferred to software-based applications running in datacenters, network nodes, end-user premises etc. Complementary to SDN and NFV, network slicing allows the creation of multiple virtual networks atop of a shared physical infrastructure whereby each virtual network has its own specific features depending on the use case. An example of different possible network slices together is illustrated in Figure A.1.

The network slicing concept introduces the possibility to enable new features such as more fine-grained QoS. In this paper we therefore propose an offline and online SDN-based solution to guarantee and optimize flows based on their priority by creating bandwidth meters who are responsible for limiting the maximum allowed bandwidth per flow.

Different emergency events are taken as use cases. During such an event, it is required to prioritize the network traffic that is coming from and going to the emergency services in the presence of large civilian crowds

Figure A.1 5G network slices running on a common underlying multi-vendor and multi-access network. Each slice is independently managed and addresses a particular use case [3].



in order to coordinate the relief and response. Every emergency event is different, and needs different types of network traffic as some situations require high bandwidth for streaming high resolution video feeds and other situations only require a low amount of bandwidth to enable communication or streaming low resolution video feeds. The bandwidth for the emergency traffic should be guaranteed while the network traffic coming from (non-prior) users should be optimized over the remaining available bandwidth. The proposed solution in this paper guarantees the bandwidth of emergency network traffic by generating SDN high-priority flows while other non-priority traffic will receive best-effort resources based on their priority within the network. To evaluate the solution, a topology is simulated whereby the routing of different best-effort flows and emergency flows is optimized. Afterwards, a tree topology is built with Zodiac switches [6] and used for the practical evaluation of the algorithm on a smaller and real topology, controlled by the Ryu SDN controller software [7].

This paper contributes to three main topics: *(i)* design of models for guaranteeing bandwidth for emergency flows while optimizing best-effort flows over the remaining network resources, *(ii)* design of a joint online-offline approach to practically implement the model and *(iii)* the validation of the model through both simulation and practical evaluation. The remainder of this paper is organized as follows: Section A.2 presents related work. In Section A.3, the problem description is given followed by the problem formulation for both splittable and unsplittable flows. Section A.4 presents the evaluation methods and the corresponding results. Finally, Section A.5

discusses conclusions and future avenues of research.

A.2 Related Work

There has been a great deal of research on providing QoS in SDN over the past few years [8]. The authors in [9, 10] presented algorithms to provide QoS, but without considering bandwidth guarantees. In [9] authors proposed a QoS solution based on SDN technology. The authors first defined a cost function which assigns a positive value to each link based on the length, bandwidth and the weight of the link. Then, they utilized the Dijkstra algorithm [11] to find multiple paths for each source and destination pair in the network. When a flow arrives, the path with the lowest congestion is selected as the routing path for the flow. Zhang et al. [10] proposed a QoS framework based on the OpenFlow protocol which dynamically calculates a path for each flow. If the flow is a QoS-required flow, an algorithm based on Dijkstra is used to find a path with the minimum delay and cost values.
An approach to allocate bandwidth and satisfy QoS requirements is presented in [12]. The authors categorized flows into QoS and best effort flows and defined a metric, used in path selection, that considers the requested rates. Shaohua et al. [13] categorized cloud applications into three levels based on the sensitivity to delay and bandwidth. The flow-based adaptive routing algorithm is presented in [14] which utilizes Dijkstra and K-shortest path [8] algorithms with the aim of maximizing the utilization of network resources. The authors evaluated the proposed algorithm through simulation. Pinto et al. [15] defined four service classes including best effort and bandwidth guaranteed classes. Each new flow is first assigned to the probing class and its behavior is monitored. After some time, if the network can support its bandwidth along the path it will be reassigned to the bandwidth guaranteed class or otherwise the best effort class. The authors in [16] designed a method to provide bandwidth guarantees by using OpenFlow meters and queues. The authors categorized flows into QoS flows which have minimum guaranteed bandwidth and best effort flows with no requirements. For each QoS flow, first, an admission control process checks whether there is a path that can accommodate the flow rate. After that, by using a meter at the ingress switch, the input rate of the flow is monitored and if it exceeds the defined rate, the packets will be marked. Using three different queues at the egress port of each switch along the path for marked and unmarked QoS and best effort flows, traffic prioritization is made possible. MPLS tunnels are used in [17] to provide end-to-end bandwidth guarantees. Similar to [16] the authors used

OpenFlow meters at the ingress switches. For each flow the input rate of the flow is monitored and based on that, a priority value is set in the header of each packet. Then, an MPLS tunnel is used to route the packets toward the egress switch and the priority of each packet specifies its output queue. Lu et al. [18] utilized preplanned network slices to both satisfy QoS requirements and maximize the overall throughput of the network. The authors used the traffic history to create network slices which have fixed configurations during the network lifetime. When a flow arrives, it is assigned to a slice by using the VLAN ID of the slice. The MaxStream framework is proposed in [19] to maximize the number of streaming sessions and bandwidth provisioning. The authors formulated two ILP problems. The first problem maximizes the number of accepted flows by considering the requested rate of the flows. Then, the set of accepted flows is used in the second problem to maximize the total rate of the accepted flows. Since the authors focused on multimedia streams, they ignored best effort flows with no QoS requirements.

The most important studies are summarized in Table A.1. In this paper, we utilize both online and offline approaches to provide bandwidth guarantees for emergency flows and maximize the total rate of best effort flows. The offline approach optimizes all existing emergency and best-effort flows while the online approach routes and allocates new incoming flows sub-optimally in between offline batches. The offline approach is defined as two models, a LP and an ILP model and the online approach is based on Dijkstra to route new incoming flows along with a greedy heuristic for bandwidth allocation. We also use OpenFlow meters to implement our method and evaluate it using Zodiac switches [20]. Only drop meter policies are used in this paper because the current OpenFlow versions [21] do not support other policies such as 2-color-marking [22] and 3-color-marking [23].

A.3 Problem Description and Formulation

In this section, the problem described in Section A.1 is first analyzed in detail. Afterwards the offline approach consisting of two formulations is presented, able to guarantee emergency flows while the best-effort flows are optimized over the remaining bandwidth in the network. Finally, the online approach is described which is able to run in practical environments. It combines the offline approach with a sub-optimal solution to handle new incoming flows in between the calculations of the offline approach.

Table A.1: Summary of Related Research

Reference	Offline/Online	Objective	Path Selection	Evaluation
[12]	Online	Satisfy the QoS requirements of the QoS flows	Greedy	Geni Testbed [24]
[15]	Online	Admission control and traffic management	Sink tree	Mininet [25]
[16]	Online	Satisfy the minimum bandwidth requirements of QoS flows	Widest shortest path	Open vSwitch [26] and physical switches
[17]	Online	Guarantee bandwidth of QoS flows	SAMCRA [27] and Dijkstra	Mininet and physical switches
[19]	Online	Maximize the number of streaming sessions and bandwidth provisioning	Two ILP problems	Mininet
This Paper	Online & Offline	Maximize the total rate of the best effort flows and guarantee bandwidth of high priority flows	Dijkstra (online), ILP problem (offline)	Physical switches

A.3.1 Problem Description

Within an SDN network, there are different OpenFlow-enabled switches connected to one or more SDN controllers and a set of best-effort flows responsible for the correct routing of the network traffic. Each flow is described using a tuple `<source, destination, class>` whereby the class describes the traffic class of the flow. Each traffic class has a priority value and enforces the lower and upper bound bandwidth rates of the flows assigned to it. At a certain moment, emergency flows are requested for prioritizing network traffic coming from and going to different emergency services. These emergency flows need to be satisfied by guaranteeing the requested bandwidth while the remaining network bandwidth should be allocated to the best-effort flows. The optimization of the best-effort flows depends on the priority of the traffic classes where a higher priority requires a larger share of the available network bandwidth.

This paper answers to the question about how to maximize the total input rate of the best-effort flows in the network while the requested rates of the emergency flows are satisfied and the bandwidth capacity constraints of the network are respected. We assume that the requested rate for the emergency flows is not higher than the total available rate in the network. To solve this problem, two offline models are defined and evaluated.

In the first model, defined by means of an ILP formulation, we assume that flows cannot be split up and each flow needs to be assigned to a single path from source to destination. In the second model, defined by means of an LP formulation, we assume that flows can be split up, allowing traffic to be separated over different links, optimizing the bandwidth resource allocation [28]. The packet reordering effect that can occur when using flow splitting can be mitigated using hash-bashed splitting and packet tagging [29].

The formulations of these two offline models are provided in Section A.3.2 and Section A.3.3. Notations used in the formulations are summarized in Table A.2. Some described constraints contain a multiplication of a continuous and a binary variable and because this cannot be directly solved by state-of-the-art solvers, they need to be linearized first. These formulations will optimize the best-effort flows over the remaining bandwidth that is not used by emergency flows. In case the offline models are not able to run in real-time, the online approach manages new incoming flows in between offline batches, providing the shortest (but possible sub-optimal) path with a greedy-based solution to allocate bandwidth to these new flows.

Table A.2: Notations Summary

Variables	
$y^i_{u,v}$	Equals 1 if the traffic for flow i passes through link (u, v)
R^i	The rate assigned to best effort flow i
Parameters	
$F \equiv M \cup B$	Set of all flows
M	Set of all emergency flows
B	Set of all best effort flows
$G = (V, E)$	The graph of the network. V is the set of nodes and E is the set of physical links. All links are bidirectional with different capacity in each direction
$Z^i_{u,v}$	The rate of flow i on link (u, v)
$Cap^{(u,v)}$	The bandwidth of the link between u and v, in the direction from u to v
W_i	The weight assigned to flow i based on the traffic class it belongs to
τ_i	The requested rate for emergency flow i
$minRate^i$ $maxRate^i$	The lower bound and upper bound for the rate of flow i based on the traffic class it belongs to.
$Source\,(i)$ $Destination\,(i)$	The source and the destination of flow i

A.3.2 The ILP formulation

$$\max \sum_{i \in B | \{u = Source(i), (u,v) \in E\}} W_i \times Z^i_{u,v} \tag{A.1}$$

Subject to:

$$\sum_{(u,v)\in E} y^i_{u,v} - \sum_{(v,u)\in E} y^i_{v,u} = \begin{cases} 1 & u = Source\,(i) \\ 0 & \text{otherwise} \\ \text{-1} & u = Destination\,(i) \end{cases} \quad \forall u \in V,\ i \in F \tag{A.2}$$

$$\sum_{i \in B} y^i_{u,v} \times R^i + \sum_{i \in M} y^i_{u,v} \times \tau_i \leqslant Cap^{(u,v)} \tag{A.3}$$

$$\sum_{(u,v)\in E} y^i_{u,v} \leqslant 1 \ \ \forall u \in V,\ i \in F \tag{A.4}$$

$$\sum_{(v,u)\in E} y^i_{v,u} \leqslant 1 \ \ \forall u \in V,\ i \in F \tag{A.5}$$

$$R^i \in \left[minRate^i,\ maxRate^i\right] \tag{A.6}$$

$$y^i_{u,v} \in \{0, 1\} \tag{A.7}$$

$$Cap\,(u, v) \geqslant \tau_i \ \ \forall i \in M \tag{A.8}$$

The objective of the ILP formulation is to maximize the sum of traffic rates of best-effort flows multiplied by their assigned weights, illustrated in (A.1), subjected to constraints [(A.2) - (A.5)]. (A.2) is the flow conservation constraint which guarantees a path from source to destination. (A.3) enforces the capacity limit of each physical link and (A.4) and (A.5) are used to prevent loops as much as possible. (A.6) and (A.7) specify the bounds for the assigned rate and whether or not traffic is passing through link `(u, v)`. Finally in (A.8), the assumption is made that the network is at least able to handle all requested emergency flows.
The second constraint contains a multiplication of a continuous and a binary variable as in $\sum_{i \in B} y^i_{u,v} \times R^i$. The constraint can be linearized as follows:

$$Z^i_{u,v} \leqslant Cap^{(u,v)} \times y^i_{u,v} \tag{A.9}$$

$$Z^i_{u,v} \leqslant R^i \tag{A.10}$$

$$R^i + Cap\,(u, v) \times y^i_{u,v} - Z^i_{u,v} \leqslant Cap^{(u,v)} \tag{A.11}$$

$$Z^i_{u,v} \in \left[0,\ \ maxRate^i\right] \tag{A.12}$$

A.3.3 The LP formulation

$$\max \sum_{i \in B} W_i \times R^i \tag{A.13}$$

Subject to:

$$\sum_{(u,v)\in E} y^i_{u,v} - \sum_{(v,u)\in E} y^i_{v,u} = \begin{cases} -R^i & u = Source\,(i) \\ 0 & \text{otherwise} \\ -R^i & u = Destination\,(i) \end{cases} \quad \forall u \in V,\ i \in B \tag{A.14}$$

$$\sum_{(u,v)\in E} y^i_{u,v} - \sum_{(v,u)\in E} y^i_{v,u} = \begin{cases} \tau & u = Source\,(i) \\ 0 & \text{otherwise} \\ -\tau_i & u = Destination\,(i) \end{cases} \quad \forall u \in V,\ i \in M \tag{A.15}$$

$$\sum_{i \in F} y^i_{u,v} \leqslant Cap^{(u,v)} \tag{A.16}$$

$$R^i \in \left[minRate^i,\ maxRate^i\right] \tag{A.17}$$

$$y^i_{u,v} \in \mathbb{R}_{\geqslant 0} \tag{A.18}$$

$$Cap\,(u,v) \geqslant \tau_i \quad \forall i \in M \tag{A.19}$$

The LP formulation uses the principles of flow splitting to solve the described problem. The objective is again to maximize the sum of the traffic rates of the best-effort flows multiplied by their assigned weights, illustrated in (A.13), subjected to constraints [(A.14) - (A.16)]. (A.14) and (A.15) are the flow conservation constraints for the best effort and emergency flows respectively. (A.16) enforces the bandwidth capacity limits of physical links. The LP formulation is solvable in polynomial time [30, 31].

A.3.4 Online approach

As shown later in Section A.4, the LP model is much faster than the ILP model and could be used in a near-real-time scenario. However, the use of the slower ILP model is obliged if the network does not support flow splitting, which is for example the case when it is built with SDN switches from Northbound Networks [6]. In this case, an alternate online approach is necessary for handling new incoming flows in between calculations of the ILP model.
The offline batches are responsible for guaranteeing the emergency flows and optimizing the remaining best-effort flows. In case a new flow arrives

during the calculation of the offline batches, the shortest path between the source and destination is determined by using Dijkstra's algorithm [32]. New incoming emergency flows obtain their requested bandwidth while newly arriving best-effort flows are assigned to the average best-effort traffic class. In case there is no bandwidth available for the new flow, the greedy heuristic calculates the bandwidth and decreases the other best-effort flows bandwidth based on their priority, as illustrated in Algorithm 3. The complete online approach, following a greedy approach, is described in Algorithm 4.

Algorithm 3 Greedy heuristic

```
τ ← requested bandwidth
C ← traffic classes sorted by priority (low to high)
for i in count(C) − 1 do
    a ← τ/2
    τ ← τ/2
    band[i] ← a
end for
band[count(C) − 1] ← τ
for each traffic_class in C do
    i ← index
    s ← number of meters in traffic_class
    for each meter in traffic_class do
        meter ← meter − band[i]/s
    end for
end for
```

A.4 Implementation, Simulation and evaluation

In this section, the solution described in Section A.3.1 is implemented based on the two offline formulations summarized in Section A.3.2 and A.3.3 and evaluated in a simulation environment. Afterwards, the online approach described in Section A.3.4 is implemented and evaluated on a practical environment consisting of Zodiac SDN switches [6].

A.4.1 Simulation Environment

The proposed offline models are first validated using simulations. The evaluated topology, as shown in Figure A.2, consists of 16 ingress/egress points of traffic and 32 switches. Switches 16-30 are backbone switches and the backbone network has the same topology as the Internet2 network [33]. This topology is used to simulate a provider network catering to about

Algorithm 4 Online approach

```
R ← average best-effort traffic rate
B ← best-effort flows
while batch is running do
    X ← new incoming flow
    if X is emergency then
        τ ← requested bandwidth by X
    else
        τ ← R
    end if
    if τ is not available then
        apply_greedy_heuristic()
    end if
    apply_flows()
end while
run_batch()
```

Table A.3: Specification of the network scenario

Source/Destination of All Flows	0 - 15
Backbone Network (40 Gbps)	16 - 30
Source/Destination Emergency Flows	{0, 2, 3, 5, 7, 8, 11, 13, 14}
DSL Network	39 - 46
Mobile Network	31 - 38

2050 flows. Switches 31-38 are mobile base stations and the ingress/egress points attached to them represent mobile users. Switches 39-46 are DSL switches and the ingress/egress points attached to them represent DSL users. The bandwidth of each the backbone network link is 40 Gbps bidirectional. The specifications of the network scenario are summarized in Table A.3. IBM ILOG CPLEX [34] v12.7 is used to implement the models and the simulations are executed on a server with 2 Xeon E5-2690 v4 CPUs operating at 2.6GHz with 16GB of memory.

We defined 3 traffic classes with ranges [0, 25000], [0, 10000] and [0, 5000] Kbps with priorities of 100, 50 and 10, respectively for best effort flows. Moreover, the requested rate of each emergency flow was randomly chosen from set $\{25000, 10000, 5000\}$ Kbps because of the variation in types of emergency network traffic. Each best effort flow was randomly assigned to a class. Each evaluation result is the average of 30 simulation runs.

Figure A.2 The simulation topology based on the Internet2 network.

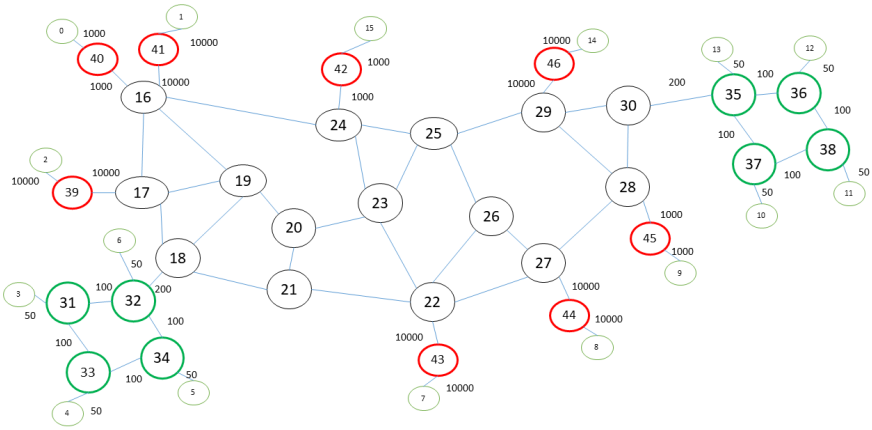


Table A.4: Comparison of the LP and ILP model simulation results

	LP Model	ILP Model
Solving Time-Before(ms)	484	18408
Solving Time-After(ms)	484	15210

A.4.2 Simulation Evaluation - Results

The performance of the two models is compared in Figure A.3. By increasing the number of best effort flows, the solving time increases in both models. However, the increase rate of the ILP model is exponentially higher than for the LP model. For 2000 best effort flows along with 50 emergency flows, the ILP model solves the problem in almost two minutes. It is worthwhile to mention that the solving time of the ILP model can further be decreased by up to one order of magnitude when using acceleration methods such as the novel algorithm based on the Benders decomposition method as described in [35].

To investigate the operational details of the models, we first generated 500 best effort flows and solved both the ILP and LP models. After that, we added 50 emergency flows and solved the problems again. Both models reported the same optimal values before and after adding the emergency flows which means that the same result is achieved by both models and the LP model solved the problem 30 times faster than the ILP model. After adding the emergency flows, the models decreased the rate of best effort flows to allocate the requested bandwidth of the emergency flows which resulted in a lower optimal values. A summary of the results is shown in

Figure A.3 The solving time of the ILP and LP models. Standard deviations are shown in the form of error bars

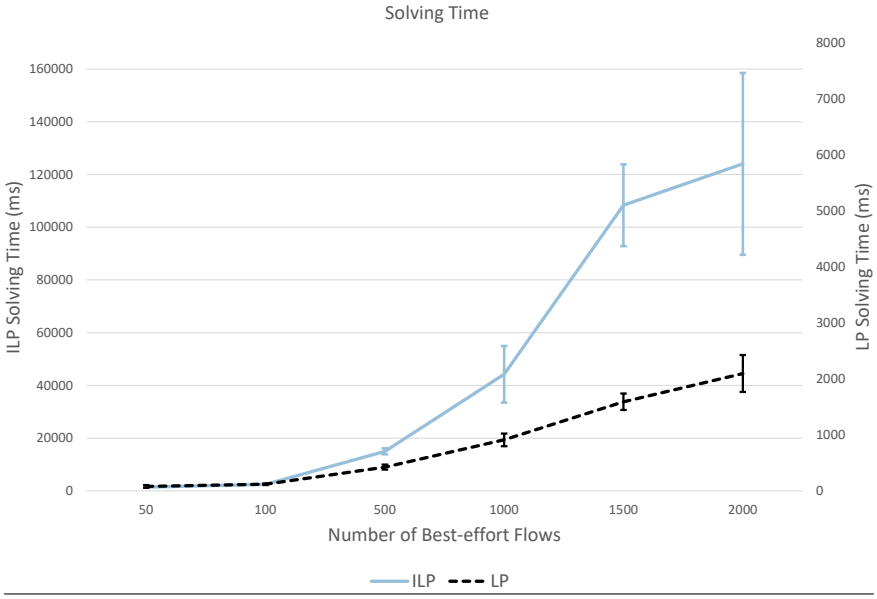
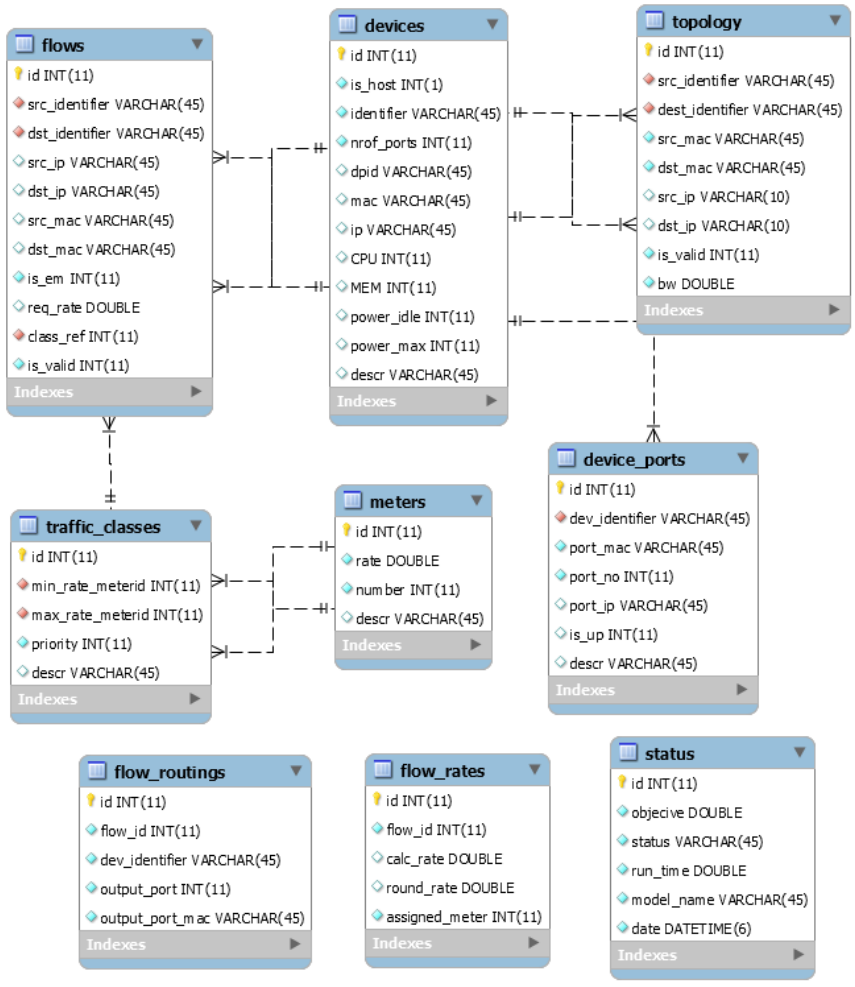


Table A.4.

A.4.3 Prototype Implementation

To implement the proposed solution, a network consisting of Zodiac SDN switches and a Ryu [7] SDN controller with 4 Intel Core i5-7440HQ running at 2.80GHz and 16 GB of memory is used. As discussed in Section A.3.4, the Zodiac SDN switches do currently not support flow splitting and thus the use of the slower ILP model is obliged in this environment. A database using the schema presented in Figure A.4 is used to store the current active flows and the information about the topology. The topology used for the evaluation is visualized in Figure A.5 and built with 1 Zodiac GX switch (sw1), 8 Zodiac FX switches (sw2 - sw9) and 10 raspberry pi's (d1 - d10). The Zodiac GX has an uplink of 1 gbps while the Zodiax FX switches have an uplink of 100 mbps. The used traffic classes are illustrated in Table A.5 and the requested bandwidth rates based on destination are summarized in Table A.6. OpenFlow v1.3 meters were used to specify the upper bound and lower bound rates of each traffic class and the ILP model is implemented with IBM ILOG CPLEX v12.7. Note that with the provided meters in this prototype, the ILP will rather assign the meter with 0 kbps bandwidth to flows with a lower priority in case there is a shortage. When more meters

Figure A.4 Topology of the database used by the prototype implementation. The tables devices, device_ports, meters, topology and traffic_classes are filled based on the practical environment. The tables flows, flow_rates and flow_routings contain the optimized best-effort and emergency flows after solving the ILP formulation.



per traffic class are allocated, a downgrade is possible, but this is currently not implemented.

Assume $\{ m_1, m_2, \ldots, m_n\}$ are n defined meter rates and $m_i \leqslant m_{i+1}\ \forall i$. The weight of best effort flow i is calculated by $\left\lceil P_i \times \frac{m_n}{m_1+1} \right\rceil$ in which P_i is the priority of the class that the flow belongs to and $\lceil x \rceil$ is the ceiling

Figure A.5 Topology of the prototype environment. Sw1 is a Zodiac GX switch, sw2 - sw9 are Zodiac FX switches and d1 - d10 are raspberry pi's.

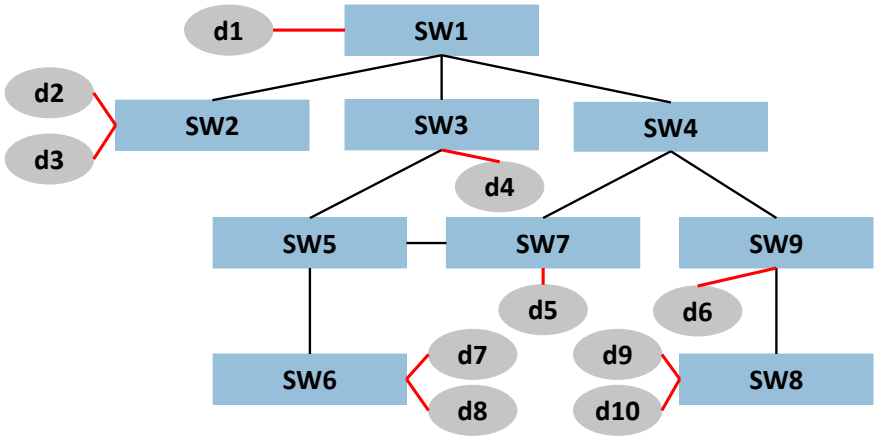


Table A.5: Traffic classes (all in kbps)

Id	Name	Minimum Rate	Maximum Rate
1	High Priority	0	25000
2	Normal Priority	0	10000
3	Low Priority	0	5000

function.

When a new flow arrives, it is added to the database. When the previous batch of the offline calculations is finished, the controller runs the ILP solver. The implementation reads the database information, solves the ILP problem and stores the results in the database. The output of the ILP is the assignment of each flow to one meter and the routing of flows over the network. To assign a flow to a meter, whether it be best effort flows or emergency flows, the implementation rounds down the calculated rate to the nearest defined meter rate. Based on the simulation results summarized in Section A.4.2, the ILP model provides optimal results with a high number of flows but not in real-time. To combat this, we run the offline model consecutively while the online approach is used to route and to apply the corresponding meter to new incoming flows. Best-effort flows will be assigned to the average meter with 10000 kpbs while emergency traffic will be assigned to their requested rate. To decrease the impact on the current best-effort and emergency flows, a greedy heuristic is applied to reassign available bandwidth from other best-effort flows, based on their

Table A.6: Requested rates per flow based on the destination. Rates between 0 and 4999 kbps are part of traffic class 3, rates between 5000 and 9999 kpbs are part of traffic class 2 and rates higher dan 10000kpbs are part of traffic class 1.

Destination	Traffic class
d1	3
d2	3
d3	3
d4	3
d5	3
d6	2
d7	2
d8	2
d9	2
d10	1

priority.

A.4.4 Prototype Evaluation - Example scenario

The prototype is evaluated to study the behavior of the online approach and the findings are illustrated in Figure A.6. When the batch calculation is running, the online approach will handle the new incoming flows. The flow responsible for the traffic going from d1 to d5, which is part of traffic class 3, is already allocated together with 87 other flows. Next at time t1, a new incoming emergency flow going from d1 to d10 is added to the network, with a requested bandwidth of 25,000 kbps. Because of its priority, the requested bandwidth is allocated and the greedy heuristic reduced the bandwidth from the other best-effort flows. The flows part of traffic class 3 have an average decrease of 284 kbps. Afterwards at time t2, a flow going from d8 to d10 is added, which is part of traffic class 1. As this is a best-effort flow, the average best-effort meter with a bandwidth of 10,000 kbps is allocated. The greedy heuristic again determines the bandwidth for each best-effort flow without impacting the current emergency flows. Finally, the batch calculations (visualized by the gray vertical line at time t3 in Figure A.6) optimizes the flows of the whole network again.
It is clear that the online approach is guaranteeing the bandwidth of the emergency flows and creates a sub-optimal solution for the new incoming

Figure A.6 Throughput of 2 best-effort flows and 1 one emergency flow. At time t1, the emergency flow is added and assigned by the online approach. At time t2, another best-effort flow is added and assigned by the online approach. At time t3, the offline batch has calculated and applied the optimal solution.

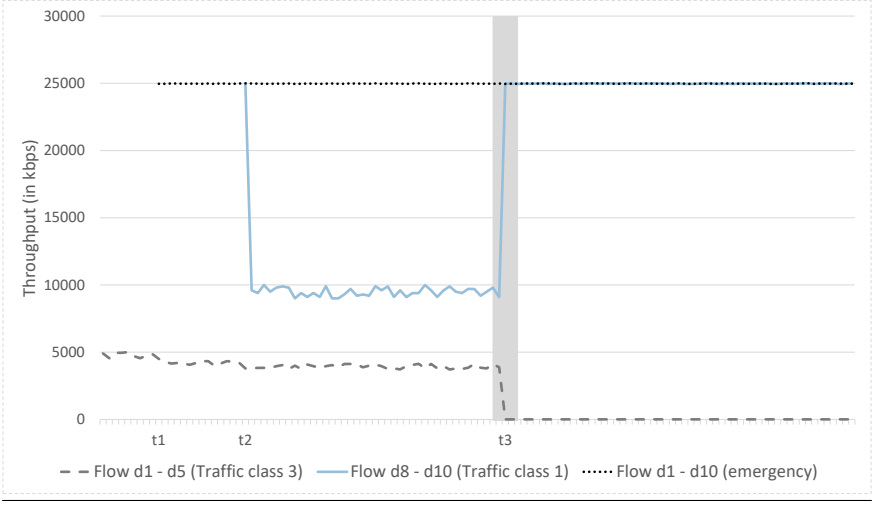


Table A.7: The ILP model results

	Before	After
Solving Time (ms)	20520.234	17489.531

best-effort flows. The sub-optimal solution has 2.76% difference per flow compared to the result of the offline batches in the whole example scenario. In some cases, this difference is 100% because the online approach does not drop any new incoming flows, while the offline batches can decide to drop a flow much faster as explained in Section A.4.3. Afterwards, the batch calculations optimizes the best-effort flows over the remaining available bandwidth not used by emergency flows. The solving time of the batch calculations before and after adding 50 emergency flows is illustrated in Table A.7 and shows that the proposed offline model can solve small-sized networks efficiently.

A.5 Conclusion

Emergency network traffic needs to have priority over best-effort traffic during emergency situations. With the expected release of 5G, slicing

concepts at network level will enable prioritization of the emergency network traffic over mobile connections. In addition, SDN principles allow to assign different QoS levels to different network slices.

In this paper, we therefore propose an SDN-based solution whereby both LP and ILP theoretical mathematical models guarantee the requested rate of emergency flows and maximize the best-effort flows over the remaining available bandwidth. Due to the calculation time of these models, an online approach handles new incoming flows in between these calculations. The shortest path, based on Dijkstra's algorithm, is calculated and a greedy heuristic is applied to obtain bandwidth from the other best-effort flows. When the new incoming flow is an emergency flow, the requested bandwidth will be allocated by any means, a new incoming best effort flow will be allocated with the average bandwidth of all the active best-effort flows.

The two offline models are first evaluated by simulations and the results show that both with the ILP and LP mathematical problems can be used whereby the ILP model exhibiting plus-second execution time while the LP model works 30 times faster for 500 best-effort flows and 50 emergency flows. Afterwards, the batch calculations together with the online solution are prototyped and evaluated on an SDN network consisting of Zodiac Switches and Raspberry pi's. The Zodiac switches do not support flow splitting, so the use of the slower ILP model is obliged. The practical evaluation shows that the online problem efficiently handles new incoming flows while guaranteeing the bandwidth for all the emergency flows and providing a sub-optimal temporary solution for the best-effort flows.

Practical evaluation of the faster LP model together with an extended evaluation of the proposed models is envisaged as future work.

References

[1] Caitlin McGarry. The Truth About 5G: What's Coming (and What's Not) in 2019, 2019. URL https://www.tomsguide.com/us/5g-release-date,review-5063.html.

[2] Haijun Zhang, Na Liu, Xiaoli Chu, Keping Long, Abdol Hamid Aghvami, and Victor C.M. Leung. Network Slicing Based 5G and Future Mobile Networks: Mobility, Resource Management, and Challenges. *IEEE Communications Magazine*, 55(8):138–145, 2017. ISSN 01636804. doi: 10.1109/MCOM.2017.1600940.

[3] Jose Ordonez-Lucena, Pablo Ameigeiras, Diego Lopez, Juan J Ramos-Munoz, Javier Lorca, and Jesus Folgueira. Network slicing for 5G

with SDN/NFV: Concepts, architectures, and challenges. *IEEE Communications Magazine*, 55(5):80–87, 2017.

[4] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2): 114–119, 2013. ISSN 01636804. doi: 10.1109/MCOM.2013.6461195.

[5] Hassan Hawilo, Abdallah Shami, Maysam Mirahmadi, and Rasool Asal. NFV: State of the art, challenges, and implementation in next generation mobile networks (vEPC). *IEEE Network*, 28(6):18–26, 2014. ISSN 08908044. doi: 10.1109/MNET.2014.6963800.

[6] Paul Zanna. Zodiac FX, 2019. URL https://northboundnetworks.com/collections/zodiac-fx/products/zodiac-fx.

[7] FUJITA Tomonori. Introduction to ryu sdn framework. *Open Networking Summit*, 2013.

[8] Murat Karakus and Arjan Durresi. Quality of service (QoS) in software defined networking (SDN): A survey. *Journal of Network and Computer Applications*, 80:200–218, 2017. ISSN 1084-8045.

[9] J. Yan, H. Zhang, Q. Shuai, B. Liu, and X. Guo. HiQoS: An SDN-based multipath QoS solution. *China Communications*, 12(5):123–133, 2015. ISSN 1673-5447.

[10] Y. Zhang, Y. Tang, D. Tang, and W. Wang. QOF: QoS framework based on OpenFlow. In *2015 2nd International Conference on Information Science and Control Engineering*, pages 380–387, 2015.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959. ISSN 0029-599X.

[12] A. V. Akella and K. Xiong. Quality of service (QoS)-guaranteed network resource allocation via software defined networking (SDN). In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 7–13, 2014.

[13] S. Cao, M. Tong, Z. Lv, and D. Jiang. A study on application-towards bandwidth guarantee based on SDN. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, 2016.

[14] S. Tomovic and I. Radusinovic. Fast and efficient bandwidth-delay constrained routing algorithm for SDN networks. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 303–311, 2016.

[15] P. Pinto, R. Cardoso, P. Amaral, and L. Bernardo. Lightweight admission control and traffic management with SDN. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–7, 2016.

[16] H. Krishna, N. L. M. van Adrichem, and F. A. Kuipers. Providing bandwidth guarantees with OpenFlow. In *2016 Symposium on Communications and Vehicular Technologies (SCVT)*, pages 1–6, 2016.

[17] C. Morin, G. Texier, and C. Phan. On demand QoS with a SDN traffic engineering management (STEM) module. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–6, 2017.

[18] You Lu, Baochuan Fu, Xuefeng Xi, Zhancheng Zhang, and Hongjie Wu. An SDN-based flow control mechanism for guaranteeing QoS and maximizing throughput. *Wireless Pers Commun*, 97(1):417–442, 2017. ISSN 1572-834X.

[19] A. Samani and M. Wang. MaxStream: SDN-based flow maximization for video streaming with QoS enhancement. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pages 287–290, 2018.

[20] Northbound networks, . URL https://northboundnetworks.com/.

[21] OpenFlow Switch Specifications. 1.5. 1. *Open Networking Foundation*, 3, 2015.

[22] Understanding CoS Two-Color Marking - TechLibrary - Juniper Networks, Jun 2019. URL https://www.juniper.net/documentation/en_US/junos/topics/concept/cos-ex-series-two-color-marking-understanding.html. [Online; accessed 5. Aug. 2019].

[23] Stephen Haddock. Frame metering in 802.1 q, 2013.

[24] GENI, . URL https://www.geni.net/.

[25] Mininet overview - mininet, . URL http://mininet.org/overview/.

[26] Open vSwitch, . URL https://www.openvswitch.org/.

[27] P. Van Mieghem and F. A. Kuipers. Concepts of exact QoS routing algorithms. *IEEE/ACM Transactions on Networking*, 12(5):851–864, 2004. ISSN 1063-6692.

[28] Daphne Tuncer, Marinos Charalambides, Stuart Clayman, and George Pavlou. Flexible Traffic Splitting in OpenFlow Networks. *IEEE Transactions on Network and Service Management*, 13(3):407–420, 2016. ISSN 19324537. doi: 10.1109/TNSM.2016.2580666.

[29] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2): 17–29, 2008.

[30] L. G. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53–72, 1980. ISSN 00415553. doi: 10.1016/0041-5553(80)90061-0.

[31] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[32] E. W. Dijkstra. A Note on Two Probles in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0029-599X. doi: 10.1007/BF01386390.

[33] Internet2 Network Infrastructure Topology, 2018. URL https://www.internet2.edu/media/medialibrary/2019/04/10/I2-Network-Infrastructure-Topology-All-legendtitle.pdf.

[34] IBM ILOG CPLEX Optimization Studio. URL https://www.ibm.com/be-en/marketplace/ibm-ilog-cplex.

[35] Behrooz Farkiani, Bahador Bakhshi, and S. Ali MirHassani. Stochastic virtual network embedding via accelerated benders decomposition. 94: 199–213, 2019. ISSN 0167-739X.