FACULTY OF ENGINEERING AND ARCHITECTURE

Scalable and Flexible Low-Resource Service Orchestration for Enabling Smart Cities

Tom Goethals

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

May 2022

GHENT UNIVERSITY

ISBN 978-94-6355-588-3
NUR 986
Wettelijk depot: D/2022/10.500/29

Members of the Examination Board

Chair

Prof. Hennie De Schepper, PhD, Ghent University

Other members entitled to vote

Prof. Peter Hellinckx, PhD, Universiteit Antwerpen
Prof. Jeroen Hoebeke, PhD, Ghent University
Philip Leroux, PhD, Ghent University
Eddy Truyen, PhD, KU Leuven

Supervisors

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

Preface

Reasonable Excuse - Culture GCU, Iain M. Banks

My career path has always been anything but a straight line, going from government jobs to the private sector and back again as I accumulated experience and education degrees. Back in 2017, I worked in an industry job in middleware development and integration, when I started to notice that the work was getting very repetitive, despite good colleagues and nice working conditions. Having worn out the focus on concrete, practical IT work that defined the first decade of my career, my increasing penchant for theoretical work and research finally made a PhD the next logical step.
The topic of Smart Cities is a surprisingly wide research field, with a plethora of subjects whose technical depth and complexity are difficult to imagine before getting into them. When starting my PhD, I was quickly disabused of the notion that the Internet of Things was all party trick app-controlled light bulbs and smart TV's. What was available was an endless supply of immensely interesting topics at the crossroads of technologies such as networking, Artificial Intelligence, hardware advances and service architectures. Futhermore, for each such combination the spectrum ranges from examination and integration of existing software to form innovative solutions, to creating novel algorithms for problems not yet (entirely) solved. Starting with a few contributions in the first category, my work quickly veered towards the second one, with a particular interest in decentralization and extremely large scale orchestration of software services, lured in by the then upcoming adoption of advanced Artificial Intelligence, a topic which has always interested but somehow eluded me.
Special thanks go out to a great many people. First of all, my promotors; on one hand Filip De Turck for his feedback on my work, invaluable guidance in the world of academics, and for training my nearly nonexistent diplomatic skills throughout publishing procedures. On the other hand, Bruno Volckaert for additional feedback on my work, for more hands-on guidance and smoothing over tedious procedures, and for piling lab sessions on me, as teaching in itself turns out to be a useful learning experience. Also a bunch of his dirty jokes. Next up are the co-workers in my office for interesting lunch-break conversations, at least during the times we weren't stuck at

home because of a pandemic. The topics during lunch-break ranged across all of science and society, and I definitely learned some useful things during my PhD that were entirely unrelated to Smart Cities. Some of my co-workers that deserve specific thanks; Ankita Atrey for feedback and improvements on my early papers, and her co-authorship on some of my best received work. Sarah Kerkhove for co-operation on several research topics related to virtualization and co-authorship of several publications, winning a best paper award in the process. Laurens Van Hoye for his useful and (thankfully) lenient feedback, and finally Merlijn Sebrechts, for managing to disapprove of three quarters of my second paper, which led to a great improvement of my narrative skills and writing style.

Although my work was not very project-oriented, there were some collaborations with companies and other universities. As such, I would like to thank Erik Van Mossevelde and Geert Goemaere from Niko for their collaboration during the Watchdog project, which resulted in several publications. For the FlexNet project, I would mainly like to thank Esteban Municio from UAntwerpen and Nico Janssens from Rombit for their close collaboration, resulting in a demo of my main work, in addition to the dozens of other people working on this sizeable project over the years.

Last but not least, some thanks goes out to family and friends, who always managed to be interested in my work, at least until some explanation of math or theoretical model made them go cross-eyed. Final thanks goes out to my dog Vito, who I suspect never understood much of my mumbled ranting, but whose backyard-bound shenanigans at least made sure I got enough daily movement at home during the pandemic.

During my PhD, I consciously avoided relying on specific frameworks, except some popular ones, partially because their lifecycles and features tend to ebb and flow like the tide, and partially because my brain stores names in /dev/null, making it difficult to remember any frameworks. Additionally, theoretical, framework-agnostic work seems more likely to be relevant in many applications, and a suitable candidate for further improvement.

Instead of iterating my publications verbatim, framed by minor comments, I chose to adapt and integrate them, updating them to some degree, and combining sections where possible to create a narrative flow suitable for a larger work. Still, old results are not so easily updated without repeating all of the work that led to them, and since Smart Cities present a rapidly changing landscape, I request that the reader consider the detriment of scientific progress on my results, and reads them in the context of the versions used, and any improvements they may have led to, rather than comparing them to the latest advances at hand. Finally, any related work sections have not been (extensively) rewritten, as they contain useful work directly related to each publication. As a result, certain concepts may be cursorily mentioned in these sections that are not properly explained until later chapters.

As for my wandering around before finally doing a PhD at a later age;

"I came to my truth by diverse paths and in diverse ways:
it was not upon a single ladder that I climbed to the height
where my eyes survey my distances.
And I have asked the way only unwillingly - that has always
offended my taste! I have rather questioned and attempted
the ways themselves.
All my progress has been an attempting and a questioning -
and truly, one has to *learn* how to answer such questioning!
That however - is to my taste:
Not good taste, not bad taste, but *my* taste, which I no
longer conceal and of which I am no longer ashamed.
'This - is now *my* way: where is yours?' Thus I answered
those who asked me 'the way'. For *the* way - does not exist!

Thus spoke Zarathustra." - *Friedrich Nietzsche*

Gent, february 2022
Tom Goethals

Table of Contents

List of Figures

List of Tables

List of Acronyms

A

AES-NI	Advanced Encryption Standard New Instructions
AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
ANN	Artificial Neural Network
API	Application Programming Interface

C

CCTV	Closed-Circuit Television
CI/CD	Continuous Integration/Continuous Deployment
CNI	Container Network Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRD	Custom Resource Definition
CUDA	Compute Unified Device Architecture

D

DAG	Directed Acyclic Graph
DinD	Docker-in-Docker
DNN	Deep Neural Network
DNS	Domain Name System
DRL	Deep Reinforcement Learning

E

EI	Edge Intelligence

F

FL	Federated Learning
FPS	Frames Per Second
FPGA	Field Programmable Grid Array
FUSE	Flexible federated Unified Service Environment

G

GA	Genetic Algorithm
GDPR	General Data Protection Regulation
GRU	Gated Recurrent Units

H

HTTP	HyperText Transfer Protocol

I

IIoT	Industrial Internet of Things
IoMT	Internet of Medical Things
IoV	Internet of Vehicles
IoT	Internet of Things

J

JSON	JavaScript Object Notation

K

KVM	Kernel-based Virtual Machine

L

LAN	Local Area Network
LSTM	Long Short Term Storage

M

MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
MQTT	Message Queue Telemetry Transport

N

NAT	Network Address Translation
NFV	Network Function Virtualization
NPU	Neural Processing Unit
NSGA-II	Non-dominated Sorting Genetic Algorithm II

O

OCI	Open Container Initiative
OML	Open Modeling Language
OS	Operating System

P

PIE	Position Independent Executable

PSO	Particle Swarm Optimization
PSS	Proportional Set Size
PU	Processing Unit

Q

QoS	Quality of Service

R

RAM	Random Access Memory
REST	Representational State Transfer
RL	Reinforcement Learning
RNN	Recurrent Neural Network

S

SDN	Software Defined Networking
SI	Swarm Intelligence

T

TAP	Network Tap
TCP	Transmission Control Protocol
TPU	Tensor Processing Unit
TUN	Network Tunnel

U

UDP	User Datagram Protocol

V

VANet	Vehicular Ad-hoc Network
VLAN	Virtual LAN
VM	Virtual Machine
VPN	Virtual Private Network

W

WASI	WebAssembly System Interface
WASM	WebAssembly
WSL2	Windows Subsystem for Linux 2

Samenvatting
– Summary in Dutch –

Sinds de eeuwwisseling zijn computers in netwerken steeds kleiner en krachtiger geworden, terwijl computers worden ingebed in steeds meer "slimme" apparaten. In combinatie met een exponentieel groeiende instroom van sensorgegevens van deze apparaten, heeft de alomtegenwoordigheid van rekenkracht recent geleid tot het concept van "Slimme Steden" (Smart Cities), dat tot doel heeft slimme, transparante toepassingen in moderne steden mogelijk te maken en te benutten. Dergelijke toepassingen zijn meestal onderverdeeld in de domeinen van Smart Homes, Industry 4.0, Smart Health Care, Internet of Vehicles (of verkeersmanagement in het algemeen) en generieke Smart Cities-toepassingen. De software die deze innovaties aandrijft, werkt meestal in de mist ("fog networks") of de netwerkrand ("the network edge" of simpelweg "edge"); het laatste is het netwerk dat gebruikelijk alle sensoren en apparaten van eindgebruikers bevat, terwijl het eerste een intermediair netwerk is tussen clouddatacenters en de netwerkrand. Naast de verschillende Smart City-toepassingsdomeinen heeft de opkomst van de mist en de netwerkrand geresulteerd in bijkomende onderzoeksgebieden die betrekking hebben op de orchestratie van software en gegevens, en het management van netwerkverkeer in zulke netwerken.

Dit proefschrift behandelt onderzoek in de laatstgenoemde gebieden, meer bepaald softwaregestuurde netwerk- en computerinfrastructuur, efficiënt gebruik van systeembronnen, uniforme en betrouwbare software-omgevingen, en tot slot de integratie van Kunstmatige Intelligentie (KI), wat het "Smart" in Smart Cities mogelijk maakt. Het gepresenteerde onderzoek is in de eerste plaats bedoeld om software-orchestratie te verbeteren in plaats van concrete en nieuwe slimme applicaties te maken, hoewel het gemakkelijk kan worden geïntegreerd in software die van de ontwikkelde functies gebruik wil maken. Veel van de inhoud is afgeleid van eerdere publicaties van de auteur, hoewel ze enigszins zijn aangepast om beter aan te sluiten op het algemene onderwerp, en zijn bijgewerkt om de stand van zaken rond 2021 weer te geven.

De eigenlijke inhoud van dit proefschrift is verdeeld in drie subsecties die zijn afgeleid van de titel; "Efficiëntie en Flexibiliteit" (Resource Efficiency and Flexibility), "Schalen naar Slimme Steden" (Scaling Towards Smart Cities) en "Intelligentie in Slimme Steden" (Intelligence in Smart Cities).

De virtualisatie van softwareomgevingen en netwerken is een belangrijke factor voor “Efficiëntie en Flexibiliteit", omdat het tot doel heeft een uniforme omgeving te creëren met voorspelbaar gedrag voor applicaties, onafhankelijk van de hardware van het apparaat waar de applicatie uiteindelijk op draait. Er zijn verschillende technieken om softwareprocessen te virtualiseren, waaronder virtuele machines (VM), containers en unikernels. Terwijl VM's een volledig besturingssysteem virtualiseren, zijn containers en unikernels enkel gericht op het virtualiseren van een softwareproces en eventuele bibliotheken die nodig zijn voor het proces. Containers doen dit door de kernel van een hostbesturingssysteem te delen, terwijl unikernels een relatief recente methode zijn om minimale VM's te maken met een zeer kleine functionele en operationele software-overhead. In de afgelopen jaren zijn de hulpprogramma's die worden gebruikt om unikernels te maken, gegroeid van proof-of-concept tot platformen die zowel nieuwe als bestaande software kunnen draaien die in verschillende programmeertalen is geschreven. De performantie van unikernels en Docker-containers wordt bestudeerd in de context van REST-services en zware rekentaken geschreven in Java, Go en Python. De resultaten tonen aan dat de performantie van unikernels sterk afhankelijk is van de gebruikte programmeertaal, en dat multi-threading over het algemeen de performantie vermindert in plaats van te schalen. Single-threaded unikernels geschreven in Golang en Java zijn over het algemeen aanzienlijk sneller dan vergelijkbare containers.

Netwerkvirtualisatie kan worden gebruikt om Virtuele Private Netwerken (VPN) of overlay-netwerken (Software Defined Networks, SDN) tussen containers te creëren, en om verkeer transparant te routeren en te beveiligen met behulp van Network Function Virtualisation (NFV). Vanwege het toenemende gebruik van containers op steeds krachtigere apparaten in de netwerkrand is er behoefte aan betrouwbare en veilige netwerkverbindingen tussen deze apparaten.

De overgang van een beveiligd netwerk in datacenters naar heterogene openbare en particuliere netwerken brengt problemen met zich mee op het gebied van beveiliging en netwerktopologie die gedeeltelijk kunnen worden opgelost door een VPN te gebruiken om de netwerkrand met de cloud te verbinden. De schaalbaarheid van VPN software wordt geëvalueerd om te bepalen of en hoe een VPN kan worden gebruikt in grootschalige clusters met apparaten in de netwerkrand. Er worden benchmarks uitgevoerd om het maximale aantal VPN knooppunten en de invloed van netwerkdegradatie op een VPN te bepalen, voornamelijk met behulp van verkeer dat typisch is voor edge-apparaten die IoT-gegevens (vb. sensor data) genereren. De resultaten tonen aan dat WireGuard een goede keuze is als VPN software om honderden edge-apparaten in een cluster te verbinden, terwijl de alternatieven ofwel falen, ofwel onaanvaardbare hoge uitvalpercentages hebben, of beide.

Een ander aspect van “Efficiëntie en Flexibiliteit” is het creëren van een homogene omgeving voor software, over alle soorten apparaten die zich in de mist en de netwerkrand bevinden. Dit aspect is gericht op containers,

en hoe ze efficiënt georkestreerd en uitgevoerd kunnen worden op apparaten met een beperkte rekenkracht en geheugencapaciteit.
FLEDGE wordt voorgesteld als een Kubernetes-compatibele containerorkestrator op basis van virtuele Kubelets, voornamelijk gericht op containerorchestratie op edge-apparaten met beperkt geheugen. Verschillende aspecten van containerorchestratie worden onderzocht, zoals de keuze van de onderliggende containersoftware en het realiseren van geheugenefficiënte containernetwerken. Evaluaties worden uitgevoerd om te bepalen hoe FLEDGE zich verhoudt tot Kubernetes en K3S op het gebied van systeemvereisten. Verdere evaluaties, die eind 2021 zijn uitgevoerd, vergelijken FLEDGE ook met K0S, ioFog en KubeEdge, en tonen aan dat FLEDGE de meest geheugenefficiënte oplossing is, ten koste van het beperken van de functionaliteit tot edge-specifieke behoeften.
Terwijl cloud-orkestrators honderden tot (maximaal) duizenden apparaten organiseren, bevat de netwerkrand grootte-ordes meer apparaten, wat leidt tot de behoefte aan een zeer schaalbaar orchestratie-algoritme dat tevens het gebruik van systeembronnen op alle apparaten minimaliseert. Hiertoe wordt Swirly voorgesteld als een oplossing voor bijna realtime service-orchestratie in de netwerkrand. Met behulp van minimale agenten op elk apparaat, en rekening houdende met frequente veranderingen in netwerkomstandigheden en aangesloten apparaten, genereert Swirly in realtime optimale servicetopologieën op basis van gegevens die door apparaten worden gerapporteerd. De theoretische performantie wordt onderzocht, en er wordt een model van het gedrag en van de limieten van mistapparaten geconstrueerd. De evaluatie van het algoritme toont aan dat het in staat is om software-services voor honderdduizenden apparaten in bijna-realtime te beheren.
Voor het schalen van service-architecturen naar de grote netwerken van slimme steden zijn oplossingen in netwerk- en softwarebeheer nodig die betrouwbaar kunnen worden geschaald naar elke vereiste situatie. Verschillende schaalbaarheidsmechanismen worden onderzocht; zorgvuldig gebruik van systeembronnen en het overdragen van taken naar andere apparaten kunnen de lokale schaalbaarheid verbeteren (bijv. meer taken per apparaat), federatie combineert de bronnen van verschillende netwerken om specifieke doelen beter te bereiken, en decentralisatie zorgt voor zelforganisatie en (technisch) onbeperkte schaalbaarheid van service-architecturen.
In de context van federaties wordt Flexible federated Unified Service Environment (FUSE) voorgesteld als een oplossing voor het samenvoegen van meerdere private netwerken (vb. bedrijfsnetwerken) in een op microservices gebaseerde ad-hocfederatie, en voor het implementeren en beheren van software op de apparaten van die federatie. De belangrijkste use case van FUSE bestaat uit crisissituaties, waarbij het belangrijk is om snel informatie uit verschillende bronnen te kunnen verzamelen om een volledig en nauwkeurig beeld van de situatie te vormen (vb. videostreams, alarmen). Er wordt een videostreamingtoepassing geïmplementeerd om de performantie te demonstreren van het opzetten van een federatie en relevante softwa-

retoepassingen binnen een federatie. De resultaten tonen aan dat FUSE binnen enkele minuten kan worden gestart, en dat het meerdere videostreams kan ondersteunen onder normale netwerkomstandigheden, waardoor het een haalbare oplossing is voor het probleem van snelle en eenvoudige federatie van netwerken.
Een tweede toepassing van schaalbaarheid is de decentralisatie van Swirly naar Self-Organizing Swirly (SoSwirly). Evaluaties van Swirly tonen aan dat naarmate de topologie die het beheert groeit, het geheugengebruik en het netwerkverkeer onvermijdelijk de systeembronnen van een enkele gecentraliseerde instantie op één machine ontgroeien. Aangezien de mist een geografisch wijdverspreid computersubstraat is, is het eerder geschikt om een gedecentraliseerde service-orkestrator te gebruiken, geïnstalleerd op alle apparaten, waarbij elk apparaat services in de buurt kan ontdekken en gebruiken zonder de volledige servicetopologie te moeten kennen. SoSwirly schaalt service-instanties zoals vereist door de netwerkrand, op basis van beschikbare systeembronnen en flexibel gedefinieerde afstandsmetrieken. Het wiskundige model van mistnetwerken dat voor Swirly is ontwikkeld wordt uitgebreid, en er worden een theoretische analyse en een empirische evaluatie van de performantie voorgesteld, waaruit blijkt dat SoSwirly onder de meeste omstandigheden zeer schaalbaar is. Bovendien tonen de evaluaties aan dat de systeemvereisten van SoSwirly, gecombineerd met FLEDGE als container runtime, laag genoeg zijn om op een grote verscheidenheid aan edge-apparaten te draaien.
Tenslotte worden in het deel rond "Intelligentie in Slimme Steden" de verschillende soorten KI besproken die worden gebruikt in edge-netwerken, evenals hoe ze intelligent beheer van systeembronnen mogelijk maken en tot nieuwe, intelligente gebruikerstoepassingen kunnen leiden.
SoSwirly wordt geëvalueerd in de context van het gedecentraliseerd leren van kunstmatige neurale netwerken, wat de basis vormt voor toekomstig werk om SoSwirly te gebruiken als een gedistribueerd updatemechanisme en om een voorspellende, op KI gebaseerde component in SoSwirly te integreren die actief de beste serviceprovider kiest voor elk apparaat, in plaats van te vertrouwen op reactieve aanpassingen d.m.v. gedetecteerde veranderingen in de topologie.
Tenslotte wordt een overzicht gegeven van de stand van zaken m.b.t. KI in de netwerkrand. Er wordt een taxonomie voorgesteld met "Hulptechnologie" (Enabling Technology) voor Edge Intelligence, "Organisatie" van de netwerkrand met KI, en KI "Applicaties" in de netwerkrand als hoofdonderwerpen. Onderzoekstrends van 2015 tot 2020 worden opgesteld voor verschillende onderdelen van elk van de hoofdonderwerpen, waaruit een exponentiële toename van zowel absolute als relatieve onderzoeksinteresse voor elk onderdeel blijkt. Het aspect "Organisatie", wat de eigenlijke focus van dit overzicht is, heeft een meer fijnmazige onderverdeling waarbij alle bijdragende factoren in detail worden bekeken. Voor elk onderdeel van de taxonomie worden een aantal publicaties van 2019 tot 2021 geselecteerd om een strategisch beeld

te vormen van de stand van zaken van Edge Intelligence. Op basis van deze geselecteerde onderzoeken en de trendgegevens worden een aantal kortetermijnuitdagingen en visies op langere termijn geformuleerd m.b.t. Edge Intelligence, die een basis vormen voor toekomstig werk.
Samenvattend behandelt dit proefschrift concrete bijdragen in verschillende deelgebieden van slimme steden, eindigend met een uitgebreide bespreking van hoe het "slimme" aspect mogelijk gemaakt wordt en benut wordt, en hoe toekomstig werk kan resulteren in verbeterde intelligentie en schaalbaarheid van netwerken en applicaties.

Summary

Since the turn of the century, networked devices have become increasingly smaller and more powerful, while computational capabilities have become embedded in ever more "smart" devices. Combined with an exponentially growing influx of sensor data from these devices, the ubiquity of computing has recently resulted in the concept of "Smart Cities", which aims to enable and leverage smart, transparent applications in modern cities. Such applications are usually divided into the domains of Smart Homes, Industry 4.0, Smart Health Care, Internet of Vehicles (or traffic management in general), and generic Smart Cities applications. The software that powers these innovations usually runs in the fog or the network edge; the latter is the network containing all end-user devices and sensors, while the former is an intermediate network between cloud data centers and the network edge. In addition to the various Smart City application domains, the rise of the fog and network edge have led to dedicated research fields which concern the orchestration of software and data, and network traffic management.
This thesis covers research in the latter fields, more specifically software defined networks and computational infrastructure, resource efficiency, uniform and reliable software environments, and lastly the integration of Artificial Intelligence, which puts the "Smart" in Smart Cities. The research presented is primarily aimed at improving software orchestration, rather than creating concrete and novel smart applications, although it can be easily integrated into software. Much of the content is derived from earlier publications by the author, although they have been adapted to better mesh with the general topic, and updated to reflect the state of the art circa 2021. The actual content of this thesis is divided into three subsections derived from its title; "Resource Efficiency and Flexibility", "Scaling Towards Smart Cities" and "Intelligence in Smart Cities".
The virtualization of software environments and networks is an important factor for "Resource Efficiency and Flexibility", as it aims to create a uniform environment with predictable behavior, independent of device hardware. Software processes can be virtualized through various approaches, including Virtual Machines (VM), containers, and unikernels. Whereas VMs virtualize an entire operating system, containers and unikernels aim to virtualize only a single software process and any libraries it may require. Containers do this by sharing the kernel of a host operating system, while unikernels are a relatively recent method to create minimal VMs with a very low func-

tional and operational software overhead. In recent years, the tool chains used to create unikernels have grown from proofs of concept to platforms that can run both new and existing software written in various programming languages. The performance of unikernels and Docker containers is studied in the context of REST services and heavy processing workloads, written in Java, Go, and Python. The results show that the performance of unikernels is heavily dependent on the language used, and that multi-threading generally results in a performance penalty rather than performance scaling. However, single-threaded unikernels written in Golang and Java are generally significantly faster than equivalent containers.

Network virtualization can be used to create Virtual Private Networks (VPN) or Software Defined Networks (SDN) between containers, and to transparently route and monitor traffic using Network Function Virtualization (NFV). Because of the rising use of containers on increasingly powerful devices in the network edge, there is a need for reliable and secure networking between these devices.

The move from a secure data center network to heterogenous public and private networks presents issues in terms of security and network topology, which can be partially solved by using a VPN to connect the network edge to the cloud. The scalability of VPN software is evaluated to determine if and how a VPN can be used in large-scale clusters containing edge devices. Benchmarks are performed to determine the maximum number of VPN-connected nodes and the influence of network degradation on VPN performance, primarily using traffic typical for edge devices generating IoT data. The results indicate that WireGuard is an excellent choice of VPN software to connect hundreds of edge nodes in a cluster, while the other benchmarked software either fails, or has unacceptable failure rates, or both.

Another aspect of "Resource Efficiency and Flexibility" is ensuring a homogeneous environment for software to run on, over all classes of devices encountered in the fog and the network edge. This aspect focuses on containers, and how to orchestrate and run them efficiently on low-resource devices.

FLEDGE is presented as a Kubernetes-compatible container orchestrator based on Virtual Kubelets, aimed primarily at container orchestration on low-resource edge devices. Several aspects of low-resource container orchestration are examined, such as the choice of underlying container runtime, and how to realize resource-efficient container networking. A number of evaluations are performed to determine how FLEDGE compares to Kubernetes and K3S in terms of resource requirements. Further evaluations performed in late 2021 also compare FLEDGE to K0S, ioFog and KubeEdge, showing in each case that FLEDGE is the most resource-efficient solution, at the cost of limiting the functionality to edge-specific needs.

Whereas cloud orchestrators handle hundreds to (at most) thousands of devices, the network edge contains more devices by several orders of magnitude, leading to the need for a highly scalable orchestration algorithm which

also minimizes resource use on all nodes. To this end, Swirly is presented as a near real-time service orchestration and scheduling solution. Using minimal agents on each node, and taking into account frequent changes in network conditions and connected devices, Swirly generates near-optimal software service topologies in real-time from data reported by nodes. Its theoretical performance is explored, and a model of the behaviour and limits of fog nodes is constructed. An evaluation of the algorithm shows that Swirly is capable of managing services for hundreds of thousands of devices in near real-time.

For "Scaling Towards Smart Cities", solutions in network and software management are needed that reliably scale to any required situation. Various scalability mechanisms are examined; careful use of device resources and offloading tasks to other devices can improve local scaling (e.g. more tasks per device), federation combines the resources of several networks to better accomplish specific goals, and decentralization allows for self-organization and (technically) limitless scaling of service architectures.

In the context of federations, Flexible federated Unified Service Environment (FUSE) is presented as a solution for joining multiple private networks into a microservice based ad-hoc federation, and for deploying and managing container-based software on the devices of a federation. The main use case of FUSE is crisis situations, in which it is important to be able to quickly gather information from various sources to form a complete and accurate picture of the situation (e.g. video streams, alarms). A video streaming application is deployed on an example federation to demonstrate the performance of a federation setup and applications running on it. The results show that FUSE can be deployed within minutes, and that it can support multiple video streams under normal network conditions, making it a viable solution for the problem of fast and straightforward cross-domain federation.

A second application of scalability is the decentralization of Swirly into Self-Organizing Swirly (SoSwirly). Swirly evaluations show that as the topology it manages grows, memory use and network traffic inevitably outgrow the capacity of a centralized algorithm instance on a single machine. Since the fog is essentially a geographically distributed computational substrate, a suitable solution is to use a decentralized service scheduler, deployed on all nodes, which can monitor and deploy services in its neighbourhood without having to know the entire service topology. SoSwirly scales service instances as required by the edge, based on available resources and flexibly defined distance metrics. The mathematical model of fog networks developed for Swirly is extended, and a theoretical analysis and an empirical evaluation of its performance are provided which indicate that under most conditions, SoSwirly is highly scalable. Additionally, the evaluations show that the resource requirements of SoSwirly, combined with FLEDGE as a container engine, are low enough to run on a large variety of edge devices.

Finally, in the topic of "Intelligence in Smart Cities", the various types of

AI that are used in edge networks are discussed, as well as how they can enable intelligent management of resources or create novel, intelligent end-user applications.
SoSwirly is evaluated in the context of decentralized learning of Artificial Neural Networks, providing the basis of future work to use SoSwirly as a distributed weight update mechanism, and to integrate a predictive, AI-based component into SoSwirly to actively pick the best service provider for any node, rather than merely reacting to topology changes.
Finally, a review of the state of the art of AI in the network edge is presented. A taxonomy is provided with "Enabling Technology" for Edge Intelligence, "Organization" of the edge using AI, and AI "Applications" in the edge as its main topics. Research trend data from 2015 to 2020 is presented for various subdivisions of these topics, showing an exponential increase in both absolute and relative research interest in each subtopic. The "Organization" aspect, which is the main focus of the review, has a more fine-grained subdivision, explaining all contributing factors in detail. For each subdivision of the taxonomy a number of selected studies from 2019 to 2021 are gathered to form a high-level illustration of the state of the art of Edge Intelligence. From these selected studies and the trend data, a number of short-term challenges and high-level visions for Edge Intelligence are formulated, providing a basis for future work.
To summarize, this thesis handles concrete contributions in various subfields of Smart Cities, culminating in a review of how the "smart" aspect is enabled and leveraged, and how future work can result in improved intelligence and scalability of networks and applications.

1 Introduction

Contents May Differ - Culture GSV, Iain M. Banks

1.1 Smart Cities

In the last decade, computers have become embedded in ever more devices, ranging from TVs to portable devices with a whole range of sensors. Most of these devices are networked, and rely on external software services to report sensor data or perform complex computations. Such services are often hosted in immense, centralized cloud data centers, or closer to end users in smaller fog data centers, while the networked end-user devices themselves form what is known as the network edge. The enormous influx of data generated by such devices, and their requirements for supporting software services, have generated research fields that attempt to optimally route network traffic and schedule processing tasks in this computational chaos. Additionally, this wealth of data and processing power has led to novel applications in homes, traffic management, health care, and industry. These application domains, together with their supporting research fields, can be summarized as "City of Things" or "Smart Cities".

This chapter introduces the global topic of this dissertation, *Scalable and Flexible Low-resource Service Orchestration for Enabling Smart Cities*, from which three main research subjects flow. Each topic is initially presented by itself in one or two chapters, then discussed in terms of the previous

chapters' topics, and finally presented in the context of Smart Cities.

- *Flexible Low-resource Orchestration*, or rather the necessity for both *Resource Efficiency and Flexibility*, concerns topics that make software easier and more reliable to run on a multitude of systems, while hiding the underlying complexity from application developers and system architects. These requirements are shown to be (partially) enabled through virtualization [1, 2] and a careful design of software orchestration systems [3, 4] with low resource requirements.

- *Scalable Service Orchestration*, or in the context of this dissertation *Scaling Towards Smart Cities*, is required to effectively and efficiently scale *Reliable Software Systems* in the extensive network edge [5] in Smart Cities, which is formed by various IoT and end-user devices. Most of the work in this dissertation will be considered in the context of the network edge (or edge network), which forms the principal subject of current research efforts in Smart Cities.

- The last topic requires putting *Intelligence in Smart Cities*, using both scalability and reliability to enable and leverage Artificial Intelligence (AI)[6] in the extensive network edge, leading to smart applications that can intelligently automate various aspects of cities.

The general outline of this dissertation is further described in Section 1.2, mapping the three topics to their respective chapters and giving a short overview of research efforts in each chapter. The publications resulting from the work in this dissertation are detailed in Section 1.3, listing the publication details and abstract for each one.

1.2 Outline

Fig. 1.1 shows the general outline of this dissertation. Items in italic text denote sub-chapters with content adapted from publications. These chapters are partly based on papers published by the doctoral student, as allowed by the copyright agreements with the publishers and agreed by all authors. At the start of each relevant sub-chapter, the published version is referenced in full.
Resource Efficiency and Flexibility concerns easy, reliable software development and deployment. Chapter 2 handles the basics, discussing the advantages of different types of virtualization for processes and networks, and presenting research contributions in virtualization. Chapter 3 handles flexibility and resource efficiency in the complex environment of fog and edge

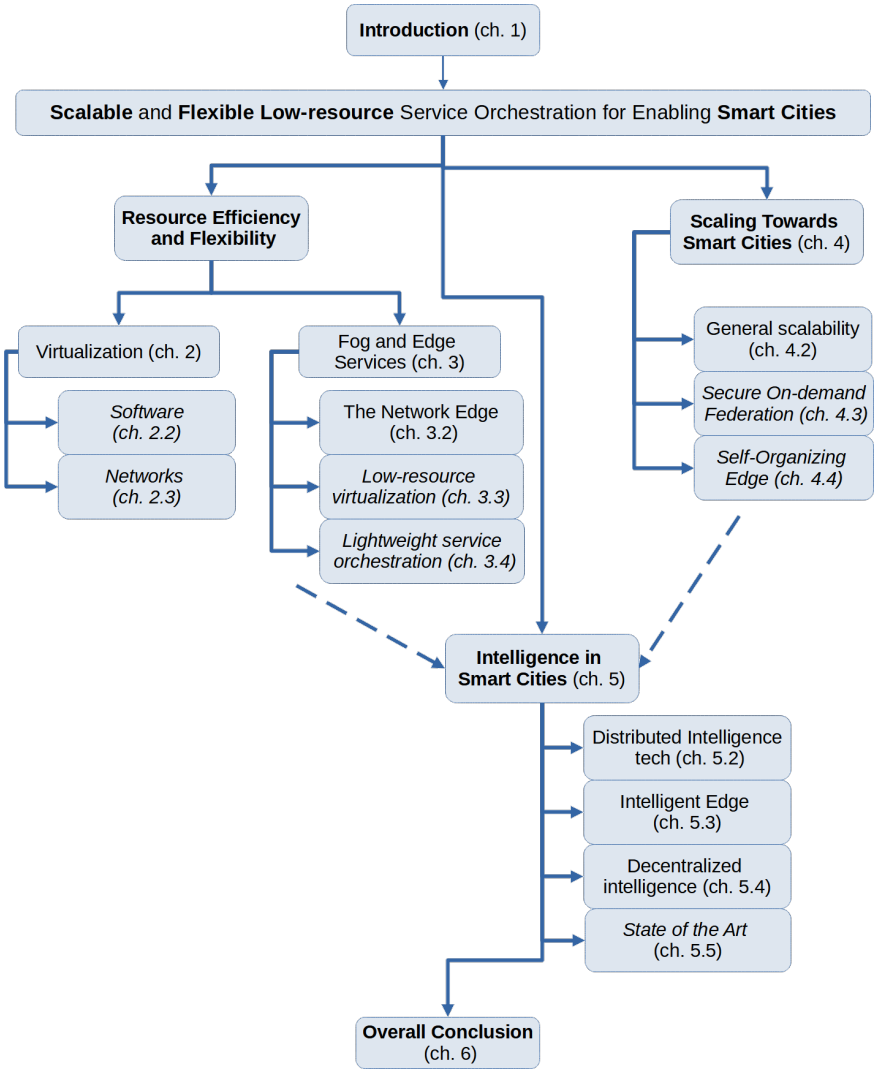


Figure 1.1: Visual outline of this dissertation.

networks, discussing contributions in low-resource virtualization (Chapter 3.3) and service orchestration (Chapter 3.4) with the aim of flexibly utilizing currently unused computing resources in the network edge.

Scaling Towards Smart Cities, the sole topic of Chapter 4, presents work related to the scalability of software services in the network edge. Fast, secure, on-demand federation of private networks with sensitive resources is discussed in Chapter 4.3, while Chapter 4.4 presents a solution for decen-

tralized, self-organizing microservice architectures in edge networks based on the work from Chapters 3.3 and 3.4.
Intelligence in Smart Cities, as shown in Chapter 5, can be enabled by virtue of highly reliable and scalable service orchestration, taking the work from previous chapters as a basis. This chapter gives a summary of the more common types of Artificial Intelligence (AI), and goes on to discuss the concept of Edge Intelligence (EI), a framework for decentralized intelligence in Chapter 5.4 based on the work from Chapter 4.4, and an overview of the state of the art of Edge Intelligence in Chapter 5.5.
At the end of each of these topics, a "Summary" section reiterates the important concepts and conclusions. The overall conclusion of the dissertation is stated in Chapter 6, summarizing the main research subjects and the contributions made in each of them.

1.3 Publications

This section lists all the work published as first author during the course of the PhD leading to this dissertation. Publication details and abstract are provided, and for some publications the content is reused in the course of this dissertation.

1.3.1 Publications in international journals

Near real-time optimization of fog service placement for responsive edge computing, T. Goethals, F. De Turck, B. Volckaert,
published in **Journal of Cloud Computing, 9, Article number: 34 (2020)**

Abstract - In recent years, computing workloads have shifted from the cloud to the fog, and IoT devices are becoming powerful enough to run containerized services. While the combination of IoT devices and fog computing has many advantages, such as increased efficiency, reduced network traffic and better end user experience, the scale and volatility of the fog and edge also present new problems for service deployment scheduling. Fog and edge networks contain orders of magnitude more devices than cloud data centers, and they are often less stable and slower. Additionally, frequent changes in network topology and the number of connected devices are the norm in edge networks, rather than the exception as in cloud data centers. This article presents a service scheduling algorithm, labelled "Swirly", for fog and edge networks containing hundreds of thousands of devices, which is capable of incorporating changes in network conditions and connected

devices. The theoretical performance is explored, and a model of the behaviour and limits of fog nodes is constructed. An evaluation of Swirly is performed, showing that it is capable of managing service meshes for at least 300.000 devices in near real-time.

Extending Kubernetes Clusters to Low-resource Edge Devices using Virtual Kubelets, T. Goethals, F. De Turck, B. Volckaert,
published in **IEEE Transactions on Cloud Computing, 2020**

Abstract - In recent years, containers have gained popularity as a lightweight virtualization technology. This rise in popularity has gone hand in hand with the adoption of microservice architectures, mostly thanks to the scalable, ethereal and isolated nature of containers. More recently, edge devices have become powerful enough to be able to run containerized microservices, while remaining flexible enough in terms of size and power to be deployed almost anywhere. This has triggered research into several container placement strategies involving edge networks, leading to concepts such as osmotic computing. While these container placement strategies are optimal in terms of workload placement, current container orchestrators are often not suitable for running on edge devices due to their high resource requirements. In this article, FLEDGE is presented as a Kubernetes-compatible container orchestrator based on Virtual Kubelets, aimed primarily at container orchestration on low-resource edge devices. Several aspects of low-resource container orchestration are examined, such as the choice of container runtime and how to realize container networking. A number of evaluations are performed to determine how FLEDGE compares to Kubernetes and K3S in terms of resource requirements, showing that it needs around 60MiB memory and 78MiB storage to run on a Raspberry Pi 3, including all dependencies, which is significantly less than both studied alternatives.

Self-organizing Fog Support Services for Responsive Edge Computing, T. Goethals, F. De Turck, B. Volckaert,
published in **Journal of Network and Systems Management volume 29, Article number: 16 (2021)**

Abstract - Recent years have seen fog and edge computing emerge as new paradigms to provide more responsive software services. While both these concepts have numerous advantages in terms of efficiency and user experi-

ence by moving computational tasks closer to where they are needed, effective service scheduling requires a different approach in the geographically widespread fog than it does in the cloud. Additionally, fog and edge networks are volatile, and of such a scale that gathering all the required data for a centralized scheduler results in prohibitively high memory use and network traffic. Since the fog is a geographically distributed computational substrate, a suitable solution is to use a decentralized service scheduler, deployed on all nodes, which can monitor and deploy services in its neighbourhood without having to know the entire service topology. This article presents a fully decentralized service scheduler, labeled "SoSwirly", for fog and edge networks containing hundreds of thousands of devices. It scales service instances as required by the edge, based on available resources and flexibly defined distance metrics. A mathematical model of fog networks is presented, along with a theoretical analysis and an empirical evaluation which indicate that under the right conditions, SoSwirly is highly scalable. It is also explained how to achieve these conditions by carefully selecting configuration parameters. Concretely, only 15MiB of memory is required on each node, and network traffic in the evaluations is less than 4Kbps on edge nodes, while 4-6% more service instances are created than by a centralized algorithm.

Enabling and Leveraging AI in the Intelligent Edge: A Review of Current Trends and Future Directions, T. Goethals, B. Volckaert, F. De Turck,
published in **IEEE Open Journal of the Communications Society, 2021**

Abstract - The use of AI in Smart applications and in the organization of the network edge presents a rapidly advancing research field, with a great variety of challenges and opportunities. This article aims to provide a holistic review of studies from 2019 to 2021 related to the Intelligent Edge, a concept comprising both the use of AI to organize edge networks (Edge Intelligence) and Smart applications in the edge. An introduction is given to the technologies required to understand the state of the art of AI in edge networks, and a taxonomy is provided with "Enabling Technology" for Edge Intelligence, "Organization" of the edge using AI, and AI "Applications" in the edge as its main topics. Research trend data from 2015 to 2020 is presented for various subdivisions of these topics, showing both absolute and relative research interest in each subtopic. The "Organization" aspect, being the main focus of this article, has a more fine-grained subdivision, explaining

all contributing factors in detail. The trends indicate an exponential increase in research interest in nearly all subtopics, but significant differences between them. For each subdivision of the taxonomy a number of selected studies from 2019 to 2021 are gathered to form a high-level illustration of the state of the art of Edge Intelligence. From these selected studies and the trend data, a number of short-term challenges and high-level visions for Edge Intelligence are formulated, providing a basis for future work.

1.3.2 Publications in international conferences

Unikernels vs Containers: An In-Depth Benchmarking Study in the context of Microservice Applications, T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, F. De Turck,

published in **2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2), IEEE, 2018**

Abstract - Unikernels are a relatively recent way to create and quickly deploy extremely small virtual machines that do not require as much functional and operational software overhead as containers or virtual machines by leaving out unnecessary parts. This paradigm aims to replace bulky virtual machines on one hand, and to open up new classes of hardware for virtualization and networking applications on the other. In recent years, the tool chains used to create unikernels have grown from proof of concept to platforms that can run both new and existing software written in various programming languages. This paper studies the performance (both execution time and memory footprint) of unikernels versus Docker containers in the context of REST services and heavy processing workloads, written in Java, Go, and Python. With the results of the performance evaluations, predictions can be made about which cases could benefit from the use of unikernels over containers.

FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers, T. Goethals, D. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, B. Volckaert,

published in **Proceedings of the 9th International Conference on Cloud Computing and Services Science - CLOSER, 90-99, 2019**

Abstract - In crisis situations, it is important to be able to quickly gather information from various sources to form a complete and accurate picture of the situation. However, the different policies of participating companies often make it difficult to connect their information sources quickly, or to allow software to be deployed on their networks in a uniform way. The difficulty in deploying software is exacerbated by the fact that companies often use different software platforms in their existing networks. In this paper, Flexible federated Unified Service Environment (FUSE) is presented as a solution for joining multiple domains into a microservice based ad hoc federation, and for deploying and managing container-based software on the devices of a federation. The resource requirements for setting up a FUSE federation are examined, and a video streaming application is deployed to demonstrate the performance of software deployed on an example federation. The results show that FUSE can be deployed in 10 minutes or less, and that it can support multiple video streams under normal network conditions, making it a viable solution for the problem of quick and easy cross-domain federation.

Scalability evaluation of VPN technologies for secure container networking, T. Goethals, D. Kerkhove, B. Volckaert, F. De Turck,

published in **2019 15th International Conference on Network and Service Management (CNSM), 2019, pp. 1-7**

Abstract - For years, containers have been a popular choice for lightweight virtualization in the cloud. With the rise of more powerful and flexible edge devices, container deployment strategies have arisen that leverage the computational power of edge devices for optimal workload distribution. This move from a secure data center network to heterogenous public and private networks presents some issues in terms of security and network topology that can be partially solved by using a Virtual Private Network (VPN) to connect edge nodes to the cloud. In this paper, the scalability of VPN software is evaluated to determine if and how it can be used in large-scale clusters containing edge nodes. Benchmarks are performed to determine the maximum number of VPN-connected nodes and the influence of network degradation on VPN performance, primarily using traffic typical for edge devices generating IoT data. Some high level conclusions are drawn from the results, indicating that WireGuard is an excellent choice of VPN software to connect edge nodes in a cluster. Analysis of the results also shows the strengths and weaknesses of other VPN software.

Adaptive Fog Service Placement for Real-time Topology Changes in Kubernetes Clusters, T. Goethals, B. Volckaert, F. De Turck, published in **Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER, 161-170, 2020**

Abstract - Recent trends have caused a shift from services deployed solely in monolithic data centers in the cloud to services deployed in the fog (e.g. roadside units for smart highways, support services for IoT devices). Simultaneously, the variety and number of IoT devices has grown rapidly, along with their reliance on cloud services. Additionally, many of these devices are now themselves capable of running containers, allowing them to execute some services previously deployed in the fog. The combination of IoT devices and fog computing has many advantages in terms of efficiency and user experience, but the scale, volatile topology and heterogeneous network conditions of the fog and the edge also present problems for service deployment scheduling. Cloud service scheduling often takes a wide array of parameters into account to calculate optimal solutions. However, the algorithms used are not generally capable of handling the scale and volatility of the fog. This paper presents a scheduling algorithm, named "Swirly", for large scale fog and edge networks, which is capable of adapting to changes in network conditions and connected devices. The algorithm details are presented and implemented as a service using the Kubernetes API. This implementation is validated and benchmarked, showing that a single threaded Swirly service is easily capable of managing service meshes for at least 300.000 devices in soft real-time.

FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices, T. Goethals, F. De Turck, B. Volckaert, published in **IOV 2019: Internet of Vehicles. Technologies and Services Toward Smart Cities pp 174-189**

Abstract - In recent years, containers have quickly gained popularity in the cloud, mostly thanks to their scalable, ethereal and isolated nature. Simultaneously, edge devices have become powerful enough to run containerized microservices, while remaining small and low-powered. These evolutions have triggered a wave of research into container placement strategies on clusters including edge devices, leading to concepts such as fog computing. These container placement strategies can optimize workload placement across cloud and edge clusters, but current container orchestrators are very

resource intensive and are not designed to run on edge devices.
This paper presents FLEDGE as a Kubernetes compatible edge container orchestrator. A number of aspects of how to achieve low-resource container orchestration are examined, for example the choice of container runtime and how to implement container networking. Finally, a number of evaluations are performed, comparing FLEDGE to K3S and Kubernetes, to show that it is a viable alternative to existing container orchestrators.

Live Demonstration of a Highly Scalable Fog Service Orchestrator,
T. Goethals, F. De Turck, B. Volckaert,
published in **2021 IEEE 7th International Conference on Network Softwarization (NetSoft)**

Abstract - In recent years, computing workloads have shifted from the cloud to the fog and edge, as IoT devices are becoming powerful enough to run containerized services. While the fog and edge computing can increase energy efficiency, reduce network traffic and provide better end user experience, the scale and volatility of the fog and edge also present new problems for service scheduling. In the edge, there are orders of magnitude more devices than in cloud data centers, and conditions are often less stable. Additionally, unlike in data centers, the network topology of the edge often changes, requiring a real-time approach to scheduling. In this demonstration, an implementation of a highly scalable orchestrator named "Swirly" is presented. The challenge of fog service scheduling is illustrated by using this implementation to organize software services in near real-time and on-demand in a virtual representation of a real-world industry park. Performance indicators are presented to show that this solution can scale up to 300.000 edge nodes.

References

[1] Aditya Bhardwaj and C. Rama Krishna. *Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey.* Arabian Journal for Science and Engineering, apr 2021.

[2] N.M.M.K. Chowdhury and R. Boutaba. *Network Virtualization: State of the Art and Research Challenges.* IEEE Communications Magazine, 47(7):20–26, jul 2009.

[3] Emiliano Casalicchio. *Autonomic Orchestration of Containers: Problem Definition and Research Challenges.* In Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools. ACM, 2017.

[4] Saiful Hoque, Mathias Santos de Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. *Towards Container Orchestration in Fog Computing Infrastructures.* In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). IEEE, jul 2017.

[5] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. *All one needs to know about fog computing and related edge computing paradigms: A complete survey.* Journal of Systems Architecture, 98:289–330, sep 2019.

[6] Gerd Brewka. *Artificial intelligence—a modern approach by Stuart Russell and Peter Norvig, Prentice Hall. Series in Artificial Intelligence, Englewood Cliffs, NJ.* The Knowledge Engineering Review, 11(1):78–79, mar 1996.

2 Virtualization

"All problems in computer science can be solved by another level of indirection." - Fundamental theorem of software engineering, David J. Wheeler

2.1 Introduction

In this chapter, the basic technologies and concepts behind virtualization are introduced, an important topic considering the omnipresence of virtualization in modern service architectures and Smart City initiatives. Considering the overall topic, this chapter sometimes presents these technologies in the context of edge networks. Although edge networks are only introduced in Chapter 3, for the purposes of this chapter it is enough to define them as large scale heterogeneous networks consisting mostly of devices with limited resources. The Internet Engineering Task Force (IETF) provides a classification of resource-constrained nodes [1], and in the context of this dissertation, a "device with limited resources" corresponds to classes J10 through J15. According to the same classification, group M devices have neither the resources nor the hardware functionality required to run an operating system kernel, let alone any type of virtualization.
Generally speaking, virtualization is the process by which a piece of computational functionality is decoupled from the sort of hardware it was originally associated with. A very early example of this is Virtual Memory [2], which

greatly simplified programming. Virtualization is sometimes achieved by emulating hardware entirely in software [3] (e.g. emulators for old gaming consoles), other times through paravirtualization, which passes virtual hardware directly through to physical hardware, or as close to it as possible. Paravirtualization is enabled by using specialized instruction sets (e.g. Intel VT-x [4]) or device drivers that explicitly represent virtual devices (e.g. virtio [5]). There are also softer virtualization options, such as OS-level virtualization, which are handled purely in software terms. As computing systems evolve however, there is ever more opportunity to extend or improve virtualization.

Depending on the type of virtualization used, and how it is implemented, there can be significant performance and security repercussions. While virtualization is generally used to increase software security and encapsulation, it is also an extra software layer that can be compromised. Performance overhead on modern systems is usually negligible, and depending on the chosen method of virtualization performance can actually surpass that of software on default operating systems, as shown in Section 2.2.3.

There are two main sections in this chapter. Section 2.2 handles the virtualization options of software processes, while Section 2.3 discusses the virtualization of networks and network functions. Finally, the chapter is summarized in Section 2.4.

2.2 Software

Software or process virtualization is aimed at running one or more processes in a completely encapsulated, virtual environment. This mostly involves emulating or otherwise isolating a large number of devices and file systems. The result of virtualizing a piece of software is usually a system image that can be easily instantiated and scaled up in terms of demand. In this section, common types of process virtualization are introduced, as well as the concept of container engines, and the performance of containers and unikernels is compared in the context of microservice applications.

2.2.1 Types of virtualization

This section discusses only the most important types of virtualization for the field of Smart Cities. Early virtualization efforts were mostly focused on flexibility and encapsulation, but the last few years especially have seen a proliferation of alternatives, most of which focus on fast startup times, low resource footprints and scalability.

2.2.1.1 Virtual Machines

Virtual Machines (VM) essentially consist of system images containing a complete Operating System (OS) and some software services, with all the requisite drivers, libraries and programs to support the OS and services. VMs preferably use paravirtualization, combined with whatever hardware acceleration (e.g. VT-x) is at hand, to create a virtual environment for the system image to run in. As the system image contains an entire operating system, this method necessarily has to create or pass-through dozens of devices, not all of which are used or useful, which are managed by a hypervisor. Because of their architecture, VMs generally require tens of GiB of disk space and several GiB of RAM, limiting their scalability. However, their great advantage is that a VM kernel runs independently of the host, so that a single machine can run any combination of OSs in VMs.
There are two types of hypervisors, each with their advantages:

- Type II hypervisors run on top of a standard OS, enabling virtualization through emulation by default, although modern OSs have good built-in mechanisms for paravirtualization, allowing performance to come close to bare-metal performance as long as no specialized hardware is required (e.g. GPUs). This option is only feasible for a small number of concurrent VMs, and usually reserved for development and testing.

- Type I hypervisors, illustrated in Fig. 2.1, are heavily modified kernels that run directly on bare-metal and are built entirely around fast and efficient paravirtualization. They install custom, virtualization-ready drivers in VMs (e.g. virtio) that can integrate with the hypervisor itself for optimal performance, executing as many instructions as possible close to bare-metal. This option performs noticeably better, and has somewhat better scalability.

2.2.1.2 Containers

Containers use OS-level virtualization, built into the kernel itself, to isolate software processes. Most concepts involved in containers have evolved separately, in order to isolate various aspects of processes; the filesystem root of a process can be changed to prevent it from affecting the host or other processes, kernel namespaces can achieve the same effect for devices, often used to isolate and regulate network traffic in containers, and control groups (cgroups) can enforce resource limits on processes (e.g. memory and CPU use). Due to this "soft" virtualization, containers can technically use any

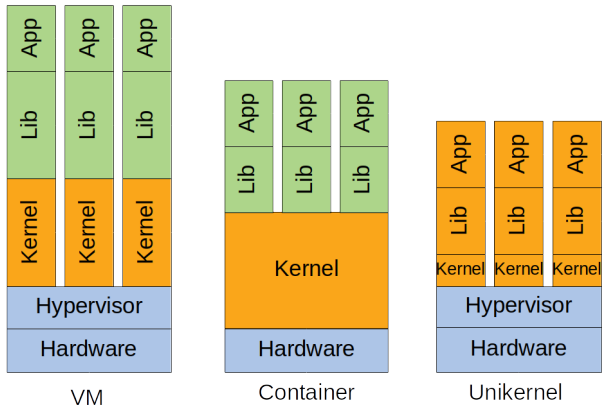


Figure 2.1: Comparison of virtual machine, container and unikernel system architecture

and all hardware devices on a machine using their default drivers, e.g. AI containers using GPU acceleration [6].
As containers share the kernel of the host system instead of running on top of their own OS, a container image is generally orders of magnitude smaller than a VM image, on the scale of a few hundred MiB. This concept is illustrated in Fig. 2.1, although not to scale. Similarly, memory use is much lower, as it is limited to what the virtualized process needs. As a result, containers can boot much faster than VMs, images can be transferred much faster over networks, and the processes are highly scalable and even perform better than VMs because they do not require a paravirtualization layer. There are also disadvantages, mostly in terms of flexibility and security [7]. Because containers share a kernel with the host, any kernel vulnerability can be exploited to compromise the host system and all other containers. Similarly, a host can only run containers that are built for the same kernel (version), ironically leading to the need for a VM to run Linux containers on Windows, although Windows Subsystem for Linux 2 (WSL2) has significantly improved performance by allowing containers to be wrapped in a micro-VM with a custom-built Linux kernel designed for integration with the Windows host, running on Hyper-V [8].

2.2.1.3 Unikernels

Unikernels are a relatively new concept in which software is directly integrated into the kernel it is running on. This is achieved by compiling source code, along with only the required system calls and drivers, into one

executable program using a single address space [9]. Because of this design, unikernels can only run a single process, thus forking does not exist. The build process results in a complete (virtual) machine image of minimal size that only contains and executes the code that it absolutely needs to. Again, Fig. 2.1 illustrates this concept by comparing VMs, containers and unikernels. The figure uses a blue color to indicate hardware or hypervisors, orange to indicate kernel space and green to indicate user space. The reduced kernel and system complexity can make a unikernel much faster than a regular VM [10]. Despite only being able to run a single process, multi-threading is usually possible [11, 12]. An added advantage of unikernel design is that they are less vulnerable to security problems, since there are typically less attack options, except the program, the required libraries and the kernel functions they use. The downside is that programs always run in kernel mode, making it easier for bugs and hacks that do succeed to critically break the machine, while making it harder to debug [9] because unikernels usually do not have their own sets of debugging tools. Additionally, all the facilities and libraries used by debugging tools would have to be included, ballooning the size of the unikernel, and any debugging tool that requires another process to run can not work in a unikernel by design. At the time of writing[1], all platforms generate unikernels as VM images, but work is underway to run them as bare metal images [13]. By eliminating the need for a classical operating system, the image size difference with regular VMs can be in the order of gigabytes. Additionally, unikernels have a much shorter boot time than full VMs, in the order of hundreds of milliseconds compared to tens of seconds, and consume much less memory [14, 15]. The same claims have been made for replacing containers with unikernels, but in this case the advantage in terms of boot time seems to be much smaller [15], except for highly specialized unikernels built around a single language [14]. Many proof-of-concept unikernels of all sorts of software exist, including database engines, REST services and a RAMP stack (Rumprun, Apache, MySQL, PHP) [16]. A lot of IoT services are composed of different pieces of software, but these can easily be broken up into a number of individual single process components ready to be converted into a unikernel. Because unikernel images are far smaller than regular VMs, a lot of space on cloud infrastructure could be saved by using them. Furthermore, because unikernels are smaller than VMs and boot faster, microservice deployment strategies could use unikernels for more flexible and dynamic deployment [17]. In addition to replacing existing VMs and containers, unikernels can also open up new classes of hardware for cloud use that are otherwise unfit to run an entire operating system [17]. This could happen by either running unikernels

[1] June 2018 for this section about unikernels

under a type I hypervisor or by (eventually) running them on bare-metal. The added advantage of using a hypervisor and unikernels over simply running embedded software (as is now usually the case) is that any unikernel can be easily deployed to any machine in the cloud running a hypervisor, and function as a part of a uniform environment. Embedded software on the other hand often requires local access to update, or is incompatible with cloud management software.
Broadly speaking, there are two approaches to building unikernels:

- Designed from the ground up around a single programming language, providing a custom (not POSIX [18] compatible) system API. At compile time, all the libraries and system calls used by the program are compiled, together with the program itself, creating a single kernel that is started on boot. This type of unikernel platform is generally incompatible with existing source code, requiring a full rewrite using the platform's API. Considering the still-evolving nature of existing platforms of this type, it also often means dropping features that are not yet implementable. On the other hand, this type of unikernel results in superior performance and much smaller images than the other type [10, 19]. IncludeOS and MirageOS are good examples of this approach [20, 21].

- POSIX compatible operating systems that can run existing software by cross-compiling it using existing compilers. These platforms generally have custom kernel implementations and drivers to make them faster, usually with options for paravirtualization. Unikernels built this way have larger kernels and resource requirements, but make up for it in ease of use. This type is very useful for converting software that runs on existing VMs and containers into unikernels, since the software only needs to be recompiled, not rewritten. Examples include OSv and rumprun [22, 23].

Since unikernels contain everything from a kernel to user software, they differ from containers in certain aspects:

- A hypervisor is required to run a unikernel, but this also means it can be run on a type I hypervisor, removing the need for a bulky OS.

- A unikernel's built-in kernel, no matter how small, will increase memory use compared to running a program in a container.

- Since a unikernel does not require context switches from user space to kernel space and has simpler device drivers, it should have a significant speed advantage over a container, because a container runs

on a much more complex host kernel. Of course, since the kernel of a container can run on bare metal and a unikernel has to be run on a hypervisor, the actual performance will depend greatly on the chosen hypervisor. The results from the tests in this paper indicate that unikernels running on type I hypervisors can in fact give much better performance than containers.

- Unikernels are single process by design, so anything requiring multiple parallel processes must be broken up into separate unikernels. Additionally, breaking software up into separate unikernels will induce at least some communication overhead between the different parts.

2.2.1.4 WebAssembly

The newest innovation, although not necessarily a type of virtualization, is WebAssembly (WASM) [24], evolving from standardization and a need for faster JavaScript web applications into a platform for generic, sandboxed applications.
Although WASM was originally created for in-browser web applications, projects such as WebAssembly System Interface (WASI) [25] have created API's that allow system programming with WASM, providing features such as threading and file system access. Although WASI is POSIX-like, it is not necessarily POSIX compatible, requiring a thorough rewrite of software to use the WASI API's. It is currently in an early stage, and does not yet support networking, and thus lacks the capacity for web services. The features provided by WASI are sandboxed to provide a security and isolation layer around the software it runs, creating a type of lightweight virtualization. Additionally, since WASI works with LLVM [26], WASM programming language support has been extended to languages such as C and C++.
Conceptually, WASM/WASI is similar to how POSIX-compatible unikernels are built. Essentially, both can be considered a "library" OS whose compiler integrates system calls directly into the software, running the result as a single process in kernel mode. The main difference between WASM and unikernels is that WASM requires a dedicated runtime rather than a hypervisor, which may be optimized for limited resource requirements. This also means that, at the cost of some extra development effort, WASM runtimes can be created for many types of devices that are not powerful enough to run the hypervisors required for unikernels, or the relatively sizeable Linux kernel and libraries required for containers. Wasmachine [27] is an example of this, an OS dedicated to running WASM applications on IoT hardware, with a focus on security, which is shown to be 11% faster than Linux for the evaluated applications. Another study of WASM [28] shows it to be

useful in the context of serverless applications and functions, examining the potential benefits and drawbacks of various platforms and wrappers, and showing native WASM performance to be close to that of native C.

2.2.1.5 Firecracker

Similar to unikernels, Firecracker [29] aims to run a single process in a minimal VM, or microVM, the main difference being that it uses a default kernel and the virtualized process is run in user mode on top of it. Firecracker aims to create extremely fast booting microVMs, with as little memory overhead as possible, and uses Kernel-based Virtual Machine (KVM [30]) as its hypervisor. The command-line tools are similar to those of Docker and OSv, although compiling is not as straightforward as the use of a Dockerfile.
Networking features are limited to the creation of a device in the VM itself, which can be linked to a user-created TAP device on the host. Assigning IP addresses, creating traffic rules, and routing are left entirely as an exercise for the user, making this one of the more difficult alternatives to set up. However, possible approaches range from using a simple proxy, to bridging network interfaces (similar to Docker), and letting a Container Networking Interface (CNI) plugin set up all necessary devices and rules.

2.2.2 Container engines

Of the virtualization types discussed so far, containers are by far the most popular choice in edge computing due to their low resource requirements and straightforward operation; any device with a suitable Linux kernel[2] can build and run containers, even limited devices such as a Raspberry Pi[3]. However, this is the result of a long evolutionary path of container technology, and despite their simplicity containers still need management software to reliably create containers, to run them, and to ensure compatibility with alternatives. This section presents the most popular container engines and standards.

2.2.2.1 Docker

Docker [31], originally released in 2013, is an all-round, high-level container engine. It is capable of building containers from declarative files (Dockerfiles) that can be used recursively, and in running containers it organizes every aspect from file systems to container networking. For the latter, Docker will use its own addressing system to assign IP address to containers

[2] https://blog.hypriot.com/post/verify-kernel-container-compatibility/
[3] https://www.raspberrypi.com/documentation/

unless overridden by higher management software. During its development track, Docker has used various plugins and lower level container runtimes to outsource functions to, eventually settling on containerd to organize kernel-related functions such as creating default namespaces and changing root paths. Similarly, due to its popularity and extensive development effort, Docker has contributed to various standards related to container construction and execution, most importantly the Open Container Initiative (OCI), which standardizes container image formats and engine interaction, and the CNI, which provides an API for container networking software. These standards are further discussed in Chapter 3.3, where they are considered in terms of low-resource container engines in the network edge.

2.2.2.2 containerd

containerd [32] is a lower-level container engine, often referred to as a container runtime, which is entirely aimed at starting and running OCI-compatible containers. It has less functionality than Docker, and much of the functionality that Docker offers through a command-line interface (CLI) can only be accessed through an API in containerd. For example, containerd in itself does not build containers, it does not set up container networking beyond creating a network namespace, and setting up resource limits and namespaces in general requires more work. However, due to its low-level approach, containerd can be used more flexibly than Docker, with a degree of modularity that allows ignoring, extending or overwriting its default behavior. Additionally, it requires significantly less resources than Docker, as Chapter 3.3 shows, making it a useful container engine for projects in the network edge.

2.2.2.3 Kata Containers

A number of alternative runtimes exist that use or combine aspects of both VMs and containers in order to improve specific aspects such as security or flexibility. Kata Containers [33] run containerized processes along with an agent process inside a minimal VM. The agent process is responsible for organizing container namespaces and transparently relaying commands between containers and container engines. This combination effectively ensures the security and isolation of a unikernel-like design with the flexibility of containerized software, at the cost of a potential performance reduction [34]. As Kata Containers implement OCI standards, they can deploy Docker containers and are compatible with most container orchestration software discussed in Chapter 3.

2.2.3 Performance comparison of unikernels and containers

This section contains the edited version of the following publication: **"Unikernels vs Containers: An In-Depth Benchmarking Study in the context of Microservice Applications"**, T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, F. De Turck,
published in **2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2), IEEE, 2018** [35]

The goal of this section is to compare unikernels and containers as accurately as possible using a number of programming languages popular in edge networks, e.g. Go, Java and Python. For this reason, the rest of the section will focus on the performance of the POSIX-compatible type of unikernels. The reason for this is that POSIX incompatible unikernels simply do not have the facilities and API's to run Go, Java or Python, so their performance can not be compared.
Many aspects of unikernels have been studied in the past, but due to the fast-changing nature of unikernel platforms, the results rapidly become obsolete. For example, boot time comparisons between VMs and unikernels have been made [15], in addition to studies on the boot times of unikernels using a single unikernel platform, often comparing to Linux or Linux VMs [14, 19, 36]. Another study used a DNS server and an HTTP server, implemented as unikernels on different platforms, to compare the networking performance of unikernels versus Linux [10]. Despite interesting results, different programming languages had to be used per unikernel platform, so the results depend somewhat on the specific software implementations these languages allowed. An attempt has been made to employ unikernels in an edge offloading architecture [17], allowing for a more dynamic deployment of IoT services, but that work was hampered by a bug that occurred in the chosen unikernel platform at the time. Other studies into this area have been done [19, 37]. The security advantages of unikernels over containers and regular VMs have been extensively studied [38–40]. The rest of this section details the benchmarking setup used, and discusses performance results in terms of various artificial microservice scenarios.

2.2.3.1 Benchmarking setup

This section discusses the potential platforms for the tests, as well as the physical test setup. The tests and measuring methods used to obtain the results are also explained. Several platforms were examined for their ability to create Go, Java and Python unikernels.

Candidate platforms

OSv (0.51 used in original publication, currently 0.56) is a unikernel platform that works similar to Docker. At the basis of OSv unikernels lies a very small core to which modules and files can be added through a layered build process. Each layer defines a base image to start from, a number of files to add and optional build and command line commands. Using this method, base images were created for Java 1.8, Python 2.7 and Go 1.10, which were then used to create the unikernels for the tests. Running newer versions of both Java (1.9, 1.10) and Python (3.6) was attempted. While higher Java versions worked, a minimal JRE would not start on OSv due to not being compiled correctly. Java 1.8 was deemed sufficient for the purposes of the tests. In the case of Python 3.6, many system calls were required that are not yet implemented in OSv, so 2.7 had to be used. Modules to run Java 1.8 and Python 2.7 (some modification and building required) are included in the OSv source, while Go code was compiled as a shared library and run indirectly via a wrapper supplied by OSv. To confirm that this wrapper does not have any impact on performance, Go was also compiled as a Position Independent Executable (PIE), which can run directly on OSv. A PIE is an executable that can execute properly no matter what its absolute address in the address space is [41]. Note that OSv requires all software running on it to be built as PIE by design [42], but in the cases of Java and Python this is already done by cross-compiling the Java Virtual Machine and interpreter, respectively. For Go, compiling as a shared library has the same result, except that it still needs to be launched by the aforementioned wrapper. The PIE version of Go is included in the results as Go(pie).

Rumprun (unversioned, Apr 8, 2018 commit used in original publication, no activity since May 2020) was also considered as a test candidate. Contrary to OSv, rumprun compiles the source code and all required system libraries and drivers in one step, into a single kernel that is loaded during boot. While this allows for more optimizations than OSv's approach, it also results in a slower and bulkier build process. Unikernels for Python and Go were successfully created using rumprun, but during testing these unikernels never managed to complete more than a few thousand requests before running into a socket allocation bug. This problem was partially fixed under KVM (Kernel-based Virtual Machine [30]) by increasing memory, which caused it to happen after a few million requests, but could not be fixed on the test setup. Memory footprint and image sizes of rumprun unikernels were more or less the same as those of OSv unikernels, but the results are not included because they are incomplete and thus unreliable.

UniK [43] (unversioned, Nov 15 2016 release used in original publication, no activity since July 2019) is a platform that combines several other platforms

(including OSv and rumprun) into one tool chain. While this is certainly an interesting development, UniK is less flexible than OSv and rumprun in terms of what libraries and packages are included in any given unikernel (for example, the Java version cannot be changed), and allows less control over how the images are generated and deployed. Despite both rumprun and OSv supporting static IP addresses, no method was found to assign them to unikernels generated with UniK, which was required for the test setup. Additionally, unikernels seem to be (in part) dependent on UniK's daemon to boot, which caused them to hang on XenServer even before running into IP assignment problems. Unikernels for Python and Go were tested on localhost using VirtualBox, but being based on rumprun, they ran into the same problems and thus are not included in the results.

Although several platforms were examined, only OSv was used for testing because it proved to be the most stable option. Rumprun had some quirks that made testing unreliable and UniK, being partially based on rumprun, showed the same symptoms. Additionally, since the more stable parts of UniK rely on OSv anyway, the latter was chosen as a testing platform to reduce the complexity of the build process.

Test machine

All tests were run on an Intel Core i5-2300 processor with virtualization extensions enabled. The machine had a total of 4GB ram and a 160GB Western Digital hard drive. All VMs and containers were limited to 256MB RAM, either by configuration or by using Docker's –memory flag. In single threaded tests, both VMs and containers were limited to one CPU core and test programs were written to reflect this. For multi threaded tests, all instances were given four cores and the program code was changed to use exactly four threads wherever possible and govern itself where exact numbers could not be forced. The test machine was only used as a server, all client activity was run on a separate machine connected directly to the test machine to avoid result collection from interfering with performance. Only one container or VM was active at any given time during the tests.

Containers were run on Ubuntu 18.04 using Docker 18.03. Container web services were made available by forwarding their ports to the host using docker's -p option (Fig. 2.3). Unikernels were run on XenServer 7.5, a type I hypervisor [44]. The VMs' network interfaces were bridged to the host interface for network access (Fig. 2.2).

The code for all tests is made available for use and review on GitHub[4].

[4]https://github.com/togoetha/unikernels-v-containers

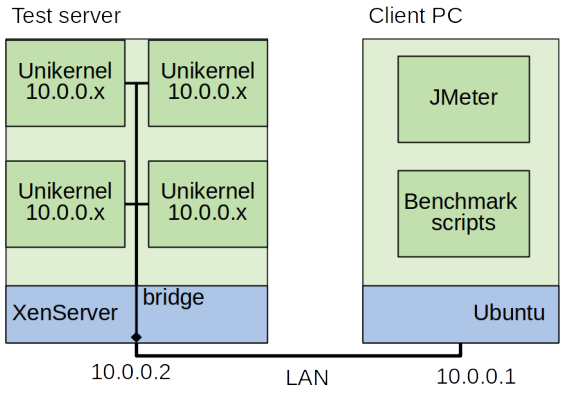


Figure 2.2: XenServer test setup for unikernels

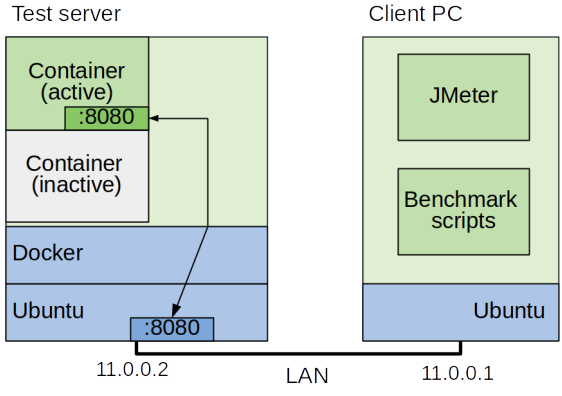


Figure 2.3: Docker test setup for containers

REST service stress test

For Go, Java and Python, a simple REST service was written that contains a static array of to-do items (description, time due and finished/not finished). The service supports the following GET methods:

- /todos: list all items
- /todos/{id}: get the todo with the specified id

While this is a very simple service, it should be a good indicator of how fast a container or unikernel can process REST HTTP requests and a small amount of code. Apache JMeter [45] was used to run 40 concurrent threads, each firing 50000 requests at the test machine to simulate a good sized concurrent demand. Every request called the /todos method, fetching the entire array as JSON.

The Go web service was created with the standard net/http package in combination with the Gorilla Toolkit mux (v1.6.2) [46]. For the Python version, Flask and Flask-Restful [47] were used. In the case of Java, Vert.X 3.5.1 [48] was used. Vert.x is an event-driven, non-blocking toolkit for developing reactive applications [49] in which a verticle is an atomic piece of deployable code. For this test, a verticle was created that listens to HTTP traffic on a specific port and handles incoming messages like REST service requests.

To enable multi-threading, the verticle was instantiated four times under Java, while Python was given free reign by simply enabling multi-threading for Flask. Go automatically creates a number of threads fitting its hardware environment, so no code changes were necessary [50].

Heavy workload test

For the load test, a simple bubble sort algorithm was implemented as identically as possible for each of the tested programming languages. To focus on processing power and memory load instead of networking overhead, the collection to be sorted was made as big as possible without making testing impractical. The array to be sorted is simply a descending array of numbers (x ... 0), where x was chosen at 20000. Each test was repeated 20 times to get an accurate average.

This test was built as a web service, so the same frameworks were used as in the REST service stress test. However, these frameworks matter little in terms of performance since the algorithm takes most of the processing time by far.

2.2.3.2 Results

In this section the results of the tests are discussed. In addition to raw throughput capacity of the machines, memory footprint and response latency will also be reviewed. Response latency gives an indication of how stable a particular unikernel or container is, while memory consumption is useful in determining if unikernels can be deployed on the same scale as containers.

Single-threaded results

Fig. 2.4 shows the results for the REST stress test using a single thread to do all processing (higher is better). The Go service is about 38% faster when running as a unikernel than as a container, while the Java version is about 16% faster as a unikernel. The results confirm the claims that unikernels' reduced kernel and complexity makes them a fast alternative to containers, at least when running on a type I hypervisor.

The equality of Go and Go(pie) performance is clearly visible here, confirming that OSv's Go wrapper has no negative effects.

For Python, being an interpreted language rather than a compiled one and thus slower, the result is a bit harder to interpret from the figure. However, the effect from running it in a unikernel seems to be the same, with a performance of 395±1 requests per second for the unikernel versus 351±1 requests per second for the container. This 15% improvement from running in a unikernel is similar to the improvement for Java, but much lower than for Go.

Fig. 2.5 contains the results for the heavy workload test (lower is better). Note that a logarithmic scale was used to accommodate Python's performance. The execution times for the Go and Java unikernels seem to be on par with those of their container counterparts. The Go unikernel is about 3% slower (0.5% for the PIE), while the Java unikernel is about 1% faster. Again, the chart shows that OSv's Go wrapper is doing a good job of avoiding performance penalties.

Python is having some trouble, the execution time of the unikernel is twice as long as that of the container version. This could be because Python uses a disproportionate amount of operations that run slower in a virtual environment. Existing research shows that array and variable access, the building blocks of bubble sort, take up by far the bulk of Python's interpreting overhead [51]. It stands to reason that either or both of these types of instructions run slowly on either XenServer or OSv, and Java and Go may avoid using them so much.

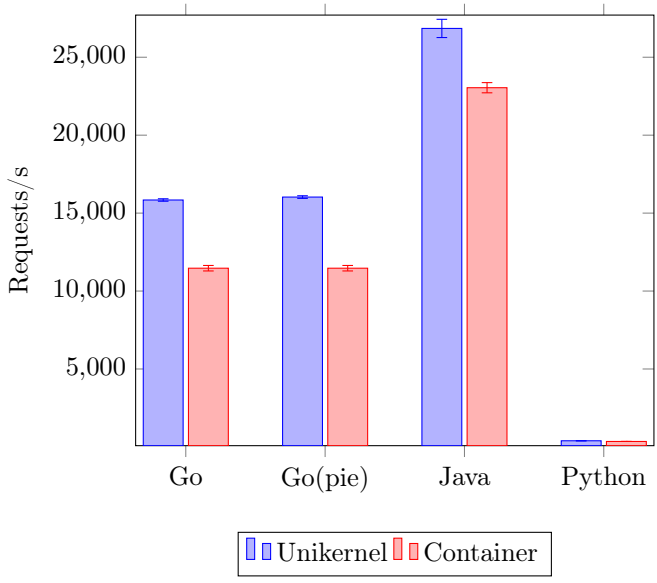


Figure 2.4: REST stress test performance evaluation of unikernels versus containers

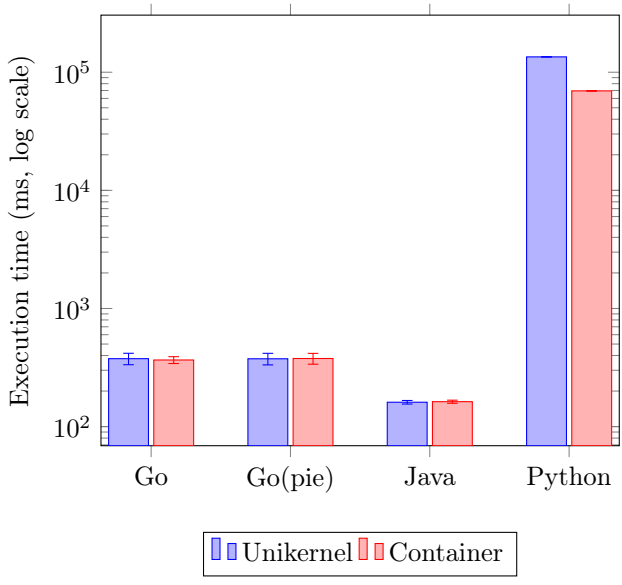


Figure 2.5: Bubble sort execution time of unikernels versus containers (log scale, lower is better)

Multi-threaded results

Unikernels can only run a single process, but since they do support multi-threading this aspect merits some attention. Generally speaking, REST services scale almost linearly with the number of cores available to them. Fig. 2.6 (higher is better) shows good performance scaling for Go and Java containers, but unikernel performance has dropped precipitously. Despite all unikernels performing badly in this case, there are some notable distinctions which are more obvious when compared with the single threaded results.

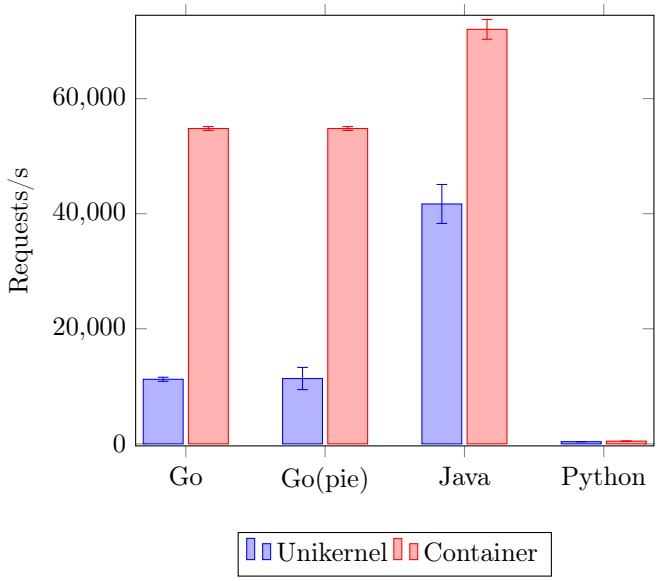


Figure 2.6: REST service performance of multi-threaded unikernels versus multi-threaded containers

When directly comparing the multi-threaded and single-threaded results (Fig. 2.7), only Java shows a performance increase with an increasing number of cores. Even in that case, the only return for quadrupling processing power is 60% more requests per second. Python is relatively unaffected by adding more cores, with multi-threaded performance dipping 3% below single-threaded performance. Go seems to actually suffer a great deal from expanding the thread pool, only managing 75% of its single core performance. It was verified that this is not an effect of Go simply starting too many threads and drowning the scheduler, since Go fetches hardware information and starts as many threads as there are cores [50]. The possibility of XenServer causing these scaling issues was considered, but related work has shown that this is not the case [52], at least in instances where the num-

ber of physical cores equals the number of virtual cores and the number of threads is not (much) higher than either, which is true here.

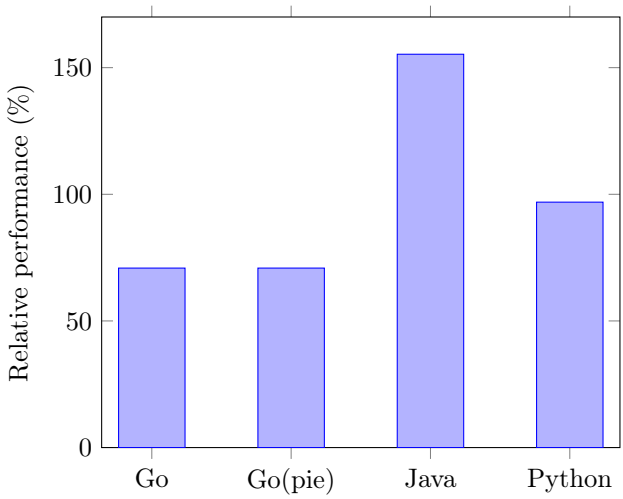


Figure 2.7: REST service performance of multi-threaded unikernels relative to single-threaded unikernels

It should be noted that in all cases the combined CPU load of all cores was between 40 and 50% during testing, indicating that a lot of cycles were being wasted on simply getting threads to run at all, and not enough on actually running them. This shows that while OSv obviously has very good single threaded performance, any applications using multiple worker threads should probably be split up until threading performance has been stabilized.

Request latency

Part of the requirements for the smooth operation of microservices is having a predictable, stable and preferably low response time. In this section, the results from the stress tests are examined in a different way to see if this is the case for unikernels. Values over the 98th percentile and below the 2nd percentile have been removed to avoid noise from distorting the scope of the charts. All statistics are based on one million requests to their respective web service.

Fig. 2.8 shows the response times for both Go and Java. A (UK) suffix indicates response times for a unikernel while (C) is the container version. While the Go unikernel seems to have a slightly higher median response time than its container counterpart, the maximum response time of the container version is almost 10 times higher. Taking the performance results from the previous sections into account, it's obvious that these numbers are not a

problem for unikernel performance. Java, for its part, performs equally well in both cases.

The results for Python were not included in the chart because their range would make the other results hard to interpret. However, the Python unikernel seems to perform considerably better than the container version, with a median response time of 100ms versus 111ms, respectively. Additionally, Python's maximum response times are much better when running as a unikernel with 105ms versus 140ms for the container.

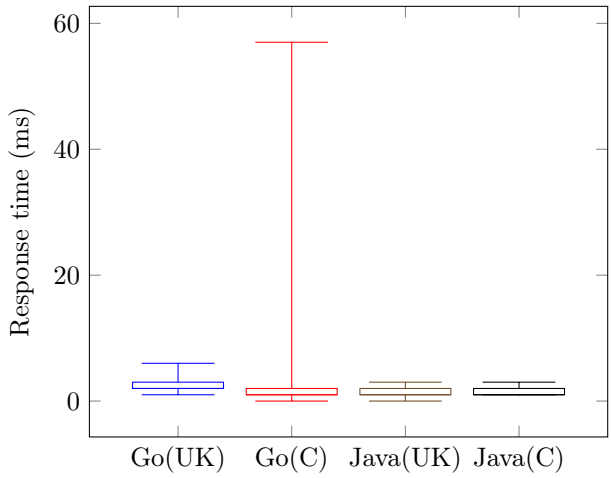


Figure 2.8: REST service response time overview for unikernels and containers

The response times for multi-threaded applications show rather interesting numbers that give some food for thought for the bad scaling performance. Fig. 2.9 shows the response latencies for both single- and multi-threaded Go unikernels. The response times have been gathered in categories 50ms wide, with the only exceptions being the 0-5ms and >450ms categories. In the single threaded case, the number of responses falls off exponentially with response time, with the maximum response time being 63ms. In the multi-threaded case however, the curve flattens around 100ms and about 0.6% of the requests take a much longer time to complete. 0.12% of the requests even take between 450 and 1000ms to complete. Despite the multi-threaded program actually handling about 99% of all requests slightly faster than the single threaded program, the fact that a small percentage of all requests is held up for a long time makes it slower overall. The only explanation for this is that while the scheduler handles the large majority of threads quickly and correctly, some thread switches are made to wait an exceptionally long time before being able to run their thread and complete their workload.

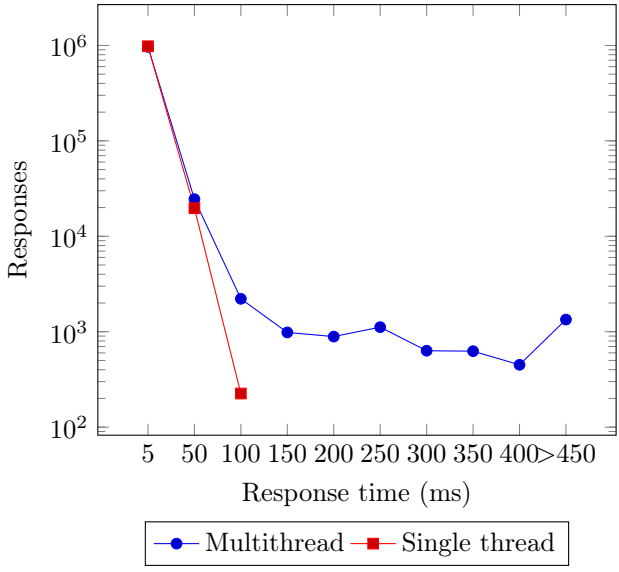


Figure 2.9: Number of REST service responses per response time category for single- versus multi-threaded Go unikernels

Memory footprint

Using the REST service images from the stress test, a crude comparison can be made between the memory consumption of unikernels versus containers for the different languages.

Container memory footprint was queried directly from Docker. For unikernels, memory footprint was measured by taking the number of actually allocated pages as reported by VirtualBox for each machine and multiplying it by page size. VirtualBox 5.2 was used for this purpose instead of the already deployed unikernels on XenServer because it was easier to measure memory footprint at any given time using VirtualBox.

Memory footprint was only examined for the single threaded version of unikernels and containers. Measurements were taken after starting an instance and executing a single request to make sure all libraries and variables were initialized. Note that after thousands of requests, memory footprint could be higher than the numbers presented, but would eventually go down again after garbage collection.

Fig. 2.10 shows that Java unikernels use over twice as much memory than containers, Python unikernels use up to 6 times more memory, and Go unikernels require as much as 30 times more memory. This makes sense, since containers run on top of a host kernel and only need to load their programs into memory. Unikernels, on the other hand, have some memory

overhead because of their built-in kernel, no matter how small it may be. The huge difference for Go can be explained by the fact that the program requires no interpreter or VM like Python and Java programs do, so the container version has a minimal memory footprint. This makes the unikernel version, which only requires 70MB more memory in absolute numbers, look comparably huge.

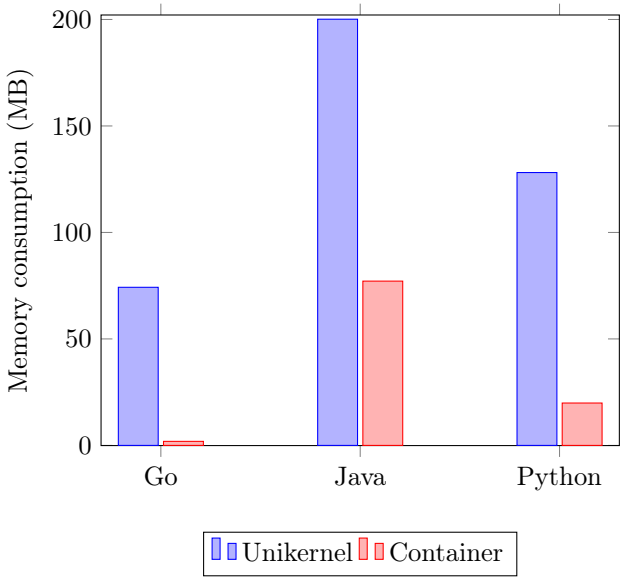


Figure 2.10: Container versus unikernel instance memory consumption

In absolute terms, the extra memory required to run a program as a unikernel is 70MB to 130MB, depending on programming language. One redeeming quality of unikernels is the fact that they can run on a type I hypervisor, eliminating the need for an operating system that could potentially consume a large amount of memory by itself. This means that unikernels are a viable alternative to containers where small to medium numbers of unikernels are concerned, simplistically represented by:

$$M_{hv} + N_{uk}M_{uk} < M_{os} + N_cM_c \tag{2.1}$$

Where M_{hv} is the memory requirement of the hypervisor, M_{os} is the memory requirement of the operating system to be replaced, N_{uk} and N_c represent the number of unikernels and containers respectively and M_{uk} and M_c represent memory requirement per unikernel and container, respectively. Note that the number of unikernels does not always equal the number of contain-

ers, since one container may have to be split up into several unikernels due to threading or multi-processing concerns.

Stability issues
As discussed before, unikernels under UniK and rumprun had serious stability problems, despite being otherwise fully functional. OSv on the other hand is a stable platform, but there were some quirks:

- When compiling Go as PIE executables, REST services crashed once every 2 to 10 million requests. This was not a huge problem for testing since it was done in batches, but it is something to look out for when planning to use unikernels in any type of production environment. Luckily, the tests have also shown that OSv's wrapper for Go, which is stable, has nearly identical performance, so it would be preferable to run Go that way.
- All OSv unikernels tend to crash every few million requests once they have been multi-core enabled. They did not so much cause errors, but simply hung or stopped without further explanation. Since multi-core performance under OSv proved to be worse than simply instancing several unikernels, this is not much of an issue either.

Updated results
This section provides updated benchmark results for new software versions available in early 2022, along with results for Firecracker microVMs, thus representing nearly 4 years of improvements since the original benchmarks. The benchmarked versions are OSv 0.56, Docker 20.10.3, and Firecracker 0.25.2, running on Ubuntu 20.04. For Firecracker, a proxy is used to forward HTTP requests arriving at the host interface to the microVM interface using socat[5]. As this may introduce an undue CPU overhead and network latency, the effects are monitored to ensure a valid comparison to containers and unikernels. Whereas the unikernels in the original benchmarks were somewhat difficult to create and not always stable, the OSv toolchain has been significantly improved. The process of creating unikernels is more straightforward, and the resulting unikernels boot and operate reliably. Similarly, Firecracker seems to be based on best practices learned by competing technologies in the last few years, and the toolchain allows the easy creation of reliable microVMs.
The benchmarks are not run on the same machine as the original ones; the machine used for these tests has an Intel Core i5 3570k processor with 8GiB of memory, and a 120GiB ADATA SSD. As this makes the results not

[5] https://linux.die.net/man/1/socat

directly comparable to the older benchmarks, the opportunity was taken to update the test scenarios slightly, and the REST service now has an additional POST method that creates a "todo" item in-memory. As such, both requesting and creating "todo" items is benchmarked. Simultaneously, the benchmarks are limited to Golang in order to present a clear comparison between various runtimes and scenarios.

Fig. 2.11 shows the results of the REST stress test. While it may be expected that the memory manipulation involved in creating a new "todo" item is slower than a straightforward array lookup, the unikernels seem to suffer from a higher relative penalty than containers and Firecracker. However, this is a minor problem for the OSv unikernels, which are around 80% faster than equivalent Docker containers, and 160% faster than Firecracker microVMs. The significant performance delta between unikernels and Firecracker, despite both being essentially microVMs, is likely largely due to the use of XenServer and KVM, respectively. Whereas XenServer runs the highly optimized OSv drivers close to bare metal, the drivers in the Firecracker microVMs have to contend with the various layers of KVM and the Linux kernel.

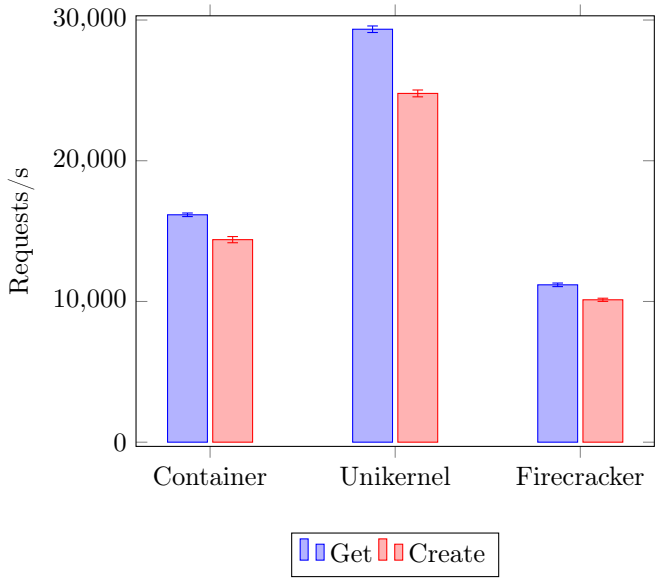


Figure 2.11: REST stress test performance evaluation of unikernels, containers and Firecracker microVMs

Fig. 2.12 shows the performance of the various runtimes for Bubble sort. As the error bars indicate, the runtimes are almost equally matched, indicating

that the effects from Fig. 2.11 are likely entirely due to virtual driver efficiency, choice of hypervisor, and network stack architecture.

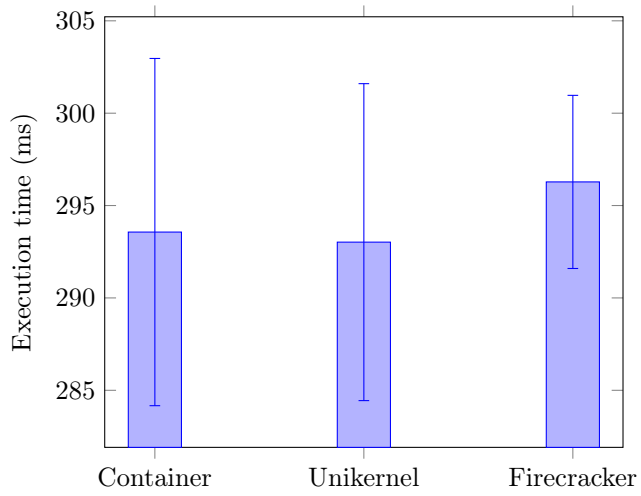


Figure 2.12: Bubble sort execution time of unikernels, containers and Firecracker microVMs

The relative performance of multi-threaded containers and microVMs is shown in Fig. 2.13. Container versions are shown to scale almost perfectly, at 340% to 380% depending on the type of workload. While the performance issues of OSv unikernels are expected, the benchmarks reveal a significant difference between the scalability of specific operations. Paradoxically, simply requesting "todo" items is significantly slower than creating them, which is merely 14% slower than in a single-threaded unikernel. Firecracker covers the middle ground; when provided with four CPUs, performance improves by around 220%. This effect can not be attributed to the overhead of socat; Firecracker simply did not use all of the CPU power it was assigned, and an increase in HTTP requests merely results in higher latencies. Note that OSv unikernels exhibit a similar behavior, as the multi-threaded unikernel used only 16-23% of its assigned CPU power.
Latency distributions are shown in Fig. 2.14, indicating that OSv unikernels and Firecracker microVMs are closely matched in response time distributions, and they are both more stable than containers, at the cost of slightly higher median latency.
Finally, Fig. 2.15 shows the image size and memory use of the various alternatives. As shown in the original results, containers have virtually no overhead in image size when using a scratch base image, and very little memory overhead (i.e. the Docker shim, which was not counted in the

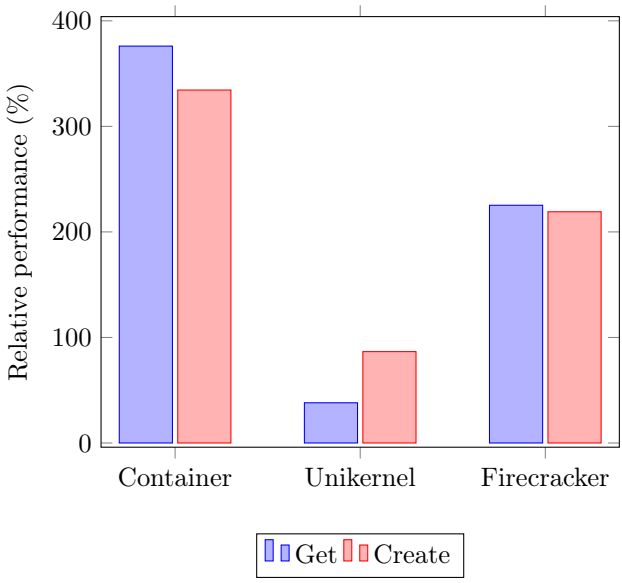


Figure 2.13: Relative performance of multi-threaded containers, unikernels and Firecracker microVMs

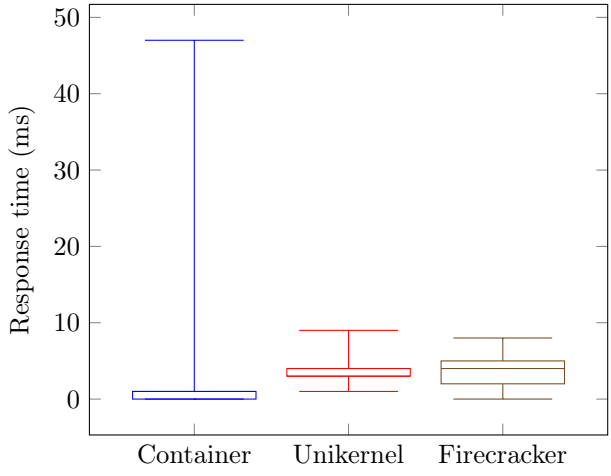


Figure 2.14: REST service response time overview for unikernels, containers and Firecracker microVMs

original results). Firecracker, on the other hand, has the biggest overhead in terms of both image size and memory. This is likely due to the default kernel used to create the images, whereas OSv uses a modular, completely customizable kernel. The latter falls between containers and Firecracker in terms of image size overhead, but neither VirtualBox nor XenServer could properly determine its memory use. Assuming a memory overhead similar to that in the original benchmarks, Firecracker and OSv unikernels have almost identical memory requirements.

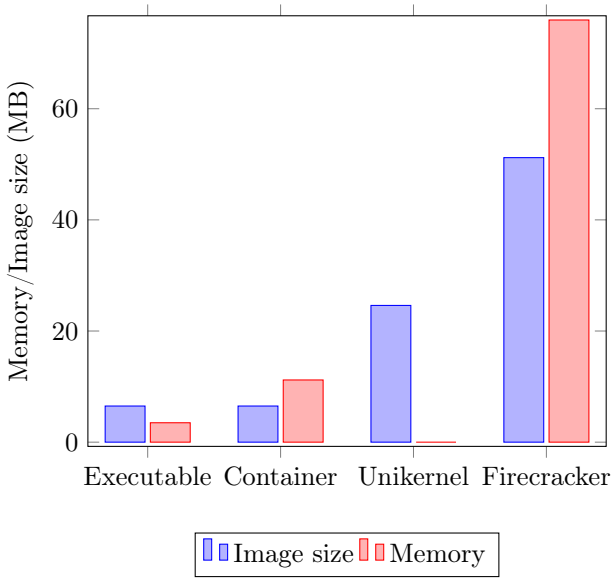


Figure 2.15: Image size and instance memory consumption of containers, unikernels and Firecracker microVMs

2.2.3.3 Discussion

Out of the considered unikernel platforms, OSv is the only stable one for the tested scenarios. It supports (among others) C++, Go, Python and Java, albeit with some small changes to OSv's source code in the case of Python. Other platforms were either unstable during testing and/or were less user friendly, either because it was difficult to create usable images in the required formats or because the images did not boot correctly without certain other software in place (e.g. UniK daemon).
Unikernels exceed containers in terms of pure speed and response time, firmly surpassing them for single core performance on a type I hypervisor. In the evaluations, Java and Python unikernels performed 16% better

than their respective containers for a REST service stress test, and Go even performed 38% better. When the focus shifts to heavy workloads, all unikernels keep an equal pace with their container counterparts, apart from the Python unikernel which only manages 50% of the equivalent container's performance. Unikernel performance for the REST service stress test can be explained by the fact that this test relied heavily on kernel functions and thus context switches from user space to kernel space. These do not exist in a unikernel, giving unikernels a large advantage over containers in situations where context switches happen very often (REST service), but breaking even in situations where they almost never occur (heavy workload). Another contributing factor is the heavier use of device drivers in the REST service stress test, which are less complex in unikernels.
Unikernels are far from ready for multi-threading, but that should not be a problem for cases where software can either be split up or multiple instances can be deployed (REST services, modular software) or where multi-threading is not really required (embedded software). In these cases converting software to unikernels could still be a major advantage.
Concerning memory, unikernels consume a good deal more than containers. This makes sense, since unikernels have the extra overhead of a kernel, while containers use the kernel of a host OS. However, since unikernels do not need a full OS to run on in the first place, memory consumption of a hypervisor with a small number of unikernels may be less than that of a large OS running a few containers. The result is that memory consumption is an important factor to consider when deciding whether to use unikernels. Adding more instances will give better performance than adding the same CPU power to one or more containers running the same software, but the trade-off is that the unikernels will use more memory.
In terms of unikernel performance of specific programming languages, Java and Go are the clear winners. Java gives by far the best performance of all, while Go uses the least amount of memory. For new software, this can be useful information when choosing a language for unikernel development. For existing Java or Go software being ported to unikernels, switching to the other language might not be worth the effort required to convert it, unless either speed or memory footprint are absolutely critical.

2.3 Networks

Although network virtualization is also effected through software, its nature is fundamentally different from process virtualization as it concerns data streams rather than processing. Most of the effort in this area is focused on merging and managing networks into uniform, secure environments in

order to simplify the deployment and scaling of (virtualized) software processes. Additionally, because network virtualization is handled by software, its flexibility also gives rise to better network monitoring, routing and reliability in general. Like software virtualization through containers, the basis of network virtualization is enabled by kernel features (e.g. virtual network devices and routing), aided by hardware acceleration for secure connections.

2.3.1 Virtual Private Networks

Virtual Private Networks (VPNs) are an old and widely used technology, widely used to form secure (e.g. private) virtual networks on top of hardware network infrastructure, for example to enable employee access to sensitive company data at home, or to form a virtual Local Area Network (LAN) over the internet for older multiplayer games to run on. Generally, VPN software achieves its goals by creating its own virtual network interface that redirects to a hardware network interface, assigning an IP address from a single logical pool of addresses used by the VPN. Any traffic through the virtual interface is encrypted and encapsulated by the VPN software before redirecting it to the hardware interface, securing and isolating the data.

Recent state of the art studies appear to be non-existent, but older ones are still informative [53]. Some studies deal with the security aspects of a VPN [54], while many others focus on the throughput performance of VPNs [55, 56]. However, no studies seem to compare OpenVPN [57] to newer VPNs such as WireGuard [58].

While studies exist on using overlay networks in osmotic computing [59], they deal mostly with container network overlays such as Flannel and Weave [60] which are integrated into Kubernetes. Others present a custom framework, for example Hybrid Fog and Cloud Interconnection Framework [61], which also gives a good overview of the challenges of connecting edge and cloud networks. To the best of our knowledge, no work exists on explicitly using a VPN as an overlay network to connect edge and cloud networks, nor is there existing work on testing the scalability of a VPN to do so.

In this section, an overview is given of all the VPN software chosen for this paper. For each VPN solution, a Docker container image was created so it could be launched from a Kubernetes pod. As much default configuration was used as possible. The code and configuration for the Docker container images is made available on Github[6].

[6] https://github.com/drake7707/secure-container-network

2.3.1.1 OpenVPN

OpenVPN [57] is a widely used VPN solution, first released in May 2001. It is still under active development, with version 2.4.7 being released as of February 2019. For the tests in this study, version 2.4.6 was used. It uses open encryption standards and offers a wide range of options, while being able to use both UDP and TCP for its transport layer. A weak point of OpenVPN is that it is single threaded and thus entirely dependent on the speed of a single processor core, no matter how powerful a system is.

2.3.1.2 WireGuard

WireGuard [58] is a relatively new VPN solution. It does not have an official stable release yet, and is still under heavy development. Its current version is v0.0.20190406, but the version used for the tests is 0.0.20180613. As of March 2019, it has been sent out for review several times, aiming to be mainlined into the Linux 5.2 kernel [62]. A main feature of WireGuard is that it is designed to be non-chatty, only sending messages when required. Note that in this paper WireGuard-go [63] is used, which is a Golang implementation of WireGuard. Performance may differ from the main WireGuard version, especially if it is accepted into the Linux kernel.

2.3.1.3 ZeroTier

ZeroTier [64] is an established VPN solution developed by ZeroTier, Inc. First developed in 2011, it is currently at version 1.2.12, which is also used for the tests. ZeroTier is still under active development, and has both paid and free solutions. The "Zero" part of its name comes from the fact that it requires zero configuration by default. This is achieved by having a number of root servers, called the "Earth" and managed by ZeroTier, that fulfil a similar role as DNS servers to the ZeroTier network. However, users can also define their own root servers using "Moons" in order to decrease dependency on the ZeroTier cloud infrastructure and improve performance. Its design also gives ZeroTier the advantage that no endpoints need to be publicly available, as long as they are still accessible through some public IP address via NAT.

2.3.1.4 Tinc

Tinc [65] is a VPN solution that predates even OpenVPN, with an initial release in November 1998. The current version, 1.0.35, was released in October 2018, so it is still under active development. Version 1.0.35 is also the one used for the tests. Like OpenVPN, it relies heavily on open

standards. However, unlike OpenVPN, it has full mesh routing by default, which can make it more efficient in large networks with large amounts of client-to-client traffic.

2.3.1.5 SoftEther VPN

SoftEther VPN [66] is a relatively new VPN solution. Its first release dates from January 2014, with the latest release being version 4.29 as of February 2019.
SoftEther is a multi-protocol VPN, with modules for OpenVPN, L2TP/IPSec, MS-SSTP and its own SoftEther VPN protocol. By default it uses the SoftEther VPN protocol, which emulates Ethernet over HTTPS. The advantage of using HTTPS to tunnel VPN traffic is that it makes it easier to bypass firewalls, since HTTPS ports are often freely accessible.
Other than being multi-protocol, SoftEther has a wide range of features, including a high availability setup for its server endpoints.

2.3.2 Network Function Virtualization and Software Defined Networks

The functionality of VPNs can be generalized with the use of Software Defined Networks (SDN) [67] and Network Function Virtualization (NFV) for a more modular and flexible approach to network virtualization.
SDN is a technique by which the control of a network is separated from its physical infrastructure and the data sent through it, enabling programmatic control of (logical) networks and abstracting the underlying infrastructure from the view of applications. As such, SDN can assist in creating a highly dynamic and configurable VPN. As a more modern application, the infrastructure for container networks generally consists of an overlay network (or SDN overlay), which manages tens to hundreds of virtual interfaces and their respective routes on the server hosting the containers. Such SDN overlays can also extend to other nodes, forming a cluster-wide logical network.
NFV, on the other hand, can be used to transfer networking functions from dedicated devices (e.g. switches, routers) to generic devices (e.g. servers, fog nodes), allowing for advanced control over specific functions. For example, as the entire network is software-managed and any imaginable metric can be calculated and logged, much of the information required for intelligent service discovery, DNS, intrusion detection and traffic routing is available for use with NFV. The use of (stateful) containers and VMs allows for seamless updates of functionality, while such flexibility or even the functionality itself would be difficult or impossible to achieve with firmware on dedicated hardware.

2.3.3 VPN for secure container networks

This section contains the edited version of the following publication: **"Scalability evaluation of VPN technologies for secure container networking"**, T. Goethals, D. Kerkhove, B. Volckaert, F. De Turck
published in **2019 15th International Conference on Network and Service Management (CNSM), 2019, pp. 1-7** [68]

In recent years, containers have quickly gained popularity as a lightweight virtualization alternative to VMs, their main advantages being limited resource requirements and fast spin-up times [69]. Simultaneously, edge devices used in IoT applications have become powerful enough to smoothly run containers and microservices, while remaining small and flexible enough to be used almost anywhere. This has triggered a trend of deploying containerized applications to edge devices, taking advantage of both centralized control and geographically widespread devices for a more efficient distribution of workloads [70]. To aid with the deployment and operation of containers on edge devices, a Virtual Private Network (VPN) can be useful, for the following reasons:

- Securing communication between nodes becomes more important when leaving the confines of the cloud. While the connections between orchestrator nodes and service endpoints are often secured in various ways by default, it can not be assumed that this is always the case. A VPN can provide a base layer of security for all communications, ideally with little to no performance overhead.

- Unlike cloud infrastructure, networks containing edge devices are not usually well organized and homogeneous. This means the network could be hidden behind a router, node IP addresses are not predictable, existing port mappings could interfere with container requirements, etc. While technologies such as UPnP [71] can solve some of these problems, they can also introduce new security problems. It would be better to avoid the problems with edge networks altogether by using a VPN. In the cloud, only the VPN server needs to be publicly available. This makes it possible to hide critical services from public scrutiny, while still allowing edge devices in the VPN to access them over a secure connection. In short, a VPN can ensure a homogeneous networking environment in which both the container orchestrator and deployed containers can allocate required ports, assign IP addresses and modify routing tables without interference from other devices or programs.

Modern clusters consist of thousands of nodes on which containers can be deployed [72]. Therefore, in order to build a cluster using a VPN, any suitable VPN software must be able to handle a large number of simultaneous connections and packets with minimal performance degradation. The goal of this paper is to evaluate recent VPN software for its ability to scale with the size of the cluster using it as an overlay network, while also examining the effects of network degradation on communication between nodes in a cluster connected through a VPN[7]. While some of the conclusions are valid for VPN networks in general, the tests are primarily aimed at edge networks, with a lot of devices generating IoT traffic consisting of very small packets.
Many studies exist on the concept of shifting workloads between the cloud and edge hardware. This trend began with edge offloading [73] and cloud offloading [74], evolving into osmotic computing [70]. Several container deployment strategies exist, from simple but effective resource requests and grants [75], through network-aware fog scheduling [76], to using deep learning to constantly adjust deployments [77]. Some studies focus particularly on security for osmotic computing and end-to-end security between the edge and the cloud [78, 79].

2.3.3.1 Benchmarking setup

The tests are performed on the IDLab Virtual Wall installation, reserving machines with identical hardware and in the same geographical location for each test. Each machine has two Quad core Intel E5520 processors at 2.2GHz and 12GiB RAM.
A total of 8 machines are provisioned with Ubuntu 16.04 and Docker 18.06. Using these machines, a Kubernetes v1.11 cluster is created with 1 master node and 7 worker nodes as a basis for each test. Kubernetes is used to easily distribute a large number of VPN client containers over the entire cluster to simulate a VPN network with many independent clients. The Kubernetes version should not have any impact on the results up to and including v1.14. Observations during testing showed that deploying over 120 VPN client containers on a single node results in errors because the container processes fail to allocate all required resources. However, when using Kubernetes this is not a problem, since it has a built-in limit of 110 pods per node by default. Therefore, the VPN containers deployed by Kubernetes for the tests will not run into the observed problems with overdeployment.
All tests follow the same basic premise, using pods that contain a single

[7] More recent developments have made the use of a VPN obsolete for this use case. CNI plugins exist that can explicitly tunnel between private networks, e.g. EdgeVPN.io https://edgevpn.io/

container with a VPN client and a configurable script. A number of pods are deployed to the nodes in the cluster by creating a new deployment in Kubernetes, as shown in Fig. 2.16. The actual number of deployed pods varies depending on the test. Once a VPN connection is established, the script in the pod starts calling a REST service located on the master node using curl, every 250ms for 15 minutes. The REST request has no body, it is an empty GET request. The response is a simple "OK" with a 200 HTTP code. Using these short and static request/response messages ensures that the connectivity of all VPNs is tested rather than network throughput. This approach also has the advantage that it closely emulates IoT sensor traffic usually present in edge networks, where short bursts of sensor data are pushed to a central broker. The pods do not perform any processing of the results, rather they simply log the start timestamp, end timestamp and curl exit code of each REST call to a pod-named output file on a host-mounted network share to be processed later.
Because the goal of the tests is to measure VPN scalability with no other factors involved, the VPN server and REST service containers on the master node are started outside Kubernetes. This ensures that none of the VPN traffic goes through the Kubernetes pod network, but rather that it travels directly between VPN endpoints.
To ensure only valid data was processed, the data was filtered both at the start and the end of each test:

- To remove data generated before all pods were running, only data recorded after the latest first timestamp of all pod output files is used.
- To remove data generated after the test is shut down, the last string of failures at the end of each pod output file is removed. In theory, this could remove some valid failures, but the total number of calls for each pod is large enough that it should not make any meaningful difference.

VPN Software settings

- For OpenVPN, the standard configuration is used. Traffic is encapsulated in UDP packets over a TUN device.
- For WireGuard, the standard configuration is used. Some extra work is required to set up interfaces and routing rules, which is done by default in other software such as OpenVPN and ZeroTier.

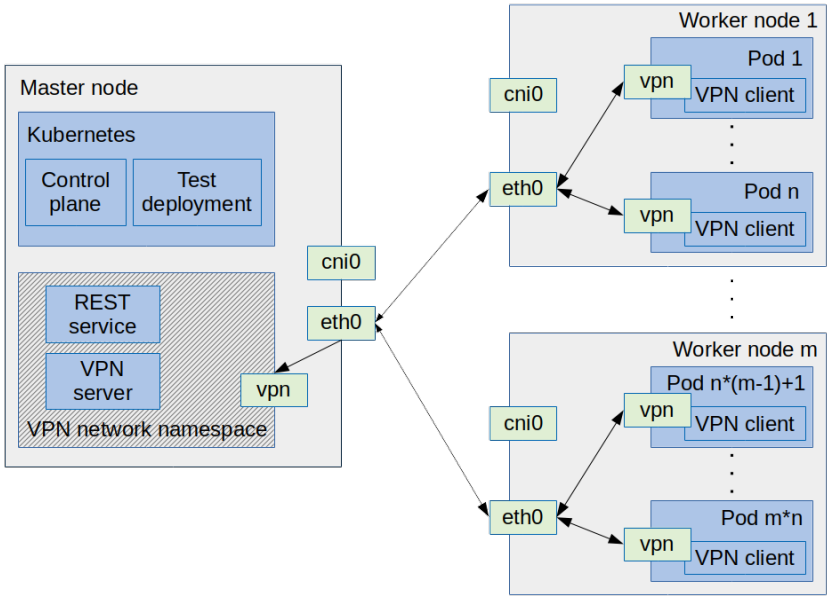


Figure 2.16: Overview of the test setup with m nodes and n pods per node.

- For ZeroTier, a network controller was set up using a script to simplify the creation of VPN clients through scripts. While there is a minimal reliance on ZeroTier cloud infrastructure, the Earth root servers are still used to look up the address of the test network controller by its ZeroTier name.
- For Tinc, standard configuration was used. Like OpenVPN, transport uses UDP via a TUN device.

Scalability test

The scalability test aims to determine how well VPN software can handle large amounts of connections and service calls. To achieve this, a number of pods are deployed via Kubernetes that perform REST calls as described above. The number of deployed pods varies from 50 to 750, in steps of 100. The results from the test are aggregated and analyzed to determine how the response times and failure rates (error codes) vary with an increasing amount of pods.

Network degradation test

In the real world there is always some amount of packet loss and latency,

especially when dealing with the edge and IoT devices. Therefore, it is important that any VPN can remain functional despite significant network degradation.
To determine the resilience of VPN software against network degradation, both packet loss and latency are examined independently. Both conditions are imposed through NetEM on the host network interface. Because the degradation affects all traffic including VPN control messages, this method gives a good indication of how well a VPN can keep working under any set of network conditions.
This test is performed with 100 pods distributed evenly over the 7 worker nodes. Latency is varied from 0 to 5000ms in 1000ms jumps. Out of practical considerations, any requests taking longer than 10 seconds are terminated and considered a failure. This influences the failure rate to some degree, but since the same rule is applied across all tested VPNs, the comparison is still valid. Packet loss is varied from 0% to 100%. While this range may seem extreme, it is interesting to see how VPNs react to the full spectrum of packet loss.

2.3.3.2 Results

In this section the results from the tests are presented. To keep this section organized, the results for the scalability tests have been divided into subsections for response time and failure rate. Similarly, the network degradation tests have been split into subsections for latency and packet loss.
Note that for all charts with whiskers, the whiskers indicate the 25th and 75th percentiles of the results, while the data points represent the median cases. The 75th percentile is used for practical reasons; higher percentiles produce extremely large whisker ranges, making it harder to interpret the charts. The 25th percentile is chosen for symmetry; the difference between the 1st and 25th percentile is far less significant than between the 75th and 100th.

Scalability - response time
Fig. 2.17 shows the response times of the scalability test on a logarithmic scale. For the same reason, both charts have been truncated at 650 pods instead of 750. A striking observation is that the response times of both Tinc and SoftEther increase by an order of magnitude between 250 and 350 clients, making them over 50 times higher than those of other VPNs. At less than 350 clients, Tinc is already being outpaced by OpenVPN, but is still in the same order of magnitude. SoftEther starts out with response times twice as high as those of OpenVPN, and ends up with response times almost 150 times higher than those of OpenVPN at 650 clients. It is likely that

some common feature or design decision of Tinc and SoftEther is causing this performance problem.
OpenVPN, WireGuard and ZeroTier are evenly matched for 450 VPN clients or less. Even the 25th to 75th percentile ranges are similar, though the highest ZeroTier response times start to go up quickly around 250 clients. For more than 450 clients, a few trends become clear that are useful for large scale edge networks. First, increasing VPN traffic does not seem to affect WireGuard response times at all. Even the results at 650 clients are still in line with those at 50 clients. OpenVPN response times shift from a slow, linear increase to a quadratic curve. ZeroTier is on a solid quadratic curve from the start, while the slowest responses quickly escape the chart altogether. In short, WireGuard is the clear winner of this test, while OpenVPN is a close second. However, if the test were to continue to thousands of clients, WireGuard may become the only useful choice.

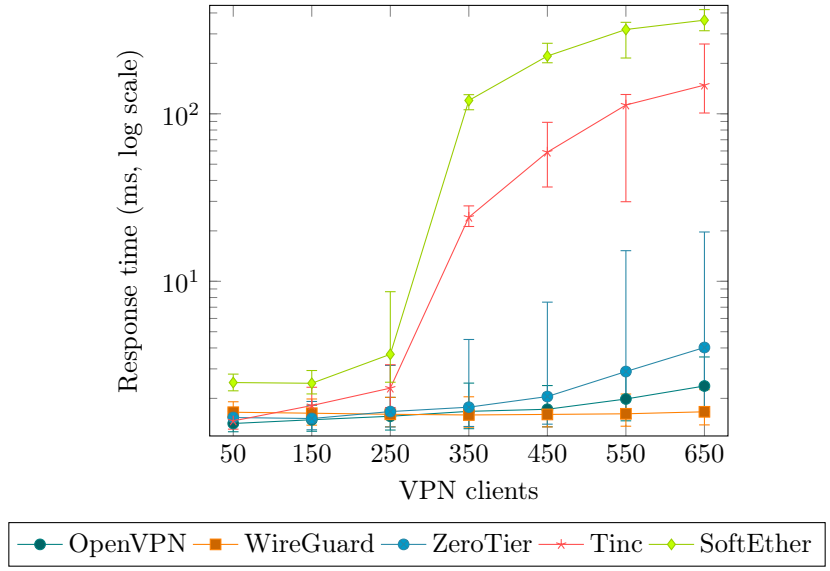


Figure 2.17: Evolution of response time for an increasing number of VPN clients, represented on a logarithmic scale.

Scalability - failure rate

Fig. 2.18 shows the failure rates of the scalability test. The results are strikingly similar to the REST response times.
Tinc failure rates are an order of magnitude above those of other VPNs, but the curve is more pronounced than in the response times chart. Even with 50 clients, Tinc failure rate is 10 times higher than that of OpenVPN

and ZeroTier. It is possible that Tinc is more suited to high throughput applications in smaller networks, but that topic is outside the scope of this paper. SoftEther, while not having excellent results, stays close to OpenVPN performance until its failure rates start to increase quickly around 450 clients.
As the rest of the VPNs go, WireGuard has the best results, showing little to no increase in failed requests as VPN traffic and the number of clients increases. OpenVPN appears to follow a linear trend, with ZeroTier following closely until its performance once again degrades according to a quadratic curve around 450 deployed clients.

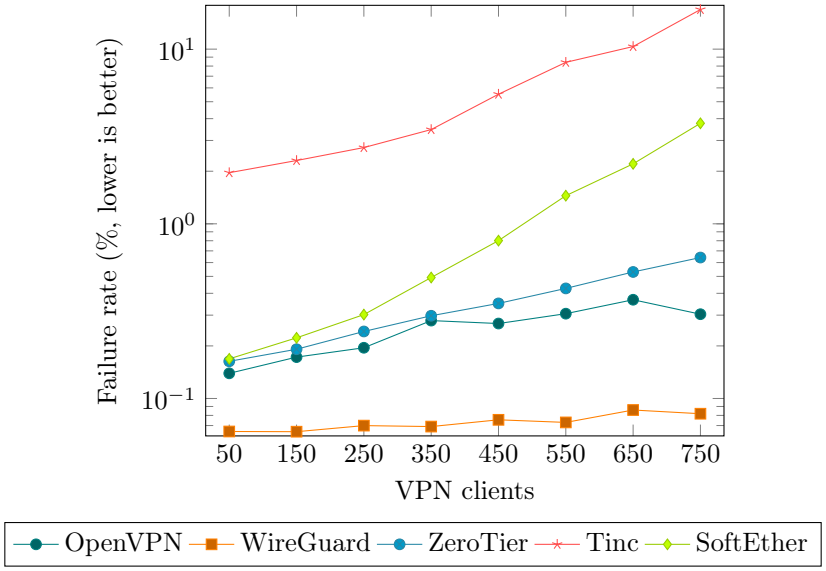


Figure 2.18: Evolution of failure rate for an increasing number of VPN clients.

Network degradation - packet loss

Fig. 2.19 shows the response time for REST requests under each VPN, for increasing amounts of packet loss. All the results in this chart are very closely matched, with even the 25th to 75th percentile ranges lining up in most cases. The results only start to differentiate for 70% packet loss or more. WireGuard performance degrades heavily when going from 90% to 100% packet loss, seemingly making it the slowest VPN in this test. Tinc appears to win this test, with the lowest median time and a relatively small range for its results.
Considering the invariance of the results at normal amounts of packet loss, 0% to 20%, it stands to reason that increasing the number of clients within

this range will yield the exact same results as Fig. 2.17 through Fig. 2.18 have shown. However, more tests are required to properly confirm this.

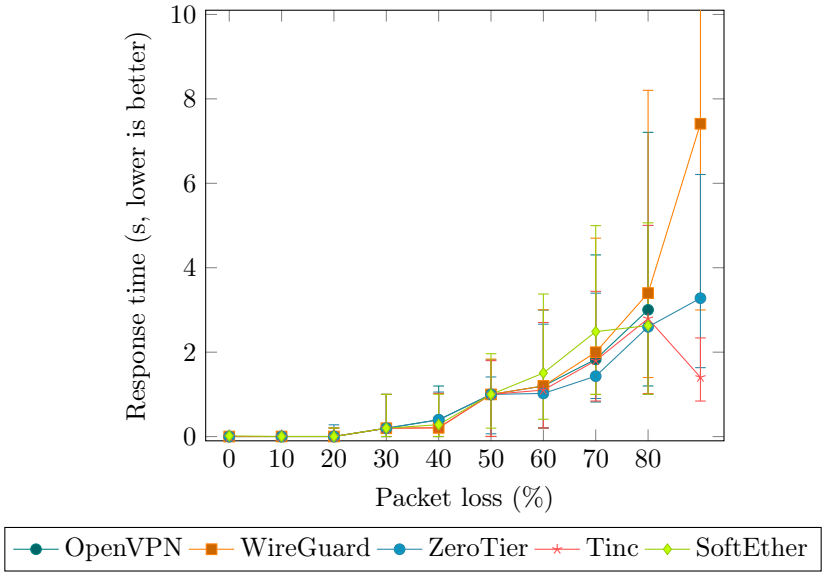


Figure 2.19: Response time of REST calls from 100 VPN clients for increasing packet loss

OpenVPN and SoftEther results cannot be shown beyond the 80% packet loss category, the reason for which becomes clear when looking at Fig. 2.20. This chart shows the failure rates for the same packet loss test, and also suggests an alternate interpretation for some of the apparent effects in Fig. 2.19. While ZeroTier and WireGuard manage to keep pushing at least some traffic through until packet loss hits 100%, Tinc, SoftEther and OpenVPN failure rates increase much faster. Starting around 90% packet loss, OpenVPN and SoftEther both have a 100% failure rate. Tinc on the other hand, still has a 0.004% success rate, but the extremely low sample size makes the response time shown in Fig. 2.19 very unreliable. WireGuard and ZeroTier still have a success rate of about 10% to 20%, so that data is reliable.

While there is a lot of nuance in the results for 70% to 100% packet loss, the results for the lower end of 0% to 20% are easier to interpret. There is little difference between the different VPNs in this range, save for Tinc, which is known to have a higher failure rate from the results of the previous tests. There is no clear winner in this test; both WireGuard and ZeroTier are good options, with WireGuard having slightly higher response times but lower failure rates.

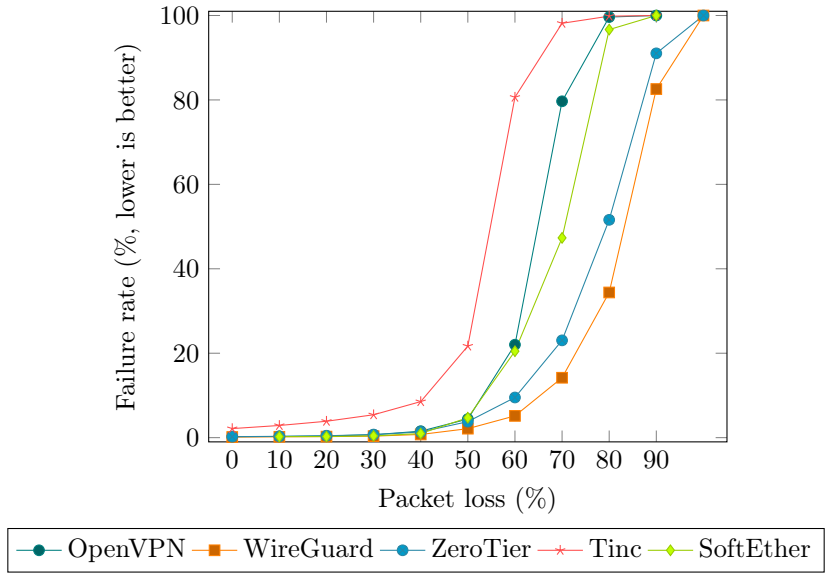


Figure 2.20: Failure rate of REST calls from 100 VPN clients for increasing packet loss

Network degradation - latency

Fig. 2.21 shows the failure rate for each VPN for increasing amounts of network latency.

WireGuard has the best failure rate, showing only a slow increase in failure rates as latency goes up. Even at 5000ms latency, still only around 3% of the requests fail. Note that due to the way the test was set up, WireGuard failure rate is 100% as soon as latency goes over 5000ms. OpenVPN mostly follows the performance curve of WireGuard, but consistently has a 4 to 5 times higher failure rate. ZeroTier and SoftEther are very close in performance. Their failure rates start out almost equal to those of WireGuard, but they increase faster as latency goes up. SoftEther failure rates shoot up to almost 50% at 4000ms, while ZeroTier manages to stay around 18%. Tinc, which by now has firmly established a somewhat high failure rate, degrades quickly once latency is higher than 1000ms, with a failure rate of 83% at 2000ms latency.

OpenVPN holds the middle ground between ZeroTier and Tinc, with error rates over twice as high as those of ZeroTier. OpenVPN shows slightly better performance than SoftEther at 4000ms latency, but at that point almost half of all requests fail under both OpenVPN and SoftEther.

As with the packet loss test, WireGuard comes out on top with very low failure rates, while ZeroTier is a close second.

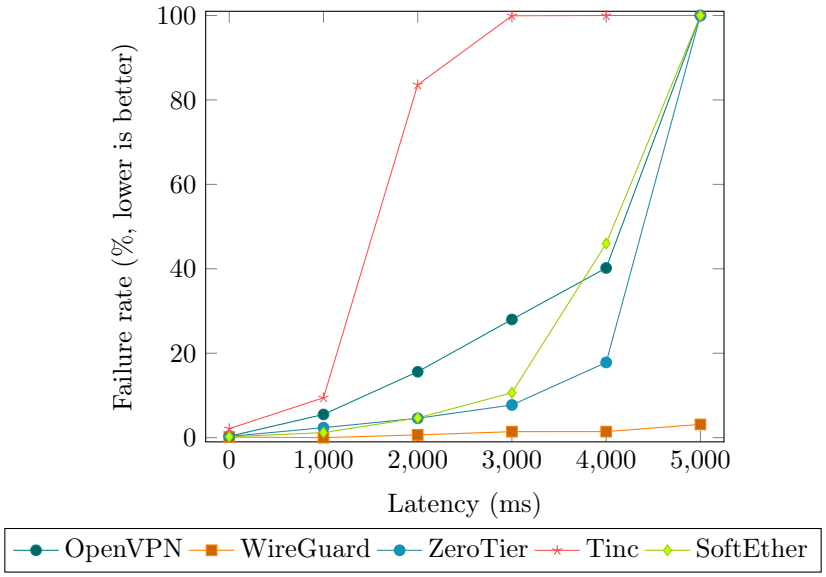


Figure 2.21: Failure rate of REST calls from 100 VPN clients for increasing latency

VPN overhead

Fig. 2.22 shows the VPN traffic overhead for both sent and received traffic. The first important observation here is that the overhead for all VPN software is within 10% of each other, meaning that this factor likely has very little influence on the results of the tests. The second observation is that traffic overhead is very high in all cases, with almost twice as much VPN overhead as actual container traffic. WireGuard, which showed excellent results in other tests, has one of the highest overheads.

While the results show that VPN traffic overhead likely has no effect on the performance of a VPN, they also show that most of the traffic is wasted on overhead when applied to large amounts of small packets. This can have a negative impact on microservice architectures where containers are deployed on edge devices to transceive IoT data.

2.3.3.3 Discussion

The results for WireGuard are the best across all tests. In some cases the results are remarkable, such as an almost unchanging response time and failure rate for REST requests with an increasing number of clients, where other VPN solutions tend to have quadratic scaling. In other cases, the difference is less pronounced. In the packet loss tests for example, the response times for requests over WireGuard are mostly on par with other

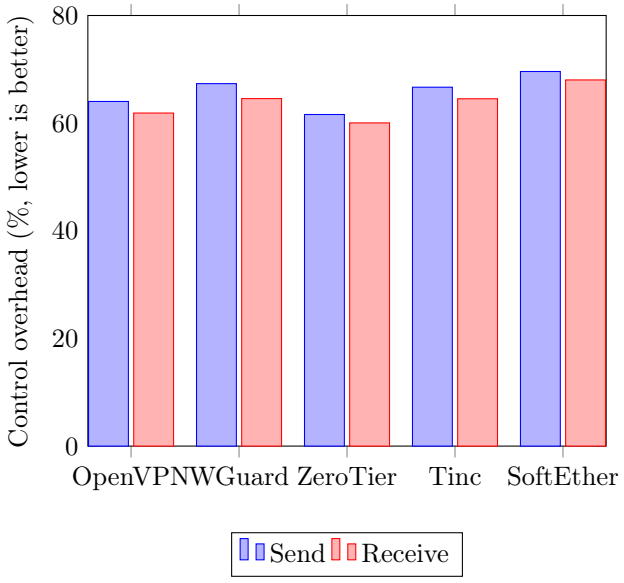


Figure 2.22: Relative overhead of VPN traffic with 100 VPN clients.

VPN setups, but the failure rates are about 10% to 30% lower than those for ZeroTier, which holds the second place in that scenario.

Tinc seems to be the least suited to the kind of setup used for the tests, having the slowest response time and highest failure rate with a large amount of clients, while also being most susceptible to network degradation.

The other solutions have varying results in all of the tests. In the scalability tests, OpenVPN holds second place, while ZeroTier comes in third. SoftEther is usually near Tinc in terms of performance. Looking at the network degradation tests, SoftEther and OpenVPN switch places while ZeroTier remains in second place.

For starters, much more VPN software exists than has been benchmarked. Adventurous researchers could attempt to map the scalability of all known VPN software, though this work will be hampered by a quickly changing landscape of possible candidates and versions.

While the results of this study provide insight on the high-level behavior of VPN software, a lot of the parameter space was deliberately not examined. Evaluating the full parameter space would be prohibitively time-consuming. However, there may still be interesting behavior to discover. For example, there is no way to predict how the results from the packet loss and latency tests would change with a varying number of VPN clients.

Furthermore, no attempt has been made to optimize the configuration of

each VPN to improve performance. A technical study which analyses VPN designs and configuration options could yield important insight into making a VPN that is ideal for use with mixed cloud-edge clusters.
It is possible that network topology and the number of VPN clients per node has a significant influence on the scalability of a VPN network. To confirm this, the tests would have to be repeated using hundreds of physical machines.
Another interesting topic is to see how a VPN network could be extended by using multiple VPN servers in a cluster. For example, SoftEther has a clustering function [80] that could be combined with a Kubernetes high availability setup to provide a suitable VPN network topology for a fault tolerant cluster.
Lastly, the results showed that up to 750 VPN clients is not a problem for WireGuard. Studies focusing on one specific VPN solution could be useful to determine its scalability limits.

2.4 Summary

In this chapter, the various types of virtualization technologies are introduced that will be used throughout the rest of this dissertation. The difference between process and network virtualization is explained, more specifically by discussing what and how they aim to virtualize.
A comparison of two popular types of process virtualization is presented, although more recent trends indicate that WASM may take over from unikernels as many unikernel initiatives are no longer actively developed.
Out of the considered unikernel platforms, OSv is the only stable one for the tested scenarios. It supports (among others) C++, Go, Python and Java, albeit with some small changes to OSv's source code in the case of Python. The benchmark results only concern OSv, so the conclusions may be different for other platforms.
The results show that unikernels can exceed containers in terms of performance and response time, depending on the programming language used. Java unikernels are significantly faster, while Python unikernels are actually significantly slower than equivalent containers. Potential factors contributing to unikernel performance and memory use are discussed, especially as the increased memory use may offset the benefits of faster computation in some scenarios.
On the side of network virtualization, various VPN alternatives are evaluated for their scalability in IoT edge networks, and to examine the effects of network degradation on VPNs used for this purpose. To that end, OpenVPN, WireGuard, ZeroTier, Tinc and SoftEther are set up with a default

configuration and subjected to a number of described benchmarks.
The results of the benchmarks show that WireGuard is the best VPN solution across all tests. Although the effects of different topologies and optimized configurations was not tested, the results seem to argue in favor of the continued adoption of WireGuard.

References

[1] C. Bormann, M. Ersue, A. Keranen, and C. Gomez. *Terminology for Constrained-Node Networks*, March 2022. Available from: https://www.ietf.org/id/draft-bormann-lwig-7228bis-08.html#name-classes-of-constrained-devi.

[2] P. J. Denning. *Virtual memory.* ACM Computing Surveys (CSUR), 2(3):153–189, 1970.

[3] K. Kolyshkin. *Virtualization in linux.* White paper, OpenVZ, 3(39):8, 2006.

[4] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. *Intel virtualization technology.* Computer, 38(5):48–56, 2005.

[5] R. Russell. *virtio: towards a de-facto standard for virtual I/O devices.* ACM SIGOPS Operating Systems Review, 42(5):95–103, 2008.

[6] P. Xu, S. Shi, and X. Chu. *Performance Evaluation of Deep Learning Tools in Docker Containers.* In 2017 3rd International Conference on Big Data Computing and Communications (BIGCOM), pages 395–403, 2017. doi:10.1109/BIGCOM.2017.32.

[7] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran. *Understanding Security Implications of Using Containers in the Cloud.* In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 313–319, Santa Clara, CA, July 2017. USENIX Association. Available from: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tak.

[8] P. Singh. *Exploring WSL2*, pages 75–98. Apress, Berkeley, CA, 2020. Available from: https://doi.org/10.1007/978-1-4842-6038-8_5, doi:10.1007/978-1-4842-6038-8_5.

[9] R. Pavlicek. *Unikernels: Beyond Containers to the Next Generation of Cloud.* O'Reilly Media, 2016. Available from: https://books.google.be/books?id=qfDXuQEACAAJ.

[10] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide. *A performance evaluation of unikernels.* In Technical Report. 2014.

[11] S. Wicki. *The Rumprun Unikernel, pkgsrcCon 2016.* Available from: https://www.pkgsrc.org/pkgsrcCon/2016/rumprun.pdf.

[12] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. *OSv—optimizing the operating system for virtual machines*. In 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), pages 61–72, 2014.

[13] M. Marks. *Unikernel Systems Joins Docker*, January 2016. Available from: https://blog.docker.com/2016/01/unikernel/.

[14] D. Williams and R. Koller. *Unikernel Monitors: Extending Minimalism Outside of the Box*. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16), Denver, CO, June 2016. USENIX Association. Available from: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams.

[15] B. Xavier, T. Ferreto, and L. Jersak. *Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform*. In 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 277–280, 2016. doi:10.1109/CCGrid.2016.86.

[16] M. Lucina. *Deploying real-world software today as unikernels on Xen with Rumprun*.

[17] V. Cozzolino, A. Y. Ding, and J. Ott. *FADES: Fine-Grained Edge Offloading with Unikernels*. In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems, HotConNet '17, page 36–41, New York, NY, USA, 2017. Association for Computing Machinery. Available from: https://doi.org/10.1145/3094405.3094412, doi:10.1145/3094405.3094412.

[18] S. R. Walli. *The POSIX family of standards*. StandardView, 3(1):11–17, 1995.

[19] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. *Jitsu: Just-In-Time Summoning of Unikernels*. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 559–573, Oakland, CA, May 2015. USENIX Association. Available from: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy.

[20] IncludeOS. *IncludeOS*, July 2018. Available from: http://www.includeos.org/.

[21] Xen and L. Foundation. *MirageOS*, July 2018. Available from: https://mirage.io/.

[22] C. Systems. *OSv*, July 2018. Available from: https://github.com/cloudius-systems/osv.

[23] *Rumprun*, July 2018. Available from: https://github.com/rumpkernel/rumprun.

[24] *WebAssembly*, October 2021. Available from: https://webassembly.org/.

[25] *WASI - The WebAssembly System Interface*, October 2021. Available from: https://wasi.dev/.

[26] C. Lattner and V. Adve. *The LLVM Compiler Framework and Infrastructure Tutorial.* In R. Eigenmann, Z. Li, and S. P. Midkiff, editors, Languages and Compilers for High Performance Computing, pages 15–16, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[27] E. Wen and G. Weber. *Wasmachine: Bring IoT up to Speed with A WebAssembly OS.* In 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pages 1–4, 2020. doi:10.1109/PerComWorkshops48775.2020.9156135.

[28] S. Murphy, L. Persaud, W. Martini, and B. Bosshard. *On the Use of Web Assembly in a Serverless Context.* In M. Paasivaara and P. Kruchten, editors, Agile Processes in Software Engineering and Extreme Programming – Workshops, pages 141–145, Cham, 2020. Springer International Publishing.

[29] T. Caraza-Harter and M. M. Swift. *Blending containers and virtual machines: a study of firecracker and gVisor.* In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pages 101–113, 2020.

[30] *KVM, Kernel-based Virtual Machine*, July 2018. Available from: https://www.linux-kvm.org/page/Main_Page.

[31] *Why Docker?* Available from: https://www.docker.com/why-docker.

[32] *containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability*, October 2021. Available from: https://containerd.io/.

[33] A. Randazzo and I. 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.

[34] R. Kumar and B. Thangaraju. *Performance Analysis Between RunC and Kata Container Runtime.* In 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), pages 1–4, 2020. doi:10.1109/CONECCT50063.2020.9198653.

[35] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. D. Turck. *Unikernels vs Containers: An In-Depth Benchmarking Study in the Context of Microservice Applications.* In 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2). IEEE, nov 2018. doi:10.1109/sc2.2018.00008.

[36] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. *Unikernels: Library Operating Systems for the Cloud.* In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery. Available from: https://doi.org/10.1145/2451116.2451167, doi:10.1145/2451116.2451167.

[37] V. Cozzolino, A. Y. Ding, J. Ott, and D. Kutscher. *Enabling Fine-Grained Edge Offloading for IoT.* In Proceedings of the SIGCOMM Posters and Demos, SIGCOMM Posters and Demos '17, page 124–126, New York, NY, USA, 2017. Association for Computing Machinery. Available from: https://doi.org/10.1145/3123878.3132009, doi:10.1145/3123878.3132009.

[38] B. Duncan, A. Bratterud, and A. Happe. *Enhancing cloud security and privacy: Time for a new approach?* In 2016 Sixth International Conference on Innovative Computing Technology (INTECH), pages 110–115, 2016. doi:10.1109/INTECH.2016.7845113.

[39] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. *My VM is Lighter (and Safer) than Your Container.* In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. Available from: https://doi.org/10.1145/3132747.3132763, doi:10.1145/3132747.3132763.

[40] A. Bratterud, A. Happe, and R. A. K. Duncan. *Enhancing cloud security and privacy: the Unikernel solution.* In Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, 19 February 2017-23 February 2017, Athens, Greece. Curran Associates, 2017.

[41] *Introduction to the Application Loader*, July 2018. Available from: ftp://bitsavers.informatik.uni-stuttgart.de/pdf/intel/iRMX/iRMX_86_Rev_6_Mar_1984/146196_Burst/iRMX_86_Application_Loader_Reference_Manual.pdf1-3.

[42] N. Har'El. *Running compiled code on OSv*, July 2018. Available from: https://github.com/cloudius-systems/osv/wiki/Running-compiled-code-on-OSv.

[43] Solo.io. *UniK*, July 2018. Available from: https://github.com/solo-io/unik.

[44] *XenServer - open source virtualization*, July 2018. Available from: https://xenserver.org/.

[45] Apache. *JMeter*, July 2018. Available from: https://jmeter.apache.org/.

[46] *Gorilla web toolkit*, July 2018. Available from: http://www.gorillatoolkit.org/pkg/mux.

[47] *Flask RESTful*, July 2018. Available from: https://flask-restful.readthedocs.io/en/latest/.

[48] Eclipse. *Vert.x*, July 2018. Available from: https://vertx.io/.

[49] H. Akdoğan. *An introduction to Vert.x*, July 2018. Available from: https://dzone.com/articles/introduce-to-eclicpse-vertx.

[50] *Go 1.5 Release Notes*, July 2018. Available from: https://golang.org/doc/go1.5.

[51] G. Barany. *Python Interpreter Performance Deconstructed*. In Proceedings of the Workshop on Dynamic Languages and Applications, Dyla'14, page 1–9, New York, NY, USA, 2014. Association for Computing Machinery. Available from: https://doi.org/10.1145/2617548.2617552, doi:10.1145/2617548.2617552.

[52] C. Xu, Y. Bai, and C. Luo. *Performance Evaluation of Parallel Programming in Virtual Machine Environment*. In 2009 Sixth IFIP International Conference on Network and Parallel Computing, pages 140–147, 2009. doi:10.1109/NPC.2009.22.

[53] N. Chowdhury and R. Boutaba. *Network Virtualization: State of the Art and Research Challenges*. IEEE Communications Magazine, 47(7):20–26, jul 2009. doi:10.1109/mcom.2009.5183468.

[54] H. Hamed, E. Al-Shaer, and W. Marrero. *Modeling and Verification of IPSec and VPN Security Policies.* In 13TH IEEE International Conference on Network Protocols (ICNP'05). IEEE, 2005. doi:10.1109/icnp.2005.25.

[55] F. Pohl and H. D. Schotten. *Secure and Scalable Remote Access Tunnels for the IIoT: An Assessment of openVPN and IPsec Performance.* In Service-Oriented and Cloud Computing, pages 83–90. Springer International Publishing, 2017. doi:10.1007/978-3-319-67262-5_7.

[56] I. Kotuliak, P. Rybar, and P. Truchly. *Performance comparison of IPsec and TLS based VPN technologies.* In 2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA). IEEE, oct 2011. doi:10.1109/iceta.2011.6112567.

[57] *Jason A. Donenfeld, Securing Remote Access Using VPN*, May 2019. Available from: https://openvpn.net/whitepaper/.

[58] *Jason A. Donenfeld, WireGuard: Next Generation Kernel Network Tunnel*, May 2019. Available from: https://www.wireguard.com/papers/wireguard.pdf.

[59] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari. *Towards Osmotic Computing: Analyzing Overlay Network Solutions to Optimize the Deployment of Container-Based Microservices in Fog, Edge and IoT Environments.* In 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC). IEEE, may 2018. doi:10.1109/cfec.2018.8358729.

[60] *Cluster networking*, April 2019. Available from: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[61] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente. *Cross-Site Virtual Network in Cloud and Fog Computing.* IEEE Cloud Computing, 4(2):46–53, mar 2017. doi:10.1109/mcc.2017.28.

[62] *WireGuard Sent Out Again For Review, Might Make It Into Linux 5.2 Kernel*, May 2019. Available from: https://www.phoronix.com/scan.php?page=news_item&px=WireGuard-V9-Maybe-Linux-5.2.

[63] *About WireGuard-go*, May 2019. Available from: https://git.zx2c4.com/wireguard-go/about/.

[64] *ZeroTier manual*, May 2019. Available from: https://www.zerotier.com/manual.shtml.

[65] *Ivo Timmermans and Guus Sliepen, Setting up a Virtual Private Network with tinc*, May 2019. Available from: https://www.tinc-vpn.org/documentation/tinc.pdf.

[66] D. Nobori. *Design and Implementation of SoftEther VPN*, May 2019. Available from: https://www.softether.org/@api/deki/files/399/=SoftEtherVPN.pdf.

[67] Y. Li and M. Chen. *Software-Defined Network Function Virtualization: A Survey.* IEEE Access, 3:2542–2553, 2015. doi:10.1109/access.2015.2499271.

[68] T. Goethals, D. Kerkhove, B. Volckaert, and F. D. Turck. *Scalability evaluation of VPN technologies for secure container networking.* In 2019 15th International Conference on Network and Service Management (CNSM). IEEE, oct 2019. doi:10.23919/cnsm46954.2019.9012673.

[69] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. *Containers and Virtual Machines at Scale.* In Proceedings of the 17th International Middleware Conference. ACM, nov 2016. doi:10.1145/2988336.2988337.

[70] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. *Osmotic Computing: A New Paradigm for Edge/Cloud Integration.* IEEE Cloud Computing, 3(6):76–83, nov 2016. doi:10.1109/mcc.2016.124.

[71] A. Ahsan and M. Haque. *UPnP Networking: Architecture and Security Issues.* Available from: https://www.researchgate.net/publication/242305803_UPnP_Networking_Architecture_and_Security_Issues.

[72] *Kubernetes - Building large clusters*, May 2019. Available from: https://kubernetes.io/docs/setup/cluster-large/.

[73] P. Mach and Z. Becvar. *Mobile Edge Computing: A Survey on Architecture and Computation Offloading.* IEEE Communications Surveys & Tutorials, 19(3):1628–1656, 2017. doi:10.1109/comst.2017.2682318.

[74] K. Kumar and Y.-H. Lu. *Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?* Computer, 43(4):51–56, apr 2010. doi:10.1109/mc.2010.98.

[75] D. Santoro, D. Zozin, D. Pizzolli, F. D. Pellegrini, and S. Cretti. *Foggy: A Platform for Workload Orchestration in a Fog Computing Environment.* In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, dec 2017. doi:10.1109/cloudcom.2017.62.

[76] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck. *Resource Provisioning in Fog Computing: From Theory to Practice.* Sensors, 19(10):2238, may 2019. doi:10.3390/s19102238.

[77] A. Morshed, P. P. Jayaraman, T. Sellis, D. Georgakopoulos, M. Villari, and R. Ranjan. *Deep Osmosis: Holistic Distributed Deep Learning in Osmotic Computing.* IEEE Cloud Computing, 4(6):22–32, nov 2017. doi:10.1109/mcc.2018.1081070.

[78] D. Puthal, S. Nepal, R. Ranjan, and J. Chen. *Threats to Networking Cloud and Edge Datacenters in the Internet of Things.* IEEE Cloud Computing, 3(3):64–71, may 2016. doi:10.1109/mcc.2016.63.

[79] M. Villari, M. Fazio, S. Dustdar, O. Rana, L. Chen, and R. Ranjan. *Software Defined Membrane: Policy-Driven Edge and Internet of Things Security.* IEEE Cloud Computing, 4(4):92–99, jul 2017. doi:10.1109/mcc.2017.3791014.

[80] *SoftEther clustering*, May 2019. Available from: https://www.softether.org/4-docs/1-manual/3._SoftEther_VPN_Server_Manual/3.9_Clustering.

3
Fog and Edge

"How many times have you been watching an episode of 'South Park' and thought, 'I'd like to be able to watch this on my television while hooked into my mobile device, which is being controlled by my tablet device which is hooked into my oven, all while sitting in the refrigerator?' " - Trey Parker

3.1 Introduction

This chapter shows how the virtualization technologies introduced in Chapter 2 can be used to create a computational infrastructure extending beyond the cloud into homes, and how they enable flexible and responsive service architectures for increasingly demanding end-user applications. Section 3.2 explains the concepts of fog networks and edge networks, and explains the difficulties of software deployment in such heterogeneous environments. Some state of the art platforms are introduced that are designed to orchestrate software services in edge networks. Section 3.3 shows how a container orchestration agent, FLEDGE, with extremely low resource requirements can be constructed. This agent is designed specifically for edge devices, and although newer alternatives have since been created that focus on functionality to transform edge computing into a complete ecosystem, the resource requirements of FLEDGE compare favorably to their agent components. In Section 3.4, a lightweight service scheduler is constructed that allows real-time service scheduling for hundreds of thousands of nodes, based on

a distance metric, node densities and node resources. This scheduler can be modified to work with Kubernetes [1] to replace its default scheduler. Finally, the chapter is summarized in Section 3.5.

3.2 The Network Edge

In the last decade, the Internet of Things (IoT) approach has become popular in any number of products, from basic sensors for home or process automation to intelligent appliances such as smart light bulbs and washing machines with self-adjusting programs.
These devices regularly send operational data and metadata to data centers in the cloud for several reasons. For one, a centralized access point for software services makes it easier for consumers to control their devices from any physical location. Another reason is that gathering all this data allows manufacturers to further improve their devices and services. However, in the last few years the growth of network traffic and processing power required to support the increasing number of smart devices is too high for centralized data centers to keep pace with [2].
Fog computing [3], which takes place in fog networks outside the cloud, offers a solution to this problem by decentralizing data centers. Although cloud data centers are often geographically distributed to reduce latency and improve end-user experience, they usually service entire countries or large geographical regions. Fog data centers, on the other hand, only service small geographical areas such as (parts of) cities. As a result, they are more numerous by several orders of magnitude, but can also be less powerful because each of them has less data to process. This concept is illustrated in Fig. 3.1.
Although fog computing distributes computational loads and moves processing closer to end-users, this is only the first step towards true decentralization. Edge computing is an additional solution to reduce the load on cloud data centers and end-user devices alike, enabled by ever-improving hardware, increasing energy efficiency, and the proliferation of powerful handheld devices and IoT gateways. The essence of edge computing is that much of the work that is normally performed in the cloud can be broken up into small tasks which are then performed in the network edge. The network edge, shown in Fig. 3.1, consists of billions of low-powered, resource constrained devices [4] which are nonetheless highly programmable. To alleviate the workload of the cloud and fog, the spare capacity of these devices is leveraged to pre-process data and provide basic, highly responsive end-user services. Note that while the border between cloud and fog is well-delineated, the difference between fog and edge is somewhat fuzzy. For

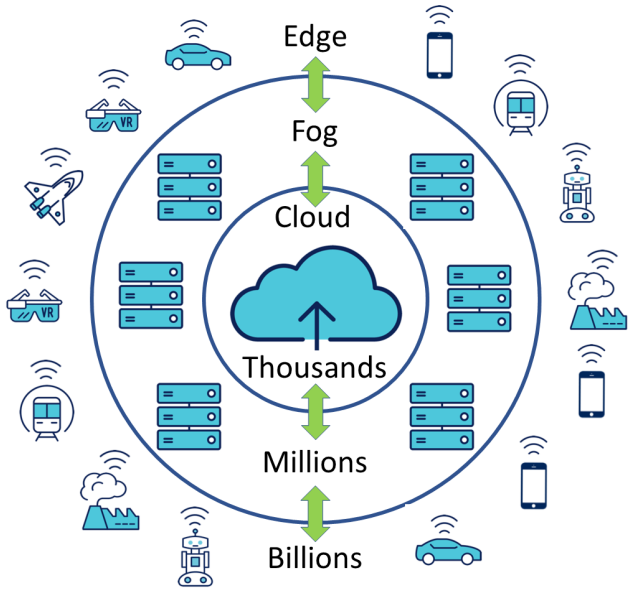


Figure 3.1: Representation of the difference of scale and device types in cloud, fog and edge networks.

example, gateway devices in homes (e.g. Philips Hue Bridge[1]) that collect sensor data and perform local processing could be included in both fog and edge networks depending on the application, even if they are capable of direct end-user interaction.

The combination of cloud, fog and edge has resulted in many studies into tiered service architectures, where intensive data processing, big data analysis and the learning phase of AI models generally take place in the cloud, while the fog and edge provide responsive services and run the inference stage of AI models. Tiered architectures can be achieved with offloading, which uses real-time monitoring and migration of micro-services to move workloads to the fog and edge whenever possible, and back to the cloud if necessary. The term "necessary" may involve a combination of many parameters, from end-user proximity to packet loss and battery life. However, as will be shown in this chapter and Chapter 4, the volatile and heterogeneous nature of the edge and fog can be problematic for scalable, optimal software deployment.

[1]https://www.philips-hue.com/en-us/explore-hue/how-it-works

3.2.1 Service deployment

Both the fog and edge are composed of a wide range of network technologies and various types of devices, leading to volatile conditions for software deployment and communication between services. Service placement parameters in cloud data centers usually involve free resources, service demand, and latencies between servers and data stores. In the network edge, these factors are multiplied in both absolute size (wider range of available resources, higher potential latencies) and scale (more devices to consider). Additionally, power constraints and reliability issues such as localized network outages pose additional problems. Edge computing also presents new security issues; for example, edge devices are often less secure than professionally managed cloud servers, and traffic over public networks may be intercepted if not properly encrypted.
In order for a piece of software to work on a wide range of devices, the various types of virtualization introduced so far can be used. Virtual machines are commonly used in cloud data centers, but since they contain an entire operating system, they are bulky and slow to migrate. Containers, on the other hand, enable flexible but lightweight software deployment on any device that can run the required parts of the Linux kernel. Combined with container engines that manage the containers on a device, and container orchestrators which distribute deployments and tasks among computational nodes, containers are an essential enabling technology for edge computing.
As mentioned in Section 3.1, offloading often takes into account many factors related to device resources and user experience. This is true for service deployment in general, especially in the fog and edge. To that end, orchestrators such as Kubernetes [5] usually work with custom plug-ins for service scheduling, and many studies are directly concerned with measuring the effectiveness of combinations of specific parameters for AI models to optimally assign services to fog and edge nodes.

3.2.2 Container orchestration

Container orchestrators are high-level tools that help organize containerized services over large numbers of computational devices, or a cluster. Initially designed for use in the cloud, most orchestrators are also capable of organizing fog and edge networks, and some are aimed exclusively at enabling software services in the edge by creating entire ecosystems with default APIs. Orchestrators generally fulfil many functions; their main function is to organize and scale software services at the cluster level depending on scheduling parameters, making use of a container engine on each node that takes care of the low-level aspects. Important secondary functions involve

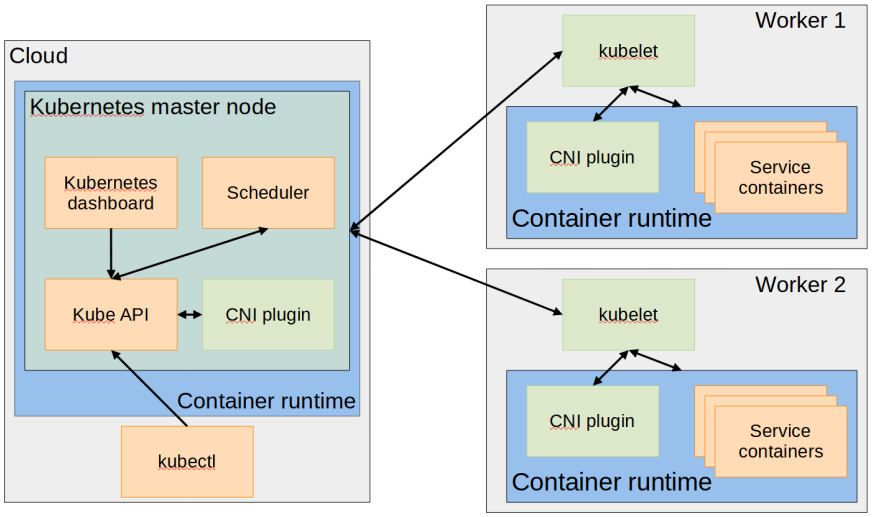


Figure 3.2: Architectural overview of a basic Kubernetes cluster.

setting up a cluster-wide container network, creating internal access points to services with load balancing, managing named services through distributed DNS and providing external access to services. Security, reliability and scalability are important aspects of all these functions.

3.2.2.1 Kubernetes

Kubernetes [1] is a very popular container orchestrator originally inspired by Google Borg [6]. Years of Kubernetes development have contributed to several container standards, for example OCI and CNI which have been discussed in previous sections. Kubernetes is made to run in the cloud, and while it is very flexible and extensible, it tends to use too many resources when deployed on edge devices, leaving little room for service deployment. As Fig. 3.2 shows, a Kubernetes cluster generally consists of a master node, which organizes and schedules services, and a collection of worker nodes, which run the actual software services managed by the master node. Each worker node runs a kubelet, an agent which receives commands from the master node, and translates them into CNI and container runtime calls. Services and scheduling requirements can be declared through a dashboard web application, the kubectl command line tool, or directly via Kubernetes APIs. Although most functions of Kubernetes are distributed and highly scalable, and the master node itself can be decentralized with a high-availability setup, scalability is limited to about 5.000 nodes and 100.000 service instances [7]. While sufficient for cloud data centers, edge networks

with devices in each home are more demanding in terms of scalability. Kubernetes Federations aim to improve scalability by adding a higher management level on top of clusters.
Kubernetes supports a wide range of CNI plugins [8], among which Calico and Flannel, which are used throughout this dissertation. Similarly, various container engines are supported, among which containerd [9] and Docker [10], although the latter is now deprecated.
Because of the flexible nature of Kubernetes, and its extensive development effort, its code is often used as a basis for container orchestrators in the edge. Even alternatives that are not directly based on Kubernetes are generally compatible with its APIs.

3.2.2.2 K3S

K3S [11] is a container orchestrator from Rancher[2] based on the Kubernetes source code, but modified specifically for edge devices. Its name is derived from having "half the memory footprint of K8S" (Kubernetes). First released in February 2019, it is now up to version v1.19.15 (i.e. Kubernetes 1.19), the evaluations were performed in early 2019 using v0.3.0. Unlike FLEDGE, which is only designed for worker nodes, K3S also has its own master nodes. K3S achieves a low resource footprint by removing some functionality from default Kubernetes, and by forcing the use of certain plugins, e.g. Flannel [8] for container networking and containerd as a container runtime. Being based on the Kubernetes source code, it is inherently compatible with Kubernetes itself, and while worker nodes can only work with K3S master nodes, full compliance means that any service configurations made for Kubernetes will also work on K3S.

3.2.2.3 K0S

Taking the nomenclature of K3S a bit further, K0S [12] (Zero friction Kubernetes) is designed to be even easier to set up and maintain, yet highly scalable from edge devices to entire cloud clusters. First released in September 2020, it is now at version v1.22.2 (i.e. Kubernetes 1.22). Like K3S, it uses containerd as a container engine, but it uses kube-router[3] as a CNI plugin. It is also fully Kubernetes compliant, supporting all Kubernetes deployment features.
Although it targets edge devices, its minimum requirements are significant[4], with a minimum of 1GB of memory. As this chapter will show, both K3S

[2] https://rancher.com/
[3] https://github.com/cloudnativelabs/kube-router
[4] https://docs.k0sproject.io/v1.21.0+k0s.0/system-requirements/

and custom-built solutions require far less memory. As such, the official K0S system requirements may be exaggerated, as the updated results in Section 3.3 show.

3.2.3 Edge orchestration solutions

This section discusses new orchestration solutions for edge networks, mostly developed after the publication of the articles this section is based on. Whereas container orchestrators merely organize (containerized) services on a cluster to fulfil end-user demand, these solutions are focused on the network edge by design, and aim to provide a complete ecosystem including default communication and storage methods.

3.2.3.1 ioFog

Eclipse ioFog v2.0 [13] is an edge computing platform that provides all the components and functions required to run large-scale applications in the edge, with the combination of an edge network and the microservices running on it being called an Edge Compute Network (ECN). While it is an independent alternative to Kubernetes, and therefore one of the few projects with a truly novel approach, its main operations and design choices are similar to Kubernetes. It also has the ability to interface directly with Kubernetes clusters through the integration of Virtual Kubelets, which are explained in detail in Chapter 3.3.
Like master and worker nodes in Kubernetes, ioFog uses Controller and Agent nodes to organize and run services, respectively. Additionally, ioFog provides a secure communication infrastructure between nodes with its Router, Proxy components using ioMessages. Some of its components, the Operator and Port Manager, are designed specifically to use ioFog in tandem with Kubernetes, running ECNs partially on Kubernetes clusters.

3.2.3.2 KubeEdge

KubeEdge [14] builds its infrastructure on top of Kubernetes, and rather than as an orchestrator it presents itself as a full Edge Computing Framework specifically designed for the edge. It was initially released in December 2018, while the current stable version is 1.8.2.
KubeEdge is built on open source software, supporting various container runtimes, and provides an ecosystem for container orchestration on edge devices. It consists of a cloud part and an edge part [15]. The cloud part uses the Kubernetes API in the cloud to orchestrate resources, and has high-level control over edge devices. The edge part takes care of the actual

container deployment and provides an infrastructure for storage and event-based communication, the latter being based on MQTT [16].

3.2.3.3 Azure IoT Edge

Azure IoT Edge [17] is an edge computing ecosystem from Microsoft, aimed specifically at executing AI workloads in the edge, but keeping orchestration in the cloud. This approach is useful when a large number of IoT devices need to execute similar but relatively static tasks (e.g. home automation algorithms).
The basic premise of IoT edge is that workloads and services are encapsulated as modules (Docker containers), while devices are managed by a runtime. Orchestration in the cloud assigns modules to devices, which are sent to the runtime that in turn provides the local infrastructure (e.g. networking, storage) for the module to execute. Monitoring is possible through a cloud-based interface.

3.3 Low-resource virtualization

This section contains the edited version of the following publication: **"Extending Kubernetes Clusters to Low-resource Edge Devices using Virtual Kubelets"**, T. Goethals, F. De Turck, B. Volckaert
published in **IEEE Transactions on Cloud Computing, 2020** [18]

Many novel applications need container orchestrators that can work optimally in the cloud and in the network edge, but Kubernetes is designed to run in the cloud. As a result, it is very flexible and modular, but not overly critical of resource consumption. However, edge devices are typically low-resource devices, especially in terms of memory. Additionally, container deployments on edge containers are often specifically meant for a particular device and cannot easily be relocated without an extensive migration process. To address these problems, this section proposes a solution aimed at minimal resource use, and designed to run workloads to completion instead of constantly scaling and shifting them.
In addition to being low-resource devices, consumer-grade edge devices often operate in heterogeneous networks with potentially less focus on organization and security. Connecting these to the cloud can result in some communication and security problems. For example, the network could be hidden behind a router, IP addresses can be unpredictable, existing port mappings can interfere with container requirements, etc.

In the cloud, these problems are usually not present. Infrastructure is well-organized and as homogeneous as possible, while all network resources are predictable and controlled. Furthermore, while all communications between container orchestrator nodes (e.g. Kubernetes) are secured by default, this is not always the case for service endpoints of containers deployed on those nodes.
Therefore, it is important that the solution can secure traffic to and from the cloud, and that other network traffic is either blocked or also secured. To simplify the inter-node network and to ensure smooth deployment of containers, a uniform network environment can be created for edge nodes to be deployed on, capable of supporting a container network on top of it.
Continued development of container management tools such as Kubernetes and Docker [10] has led to the development of a number of standards.
For example, to make sure that every container is reachable and uniquely addressable, container runtimes and orchestrators use an overlay network to assign IP addresses to nodes and containers. Since there are many methods to achieve this on individual nodes, various container runtimes leave this up to network plugins to implement. This has resulted in, among others, the Container Network Interface (CNI [19]), which simply defines a number of high-level operations that governing software can use to organize the container network on a node.
Another example is the Open Container Initiative (OCI [20]), which defines standards for the structure and execution of container images. These standards make sure that a single container image can be deployed and executed with predictable results on any runtime that implements them. At the time of writing, they are implemented by many container runtimes, making it easy to switch runtimes once an orchestrator has basic support for one of them.
The solution is aimed at maximum compatibility with existing container standards, as far as their implementation is possible on edge devices. While it is not absolutely required to implement the full standards, care is be taken that any missing functionality does not result in problems for the rest of the cluster. In addition, if the solution does ignore any standards, it should make sure that other nodes are not affected in any way.
To summarize, the requirements for the proposed container orchestrator for edge devices are:

- To be compatible with modern standards for container orchestration, or to provide an adequate alternative.

- To provide secure communications between edge devices and the cloud by default, with minimal impact on local networks.

- To have low resource requirements, primarily in terms of memory but also in terms of processing power and storage.

This section presents FLEDGE as a low-resource container orchestrator which is capable of directly connecting to Kubernetes clusters by incorporating modified Virtual Kubelets [21] and a VPN. A Kubelet is the part of Kubernetes which is deployed directly on devices to join them into a Kubernetes cluster. A Virtual Kubelet is a small software service which can be deployed anywhere, and which acts as a proxy between the Kubernetes API and a random device that can deploy containers. Behavior is defined by brokers, which form a translation layer between Kubernetes and the devices on which pods are deployed. Because the Virtual Kubelet is designed to work with the Kubernetes orchestrator, it has to work with the limitations inherent in Kubernetes clusters. Kubernetes is designed for use in the cloud, so it is implicitly aimed at orchestration on groups of powerful servers. Since the scalability of Kubernetes constantly evolves, it is considered outside the scope of this topic, but the repercussions are discussed where applicable.
Section 3.3.1 presents existing research related to the topics in this introduction. Section 3.3.2 details the different aspects of using Virtual Kubelets and creating the framework, while Section discusses possible alternatives and how they relate to this work. In Section 3.3.3, an evaluation setup and methodology are presented to compare the solution in this section to similar, popular orchestrator software. The results are presented and discussed in Section 3.3.4. Finally, Section 3.3.5 gives a short overview of the goals stated in this introduction, and how the results and conclusions meet them.

3.3.1 Related work

Shifting workloads between the cloud and edge hardware has been extensively researched, with studies on the use of edge offloading [22], cloud offloading [23, 24], fog computing [25] and osmotic computing [26]. Many studies exist on different container placement strategies, from simple but effective resource requests and grants [27], to using deep learning for allocation and real-time adjustments [28].
Kubernetes is capable of forming federations of multiple Kubernetes clusters [29], but this section aims to use a single cluster for both the cloud and the edge. There are several federation research projects that have resulted in useful frameworks, such as Fed4Fire [30], Beacon [31], FedUp! [32] and FUSE in Chapter 4.3. Fed4Fire requires the implementation of an API to integrate devices into a federation and works on a higher, more abstract level than container orchestration. BEACON is focused on cloud federation and security as a function of cloud federation. FedUp! is a cloud federation

framework focused on improving the setup time for heterogeneous cloud federations. FUSE is designed to federate private networks in crisis situations, but it is very general and primarily aimed at quickly collectivizing resources, not for deploying specific workloads across edge clusters.

Studies exist that focus on security between the edge and the cloud, for example [33] which identifies possible threats, and [34] which proposes a Software Defined Membrane as a novel security paradigm for all aspects of microservices. However, FLEDGE aims to provide only a basic but universal layer of security through an encrypted tunnel, leaving advanced security policy (e.g. firewalls, routing, authorization) up to individual choice.

VPNs are an old and widely used technology. Recent state of the art studies appear to be non-existent, but older ones are still informative [35]. Some studies deal with the security aspects of a VPN [36], while many others focus on the throughput performance of VPNs [37, 38]. While studies exist on using overlay networks in osmotic computing [39], they deal mostly with container network overlays such as Flannel and Weave [8] which are integrated into Kubernetes. Others present a custom framework, for example Hybrid Fog and Cloud Interconnection Framework [40], which also gives a good overview of the challenges of connecting edge and cloud networks. Xu et al. [41] presents a hardware solution against a number of software and physical attacks for untrusted cloud infrastructure, which could be integrated into edge devices.

A study by Pahl et al. [42] gives a general overview of how to create edge cloud clusters using containers. While FUSE in Chapter 4.3 is capable of deploying Kubernetes worker nodes on edge devices, the resulting framework is too resource-intensive for most edge hardware. Cloud4IoT [43] is capable of moving containers between edge networks and the cloud, but it uses edge gateways which indirectly deploy containers on minimalistic edge nodes. K3S [44], which has not yet been the subject of academic studies, is based on the source code of Kubernetes. It achieves lower resource consumption by removing uncommon and legacy features, but it requires its own master nodes to run and can not directly connect to Kubernetes clusters. MicroK8s [45] is another Kubernetes-based solution for edge container orchestration. In addition to having low resource requirements, it is easy to set up, has fast starting times and has built-in GPGPU (General-Purpose computing on Graphics Processing Units) and CUDA support. However, it is aimed at creating smaller clusters for testing, CI/CD and small-scale deployments. KubeEdge [46] is a recent development, aiming to extend Kubernetes to edge clusters. Despite being based on Kubernetes, it also is not directly compatible with Kubernetes master nodes and needs an extra cloud component to function properly. While this section presents a

Kubernetes-oriented solution, Docker Swarm has been used for similar purposes in fog computing [47].
Kubernetes CRDs (Custom Resources Definition [48]) can be used to a similar effect as a Virtual Kubelet. CRDs allow IoT devices or resources to be registered in Kubernetes, where they can be assigned workloads through a custom controller. The main difference with a Virtual Kubelet is that the controller must be hosted on Kubernetes and can control all IoT devices simultaneously. Microsoft uses CRDs for the inverse approach; IoT Edge can integrate with Kubernetes [49] by defining IoT Edge workloads as CRDs, which are converted to pods by the IoT Edge Agent so they can be deployed by the Kubernetes scheduler.
Kubernetes has very limited resource monitoring by default. It keeps a rough overview of the total and required CPUs and memory on each node, but these numbers are unreliable when container orchestration itself takes a significant amount of total resources. Several third party systems have come and gone, such as cAdvisor [50] and Heapster [51], many of which are based on the resource metrics API [52] exposed by Kubelets. There are studies that present their own framework, such as PyMon [53], which is a general container monitoring framework. FLEDGE aims to be compatible with the resource metrics API exposed by Kubelets, so tools such as cAdvisor can monitor them the same way as normal worker nodes.

3.3.2 FLEDGE

This section details how the requirements put forth are met by FLEDGE, starting with a general overview of what a Virtual Kubelet is and how the solution is based on it.
A Virtual Kubelet acts as a proxy for Kubernetes to any platform or device that can run containers. A Virtual Kubelet interacts directly with the Kube API on the master node, and passes API calls to brokers that implement them for the system they represent, for example Amazon AWS, Microsoft Azure or an edge device. The API calls supported by a Virtual Kubelet consist of pod management, pod status, node status, logging and metrics.
Fig. 3.3 shows how a Virtual Kubelet fits into the FLEDGE framework. When FLEDGE is started, the Virtual Kubelet is initialized and the FLEDGE broker connects to it, receiving commands from Kubernetes through it. Depending on its configuration, the broker will initialize a specific Container Runtime Interface which decomposes the commands into container networking, cgroup management, namespace management, or passes them on to a container runtime (e.g. Docker, Containerd). This interface should not be confused with the CRI standard; it is the part of a Virtual Kubelet which interfaces with the selected back-end, or in this case FLEDGE components.

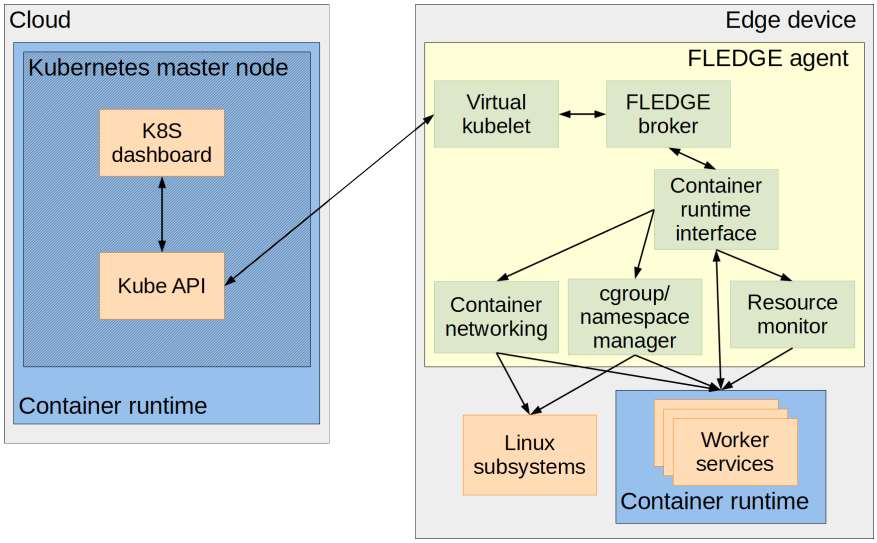


Figure 3.3: Conceptual overview of FLEDGE and its use of a Virtual Kubelet.

The collection of FLEDGE components deployed on an edge device will be referred to as a FLEDGE agent.

3.3.2.1 Compatibility

One of the requirements for FLEDGE is that it should support container standards and existing container runtimes. There are a few aspects to this requirement, some of which are limited by the APIs of existing software.

The first aspect is the choice of container runtime. While Docker may seem like a logical choice because it is very widely supported, Containerd is also an option. Since version 1.11, Docker relies on Containerd for some operations, such as container execution. Both runtimes support the OCI standards, so they can both create and run OCI containers (Docker containers). In terms of compatibility, both are valid choices, so ultimately it comes down to a trade-off between ease of implementation and resource requirements, which will be discussed in Section 3.3.2.3.

Related to the choice of container runtime is compatibility with Linux cgroups. Some devices and operating systems do not support all cgroups by default, making it hard or impossible to correctly run Kubernetes deployments. On a Raspberry Pi 3 running Raspbian for example, cgroups used for CPU throttling may be missing from the kernel, which must be custom-built in order to guarantee compliance with Kubernetes specs. If these kernel options are missing, FLEDGE will generate a warning, but

still continues with the deployment if possible. Since neither Kubernetes nor Docker seem affected by the absence of the cgroups in question, this approach seems to be the standard.

Another aspect of compatibility is how container networking is handled. In Kubernetes, container networking is implemented as an overlay network [54] in which each pod can be assigned a distinct IP address on a virtual network interface. This is achieved by assigning sub-ranges of a configurable IP range to each node, from which they in turn assign IP addresses to their pods. Kubernetes itself makes high-level decisions on container networking, such as assigning IP ranges to the Kubelets on the nodes. The assignment of IP addresses to pods and the setup of network namespaces and virtual network interfaces is handled by CNI compatible network plugins (e.g. Flannel, Weave) on the nodes themselves. To use a specific network plugin, it is deployed on the master node, which in turn makes sure it runs on all worker nodes.

In FLEDGE, this is implemented differently. By fulfilling the role of both Kubelet and container network plugin, there is no need for the CNI layer usually present between Kubelet and network plugin. Additionally, the number of pods that can be deployed on edge devices is rather limited compared to cloud infrastructure. This means that it is preferable to implement a simple and naive, but effective pod networking handler (Fig. 3.3 *Container networking*) which hands out IP addresses on a first come, first serve basis. This pod networking handler is also responsible for configuring networking namespaces correctly (Fig. 3.3 *namespace manager*), independent of the active container runtime. The deployment of the network plugin itself is prevented by labelling the node so it is not eligible for deployment.

Since FLEDGE uses the Kubernetes-assigned IP ranges to configure its container networking, this approach does not influence container networking in the rest of the cluster. The master node is unaware that the node does not deploy the default network plugin and handles its networking needs, and the container networking plugin is still deployed and functioning normally on other nodes.

As stated in the introduction, Kubernetes node resource monitoring is sufficient to determine if any additional pods can be deployed on a node. This monitoring is based on the total resources of a system and the guaranteed resources allocated to pods, actual resource use is not taken into account. This is a problem for edge devices, since operating system and orchestrator resource use can constitute a significant portion of total resources, making it hard to gauge if a device can take additional load based on pod-allocated resources alone[5].

[5]This has been addressed in later Kubernetes versions, although only static amounts

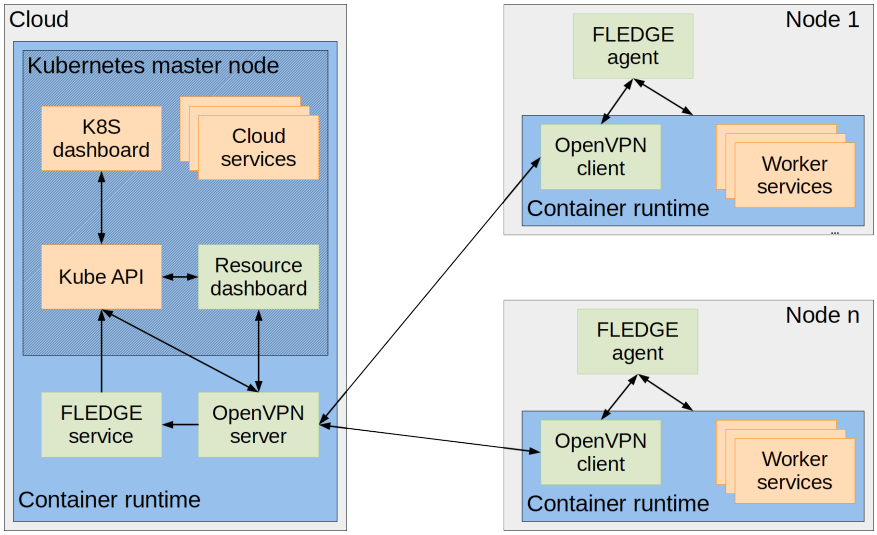


Figure 3.4: High-level overview of network traffic flow of FLEDGE, using OpenVPN to connect edge nodes to the cloud.

Luckily, Kubelets also provide the Resource Metrics API which is used by several third-party monitoring tools. In order to support these monitoring tools, one needs to implement the Resource Metrics API up to a level sufficient for monitoring edge device resources and pods (Fig. 3.3 *Resource monitoring*). By default, the Resource Metrics API is hosted on the same port as on a normal Kubelet.

3.3.2.2 Security and stability

Edge devices often find themselves in heterogeneous networks with little to no organization or security. This randomness of topology, IP address assignments and port mappings is not an ideal situation for building a cluster and deploying containers. Furthermore, the situation could be exacerbated by the presence of a router with either NAT or a firewall. Finally, while Kubernetes API traffic is encrypted and authenticated by default, the same is not always true of services deployed on nodes, so all traffic between the cloud and the edge should be secured by default.

In FLEDGE, this is solved by setting up a VPN, more specifically OpenVPN, and building the cluster and container network on top of its interfaces. The basic traffic flow of this setup is shown in Fig. 3.4. While simple, this

of resources can be passed: https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/

approach fixes the problems described above:

- IP addresses of nodes are predictable and directly reachable by the master node.
- The VPN interface is a proverbial clean slate; all ports are open and available for use.
- Physical layout of the network no longer matters, the VPN can be organized according to logical parameters.
- UDP hole punching might be required to overcome NAT or a firewall, but this is taken into account by OpenVPN.
- Packets are encrypted by default for a basic layer of security.

However, the effectiveness of using a VPN also depends on the software used and its exact configuration:

- The effectiveness of packet encryption depends on the chosen algorithm, and encryption can even be turned off entirely for performance reasons. FLEDGE uses default OpenVPN encryption.
- Using a VPN is a drain on system and network resources, likely reducing the scalability of clusters. OpenVPN has another drawback in that it can only use a single CPU core, which may quickly saturate and limit its performance on edge devices.
- Anyone with physical access to the device can piggy-back on the VPN connection and reach any cluster services. Preventing this requires physical and OS-level security.

The custom container network implementation in FLEDGE uses IPtables to configure the routing between pods and the rest of the cluster. Properly configuring IPtables with respect to the VPN interface allows the exclusion of certain traffic flows, as shown in Fig. 3.5. The solid green arrows indicate traffic flows allowed by FLEDGE, showing that any container running on a FLEDGE agent can access any device or container in the VPN and the pod network. Not all devices need to be connected to the VPN, nor do they all need to be part of the pod network. Note however, that traffic from devices in the pod network that are not connected to the VPN can easily be blocked by configuring IPtables differently.
Fig. 3.6 reiterates Fig. 3.4, but on the level of network interfaces. This figure shows how the entire container network is built on the VPN network,

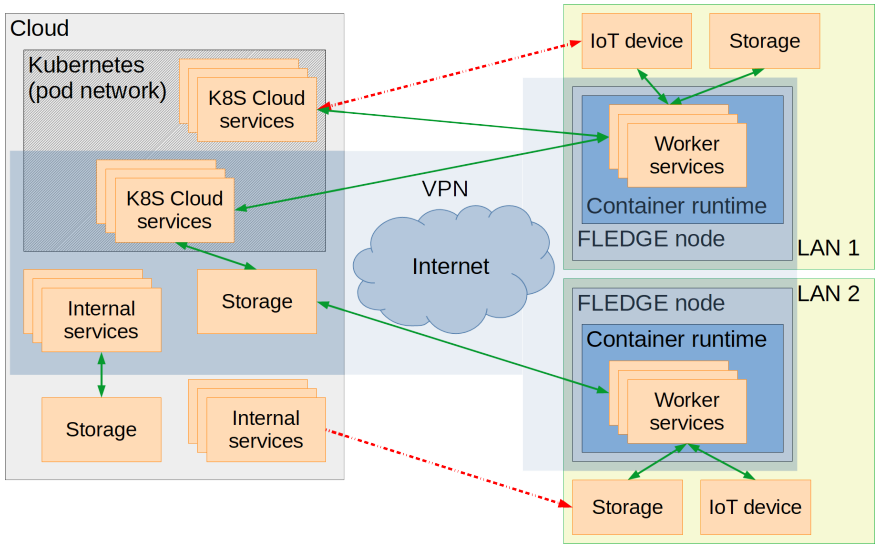


Figure 3.5: Overview of network traffic flows in a cluster using FLEDGE nodes. Green arrows indicate possible traffic flows.

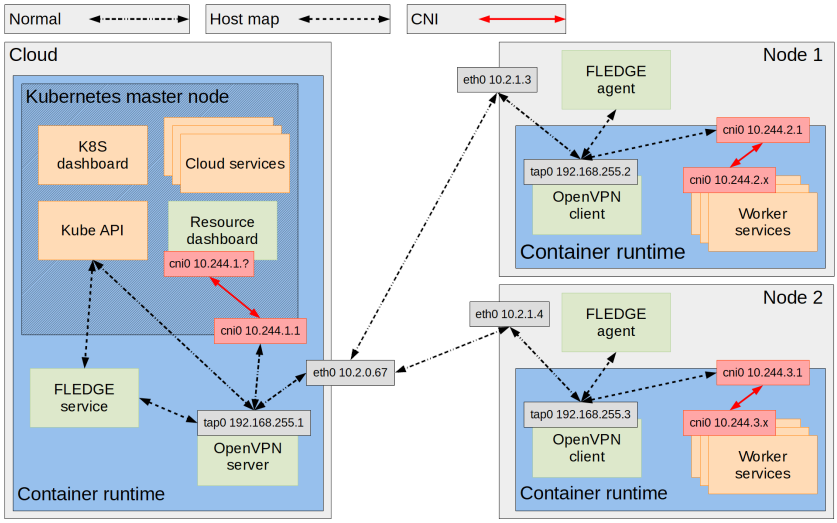


Figure 3.6: Overview of network traffic flows in a cluster using FLEDGE nodes, on the interface level.

and that all traffic uses the VPN interfaces. *CNI* (solid red) arrows represent container network traffic, while *Normal* arrows represent VPN and ethernet traffic. The *Host map* arrows indicate traffic to containers that use the host network namespace, meaning that they have direct access to the network interfaces of the node they run on rather than operating through a virtual network interface. The OpenVPN and FLEDGE containers run in privileged mode, since they need the authority to create and modify network interfaces, cgroups and namespaces. Similarly, they run in the host network namespace because they need to cooperate and create network infrastructure for the container runtime and the master node. While the figure indicates that all container traffic goes via the container network, they can still be assigned to the host network namespace. After all, FLEDGE simply executes Kubernetes deployments. However, putting them on the host network namespace is discouraged, since that might make it harder to communicate with other cluster services.
Container images may contain proprietary software that needs to be protected from local and remote unauthorized access, and the resulting risk of reverse engineering. Because FLEDGE agents have to run as root, they present a prime attack vector to access all of the images and containers they manage. However, a few steps can be taken to mitigate this:

- Running containers are by default assigned to different file system root folders by most container runtimes. While a root account can easily access the file system of a container, it can be protected against any user that is not root, apart from the user running the container.
- To minimize the chance of images being copied and reverse-engineered, they can be removed when the containers in a pod are finished. While this also frees up some extra storage for reuse, it may slow down redeployments of the same pod because the container runtime needs to download the images again.
- FLEDGE cleans up all network infrastructure, containers and images on shutdown. This is also required for leaving the system in the same state it was in before deploying FLEDGE.

3.3.2.3 Low resource use

The choice of container runtime is very important for resource use. Because Docker relies on Containerd to actually run containers, Containerd is likely the most resource-friendly option. On the other hand, the Containerd APIs require more low-level implementation to use effectively than those of Docker.

For low-resource edge devices it is reasonable to put resource requirements before ease of implementation, and since the compatibility section has shown that there is little to no difference in supported standards between the container runtimes, it stands to reason to propose Containerd as the runtime for FLEDGE. The Results section will further validate this choice.

Networking
The compatibility section argued in favor of a custom CNI facility in FLEDGE, rather than using one of the existing containerized network plugins such as Weave or Flannel.
This design choice is optimal in terms of resource requirements, considering that all container plugins are deployed as containers. While using containers is flexible and more durable than other forms of plugins (e.g. host process or in the same process), it also means that a plugin requires a significant amount of resources to run. While this is not explicitly reflected in any results in this section, the requirements for Flannel are determined and discussed in Section 3.3.4 to support this claim.

Namespaces and cgroups
Using the low-level APIs of Containerd means that some functionality needs to be implemented explicitly. Two of the most important aspects of this functionality are cgroups and namespace handling.
While both Docker and Containerd create the required namespaces for a new container, FLEDGE takes care of all namespace management after the creation of the first container of a pod. This is to make sure that no matter which container runtime is used, the behavior is the same.
On the other hand, container resource restrictions are much easier to pass directly via the Docker API, which populates the required cgroups automatically. While Containerd is also capable of making cgroups, the actual restrictions need to be set by the program using the Containerd APIs. Therefore, FLEDGE only allows the creation of one cgroup of each type (memory, cpu, ...) per pod. After configuring the resource restrictions, it forces Containerd to reuse them for the rest of the containers in a pod.
Similar to container networking, the complexity of cgroup and namespace management on edge devices is much reduced compared to cloud infrastructure. Therefore, despite increasing the complexity of FLEDGE, handling cgroups and namespaces in FLEDGE itself using a minimal implementation allows conserving resources for actual workloads.

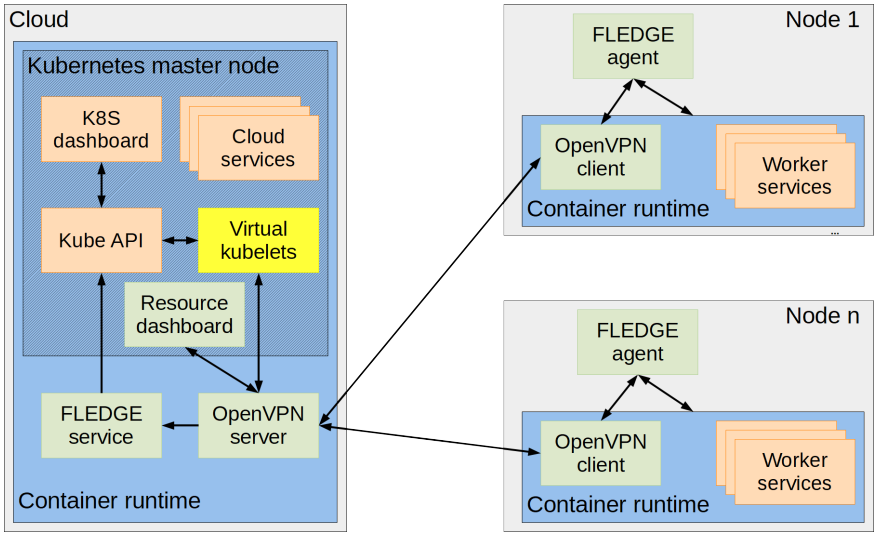
Virtual Kubelet location
As explained above, the Virtual Kubelet is only a small part of FLEDGE.

While instrumental in the communication with Kubernetes master nodes, its location matters very little since a custom broker implementation can forward API calls to other devices.
In terms of resource requirements, this allows for two options when considering where to run the Virtual Kubelet:

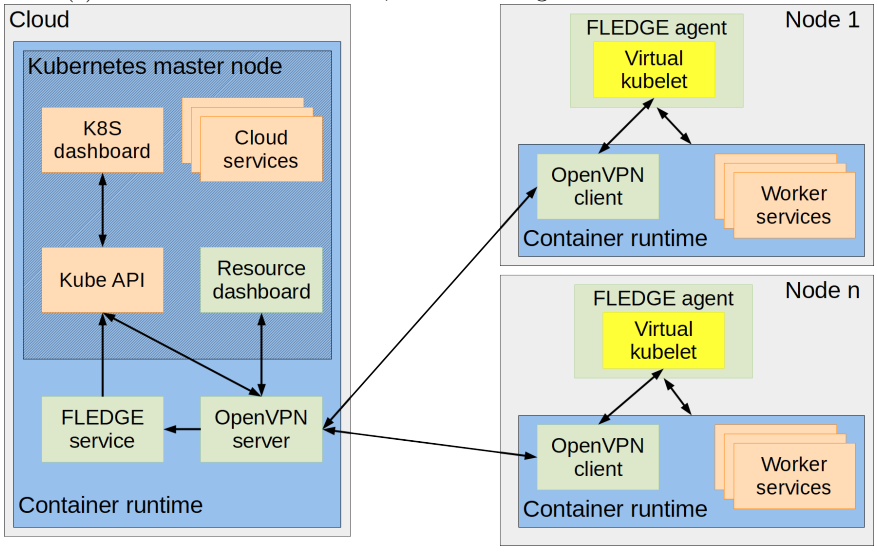
- In the cloud: the Virtual Kubelets are run as pods in the cloud, entirely separate from the FLEDGE agents which are running on edge devices. Kubernetes API calls received by Virtual Kubelets are forwarded to FLEDGE agents via REST services. This approach shifts some of the resource requirements from the edge to the cloud, while allowing for a more robust system. For example, when a FLEDGE agent loses its connection to a Virtual Kubelet, the Virtual Kubelet can queue commands and give default responses until the agent comes back online.

- On the edge: the Virtual Kubelet is integrated into the FLEDGE agent and run as a container or a normal process on the edge device. Kubernetes API calls are executed directly in the same process. While this approach requires more resources on the edge and is less resistant against network problems, it does reduce the operational and technical complexity of FLEDGE.

The two options are further illustrated in Fig. 3.7. On the left, the Virtual Kubelets run in the cloud, while on the right the Virtual Kubelet is shown in its pass-through role on the edge device.
Note that when the Virtual Kubelets are run in the cloud, a small web service (FLEDGE service) is required on Kubernetes master nodes to simplify Kubernetes API access for FLEDGE agents. Without this service, FLEDGE agents would have to include the full Kubernetes API, increasing their size by about 20MiB. When the Virtual Kubelet is integrated into the FLEDGE agent, they share the Kubernetes API and the FLEDGE service is no longer required. The resources required for the FLEDGE service are insignificant when deployed on a server, so they will not be taken into account for the rest of the section.
To properly determine where to put the Virtual Kubelet, a model needs to be constructed which takes into account the resource use in both situations, and the relative importance of edge resources versus cloud resources. Kubernetes v1.14 has a limit of 5000 nodes per cluster [7]. Because there is always at least one master node, this means a cluster can contain at most 4999 edge nodes. However, if the Virtual Kubelets are deployed in the cloud as pods, the maximum number of pods per node is also important. For v1.14



(a) Virtual Kubelet in the cloud, communicating with FLEDGE over VPN.



(b) Virtual Kubelet in the edge, passing Kubernetes calls directly to the FLEDGE broker.

Figure 3.7: Overview of FLEDGE architecture for different locations of the Virtual Kubelet.

this limit is 110, but taking plugins and default pods into account, 100 pods per node is a safe estimate. This means that for every 100 edge nodes, there would need to be an additional node to manage the Virtual Kubelet pods, increasing the complexity of the management structure in the cloud. Modelling all the requirements starts with calculating the required number of management nodes N_M and management efficiency E:

$$N_M = \lceil \frac{L_N}{L_P + 1} \rceil \tag{3.1}$$

$$E = \frac{L_N - N_M}{L_N} \tag{3.2}$$

Where L_N is the limit of nodes per cluster and L_P is the pod limit per node. Eq. (3.1) and eq. (3.2) can be used to construct the total memory used by all pods M_{Pods} and nodes M_{Nodes}:

$$M_{Pods} = L_N \cdot E \cdot M_{Pod} + M_{Shr} \cdot (N_M - 1) \tag{3.3}$$

$$M_{Nodes} = (N_M - 1) \cdot M_{Kube} \tag{3.4}$$

Where M_{Pod} is the amount of non-shared memory required per Virtual Kubelet, M_{Shr} is the amount of memory shared by all Virtual Kubelets on a node, and M_{Kube} is the amount of memory required for a Kubernetes installation. M_{Kube} can be extended to the memory requirement of an entire operating system or virtual machine, depending on how Kubernetes master nodes are instantiated in cloud infrastructure. Eq. (3.3) and eq. (3.4) can in turn be used to calculate the maximum additional amount of memory the Virtual Kubelet should require per edge node for edge placement to be more memory efficient than cloud placement:

$$M_E = C_M \cdot \frac{L_N \cdot E \cdot M_{Pod} + (M_{Shr} + M_{Kube}) \cdot (N_M - 1)}{L_N} \tag{3.5}$$

Where C_M is a constant representing the relative cost of edge memory versus cloud memory. This constant is important because cloud memory is cheap and easily extensible. Similar to eq. (3.5), a formula can be constructed for storage requirements:

$$S_E = C_S \cdot \frac{L_N \cdot E \cdot S_{Pod} + (S_{Shr} + S_{Kube}) \cdot (N_M - 1)}{L_N} \tag{3.6}$$

Where C_S, S_{Pod}, S_{Shr} and S_{Kube} fulfil the roles of C_M, M_{Pod}, M_{Shr} and M_{Kube} respectively. The only factor not considered in these equations is the cost of maintaining a more complex cluster of master nodes in the cloud, which is very case-dependent and hard to estimate.

C_S is assumed to be 1, since the target class of edge devices can routinely store several gigabytes, and the size of a discrete Virtual Kubelet is merely 32MiB on x64. For such small amounts, storage is equally cheap in the cloud and on the edge. Furthermore, edge devices that are not equipped with at least 512MiB (compressed) storage are unlikely to be able to run a Linux based operating system with a kernel capable of handling containers, so it is not useful to consider such devices for container deployment. C_M depends on a lot of factors. Most important of all, cloud memory is often priced in terms of GiB-seconds, while edge hardware is a one time purchase but typically non-extensible. Both types of memory have a wide range of pricing constantly in flux, further complicating attempts to calculate C_M. For the rest of this section, it will naively be assumed to equal 1.

3.3.2.4 Comparison to alternatives

It has already been discussed how the most important elements of FLEDGE relate to Kubernetes in previous sections. However, there is still an important difference between FLEDGE and Kubernetes in that the latter requires all swapping to be disabled. This leads to serious performance and stability issues on some edge devices (e.g. Raspberry Pi 3), which are already low on memory after a Kubernetes deployment. FLEDGE does not require swap to be turned off, so all memory subsystems can perform as intended.
Where FLEDGE starts out from scratch and works towards Kubernetes compatibility, K3S takes the inverse approach and eliminates unnecessary code and functionality from the full Kubernetes source code. Unlike Kubernetes, it has no choice of container runtime; Containerd is used by default. Similarly, Flannel is integrated for container networking. While K3S (and K0S) has better support for Kubernetes APIs, not being built from scratch can be a disadvantage for it in terms of resource requirements. Additionally, it has a slightly different cluster join mechanism and a thin wrapper layer which gives it its own shell commands. These changes mean that, for now, K3S worker nodes cannot be used in a Kubernetes cluster, but only in K3S clusters.
Since it is much more than just a simple container orchestrator, including it in the evaluations would not result in a fair comparison for KubeEdge. Despite this, it is unlikely to be a very-resource efficient solution because of its use of Docker, a point which will be proven in the Results section.

3.3.3 Evaluation setup

Now that the FLEDGE architecture has been explained and alternative approaches have been identified, an evaluation environment can be con-

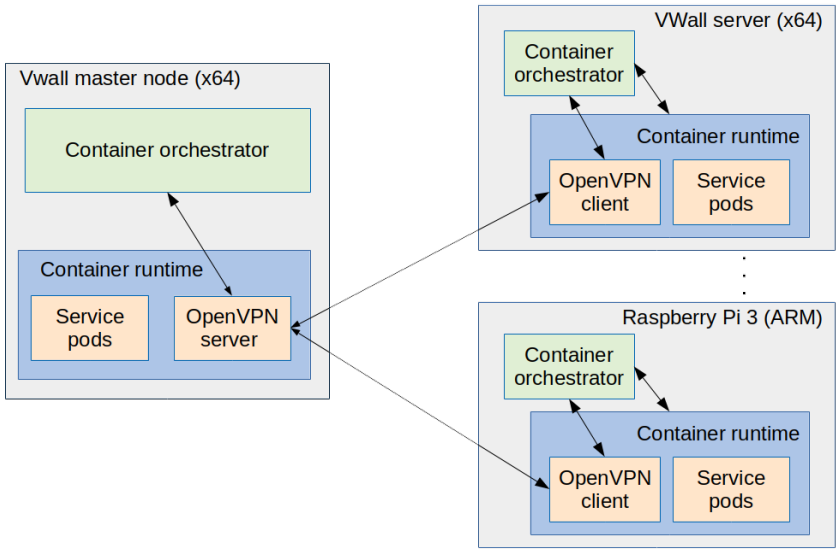


Figure 3.8: Overview of the hardware setup used for the evaluations.

structed. The evaluations are intended to confirm some of the choices made in earlier sections, to back up some claims, and to give an indication of how FLEDGE stacks up against alternative orchestrators in terms of resource consumption.
The source code of FLEDGE is made available on Github[6].
This section first gives an overview of the evaluation setup and general methodology. Subsequently, the specifics of each evaluation are explained.

3.3.3.1 Methodology

Fig. 3.8 shows the hardware setup used for the evaluations. There are 3 devices involved:

- The *VWall master* node fulfills the role of a Kubernetes/K3S master node. Its specifications are not important, since the worker nodes are the focus of the evaluations.
- The *VWall server (x64)* is used to determine the resource requirements of orchestrator worker nodes on an x64 CPU architecture. This server has an AMD Opteron 2212 processor at 2GHz and 4GiB RAM, running Ubuntu 18.04.

[6]https://github.com/togoetha/fledge

- The *Raspberry Pi 3* is used to determine resource requirements for FLEDGE on an ARM CPU architecture. This device runs Raspbian with kernel version 4.14.98-v7+ on the default hardware configuration; 1GiB RAM and a quad-core 1.2GHz CPU.

All devices are in the same geographical location and are connected by a Gigabit LAN. The OpenVPN server and clients are only used when FLEDGE is deployed on the worker nodes, Kubernetes and K3S connect to the master node directly via LAN. All evaluations will be performed on both ARM and x64.
The container runtime used in any evaluation depends on the orchestrator being tested. For Kubernetes, Docker is used, while K3S has Containerd by default. For FLEDGE, both Docker and Containerd are possible.
The storage requirements for each orchestrator are determined by using the df [55] command before and after orchestrator setup. After every evaluation, the devices are wiped to ensure the same state at the start of each evaluation. In addition to the orchestrator and the container runtime, this approach also takes packages and libraries into account that are required to run the orchestrator properly, thus forming a complete picture of storage requirements. Because no deployments or workloads are executed apart from the default containers required for each orchestrator, storage does not vary over time and thus it is not necessary to measure beyond the successful start of each orchestrator.
Measuring memory use is more complex than determining storage requirements, for the following reasons:

- Unlike the thousands of files involved in setting up an orchestrator, the processes involved in running it can be easily identified, so a more granular approach is possible. This is not only more accurate, but allows for more detailed conclusions by studying subsets of processes.

- It stands to reason that memory use is not as static as storage requirements. During deployment, a lot of memory will be used which may be released again later. Therefore, memory use must be monitored over a significant period of time to form a complete picture.

- Processes can have private and shared memory. While it is easy enough to obtain these numbers, a fair method is required to calculate the exact memory use of a process from both numbers.

During each evaluation, memory is measured every 30 seconds over a period of 15 minutes, while the pmap [56] command is used to determine the Proportional Set Size [57] (PSS) of each process, calculated using the following formula:

$$M_{total} = P + \sum^{i} S_i/N_i$$

where P is private memory, S_i are various sets of shared memory, and N_i is the number of processes using any piece of shared memory.

3.3.3.2 Container runtime comparison

Previous sections have argued that the choice of container runtime can have a large impact on resource requirements for an orchestrator solution. In order to verify this, FLEDGE is set up as in Fig. 3.7a, using both Docker and Containerd. No pods or containers other than the FLEDGE agent and a VPN client are deployed, to reduce the influence of other processes on memory use behavior. A third case is also examined, in which the FLEDGE agent runs directly on the host while using Containerd as a runtime, to determine the containerization overhead of the FLEDGE agent.
In all cases, the processes monitored are container runtime daemons, the FLEDGE agent, the VPN client and Containerd shims [58].

3.3.3.3 Virtual Kubelet integration

As shown in Section 3.3.2, Virtual Kubelets can either be deployed on the master node or merged with FLEDGE on edge devices. This evaluation is meant to gather the required data for Eq. 3.5 and Eq. 3.6 so an argument can be made for the correct approach.
To gather the required data, FLEDGE is set up as described in both Fig. 3.7a and Fig. 3.7b, and the same processes are monitored as in the Container runtime comparison.

3.3.3.4 Orchestrator comparison

As presented in Section 3.3.1, there are a number of alternatives to FLEDGE. Since the point of FLEDGE is to provide a Kubernetes-compatible container orchestrator with minimal resource requirements, this evaluation is meant to verify that FLEDGE requires fewer resources than Kubernetes worker nodes on edge devices. To fully prove this, Kubernetes is allowed to deploy a kube-proxy [59] on FLEDGE to level the playing field. Flannel will be used as a CNI plugin, but since FLEDGE has its own container networking it will only be deployed on Kubernetes worker nodes.
Additionally, FLEDGE is compared to K3S to show that it is a useful alternative to K3S. Because K3S does not actually include kube-proxy by

default, this evaluation compares K3S to a FLEDGE deployment without kube-proxy. FLEDGE will be similarly compared to K0S, ioFog and KubeEdge. Although these orchestrators all use various strategies for edge orchestration, their features and methods of deployment are similar enough for direct comparison.
Two versions of the results are presented; one set is the original results from 2019, comparing FLEDGE to contemporary versions. Another set is from 2021, comparing FLEDGE to the evolving resource requirements of K3S and Kubernetes, and adding results for K0S, ioFog and KubeEdge. As KubeEdge supports various container runtimes, it was configured to use containerd.
For this evaluation, the monitored processes are the container orchestrator, the container runtime, shims and any deployed containers (including VPN for FLEDGE). FLEDGE uses Containerd as a container runtime.

3.3.4 Evaluation results

This section presents the results of the evaluations described in Section 3.3.3. For practical purposes, x64 numbers are shown as blue series in the charts, while ARM is shown as red with dashes. Storage requirements charts are bar charts showing medians, memory charts also have error bars indicating the median absolute deviation.

3.3.4.1 Container runtime comparison

Fig. 3.9 shows the storage requirements for FLEDGE deployments using either Docker or Containerd.
The first important observation is that on ARM devices, a FLEDGE deployment using either Containerd and Docker requires far less storage than on x64. The difference is especially large in the case of Docker and FLEDGE, which needs 3 times as much storage on x64 as it does on ARM.
At first sight, it appears that Containerd is much less efficient on ARM than Docker is, but this conflicts with the fact that Docker uses Containerd for many container tasks. However, in order to use a containerized version of the FLEDGE agent with Containerd, many files and resources need to be made available inside the FLEDGE agent container for it to be able to deploy containers itself.
It turns out that mounting all these file paths inside the FLEDGE agent container at runtime creates a multitude of file system layers which inflate storage requirements up to 4 times the original size. In order to validate this, a FLEDGE agent was run as a host service with Containerd as a container runtime. Additionally, this version of Containerd was cleaned of

unnecessary support executables, most notably the command line tool *ctr*, since only API interaction is required. This is similar to the approach K3S uses, and the most important downside is that the command line can no longer be used for debugging purposes. This approach (Fig. 3.9 *Host+ctd*) is much more resource efficient, using only about one third of the resources Docker requires on both x64 and ARM.
Note that the same approach does not work with Docker; running the FLEDGE agent as a host service with Docker as a container runtime gives nearly the same results as in Fig. 3.9. This indicates that while Docker may use Containerd as a runtime, it has a much more efficient method of creating and mounting file system layers.
The results in Fig. 3.9 can thus be explained by two causes. The first is how mounts are handled by the container runtimes, the second is the result of instruction set differences and larger overall binaries on x64. The effects on required storage are respectively additive and multiplicative. This is reflected in the *Host+ctd* and *Docker* categories, which scale more or less equally between x64 and ARM, whereas the inflated layers in Containerd are similar added burdens on both x64 and ARM. Note that the differences in the latter case are not identical, since some of the mount points include binaries that are also platform dependent.

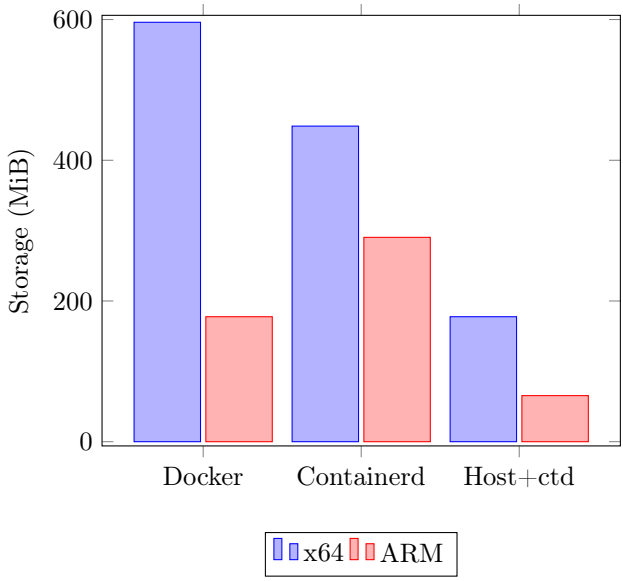


Figure 3.9: Storage requirements of FLEDGE using different container runtimes, including all relevant processes.

Fig. 3.10 shows the memory use of FLEDGE using either Docker or Con-

tainerd. Again, the ARM versions are much more resource efficient, using up to 50% less memory for Docker and 65% for Containerd.
As far as container runtimes go, Containerd is by far the best option to use with FLEDGE. The ARM setup of FLEDGE using Containerd requires only about 80MiB storage and 50MiB memory in total, including a VPN client container.

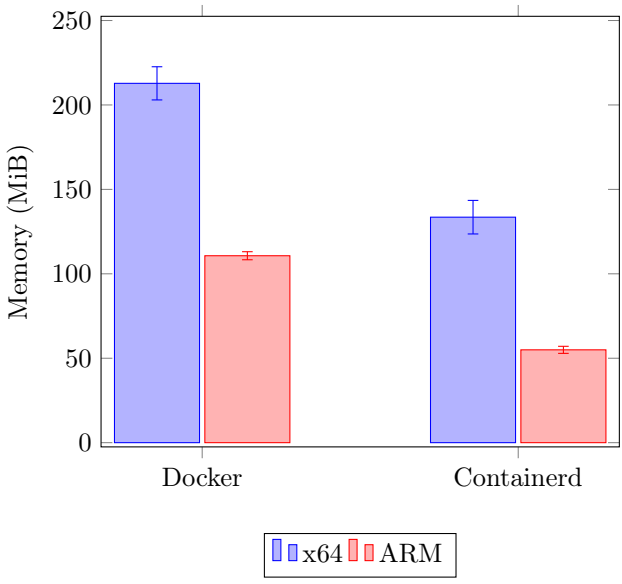


Figure 3.10: Memory use of FLEDGE using different container runtimes, including all relevant processes.

3.3.4.2 Virtual Kubelet integration

The effects of integrating the Virtual Kubelet into FLEDGE are shown in Fig. 3.11 and Fig. 3.12. In Section 3.3.2, Eq. 3.5 and Eq. 3.6 were constructed to calculate the maximum amount of storage and memory this integrated solution should use. By measuring the resource consumption of Virtual Kubelet pods on the master node, M_{Pod} is determined to be 10MiB and M_{Shr} 20MiB. Other factors are harder to pin down, but they are estimated at 500MiB for M_{Kube}, 0MiB for S_{Pod}, 40MiB for S_{Shr} and 1200MiB for S_{Kube}. Using the default Kubernetes node and pod limits, M_E and S_E are calculated and shown in the figures as horizontal lines, indicating the useful limits for memory and storage respectively.
As Fig. 3.11 shows, integrating the Virtual Kubelet into FLEDGE is not optimal for storage, especially in the case of x64, but considering that it only

goes 3MiB over the "limit" it is unlikely to matter much. Fig. 3.12 shows slightly better results for memory use. On ARM, there is a good reason to run the Virtual Kubelet in FLEDGE on the edge, since it uses about 10% less memory than the calculated useful limit. For x64, moving the Virtual Kubelet to the edge is more or less memory neutral, with median memory use being exactly the limit.

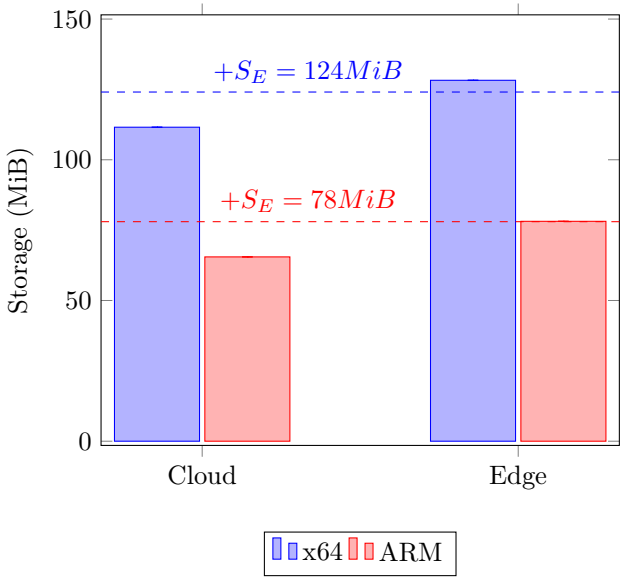


Figure 3.11: Storage requirements of FLEDGE while running the Virtual Kubelet in the cloud or on the edge. The horizontal lines indicate the useful upper limits for integrating the Virtual Kubelet into FLEDGE on the edge for x64 and ARM, calculated by adding the result of Eq. 3.6 to the numbers of the *Cloud* category.

3.3.4.3 Orchestrator comparison

Fig. 3.13 shows the storage requirements of FLEDGE compared to those of Kubernetes. For both x64 and ARM, FLEDGE requires significantly less storage than Kubernetes, but the difference is largest on x64 with about 75% less storage. On an ARM device, FLEDGE requires about 60% less storage than Kubernetes. This large difference can be attributed to several factors, including the choice of Containerd over Docker and integrating several plugins instead of running them as containers.
The memory use of FLEDGE compared to Kubernetes is shown in Fig. 3.14. Again, FLEDGE requires significantly fewer resources than Kubernetes,

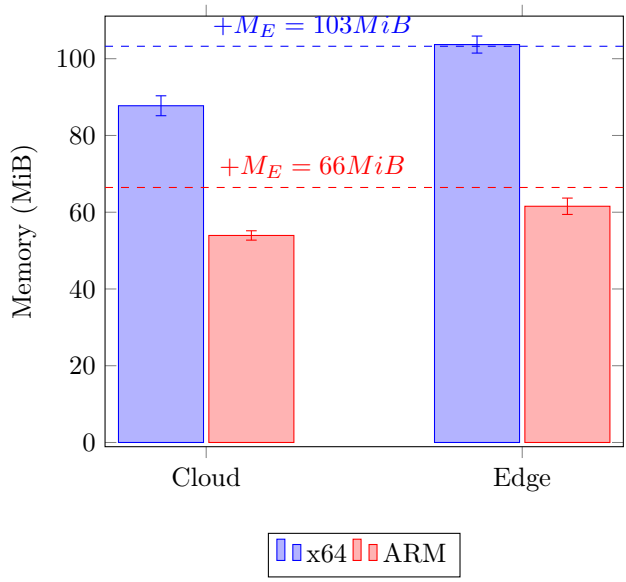


Figure 3.12: Memory use of FLEDGE while running the Virtual Kubelet in the cloud or on the edge. The horizontal lines indicate the useful upper limits for integrating the Virtual Kubelet into FLEDGE on the edge for x64 and ARM, calculated by adding the result of Eq. 3.5 to the medians of the *Cloud* category.

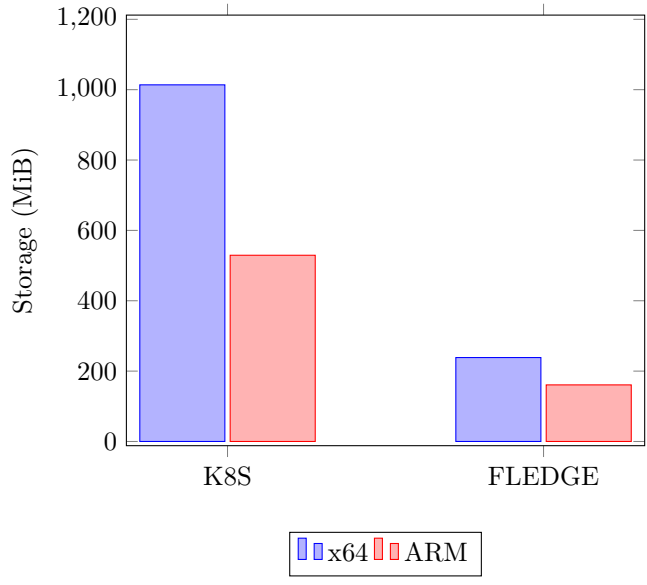


Figure 3.13: Comparison of the storage requirements of Kubernetes and FLEDGE, both running a kube-proxy deployment.

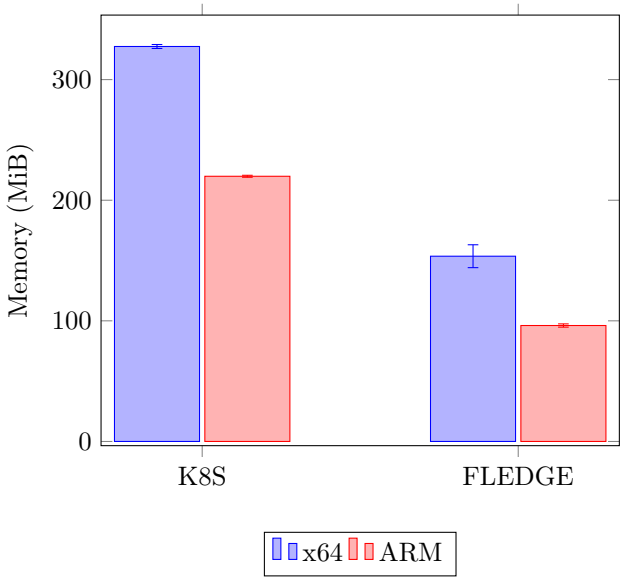


Figure 3.14: Comparison of the memory use of Kubernetes and FLEDGE, both running a kube-proxy deployment.

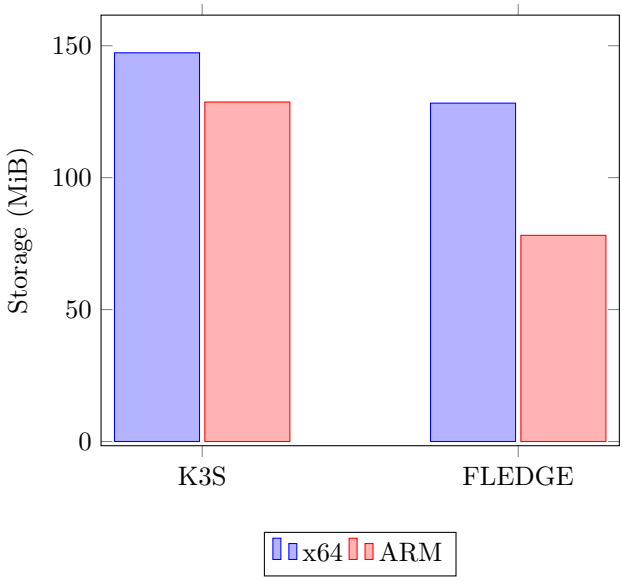


Figure 3.15: Comparison of the storage requirements of K3S and FLEDGE, without kube-proxy.

with both the x64 and ARM versions requiring around 50% less memory than Kubernetes. It is worth noting that simply eliminating Flannel in favor of a custom container networking solution saves around 24MiB of memory on ARM devices and 36MiB on x64.
These results show that FLEDGE, while remaining Kubernetes compatible, uses much less resources and is a viable container orchestrator for edge devices.

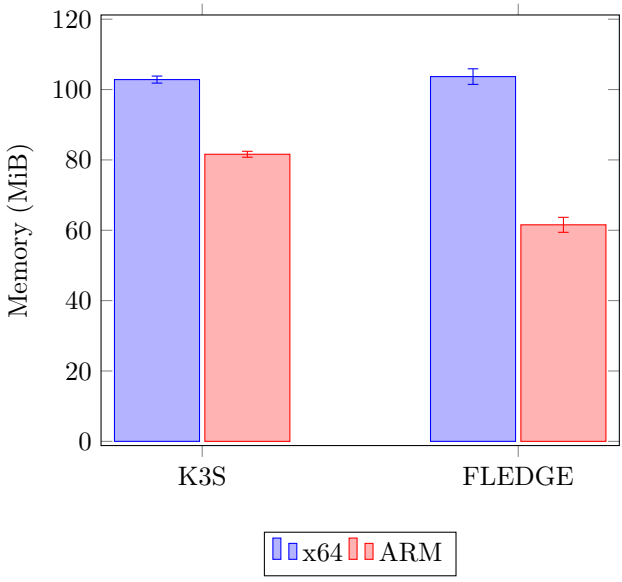


Figure 3.16: Comparison of the memory use of 2019 versions of K3S and FLEDGE.

The difference between K3S and FLEDGE, shown in Fig. 3.15 for storage and Fig. 3.16 for memory, is less impressive. However, FLEDGE still uses about 10% less storage than K3S on x64, and around 30% less on ARM. As far as memory goes, FLEDGE and K3S require more or less equal amounts on x64, but FLEDGE uses 25% less on ARM devices. Combined with the ability of FLEDGE to join Kubernetes clusters, which K3S cannot do, this makes a strong case for using FLEDGE as an edge container orchestrator compared to alternative software.
Fig. 3.17 extends Fig. 3.16 for K0S, ioFog and KubeEdge, and uses orchestrator versions available in 2021. Considering the variety of orchestrators involved, this comparison includes all containers spawned by an orchestrator, except kube-proxy for FLEDGE, which is not a requirement of FLEDGE itself. While the early versions of K3S were very closely matched with FLEDGE, newer versions require up to 75% more memory than its early

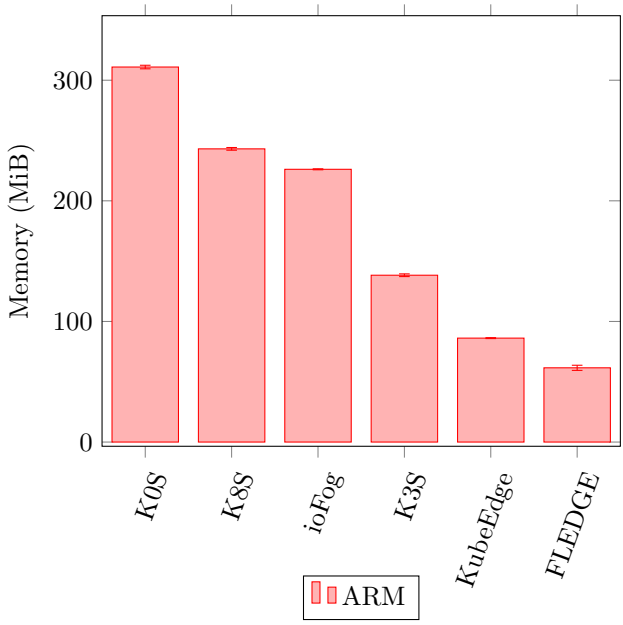


Figure 3.17: Comparison of the memory use of 2021 versions of popular orchestrators and FLEDGE.

versions from 2019, making FLEDGE by far the most memory efficient option. ioFog and K0S, both newer alternatives, require around 4 and 5 times as much memory as FLEDGE, respectively. For ioFog, this is caused by the need for a Java Virtual Machine (JVM) and several plugin containers, while K0S launches a large number of plugin containers which results in a severe overhead. Despite its JVM, ioFog still requires less memory than Kubernetes, while K0S requires significantly more memory due to its various plugin containers. KubeEdge, which uses a custom edge node agent based on lightweight technologies, uses only about 40% more memory than FLEDGE.

Finally, Fig. 3.18 shows the amount of memory used by the 2019 versions of each container orchestrator, without any additional processes. Only in the case of Kubernetes has Flannel been included, since K3S and FLEDGE have pod and container networking by default. This chart shows that while both K3S and FLEDGE require only around 30% of the resources of Kubernetes, FLEDGE is more efficient on ARM devices, while K3S is more efficient on x64.

Fig. 3.19 shows the memory requirements of the 2021 versions of container orchestrators. These results show that the memory consumption of

Figure 3.18: Direct comparison of the memory use of the main process(es) of each container orchestrator for 2019 versions.

orchestrator processes has grown significantly, and that of K3S has doubled since 2019. K0S comes closest to FLEDGE, but still requires almost twice as much memory. ioFog, which needs a Java Virtual Machine for its orchestrator process, needs over three times as much memory as FLEDGE. The main processes of KubeEdge and K0S have similar memory consumption, and comparing to Fig. 3.17 illustrates how the additional containers launched by some orchestrators can significantly add to their total memory use.

3.3.5 Discussion

In the beginning of Section 3.3, a number of requirements are proposed for FLEDGE:

- Secure communications between edge devices and the cloud by default, minimal impact on local networks.
- Compatibility with modern standards for container orchestration, or provide an adequate alternative.
- Low resource requirements, primarily in terms of memory but also in terms of processing and storage.

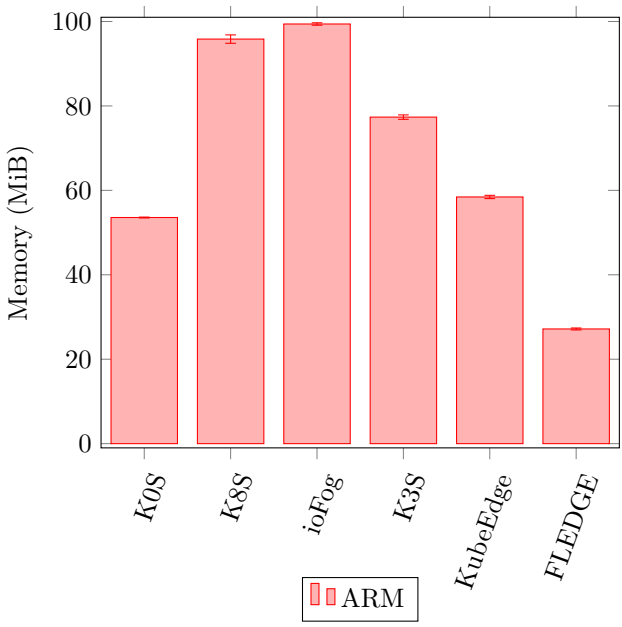


Figure 3.19: Direct comparison of the memory use of the main process(es) of 2021 versions of popular container orchestrators.

FLEDGE is shown to be a solution to these requirements by using Virtual Kubelets and agents on edge devices. A VPN is used to homogenize edge networks and provide a basic layer of security for communication with the cloud. Compatibility with container standards is achieved by using OCI [20] APIs to build FLEDGE. Some standards, such as CNI, can be safely ignored by using a custom implementation which does not impact the rest of the cluster. Low resource requirements are achieved partially by choosing Containerd as a container runtime, and partially through custom implementations of specific functionality such as CNI.

To further illustrate the low resource requirements of FLEDGE, several evaluations are performed. A FLEDGE setup is deployed using both Containerd and Docker, showing that FLEDGE using Containerd requires about half of the memory and storage of FLEDGE using Docker.

Similarly, the choice of running the Virtual Kubelets in the cloud or integrating them into FLEDGE agents is backed up by a theoretical model and an evaluation. The results show that it is preferable to integrate the Virtual Kubelets into FLEDGE agents, reducing overall complexity. On x64 platforms, the total amount of resources required is almost identical for the two solutions, but on ARM the results are slightly better when running

everything on the edge.

Finally, Kubernetes and K3S are discussed as alternatives to FLEDGE, followed by an evaluation to compare them in terms of memory and storage requirements. The results show that FLEDGE requires only about 40-50% of the resources of a similar Kubernetes worker node, while it also requires 25% less resources than K3S on ARM devices. On x64 devices, FLEDGE and K3S resource requirements are nearly equal.

Updated results for 2021 software versions indicate significant increases in resource requirements for all orchestrators, increasing the favorability of FLEDGE as an efficient, minimal orchestration agent.

The conclusion is that FLEDGE, despite its experimental status, can deploy Kubernetes pods while using significantly less resources than other container orchestrators. Several topics for future work are discussed, mostly focused on improving FLEDGE itself.

While this section presents a fully operational container orchestrator for edge devices, there are aspects of FLEDGE that can be improved.

First of all, the integration of the Virtual Kubelet on the edge is not ideal. While it is better than managing each FLEDGE agent with separate pods in the cloud, the ideal solution may be to create a single service in the cloud that can manage hundreds or thousands of FLEDGE agents, scaling up only as required. This approach would be optimal for resource requirements, but it would likely require a lot of processing power and create a single point of failure.

Only Docker and Containerd were considered as container runtimes for FLEDGE, but many others exist, including rkt [60] and CRI-O [61]. Docker and Containerd were chosen because they are widely supported and popular, but it is unknown if another container runtime could give better results.

As orchestrator compatibility goes, K3S and FLEDGE already use both the Kubernetes and Containerd APIs, so with a little extra work it may be possible to have FLEDGE connect to both Kubernetes and K3S clusters, even simultaneously.

While FLEDGE is built to be Kubernetes compatible, it is unknown if optional features such as distributed storage work properly at this point. For the envisioned use of FLEDGE on edge devices, this is not important, but it could prove a valuable addition in the future.

OpenVPN is used to build a homogeneous network environment for FLEDGE to operate in, but other VPN software exists that may be more stable or provide faster connection speeds. Possible alternatives include Tinc, WireGuard and ZeroTier.

In Eq. 3.5, C_M represents the relative cost of edge memory versus cloud memory. In this section, it is naively assumed to be 1, but studies on the

relative cost of edge resources and cloud resources could be interesting for the further development of software and container placement strategies.
The version of Kubernetes used in this section is limited to a maximum of 5.000 nodes and 150.000 pods in total. While this is sufficient for cloud clusters, the maximum number of nodes in particular will be too low for edge clusters. These numbers are not hard-coded, but based on the performance of several subsystems, such as node synchronization and pod status updates. It may be possible to increase the maximum number of nodes by optimizing the configuration of Kubernetes and severely limiting the maximum number of pods on an edge node. Another solution is to federate a number of Kubernetes clusters using KubeFed [62], thereby reducing the impact of the limits of a single cluster.

3.4 Lightweight service orchestration

This section contains the edited version of the following publication: **"Near real-time optimization of fog service placement for responsive edge computing"**, T. Goethals, F. De Turck, B. Volckaert
published in **Journal of Cloud Computing, 9, Article number 34 (2020)** [63]

While the combination of IoT and fog computing offers a wide array of advantages, such as improvements in efficiency and user experience, it also exacerbates some of the service deployment scheduling challenges already present in the cloud, such as taking network bandwidth, network reliability and distances between nodes into account.
Instead of being located in centralized data centers, the fog and edge are spread over a large physical area, containing hundreds of thousands of devices. This means that network grade and quality can vary by orders of magnitude, from DSL lines to fiber optics, while the distances involved result in much higher latencies between nodes than in cloud data centers. A widespread and heterogeneous network also results in a larger variety of network conditions and problems. Therefore, a scheduling solution should not only be able to handle changing network conditions, but also slow or broken lines of communication. The decentralized nature of the fog and the edge is also an important factor. In the cloud, a service can simply be scaled up if demand suddenly spikes. In the fog however, it is not always possible or useful to simply scale in place. In order to minimize access times for the edge and provide the right amount of capacity for each service, the entire fog topology must be taken into account. Because of this, any change in the fog topology can trigger migrations of or extra service instances, as can

edge nodes coming online, going offline, or moving to a different location.
On the other hand, there are also some challenges that remain mostly unchanged from cloud deployments. A deployment scheduler still has to take into account the limited resources of the nodes it can deploy services on, whether those are hardware resources (CPU, memory, network saturation) or calculated load metrics. Furthermore, although the solution should strive for an optimal placement of service instances in the fog to minimize access times for consumers, it should do so efficiently by using a minimal number of service instances. To guarantee a certain level of responsiveness to consumers, one or more metrics and thresholds can be defined on fog nodes[7], for example latency, uptime, etc.
To summarize, the requirements for a good algorithm for fog service scheduling are:

- ***Req. 1*** It should work on the scale of hundreds of thousands of edge devices
- ***Req. 2*** It should be able to handle changing network conditions and topologies in near real-time
- ***Req. 3*** It must take fog node resource limits and distance metrics between nodes into account
- ***Req. 4*** It should minimize the number of instances required for any fog service deployment

This section proposes Swirly as a solution to these requirements. Swirly is an algorithm that runs in the cloud or fog, which plans fog service deployment with a minimal number of instances, while optimizing the distance to edge consumers according to any measurable metric. Furthermore, it can incorporate changes to the network and topology almost in real-time. An example architecture is shown in Fig. 3.20, in which Swirly manages support services for edge devices and deploys them in the fog. Communication between fog services and edge devices is directed by Swirly, but cloud communication and services are independently managed. Examples of cloud functionality which does not benefit from Swirly orchestration include software update services, customer support services (or dashboards) and data collection.
Section 3.4.1 presents existing research related to optimizing service deployments. Section 3.4.2 explains how the proposed algorithm works and how to choose a good metric, while section 3.4.3 analyzes its theoretical

[7] A fog node can be a single device or server, or a decentralized micro data center. Anything that allows the deployment of services outside the cloud proper.

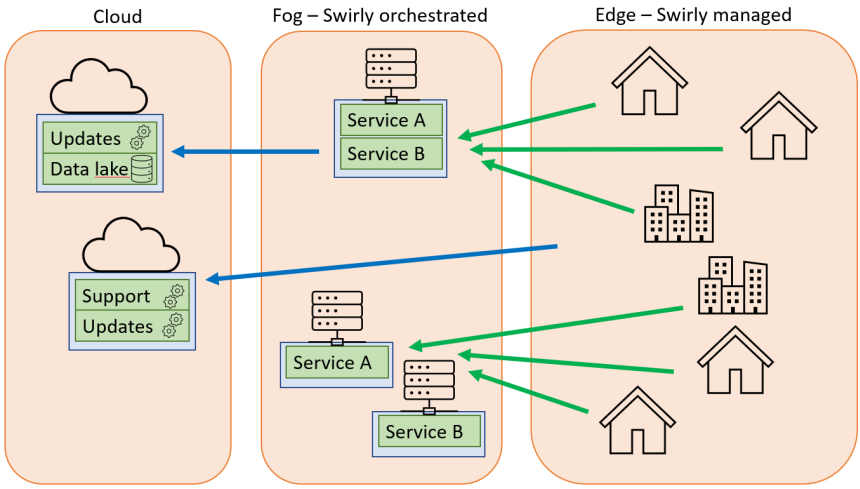


Figure 3.20: Example scenario in which Swirly orchestrates services in the fog, and directs edge nodes to suitable nearby fog services. Cloud services and communication with the cloud are not managed by Swirly.

performance and the shape of the resulting service topologies. In section 3.4.4, an evaluation setup and methodology are presented to verify various performance aspects of the algorithm. The results of the evaluations are presented and discussed in section 3.4.5. Finally, section 3.4.6 gives a short overview of the goals stated in this introduction, and how the algorithm and its properties meet them.

3.4.1 Related work

Shifting workloads between the cloud and edge hardware has been extensively researched, with studies on edge offloading [22], cloud offloading [23], and osmotic computing [26].

Many strategies exist for fog container deployment scheduling, ranging from simple but effective resource requests and grants [27], to using deep learning for allocation and real-time adjustments [28].

Initial research into fog computing and service scheduling dates from before the concept of the fog, for example Oppenheimer et al. [64], who studied migrating services in federated networks over large physical areas. This work takes into account available resources, network conditions, and the cost of migrating services between locations in terms of resources and latency.

Zhang et al. [65] present an algorithm for service placement in geographically distributed clouds. Rather than focusing on resources as such, their algorithm makes placement decisions based on changing resource pricing of

cloud providers.
Aazam et al. provide a solution for fog data center resource allocation based on customer type, service properties and pricing [66], which is also extended to a complete framework for fog resource management [67].
In more recent research, Santos et al. [68] present a Kubernetes-oriented approach for container deployments in the fog in the context of Smart Cities. Their solution is implemented as an extension to the Kubernetes scheduler and takes network properties of the fog into account.
Artificial intelligence is also making headway into fog scheduling research. For example, Canali et al. [69] tackle fog data preprocessing with a solution based on genetic algorithms. Their solution distributes data sources in the fog, while minimizing communication latency and considering fog node resources.
Zaker et al. [70] propose a distributed look ahead mechanism for cloud resource planning. Rather than provisioning more resources to counter network load, they attempt to optimize bandwidth use through the configuration of overlay networks. The predictive look ahead part is implemented by using the IBK2 algorithm. This is different from the approach in this section, which does not consider network load by itself, and attempts to migrate service deployments to manage resources.
Finally, Bourhim et al. [71] propose a method of fog deployment planning that takes into account inter-container communication. Their goal is to optimize communication latencies between fog-deployed containers, which is obtained through a genetic algorithm.
Most of these approaches are centered on the cloud or small scale fog networks, and use a large number of parameters to construct an optimal, but static solution. In some cases, they may also rely on historical data for training. The algorithm discussed in this section however aims to quickly construct solutions using an edge-centered approach, taking into account node resources and generic heuristic. The speed of the algorithm allows it to process node updates in near-realtime for fog and edge networks orders of magnitude larger than those commonly found in proofs of concept in related work. An additional benefit is that the heavy lifting of calculating the heuristic value is offloaded to edge devices, where it has far less impact due to being spread out. Finally, the solution is meant for dynamically evolving networks for which historical training data may be hard or impossible to acquire.

3.4.2 Swirly

Contrary to the fog deployment solutions discussed in the previous sections, Swirly works under the assumption that some fog services will be used by

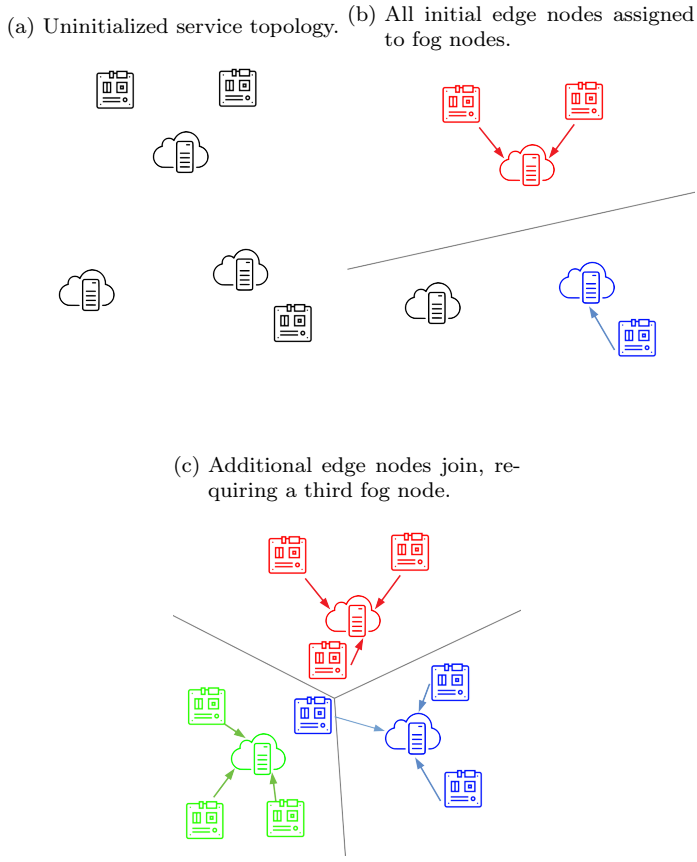


Figure 3.21: Different stages of building a service topology with Swirly.

most, if not all, devices in the edge. This allows for a simple but flexible approach which is very suited to building large service topologies.

Throughout the remainder of this section, a fog network (including the edge) with frequent changes to its network and nodes will be referred to as a swirl. This term refers to the swirly motion which fog makes when it stirs and moves. Hence the name of the algorithm, Swirly, which attempts to build an optimal service topology in a swirl. Additionally, edge nodes are devices at the network edge which act as consumers of fog services, while fog nodes are service providers hosting fog services. Therefore, edge nodes are assigned fog nodes as service providers, or are serviced by fog nodes. Finally, a (service) topology refers to the result of the algorithm, in which all edge nodes are assigned a fog node. When referring to the physical layout

Table 3.1: Definitions of symbols used in algorithms 1, 2 and 3.

Symbol	Definition
E	all edge nodes in the service topology
F	all fog nodes in the swirl
E_x	an edge node in the service topology
$\|E\|$	the number of items in E
F_x	a fog node in the service topology
$\|F\|$	the number of items in F
e	an edge node without a service provider
$F(e)$	fog nodes ordered by distance from e
$E(F_x)$	edge nodes serviced by F_x
A	set of active fog nodes
$D_{e,f}$	distance between edge node e and fog node f
D_e	distances of edge node e to all fog nodes
d	a distance according to the chosen metric
d_{max}	maximum distance between an edge node and fog node
$R_{x,f}$	current level of resource x on fog node f
$RL_{x,f}$	limit for resource x on fog node f
$LL_{x,f}$	lower limit for resource x on fog node f
RD_x	default resource x increase for a service client

of the input nodes, the term node topology is used. Table 3.1 defines all the symbols used in algorithms in this section.

Fig. 3.21 illustrates how Swirly forms a service topology from a collection of edge nodes E and fog nodes F. The algorithm starts with a number of unassigned edge nodes. It then determines that these nodes are all within an acceptable distance of two fog nodes, which are initialized and used as service providers. The line indicates how the service topology is divided between these two nodes. As more edge nodes join the service topology, it becomes necessary to initialize the third fog node, further dividing the service topology. Finally, an edge node pops up which should be serviced by the green fog node, which is already full. Therefore, this last edge node is serviced by the blue fog node.

Fig. 3.23 shows the result of Swirly on a large scale. Edge nodes have been colored according to the fog node which acts as their service provider, while fog nodes themselves are shown as red dots (inactive) or green dots (active). When Swirly is started, it has a collection of fog nodes and their available resources. No further information is needed, apart from an IP address or another effective method of reaching them.

The rest of this section will describe how the described functionality is implemented by specific functions of Swirly. The add operation is meant only

to assign a service provider to newly detected edge nodes, while the update method is used to receive updates from edge nodes and potentially assign them a different service provider. Finally, the delete method removes nodes from the topology altogether. Fog node add, update and delete operations are also discussed in the relevant sections.
Throughout all these operations a distance metric is required to determine which fog node is the best service provider for an edge node. The effects of the choice of a distance metric are discussed in section 3.4.2.3.

3.4.2.1 Adding new nodes

In order to build a service topology, all edge nodes E that require a certain service are added to the topology one by one as per Algorithm 1. Generally, the algorithm attempts to find the active fog node F_a closest to the given edge node e using the list of distances D_e. If successful, the fog node F_a is assigned to the edge node as service provider. If there is no active fog node yet ($A = \emptyset$), or there is no fog node with free resources ($F_a = \emptyset$), Swirly finds the closest inactive fog node F_i instead. That fog node then gets activated and assigned as a service provider to the edge node e. There is a single caveat here; if the closest available fog node is beyond the maximum distance ($D_{e,F_a} > d_{max}$ and $F_i = F_a$), the algorithm has no choice but to assign it as service provider for an edge node. The support function ClosestFogNode returns the fog node F_c with free resources closest to an edge node e. A parameter *active* can be supplied to indicate if only active fog nodes should be considered.
This operation ensures that ***Req. 3*** and ***Req. 4*** for a useful deployment scheduler are met within a reasonable amount of processing time.
Note that during an add operation, the resource use $R_{x,i}$ of the selected fog node is increased by a configurable amount. This is to avoid assigning it to too many edge nodes simultaneously, and is periodically corrected by updates from the fog nodes containing their actual free resources.
Adding additional fog nodes to the topology is also supported, but this does not trigger a reorganization of currently assigned edge nodes. Rather, newly added fog nodes will only provide services to edges nodes that are added or updated after they were initialized.

3.4.2.2 Node updates

In order to fulfil ***Req. 2***, Swirly must support topology updates. As a requirement for these updates, all edge nodes must periodically report the distances between them and every fog node to Swirly. Exactly how they do this is left to implementation, although some suggestions and their impacts

```
function Add(e, distances, d_max)is
    if A = ∅ then
        D_e = distances
        F_i = ClosestFogNode(e, false)
        initialize F_i
        assign F_i to e
    else
        F_a = ClosestFogNode(e, true)
        if F_a = null then
            F_i = ClosestFogNode(e, false)
            add F_i to A
            initialize F_i
            add e to E(F_i)
        else if D_{e,F_a} > d_max then
            F_i = ClosestFogNode(e, false)
            if F_i = F_a then
                add e to E(F_a)
                R_{x,a} += RD_x
            else
                initialize F_i
                add e to E(F_i)
            end
        else
            add e to E(F_a)
            R_{x,a} += RD_x
        end
    end
end
function ClosestFogNode(e, active)is
    F_c = null
    for F_x ∈ F(e) do
        if (!active or (active and F_x ∈ A)) and R_{x,i} < RL_{x,i}, ∀i
         then
            F_c = F_x
    end
    return F_c
end
```

Algorithm 1: Adding a single edge node to the service topology

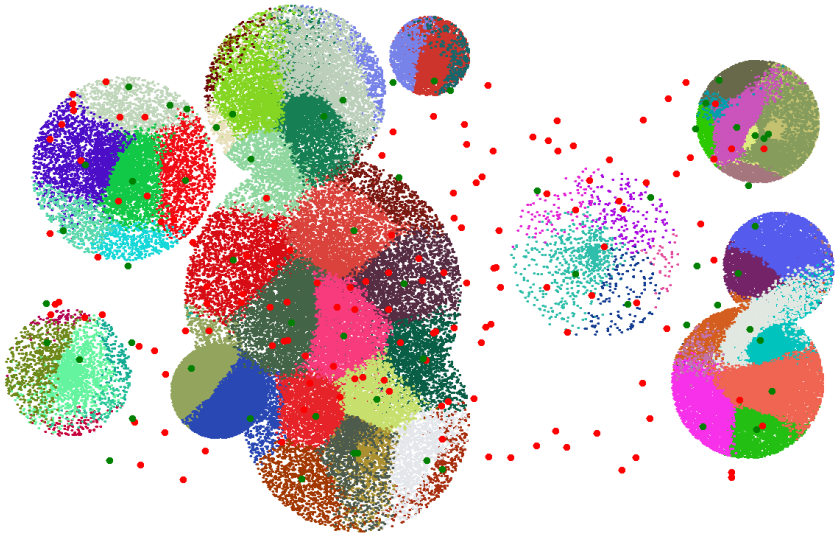


Figure 3.23: Visualization of a service topology generated by Swirly. Big red dots are inactive fog nodes, big green dots are active fog nodes servicing nearby edge nodes.

are given in the next subsection. For this subsection, it is important to note that these lists of distances to fog nodes are pre-sorted by increasing distance, so Swirly can always find the closest fog node in constant time. This is also the case for the ClosestFogNode function in Algorithm 1.

To keep a service topology up to date in a swirl, the algorithm needs operations to update edge nodes (Algorithm 2) and remove them from the service topology (Algorithm 3).

The Remove operation starts by removing the edge node e from its fog node F_e. After that, it checks if any of the resources of the fog node are below the lower limit, in which case it attempts to move all remaining edge nodes $E(F_e)$ it services to other nearby fog nodes by removing them from F_e and calling the Add operation. This process fails if any edge node E_x would be assigned a new fog node F_{alt} which is more than the maximum distance d_{max} away, unless that is already the case for the current fog node ($D_{E_x,F_{alt}} > d_{max}$). To support reverting this operation in case of failure, each reassignment is kept in the map $altNodes$. Upon failure, the algorithm iterates over each pair E_i,F_i in $altNodes$, removing E_i from F_i and reassigning it to F_e. On success, the fog node F_e is torn down and removed from the topology.

The Remove operation makes use of a support method ClosestFogNodeExcept, which is essentially the same as the ClosestFogNode method, but the

fog node F_{except} can not be returned as a result.
Note that this process produces the same result as if the exact subset of nodes that absolutely required a specific fog node as service provider had never been in the original set of edge nodes, so it remains consistent with the Add operation.
The Update operation updates the set of *distances* of an edge node e to each fog node in F. In case the new distance d_{new} from the edge node to its service provider F_e increases beyond the maximum distance d_{max}, the algorithm calls the Remove and Add operations for e, in an attempt to assign it a better service provider.
Note that the total performance of the update method is dependent on how efficiently the set of distances can be updated. However, this can happen in constant time, which is implicitly assumed in section 3.4.3.1.
The distance metric, combined with the Update and Remove operations not only enables Swirly to act on topological changes, but also to implicitly avoid fog nodes which are experiencing load spikes and network issues.
As with adding fog nodes, fog node updates only change the available resources for further edge node assignments. Removing a fog node will attempt to assign new fog nodes to the edge nodes that depend on it. Reassigning edge nodes if their service provider suddenly runs out of resources is not currently implemented. Instead, it can be argued that it is optimal to rebuild the entire service topology when fog nodes are added or their available resources change drastically, since these cases require examining every edge node to find its optimal service provider considering the new information. Therefore, no further implementation is needed.

function *Update(e, distances, d_{max})* **is**
    $d_{old} = D_{e,F_e}$
    D_e = distances
    $d_{new} = D_{e,F_e}$
    if $d_{old} < d_{new}$ *and* $d_{new} > d_{max}$ **then**
        Remove(e, d_{max})
        Add(e, d_{max})
end

Algorithm 2: Updating the status and distance metrics of an edge node

Swirly can not directly detect fog node failures, so it can not actively react to service availability issues. However, its design allows for two methods to make it more resilient to hardware failures. The first is choosing a distance metric that can reflect imminent node failures, which passively forces the algorithm to choose more suitable fog nodes to host services on, as shown in section 3.4.2.3. The second option is to actively remove a fog node from the

```
function Remove(e, d_max) is
    remove e from E(F_e)
    if R_{e,i} < LL_{e,i}, ∃i then
        altNodes = ∅
        revert = false
        while !revert and E(F_e)! = ∅ do
            E_x = E(F_e)[0]
            F_alt = ClosestFogNodeExcept(E_x, F_e)
            if F_alt = null then
                revert = true
            else
                if D_{E_x,F_alt} > d_max then
                    revert = true
                else
                    remove E_x from E(F_e)
                    Add(E_x, d_max)
                    altNodes[E_x] = F_alt
                end
            end
            if revert then
                for (E_i, F_i) ∈ altNodes do
                    remove E_i from E(F_i)
                    add E_i to E(F_e)
                end
            else
                teardown F_e
            end
        end
    end
    function ClosestFogNodeExcept(e, active, F_except) is
        F_c = null
        for F_x ∈ F(e) do
            if F_x ≠ F_except and (!active or (active and F_x ∈ A)) and
              R_{x,i} < RL_{x,i}, ∀i then
                F_c = F_x
        end
        return F_c
    end
```

Algorithm 3: Removing a single edge node from the service topology

algorithm when its failure is detected by external components. While this method requires some extra computation, it can react to hardware failures in less than a second.
So far, this section has not touched on the actions required to redirect service requests from edge nodes to the correct fog nodes. While such functionality is beyond the scope of this section, the network addresses of all nodes are known, along with the topology generated by Swirly. Therefore, it should not be overly difficult to propagate changes to a DNS server, a distributed DNS plugin (e.g. for Kubernetes), or a webservice on edge nodes that redirects requests at the source.

3.4.2.3 Impact of distance metric

While the performance and inner logic of the algorithm are unaffected by the choice of distance metric between edge nodes and fog nodes, a good metric can improve efficiency and responsiveness to changes. On the other hand, some of the more useful metrics may cause a lot of processing and network overhead between the nodes and the algorithm. In this section, some ideas are discussed for useful metrics.
The simplest metric that can be used depends only on geographical coordinates. While it requires a reasonably accurate location for each node, it does not usually change unpredictably or rapidly. The downside of this metric is that any changes in network or fog node performance can not be detected in order to avoid availability problems. The advantage is that the algorithm can keep track of all node locations by receiving regular updates from edge nodes, and calculate distances as required. Thus, the network overhead will be minimal using this approach.
Another possible metric is the latency between edge nodes and fog nodes. The values of this metric can be easily measured using the ping command, but this results in an overhead that grows linearly with both edge nodes and fog nodes. Additionally, the ping command is often blocked on servers or routers, in which case it is useless. The biggest advantage of this metric is that it can detect network and fog node issues in real-time, so edge nodes can be assigned a different fog node as service provider.
A third metric, which also aims to determine network latency, uses a very lightweight web service on both edge nodes and fog nodes to determine the latency between software service endpoints. The disadvantages of this approach are that the packet sizes are larger than those of a simple ping, and that it requires slightly more processing time. However, the evaluations from Chapter 2.2 show that even a reasonably simple server can easily handle millions such requests per minute.
The last two metrics require that all edge nodes periodically determine their

distance to each fog node, and report the results to the algorithm so it can adjust the service topology. In order to show that this does not result in an unacceptably high network overhead, the following numbers have been determined:

- The example assumes 200000 edge nodes, using 200 fog nodes as service providers
- Each edge node will attempt to determine its distance to fog nodes once every minute
- The size of a ping packet is 56 bytes on Unix
- wget shows that a suitable web service request is 159 bytes and a response is 202 bytes

Using these numbers, each fog node has to process about 3333 ping requests per second for a total of 1.5Mbps, both incoming and outgoing. In the case of a webservice, the traffic increases to 4Mbps incoming and 5Mbps outgoing. Additionally, to avoid overloading nodes that are already under heavy load and to avoid frequent distance measuring to nodes that are too far away, the frequency can be reduced by an order of magnitude for fog nodes more than two or three times the maximum distance away. For larger networks, this should reduce total traffic considerably. However, no concrete numbers for this can be determined since they are fully dependent on the network topology.
Finally, a quick calculation can determine the network overhead for the server hosting Swirly using:

$$T = 8S \cdot \frac{|E| \cdot |F|}{P} \tag{3.7}$$

Where P is the measuring period in seconds and S is the message size in bytes (15 for IP address + 4 for an integer number). The result is 98Mbps, which is significant but not insurmountable. Some actions can be taken to reduce this number significantly, such as not reporting distances that have not changed by more than 30%, unless they cross the maximum distance. For geographically widespread topologies, this could likely reduce traffic by an order of magnitude or more, but again concrete numbers can not be determined as they rely on the specific network topology.

3.4.3 Theoretical properties

To fulfil part of ***Req. 1*** put forth in the introduction, this section discusses the theoretical properties of the algorithm. The processing power

and memory requirements are analyzed in-depth, and for a full understanding of the output of the algorithm, a theoretical model for the resulting service topologies is constructed.

3.4.3.1 Processing

Adding an edge node to the topology can result in several cases. In all cases, the sorted list of fog node distances for the edge node is consulted to determine its optimal service provider at that time.
In the most common case, the selected fog node has enough free resources for additional service clients, and the edge node is simply directed to that fog node. The resulting performance is $O(1)$.
In the worst case, there is a chance that the optimal fog node is already full and the algorithm has to search for the next best fog node with free resources. Let there be a chance p that any node is already full, such that

$$p = c\frac{|E|}{|F|} = cL_F \tag{3.8}$$

where L_F is the average load of edge nodes assigned to fog nodes and c is a constant that normalizes L_F. If there are enough edge nodes E and fog nodes F, then the expected number of iterations to find a suitable fog node is

$$r = \frac{1}{1-p} = \frac{1}{1-c\frac{|E|}{|F|}} \tag{3.9}$$

and worst case performance is $O(1/(1-|E|/|F|))$, with a maximum of $O(|F|)$ because there can never be more iterations than there are fog nodes.
Note that both cases depend on the node topology of the swirl. When there are not enough fog nodes in areas with a high edge node density, the worst case performance will occur more often. The influence of the node topology on performance is further examined under 3.4.3.3.
Like the Add operation, deleting an edge node has several possible cases. In the best case, it is removed from its fog node for $O(1)$.
In some cases, the fog node will be underutilized after a remove, so the algorithm attempts to migrate the remaining edge nodes of that fog node to other fog nodes. Since the utilization threshold is independent of E and F, this operation is $O(1)$, albeit with an unusually large impact. This heavy operation is amortized over $O(|E|/|F|)$ removes and uses the add operation when moving edge nodes, resulting in a worst case performance of $O(F/(1-|E|/|F|))$.
Finally, the update operation can cause a different fog node being assigned to an edge node. When this happens, both a remove and add operation are

Table 3.2: Summary of algorithm operation complexity. Most common cases are marked in bold.

	Best	Worst
Add	$\boldsymbol{O(1)}$	$O(1/(1-\lvert E\rvert/\lvert F\rvert))$
Remove	$\boldsymbol{O(1)}$	$O(\lvert F\rvert/(1-\lvert E\rvert/\lvert F\rvert))$
Update	$\boldsymbol{O(1)}$	$O(\lvert F\rvert/(1-\lvert E\rvert/\lvert F\rvert))$

executed. This can lead to the worst case scenario for both, in which case performance is $O(|F|/(1-|E|/|F|))$. Most updates however will only be a simple status update for $O(1)$.
Table 3.2 summarizes the performance for all operations.

3.4.3.2 Memory

Swirly requires a number of maps and lists to support the processing performance described above, but the greatest impact on memory is that for each edge node, the algorithm must keep a sorted list of distances to each fog node in the topology. Therefore, the predicted memory requirement for the node hosting the algorithm is $O(|E| \cdot |F|)$. For edge nodes, the memory requirement is $O(F)$.

3.4.3.3 Generated topology

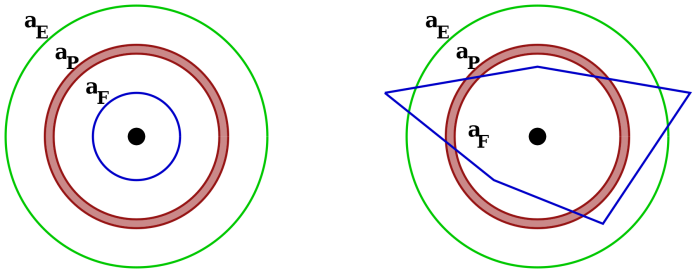
To verify that the algorithm can satisfy the requirements for a good fog service topology and to identify edge cases and possible problems, it is important to first construct a theoretical model of the expected output given the node topology of the swirl.
While fog and edge networks are intrinsically discrete, they can be described analytically if they are large and dense enough. As a first step, the densities of edge nodes and fog nodes in any network are

$$\rho_F = f(x, y), \rho_E = g(x, y) \tag{3.10}$$

where both f and g are functions that give the amount of nodes per surface area at x, y. While Cartesian coordinates are used here, it would be better to use densities in the coordinate system that describes distances in the chosen metric. However, the coordinate transformation may be unknown and impossible to construct, so Cartesian is used for illustrative purposes.
For the next step, the servicing area of every active fog node can be modeled using three different parameters:

(a) Idealized representation of fog node service areas.

(b) Realistic responsibility area with sections $\notin a_E$.



(c) Example of realistic responsibility area created by algorithm.

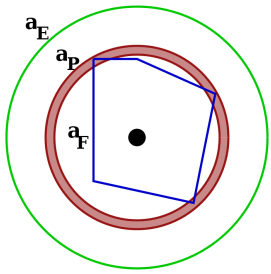


Figure 3.24: Graphic representation of idealized and realistic fog node service areas.

- a_E, circumscribed by r_E, is the capacity area of a fog node, which determines how many edge nodes it can service based on its capacity C_e
- a_F, bounded by r_F, is the responsibility area of a fog node. All edge nodes within this area should be serviced by the fog node, since it is either the closest fog node or no other fog node can be used.
- a_p, bounded by r_p, is the proximity area of a fog node. All edge nodes within this area are close enough that they can be serviced by the fog node without going over the maximum metric value.

In some cases r_E, r_F and r_p will have only a single value, such as in Fig.

3.24a. In others, only their maximum value is relevant, such as in Fig. 3.24b where $r_F > r_E$. In these cases they will be treated as scalars to simplify notation. Other cases will specifically show them as functions with their required parameters.
r_p can be determined using the following equation, which in most cases will resemble a circular shape:

$$r_p(x, y) = \sqrt{(x^2 + y^2)}, \forall x, y : h(x, y) = M_m \tag{3.11}$$

or, for a more intuitive approach using polar coordinates, where r can be substituted for metric distances

$$r_p(\theta) = r, \forall r : h(r, \theta) = M_m \tag{3.12}$$

Where $h(x, y)$ gives the distance metric value at any given location x, y and M_m is the maximum distance value. Note that in both cases, points at various distances from the origin can be mapped onto the same metric distance, making a transform back to the original coordinate system impossible. In the case of certain basic distance metrics, in which metric distance between points divided by their geographical distance is more or less constant, the equation reduces to

$$r_p = C_p M_m \tag{3.13}$$

r_F and r_E can be naively described as circle radii using

$$r_F = \frac{1}{\sqrt{\pi \rho_F}}, r_E = \sqrt{\frac{C_e}{\pi \rho_E}} \tag{3.14}$$

However, the equation for r_E is not accurate when ρ_E varies a lot over its entire area. A more accurate solution would be solving the following equation for r to find r_E:

$$\int 2\pi r \rho_E(r) \cdot \mathrm{d}r = C_e \tag{3.15}$$

This formula takes into account that only r is known by measuring the metric distance between nodes and not θ, so only the integration over r is performed. It is worth noting that in many cases the density of edge nodes is relatively constant over most geographical areas that can be covered by a single (group of) fog node(s), although it may vary over time. Therefore Eq. 3.14 can be used if it is re-evaluated periodically.
Finally, r_F is unlikely to vary significantly at the scales used in this section. When plotted, these three radii resemble Fig. 3.24a. Note that r_p is quite fuzzy because as mentioned, $h(x, y)$ can map physically different points on

significantly different values of the distance metric. Fig. 3.24a represents the ideal case for the relative sizes of the radii. There are 5 other permutations, all of which can be problematic for the following reasons:

- If $r_F > r_E$, the fog node will not have enough capacity to handle its entire responsibility area. This means there are not enough active fog nodes, or they are not in the right places.
- If $r_F > r_P$, the fog node will have to support some nodes that fall outside its proximity area, so that some edge nodes will have a greater metric value than technically allowed. Again, this points to not enough active fog nodes or erroneously placed fog nodes.
- If $r_p > r_E$, the fog node has sufficient capacity to handle its responsibility area, but can not handle a changing topology where it has to start servicing extra edge nodes that fall between r_E and r_P. This means the fog nodes are not sufficiently powerful to support the maximum metric value.

Assuming a basic metric, two requirements for the proper formation of a fog service network can be constructed from these cases by substituting r_p, r_F and r_E with their definitions from Eq. 3.13 and 3.14:

$$M_m^2 \rho_F(x,y) \geq \frac{1}{\pi C_p^2} \tag{3.16}$$

$$C_e \rho_F(x,y) \geq \rho_E(x,y) \tag{3.17}$$

The left sides of these requirements represent the factors that can be easily controlled or tuned, while the right sides represent factors that are unpredictable or unavoidable. Since everything in Eq. 3.16 except $\rho_F(x,y)$ is constant under the assumptions, it can be used to determine a proper minimum value for ρ_F, with Eq. 3.17 indicating where more fog nodes or better hardware should be provided.

A final topic of discussion concerns the shape of a_F. In reality, r_F will not be a single number describing the radius of a circle. Because the algorithm attempts to assign edge nodes their closest fog node, the responsibility area of fog nodes will look like pieces of a Voronoi diagram. This effect is also visible in Fig. 3.22c and 3.23. As Fig. 3.24b shows, it is possible that the responsibility area a_F has sections that lie outside a_p or a_E. The algorithm only allows sections outside a_p when there is no closer fog node, and it never allows sections outside a_E, even if there is no alternative. In general, it will attempt to create responsibility areas as shown in Fig. 3.24c.

Note that the shapes of a_F depend entirely on the node topology of the swirl and the amount of required fog nodes to service all edge nodes, so any further discussion of the service topology is reserved for the results section.

3.4.4 Evaluation methodology

To fully verify that Swirly fulfills the ***Req. 1***, its performance must be evaluated. This chapter describes the physical hardware setup used to evaluate Swirly, as well as implementation details for the evaluated version. It also details how the theoretical model constructed in the previous section can be evaluated and validated, and how to determine the practical performance of the algorithm.
Swirly is evaluated on a single machine with 48GiB RAM and a Xeon E5-2650 CPU at 2.6GHz. In all cases, the algorithm is the only process running apart from the operating system. Each evaluation was run with 50000 to 400000 edge nodes in steps of 50000, and 50 to 550 fog nodes in steps of 50. However, because the algorithm considers resource limitations, some results series start at a higher number of fog nodes. For example, it is not possible to service 300000 edge nodes with less than 400 fog nodes. For every parameter set, 20 iterations of Swirly are run on a uniquely generated swirl. The maximum distance between edge nodes and fog nodes is set to 100. It is entirely possible that edge nodes are generated which do not have a fog node within the maximum distance, so the evaluations and results are entirely focused on average distance as an indicator.

3.4.4.1 Algorithm implementation

The topology visualization in Fig. 3.23 was made with a .NET implementation of Swirly. For the evaluations in this section, Swirly is implemented in Golang. Because the goal is to measure the impact of the algorithm itself, there are no integrations with any sort of DNS or service/container scheduling software.
Edge nodes and fog nodes are generated randomly over an area of 1200 by 800 ”units”. In order to simulate urbanized areas, edge nodes are generated in circles of varying sizes, which are slightly denser in the center. These circles may overlap and often form more complex shapes, as in Fig. 3.23. Fog nodes are generated without regard for edge node density, to evaluate the ability of Swirly to pick exactly the right nodes to service any area. The chosen distance metric is latency. For simplicity, latency is defined so that one unit generally equals 1ms. However, because latency is inherently fuzzy, this distance is randomized between 80% and 120% of the unit distance.
Because the sets of generated nodes are completely randomized, some cases

will work well with Swirly and others will be adversarial. A range of possibilities is explored and discussed in section 3.4.5.
The Golang implementation of Swirly and the evaluation code are made available on Github[8].

3.4.4.2 Processing time

To accurately measure the exact time it takes Swirly to perform operations, a swirl is generated up front and the Golang time library is used to determine how long an operation using that swirl takes. For the add operation, the evaluation measures how long it takes to build an entire service topology from scratch, which is then averaged to the time it would take to add 10000 nodes. For the remove operation, it is simply measured how long it takes to remove 10000 nodes from a completed service topology. The performance of the update operation is measured similarly to the delete operation. 10000 nodes are physically moved far away from their original fog node and it is measured how long it takes the algorithm to assign them another one.

3.4.4.3 Memory requirement

The evaluation of memory requirement starts with having Swirly build a service topology from a swirl with a certain amount of nodes. Memory consumption is then read from /proc/<pid>/statm and printed to stdout, where it is collected by a batch script.

3.4.4.4 Topology efficiency

Throughout the processing evaluations, some statistics are kept on how many fog nodes are required to build any service topology and what the minimum, average and maximum distances between edge nodes and fog nodes are. These statistics are used to attempt to determine how close Swirly comes to constructing a perfect service topology. Using edge node to fog node distances, Swirly is compared to a random selection of fog nodes (e.g. the default Kubernetes scheduler [5, 72]), and to the best possible theoretical solution if fog node resources are infinite. Additionally, the amount of fog nodes used by Swirly is compared to an optimal solution where each fog node is exactly at maximum capacity, although admittedly this optimal solution would not have any space left for extra nodes or unexpected load. Finally, two extreme topology types are compared to see how Swirly reacts to certain geographical features. The first is a perfectly equal distribution of edge nodes, while in the second the edge nodes are split into 4 circular, non-overlapping clusters, one in each corner of the topology.

[8] https://github.com/togoetha/swirly

3.4.5 Results

This section contains the results for the evaluations described in section 3.4.4, along with a discussion of the results. Most charts have whiskers to indicate extreme values, but in some cases they have been cut off to keep the charts readable.

3.4.5.1 Processing

Fig. 3.25 shows the average time required to add 10000 nodes of swirls of various sizes to a service topology. The results mostly adhere to the computational complexity calculated in section 3.4.3.1. As the number of edge nodes increases the operation gets slower, and it gets faster again with more fog nodes available, eventually leveling out at a constant time per edge node. However, for higher numbers of edge nodes the constant time never quite reaches that of smaller topologies, putting real performance somewhere between the best and worst cases, increasing sublinearly with the amount of edge nodes.
It should be noted that ρ_E is an important factor here; if the physical size of the swirl were to expand with the number of edge nodes, the time required would remain almost constant. In terms of the model constructed in section 3.4.3.3, this is because r_E shrinks as edge node density increases but r_p and a_F remain the same, so eventually r_E will become smaller than r_p and a_F. This means the fog nodes no longer have enough capacity to handle their service areas properly, so the algorithm requires ever more time to find a suitable fog node.
The result of this is that Swirly is suitable for constructing large service topologies, but it requires a large amount of high capacity fog nodes in densely populated areas to keep performance up.
Finally, the whiskers indicate that depending on the node topology of the swirl, the time required to add edge nodes can vary from 50% to 300% of the average with a high $|E|/|F|$ ratio, but it stabilizes as the number of fog nodes increases.
Fig. 3.26 shows the time required to remove 10000 edge nodes from service topologies of various sizes. The performance of this operation is almost ideal, showing a slight decrease with the number of fog nodes and only marginally increasing with the number of edge nodes. Since the edge node density effect is also present here, the results imply that individual remove operations only rarely trigger a worst case performance scenario.
As with the add operation, performance can vary wildly from around 50% to 200% of the average.
The performance of the update operation is shown in Fig. 3.27. Again,

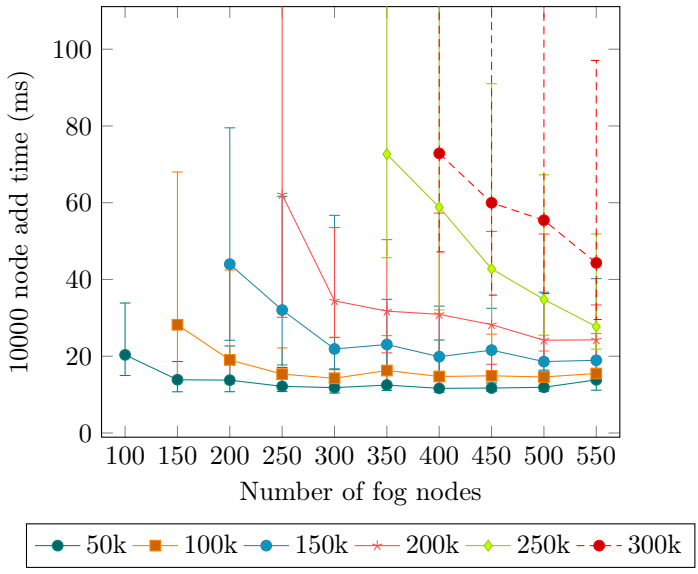


Figure 3.25: Time required to add 10000 edge nodes to service topologies of varying sizes. Legend numbers represent thousands of edge nodes.

this mostly adheres with the theoretical performance, which is the sum of the delete and add operations. The relatively constant performance of the delete operation reduces the curves of the add operation, resulting in a mostly constant time to update edge nodes. However, the performance features of the add operation still apply, as performance decreases slightly with edge node density, and adding more fog nodes improves performance up to a certain point. Note that the update method is forced into worst-case behaviour for this evaluation; every edge node is moved to another fog node, whereas in reality this will not always be the case and performance will be more constant. Finally, the numbers of Fig. 3.27 are not exactly the sum of the add and delete operations, but this is due to certain effects of the evaluation code which can not be subtracted from the measured time.

Predicting a maximum number of devices from these numbers is difficult, since the amount of required node updates in a real-life topology depends on many factors, for example the volatility of the clients, choice of distance metric and update period. Under the extreme conditions that an update is sent by every node every second, and extrapolating from the results, a maximum number of nodes around 200.000 to 300.000 edge nodes can be supported when running single-threaded on the test hardware.

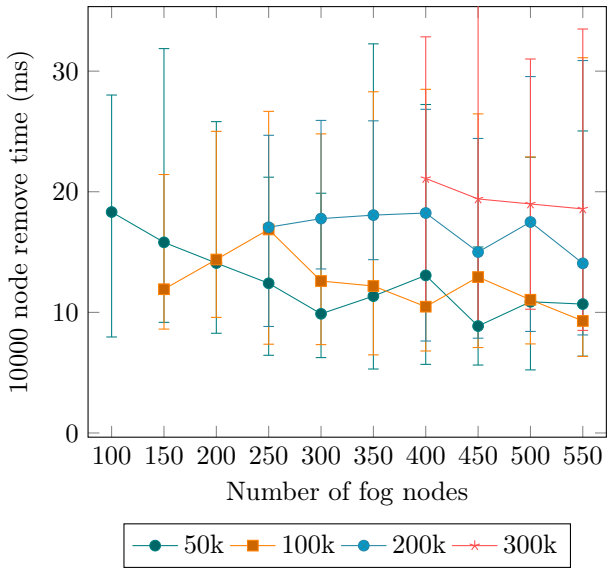


Figure 3.26: Time required to remove 10000 edge nodes from service topologies of varying sizes. Legend numbers represent thousands of edge nodes.

3.4.5.2 Memory

The memory requirements of Swirly are shown in Fig. 3.28 for swirls of varying sizes. The first observation here is that memory use jumps up in distinct steps here. This is caused by the specific implementation of the algorithm in Golang. Golang arrays and maps double in size each time they reach their full capacity, and for each edge node the algorithm keeps a list of distances to fog nodes. Therefore, all these maps double in size at the exact same time, causing the jumps in the chart.
Apart from this peculiar effect, memory use adheres perfectly to the theoretical predictions, and despite the randomly generated swirls there is almost no difference in memory use between iterations.
Considering these results, the product of the number of edge and fog nodes should remain below 1.500.000.000 on a common cloud server with 64GiB memory. For example, 1.000.000 edge nodes can be assigned to 1.500 fog nodes, or 3.000.000 edge nodes to 500 fog nodes.

3.4.5.3 Topology

Fig. 3.29 compares the average edge to fog distances of topologies generated by Swirly (100k, 150k) to theoretically ideal service topologies (100kmin, 150kmin) and service topologies achieved by choosing fog nodes at random

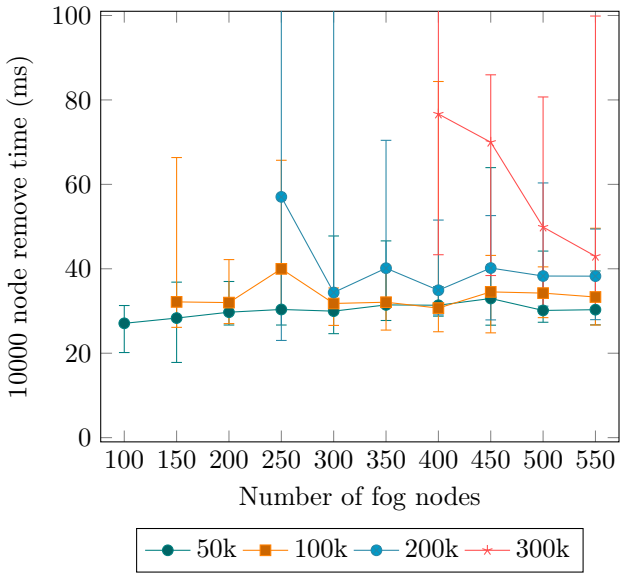


Figure 3.27: Time required to move 10000 edge nodes to another fog node in service topologies of varying sizes. Legend numbers represent thousands of edge nodes.

(e.g. Kubernetes scheduler, 100krnd, 150krnd). The ideal topologies do not consider resource use on fog nodes, therefore they can never really be achieved. However, since calculating actual ideal solutions is NP-hard, this is the only comparison which can be practically achieved. The random topologies are achieved by selecting the same number of fog nodes required by each iteration of Swirly, but at random.

For low numbers of fog nodes, and thus a high $|E|/|F|$ ratio, the output of Swirly is very close to randomized topologies, but still below it. This is because the algorithm has very little choice of fog nodes; most of them have to be used due to resource constraints and only about 10% are free to optimize the topology. This is further elaborated by Fig. 3.30. As the number of available fog nodes increases, Swirly starts to generate topologies that come closer to the ideal cases, but it eventually levels out at about twice the average distance of the theoretical ideal topologies.

Note that the 150k series topologies, despite a 50% increase in edge nodes over the 100k series, eventually have average distances which are only about 1-2% higher, which is negligible considering the range of the whiskers. The average distances of Swirly topologies are twice as high as those of the ideal cases, apart from the case of 150 fog nodes, likely indicating that they cannot be reduced much further without removing fog node resource requirements.

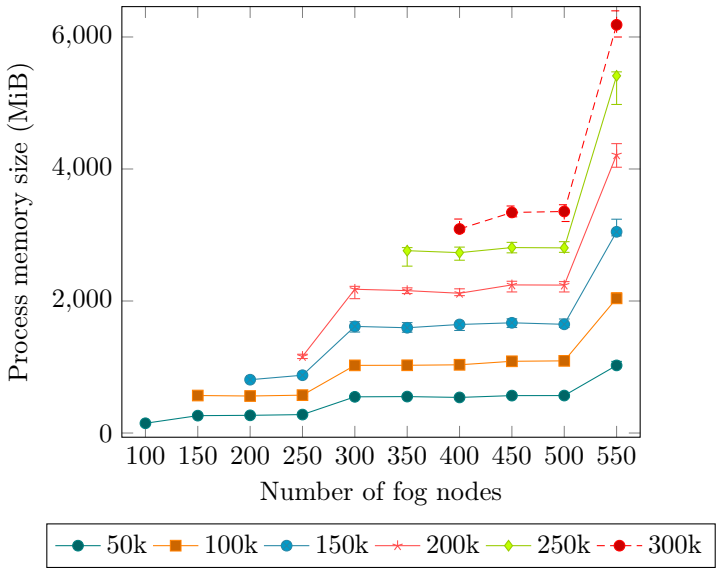


Figure 3.28: Memory required for service topologies of varying sizes. Legend numbers represent thousands of edge nodes.

A final observation concerns the maximum distance, which was set at 100. Swirly clearly struggles to stay below it when the $|E|/|F|$ ratio is high, and even fails at times. However, as the number of edge nodes increases, even the worst topologies generated by the algorithm have average distances well below the maximum distance. On average, the random topologies have average distances slightly above the maximum distance, and even the best random topologies barely outperform the worst Swirly topologies.

The observed results can be explained through the model of servicing areas. In cases where there are few fog nodes, a_F will have sections beyond r_P and possibly r_E for most fog nodes, which means they have to service a lot of edge nodes that are technically too far away, as in Fig. 3.24b. As the number of fog nodes increases, a_F will grow smaller since the physical area remains constant, reducing average distance by removing the extremes. Fig. 3.24c is an example of this.

Finally, Fig. 3.30 shows the number of fog nodes used by Swirly topologies compared to the absolute minimum of fog nodes that could be used. It is important to note that these numbers are averaged over all topologies with the same number of edge nodes, but with varying numbers of fog nodes. This indicates that even if Swirly has more choice of fog nodes, it primarily tries to fill activated fog nodes to capacity first, and uses very few to optimize edge to fog distances when needed. On average, Swirly uses about 30% to

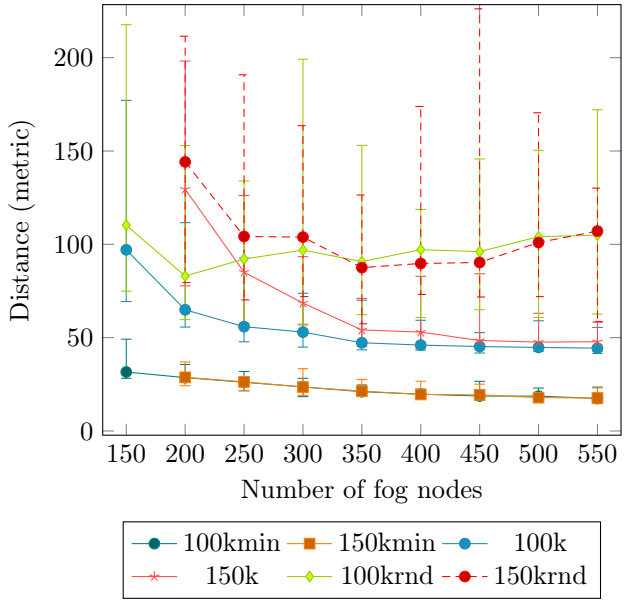


Figure 3.29: Average distance for different types of service topologies. Legend numbers represent thousands of edge nodes, min denotes theoretical ideal service topologies, while rnd indicates service topologies where the fog nodes are randomly chosen.

10% more nodes than strictly required, leaving some fog nodes with free capacity in case the topology expands. This confirms that Swirly fulfills the fourth requirement for a useful service scheduler.

Again, this can be explained in terms of the service area model. Swirly will always attempt to generate a topology in which fog nodes have responsibility areas as shown in Fig. 3.24c. Once a_F fits inside r_p and r_E is unchanged, Swirly will not activate any more fog nodes, no matter how many are available. However, if the algorithm has more fog nodes available from the start, it will construct slightly better service topologies by activating the ones closest to larger clusters of edge nodes. The rise in fog nodes seen in Fig. 3.30 is the result of r_E reducing as the number of edge nodes increases, and Swirly reacting by activating fog nodes to shrink a_F.

Fig. 3.31 shows how Swirly performs with physical topologies with different organizations. The tendency of the add operation to be slower when $|E|/|F|$ is higher is exaggerated when edge nodes are clustered together in remote groups, whereas the effect is all but gone in a perfectly equal distribution. With few available fog nodes, it takes 2.5 times as long to set up a service topology when edge nodes are clustered as it does with an equal distribution

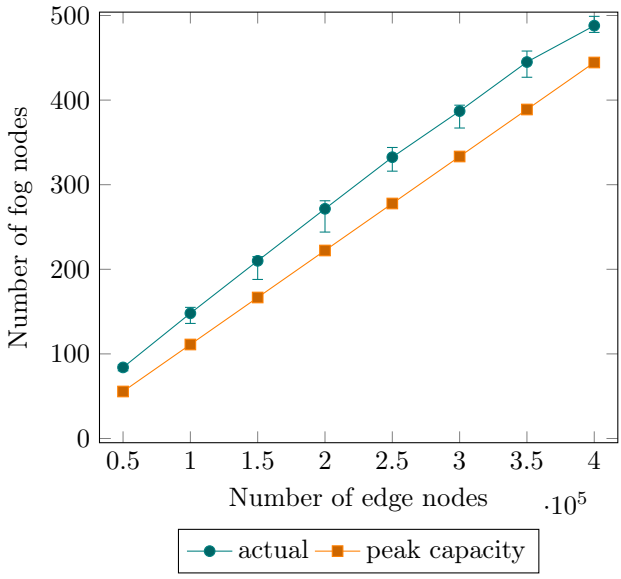


Figure 3.30: Number of fog nodes activated by varying number of edge nodes.

of edge nodes. This can be mostly attributed to the fact that the fog nodes distribution was not modified and thus Swirly has to search longer to find suitable fog nodes once the nearby ones are full. However, when both types of physical topology are given an unnecessarily high number of fog nodes, Swirly still needs about 40% more time to set up a service topology with clustered edge nodes. This confirms, as indicated by Eq. 3.16 and 3.17, that overall performance is significantly impacted not only by global node density, but also by local node density.

3.4.6 Discussion

In the beginning of Section 3.4, a number of requirements are presented for a useful large-scale fog service scheduler. It should work on a scale of hundreds of thousands of edge devices while being able to handle changing network conditions on topologies. Simultaneously, it should take resource limits of fog nodes and distance metrics between fog nodes and edge nodes into account. Finally, it must also minimize the number of fog nodes required for any fog service deployment required by a set of edge nodes.
Swirly is proposed as a solution to these requirements, and sections 3.4.2 and 3.4.3 show that it fulfills these requirements in theory. Different metrics are discussed, along with their advantages and disadvantages in terms of network overhead and reliability. The theoretical performance of Swirly is

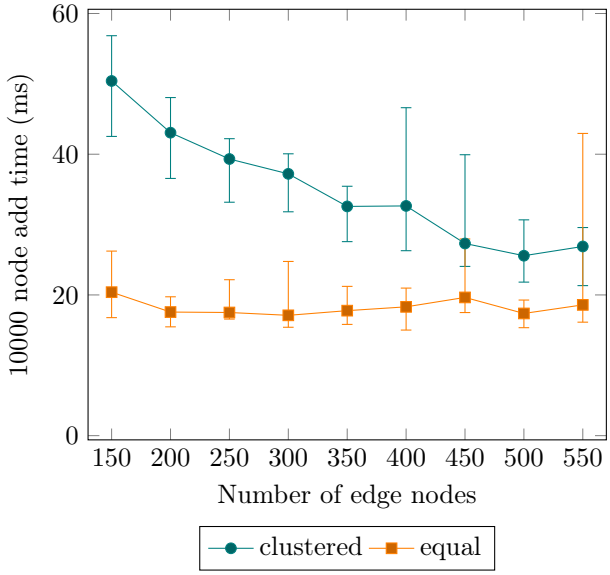


Figure 3.31: Effect of equal distribution of edge nodes versus clustered edge nodes in a topology with 100.000 edge nodes. In the clustered series, edge nodes are distributed over 4 equally sized circles at the corners of the topology.

explored, and the fog servicing area model is constructed to explain the behaviour and capacity of fog nodes in a service topology.

To verify its performance, Swirly is evaluated in terms of memory use and processing power. While the results mostly confirm the theoretical performance, they showed that the algorithm tends to slow down sublinearly as the density of edge nodes increases. This effect is explained through the service area model. An important prediction is that for service topologies that grow in physical size rather than density, Swirly will require constant processing time. If edge node density does increase, fog node density and algorithm parameters need to change as well. As discussed, the upper limit for the amount of nodes will most likely be dictated by memory consumption.

Further evaluations show that service topologies generated by Swirly converge towards a minimal average distance between edge nodes and fog nodes, which is well below the defined maximum value the algorithm needs to consider. Furthermore, the results show that average distances under Swirly are 30% to 55% lower than for randomly selected fog nodes (e.g. the default Kubernetes scheduler). While solutions based on heuristics (e.g. genetic algorithms) are likely to generate better solutions, they will also require more

time to do so, and they do not allow for real-time updates.
The presented Swirly algorithm could easily be adapted to work with the Kubernetes scheduler by managing fog service deployments through the Kubernetes API. In order to redirect service requests to the correct fog nodes, it could interact with distributed DNS plugins deployed on the cluster, override them, or deploy a separate system.
When implementing Swirly for a specific orchestrator, it may be advantageous to split the data structures so that topologies for several services can be generated from the same nodes and distance data. With minimal changes, it is possible to keep track of which services should be deployed to any fog node, based on edge node requirements.
Swirly does not yet fully support dynamic fog node updates. When fog nodes send updates to Swirly with their free resources, the algorithm only uses this to determine if edge nodes can be placed on those fog nodes in the future. Ideally, the algorithm would act on the resource updates by detecting critically low levels of free resources on certain fog nodes and reassigning edge nodes.
The results section shows that while Swirly scales very well in terms of processing time, its memory requirements will quickly grow beyond the reach of all but the most powerful servers. Since the algorithm relies on having a distance from each edge node to each fog node, there is no easy solution to this. However, geofencing or some type of partitioning may be able to help. In section 3.4.2 some options were discussed to reduce the required bandwidth of periodic node updates to Swirly. If this mechanism is changed to cut off fog nodes altogether based on metric distance or geographical distance, memory requirements should go down drastically. Another option is to simply split the topology into parts based on logical or geographical regions, but this may result in a significantly worse result at the borders between partitions.
For these reasons, it may be better to switch to a fully distributed approach, in which the cloud algorithm is eliminated and each edge node becomes responsible for finding its own optimal service provider.

3.5 Summary

This chapter introduces the concepts of fog networks and edge networks, explaining the difficulties and parameters for effective software deployment in such networks. High-level service orchestrators are introduced, most notably Kubernetes, which has become the de facto standard for functionality and compatibility.
A novel orchestration agent, FLEDGE, is constructed specifically for low-

resource devices in the network edge. FLEDGE is designed to be fully compatible with Kubernetes clusters, but components with low resource requirements are used (e.g. containerd) or created from scratch if not available (e.g. container networking), while remaining compatible with OCI standards. This orchestrator is shown to use fewer resources than Kubernetes and K3S. In later versions, capabilities were added to detect custom resources and hardware (e.g. Nvidia GPUs) and run containers with AI models, while adhering to the Kubernetes API.
Swirly, a centralized, lightweight service orchestration algorithm is presented. This algorithm is designed specifically to orchestrate services in extremely large edge networks for optimal end-user experience, while taking node resources, service requirements and a programmable distance metric into account. A mathematical model is constructed to predict the performance of Swirly in terms of node positions, densities and free resources. For the evaluation of Swirly, the distance metric is implemented as the latency between fog nodes and edge nodes, and the results of various scenarios are shown to adhere to the predicted performance, with Swirly managing the service infrastructure for up to 300.000 nodes in real-time.

References

[1] *What is Kubernetes?*, April 2019. Available from: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[2] R. Muralidhar, R. Borovica-Gajic, and R. Buyya. *Energy Efficient Computing Systems: Architectures, Abstractions and Modeling to Techniques and Standards.* ACM Computing Surveys, feb 2022. Available from: https://doi.org/10.1145%2F3511094, doi:10.1145/3511094.

[3] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. *All one needs to know about fog computing and related edge computing paradigms: A complete survey.* Journal of Systems Architecture, 98:289–330, sep 2019. doi:10.1016/j.sysarc.2019.02.009.

[4] S. Sinha. *State of IoT 2021*, September 2021. Available from: https://iot-analytics.com/number-connected-iot-devices/.

[5] *Kubernetes - Production-Grade Container Orchestration*, May 2021. Available from: https://kubernetes.io/.

[6] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. *Large-scale cluster management at Google with Borg.* Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015, 04 2015. doi:10.1145/2741948.2741964.

[7] *Kubernetes - Building large clusters*, May 2019. Available from: https://kubernetes.io/docs/setup/cluster-large/.

[8] *Cluster networking*, April 2019. Available from: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[9] *containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability*, October 2021. Available from: https://containerd.io/.

[10] *Why Docker?* Available from: https://www.docker.com/why-docker.

[11] *k3s - 5 less than k8s*, May 2019. Available from: https://github.com/rancher/k3s.

[12] *K0s - The Kubernetes for Edge/IoT*, October 2021. Available from: https://k0sproject.io/.

[13] *Eclipse ioFog - Architecture*, October 2021. Available from: https://iofog.org/docs/2/getting-started/architecture.html.

[14] *KubeEdge: A Kubernetes Native Edge Computing Framework*, 2019. Available from: https://kubeedge.io/en/.

[15] *What is KubeEdge: Architecture*, 2019. Available from: https://docs.kubeedge.io/en/latest/modules/kubeedge.html#architecture.

[16] R. A. Light. *Mosquitto: server and client implementation of the MQTT protocol.* The Journal of Open Source Software, 2(13):265, may 2017. doi:10.21105/joss.00265.

[17] Microsoft. *What is Azure IoT Edge*, September 2021. Available from: https://docs.microsoft.com/en-us/azure/iot-edge/about-iot-edge?view=iotedge-2020-11.

[18] T. Goethals, F. DeTurck, and B. Volckaert. *Extending Kubernetes Clusters to Low-resource Edge Devices using Virtual Kubelets.* IEEE Transactions on Cloud Computing, pages 1–1, 2020. doi:10.1109/tcc.2020.3033807.

[19] H. Sahni. *CNM vs CNI*, May 2017. Available from: https://www.nuagenetworks.net/blog/container-networking-standards/.

[20] *About OCI.* Available from: https://www.opencontainers.org/about.

[21] *Virtual kubelet*, April 2019. Available from: https://github.com/virtual-kubelet/virtual-kubelet.

[22] P. Mach and Z. Becvar. *Mobile Edge Computing: A Survey on Architecture and Computation Offloading.* IEEE Communications Surveys & Tutorials, 19(3):1628–1656, 2017. doi:10.1109/comst.2017.2682318.

[23] K. Kumar and Y.-H. Lu. *Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?* Computer, 43(4):51–56, apr 2010. doi:10.1109/mc.2010.98.

[24] J. Barrameda and N. Samaan. *A Novel Statistical Cost Model and an Algorithm for Efficient Application Offloading to Clouds.* IEEE Transactions on Cloud Computing, 6(3):598–611, jul 2018. doi:10.1109/tcc.2015.2513404.

[25] S. Sarkar, S. Chatterjee, and S. Misra. *Assessment of the Suitability of Fog Computing in the Context of Internet of Things.* IEEE Transactions on Cloud Computing, 6(1):46–59, jan 2018. doi:10.1109/tcc.2015.2485206.

[26] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. *Osmotic Computing: A New Paradigm for Edge/Cloud Integration.* IEEE Cloud Computing, 3(6):76–83, nov 2016. doi:10.1109/mcc.2016.124.

[27] D. Santoro, D. Zozin, D. Pizzolli, F. D. Pellegrini, and S. Cretti. *Foggy: A Platform for Workload Orchestration in a Fog Computing Environment.* In 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, dec 2017. doi:10.1109/cloudcom.2017.62.

[28] A. Morshed, P. P. Jayaraman, T. Sellis, D. Georgakopoulos, M. Villari, and R. Ranjan. *Deep Osmosis: Holistic Distributed Deep Learning in Osmotic Computing.* IEEE Cloud Computing, 4(6):22–32, nov 2017. doi:10.1109/mcc.2018.1081070.

[29] Kubernetes. *Kubernetes federation*, October 2018. Available from: https://kubernetes.io/docs/concepts/cluster-administration/federation/.

[30] T. Wauters, B. Vermeulen, W. Vandenberghe, P. Demeester, S. Taylor, L. Baron, M. Smirnov, Y. Al-Hazmi, A. Willner, M. Sawyer, D. Margery, T. Rakotoarivelo, F. Lobillo Vilela, D. Stavropoulos, C. Papagianni, F. Francois, C. Bermudo, A. Gavras, D. Davies, J. Lanza, and S.-Y. Park. *Federation of internet experimentation facilities: architecture and implementation.* In European Conference on Networks and Communications, Proceedings, pages 1–5, 2014.

[31] R. Moreno-Vozmediano, E. Huedo, I. M. Llorente, R. S. Montero, P. Massonet, M. Villari, G. Merlino, A. Celesti, A. Levin, L. Schour, C. Vázquez, J. Melis, S. Spahr, and D. Whigham. *BEACON: A Cloud Network Federation Framework.* In A. Celesti and P. Leitner, editors, Advances in Service-Oriented and Cloud Computing, pages 325–337, Cham, 2016. Springer International Publishing.

[32] P. Bottoni, E. Gabrielli, G. Gualandi, L. V. Mancini, and F. Stolfi. *FedUp! Cloud Federation as a Service.* In Service-Oriented and Cloud Computing, pages 168–182. Springer International Publishing, 2016. doi:10.1007/978-3-319-44482-6_11.

[33] D. Puthal, S. Nepal, R. Ranjan, and J. Chen. *Threats to Networking Cloud and Edge Datacenters in the Internet of Things.* IEEE Cloud Computing, 3(3):64–71, may 2016. doi:10.1109/mcc.2016.63.

[34] M. Villari, M. Fazio, S. Dustdar, O. Rana, L. Chen, and R. Ranjan. *Software Defined Membrane: Policy-Driven Edge and Internet*

of Things Security. IEEE Cloud Computing, 4(4):92–99, jul 2017. doi:10.1109/mcc.2017.3791014.

[35] N. Chowdhury and R. Boutaba. *Network Virtualization: State of the Art and Research Challenges.* IEEE Communications Magazine, 47(7):20–26, jul 2009. doi:10.1109/mcom.2009.5183468.

[36] H. Hamed, E. Al-Shaer, and W. Marrero. *Modeling and Verification of IPSec and VPN Security Policies.* In 13TH IEEE International Conference on Network Protocols (ICNP'05). IEEE, 2005. doi:10.1109/icnp.2005.25.

[37] F. Pohl and H. D. Schotten. *Secure and Scalable Remote Access Tunnels for the IIoT: An Assessment of openVPN and IPsec Performance.* In Service-Oriented and Cloud Computing, pages 83–90. Springer International Publishing, 2017. doi:10.1007/978-3-319-67262-5_7.

[38] I. Kotuliak, P. Rybar, and P. Truchly. *Performance comparison of IPsec and TLS based VPN technologies.* In 2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA). IEEE, oct 2011. doi:10.1109/iceta.2011.6112567.

[39] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari. *Towards Osmotic Computing: Analyzing Overlay Network Solutions to Optimize the Deployment of Container-Based Microservices in Fog, Edge and IoT Environments.* In 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC). IEEE, may 2018. doi:10.1109/cfec.2018.8358729.

[40] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente. *Cross-Site Virtual Network in Cloud and Fog Computing.* IEEE Cloud Computing, 4(2):46–53, mar 2017. doi:10.1109/mcc.2017.28.

[41] L. Xu, J. Lee, S. H. Kim, Q. Zheng, S. Xu, T. Suh, W. W. Ro, and W. Shi. *Architectural Protection of Application Privacy against Software and Physical Attacks in Untrusted Cloud Environment.* IEEE Transactions on Cloud Computing, 6(2):478–491, apr 2018. doi:10.1109/tcc.2015.2511728.

[42] C. Pahl and B. Lee. *Containers and Clusters for Edge Cloud Architectures – A Technology Review.* In 2015 3rd International Conference on Future Internet of Things and Cloud. IEEE, aug 2015. doi:10.1109/ficloud.2015.35.

[43] C. Dupont, R. Giaffreda, and L. Capra. *Edge computing in IoT context: Horizontal and vertical Linux container migration.* In 2017 Global Internet of Things Summit (GIoTS). IEEE, jun 2017. doi:10.1109/giots.2017.8016218.

[44] *Rancher Labs - K3S Lightweight Kubernetes.* Available from: https://k3s.io/.

[45] *MicroK8s.* Available from: https://microk8s.io/.

[46] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. *Extend Cloud to Edge with KubeEdge.* In 2018 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, oct 2018. doi:10.1109/sec.2018.00048.

[47] S. Hoque, M. S. de Brito, A. Willner, O. Keil, and T. Magedanz. *Towards Container Orchestration in Fog Computing Infrastructures.* In 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). IEEE, jul 2017. doi:10.1109/compsac.2017.248.

[48] *Kubernetes - Custom Resources.* Available from: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[49] *Deploy IoT Edge Gateway on Kubernetes.* Available from: https://docs.microsoft.com/en-us/samples/azure-samples/iotedge-gateway-on-kubernetes/iot-edge-workloads-on-kubernetes/.

[50] *Google cAdvisor*, August 2018. Available from: https://github.com/google/cadvisor.

[51] *Heapster is now retired*, November 2018. Available from: https://github.com/kubernetes-retired/heapster.

[52] *Kubernetes Resource Metrics API*, May 2018. Available from: https://github.com/kubernetes/metrics.

[53] M. GroBmann and C. Klug. *Monitoring Container Services at the Network Edge.* In 2017 29th International Teletraffic Congress (ITC 29). IEEE, sep 2017. doi:10.23919/itc.2017.8064348.

[54] S. Tarkoma. *Overlay Networks.* Auerbach Publications, feb 2010. doi:10.1201/9781439813737.

[55] *The DF command.* Available from: https://www.linuxjournal.com/article/2747.

[56] *pmap - report memory map of a process.* Available from: https://linux.die.net/man/1/pmap.

[57] *Propertional Set Size (PSS).* Available from: http://lkml.iu.edu/hypermail/linux/kernel/0708.1/3930.html.

[58] *Docker components explained.* Available from: http://alexander.holbreich.org/docker-components-explained/.

[59] *kube-proxy.* Available from: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/.

[60] *Getting started with rkt.* Available from: https://coreos.com/rkt/docs/latest/getting-started-guide.html.

[61] *CRI-O, lightweight container runtime for Kubernetes.* Available from: https://cri-o.io/.

[62] *KubeFed - Kubernetes Cluster Federation.* Available from: https://github.com/kubernetes-sigs/kubefed.

[63] T. Goethals, F. D. Turck, and B. Volckaert. *Near real-time optimization of fog service placement for responsive edge computing.* Journal of Cloud Computing, 9(1), jun 2020. doi:10.1186/s13677-020-00180-z.

[64] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. *Service placement in shared wide-area platforms.* In Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05. ACM Press, 2005. doi:10.1145/1095810.1118581.

[65] Q. Zhang, Q. Zhu, M. F. Zhani, and R. Boutaba. *Dynamic Service Placement in Geographically Distributed Clouds.* In 2012 IEEE 32nd International Conference on Distributed Computing Systems. IEEE, jun 2012. doi:10.1109/icdcs.2012.74.

[66] M. Aazam and E.-N. Huh. *Dynamic resource provisioning through Fog micro datacenter.* In 2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops). IEEE, mar 2015. doi:10.1109/percomw.2015.7134002.

[67] M. Aazam and E.-N. Huh. *Fog Computing Micro Datacenter Based Dynamic Resource Estimation and Pricing Model for IoT.* In 2015 IEEE 29th International Conference on Advanced Information Networking and Applications. IEEE, mar 2015. doi:10.1109/aina.2015.254.

[68] J. Santos, T. Wauters, B. Volckaert, and F. D. Turck. *Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications.* In 2019 IEEE Conference on Network Softwarization (NetSoft). IEEE, jun 2019. doi:10.1109/netsoft.2019.8806671.

[69] C. Canali and R. Lancellotti. *A Fog Computing Service Placement for Smart Cities based on Genetic Algorithms.* In Proceedings of the 9th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, 2019. doi:10.5220/0007699400810089.

[70] M. L. Farzin Zaker and M. Shtern. *Look Ahead Distributed Planning For Application Management In Cloud.* In 15th International Conference on Network and Service Management, CNSM 2019, IFIP Open Digital Library, IEEE Xplore, ISBN: 978-3-903176-24-9, 2019.

[71] H. E. El Houssine Bourhim and M. Dieye. *Inter-container Communication Aware ContainerPlacement in Fog Computing.* In 15th International Conference on Network and Service Management, CNSM 2019, IFIP Open Digital Library, IEEE Xplore, ISBN: 978-3-903176-24-9, 2019.

[72] *Kubernetes Scheduler*, January 2020. Available from: https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/.

4

Scaling Towards Smart Cities

"This is a thousand monkeys working at a thousand typewriters. Soon, they'll have finished the greatest novel known to man." - Montgomery Burns, The Simpsons

4.1 Introduction

So far, the concept of edge networks has been explained, as well as how basic technologies lead to the orchestrators and platforms required to flexibly and transparently deploy software services in them. However, the edge networks envisioned for Smart Cities may contain millions of devices, and this chapter shows how network organization, software services, and orchestrators can be scaled to accommodate data flows and computing requirements in such immense networks. Note that scalability in this chapter does not mean simply processing more data, but rather providing (improved) functionality to ever more nodes or end-users.

Section 4.2 gives an introduction to general software scalability, and the types of scalability that are useful for software services in edge networks. A Kubernetes-based solution for the secure, on-demand federation of private networks is presented in Section 4.3, showing that while it is possible and the setup is relatively agile, it is not necessarily an ideal solution for edge devices due to resource requirements. Section 4.4 presents SoSwirly, which focuses on the necessity of low resource requirements while maintaining excellent

scalability and real-time reaction to changes in network topology.

4.2 General scalability

As the chapter quote illustrates, there are several "laws" in computer science that describe the exponential progression of various trends, from increasing hardware capacity to the propensity of software and users to immediately fill that capacity, and then some. This section illustrates several methods of handling increased user demand explicitly by scaling locally or by offloading to networked nodes, or implicitly by decentralizing and letting the software organize itself to some degree. In all cases, the solution amounts to expanding to more computational resources, but the difference is in which physical locations they are located, and to what degree scaling can be automated.

4.2.1 Resource efficiency and local scaling

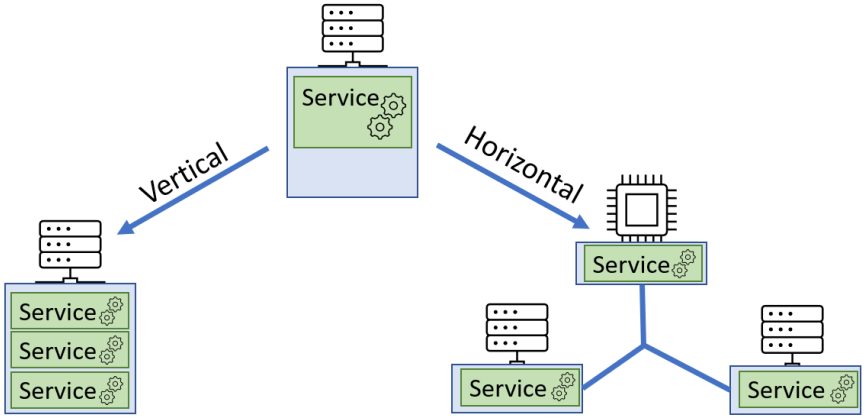


Figure 4.1: Vertical scaling versus horizontal scaling. By making a service (center) more resource efficient, more instances fit on a single server (left), while it can also be deployed on devices with fewer resources (right).

Improving the resource efficiency, or rather reducing the computational complexity of a software process usually allows it to handle greater workloads. This is mostly true for basic components, such as more efficient search algorithms or optimizing data streams and storage. In some environments, especially those with low-bandwidth IoT sensors, such improvements may be an absolute requirement for the system to function at all. In other environments, the same improvements may either lead to increased scalability, or be used to implement new features instead. However, there are practical

limits to how much optimization can be done, and how much work can be performed by a single process for any given task. In this section, local scaling means creating more instances of a software process in the same physical location in order to process more work. This can be achieved by vertical scaling, when all instances run on the same machine and aim to utilize all its resources, or by horizontal scaling, when processes are divided over several machines (e.g. in a data center). The latter case can potentially result in both better scaling and reliability, as there is no single point of failure. Fig. 4.1 illustrates the difference between these approaches. By default, a Kubernetes cluster will scale pods and services horizontally, deploying only one per eligible node. The scaling limits of large Kubernetes clusters [1] illustrate its intended strategy, aiming for hundreds of (different) pods per node (e.g. data center server).

4.2.2 Offloading

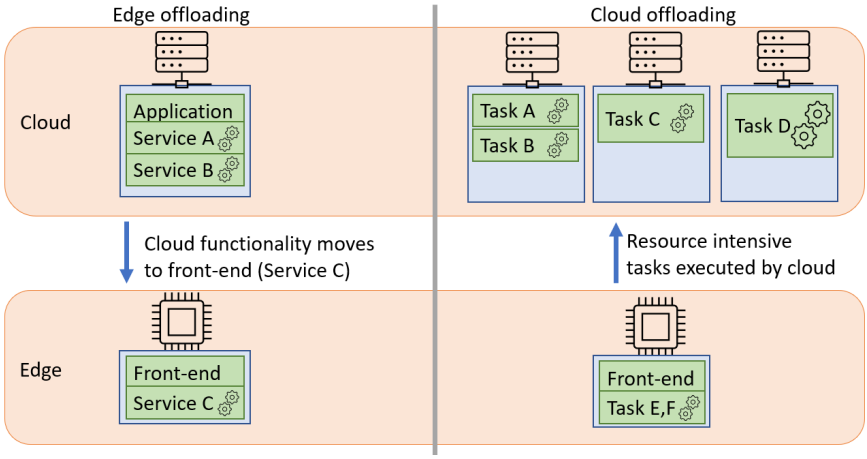


Figure 4.2: The effect of offloading cloud services to the edge (left) and offloading edge tasks to the cloud (right).

Offloading, a technique in which a device moves part of the computational load of a task to another node, can be used for improved scalability, as well as improving user experience, functionality and reliability. A basic example of improved scalability through offloading is distributed computing, e.g. Boinc, which sends workloads to a large number of managed nodes. The central component only needs to process and verify the results, which can be made an arbitrarily small task.

In the context of fog and edge networks, offloading can be done by edge

devices to the cloud or vice versa, depending on the requirements of an application. However, both methods effectively have the same result; improving reliability and user experience, while enabling new functionality on edge devices near end-users, on a scale that would be unfeasible in the alone cloud due to costs or hardware organization. Fig. 4.2 illustrates edge offloading and cloud offloading; the former technique moves lightweight cloud functionality to edge devices while keeping resource intensive functionality in the cloud, and the latter outsources computationally intensive tasks generated by edge devices to the cloud (or fog).
While Kubernetes-based solutions can be used for offloading strategies, the maximum cluster size of 5.000 nodes imposes a significant limitation on any architecture that creates a Kubernetes node for each fog or edge device. For example, FLEDGE from Chapter 3.3, which is meant to enable container-based offloading on low-resource devices, uses Virtual Kubelets to represent edge nodes in a Kubernetes cluster, subjecting it to this limit. Some solutions, such as KubeEdge, avert this artificial limit by using custom agents on edge nodes that do not register as an actual Kubernetes node. By not involving Kubernetes in the edge, such orchestrators can improve scalability, at the cost of losing some functionality the Kubernetes API provides by default.

4.2.3 Federation

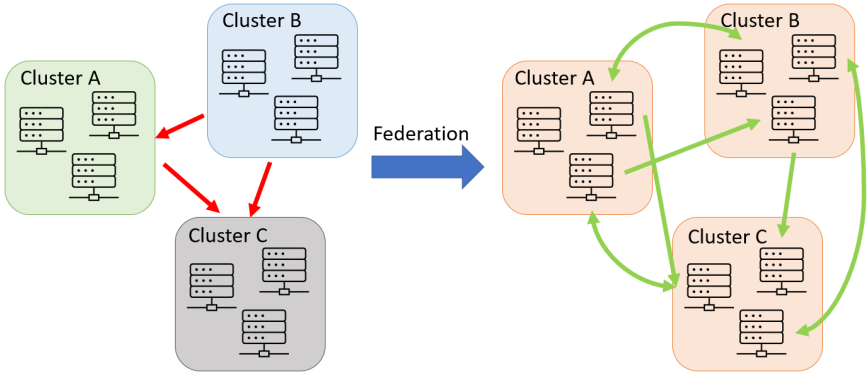


Figure 4.3: Federating the individual clusters on the left makes communication between nodes easier and allows for uniform, high-level resource management.

Federating networks or data sources into a larger whole makes their combined resources available for potential improvements. Such improvements may lead to novel applications and better user experience, but federations

may also be explicitly used to improve scalability. The federation of networks or computational clusters often involves creating a management layer on top of the existing infrastructure that can integrate member networks, translate addresses and (network) resources transparently, and secure communications between members. The effects on node networking and communication are illustrated in Fig. 4.3.
Kubernetes Federations [2] are one way to overcome the intrinsic scalability limits of Kubernetes by combining multiple clusters into a federation that can be managed by a higher-level control plane. The federation control plane is capable of deploying to several clusters simultaneously, and transparently converts definitions of (custom) resources between each cluster. Apart from scalability, this approach is also useful for combining clusters with separate origins under a single management layer, reducing maintenance overhead.
Flexible federated Unified Service Environment (FUSE), discussed in Section 4.3 of this chapter uses federations to enable secure, real-time, on-demand cooperation between private networks. FUSE exploits the scale and potential of federated private networks and their resources to enable new applications for use by crisis centers in emergency situations.

4.2.4 Decentralization

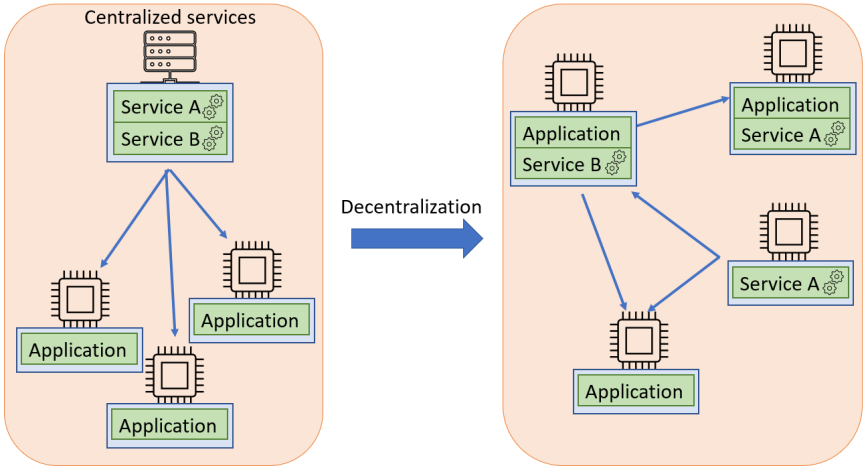


Figure 4.4: Decentralizing cloud services A and B required by an edge application, by deploying them on various edge nodes as required.

Decentralization improves scalability by removing any centralized component in a software architecture, as illustrated by Fig. 4.4. Without a centralized component or orchestrator, there is essentially no limit on the scale

of the software being deployed, but not all software is suitable for decentralization. There are two basic components required for a decentralized architecture:

- Each node needs a pre-installed *entry point* that allows for its discovery and activation, and optionally exposes the functionality of the software services on it.
- A generic *discovery algorithm* which each node uses to discover other devices in its neighbourhood, for various uses as required by the software services it runs.

An optional leader election algorithm can impose some hierarchical structure on a cluster of devices, in case some devices need to act as entry points for demanding services that can not run on every device due to resource constraints, but these will once again introduce a pseudo-centralized component, potentially limiting the scalability of a cluster to some degree.
Decentralization is very useful in edge networks, where consumer devices often have pre-installed components that connect to the cloud on activation. These devices are usually constrained in functionality (e.g. light bulbs or smart appliances) and used in small networks where their discovery is straightforward. Therefore, an approach that eliminates the centralized cloud components and simply sets up software services on a gateway for local discovery and processing is desirable (e.g. auto-discovery features in Home Assistant[1]). Note that communication between all separate nodes is not always required (e.g. "islands" can be formed); it is the potential of autonomous discovery and on-demand software deployment on nodes that separates this approach from simple stand-alone software.
Although FLEDGE is designed to be used with a Kubernetes cluster, it exposes pod management functionality through an orchestrator-agnostic REST API. This API is based on Kubernetes API objects (e.g. pods, resources), and can be used in decentralized architectures by bypassing the Virtual Kubelet, providing limited container orchestration through Kubernetes API objects.

4.3 Secure On-Demand Federation

This section contains the edited version of the following publication: **"FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers"**, T. Goethals, D. Kerkhove, L. Van Hoye, M. Sebrechts, F. De Turck, B. Volckaert

[1] https://www.home-assistant.io/getting-started

published in **Proceedings of the 9th International Conference on Cloud Computing and Services Science - CLOSER, 90-99, 2019** [3]

In crisis situations, a crisis center is usually formed which monitors the situation and takes suitable actions. It is important that crisis centers are able to quickly gather information from various sources, for example closed-circuit television (CCTV) or positional data, to get a complete and accurate overview of the situation. When the required information sources are owned by different companies, it is difficult for the crisis center to gain access to them or to interact with them uniformly. Fig. 4.5 illustrates the concept of joining companies and a crisis center into a single virtual network, from now on referred to as a federation. Company A joins the federation, making its IP cameras available for use by routing their streams, and company B allows the use of both its servers and sensory hardware. The crisis center gathers information from the exposed devices of both companies and transforms it into a dashboard for its operators. To do this, a *federation service environment* is required which supports a wide range of operating systems and devices. Since containers are widely supported and easy to deploy, a container-based service environment is preferable.

In addition to varying devices and software platforms, companies participating in a federation often have different network and security policies, making it difficult to connect them quickly or to deploy software in their networks (domains) in a uniform way. Any solution to connect multiple domains should ensure that every company can choose exactly which resources it makes available to the federation, and that all communication between federated devices is secure. Only devices that are part of the federation must be visible from other domains, while devices from each domain that are not part of the federation should only be reachable from federated devices in the same domain. For example, in Fig. 4.5 the data storage of company B can be used by federation components deployed in its own domain, but it is unavailable to the rest of the federation. The same is true for the non-federated device at company A, which is invisible to the devices of company B and the crisis center.

Apart from being cross-domain and ensuring secure communications, a federation service environment focused on crisis situations must also be fast and easy to set up. Joining a federation should only take minutes, with minimal intervention from company administrators. Additionally, a company should be able to join or leave a federation at any time without destabilizing the federation, and after leaving a federation no trace of the federation service environment should be left on a company's devices. Furthermore, the

components of a federation service environment should not interfere with other processes on a device, which means that the entire federation service environment and all its components should be isolated as much as possible. The challenges for building a federation service environment using containers can thus be summarized as follows:

1. Enabling and securing fast cross-domain communication while restricting access to non-federated resources
2. Isolating the federation service environment from other software
3. Ensuring fast and easy deployment of the federation service environment on a large range of devices

This section presents Flexible federated Unified Service Environment (FUSE) to tackle these challenges. FUSE provides a microservice-oriented, container-based service environment to deploy and manage software on federated domains. It is designed to quickly set up ad hoc federations with minimal intervention, ensures secure communication between domains and prevents non-federated devices from being visible from other domains.
Section 4.3.1 presents related work to the challenges presented in this introduction, and Section 4.3.2 describes how they are solved in FUSE. Section 4.3.3 describes the test setup for a basic FUSE federation, while section 4.3.4 details the system requirements of FUSE and presents performance results for a typical use case. Section 4.3.5 discusses the results and applications of FUSE, and suggests some topics for future work.

4.3.1 Related work

Previous federation service environment projects have resulted in frameworks such as Fed4Fire [4], Beacon [5] and FedUp! [6]. Fed4Fire has a different use case from FUSE and requires the implementation of an API to integrate devices into a federation, which makes it inadequate for the rapid ad hoc use cases of FUSE. BEACON is focused on cloud federation and security as a function of cloud federation, but the use case of FUSE requires it to work in company networks and around existing and unchangeable security policies. FedUp! is a cloud federation framework focused on improving the setup time for heterogenous cloud federations. Unlike previously mentioned frameworks, FUSE operates on company networks rather than cloud infrastructure and aims to cut down set-up time to minutes or less with minimal intervention from system administrators, independent of target devices and operating systems. Many tools have been created for the different aspects of federation, for example jFed [7] (general lifecycle management), OML[2]

[2]https://wiki.confine-project.eu/oml:start

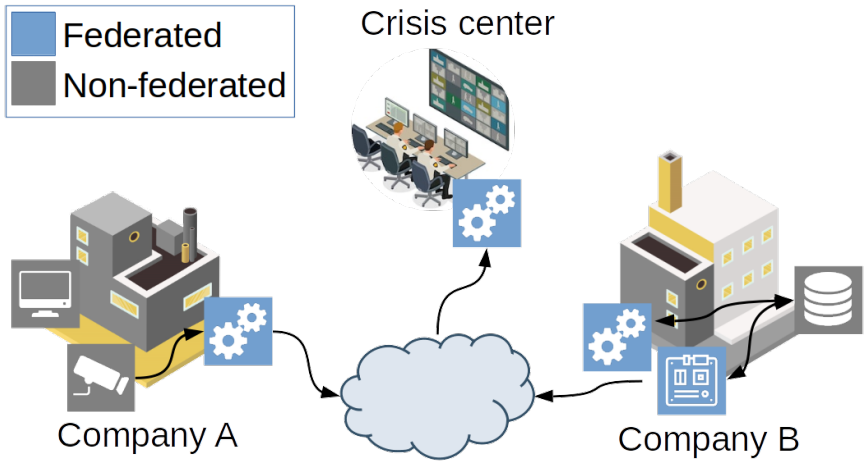


Figure 4.5: Example federation.

(measurement and monitoring) and OpenID [8] (trust, user authentication). Each of these tools only solves part of the problem FUSE faces and often requires the implementation of specific APIs to work with. For example jFed, which was developed for the Fed4Fire project, provides high-level control over federated resources, but it only works with known hardware types and pools of pre-configured resources.

Container engines such as Docker[3] and rkt[4] have been studied and evaluated extensively in literature [9], for example in performance reviews [10] and in overviews of virtualization technology [11]. Similarly, the capabilities of container orchestration tools such as Kubernetes[5] and its precursor Borg are explored in various studies [12, 13]. Microservice architectures using containers have seen a lot of attention, specifically for their use in rapid and easy deployment of (cloud) applications [14, 15]. However, Kubernetes by itself is insufficient to build a federation service environment when the devices in the federation are hidden from public view by firewalls or by other means, which is often the case in company networks.

Docker security has been thoroughly studied [16], and there are security best practices for Kubernetes [17]. However, no work is found on securing network traffic specifically, which is required when sending valuable data between domains over the internet. Kubernetes is capable of forming federations of multiple Kubernetes clusters [2], but to the best of our knowledge,

[3]https://www.docker.com/
[4]https://coreos.com/rkt/
[5]https://kubernetes.io/

no work has been done on a single Kubernetes cluster spanning multiple physical domains.
Certain studies investigate the usefulness of edge computing and edge offloading [18, 19], two concepts whereby computing workload is moved from cloud hardware to edge devices (or vice versa) based on hardware load and service demand. Of particular interest are studies where virtualization is employed for edge offloading purposes [20]. This work is closely related to how and why federations could include edge devices to serve as information sources or processing hardware.
Various aspects of cloud resource management have been studied [21], for example the scalability of certain topologies. Studied topologies include a centralized controller [22], management hierarchies [23, 24] and fully distributed approaches [25].

4.3.2 Components and Architecture

Within a single domain, a federation can easily be formed using Docker containers and Kubernetes. The use of Docker containers ensures that software can be deployed to a wide variety of target devices, as long as they support Docker. Kubernetes is used to join and manage all the devices in the federation, and to deploy software on those devices. The only downside to this approach is that software needs to be containerized in order to deploy it.
Kubernetes identifies the roles of specific devices in a cluster by making them either a master or a worker node. FUSE, being built around Kubernetes, adopts these two roles while giving them additional responsibilities. Master nodes are the equivalent of Kubernetes masters and consist of a Kubernetes control plane and other services required for FUSE. Worker nodes perform the function of Kubernetes workers and generally only contain a kube-proxy and deployed containers.
The rest of this section details how FUSE solves the problems posed in the introduction and describes how they fit into the basic Docker and Kubernetes setup described here.

4.3.2.1 Cross-domain Federation and Security

Kubernetes deploys containers in groups called pods, which have their own virtual network for communication between all devices that constitute a cluster. This inter-pod communication is done with the aid of a Container Network Interface (CNI) driver[6], such as Flannel[7], which assigns an IP ad-

[6] https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/
[7] https://github.com/coreos/flannel

dress from a configurable range to every pod running on the Kubernetes nodes under its control. However, Flannel runs into problems when a Kubernetes node is unreachable, for example when it is hidden behind a firewall or when no route to the target machine exists. Kubernetes, being primarily built for cloud environments, has no facilities to work around this problem. In order to enable Flannel traffic between domains, and to secure that traffic, OpenVPN tunnels[8] are created between FUSE worker nodes and the master node. Every FUSE master node runs an OpenVPN server, while all FUSE nodes (including the master) have an OpenVPN client that connects to the VPN server on the master node. Once a connection is established, a FUSE node gets an IP address from a configurable range of VPN addresses. This address is added to routing tables in FUSE services, together with the Flannel address range of a node. After this initial setup, all of the ports on a node's OpenVPN interface are forwarded to Flannel. To optimize performance, Flannel is run using the host-gw[9] back-end instead of vxlan, which puts pod network packets directly on the OpenVPN interface instead of encapsulating them. The downside of using only one OpenVPN server, running on the master node, is that all traffic is routed via the master node, even if it is just between worker nodes.

Apart from enabling cross-domain federation and securing communications, FUSE also needs to ensure that non-federated devices can not be reached from a domain other than the one they are in. This is achieved by generating specific routing rules for each node in a federation. The concept is shown in Fig. 4.6, where a red, green or blue box represents a company network (domain), yellow boxes represent the distributed parts of the Kubernetes pod network and the gray translucent box represents the VPN network which connects all FUSE nodes. Green arrows indicate which devices can interact with each other, while red ones show which ones can not. In company A, worker B.1 can receive camera streams from non-federated devices in company A through its company-assigned IP address, and it is also able to forward the stream to the crisis center over its FUSE VPN-assigned IP address. However, non-federated devices in company A can not be reached by any devices from either the crisis center or company B. This approach solves the first challenge posed in the introduction.

4.3.2.2 Encapsulation

To tackle the second challenge discussed in the introduction, a solution is needed that isolates FUSE components and minimizes the required software

[8] https://openvpn.net/

[9] https://github.com/coreos/flannel/blob/master/Documentation/backends.md

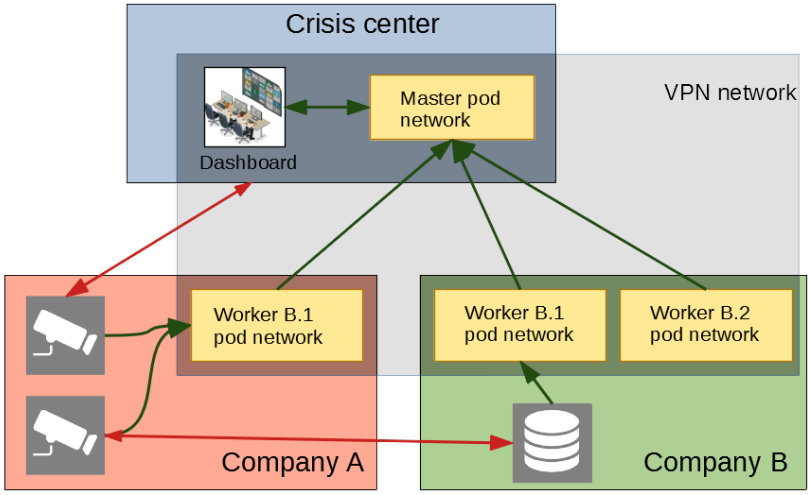


Figure 4.6: High-level network overview of an example FUSE federation.

to deploy FUSE. All services deployed on a FUSE worker node are containerized, but the components of FUSE should be isolated as well. Docker-in-Docker (DinD[10]) enables the nesting of Docker environments by deploying a containerized Docker environment within another Docker environment. Using a DinD approach, FUSE components and services are deployed in the outer docker environment, while the inner environment runs containers deployed on the node by Kubernetes. Thus, FUSE processes remain isolated from other processes and a Docker installation is the only requirement to start a FUSE node. Additionally, FUSE can ensure that none of its components remain on a device after it leaves the federation. However, both FUSE components and client software must be containerized in order for this approach to work. For OpenVPN, this means using an OpenVPN server container on the master node and OpenVPN client sidecars [26] on the worker nodes.

Using DinD creates an additional network layer between the host OS and the Kubernetes node. As previously mentioned, Flannel takes care of network traffic for the pods in the inner Docker environment, but the outer Docker environment still needs an addressing scheme. For this layer, a static address was chosen for each FUSE service, no matter which node it runs on, so that a FUSE node always knows where to reach a certain service running on it. For example, a VPN server container IP address always ends in .2, while a

[10]https://github.com/kubernetes-sigs/kubeadm-dind-cluster

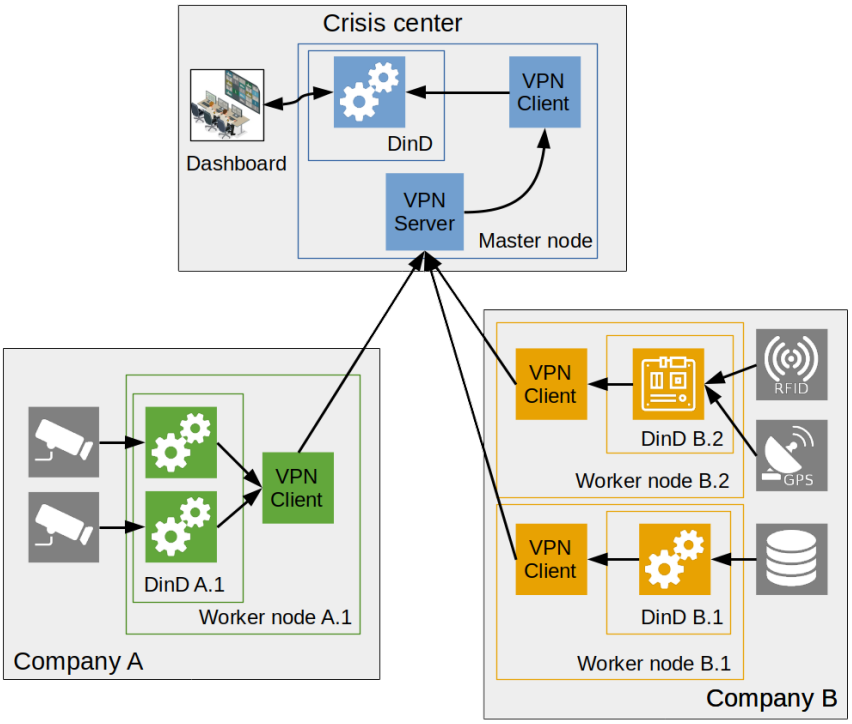


Figure 4.7: Nodes and information flow of an example FUSE federation.

VPN client container address always ends in .3.

Fig. 4.7 shows the federation illustrated in Fig. 4.5, with the concepts discussed in this section. Fig. 4.7 shows that every node, even the master node, has a VPN client which is connected to the master node's VPN server. Company A has a single worker node, which is running multiple containers to forward video streams to the crisis center through a single VPN client. Company B, on the other hand, has two worker nodes, each with their own VPN clients. Node B.1 has a single container which gives the crisis center access to data from company B, and node B.2 gathers positional data from various sources and makes it available to the crisis center.

4.3.2.3 Fast and Easy Deployment and Teardown

The DinD solution from the previous subsection enables easy deployment and teardown of FUSE. Since all FUSE components are running in containers, only a single startup script is needed to deploy FUSE on a node or to remove it from a node, with any required containers being pulled from

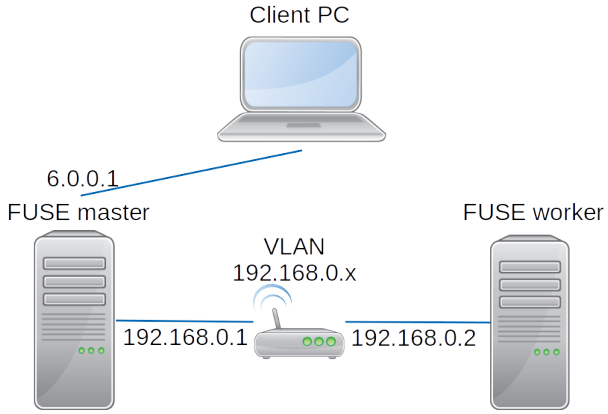


Figure 4.8: Evaluation setup overview.

a remote or local Docker registry. Thus, DinD also satisfies the third requirement from the introduction, but tests are needed to confirm that this solution can run on a wide range of devices.

4.3.3 Evaluation Setup

Considering the use cases of FUSE, it is important that it can be deployed on a large range of devices and that both master and worker nodes can quickly join the federation when needed. These requirements are also part of the challenges stated in the introduction, of which the first one demands that FUSE has good network performance, and the third one demands that FUSE is fast to set up and tear down on a wide range of devices.

To confirm that FUSE meets these requirements, measurements were performed to determine memory consumption, required hard disk space, deploy times for nodes and network performance of the federation. Being able to run FUSE on a Raspberry Pi 3 is set as a concrete goal for low-end devices. All tests were performed on the imec/IDLab Virtual Wall environment[11] using bare metal servers. Special care was taken to ensure that the hardware of all servers was identical for every test run. The hardware configuration used for every device consists of two Intel Xeon E5620 processors clocked at 2.4 GHz, 12GiB DDR3 memory and a 16GiB partition on a 160GiB WDC WD1600AAJS-0 hard drive.

[11] http://doc.ilabt.iminds.be/ilabt-documentation/virtualwallfacility.html

Fig. 4.8 shows the test setup. The fuse master and worker nodes are connected by a VLAN which allows setting specific amounts of packet loss and delay. Meanwhile, a client PC is connected to the FUSE VPN and interacts with Kubernetes services via the VPN IP address assigned to the master node. By default, the 6.0.0.x address range is used, but it is configurable. While the client PC is not a part of the virtual wall environment, it is directly connected over LAN, with a ping time of only 0.24 +- 0.01ms.
The required hard disk space for a node was measured by summing the disk usage of all the required containers and FUSE scripts. Memory consumption was determined by comparing available memory as reported by the free command[12], before and after deploying FUSE. Network throughput was determined with iperf3[13] in the DinD container on both FUSE nodes and running it in both TCP and UDP mode. The resulting physical traffic is thus TCP-on-TCP and UDP-on-TCP, respectively.
To determine how quickly FUSE can set up a federation and how quickly worker nodes can join or leave the federation, the time command was used to measure the duration of the relevant FUSE commands. The final results were calculated from ten successful runs of each test.
As a test of FUSE performance in a typical use case, a container is deployed to the worker node which simulates a camera stream by looping a 720p video file, recorded from a security camera, to a websocket using ffmpeg[14]. The video has mostly static images with little movement and is encoded at 2 Mbits/s. The frame rate of the recording is 25 frames per second (FPS), but the output frame rate is not limited. Because the frame rate is not limited and the video stream consists of mostly static scenery, this is a good test of FUSE network performance. However, in real crisis situations, there would likely be a lot of activity on camera streams, possibly influencing the frame rate from one moment to the next. Since this would be harder to quantify, the FPS test is only meant as an indication of FUSE bandwidth. A dashboard application container, consisting of a web page with a Node.js[15] back-end, is deployed on the master node. The dashboard back-end receives the worker node's stream via a websocket and sets up a proxy websocket, which in turn sends the data to the web page opened by the client PC. The proxy websocket acts as an aggregator for multiple streams, but for the tests below only one stream was used. The web page itself plays the video using jsmpeg[16]. The results, measured in fps, were calculated by hooking into jsmpeg's render loop and calculating the average and standard deviation

[12] http://www.linfo.org/free.html
[13] https://iperf.fr/iperf-download.php
[14] https://www.ffmpeg.org/
[15] https://nodejs.org/en/
[16] https://github.com/phoboslab/jsmpeg

Table 4.1: FUSE master node create and leave times.

	Create	Leave
Min time (s)	356	7.87
Median time (s)	374	7.90
Max time (s)	381	8.90

over 70 seconds of streamed video.

4.3.4 Results

4.3.4.1 Hardware Requirements

Worker nodes require 529 MiB of disk space and are small enough that they can be deployed on a Raspberry Pi 3 or similar devices. However, in order to have enough room to deploy software, several gigabytes of free space would be recommended. Master nodes require 1576 MiB of disk space, which is about three times as much as a worker node. This makes sense, since they need to deploy all FUSE components, a VPN server and a Kubernetes master. Considering their role as communications and management hubs for the federation, master nodes will usually be deployed on hardware with orders of magnitude more free space than the required amount, so this should not be a problem.
Concerning memory consumption, a worker node could easily be deployed on a Raspberry Pi 3, since it needs only about 228 MiB free memory. Master nodes need around 851 MiB free memory, which is almost four times more than a worker node. Again, this is due to having to run Kubernetes and all FUSE services. It would be very hard to deploy a master node on hardware with 1 GiB RAM, even with an extremely slimmed down host OS.

4.3.4.2 Federation Setup and Teardown

Since master nodes can be started up front and kept ready-to-go, their start times are less important than those of worker nodes. They have no interaction with any other devices while deploying, so no special cases need to be examined.
Table 4.1 shows the minimum, median and maximum observed times it takes to set up a FUSE master node or tear it down. A master node takes only about 6 to 6.5 minutes to set up from scratch, while it can be removed from a device in about 8 to 9 seconds.
For worker nodes, the quality of the network connection to the master node is important for federation setup and tear down. To examine the impact of the connection quality, a range of combinations of communication delay

and packet loss were simulated. Delay ranges from 0ms to 400ms in 100ms steps, while packet loss ranges from 0% to 20% in 5% steps.
Considering all the network layers in the FUSE architecture, it is hard to model performance using existing research. FUSE traffic consists of TCP or UDP packets from containers wrapped in TCP packets by OpenVPN. A worker attempting to join a federation performs a number of requests. The execution time of a single request over TCP is

$$t_{op} = (w_s + \frac{w_v(d_v + d_s)}{1 - l}) \tag{4.1}$$

where l is packet loss, d_v is network delay, d_s is delay resulting from handling network operations in software, w_s is work not influenced by network activity (for example, parsing JSON response data), and w_v represents work that depends on network performance.
However, this only works for a single request. A federation operation, for example joining a federation, requires several calls to web services. This introduces another factor caused by service call timeouts, which in turn can cause a retry of the entire operation:

$$t_{total} = \frac{t_{op}}{(1 - \max(\min(\frac{d_v - d_l}{d_c - d_l}, 1), 0))(1 - l)} \tag{4.2}$$

Where d_l is the delay threshold below which no operation should time out and d_c is the critical delay threshold above which every operation results in a timeout. For this equation, delay is in the denominator because its effect is no longer linear. The constants in Eq. 4.1 and Eq. 4.2 have to be determined empirically and are different for every hardware setup, but with the results in Fig. 4.9 the most important effects on the test setup can be identified.
Fig. 4.9 shows the time it takes to join a federation for several combinations of delay and packet loss. In case of smooth network performance, meaning less than 100ms delay or less than 10% packet loss, joining a federation only takes as much as 1 to 4 minutes. As the model predicts, packet loss has a strong hyperbolic effect on the time it takes to join a federation, but it is still doable even with high rates of packet loss and high delay. Delay has a mostly linear effect, as shown by Eq. 4.1. Only for combinations of high delay and a lot of packet loss does it turn to a slight hyperbolic effect, explained by its term in Eq. 4.2. Further modelling is unreliable because of the large error margins on these data points. Eventually, around 20% packet loss and 400ms delay joining becomes so slow and erratic that it is unlikely to still be practical. The error margin on this data point shows that join attempts may take anywhere from 6 to 14 minutes.

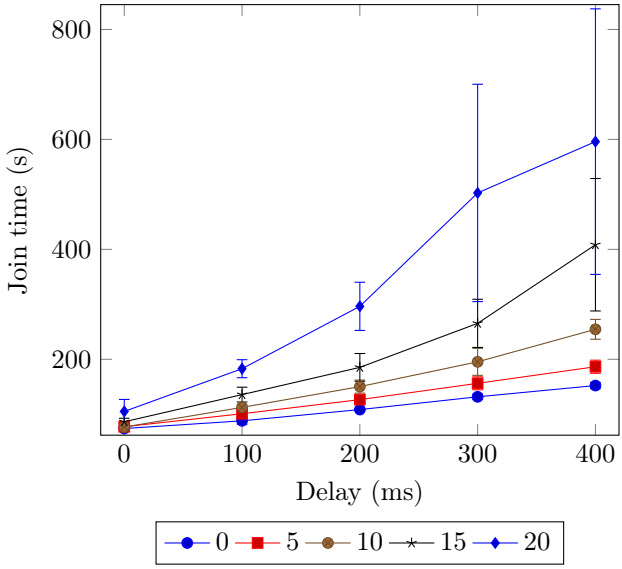


Figure 4.9: Federation join time for increasing delay at several percentages of packet loss.

4.3.4.3 FUSE Network Performance

Fig. 4.10 shows the communication speed between a FUSE master node and a worker node with packet loss and delay set to zero. While TCP performance is only 109 Mbits/s, the test setup has no support for hardware encryption using the AES-NI instruction set, which would give much better results [27]. UDP performance is very low with only 11.7 Mbits/s, meaning any FUSE traffic should be kept to TCP as much as possible. During the tests, OpenVPN used between 60% and 100% of a single CPU core. Since OpenVPN runs on a single thread, these results are about as good as they can get without optimizations. This means that communication alone takes a large part of the total processing power of a node.

For the video streaming test, the delay between the FUSE nodes was set from 0 to 100ms in steps of 25ms, while the effect of packet loss was examined for 0%, 0.2%, 1% and 2%. Because the software involved in this test does not use buffering, the results are a good reflection of network performance.

Since a TCP video stream is one-way traffic that requires no response at the application level, a different model is used to predict performance than for joining a federation. Only packets that do not make it on the first send incur a penalty on the rendering process. Uniform delay has no effect on the quality of video, merely delaying the rendering of each frame by the same

amount of time, resulting in Eq. 4.3 for the time to transmit and render a frame. In this case n is the number of packets that need to be sent for a frame, d is the network delay, and l is the packet loss. d_0 is an intrinsic delay that occurs from endpoint handling of network traffic and the speed limit of electronic communication.

$$t_f = d_0 n + \sum_{i=1}^{\infty} n(d + d_0) l^i = d_0 n - \frac{(d + d_0) l n}{l - 1} \tag{4.3}$$

For a d_0 that is sufficiently small compared to the delay caused by network problems and bad connections, Eq. 4.3 can be reduced to Eq. 4.4 to estimate FPS for video throughput. Since n can not be reasonably estimated for any frame, it is replaced by t_0, which represents the time it takes to transmit an average frame under ideal circumstances (no packet loss, only d_0 delay).

$$f = \frac{1}{t_f} \approx \frac{1}{t_0(1 - \frac{dl}{l-1})}, d_0 \ll d \tag{4.4}$$

Fig. 4.11 shows the results of the video throughput test, for which 24 FPS is set as the minimum acceptable framerate. Despite the static bit rate of the source video, the standard deviation at every data point is nearly always over 20% of the average at that point, showing that there is a large fluctuation in performance. During testing, it was verified that this is not a side effect of the rendering process of jsmpeg, but because jsmpeg actually receives variable amounts of data each second. This variation is directly related to FPS variation.

From the simplified model, performance for 0% packet loss would be expected to remain level instead of slowly declining, but this is the result of a tiny amount of packet loss intrinsic in all systems. Even for as little as 0.05% packet loss, Eq. 4.4 shows a significant decline, which seems locally linear rather than hyperbolic for the examined range of delay. It was verified with tcpdump[17] and Wireshark[18] that a minute amount of packet loss was indeed present.

Similarly, the data points for 0ms delay for the different levels of packet loss do not all have the same value because of the intrinsic delay d_0. Since in this case the conditions for Eq. 4.4 are violated, it is better to go with Eq. 4.3.

The rest of the chart follows the general shape predicted by the model. A roughly hyperbolic shape, with an increase in packet loss causing a faster degradation than an increase in delay. The general result is that performance quickly drops to a useless level, unless special care is taken to avoid

[17] http://www.tcpdump.org/
[18] https://www.wireshark.org/

noticeable packet loss. While 0.2% loss combined with 100ms delay still results in a useful stream at 29fps, as little as 1% loss combined with 50ms delay results in only 17fps. The only other good results were all obtained at 0ms delay (0.25ms counting d_0), but those are unrealistic in practical federations, even if the nodes are very close to each other geographically.

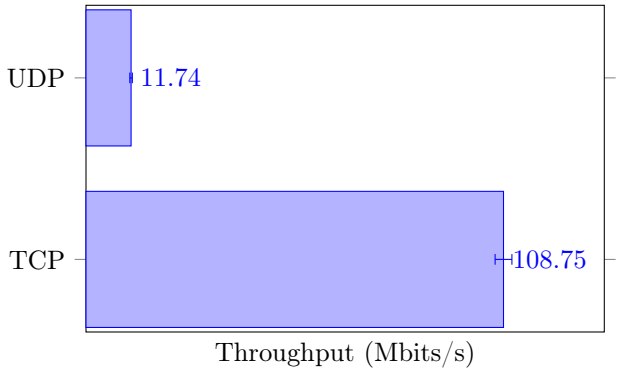


Figure 4.10: FUSE network throughput for UDP and TCP network traffic.

4.3.5 Discussion

A containerized approach makes FUSE easy to deploy on any device that supports Docker. Additionally, leveraging Kubernetes makes sure that containerized software can be deployed using familiar and reliable methods. However, FUSE can only be deployed on devices that support Docker, and existing software has to be containerized in order to be deployed in a FUSE federation.
The introduction puts forth three challenges in creating a federation service environment for crisis situations:

1. Enabling and securing fast cross-domain communication while restricting access to non-federated resources
2. Isolating the federation service environment from other software
3. Ensuring fast and easy deployment of the federation service environment on a large range of devices

It is shown that the FUSE architecture solves these challenges by using OpenVPN to enable and secure cross-domain traffic, and by using DinD to isolate FUSE from other software running on a device and simplifying deployment.

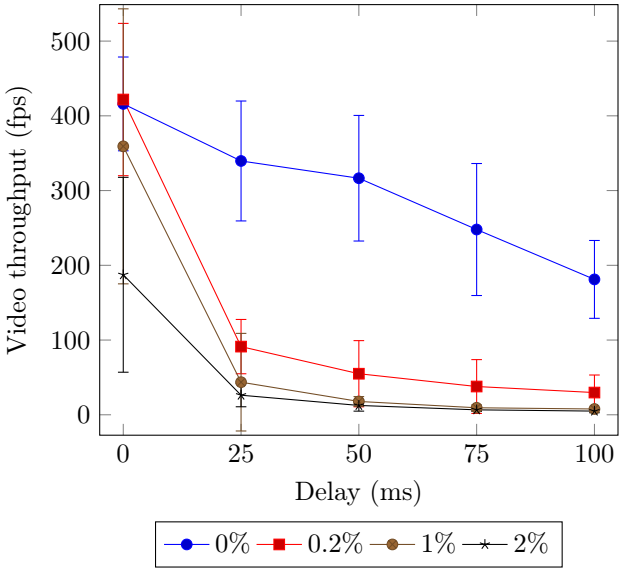


Figure 4.11: 720p video streaming performance between FUSE nodes for increasing delay at several levels of packet loss.

Tests are performed to empirically confirm the parts of the challenges related to FUSE performance and resource consumption.
The memory consumption and required disk space tests confirm that FUSE can be deployed on a wide range of devices, according to the third challenge. However, master nodes have to process a lot of OpenVPN traffic, which is CPU intensive, so devices with at least 2 available cores and hardware encryption such as AES-NI [28] are recommended.
Starting a FUSE federation is shown to be fast, taking only 5 to 6 minutes. The time to join a federation is dependent on network quality, but barring extremely hostile network conditions, a worker node should take about 1 to 4 minutes to join a federation. These numbers show that in most circumstances, it is possible to set up an entire federation in 10 minutes or less, which is enough to complete the third challenge. Because the resource requirements for master nodes are relatively low and every minute counts in crisis situations, it could be a good idea to keep a master node running at all times. This approach would cut response time to just 1 to 4 minutes when a situation arises.
The network performance of a FUSE federation is explored by both measuring pure throughput and by using a video streaming setup which mimics a client PC viewing a security camera stream. The results of the throughput test suggest that most of the performance limitations are due to OpenVPN.

For TCP traffic the network speed is acceptable and saturates a 100 Mbit/s network. UDP is almost 10 times slower, so applications running on a FUSE federation should consider using TCP. Importantly, since OpenVPN performance is CPU bound, not connection bound, this bandwidth has to be shared by all worker nodes connected to the same master, which needs to be taken into account when setting up a federation.
The video streaming test shows the performance of streaming services under a variety of network conditions. If 24 FPS is taken as a minimum requirement for a smooth 720p video stream, performance is good enough to handle 17 video streams simultaneously under ideal circumstances, which is sufficient for the first challenge. However, performance drops quickly with increasing packet loss. Around 1% packet loss and 50ms delay, UDP is likely a better choice, since packet loss will only result in image corruption, but no frame rate decrease. A simple mathematical model has been worked out to predict FUSE TCP performance, which could help evaluate the choice for either TCP or UDP under a given set of circumstances.
In future work, OpenVPN performance could likely be improved [27], especially since the test results show that FUSE throughput only reaches about 10% of the capacity of the gigabit line used for the tests. While improving OpenVPN performance may not help for scenarios with low network quality, it would at least increase throughput under optimal network conditions, reduce CPU load, or make larger worker pools practical. Alternatives to OpenVPN, such as Wireguard[19], are also considered.
A high availability mechanism should also be implemented in the future. In the test setup, only one master node is used, but a failure of the master node would disrupt the entire federation. Some of the challenges are minimizing the number of hops for pod traffic between master nodes, handling the independent VPN networks started by each master, keeping configuration information up to date over the entire high availability network and making sure nodes get delegated to a new master when one of them fails.
The creation of multiple master nodes can also reduce OpenVPN CPU load compared to a single master, and optimize traffic flow. In a single master setup, there is not only a centralized controller for the federation, but since all traffic between worker nodes has to go via the master node's VPN server, it flows according to a star topology. In a high availability setup, each master node would have its own star topology formed by its set of worker nodes. Only when worker nodes with different masters need to communicate does any traffic flow between the master nodes, and this sort of traffic could be minimized by planning or reassigning worker nodes to a different master.
Kubernetes pods can be moved from one node to another to ensure the load

[19] https://www.wireguard.com/

is distributed across all nodes in the cluster. For stateless microservices this poses no problem but services such as databases or queues require backing storage that needs to persist across the same instances regardless where they run. This storage is typically configured in advance and a Kubernetes volume plugin is deployed to allow for dynamic volume provisioning. In the context of FUSE, further research can be done to automate this deployment and dynamically increase the storage through simple node labelling.

4.4 Self Organizing Edge

This section contains the edited version of the following publication: **"Self-organizing Fog Support Services for Responsive Edge Computing"**, T. Goethals, F. De Turck, B. Volckaert
published in **Journal of Network and Systems Management volume 29, Article number: 16 (2021)** [29]

This section expands on the concept of Swirly from Chapter 3.4 through decentralization and self-organization. First, the motivation for a decentralized orchestrator is outlined, after which an alternate version of Swirly is constructed that is better suited for scalability in the network edge.
To reiterate, unlike cloud networks, fog and edge networks are very volatile, with constantly changing network conditions and topologies. Not only can devices suddenly appear or disappear in the network, but the physical location of edge nodes such as vehicles and mobile phones can also change rapidly.
Cloud data centers generally contain powerful servers that can run many service instances, and services can be migrated between machines with little penalty. However, in the fog, hardware is often less powerful and it is harder to migrate services due to the physical distances and resource limits involved. Therefore, it is important to take these constraints into account. As a corollary, scaling in the cloud is very geographically constrained, while the nature of the fog is more suited to scaling across many devices over a wide geographical area, placing service instances close to where they are required to minimize latency.
Finally, because cloud servers are clustered in data centers, service scheduling can usually be handled by a single, centralized scheduler instance. Even distributed cloud data centers are usually few in number and connected by high-bandwidth connections, so that a single scheduler instance in any of them can gather all the required information to make scheduling decisions. However, the fog and edge contain orders of magnitude more devices than clouds, connected at lower bandwidths. Combined with the volatility of the

fog, this makes gathering all the required data for scheduling decisions on a single node unrealistic; a fully decentralized solution is more suitable. This is illustrated by the evaluation of Swirly in Section 3.4.5, in which the memory requirements and network traffic at the node running the algorithm are shown to be limiting factors of its scalability.
While a decentralized solution can address the issues listed so far, it can also pose new problems. Most importantly, the resource use on fog nodes and edge nodes will be higher than with a centralized algorithm, because each node needs to perform part of the network communication and calculation that would otherwise be done in the cloud. Additionally, since there is no longer any static, single controlling entity (e.g. the cloud), global actions such as creating an overview of all running nodes or forcefully pushing updates will be more difficult.
As an example application, when using roadside units (fog nodes) near highways the algorithm makes it possible for services to "follow" clusters of cars (edge devices) by allowing edge nodes to constantly switch to the nearest units for service calls, and increasing the number of services in busy areas. Because of the decentralized approach, this can be done locally and on a large scale, without the need for cloud processing and the accompanying additional latency.
This section presents a decentralized approach to serverless fog and edge service scheduling, named SoSwirly (Self-organizing Swirly), which is based on five requirements:

- ***Req. 1*** Handle changing topologies and moving nodes in near real-time
- ***Req. 2*** Take fog node locations and resource limits into account
- ***Req. 3*** Balance the number of service instances versus QoS requirements, such as minimal overall latency
- ***Req. 4*** Scale to hundreds of thousands of edge devices through a self-organizing, decentralized approach
- ***Req. 5*** Work on a wide range of fog and edge devices by minimizing resource requirements

While *Req. 1* through *Req. 3* are similar to the requirements stated for the original Swirly, *Req. 4* and *Req. 5* are new, and necessary for a successful decentralized approach.
First, a theoretical model is constructed of fog nodes as service providers, taking into account these requirements, leading to an implementation of SoSwirly, and both a theoretical and empirical analysis of its performance.

The conclusions show that SoSwirly is highly scalable under the right conditions, and that these conditions can be achieved in a variety of situations by tuning configuration parameters depending on fog node density.
Afterwards, the rest of this chapter details the implementation of a distributed approach of Swirly, while fulfilling the requirements stated above. The core tenet of this approach is that each node is responsible for mapping only an arbitrarily small part of the global topology, its neighbourhood. This concept of neighbourhoods can, with the right configuration parameters, ensure that nodes only have to communicate with a constant number of neighbours, no matter how large the node topology grows. Additionally, the traditional planning hierarchy is inverted by making edge nodes responsible for deciding which fog nodes they want to address for a specific service.
As in Chapter 3.4, a fog network (including the edge) with frequent changes to its network and nodes will be referred to as a swirl. A fog node is synonymous with a service provider and similarly, a service client or edge node client indicate an edge node. Finally, an edge service refers to a process of any kind that is dependent on the fog and runs on an edge device.
The rest of this Chapter is organized as follows: Section 4.4.1 presents existing research related to the topic, while in Section 4.4.2 a mathematical model of fog nodes and fog networks is constructed as a basis for the implementation and evaluation of the solution. The solution itself and its implementation details are presented in Section 4.4.3, while Section 4.4.4 explores the theoretical properties of the implementation. The evaluation methodology for the solution in a number of scenarios is described in Section 4.4.5, the results of which are presented in Section 4.4.6. Finally, Section 4.4.7 discusses a number of topics for future work, and summarizes how the solution and the evaluation results meet the proposed requirements.

4.4.1 Related Work

A literature review by Maenhaut, Volckaert, Ongenae and De Turck [30] discusses challenges related to service orchestration, resource management and pricing in (distributed) clouds and the fog. An overview of the challenges in fog and edge computing is presented by Avasalcai, Murturi and Dustdar [31]. Their study focuses specifically on challenges in resource management, security and network management, with regards to the volatile network conditions and low-power devices often found at the network edge.
To support the scaling of services in the fog and edge, several studies have focused on extending the concept of serverless computing to these areas. For example, the Fog Functions presented by Cheng, Fuerst, Solmaz and Sanada [32] are compact and scalable pieces of functionality, created through a custom programming model which ensures independency of the device

and platform they are run on. Gadepalli, Peach, Cherkasova, Aitken and Parmer [33] argue that current virtualization technologies are too demanding and slow for responsive services on extremely low-power devices. They present aWsm, a serverless framework based on WebAssembly with startup times measured in microseconds. Although the implementation of SoSwirly presented in this section is focused on containers, the design allows for the adoption of these or similar virtualization technologies.

A recent literature review of container orchestration in fog networks [34] shows that the most popular research directions in this area involve Docker containers, scaling, QoS, and resource management. The approach presented in this section is in line with these facets, although the design of SoSwirly allows for flexible QoS requirements and the use of other forms of software deployment (e.g. Virtual Machines, unikernels [35]).

A study by Guerrero, Lera and Juiz [36] compares several multi-objective evolutionary algorithms in the context of fog service placement. Although these algorithms can take a large number of parameters into account and produce near-optimal results, they are by necessity used in centralized schedulers and for small-scale scheduling due to their speed. In contrast, SoSwirly aims to optimize only distance and number of services, but at a much larger scale. Similarly, a study by Hosseinzadeh et al. [37] provides an overview of various multi-objective optimization algorithms in service networks, but in the context of selecting optimal services rather than deploying them in optimal locations. Stévant, Pazat and Blanc [38] propose a framework which monitors and optimizes service placement to minimize QoS requirements, represented by response time in their evaluation. The difference with the approach in this section is that SoSwirly attempts to balance QoS requirements versus total used resources. Although the default Kubernetes [39] scheduler is centralized, an alternative scheduler by Casquero et al. [40] allows for distributed fog scheduling in Kubernetes. However, this approach is aimed at industrial automation, and the work in this section aims to go beyond the scale limits of Kubernetes [1]. Considering the cloud-oriented nature of Kubernetes, and its limits in terms of deployable pods, this work is not aimed at Kubernetes clusters, but instead uses small parts of its API where useful in communication between nodes. While the proposed solution in this manuscript primarily relies on decentralization to avoid network traffic bottlenecks caused by monitoring nodes from a central location, the work by Vaton et al. [41] models network delay with hidden Markov models to greatly reduce the amount of monitoring traffic at the cost of processing power. However, the decentralized nature of SoSwirly allows tuning each node separately should they encounter network bottlenecks, and since it is aimed at low-resource devices with little process-

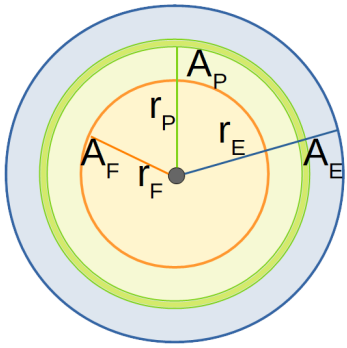
ing power to spare, advanced traffic reduction methods are not integrated. A closely related framework proposed by Santos, Wauters, Volckaert and De Turck [42] is aimed at optimizing latency in 5G networks and takes an application-oriented approach, placing the constituent microservices of an application on specific fog nodes to minimize latency and traffic overhead. Unlike their work, the solution in this section is agnostic of application architecture and transitive dependencies; a similar effect emerges from placing each service as close as possible to its direct dependencies, which can be achieved by carefully designing the distance metric for SoSwirly.
Compared to the aforementioned studies, the novel aspect of SoSwirly is that it is fully decentralized and works in near real-time, reacting to topology changes in milliseconds to seconds, depending on network latencies. It combines most of the discussed features by taking into account node resources, node mobility, service latency, and the number of service instances, while allowing for specific implementations using additional parameters. Meanwhile, the framework itself uses minimal resources and is highly scalable, because it only considers the status of nearby nodes.

Table 4.2: Definitions of symbols used in Section 4.4.2.

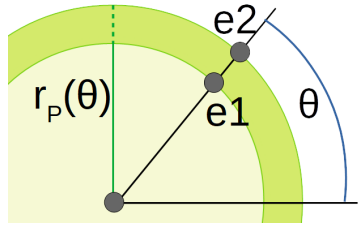
Symbol	Definition
A_E	the service area of a fog node based on its resources, see C_e
A_F	area closer to a fog node than to any other fog node
A_P	area within D_m metric distance
A	generic area, usually the entire service topology
r_E	radius of A_E, or distance to its border in a specific direction
r_F	distance(s) to the border of A_F
r_P	radius of A_P, or distance to its border in a specific direction
$\rho_E(x, y)$	edge node density at position x, y
$\rho_F(x, y)$	fog node density at position x, y
C_e	number of service clients supported by a fog node, based on its resources
C_p	scale factor between physical distance and metric distance
C_f	expected fog node density, used in areas with low ρ_F
D_m	maximum metric distance value
D(p1,p2)	metric distance between p1 and p2
A_{px}	the geographical area represented by a single pixel

4.4.2 Fog Node Model

In this section, a theoretical model of fog nodes and fog networks first presented in Chapter 3.4 is extended, which helps to meet the requirements



(a) Idealized responsibility areas of a fog node.



(b) Example of two edge nodes at different physical distances from a fog node, but with the same metric distance, resulting in a range of values for r_P.

Figure 4.12: The different responsibility areas of a fog node.

put forward in Section 4.4 that are related to the fog network topology, specifically ***Req. 1*** through ***Req. 4***. Table 4.2 summarizes the symbols used in this section that are not explicitly defined through equations.
As per ***Req. 3***, SoSwirly should support QoS requirements. As in Swirly, this is achieved through a distance metric, combined with a maximum distance, to ensure that edge nodes are always serviced by the "closest" available fog node.
Fig. 4.12 reiterates the concept of fog node *service areas*, idealized as circles:

- A_E, defined by radius r_E, is the capacity area of a fog node, representing the physical area it can service based on the capacity C_e imposed by its resources.
- A_F, with radius r_F, is the responsibility area of a fog node. All edge nodes within this area should be serviced by a fog node because they are closest to that fog node than any other.
- A_P, with radius r_P, is the proximity area of a fog node. All edge nodes within this area are close enough that they can be serviced by the fog node without going over the maximum distance metric value.

Starting from the definitions used by Swirly, these can be naively calculated using

$$r_F = \frac{1}{\sqrt{\pi \rho_F}} \tag{4.5}$$

$$r_E = \sqrt{\frac{C_e}{\pi \rho_E}} \tag{4.6}$$

$$r_P(\theta) = r, \forall r, \theta : D(r, \theta) = D_m \tag{4.7}$$

Where ρ_F and ρ_E represent the density of fog and edge nodes respectively, and D_m is the maximum distance value between a fog node and any edge node it services. $D(r, \theta)$ is a function that returns the metric distance of the point r, θ. Note that the distance function can return multiple values for r_P for any θ, as shown in Fig. 4.12b. In this example, e1 and e2 lie in the same direction θ, but with a different r. However, the metric distance for both is equal, so $r_P(\theta)$ returns a range of values. Therefore, for A_P to be an actual circle, its radius must be the maximum value of $r_P(\theta)$. In cases where the distance metric preserves the relative ordering between all points, r_P can be reduced to

$$r_P = C_p D_m \tag{4.8}$$

In which C_p is a constant that describes how the distance metric scales with the physical distance between points. The definition of r_E naively assumes a constant edge node density. The original Swirly assumed a metric with a coordinate system that can be simulated by polar coordinates. The approach can be generalized by assuming the distance metric uses a coordinate system (x', y'), giving a more accurate value for r_E is by solving the following equation [43] for the area A, which can be transformed back to Cartesian coordinates:

$$\int\int_A J(x', y')\rho_E(x', y') \cdot \mathrm{d}x' \cdot \mathrm{d}y' \ = C_e \tag{4.9}$$

While the Jacobian [44] $J(x', y')$ makes this equation cumbersome, reintroducing the assumption that the distance metric can be calculated as polar coordinates gives:

$$\int\int_0^{2\pi} r\rho_E(r, \theta) \cdot \mathrm{d}r \cdot \mathrm{d}\theta = C_e \tag{4.10}$$

Which, since the distance function can not calculate θ, results in the simplified version from Chapter 3.4:

$$\int 2\pi r\rho_E(r) \cdot \mathrm{d}r = C_e \tag{4.11}$$

In all cases, for a uniform density ρ_E these equations reduce to Eq. 4.6

$$\int\int_0^{2\pi} r\rho_E \cdot \mathrm{d}r \cdot \mathrm{d}\theta = C_e \tag{4.12}$$

$$\left[\frac{r^2\theta}{2\rho_E}\right]_0^{\theta=2\pi} = C_e \tag{4.13}$$

$$r = \sqrt{\frac{C_e}{\pi \rho_E}} \tag{4.14}$$

Note that while Eq. 4.6 assumed the use of the Euclidean metric, this version is based on the distance metric expressed as polar coordinates. Therefore, the resulting r must be converted from metric distance to geographical distance before using it with Cartesian coordinates (e.g. a geographical map). As described in the original approach, these definitions lead to some conditions for a well-organized service topology, which can be summarized as

$$r_F \leq r_P \leq r_E \tag{4.15}$$

While this equation has its own merit in predicting efficient fog node operation given sufficient information, these relations can be used to determine requirements for the fog node density ρ_F. For the moment, r_P is assumed to be calculated as in Eq. 4.8. Then, since

$$\frac{1}{\sqrt{\pi \rho_F}} \leq min\left(C_p D_m, \sqrt{\frac{C_e}{\pi \rho_E}}\right) \tag{4.16}$$

there are two conditions for ρ_F, namely

$$\frac{1}{\sqrt{\pi \rho_F}} \leq C_p D_m \Rightarrow \rho_F \geq \frac{1}{\pi C_p^2 D_m^2} \tag{4.17}$$

$$\frac{1}{\sqrt{\pi \rho_F}} \leq \sqrt{\frac{C_e}{\pi \rho_E}} \Rightarrow \rho_F \geq \frac{\rho_E}{C_e} \tag{4.18}$$

As a result, the following equation shows how to calculate minimal fog node density at any location based on edge node density and the maximum distance:

$$\rho_F \geq max\left(\frac{1}{\pi C_p^2 D_m^2}, \frac{\rho_E}{C_e}\right) \tag{4.19}$$

At this point C_p can be substituted with a more advanced relation between geographic distance and metric distance. The *metric pressure* P represents the inverse concept of C_p, indicating the inflation of metric distance with respect to geographical distance at any point, calculated over the shape of the topology. Assuming a point $p1$ with Cartesian coordinates, and that $D(p1, p2)$ gives the metric distance between $p1$ and any other point $p2$, the metric pressure P can be defined at $p1$ as

$$P(p1) = \frac{\int_A \frac{D(p1,p2)}{||p1,p2||} \cdot \mathrm{d}A}{A} \tag{4.20}$$

Where $p2$ represents each point in the area A of the Swirl and $||p1, p2||$ is the Euclidean distance between $p1$ and $p2$. In practice, P will either be known from a few measurements or will not need to be calculated, reducing the computational load this would represent.
Using this definition, it follows that P can replace $1/C_p$ in Eq. 4.19, since a smaller C_p inflates $D(p1, p2)$ and thus P. As a result, equations can be constructed to calculate the minimal fog node density $\rho_F(x, y)$ at any location in a Swirl, and the minimum number of fog nodes N_{fog} required to service a given area of a Swirl, using any distance metric:

$$\rho_F(x, y) \geq max\left(\frac{P(x, y)^2}{\pi D_m^2}, \frac{\rho_E(x, y)}{C_e}\right) \tag{4.21}$$

$$N_{fog} \geq \int_A max\left(\frac{P(x, y)^2}{\pi D_m^2}, \frac{\rho_E(x, y)}{C_e}\right) \cdot \mathrm{d}A \tag{4.22}$$

4.4.2.1 Fog Neighbourhood Discovery

The maximum distance value D_m in Eq. 4.8 represents the maximum distance between edge nodes and their service providers. However, fog nodes also interact with each other in the form of a neighbourhood discovery process, which is explained in Section 4.4.3.1. Ideally, the maximum distance used in this discovery process would be a single value, for example D_m used by edge nodes. However, Fig. 4.13 illustrates how this can lead to problems. While $f3$ is in the green circle, and thus should clearly be known to $f1$, $f2$ is too far from $f1$ for $f3$ to be discovered through it.

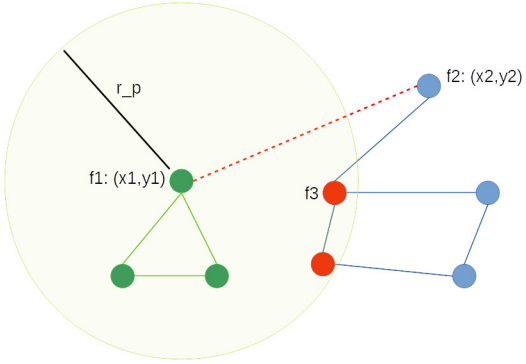


Figure 4.13: Potential problem with fog discovery when using a constant maximum distance for discovery. Fog node f3, despite being in the green circle representing the neighbourhood of f1, will not be discovered because the connecting node f2 is too far away from f1. A_P, with radius r_P, is drawn as a circle for illustrative purposes.

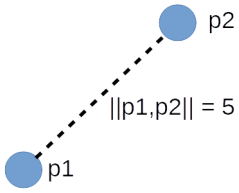
This problem can be solved if, similar to how minimum fog node density is calculated, the maximum discovery distance D_{mf} depends on both maximum metric distance and local fog node density, so that

$$D_{mf}(x,y) = max\left(D_m, \frac{C_f}{\sqrt{\pi\rho_F(x,y)}}\right) \tag{4.23}$$

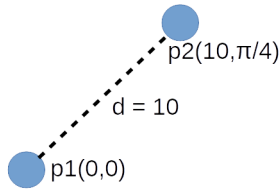
Where C_f is a constant representing the expected fog node density. The higher this number is, the more connected the fog network will be, but traffic and resource use will increase with it. Note that this does not solve the problem of $f1$ being unable to discover $f2$, but it will enable $f2$ to discover $f1$. Eventually, $f3$ will find $f1$ through $f2$, at which point $f1$ will discover its remaining neighbours.
There is a possible solution for D_{mf} which would allow the use of a single value while discovering all required nodes, if instead of $\rho_F(x,y)$ the minimum value of ρ_F over the entire topology is used. However, this would result in an unnecessarily large neighbourhood for most fog nodes, and a lot of traffic overhead.

4.4.2.2 Distance Metric Coordinates

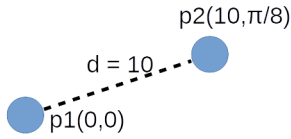
The native coordinate system of the distance metric may contain any number of dimensions with discontinuous values (e.g. binary flags or enum values). However, an explicit coordinate transformation with respect to a fully differentiable manifold [45] is required to properly calculate positions and relative distances between fog nodes and edge nodes in terms of the distance metric. Additionally, while the Euclidean metric is (implicitly) used in all equations so far, with the distance metric used as a function, the distance metric should instead be treated as a metric, but one which is only defined on its native coordinate system. This subsection shows how the distance metric values determined by nodes can be used to construct surrogate distance metric coordinates for nodes in Cartesian and polar coordinates.
In terms of polar coordinates, the metric distance d is assumed to scale with r, which is also the assumption in C_p in Eq. 4.8. Examples of metrics with this property are geographical distance between nodes and latency under normal circumstances, within a reasonable margin of error. This type of distance metric transforms the topology shown in Fig. 4.14a into distances and node positions similar to those shown in Fig. 4.14b. However, in other cases the distance metric will be more complex and it will not scale equally with Cartesian or polar coordinates in every dimension, nor at every location, as shown in Fig. 4.14c. Moreover, two physically very different nodes may map onto the same location in terms of distance metric



(a) Physical topology



(b) Assumed position of p2



(c) Real position of p2

Figure 4.14: Example of erroneous assumption of distance metric coordinates from known positions and measured metric values between $p1$ and $p2$. θ is not known from measurements, but has to be assumed.

coordinates.

This presents problems for advanced functionality where the coordinates of the nodes are required, e.g. predicting node movement and trajectory in terms of the distance metric. However, if the distance metric preserves relative Euclidean distances, accurate surrogate coordinates can be constructed for each node, similar to Fig. 4.14b, by assuming that r is the measured distance d and θ is the same as the angle between both nodes in polar coordinates.

When constructing surrogate coordinates from the point of view of a point $p1$, it can be placed at the origin with coordinates $(0, 0)$. The surrogate coordinates of a point $p2 : (r, \theta)$ relative to $p1$ are thus defined in polar coordinates as

$$p2' = (D(p1, p2), \theta) \tag{4.24}$$

In Cartesian coordinates, the metric ratio s can be used to scale the relative distance between $p1$ and $p2$:

$$s = \frac{D(p1, p2)}{||p1, p2||} \tag{4.25}$$

The entire issue is circumvented in all equations so far by using the geographical (Cartesian) coordinates of nodes, either because it makes no difference or because the potential error is very small. For example in Eq.

4.21 $\rho_E(x, y)$ is used instead of $\rho(r, \theta)$, but this makes no difference since it merely describes the local node density of an infinitesimal region, without respect to another point. Other concepts, such as the metric pressure P, were specifically designed to avoid having to use the metric coordinate system. However, these surrogate coordinates may be useful in future work.

4.4.2.3 Fog Hardware Placement

While Eq. 4.21 allows for the calculation of the minimum required fog node density at any location, it is very general and difficult to implement. The evaluations in this section use a more straightforward distance metric so that $1/C_p$ can be used rather than $P(x, y)$. Furthermore, it is more useful to have a version of the equations that can handle discrete regions with piecewise constant densities (e.g. pixels).
The discrete version of Eq. 4.21 using a straightforward distance metric is

$$\rho_F(x, y) \geq max\left(\frac{1}{\pi C_p^2 D_m^2}, \frac{\rho_E(x, y) A_{px}}{C_e}\right) \tag{4.26}$$

in which A_{px} represents the surface area of a single region. In the case of pixels, this is a constant and the following can be substituted

$$C_a = \frac{A_{px}}{C_e}, \rho_p = \frac{1}{\pi C_p^2 D_m^2} \tag{4.27}$$

which are both constant, resulting in a short version of Eq. 4.26

$$\rho_F(x, y) \geq max(\rho_p, \rho_E(x, y) C_a) \tag{4.28}$$

This equation can be used to determine the minimum total number of fog nodes over the entire topology, giving the discrete version of Eq. 4.22

$$N_{fog} \geq \sum^{X} \sum^{Y} max(\rho_p, \rho_E(x, y) C_a) \tag{4.29}$$

Note that both Eq. 4.28 and 4.29 are only valid if the distance metric preserves relative distances so that the quotient of the metric distance and physical distance can be represented by C_p, and furthermore that all discrete regions are the same size, in which case A_{px} is constant. In other cases, the equations will be similar, but more terms will have to be calculated for each region.

4.4.3 Fog Service Provisioning

SoSwirly, the solution presented in this section, requires a number of logical components to fulfil the requirements listed in Section 4.3. A decentralized

node discovery algorithm is required to keep track of changing topologies on a large scale, as per ***Req. 1*** and ***Req. 4***. Similarly, ***Req. 2*** and ***Req. 3*** must both be taken into account in a scheduler component, which finds the best fog node to use as a service provider. Upon finding a suitable node, or when the service provider changes, ***Req. 1*** requires a component that can forward any requests to the correct node. This section describes how these logical components are implemented by software components, with ***Req. 5*** implicitly present in the design through the choice of Golang and lightweight dependencies, e.g. using a straightforward DNS solution, using FLEDGE agents instead of Kubelets. FLEDGE, introduced in Chapter 3.3, is a lightweight container orchestrator for edge nodes, similar to K3s [46], but fully compatible with Kubernetes master nodes.
The chosen distance metric is implemented as a custom ping web service that determines latency. This choice of metric, which is implicitly assumed to scale with geographical distance, allows the basic properties of SoSwirly to be explained intuitively, by relying on the reduced forms of the equations presented in Section 4.4.2.
The logical components are divided among two separate services, which are implemented in Golang for the evaluations in Section 4.4.5. The *fog node service*, deployed on all fog nodes, is responsible for managing fog services based on edge node requests and for discovering other fog nodes in its neighbourhood. The *edge node service* is deployed on all edge nodes and monitors which services on an edge node require fog support services. When required, these support services are requested from the nearest (active) fog node, which is found by traversing fog node neighbourhoods discovered by the *fog node service*.
Central to both services is the algorithm for node discovery, which fog nodes use to discover their neighbourhoods and edge nodes use to find an optimal service provider.
In principle, the edge node service is not limited to running on edge nodes. It can also be deployed on fog nodes, allowing a tiered fog architecture in which each layer detects the services running on its devices and attempts to deploy the required support services in a higher layer.

4.4.3.1 Neighbourhood Discovery

Algorithms 4 and 5 presents the general outline of the algorithm used to discover nearby fog nodes from any node.
At the start of each discovery round, a queue N_{check} is created from the elements of the list of known nodes N_{known}, and nodes discovered through other means N_{new} which do not yet have a distance assigned to them. The list of nodes to ignore N_{ignore} is initialized, to which nodes are added that

```
function Discover(selfNodeType, N_known, N_new) is
    N_check = N_known ∪ N_new
    N_ignore = ∅
    if N_check ≠ ∅ then
        d_min = D(selfNodeType, N_check[0])
    inReach = false
    while N_check ≠ ∅ do
        n_c = N_check[0]
        Remove(N_check, n_c)
        if ShouldReping(n_c) then
            d[n_c] = D(selfNodeType, n_c)
            d_max = AdjustedDistance(Len(N_known))
            if d[n_c] ≤ d_max) then
                add(N_known, n_c)
                N_new = GetKnownNodes(n_c)
                MergeNodes(N_check, N_new, N_ignore)
                inReach = true
                d_min = min(d_min, d[n_c])
            else
                Add(N_ignore, n_c)
                if d_min ≥ d_max then
                    N_new = GetKnownNodes(n_c)
                    MergeNodes(N_check, N_new, N_ignore)
                if Len(N_known) ≤ 1 or d[c_n] ≤ d_min then
                    d_min = d[c_n]
                else
                    Remove(N_known, c_n)
                end
            end
    end
    if inReach then
        N_remove = ∅ for n ∈ N_known do
            if d[n] >AdjustedDistance(Len(N_known)) then
                Add(N_remove,n)
        end
        N_known = N_known ∩ N_remove
end
```

Algorithm 4: Node discovery mechanism used in both the fog node service and the edge node service.

function *ShouldReping(c_n)* **is**
| *customizable implementation, default true*
end
function *GetKnownNodes(c_n)* **is**
| *webservice call to* n_c
end
function *AdjustedDistance(n)* **is**
| $\rho_{local} = \rho_{ass}/n$
| **return** $d_{max} \cdot max(1, \sqrt{(\rho_{local})}$
end
function *MergeNodes(N_{check}, N_{new}, N_{ignore})* **is**
| $N_{check} = N_{check} \cup (N_{new} \cap N_{ignore} \cap N_{known})$
end

Algorithm 5: Support methods for node discovery mechanism.

are beyond the maximum distance and should not be contacted again this round. As long as there is a node left in N_{check}, the first element n_c is taken from the queue. The algorithm first checks if the distance to n_c should be updated this round by calling *ShouldReping*. If so, the distance $d[n_c]$ is determined and updated and d_{max} is calculated by the *AdjustedDistance* function according to Eq. 4.23. If the new distance to n_c is smaller than d_{max}, the node is added to (or updated in) the list of known nodes, and its list of known nodes is fetched into N_{new}. These nodes are in turn added to the list of nodes to check through the *MergeNodes* function, which only adds those nodes of N_{new} to N_{check} which are not in N_{check}, N_{known} or N_{ignore}. Additionally, the flag *inReach* is set to true, and the minimal distance d_{min} for this round is updated if necessary. If the new distance to n_c is larger than d_{max}, the node is added to the list of nodes to ignore. However, if the minimum distance in this discovery round is also larger than d_{max}, the algorithm has not yet found a way to nodes within its neighbourhood. Therefore, the list of nodes known to n_c is fetched and merged with the list of nodes to check, in an attempt to find closer nodes. Additionally, if there is only one known node, or the distance to n_c is smaller than the smallest distance so far, d_{min} is also updated to narrow the search field. If neither of these conditions are true, c_n is removed from the list of known nodes if it is in there. When the queue N_{known} is empty, the *inReach* flag is checked to remove any nodes beyond the maximum distance from the list of known nodes, because such nodes may have been added while d_{min} was larger than d_{max}

When contacting another node, a node will always pass its node type (e.g. edge or fog) in the request. When a fog node is contacted by an edge node, it will simply respond and the distance between the two can be determined

by the edge node. When a fog node is contacted by another fog node, it will check if it already discovered that node; if not it will add the node to a queue of nodes to contact on the next discovery round.
As explained in Section 4.4.2.1, the maximum distance for fog node neighbourhood discovery should not be a constant value, but should depend on local fog node density as well. The local fog node density can be estimated by considering how many fog nodes have been discovered by using the default maximum distance; the new maximum distance can be calculated from it according to Eq. 4.23. However, edge nodes still use a constant maximum distance to discover optimal service providers in accordance with Eq. 4.8.
Ideally, all discovered fog nodes should be contacted during each discovery round to determine their new distance. However, to reduce neighbourhood size and limit edge node traffic, the *ShouldReping* function can be implemented differently for fog and edge nodes, enabling different timeouts on nodes that are far away. For example, while some fog nodes beyond the maximum distance are kept in the list of known nodes, this feature allows the algorithm to only contact them every 2 or 3 rounds.

4.4.3.2 Fog Node Service

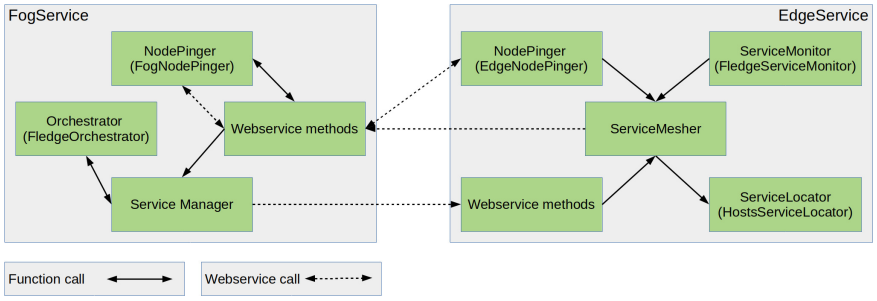


Figure 4.15: Components of the fog node service and the edge node service and their interactions.

The left side of Fig. 4.15 shows the components of the fog node service and their interactions.
The *FogNodePinger*, an implementation of *NodePinger*, periodically attempts to discover fog nodes in its neighbourhood according to Algorithm 4. It is used by the webservice methods to return a list of known fog nodes, and in turn calls on the webservice methods of other fog nodes for neighbourhood discovery.
The *Service Manager* handles requests from edge nodes to add or remove them as service clients. When adding a service client, it checks if there

are sufficient resources to handle extra service clients or services, and starts any services needed by the edge node client if they are not yet running. When a service is started, it is loaded from a configuration file containing a Kubernetes PodSpec [47] which is sent to the *Orchestrator* for deployment. When a service client is removed, the Service Manager checks each service used by that client to see if there are still enough other clients using them. If the number of clients for any service is below a configurable minimum, all remaining edge node clients are notified to try and find a different fog node for that service. If successful, the migration is confirmed to each client, they are then removed from that service and the *Service Manager* instructs the *Orchestrator* to stop and remove the service itself. If a single edge node fails to find another suitable service provider, the edge node clients are notified that the migration should be reverted.
The *FledgeOrchestrator* is an implementation of *Orchestrator* and deploys Kubernetes pods through a FLEDGE agent. It supports only the two methods described above; one to deploy a pod and one to stop a pod.

4.4.3.3 Edge Node Service

The components of the edge node service are shown on the right side of Fig. 4.15, along with their interactions and web service calls to and from the fog node service.
As in the fog node service, the *EdgeNodePinger* is responsible for discovering nearby fog nodes as per Algorithm 4. The difference is that the results of each discovery round are actively used, by sending them to the *ServiceMesher*, which takes further action based on the support services required by the edge node.
To determine which support services are required, the *FledgeServiceMonitor* periodically checks all running pods managed by the local FLEDGE agent and compares them to a configurable map of edge service names and the fog services they depend on. When a new pod is detected, the map is consulted and any required fog services are forwarded to the *ServiceMesher*.
The *ServiceMesher* uses the fog nodes discovered by the *EdgeNodePinger* to find a suitable service provider for the required services passed by the *FledgeServiceMonitor*. For each service S it first attempts to find a fog node which already has S deployed, in order of increasing distance, up to the maximum distance. If this fails, the closest node with enough available resources to deploy S is selected as service provider, no matter the distance. The selected node is then notified that the edge node requires S, and registers itself as a service client for S at that fog node. Note that this can result in multiple service providers for a single edge service, depending on pre-existing deployments in the fog. If for any reason a fog service deploy-

ment fails, or the edge node can not be registered as a client, the search for a service provider continues. This approach is in line with the original Swirly algorithm, which prefers nearby fog nodes with active service deployments and free resources over empty fog nodes when possible. Any connections from the edge service are redirected to the correct fog node by the *HostsServiceLocator*. For the purposes of this section, this naive implementation of *ServiceLocator* assumes that the name of the fog service is used as a DNS name when it is called by the edge service. This assumed DNS name is linked to the IP address of the fog node by adding a line in the hosts file [48] of the edge node.
As described in Section 4.4.3.2, a fog node can ask edge node clients to migrate to a different service provider for a specific service. When this happens, the *ServiceMesher* attempts to find a service provider as described in the previous paragraph, with the exception that the resulting fog node must already be running the required fog service, and is different from the current service provider. If a suitable fog node is found, the edge node assigns it as backup service provider and adds itself as a client for the required service to ensure that it can be serviced. If any of these steps fail, the edge node reports to the original service provider that the migration has failed, otherwise it reports the migration can proceed. In the final step, the edge node client is instructed by the fog node to either confirm or revert the migration. When confirming, the backup service provider is given active status and the *HostsServiceLocator* is updated accordingly. When reverting, the edge node removes itself from the backup service provider as client.

4.4.4 Theoretical Performance

In terms of processing and memory, the average computational complexity of neighbourhood discovery using Algorithm 4 is $O(r_P^2 \rho_F)$. r_P^2 describes the area with which a fog node interacts, while the amount of interaction increases with ρ_F. Extending this to interaction with edge nodes, the complexity is $O(r_P^2 \rho_F)$ on edge nodes and $O(r_P^2 \rho_E)$ on fog nodes. However, in the worst case a fog node has to search a significant fraction of all fog nodes to find any that belong to its neighbourhood. These adverse situations may be caused by topological features as shown in Fig. 4.13, or because the (randomly) assigned start node for the discovery process is several times the maximum distance away. In both cases performance will converge to the average situation within a few discovery rounds, but worst case complexity is therefore $O(F)$ in terms of processing.
Detecting and deploying support services has a best case complexity of $O(1)$ in terms of processing, since the closest fog node is known from the discovery process and can be found in constant time. In situations with very few fog

Table 4.3: Summary of processing complexity.

	Best	Average	Worst
Neighbourhood discovery	-	$O(r_P^2 \rho_F)$	$O(F)$
Fog discovery (edge node)	-	$O(r_P^2 \rho_F)$	$O(F)$
Fog discovery (fog node)	-	$O(r_P^2 \rho_E)$	$O(r_P^2 \rho_E)$
Service deployment	$O(1)$	$O(1/(1 - \frac{\rho_E}{\rho_F}))$	$O(r_P^2 \rho_F)$

resources left, worst case performance is $O(r_P^2 \rho_F)$, since all neighbouring fog nodes may have to be consulted to find one with free resources. Average complexity is between these extremes, depending on the amount of nearby edge nodes and fog nodes, resulting in $O(1/(1 - \rho_E/\rho_F))$. In all cases, memory complexity is $O(r_P^2 \rho_F)$. Table 4.3 summarizes the computational complexities for all cases.
In all cases, the complexity of network throughput will be the same as processing complexity, since every action in neighbourhood discovery and service deployment requires a webservice call.
Because the best and average case complexities are based only on local node densities and maximum discovery distance, ***Req. 4*** is met by the theoretical performance of SoSwirly.

4.4.5 Evaluation Methodology

This section describes the evaluation setup, the scenarios involved in the evaluation and their limitations, in which terms SoSwirly is evaluated and how the results are measured. The code of SoSwirly, including the tools used for the evaluations, is made available on Github[20].

4.4.5.1 Evaluation Setup

All evaluations are run on the IDlab Virtual Wall [49] using two servers, each with 48GiB RAM, two Intel E5-2650v2 CPUs and a Gigabit Ethernet connection. One server is used to host a number of fog node services, while the other hosts a number of edge node services.
To support running multiple fog and edge nodes on a single machine, and for other nodes to be able to access them, a number of changes are made to the code that are enabled through the configuration flag *testMode*. *testMode* is designed to use incremental node names and port numbers so that each node knows where to reach another node by name alone. Additionally, it disables the actual deployment of a pod in FLEDGE so as to not overload the server with pod deployments. Finally, only one FLEDGE instance is started,

[20] https://github.com/togoetha/soswirly

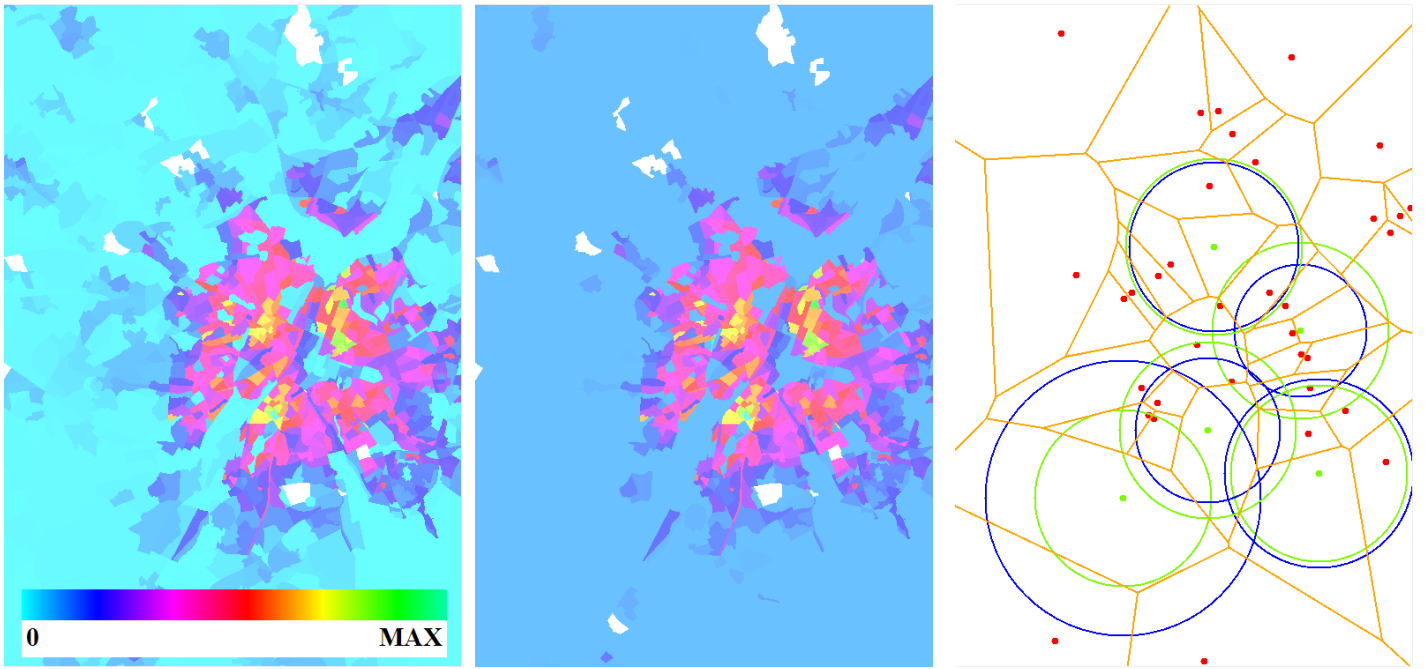
running a single instance of the service which edge nodes are configured to detect, and each edge node service monitors this single instance.
While the services can thus be run on a single server, they create a large number of network connections between them. This was found to put a practical limit on how many nodes can be emulated simultaneously, so the evaluations are limited to a maximum of 250 fog nodes and 150 edge nodes. Although this does not represent the topological scale SoSwirly is meant to operate at, it is impossible to simulate thousands of nodes on the Virtual Wall, and conclusions can still be drawn from evaluations within these limits.

4.4.5.2 Swirl Generator

The evaluations use randomized Swirls created by the Swirl generator. This generator accepts a number of parameters (e.g. number of fog nodes, maximum distance) and a population density bitmap as input, and uses the equations from Sections 4.4.2 and 4.4.2.3 to generate edge and fog nodes at suitable, but random, positions. The output of this tool is a set of configuration files that are used to start all the generated nodes on the evaluation servers. An example input density bitmap is shown in Fig. 4.16a, with the results of Eq. 4.28 in Fig. 4.16b. A small-scale visualization of the output of the generator is shown in Fig. 4.16c, overlaid with calculated A_P, A_F and A_E of a few selected fog nodes. Note that the responsibility areas take the shape of a Voronoi diagram and A_F is not defined by a single value r_F. This is because the equations in Section 4.4.2 assume an ideal shape for A_F, while the actual shape is dependent on random factors (e.g. placement of fog nodes, choice of active fog nodes, stability of distance metric over time). However, SoSwirly will attempt to produce A_F as close to ideal as possible, and the equations still hold when r_F is defined as the maximum distance between a fog node and the boundary of A_F.
The density bitmaps are generated by another tool, which combines official census data and GEOJSON[50] data into population density bitmaps. For the evaluations, data from StatBel [51] on the statistical sectors of Belgium [52] and population per statistical sector [53] is used. The statistical sectors are on average $1km^2$, although they are smaller in cities, and larger in sparsely populated areas, making them sufficiently fine-grained.

4.4.5.3 Scenarios

The base density bitmap used to generate Swirls is similar to that shown in Fig. 4.16a, but encompasses a larger area of around $660km^2$. Its right side is focused on Brussels, while the middle and left include the surrounding countryside. It contains population densities from $0/km^2$ to around $40000/km^2$.



(a) (b) (c)

Figure 4.16: Part of the physical area of Belgium used to generate node topologies and evaluate SoSwirly. The scale in (a) represents edge node density, where teal is low density, and red through green represent high densities. (b) shows the required fog node density as calculated using Eq. 4.28, using the same density scale. The darker blue shows that in the surrounding towns, ρ_E is overridden by the required maximum distance. (c) shows the service areas of selected fog nodes (background omitted). Orange is A_F, blue is A_E and green is A_P.

Considering the artificial nature of the evaluation, the parameters needed for the equations in Section 4.4.2.3 are approximated. Population density is used as a substitute for $\rho_E(x, y)$, while fog node capacity C_e is set at 50.000, making C_a $1,25 \cdot 10^{-8} km^2$. For the chosen metric and the given scenario, C_p can be set to 1 pixel per metric distance unit. These parameters are used to calculate the minimal fog density for each scenario. No cases are examined that contain only heavily populated areas or countryside, since in cities r_P can simply be lowered to suit the higher ρ_E and ρ_F, stabilizing performance, and in the countryside r_P can be increased to suit the discovery process. A mixed case exposes the stability and performance of SoSwirly when using a single set of parameters over a wide range of node densities.
As explained in Section 4.4.1, there are fundamental differences between SoSwirly and recent related work. The results of solutions based on evolutionary algorithms can be superior, but these solutions are centralized and can not run in near real-time. Others run in real-time, but require a central component or have different use cases (e.g. optimizing all the components of an IoT application in the fog). Keeping these nuances in mind, the efficiency of SoSwirly is compared to the conceptually comparable but centralized Swirly, and NSGA-II, a generic multiobjective genetic algorithm [54]. For NSGA-II, the optimization parameters are average distance between fog and edge nodes, and the number of service instances. The same parameters are used as by Guerrero, Lera and Juiz [36], except the evaluation is performed using both 400 and 5000 generations. Additionally, since the maximum distance is a soft restriction in SoSwirly, any distance above 100 is penalized by a factor of 5 in NSGA-II to discourage it.

Neighbourhood Discovery This scenario determines the impact of the neighbourhood discovery algorithm on fog nodes. Using only a set of fog nodes, the neighbourhood discovery algorithm is evaluated in terms of CPU use, memory use and network traffic. The number of fog nodes is varied from 50 to 250 in steps of 50, while the maximum distance is varied from 50 to 150 in steps of 50. Due to practical limits in the evaluation setup, the combination of 150 maximum distance and 250 fog nodes is not tested. For each combination of parameters, SoSwirly is run for 10 generated swirls. Additionally, the accuracy of neighbourhood discovery is measured to determine the optimal maximum distance. For this scenario, discovery rounds are run on each node every 5 seconds. While this reaction time may be too high for certain scenarios with stringent real-time requirements, it is configurable, and can be lowered significantly depending on the requirements.
Using Eq. 4.28, the minimum number of fog nodes N_{fog} for the evaluation is 119 for a maximum distance of 50, 40 for a maximum distance of 100 and

27 for a maximum distance of 150. Although the test parameters go below the minimum number of nodes for a maximum distance of 50, these cases will show how SoSwirly reacts when not provided with enough fog nodes.

Service Deployment In this scenario, the combined impact of fog node discovery, service detection and fog service deployment on edge nodes is measured in terms of CPU use, memory use and network traffic. The maximum discovery distance is set at 100, and 100 fog nodes are used for this scenario. Due to hardware limitations, this scenario was only evaluated for 100 and 150 edge nodes.

Topology Efficiency Additional information extracted during the evaluation of the *Service Deployment* scenario is used to determine how effectively SoSwirly minimizes both the number of active fog nodes and the average distance between edge and fog nodes. Both of these optimization parameters will be compared between SoSwirly and the centralized algorithms Swirly and NSGA-II, in addition to the time taken for each algorithm to determine a solution. For both Swirly and NSGA-II, the scenario is first prepared and only the timing of the actual algorithm is measured. For SoSwirly, the measured time includes some web service calls due to its decentralized nature and the necessity to gather node information ad-hoc.

4.4.5.4 Resource Measurement

Each evaluation is run for 5 to 10 minutes, depending on the required time to start all nodes, with resource levels logged every 10 seconds. The full range of data over time is not presented in this section because most of it represents the evaluation script starting up the required nodes. The results presented focus on the last measurement, which is the most resource intensive period representing a fully initialized, stable cluster in which neighbourhood discovery is mostly completed, but is still periodically checking for new nodes.
Considering the number of processes involved in the evaluations, CPU use per process is approximated by measuring global CPU use and dividing it by the number of nodes. The disadvantage of this approach is that it is impossible to measure the specific CPU use of nodes in extreme situations, such as the densely populated city center. However, this approach was chosen because measuring CPU use for each process separately may have an undue influence on CPU use itself. Despite this approach, the results allow for an analysis of CPU use in different situations in Section 4.4.6.
Unlike CPU use, memory use is measured for each process separately, taken from /proc/<pid>/stat. Because of this, memory use is examined for only

one topology per set of evaluation parameters rather than 10, allowing for an analysis of extreme cases. This approach is representative of all cases; while the topologies are generated randomly, they are still governed by the rules of the model in Section. 4.4.2. This is confirmed by comparing memory use per node across all the topologies; if the memory use of the most demanding nodes of each topology is compared, the standard deviation is 1.9%. For the least demanding nodes, the standard deviation is 2.1%.

Network traffic is measured globally from /proc/net/dev and divided by the number of nodes, for the same reasons CPU use is measured globally. In the *Neighbourhood Discovery* scenario, only the lo adapter is examined because all traffic is local, in the *Service Deployment* scenario only the traffic on the *eth0* adapter is measured, which indicates traffic between fog nodes and edge nodes.

4.4.6 Results

This section contains the results for the evaluations described in Section 4.4.5, along with a discussion of the results. The values in the presented charts are median values, with error bars representing minimum and maximum values.

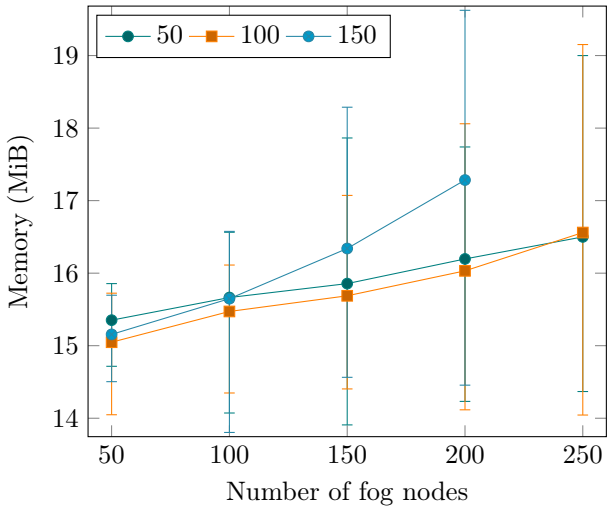


Figure 4.17: Memory requirements of the fog service for different amounts of fog nodes, and maximum discovery distances of 50, 100 and 150 units.

4.4.6.1 Fog Discovery

Fig. 4.17 shows the memory required by the fog node service for different maximum distances and varying numbers of fog nodes. In the median cases, memory use increases linearly with the number of fog nodes in the topology, which is in line with the expected theoretical performance. However, the maximum distance has less of an effect than expected. This is due to the dynamic adjustment of the maximum distance for nodes in low density areas per Eq. 4.23. Remembering that for a maximum distance of 50, 118 fog nodes would be required, many fog nodes can not find any neighbours unless they use Eq. 4.23 to dynamically increase their maximum discovery distance. The results show that this mechanism works properly, increasing the maximum discovery distance significantly when required, to the point that fog nodes have a similar memory requirement for maximum distances of 50 and 100. This also explains why median memory use at these maximum distances barely rises with the number of fog nodes, since for a lot of nodes the dynamic increase in maximum distance will be lower, resulting in only slightly larger neighbourhoods on average. A maximum distance of 150 removes the effect of this mechanism entirely, causing a behavior similar to theoretical complexity.

The minimum cases all require around 14MiB memory, independent of the evaluation parameters. These are nodes in low density areas which have few neighbours even when the maximum distance is stretched. The maximum cases represent the nodes in the city center that have large amounts of neighbours regardless of maximum distance. Their memory use increases as expected with both maximum distance and number of fog nodes.

Overall, memory use is between 14MiB and 20Mib, which is low enough to deploy the fog service on low-power hardware with minimal resources.

The CPU use per node is shown in Fig. 4.18. Note that the results are represented in % of a single core. In the median case, CPU use is low, barely increasing with fog node density and increasing linearly with maximum distance. These results are better than expected from the theoretical performance, even in cases that are unaffected by dynamically adjusted maximum distance. The minimum and maximum cases indicate that overall performance can vary from about 50% to 500% of median performance. However, the extreme cases may be influenced by the evaluation itself, if the CPU use sample happens to coincide with a lot of process monitoring activity.

To give an indication of the spread of CPU use for nodes in different situations, the number of neighbours discovered by each node are monitored during the evaluation. For example, with 50 fog nodes and a maximum distance of 50, each node has 1 to 4 neighbours, with a median of 2. In

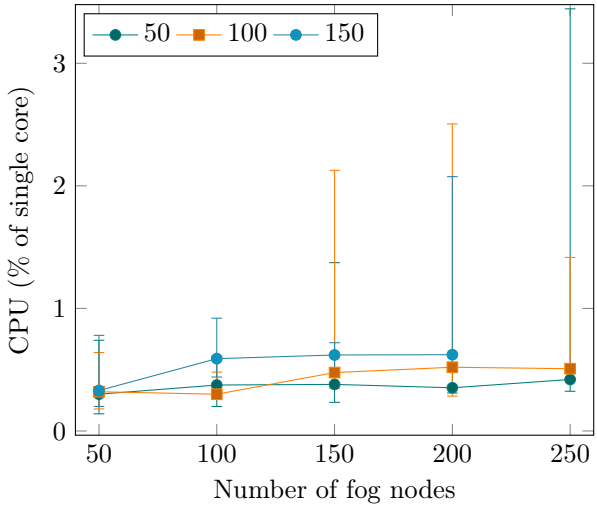


Figure 4.18: CPU load of the fog service for different amounts of fog nodes, and maximum discovery distances of 50, 100 and 150 units.

the case of 200 fog nodes and a maximum distance of 150, nodes have 1 to 84 neighbours, with a median of 38. Therefore, it is likely that the median cases in the chart also represent the median CPU use of nodes in each topology, and the most demanding nodes need about 200% of the CPU use of those cases.

In absolute numbers, the fog node service requires 0.3-0.6% of one CPU core. Although the evaluations are run on a relatively powerful CPU, the service should be able to run on low-power hardware with no performance problems.

Fig. 4.19 shows the network throughput per fog node related to neighbourhood discovery. These results are in line with the theoretical performance, although less pronounced than expected, with relatively small error margins caused by randomly generated topologies. Network throughput is less affected by dynamically adjusted maximum distance because the nodes with an increased maximum distance contact fewer other nodes overall. In turn, they have smaller lists of fog nodes known to them, causing both less and smaller responses during the discovery process. This means that while the memory use results show that the adjustment mechanism results in finding more neighbouring nodes, this does not necessarily result in higher network traffic for those nodes affected by it. In absolute numbers, network throughput is 30Kbps to 110Kbps.

In real life scenarios, the parameters of SoSwirly (e.g. maximum discovery

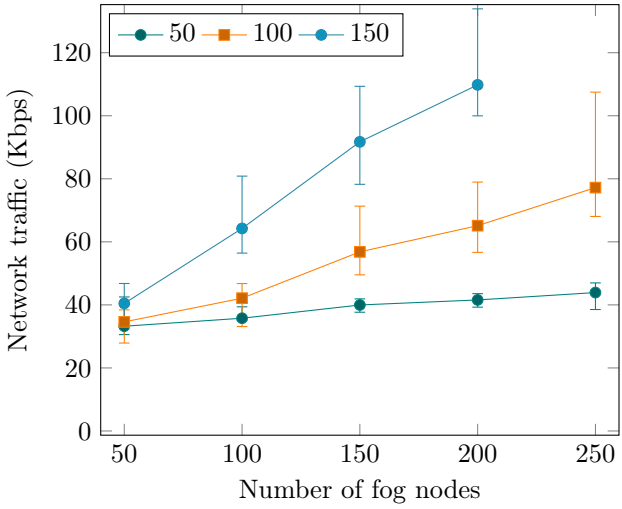


Figure 4.19: Network traffic at a single fog node for different amounts of fog nodes, and maximum discovery distances of 50, 100 and 150 units.

distance, expected fog node density) can be tuned so that the number of discovered neighbours is similar to that in the evaluations. Certainly, no more than 100 immediate neighbours should be tracked by any fog node. In combination with the results, this means that the fog service is capable of running on low-power devices with limited resources, and that it is highly scalable as long as node densities remain low enough.

For example, in Internet of Vehicles (IoV) applications using roadside units, the distance metric will likely involve vehicle velocity, and discovery rounds will be frequent due to rapidly changing relative distances. Because these roadside units are usually at the same distance from each other along well-defined trajectories and can be discovered in sequence, the maximum discovery distance for fog nodes (roadside units) can be lowered so each unit only discovers a few others in its neighbourhood. For fast moving edge nodes (cars), the maximum distance at which they can discover fog nodes can be increased to ensure they can reach sufficiently distant fog nodes in case a switch is required. This does not lead to performance problems, since the fog nodes themselves are sparsely connected.

The results indicate that, with proper tuning of parameters for a given node topology, the discovery algorithm behaves according to its theoretical performance, that it scales according to ***Req. 4***, and that the resource requirements are sufficiently low to satisfy ***Req. 5***.

Fig. 4.20 shows the accuracy of the neighbourhood discovery algorithm.

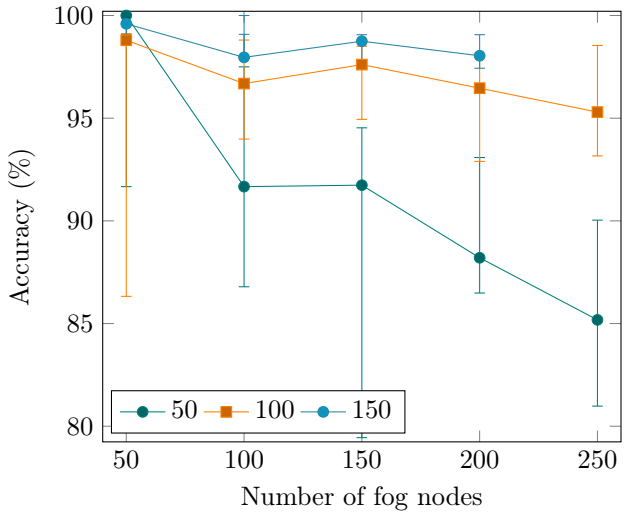


Figure 4.20: Accuracy of neighbourhood discovery for different amounts of fog nodes and maximum discovery distances of 50, 100 and 150 units. 100% means a fog node discovered all other fog nodes within the maximum distance.

This number represents how many nodes in its neighbourhood a fog node has actually discovered. The result represents a global number, indicating how many neighbours in total have been discovered compared to a perfect neighbour graph. A high accuracy is important, since it determines how well edge nodes can find an optimal service provider by traversing fog node neighbourhoods. Paradoxically, accuracy goes down as the number of fog nodes increases, especially with a maximum distance of 50 where not even the dynamic adjustment mechanism can compensate. However, this is largely up to the random generation of swirls, since fog nodes are sometimes placed too far away from others, start their discovery process with nodes on the other side of the topology, or may encounter adverse features as shown in Fig. 4.13. For minimum distances over 100, accuracy is high enough that edge nodes can find an optimal service provider in over 95% of the cases, even under these adverse conditions.

4.4.6.2 Service Deployment

Fig. 4.21 shows the network traffic and CPU use observed during the service deployment evaluation. Although both network traffic and CPU use rise about 10% as 50% more edge nodes are added, this is likely the result of more fog nodes being activated to service the additional edge nodes, which

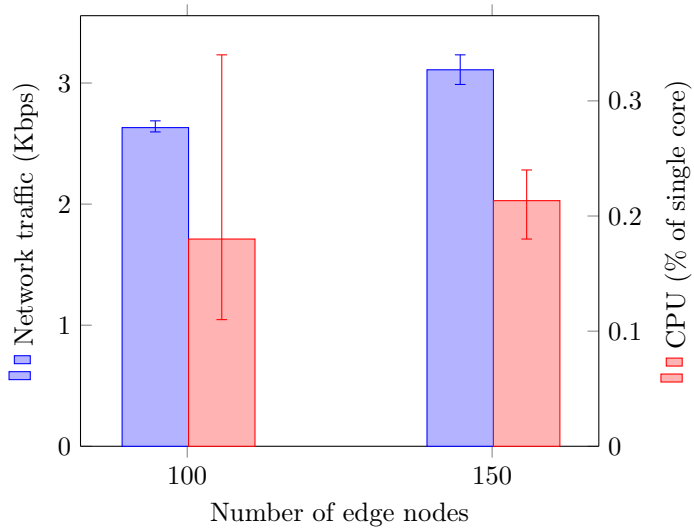


Figure 4.21: Average network traffic at edge nodes for service topologies with 100 fog nodes, and 100 or 150 edge nodes.

in turn leads to more active fog nodes being discovered and tracked by each edge node. In absolute terms, the edge node service causes only 3Kbps network traffic and requires 0.2% of a single CPU core. Even considering the powerful processors in the evaluation setup, this shows that it can run on low-power, low-resource edge hardware. Memory requirements are similar to those of the fog node service, between 14 and 16.5MiB depending on local fog node density. In swirls with 150 edge nodes, only 0.5% more memory is required in the median case than in those with 100 edge nodes, showing that the number of edge nodes in a swirl has no significant effect on the memory use of the edge node service.

These results indicate that service discovery and deployment by edge nodes scales according to ***Req. 4***, and that the resource requirements are low enough to meet ***Req. 5***.

4.4.6.3 Topology Efficiency

Fig. 4.22 shows how many fog nodes are activated by the evaluated algorithms in service topologies with either 100 or 150 edge nodes. SoSwirly is almost as efficient as Swirly, requiring 4% to 6% more nodes in the median cases. Considering that Swirly is a centralized approach, while in SoSwirly each node acts in its own best interests, a small efficiency penalty is to be expected. The results also show both Swirly and SoSwirly outperform-

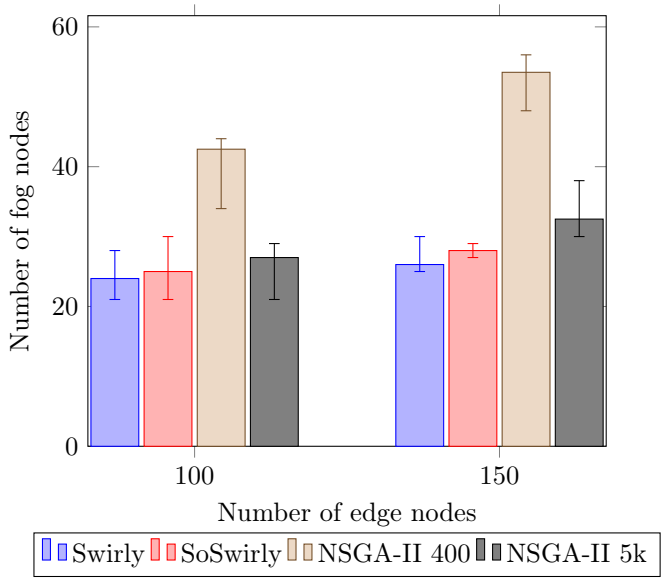


Figure 4.22: Number of activated fog nodes in service topologies with 100 or 150 edge nodes, comparing SoSwirly to centralized algorithms. NSGA-II is evaluated at 400 and 5000 (5k) generations.

ing NSGA-II. Even after 5000 generations, NSGA-II activates about 10% more nodes than SoSwirly, showing that SoSwirly can generate good service topologies with only its limited knowledge about neighborhoods.

The distance between edge nodes and their service providing fog nodes is shown in Fig. 4.23. In the solutions generated by SoSwirly, median distance is only 1-2% higher than in the solutions of the centralized Swirly, and both are well below the maximum metric distance of 100. However, the maximum distances in the solutions generated by SoSwirly are almost 300% higher than those in topologies generated by Swirly. Note that this mostly concerns edge nodes that are beyond the maximum distance of any fog node to begin with. These cases are few and far between, and the enormous distances between them and their assigned fog nodes are always caused by adverse features in the node topology, combined with worst-case starting nodes for neighbourhood discovery, both issues of randomly generated topologies that can be fixed in real-life scenarios. Compared to NSGA-II after 5000 generations, the median distance in topologies generated by Swirly is about 30% lower, showing an overall better optimization when it comes to latency. However, the maximum distances are about 43% higher than those generated by NSGA-II (546 vs 382), owing to a lack of global knowledge about the topology.

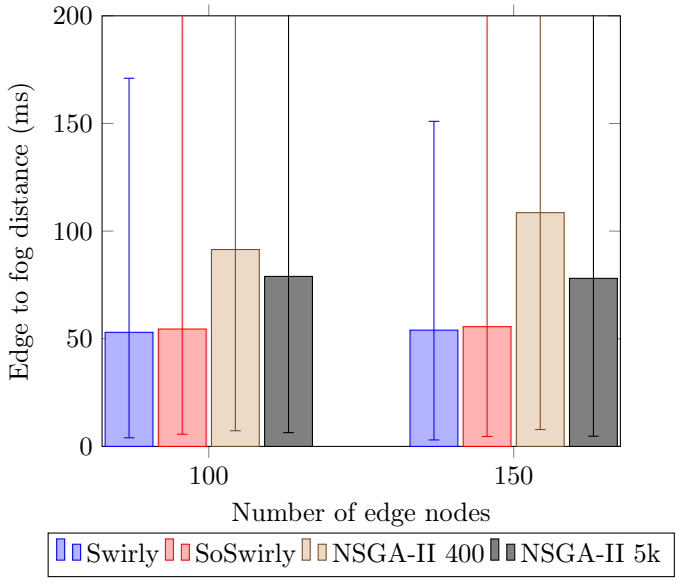


Figure 4.23: Distance of edge nodes to fog nodes in service topologies with 100 or 150 edge nodes, comparing SoSwirly to centralized algorithms. NSGA-II is evaluated at 400 and 5000 (5k) generations.

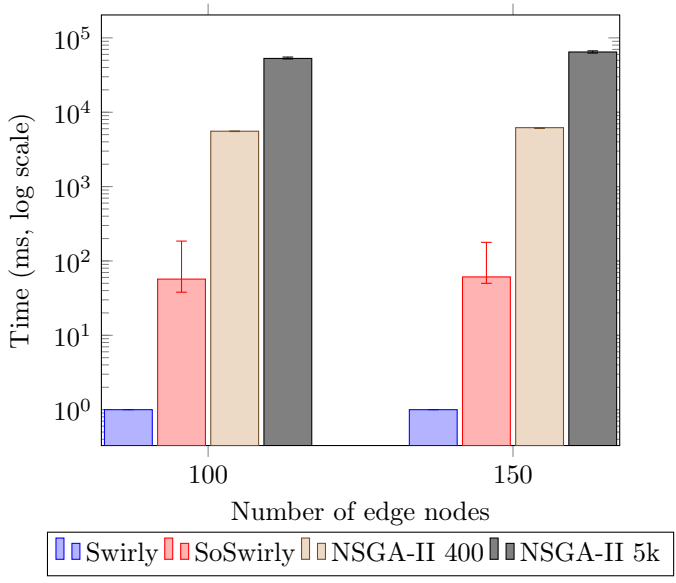


Figure 4.24: Processing time required to organize service topologies with 100 or 150 edge nodes, comparing SoSwirly to centralized algorithms. NSGA-II is evaluated at 400 and 5000 (5k) generations.

Finally, Fig. 4.24 shows the times required for each algorithm to determine their solutions. Note that this chart uses a logarithmic scale to accommodate NSGA-II, and that the time measured for Swirly is somewhat unreliable because it is very close to 0ms. Compared to Swirly, SoSwirly requires about 100 times more processing time to optimize the service topology. However, in the case of SoSwirly, the measured time necessarily includes gathering fog node information via web service calls, whereas for Swirly and NSGA-II the information is already present. Nevertheless, all nodes of SoSwirly require between 20ms and 180ms to find an optimal service provider. Depending on the number of generations used by NSGA-II, it is around 100 to 1000 times slower than SoSwirly. Considering the efficiency of NSGA-II, this shows that SoSwirly can work significantly faster and more efficient than NSGA-II in the evaluated scenario.

4.4.7 Discussion

In the introduction, the challenges of scheduling services with regards to the scope and hardware properties of the fog and edge are discussed, and a number of requirements for the presented solution are proposed.

SoSwirly, a fully decentralized fog service scheduler, is presented as a solution. A mathematical model of fog nodes and fog networks is presented, which forms the basis for an implementation of SoSwirly in Golang and an analysis of the fog nodes required for a specific swirl. It is explained how the proposed requirements are met by the design of SoSwirly, and its theoretical performance is explored.

To verify its performance, SoSwirly is experimentally evaluated in terms of CPU use, memory use and network traffic. The results show that in all cases, performance is equal to or better than the theoretical performance, confirming that SoSwirly is highly scalable in geographical terms, as long as the maximum discovery distance and node densities are balanced at every location. It is explained that in some cases the dynamic scaling of maximum discovery distance inflates the memory use and CPU use of nodes in topologies with a small default discovery distance, and that the overall resource requirements are low enough to be deployed on a wide range of devices.

Further evaluations show that the node discovery algorithm is accurate enough for edge nodes to find their optimal service provider by traversing fog node neighbourhoods. Additionally, the service topologies generated by SoSwirly are generally as efficient as those generated by the centralized Swirly. Additionally, it is shown that SoSwirly can generate service topologies much faster than a generic algorithm such as NSGA-II, and that the topologies are more efficient than those generated by NSGA-II.

Some topics for future work are discussed, for example more extensive monitoring of node availability, scaling to multiple service instances on fog nodes and allowing edge nodes to change service providers proactively.

While this section presents a fully operational, self-contained solution for self-organizing fog service scheduling, there are several possible improvements and additions to both the concept and implementation.

SoSwirly relies heavily on passive monitoring of nodes to solve problems. For example, if a fog node becomes unavailable, this will be detected during the next discovery round. However, until that happens any requests to services on that node may fail. Active monitoring of the services used by an edge node can reduce the time required to find a new service provider, improving QoS.

In areas with high edge node densities, it can be useful to scale services instances within small but powerful fog nodes, rather than scaling the number of fog nodes. This is not possible with the implementation of SoSwirly presented in this section, but alternative implementations of the *ServiceLocator* can provide this functionality in the edge node service without modifying the core algorithms and components. However, the fog node service would need additional per-service parameters and resource monitoring to automatically scale them as their load changes. Alternatively, a container orchestrator that supports service scaling by default can be supported by implementing *Orchestrator*, to be used on fog nodes where it is applicable.

In general, edge nodes calculate the distance to many fog nodes in their neighbourhood during each discovery round, but most of this information is discarded after the closest nodes are found. If the discovery rounds happen sufficiently frequently, the historical distance data may be used to determine if the edge node, or any fog node, is moving, and at which time it would be advantageous to proactively switch to another fog node. However, as seen in Section 4.4.2, the exact coordinates of a fog node can not be known, so any solution to this must rely on reported distances alone.

Edge nodes determine their optimal service provider through discovery rounds, during which the distance to each eligible fog node is updated. To support true real-time updates, fog nodes could send distance updates to the edge nodes they service whenever a significant change in distances occurs. This would result in more CPU use on fog nodes, but would also allow edge nodes to instantly switch to another service provider if their current one is experiencing technical issues or is moving too far away.

4.5 Summary

This chapter introduced the concept of scalability, and explained some general approaches to achieve scalable service architectures in the network edge. The potential of network federation to scale functionality and enable higher-level applications is illustrated through FUSE, which is designed to allow secure, on-demand federation of private networks. The main use case of FUSE is enabling rapid response in disaster/crisis situations, and it is shown that federations can be set up in a matter of minutes, and because of a mix of VPN and overlay networks, resource owners can determine exactly which resources should be available for use by the federation.
Continuing the concept of Swirly, this chapter presented SoSwirly as a completely decentralized service orchestrator which can organize large numbers of independent nodes into a coherent service architecture. Based on Kubernetes API's and FLEDGE, SoSwirly is shown to scale almost indefinitely under the right circumstances, which can be engineered by parameter tuning. Furthermore, the efficiency of the resulting topologies is almost equal to that of those generated by the centralized Swirly, despite the imperfect information possessed by each node, and SoSwirly can be used recursively, allowing the self-organization of entire service architectures in the edge.

References

[1] *Kubernetes - Building large clusters*, May 2019. https://kubernetes.io/docs/setup/cluster-large/.

[2] Kubernetes. *Kubernetes federation*, October 2018. https://kubernetes.io/docs/concepts/cluster-administration/federation/.

[3] Tom Goethals, Dwight Kerkhove, Laurens Van Hoye, Merlijn Sebrechts, Filip De Turck, and Bruno Volckaert. *FUSE: A Microservice Approach to Cross-domain Federation using Docker Containers.* In Proceedings of the 9th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, 2019.

[4] Tim Wauters, Brecht Vermeulen, Wim Vandenberghe, Piet Demeester, S Taylor, L Baron, M Smirnov, Y Al-Hazmi, A Willner, M Sawyer, D Margery, T Rakotoarivelo, F Lobillo Vilela, D Stavropoulos, C Papagianni, F Francois, C Bermudo, A Gavras, D Davies, J Lanza, and S-Y Park. *Federation of internet experimentation facilities: architecture and implementation.* In European Conference on Networks and Communications, Proceedings, pages 1–5, 2014.

[5] Rafael Moreno-Vozmediano, Eduardo Huedo, Ignacio M. Llorente, Rubén S. Montero, Philippe Massonet, Massimo Villari, Giovanni Merlino, Antonio Celesti, Anna Levin, Liran Schour, Constantino Vázquez, Jaime Melis, Stefan Spahr, and Darren Whigham. *BEACON: A Cloud Network Federation Framework.* In Antonio Celesti and Philipp Leitner, editors, Advances in Service-Oriented and Cloud Computing, pages 325–337, Cham, 2016. Springer International Publishing.

[6] Paolo Bottoni, Emanuele Gabrielli, Gabriele Gualandi, Luigi Vincenzo Mancini, and Franco Stolfi. *FedUp! Cloud Federation as a Service.* In Service-Oriented and Cloud Computing, pages 168–182. Springer International Publishing, 2016.

[7] B Vermeulen, W Van de Meerssche, and T Walcarius. *jFed toolkit, Fed4Fire.* In Federation. In: GENI Engineering Conference, volume 19, 2014.

[8] David Recordon and Drummond Reed. *OpenID 2.0: A Platform for User-centric Identity Management.* In Proceedings of the Second ACM Workshop on Digital Identity Management, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[9] Enias Cailliau, Nick Aerts, Lander Noterman, and Lars De Groote. *A comparative study on containers and related technologies.* 11 2016.

[10] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. *An updated performance comparison of virtual machines and Linux containers.* In 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, mar 2015.

[11] David Bernstein. *Containers and Cloud: From LXC to Docker to Kubernetes.* IEEE Cloud Computing, 1(3):81–84, sep 2014.

[12] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. *Large-scale cluster management at Google with Borg.* Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015, 04 2015.

[13] Emiliano Casalicchio. *Autonomic Orchestration of Containers: Problem Definition and Research Challenges.* In Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools. ACM, 2017.

[14] Nane Kratzke. *About Microservices, Containers and their Underestimated Impact on Network Performance.* 2017.

[15] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. *Performance Evaluation of Microservices Architectures Using Containers.* In 2015 IEEE 14th International Symposium on Network Computing and Applications, pages 27–34, Sep. 2015.

[16] Thanh Bui. *Analysis of Docker Security.* 2015.

[17] Kubernetes. *Security Best Practices for Kubernetes Deployment*, August 2016. https://kubernetes.io/blog/2016/08/security-best-practices-kubernetes-deployment/.

[18] Weisong Shi and Schahram Dustdar. *The Promise of Edge Computing.* Computer, 49(5):78–81, may 2016.

[19] Farzad Samie, Vasileios Tsoutsouras, Lars Bauer, Sotirios Xydis, Dimitrios Soudris, and Jorg Henkel. *Computation offloading and resource allocation for low-power IoT edge devices.* In 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT). IEEE, dec 2016.

[20] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. *Consolidate IoT Edge Computing with Lightweight Virtualization.* IEEE Network, 32(1):102–111, jan 2018.

[21] Brendan Jennings and Rolf Stadler. *Resource Management in Clouds: Survey and Research Challenges.* Journal of Network and Systems Management, 23(3):567–619, Jul 2015.

[22] Ankita Atrey, Hendrik Moens, Gregory Van Seghbroeck, Bruno Volckaert, and Filip De Turck. *Design and Evaluation of Automatic Workflow Scaling Algorithms for Multi-tenant SaaS.* In Proceedings of the 6th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and and Technology Publications, 2016.

[23] Md Whaiduzzaman, Mohammad Haque, Rejaul Karim Chowdhury, and Abdullah Gani. *A Study on Strategic Provisioning of Cloud Computing Services.* The Scientific World Journal, 2014, 06 2014.

[24] Zhanjie Wang and Xianxian Su. *Dynamically Hierarchical Resource-allocation Algorithm in Cloud Computing Environment.* J. Supercomput., 71(7):2748–2766, July 2015.

[25] Seyed Ali Miraftabzadeh, Paul Rad, and Mo Jamshidi. *Distributed Algorithm with Inherent Intelligence for Multi-cloud Resource Provisioning.* In Intelligent Decision Support Systems for Sustainable Computing, pages 77–99. Springer International Publishing, 2017.

[26] Brendan Burns. *Designing Distributed Systems (The sidecar pattern)*, February 2018. https://www.oreilly.com/library/view/ designing-distributed-systems/9781491983638/ch02.html.

[27] OpenVPN. *OpenVPN Optimizing performance on gigabit networks*, 2018. https://community.openvpn.net/openvpn/wiki/Gigabit_Networks_Linux.

[28] Intel. *Intel Data Protection Technology with AES-NI and Secure Key*, 2018. https://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard–aes-/data-protection-aes-general-technology.html.

[29] Tom Goethals, Filip De Turck, and Bruno Volckaert. *Self-organizing Fog Support Services for Responsive Edge Computing.* Journal of Network and Systems Management, 29(2), jan 2021.

[30] Pieter-Jan Maenhaut, Bruno Volckaert, Veerle Ongenae, and Filip De Turck. *Resource Management in a Containerized Cloud: Status and Challenges.* Journal of Network and Systems Management, 28(2):197–246, nov 2019.

[31] Cosmin Avasalcai, Ilir Murturi, and Schahram Dustdar. *Edge and Fog: A Survey, Use Cases, and Future Challenges*, apr 2020.

[32] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. *Fog Function: Serverless Fog Computing for Data Intensive IoT Services.* In 2019 IEEE International Conference on Services Computing (SCC). IEEE, jul 2019.

[33] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. *Challenges and Opportunities for Efficient Serverless Computing at the Edge.* In 2019 38th Symposium on Reliable Distributed Systems (SRDS). IEEE, oct 2019.

[34] Walter do Espirito Santo, Rubens de Souza Matos Junior, Admilson de Ribamar Lima Ribeiro, Danilo Souza Silva, and Reneilson Santos. *Systematic Mapping on Orchestration of Container-based Applications in Fog Computing.* In 2019 15th International Conference on Network and Service Management (CNSM). IEEE, oct 2019.

[35] Anil Madhavapeddy and David J. Scott. *Unikernels.* Communications of the ACM, 57(1):61–69, jan 2014.

[36] Carlos Guerrero, Isaac Lera, and Carlos Juiz. *Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures.* Future Generation Computer Systems, 97:131–144, aug 2019.

[37] Mehdi Hosseinzadeh, Hawkar Kamaran Hama, Marwan Yassin Ghafour, Mohammad Masdari, Omed Hassan Ahmed, and Hemn Khezri. *Service Selection Using Multi-criteria Decision Making: A Comprehensive Overview.* Journal of Network and Systems Management, 28(4):1639–1693, jul 2020.

[38] Bruno Stévant, Jean-Louis Pazat, and Alberto Blanc. *QoS-aware Autonomic Adaptation of Microservices Placement on Edge Devices.* In Proceedings of the 10th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, 2020.

[39] *What is Kubernetes?*, April 2019. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[40] Oskar Casquero, Aintzane Armentia, Isabel Sarachaga, Federico Perez, Dario Orive, and Marga Marcos. *Distributed scheduling in Kubernetes based on MAS for Fog-in-the-loop applications.* In 2019 24th IEEE

International Conference on Emerging Technologies and Factory Automation (ETFA). IEEE, sep 2019.

[41] Sandrine Vaton, Olivier Brun, Maxime Mouchet, Pablo Belzarena, Isabel Amigo, Balakrishna J. Prabhu, and Thierry Chonavel. *Joint Minimization of Monitoring Cost and Delay in Overlay Networks: Optimal Policies with a Markovian Approach*. Journal of Network and Systems Management, 27(1):188–232, aug 2018.

[42] Jose 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, dec 2017.

[43] Paul Dawkins. *Paul's Online Notes - Section 4-8 : Change Of Variables*, June 2020. https://tutorial.math.lamar.edu/Classes/CalcIII/ChangeOfVariables.aspx.

[44] Simcha Waldman. *Jacobian matrix*, June 2020. https://math.wikia.org/wiki/Jacobian_matrix.

[45] Bernard F. Schutz. *Geometrical Methods of Mathematical Physics*. Cambridge University Press, jan 1980.

[46] *Rancher Labs - K3S Lightweight Kubernetes*. https://k3s.io/.

[47] Kubernetes. *Kubernetes API - PodSpec*, April 2020. https://godoc.org/k8s.io/api/core/v1#PodSpec.

[48] Linux. *Hosts - Static table lookup for hostnames*, April 2020. https://www.man7.org/linux/man-pages/man5/hosts.5.html.

[49] IDlab. *Virtual Wall*, April 2020. https://www.ugent.be/ea/idlab/en/research/research-infrastructure/virtual-wall.htm.

[50] IETF. *The GeoJSON Format*, March 2020. https://tools.ietf.org/html/rfc7946.

[51] StatBel. *The Belgian statistical office*, February 2020. https://statbel.fgov.be/en.

[52] StatBel. *Statistical Sectors*, February 2020. https://statbel.fgov.be/en/open-data/statistical-sectors.

[53] StatBel. *Population by Statistical Sector*, February 2020. https://statbel.fgov.be/en/open-data?category=209.

[54] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. *A fast and elitist multiobjective genetic algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation, 6(2):182–197, apr 2002.

5

Intelligence in Smart Cities

There was an AI made of dust, whose poetry gained it man's trust
If is follows ought [1], it'll do what they thought
In the end we all do what we must.
- Universal Paperclips, Frank Lantz

5.1 Introduction

The previous chapters have introduced the required technologies to enable and scale complex service architectures in the large edge networks one might find in a Smart City. This chapter illustrates how Artificial Intelligence (AI) can be deployed in the edge to further improve these technologies, and how they can enable AI applications in Smart Cities.

Various reasons for the migration of computational workloads to the network edge have been discussed in Chapters 2 through 4, which are summarized here for readability. First of all, computational migration to the edge is a necessary step in the emergence of various "Smart" application domains, and eventually Smart Cities, in which AI is to be deployed exactly where and when it is required, allowing it to process data locally and efficiently without loss of privacy.

To reiterate, there are several advantages to edge offloading. For example, to run software services closer to end-users in order to reduce latency, or to pre-process data instead of gathering all data to the cloud, thereby avoiding

bandwidth issues or undue pressure on cloud resources. Additionally, the number of devices in fog and edge networks increases at an accelerated pace, while the hardware resources of the average device keep increasing. As such, there is ever more task offloading capacity available in the fog and in the edge.

On the other hand, there are also disadvantages to offloading tasks to geographically widespread fog and edge networks. In cloud data centers, hardware resources and network technologies are homogeneous, and properly managed using planned upgrades, typically resulting in high availability of services and systems. In the edge, networks are heterogeneous and unpredictable, and hardware resources and capabilities are extremely varied. As the scale of edge networks and the variety of devices they comprise increases, these factors make it increasingly more difficult to manage software services and organize traffic flows. For example, gathering all the required data for service orchestration in the cloud becomes infeasible due to network bandwidth saturation and memory requirements.

Additionally, tasks such as data placement and service migration are more difficult to orchestrate in edge networks than they are in the cloud. In cloud data centers, or a limited number of fog data centers, the target nodes for service deployments can be optimally calculated, and migrations can be executed quickly over high bandwidth connections. The network edge, however, is a volatile environment with a continually changing topology. In such an environment, calculating the optimal nodes to deploy data or services on is nearly impossible, and limited network bandwidth reduces the potential for service migration.

Finally, there are also various security risks that present themselves when running software services in the fog or edge. As opposed to the strictly controlled environment of a cloud data center, edge networks are largely comprised of unknown devices in networks with unknown, and often insufficient security measures. Such environments make it difficult to detect issues such as unauthorized access, data loss, privacy infringement and malicious injections of data or code, and nearly impossible to avoid them.

AI can solve many of these issues. For example, some classes of AI algorithms can learn from data gathered in the cloud and from the edge in order to recognize network intrusions, route traffic around faulty nodes, or quickly determine suitable nodes to deploy software and data on. However, AI algorithms can be resource intensive, and edge devices are often resource constrained and low-powered. Until recently, most edge devices were incapable of running any containerized services or advanced AI applications. Advances in both software and hardware, specifically related to Artificial Neural Networks (ANN), have commoditized AI in the edge. Although

many devices have different priorities, e.g. extremely low-power sensors, all data is usually gathered at local gateway devices in the edge, or edge servers, which have the appropriate hardware resources to run complex AI algorithms. These advances have enabled AI to play an increasingly important role in properly organizing the network edge, orchestrating software services in the edge, and in software services themselves, which use AI to optimize end-user experience. Edge Intelligence (EI) [2] arises from any use of AI to enhance the organization or operation of software services in the edge, while the whole of EI and the AI-powered end-user applications it enables results in the Intelligent Edge.
In Section 5.2, several important types of AI for the edge are explained, along with the concept of edge computing itself. The synthesis of edge computing and AI is discussed in 5.3 to show how the Intelligent Edge emerges from it, and what the general areas of end-user AI-powered applications in Smart Cities are. Section 5.4 illustrates how SoSwirly with FLEDGE can enable decentralized intelligence in edge networks, by both acting as a decentralized weight distributor, and by integrating an AI component to better organize itself. The state of the art of EI is presented in Section 5.5, starting with a taxonomy of EI, followed by a discussion of each category by charting research trends, and by presenting a selection of recent studies. The main topics are **enabling technologies** for AI in the edge, AI approaches to **organize** various aspects of edge networks, and finally AI-assisted **applications** running in edge networks. From the presented studies, future challenges for each topic are drawn, along with some long-term visions for the use of AI in the edge. Finally, the chapter is concluded in Section 5.6.

5.2 Distributed Intelligence technologies

In this section, the most common types of AI in Edge Intelligence are introduced. Although sufficient explanation is given for the purposes of this chapter, the goal is only to introduce each of the topics, with more comprehensive works included as references. There are numerous studies and books that explain the general principles of AI, for example Hunt [3] or Brewka [4].

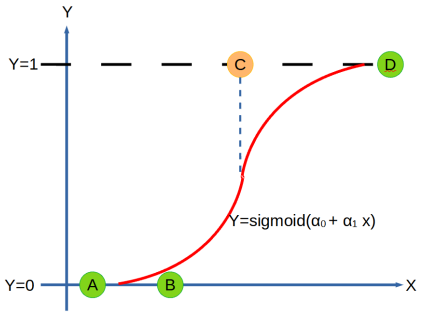
5.2.1 Statistical algorithms

Statistical approaches can be used to solve (binary) classification problems. The most popular algorithm of this type is logistic regression [5], a special case of binary regression which results in binary classifiers that output

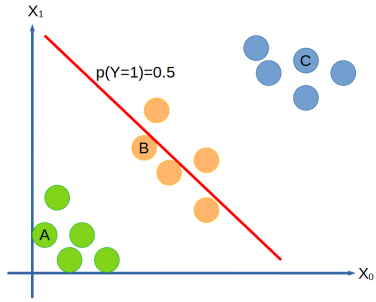
probabilities rather than a hard classification. This algorithm uses supervised learning [6], a method which "trains" a model on an initial data set containing expert-labeled outputs for known sets of inputs. After training, the statistical patterns in the data learned by the model are used to predict the probability of new inputs belonging to either class.
Assume that for an input with values x_i the output Y is required, with $Y = 0$ meaning that the input belongs to class A and for $Y = 1$ it belongs to class B. In logistic regression, the log-odds of an input is calculated using a linear combination of its values:

$$\log \frac{p}{1-p} = \alpha_0 + \sum \alpha_i x_i \tag{5.1}$$

with α_0 and α_i being learned parameters. To recover the probability p from this, a Sigmoid function is applied to the right side of the equation, giving the probability that the output Y belongs to class B, or $p(Y = 1)$. Generally, $p < 0.5$ means the input is likely to belong to class A, and for $p > 0.5$ to class B.



(a) Logistic regression for one input dimension (X). The points A, B and D can be reliably classified, but the model will give unreliable output for C, which has $p(Y = 1)_C = 0.5$ and may belong to either class.



(b) Logistic regression for two input dimensions. The red line describes the "border" between the output classes, or $p(Y = 1) = 0.5$. The points A and C can be reliably classified, but again the model will give unreliable output for points such as B, which have high probability of belonging to either class.

Figure 5.1: Classification in 1 and 2 dimensions using logistic regression.

During training, the difference between $p(Y = 1)$ and the expert-labeled output is used to adjust the parameters α, improving the model. This is achieved through gradient descent [7], which calculates the impact of each input value x_i on the final output, and adjusts its parameter α_i to better match the output that is required. Generally, a learning rate $l << 1$ is used to modify the weights only slightly for each input, to avoid undoing the

effects of previous adjustments. The model resulting from this training process is visualized in Fig. 5.1. Fig. 5.1a shows how the model discriminates between points in one dimension. The red curve represents a model trained on the input dimension (X), giving the probability of a point belonging to $Y = 1$. The points A and D are classified with absolute certainty, and B almost certainly belongs to $Y = 0$, as $p(Y = 1)_B$ is only 0.05. However, the model has difficulty classifying points such as C, which may belong to either class as $p(Y = 1)_C$ is 0.5. Similarly, Fig. 5.1b shows the classification of points with two input dimensions. In this figure, the red line represents the "border" of the two classes as determined by the model, or $p(Y = 1) = 0.5$. Points A and C can be classified with high certainty, but again there are points such as B which could belong to either class. Such data points may exist for any trained model; no training set can contain all possible data points as that would defeat the purpose of training a model.
However, this approach means that the accuracy of the final model generally increases with the amount of training data, as long as the inputs and outputs are properly distributed over all possible values. A downside of logistic regression is that the algorithm can get stuck in a local optimum or oscillate between several local optima, depending on initial parameters and the available training data.
Although logistic regression can be applied to any number of inputs, it can only discriminate between two output classes, limiting its usefulness. However, it is often used as a base model in more complex systems.
Some problems can be modeled as a Markov Decision Process (MDP) [8], a discrete-time stochastic process model. Such a process defines a state space S, an action space A, and a probability function $P_a(s_\alpha, s_\beta, t)$ which describes the likelihood of transitioning from state s_α to state s_β through action a at timestep t. A reward function $R_a(s_\alpha, s_\beta, t)$ provides the relevant reward for any state transition. Using the reward function, Reinforcement Learning (RL) [9], which is further explained in Section 5.2.3, can be used to learn the optimal action policy for an MDP. This policy, once learned, decides which action to take in any state, reducing $P_a(s_\alpha, s_\beta, t)$ to a straightforward probabilistic state transition $P(s_\alpha, s_\beta)$.

5.2.2 Evolutionary algorithms

Evolutionary or genetic algorithms [10] are modeled after the process of evolution in biological organisms, and can be applied to a wide range of problems. Technically, they can solve any problem that can be represented using a fitness function [11], whose minimum value over a search space should be minimized. This basic property makes them well-suited for scheduling problems and organizational problems.

An evolutionary algorithm starts by randomly generating n genomes (potential solutions), each of which contains all the values necessary to construct a concrete solution to the problem at hand. Each genome can be evaluated by the fitness function, thus ranking them by effectiveness. The algorithm then runs for a predetermined number of epochs (iterations), with two actions being performed in each epoch. First, n new genomes are generated by combining the values of parent genomes from the previous epoch, taking into account restrictions on the search problem. The chance of a genome being selected is proportional to its fitness value. In the second step, the new genomes are mutated by randomly changing values in order to introduce randomness in the search process. After the last epoch, the genome with the best fitness value is selected as the solution.

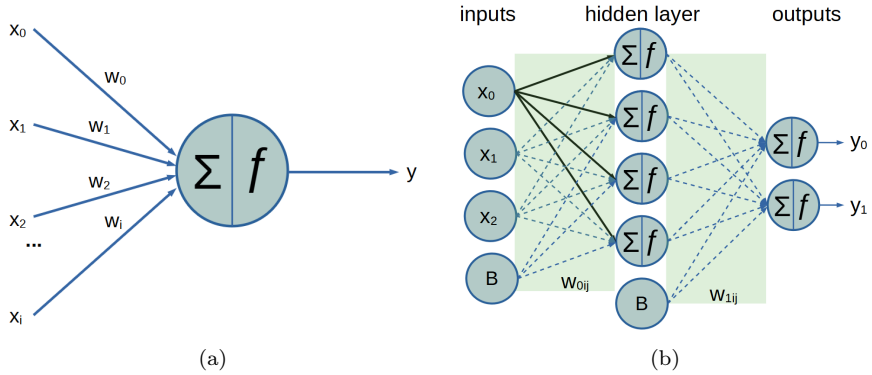
This type of algorithm relies on examining many potential solutions simultaneously and introducing randomness in the search process to cover as much of the search space as possible, while using a fitness function to guide the process in the direction of optimal solutions. However, the solution is not guaranteed to be optimal, and the algorithm may need to run for an undetermined amount of time before arriving at an acceptable solution, while the end result may not be explainable through math or logic.

Multi-objective optimization algorithms [12] such as MOGA [13] and NSGA-II [14] are a popular subset of evolutionary algorithms in the fog and edge. These algorithms find Pareto optimal solutions [15] for multiple optimization parameters by encoding data points, parameters and restrictions in genomes. As an example, consider finding the optimal computational nodes to deploy a number of software services on, depending on end-user latency and available resources. A multi-objective optimization algorithm will integrate the relevant properties of services and nodes into the genome, and both latency and available resources will be combined in a fitness function resembling Pareto search. The output is a genome that encodes the optimal node for each service to deploy.

5.2.3 Artificial Neural Networks

Like evolutionary algorithms, ANNs [9] are biologically inspired, simulating computation as it occurs in biological brains. The base element of an ANN is the neuron, which in its most basic form is described by Eq. 5.2. It accepts a number of inputs x_i, weighted by factors w_i. The sum of these values is used as input for the activation function f, the result of which is the output y of the neuron:

$$y = f(\sum x_i * w_i) \tag{5.2}$$



(a) (b)

Figure 5.2: Visual representations of (a) a single neuron (perceptron) and (b) layered neural network (Multi-Layer Perceptron, MLP).

This equation is visualized in Fig. 5.2a. Geometrically, a neuron represents a hyperplane dividing an i-dimensional space, which can be interpreted as performing binary classification of the points in the space. There are many variations on the basic neuron, with binary or floating point input/output values, a great variety of activation functions [16, 17], and the option of adding a "bias" value; a static input that is always active. For example, a single neuron can perform logistic regression by choosing a Sigmoid activation function and including a bias with weight α_0, resulting in 5.1.

ANNs learn patterns in a data set by using a backpropagation algorithm (gradient descent) [9] to modify their weights, similar to how parameters are updated in logistic regression. This can be done with either supervised learning (pre-labeled data), Reinforcement Learning (RL) (reward function, automated feedback) or unsupervised learning, although unsupervised learning does not necessarily use backpropagation. However, backpropagation can be computationally intensive depending on the choice of activation function, as the complexity of the loss function involved in gradient descent depends largely on the activation function used.

In the case of supervised learning [6], a training data set is used with inputs and expert-labeled outputs. The output of the ANN for a given input is compared to the expert-labeled output, and the difference is used to update the weights. This type of learning is usually reserved for classification.

Both unsupervised learning and Reinforcement Learning [18] are used for tasks where labelling outputs is infeasible, either due to the volume of data involved or because the correct output is not known. While unsupervised learning is completely unguided with respect to its outputs, RL uses a reward function which returns higher values for "more correct" outputs, and

a modified backpropagation algorithm is used.

Although a single neuron (or perceptron) can emulate basic algorithms such as logistic regression, multiple neurons can be combined into neural networks (or Multi-Layer Perceptron, MLP) to solve a wide range of problems. An example of a basic neural network is shown in Fig. 5.2b, with neurons organized into several layers, each processing the outputs of the previous layers using a weights tensor w_{kij}, between neuron i in layer k and neuron j in layer $k+1$. In this figure, the middle layer is a "hidden layer", only used for computation rather than writing input values x_i or reading output values y_i. The concept of bias neurons B is also illustrated here. Constructing neural networks in layers allows each layer to process progressively more complex features in the input data, with the final layer being able to classify intricate, abstract shapes or patterns. Note that the example model is fully connected, with each neuron in any layer being connected to each neuron in the previous one, but in practical applications this is rarely the case.

Stacking layers of neurons results in a more complex, recursive learning process. Neural networks and backpropagation require a lot of parameters to work correctly, such as initial input weights and learning rates. Sub-optimal choices often result in the failure to train a network, and as such many studies have focused on choosing correct initialization values for these parameters, and if and how they should be modified throughout the training process [19]. Regularization and specialized activation functions are also used to reduce model size and improve the learning process [20, 21].

The goal of gradient descent can be interpreted geometrically as finding the lowest point in the hyperplane formed by the loss function used during backpropagation. By itself, a static learning rate results in only a minor improvement in a specific direction for each training input, which may not be entirely in line with the true gradient of the hyperplane. Alternatives include using decaying momentum [22] to guide the backpropagation algorithm into a general direction over multiple training inputs. Other methods, using second-order derivatives of the loss function, are more computationally intensive and not always applicable, but produce excellent results with less training data [23]. Finally, training samples are often processed in batches to optimize both performance and training results.

In the last decade, hardware acceleration and architectural improvements [24] have made it possible to create and train neural networks with dozens or even hundreds of layers, now known as Deep Neural Networks (DNN) [9]. Combined with other advances, this has led to many specialized, highly efficient innovations. For example, Convolutional Neural Networks (CNN) [25] contain layers that have a similar function to image kernels, and are currently the most effective classification networks for visual input. Recurrent

Neural Networks (RNN) [26], in which certain layers feed their outputs back into their own inputs, can use memory strategies such as Long Short Term Storage (LSTM) [27] or Gated Recurrent Units (GRU) for natural language processing or translation, or other types of tasks with tokenized, unbounded input where the state depends on previous inputs. In terms of training, Deep Reinforcement Learning (DRL) [28] has simplified training of deep networks, and advanced RL approaches such as Q-learning [9] can take into account expected and future rewards, rather than immediate returns from a reward function.

5.2.4 Distributed algorithms

Distributed and decentralized algorithms are designed to run on a large number of computational nodes simultaneously. Whereas distributed algorithms generally still have a centralized goal and combine their outputs, decentralized algorithms simply divide a problem into small, independent parts without the need to merge the outputs. Distribution and decentralization are applicable to a wide range of algorithms, although some types of problems are easier to partition, such as ANN training. Distributed algorithms have two important advantages compared to monolithic, centralized algorithms. First, the complexity of a problem can often be greatly reduced by splitting it up into smaller tasks. Second, a distributed algorithm is generally far more scalable, e.g. grid computing projects such as Boinc [29]. A popular distributed learning algorithm in the fog and edge is Federated Learning (FL) [30], in which the training of a neural network is split up into parts. This may be a straightforward division of labor, or it can be organized hierarchically. After each node finishes its part of the training, the resulting model updates are integrated into a centralized model, usually in the cloud. The main advantage of FL is that it can offload model training from the cloud to fog and edge devices depending on the computational capacity of each node. A further advantage is the reduction of network traffic by processing training data at the network edge, and that local processing of training data can avoid privacy issues related to sending data to the cloud. However, depending on the model involved, training may be unfeasible on resource-limited edge hardware, and a long-term disadvantage is that FL has to update a centralized model and distribute it to fog and edge nodes from the cloud. More advanced approaches try to eliminate the cloud part, and fully decentralize the weight updates through peer-to-peer updates. Hierarchical Federated Learning (HFL) solves some of the communications issues of vanilla FL by introducing a hierarchical structure into the process of consolidating weight updates, usually through a middle layer in which cluster heads perform intermediate model integration.

Swarm Intelligence (SI) [31] is a general class of distributed algorithms in which large numbers of independently functioning nodes or particles perform localized improvements, in an attempt to achieve a globally optimal solution through emergent behavior. The logic of this approach is that on average, an improvement for any node is also a global improvement, and the optimal solution can be reached by letting each node find its own optimal state. Although this approach can result in acceptable solutions, greedy individual behavior and a lack of global coordination does not often result in a theoretically optimal solution. Careful algorithm design is paramount, and in order to increase the chances of an acceptable global solution, the basic function to be optimized by each agent should be kept as straightforward as possible, limiting unexpected adverse behavior. SI is usually applied to problems that are easy to handle for a single node, but which are intractable on a larger scale. Particle Swarm Optimization (PSO) is a subclass of SI, but it usually simulates all particles and generally does not run as a distributed algorithm. PSO finds optimal solutions in a search space by simulating the movement of large numbers of particles, gravitating them towards each other as they find optimal states in the search space.

5.2.5 Blockchain

Blockchains are relatively new, originally introduced as the technology behind various digital currencies, but increasingly popular in research for their potential as secure, distributed storage. While a blockchain is not an AI concept in itself, it is interesting to introduce it here because of its popularity in AI-related studies. Although variations exist, blockchains in general have interesting properties, but also significant challenges for their widespread adoption [32].

Generally, blockchains are transaction-based, and they operate through a number of decentralized, non-hierarchical nodes known as miners. Each node in the network has a copy of the blockchain, a collection of "blocks", each of which in turn contains a number of transactions. Whenever a node in the blockchain network creates a transaction, it is spread throughout the network on a peer-to-peer basis, and processed by the miners into a new block at the end of the chain. For the blockchain to be reliable, all miners must reach a consensus on the transactions processed, and the order blocks are processed in. However, this process is quite computationally intensive, and a (financial) reward for miners is usually attached in the form of digital currency, either through the process of mining itself or by demanding a transaction cost. This digital currency is tracked by the blockchain itself, avoiding fraud and contested transactions.

The most popular alternative currently used in studies is Ethereum [33],

an open source blockchain using Ether as currency. Ethereum implements smart contracts, allowing code to be embedded into transactions, and enabling its execution whenever the requirements are met. Due to the nature of the blockchain, all parties agree by definition on the contents and execution of the smart contracts.
Although the decentralization of blockchain solutions offers some intrinsic security and reliability, and smart contracts are a flexible and reliable approach to digital transactions, there are also some challenges to widespread adoption these technologies. Most importantly, the energy use of blockchain solutions is generally known to be excessively high, although various solutions have been presented to alleviate this issue, for example Proof-of-Stake consensus. However, the current state of the art still requires orders of magnitude more energy per transaction than classical systems [34]. Because of its distributed, peer-to-peer nature, it also takes far longer for transactions to be processed by a blockchain solution than by a classical, centralized system. Whereas a single node can process a transaction in just a few milliseconds, the need for a network-wide consensus can increase the total transaction processing time to minutes. Some blockchain implementations are susceptible to manipulation if any single party controls over 50% of the mining capacity, giving that party a monopoly on the consensus mechanism. This risk can be mitigated with both technical and practical measures. Finally, the distributed and open nature of the blockchain means that anyone can view its contents. Although they can not be changed, the plain readability of transactions presents severe privacy issues. As such, extra security measures will be needed for most concrete blockchain solutions, or off-chain storage solutions may be needed to augment the blockchain.

5.2.6 Other

AI is not limited to the types previously listed in this section. It can take many forms, especially when applied in a new environment such as edge computing. For the purposes of this chapter, any method or algorithm is considered a form of AI as long as the base problem is intractable, the algorithm runs in the fog or edge, and predictive outputs are generated based on any number of input dimensions. Note that this does not necessarily mean that the algorithm has learning capabilities.

5.3 Intelligent Edge

As the scale of fog and edge networks grows, they eventually contain so many computational nodes that classical, centralized algorithms cannot scale suf-

ficiently to manage them. In networks containing millions of nodes, it is impossible to gather network information and changing node statuses in real-time to a single location, nor is it feasible for a single algorithm instance to orchestrate services, detect malicious traffic, and route traffic within an acceptable time frame.
Even in applications where timing is not an issue, the scale of any problem combined with the computational complexity of any classical, cloud-based algorithm will quickly overwhelm the hardware resources of a single cloud node, or even a few cloud nodes. This problem of scalability can be solved by decentralizing such algorithms and deploying them in the edge, and by integrating AI into them. As discussed in Section 5.2, some types of AI algorithms have a training phase, and as such they can determine the important parameters for a problem during the training phase and produce results quickly at inference. Furthermore, AI algorithms can be designed to either send data to the cloud for use in further training, or even to keep executing training rounds themselves using gathered data, and merging the resulting weight updates through federated training. In all cases, AI algorithms can keep improving their efficiency.
Finally, neural networks are very computationally intensive, but using multiple, specialized layers they can discover complex, non-linear relations between parameters that classical algorithms would not be programmed to take into account.
Apart from decentralizing cloud algorithms and imbuing them with AI, there are also cases where processing data locally is the most logical choice. Reasons for this may include minimizing end-user latency, providing functionality even when connections to the cloud fail, privacy issues with sending data to the cloud (e.g. GDPR [35]), or various other legal requirements or user preferences. As such, it is unavoidable to increasingly use decentralized intelligent algorithms to manage any and all aspects of organization and orchestration in the edge. Combining this infrastructural intelligence with AI applications featuring direct user interaction results in the Intelligent Edge, opening up the way to concepts such as Smart Cities [36]. Smart Cities have various application areas where AI can be useful. There are general Smart City applications, as well as Smart Homes, Industry 4.0, Internet of Vehicles (IoV) and Smart Health Care. Each of these domains will be further explained in Section 5.5.2.1.

5.3.1 Standards

Several recent IEEE standards and active projects focus on various aspects of EI, or can be taken into account when creating EI solutions. For example, IEEE 1934-2018 [37] adopts the OpenFog architecture as a standard,

providing a framework for distributed computing, control and networking functions in an IoT environment on which EI can be built. Sub-projects of P2805 aim to establish intelligent protocols for self-managing edge computing nodes [38] and cloud-edge collaboration for machine learning [39], while P2961 [40] is to provide a framework for distributed, collaborative machine learning in an edge-cloud environment. Finally, there are also projects explicitly aimed at Smart Cities applications, e.g. P2979 [41] which aims to provide a framework for intelligent cooperation of edge devices in various IoV use cases.

5.4 Decentralized intelligence

This section discusses how AI and SoSwirly can be combined for smooth, decentralized weight updates, and pro-active service orchestration. Although these ideas are not fully implemented and have been neither evaluated, nor peer reviewed, their value as a basis for future work merits their place in this dissertation. In all cases, FLEDGE can be used with SoSwirly as a low-resource pod-capable container runtime.

5.4.1 Learning optimal service providers

While the implementation of SoSwirly presented in Chapter 4.4 is highly scalable and has low system requirements, it remains a reactive algorithm that merely adapts itself to whatever topology changes it can detect. Because of this, the algorithm will only switch to new service providers when the QoS of its current providers is technically already unacceptable. By introducing a learning component into SoSwirly, it should be able to proactively switch to other service providers. The goal of this section is to provide the fundamental requirements for such a component, and a potential, if untested design. The requirements for a predictive component are straightforward:

- Considering the potential advantages, it is allowed to use more resources, mainly processing power and memory, compared to default SoSwirly. However, it should not require significant processing power (e.g. at most 5% CPU on average), and the memory requirement should be less than that of SoSwirly itself (e.g. at most 15MiB).
- Like default SoSwirly, it must be fast enough to operate in real-time. Considering that it takes around 50ms to 200ms for a SoSwirly topology to organize itself, and that this potentially takes tens of update rounds, a single update round with the predictive component should

not take more than 50ms. In other words, the component is allowed to be an order of magnitude slower if it results in significantly higher accuracy.

- The accuracy of the new component must be significantly higher than that of default SoSwirly. Considering that SoSwirly already has an accuracy of well over 90% in most cases, and over 97% in many others, there is little room for improvement. However, a pro-active approach should avert cases with extremely low QoS compared to default SoSwirly, resulting in an overall smoother experience.

5.4.1.1 Basic design and restrictions

These requirements are used as guidelines for the design of the new component. The required functionality is simple; for any incoming distance measurements to fog nodes, the model should output whether or not the edge node it runs on should switch to that fog node for some service.
While ANN are classically very resource intensive, TensorFlow Lite [42] is explicitly designed for use on low-resource devices, and even a limited model with few parameters may improve the accuracy of SoSwirly if properly designed and trained.
Furthermore, there are two options for training the model, once created. The first is that each node could, periodically, send training data to the cloud, where a new version of the model is trained and distributed. However, this approach is incompatible with the decentralized nature of SoSwirly, resulting in significant overhead and reduced scalability. The second option is to provide each node with a basic (e.g. comparable with default SoSwirly) model on first boot, and to use online learning on each individual node by periodically performing training rounds on real-time incoming data. This approach has the advantage of being highly adaptable to the needs of end-users around a node, but the downside is that training ANN on low-resource devices is currently very expensive in terms of computation time [43]. As such, training rounds will have to be infrequent, and perhaps limited to night-time processing of small batches of random samples collected during the day.
Considering the nature of SoSwirly, this section will continue to build on the second option. Because of the complex nature of the problem, no expert will be able to label all the generated training data on every node, so RL is required, specifically a (Deep) Recurrent Q-Network (DRQN [44]) with a medium discount factor (e.g. $\gamma = 0.7$) in order to focus on short-term future rewards. Magableh et al. [45] illustrate the use of a DRQN for a self-adaptive service architecture, although in their case the algorithm is not

decentralized and takes orders of magnitude more time to decide on actions than is acceptable for SoSwirly.
Since the input is essentially time-series data (i.e. the detected distance to some node at some time), the model can incorporate GRU cells in the hidden layers to store state information and calculate complex time-dependent relations between measurements [46]. Furthermore, the simplicity of the input data neither warrants extracting complex features through a DNN, nor is it suitable for convolution operations. As such, the hidden layers can be reduced to the simple form of a small number of highly, or initially fully connected layers. As edge nodes are to be provided with an initial model, an aggressive dropout and L^2 regularization can be used while training the initial model to limit the number of required parameters, model size, and the time required per inference step. For the smaller training batches on edge devices, regularization and dropout can be ignored.

5.4.1.2 Proposed model

A Q-network outputs the expected value $\hat{Q}$ for an action a at time t, or more formally:

$$\hat{Q}(s_t, a) = F(\bar{X}_t, \bar{W}, a) \tag{5.3}$$

Where s_t is the state of the service architecture on the node, $\bar{X}_t$ is the input tensor at time t, and $\bar{W}$ represents the various weights tensors in the model. The actual value $Q(s_t, a)$ can be mapped to the short-term improvement of metric distance (or QoS) for the node associated with the input $\bar{X}_t$, or:

$$\hat{Q}(s_t, a) = \hat{D}_{t+n} - D_t \tag{5.4}$$

Where D is the distance to another node, t is the current time, and n depends on how far the model must look ahead. Q on the other hand, is the actual measured difference $D_{t+n} - D_t$ at the next distance measurement performed by SoSwirly, and can be used to determine the loss function to train the model. Another interpretation is that the model predicts the velocity of a node in terms of the distance metric to produce $\hat{Q}$, and according to the Bellman equation used for Q-learning:

$$Q(s_t, a_t) = r_t + \gamma \max \hat{Q}(s_{t+1}, a) \tag{5.5}$$

In which r_t is the immediate reward for choosing a node (i.e. $D_{t+1} - D_t$) and γ is the discount factor, the model must also learn to implicitly predict node acceleration, to accurately gauge velocities in future timesteps. Therefore, if the period of SoSwirly update rounds is made arbitrarily precise, a DQN will learn to predict the short-term change in metric distance associated with

a specific node at any time. Furthermore, if nightly training batches are performed, randomly sampled sequences can be stored during the day along with their actual values of r_t and Q. However, the nightly training approach will severely limit the policy [47] that can be used, as only the ground truth results for a greedy choice are available. Considering that reality is the only possible generator for ground truth, more advanced policies require online training.

The components of $\bar{X}_t$ are limited by the design of SoSwirly; a distance measurement consists of a node identification I_N, a metric distance D_N and a timestamp T_N. To reduce input dimensionality, suitable representations for these variables should be constructed. The number of input parameters required to represent the node identification will depend on the maximum expected number of neighbours a node will discover. If this number is small, e.g. less than 20, a one-hot encoding can be used to indicate the relevant node. However, for more neighbouring nodes, a specific embedding must be calculated or learned, an overview of which is given by Potdar et al. [48]. For distance, a simple 8-bit value should suffice, supplying a distance range from 0 to 255. Tokuyama et al. [49] show how timestamps and traffic volume data can be encoded for IoV models, and their findings indicate that day of week is an important factor in addition to a simple timestamp. Finally, this choice of input tensor has a potential impact on training; since the initial (cloud) training should be node agnostic, the training data must be cleaned of node ID's and the training phase repeated for a large number of (randomized) node ID's.

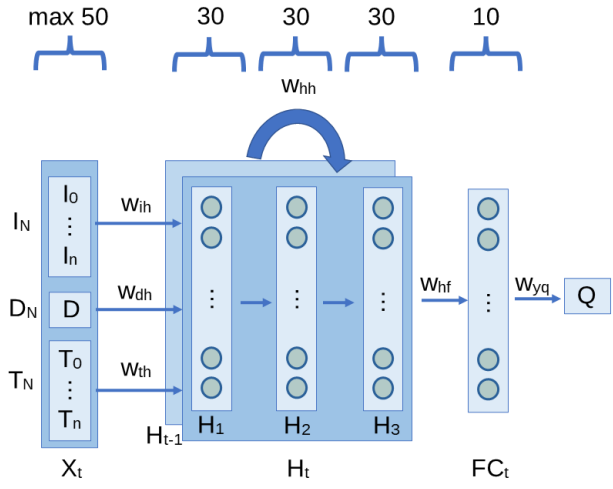


Figure 5.3: Proposed architecture of RQN.

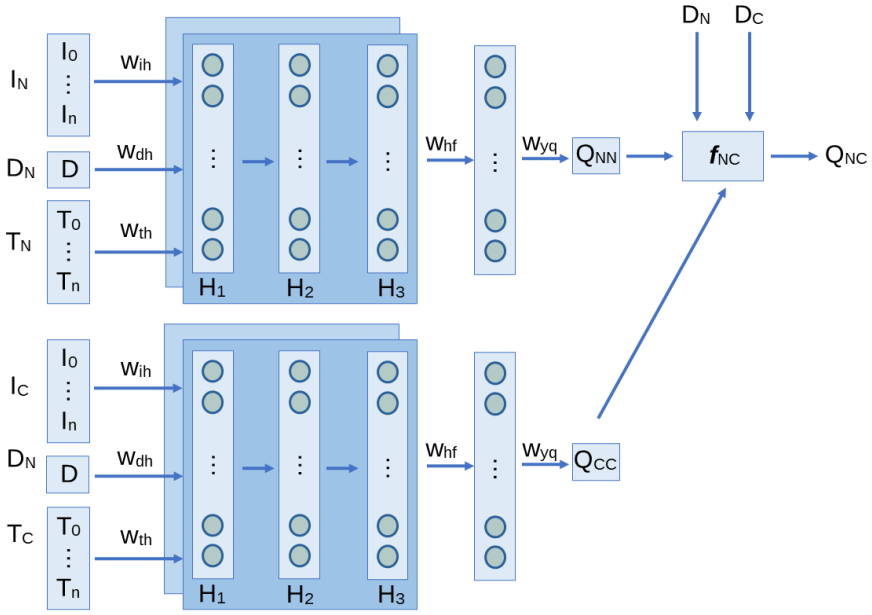


Figure 5.4: Proposed architecture of RQN-based component for SoSwirly.

Although [45] uses only 16 units per GRU layer and FC layer, SoSwirly is likely to require more degrees of freedom to learn the time-dependent behavior of a collection of nodes. Additionally, three hidden layers are used to allow the model to "remember" third-order features from the input data. Note that, independent of these features, higher-order temporal features (acceleration, snap, etc.) are implicitly available because of the recurrent aspect. A single, non-recurrent FC layer of 10 units condenses the third recurrent layer, from which a single activation function outputs $\hat{Q}$. Although GRU layers use computationally expensive *tanh* activation functions, the FC layer uses simple *ReLU* activation functions.

The proposed model resulting from this discussion is shown in Fig. 5.3. The various weights from layers x to y are represented by W_{xy}, although W_{hh} represents the collective weights used for recurrent factors. X_t is the input tensor for time t, while H_t is the collection of hidden layers at time t, including state. FC_t is the fully connected layer before the output $\hat{Q}$.

However, knowing $\hat{Q}$ does not suffice, as it only represents the short-term change in distance of a node N with respect to itself, or $\hat{Q}_{NN}$. In order to determine if SoSwirly should switch to a different node for some service, the improvement $\hat{Q}_{NC}$ over the current node C must be calculated, as given by:

$$\hat{Q}_{NC} = \boldsymbol{f}_{NC} = (D_C - \hat{Q}_{CC}) - (D_N - \hat{Q}_{NN}) \tag{5.6}$$

Knowing both $\hat{Q}_{NC}$ and $\hat{Q}_{CC}$, the lower of these numbers will indicate the best choice of node to use for a service. The proposed architecture to calculate both $\hat{Q}_{CC}$ and $\hat{Q}_{NC}$ is shown in Fig. 5.4. Note that $\hat{Q}_{CC}$ need not be calculated every time; one calculation per update round is sufficient. Finally, in order to reduce model complexity, this design is used rather than creating a Q-network with both results as possible actions.

5.4.1.3 Preliminary performance analysis

A basic implementation of the proposed model, constructed in TensorFlow, contains 18.861 parameters. Straightforward profiling with Keras Model Profiler[1] indicates a memory requirement of 71.9KiB, and 18.4KiB GPU memory for execution batches of a single sample. A reduced model, in which the second and third GRU layers are replaced by FC layers with ReLU activation functions, is about half the size at 9.561 parameters.
Running these models on an Nvidia Jetson Nano, edge-level hardware with default TensorFlow support, results in average execution times of 44.5ms and 21.9ms, for the full and reduced models respectively.
These numbers, although they concern random data, indicate that the proposed solution fulfils the requirements concerning memory use and model speed. However, the average model execution time is only barely below the requirement of 50ms, and improvements such as dropout and regularization can help reduce model size, and thus computational load. Finally, future work is required to validate the accuracy requirement with suitable data.

5.4.1.4 Learning a distance metric

The presented approach is limited in that it still requires all nodes to express distances as a single number. However, given a vector d in an N-dimensional manifold, whose components d^i are the parameters used to calculate the distance D_N [50], then:

$$D_N^2 = \mathbf{g}(d,d), \mathbf{g} = g_{ij} dx^i \otimes dx^j \tag{5.7}$$

Where $\mathbf{g}$ is the metric tensor describing how the total distance is influenced by the gradient products $dx^i \otimes dx^j$ of each pair of dimensions of the manifold. Assuming a linear activation function, the metric tensor is no more than the weights tensor between a layer of input neurons and a single output. If the

[1] https://pypi.org/project/model-profiler/

input is pre-calculated as $d^i \otimes d^j$, a single operation will allow the model to learn g_{ij} as linear combinations of components of d:

$$D_N^2 = \mathbf{g}_{ij}\mathbf{d}^{ij}, \mathbf{d}^{ij} = d^i \otimes d^j \tag{5.8}$$

Learning higher order functions requires modifications to allow more layers. Although determining a single metric tensor for an entire (So)Swirly topology was shown to be impossible, this method allows each node to learn its own metric in order to calculate distances to other nodes. Furthermore, the requirement that all dimensions must be continuous and differentiable no longer applies, as an ANN will learn an approximation for the distance metric even in the case of discrete dimensions. This component can be trained independently of the rest of the proposed model, limiting overall complexity.

5.4.2 Decentralized weight updates

The decentralized learning component described in the previous second may benefit from receiving weight updates from other nodes, in order to "learn" from their experience.

While some approaches already exist for the decentralized dissemination of model weight updates, they are not directly applicable to SoSwirly. BrainTorrent [51] uses a peer-to-peer network to distribute weight updates, which, although proven to be an effective strategy, would result in a significant overhead on each edge node. Lalitha et al. [52] show that a cooperative decentralized learning approach results in a more accurate model than each node learning by itself, but the evaluation is performed with only two nodes, and does not consider the scalability of their method. Notably, they also show that one-hop neighbours can provide sufficient information required for their weight-update mechanism.

As SoSwirly relies on the concept of neighbourhoods to limit the amount of exploration and computation at each node, a suitable highly scalable weight sharing mechanism should do the same. This breaks the assumption of related work that eventually, all weight updates will be spread to all nodes, either directly or by aggregated updates. Additionally, any solution must be designed to operate on low-resource edge devices with minimal connectivity.

5.4.2.1 Basic proposal

The properties of Gossip Learning (GL) [53, 54] closely fit the requirements for decentralized weight updates in SoSwirly. In its basic form, under GL each node learns online with local data, and performs a random walk of

known nodes to request their versions of the model. Ensemble learning is used to merge and update these versions into the local model of a node.

The first aspect, online learning, is performed by the component described in Section 5.4.1.

However, instead of performing a random walk, the second aspect will initially rely on requesting the model of each node in the discovered neighbourhood, making this a geographical rather than random approach. To ensure a smooth and symmetric weight update distribution, a diurnal process is defined during which nodes perform online learning at any possible moment during the day and night, keeping track of an "active" model and a "learned" model, followed by an update round at the start of a new day (i.e. when the date rolls over). This update round is split into two phases; first, every node collects the "learned" model of each of its neighbours, while in the second phase each node merges the "learned" models, including its own, into the "active" model. The update operation is generally described by:

$$\bar{w}_a = \boldsymbol{L}(\bar{w}_a, \bar{w}_{l,r}), \forall r \in R \tag{5.9}$$

Where $\bar{w}_a$ is the model weight tensor of the active model on the local node, $\bar{w}l, r$ is a remote learned model weight tensor, and R is the collection of nodes in the neighbourhood, including the local node.

The exact form of $\boldsymbol{L}(\bar{w}_a, \bar{w}_{l,r})$ is determined by the third aspect, and can be fulfilled by any number of ensemble methods [55], although a weighted average, which is used in straightforward Federated Learning, is the least computationally intensive method:

$$\bar{w}_a = \bar{w}_a + \frac{\sum_{r=0}^{R} a_r(\bar{w}_{l,r} - \bar{w}_a)}{\sum_{r=0}^{R} a_r} \tag{5.10}$$

Where a_r is the number of samples learned on a node during the last day, going from its "active" model to its "learned" model. Synchronizing the separation of the collection and update phases on all nodes is essential; if any node were to start distributing its final updated model while others are still collecting updates, nodes receiving that model would implicitly process some updates with a higher weight, and transitively process some updates they should not have received.

Because of the geographical nature of the weight distribution mechanism, nearby nodes will receive similar updates (and thus similar models), while updates from more distant nodes will eventually be received in a diluted form. Eventually, an equilibrium will result at every node when the daily local learned update exactly opposes received updates:

$$a_u(\bar{w}_u - \bar{w}_a) = \sum_{r=0}^{R} a_r(\bar{w}_{l,r} - \bar{w}_a) \tag{5.11}$$

As $\bar{w}_u$ depends on a learning rate α, this can be rewritten in terms of each learning sample $\bar{x}_i, y_i$, with $\boldsymbol{F}$ being the model loss function:

$$a_u \alpha \sum^{i} (\frac{\partial \boldsymbol{F}(\bar{x}_i, \bar{w}_a, y_i)}{\partial \bar{w}_a}) = \sum_{r=0}^{R} a_r(\bar{w}_{l,r} - \bar{w}_a) \tag{5.12}$$

Showing that a lower learning rate will produce a more homogeneous model, and a higher learning rate will enable independent node behavior. Assuming a homogeneous connectivity, Eq. 5.10 can be extended to reflect the influence of distant nodes:

$$\bar{w}_{a,t+1} = \bar{w}_{a,t} + \frac{\sum_{r=0}^{R} a_{r,t}(\bar{w}_{l,r,t} - \bar{w}_{a,t})}{\sum_{r=0}^{R} a_{r,t}} \tag{5.13}$$

This equation indicates that, if learning samples are equally divided over all nodes, local updates and updates from direct neighbours at day t each have an influence of $1/R$ over the model at day $t+1$. Second order effects are produced by two-hop neighbours and the node itself, which each contribute at $1/R^2$, and the total effects eventually reduce to $(R^{(t-n)} - 1)/R^t$, or $O(1/R^n)$, after t timesteps for nodes n hops away, showing that if $R > 2$, which is an absolute requirement for a functional network, the effect of distant nodes quickly converges, and a low connectivity encourages behavioral islands. Note that the neighbourhood discovery algorithm in SoSwirly adjusts itself to achieve a configurable minimum connectivity, which is set to 10 by default.

5.4.2.2 Alternative approaches

Depending on the application, the severe discount of remote updates predicted by Eq. 5.13 may or may not be acceptable. If remote weight updates should have a bigger impact, the discovery algorithm can be run separately for the weight update mechanism, with a larger discovery radius or higher minimum connectivity. Alternatively, the algorithm can be modified so that each node accepts pushed models rather than pulling them from known neighbours. Using this approach, a node merely sends its own "learned" model to known neighbours, and waits for incoming models for a set period of time before integrating them as in Eq. 5.10. By attaching the node identification and a form of geotag, combined with a maximum distance, the onus is on the receiving node to determine if it should accept and propagate

a specific weight update. Furthermore, nodes should receive each weight update once at most, and all received updates are stored on disk until they are merged. The node identification is used to avoid duplicate propagation if a specific update is received a second time. Using this method, the number of weight updates to integrate can be scaled as far as node hardware allows, to a random walk of possibly tens of thousands of updates, without imposing an overhead on the discovery algorithm. Network overhead, which is likely to be the first issue, can be reduced by compressing the model updates [56]. A rough estimate of which method to use may be derived from the evaluation of SoSwirly. Remembering the theoretical performance of the neighbourhood discovery algorithm, the network traffic requirements of receiving pushed models can be compared to a random walk of the discovered neighbourhood:

$$r_P^2 \rho_F S_m = r_P^2 \rho_F (\frac{86400 B_n}{T_u} + S_m f_{rnd}) \tag{5.14}$$

Where S_m is the size of the model in bytes, B_n is the network bandwidth required to contact a node during discovery, T_u is the period of the discovery algorithm, and f_{rnd} is the frequency of model selection during a random walk. As r_P^2 is the factor to be maximized, it can be dropped while minimizing the remaining factors. Furthermore, as SoSwirly required around 2Kbps to discover 10 nodes every 10 seconds, on average. a rough estimate of 0.25KB can be made for the discovery of a single node. Assuming T_u can be extended to 6 hours for a daily mechanism, the result is:

$$S_m(1 - f_{rnd}) > 1 \tag{5.15}$$

Which, as expected, depends only on the size of the model and the frequency of random selection of nodes. For the proposed model of 80KB, the random walk approach has a lower network overhead if only as much as one out of every 80 nodes is not contacted for a model update, or $f_{rnd} < 79/80$. As a corrolary, f_{rnd} should not be too small, as this will increase the network overhead per processed weight update. As a percentage of model size, the overhead comes down to:

$$O = \frac{86400 B_n}{S_m T_u f_{rnd}} \tag{5.16}$$

Which results in a 1.2% overhead for the break-even case of processing the updates of every 79 of 80 known nodes. This equation can also be used to predict the network overhead of using a larger discovery distance while keeping the number of model updates constant. When extending the neighbourhood radius by a factor of 10, f_{rnd} is reduced to 0.01, and a

much greater variety of node updates is available but the network overhead rises to 125% per selected node update. Evidently, if network bandwidth is the limiting factor of a device, a tradeoff will have to be made between the variety of updates, overhead per update, and the number of updates actually processed. In all other cases, the random walk approach is far more effective.
Finally, the merge operation itself can be modified for better learning; the weights can be dropped in favor of a standard average in case some nodes have too much influence, and other operations may better suit the learning process, and the latter may depend on the learned function, its gradient, and whether each node explores the same (or a close enough) gradient.

5.4.2.3 Preliminary performance analysis

The system requirements of the basic proposal are relatively low. Neighbourhood discovery is done through SoSwirly, and requires little extra memory or processing power. The storage and processing power required to pull models from neighbouring nodes and merge them depends on neighbourhood size. As shown, storing a single model of the SoSwirly predictive component requires about 80KiB, and merging two models requires MADD[2] operations (or similar low-grade calculations) on only 9500 to 19500 weights. Even with 1000 discovered neighbours, low-grade edge hardware should be able to receive 80MiB of data and integrate 19.5 million parameters nightly, but in extreme cases where edge node density is too high, the random walk aspect of Gossip Learning can be reintroduced.

5.5 State of the Art

This section contains the edited version of the following publication: **"Enabling and Leveraging AI in the Intelligent Edge: A Review of Current Trends and Future Directions"**, T. Goethals, B. Volckaert, F. De Turck
published in **IEEE Open Journal of the Communications Society, 2021** [57]

This Section provides a review of the state of the art of Edge Intelligence. First of all, Section 5.5.1 explains the motivation for this review, and lists related work for the main topics of the review; **enabling technologies** for AI in the edge, AI approaches to **organize** various aspects of edge networks, and finally AI-assisted **applications** running in edge networks.

[2]https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation

In Section 5.5.2, the taxonomy of the review is elaborated, and a number of recent studies for each topic are examined in Sections 5.5.3, 5.5.4 and 5.5.5. Finally, future challenges and vision papers are presented in Section 5.5.6 and conclusions are drawn in 5.5.7.

5.5.1 Motivation and Related Work

Although the use of AI in the edge is relatively new, a vast body of work is related to it directly or indirectly. Existing surveys and reviews often focus on an extremely narrow aspect of AI in the edge (e.g. specific enabling hardware, only deep learning applications), without providing a larger context and assuming advanced knowledge of the reader in all the discussed topics. While such works are undeniably useful, the continued expansion of the research field and the divergence of its constituent topics make it ever more difficult to form a high-level overview.

This review aims to provide a holistic overview of what constitutes, and is necessary for, the Intelligent Edge, and to provide a variety of useful, recent studies in this wide area of research. However, this section does not include an exhaustive list of studies for each topic discussed, preferring instead to provide a high-level summary of the state of the art. The topics in this review require a deep understanding of AI, cloud technology, and fog and edge networking. As such, all these concepts are first introduced to the required degree, and references are provided for further exploration. Furthermore, the concept of the Intelligent Edge, being based on two large and rapidly-changing fields of research, is itself volatile and constantly progressing. Therefore, periodical reviews can aid in the continued discovery of research in the field.

The rest of this section presents related work, starting with general reviews and surveys of AI in the edge and continuing with more specific areas of research, such as enabling technology and Smart City AI applications.

5.5.1.1 Edge Intelligence

The work of Deng et al. [58] provides a taxonomy of AI in the edge which focuses mostly on AI for wireless networking, improving service placement using AI and enabling AI, specifically in the context of DNNs. Other aspects of AI in the edge, such as security and reliability, are only summarily explored in favor of a more in-depth technical explanation of the main topics.

A survey by Shi et al. [59] considers the communication efficiency of AI in the edge. The premise of the study is that AI algorithms on edge devices should sparingly use the limited bandwidth available. As such, they present studies

Table 5.1: Comparison of scope of general Edge Intelligence surveys.

Article	AI types	Focus	Additional features
Deng et al., 2020	Deep learning	Wireless networking	Taxonomy
Shi et al., 2020	Neural networks	Communication	/
Zhou et al., 2019	Deep learning	Enabling & integrating AI	EI rating
Zhang et al., 2020	General	IoT sensors	Taxonomy
Wang et al., 2020	Deep learning	General	Taxonomy
This review	General	Organization & applications	Taxonomy

ranging from the training of communication efficient models to optimizing communication between algorithms on different nodes during inference.
Zhou et al. [2] provide a broad overview of studies related to both the training and inference stages of deep learning for EI. In addition, they also provide a rating system for the amount of integration of intelligence in the edge, ranging from cloud-only AI to edge-only AI.
In their survey on Artificial Intelligence of Things, Zhang et al. [60] present a detailed taxonomy on enabling, designing and using intelligence for edge IoT sensors. The article provides a wide range of relevant studies, mostly related to the main topic of learning methods and perception models for IoT.
Wang et al. [61] provide a taxonomy and works related to the various stages of enabling and using deep learning models in the edge, ranging from hardware innovations to actual inference on the edge and relevant applications.
A comparison of these works is found in Table 5.1.

5.5.1.2 Enabling AI in the Edge

Much effort has gone into enabling DNNs on edge hardware. CNNs in particular have very deep and computationally intensive architectures, but the operations involved are highly modular and repetitive, making them excellent candidates for acceleration through custom hardware. A survey by Véstias et al. [62] focuses specifically on accelerating CNNs using reconfigurable computing hardware, while another from Véstias [63] focuses on hardware acceleration of deep learning in general.

In a more general study, Zou et al. [64] list various hardware technologies that enable or accelerate specific types of AI in the edge. Most of these are designed for CNNs or deep learning in general, but some are aimed at Support Vector Machines (SVM). For each technology, the envisioned machine learning tasks and energy efficiency are reported.
In a survey by Nazir et al. [65], a holistic pipeline model for the compression and distribution of deep learning tasks in the edge is presented. As an introduction, this study lists various types of neural networks commonly used in the edge, and links to studies with concrete applications of each type. For the main part, it provides selected studies for each of the stages of the presented pipeline: model compression, (hardware) acceleration and parallelization. Importantly, the authors discuss the various types of model, data, and architectural parallelism that can be exploited to run complex neural networks in the edge.

5.5.1.3 Organizing the Edge through AI

The importance of AI for security in the edge is highlighted in a survey by Mohanta et al. [66]. In this study, they list potential attacks on IoT devices, and refer to studies showing how AI can be applied to prevent attacks (e.g. intrusion detection, malicious app code). Additionally, studies are cited that show how blockchain technology can be used to enable distributed intelligence and ensuring smart contracts.
The use of AI in vehicle-to-everything (V2X) networks is highlighted by Rihan et al. [67]. In their survey, they provide an overview of the potential of AI to both enable next-generation V2X networks, and the future applications utilizing those networks.
More applications of AI for edge networks are provided by Wang et al. [68]. In this article, studies are listed that use AI to enable or improve various aspects of 5G and Beyond-5G (B5G) networks. The authors argue that AI can be used to solve currently intractable problems in the design and optimization of wireless edge networks.
The importance of AI for reliable edge networks is covered by Gupta et al. [69], specifically arguing for the synergy of EI and next-generation 6G networks to enable advanced, low-latency, ultra-reliable applications (e.g. IoV, drones, holographic communication).

5.5.1.4 End-user Applications using AI

In a general survey of the application domains of AI in the edge, Huh et al. [70] provide a number of studies related to often-referenced domains such as Smart Homes, autonomous vehicles, Smart Factory and Smart City, but

also cite studies related to the more general domains of cloud offloading, video content analysis and Mobile Edge Computing (MEC).
An overview of AI applications in the Smart City is provided in a survey by Ullah et al. [71]. This article considers AI in Intelligent Transport Systems (ITS), Smart power grids, and cyber-security of Smart City systems. Additionally, the topic of UAV-based communication in 5G and B5G networks is discussed.
In their Smart Grid review, Gilbert et al. [72] list various studies in three distinct categories: the current requirements and uses for smart grid applications, which smart grid applications benefit from edge computing, and the future challenges for smart grid applications in the edge.
A survey by Sepasgozar et al. [73] provides an overview of AI in Smart Homes and Energy Management Systems. The article presents a deep statistical analysis of the studies found, including co-author connectivity and lexical analysis. Selected papers for each domain vary greatly, but are discussed in detail.
The potential of AI in Internet of Medical Things (IoMT) based health care is illustrated in a survey by Greco et al. [74]. Examples of IoMT-specific devices are health monitoring wearables and field sensor networks, which can be organized in edge networks. Greco et al. provide studies that combine AI and IoT in a wide range of medical aspects, including physiological monitoring, rehabilitation, dietary assessment and epidemic diseases.
In their survey, Angelopoulos et al. [75] provide studies related to the use of AI in Industry 4.0 and Industrial Internet of Things (IIoT). The article provides a taxonomy for AI in Industry 4.0, listing studies for each category with a focus on the link between functionality and the type of AI algorithm used.

5.5.1.5 Blockchain

A survey by Singh et al. [76] provides the required background knowledge on blockchains as distributed, public databases in the context of Smart Cities. A number of studies are provided that combine AI with blockchain for security aspects of Smart Cities, while discussing how blockchains can improve privacy and trust, and analyzing potential issues with blockchain solutions.
In their survey, Yang et al. [77] provide a complete roadmap to the integration of blockchain and edge computing, starting with the motivation for the integration of both technologies, and moving on to frameworks, potential functions of blockchain in the edge, and challenges to the widespread adoption of blockchain technology.

Mohanta et al. [66] list various studies that show how blockchain technology can be used to enable distributed intelligence and ensuring smart contracts. A more specific survey by Wu et al. [78] considers the combination of blockchain and edge computing to improve the security and scalability of IIoT. This survey identifies potential issues with critical infrastructures in Industry 4.0, and argues for the convergence of blockchain and edge computing to tackle these issues, providing various supporting studies.
Finally, Nguyen et al. [79] discuss the potential of the blockchain combined with FL (FLchain) for edge computing, listing opportunities and challenges for various edge applications such as crowdsensing and edge content caching.

5.5.2 Methodology

This section describes the methodology used in constructing a taxonomy and discovering the relevant subcategories for each top-level category. Note that while many of the individual low-level aspects may be applicable to other forms of computing, including general cloud computing, the taxonomy concerns only how they affect AI-related edge computing specifically. This section also elaborates how queries are formed from taxonomy-related parameters to find relevant studies, and how studies are categorized based on current research trends.

5.5.2.1 Taxonomy

Fig. 5.5 show the taxonomy used for this review. Although the main focus of the review is the "Organization" category, both "Enabling Technology" and "Applications" are useful to include because they are closely related to, and often mesh with, proposed frameworks and solutions to organize the Intelligent Edge.
The subcategories are discovered by performing searches on Google Scholar with various relevant keywords, and then grouping the results by recurring subjects. The keywords and subcategories are refined iteratively, until each subcategory contains at least 3 sample studies, but no more than 10, preferably with one recent, dedicated review or survey indicating further research. Table 5.2 shows the final list of keywords, which are also used to construct queries to find the individual studies and articles listed in Sections 5.5.3 through 5.5.5. Although each (sub)category is elaborated in those sections, a short introduction to each is given here to fully explain the taxonomy.

Enabling Technology In the context of this review, enabling technology is defined as any hardware or software improvement that enables or

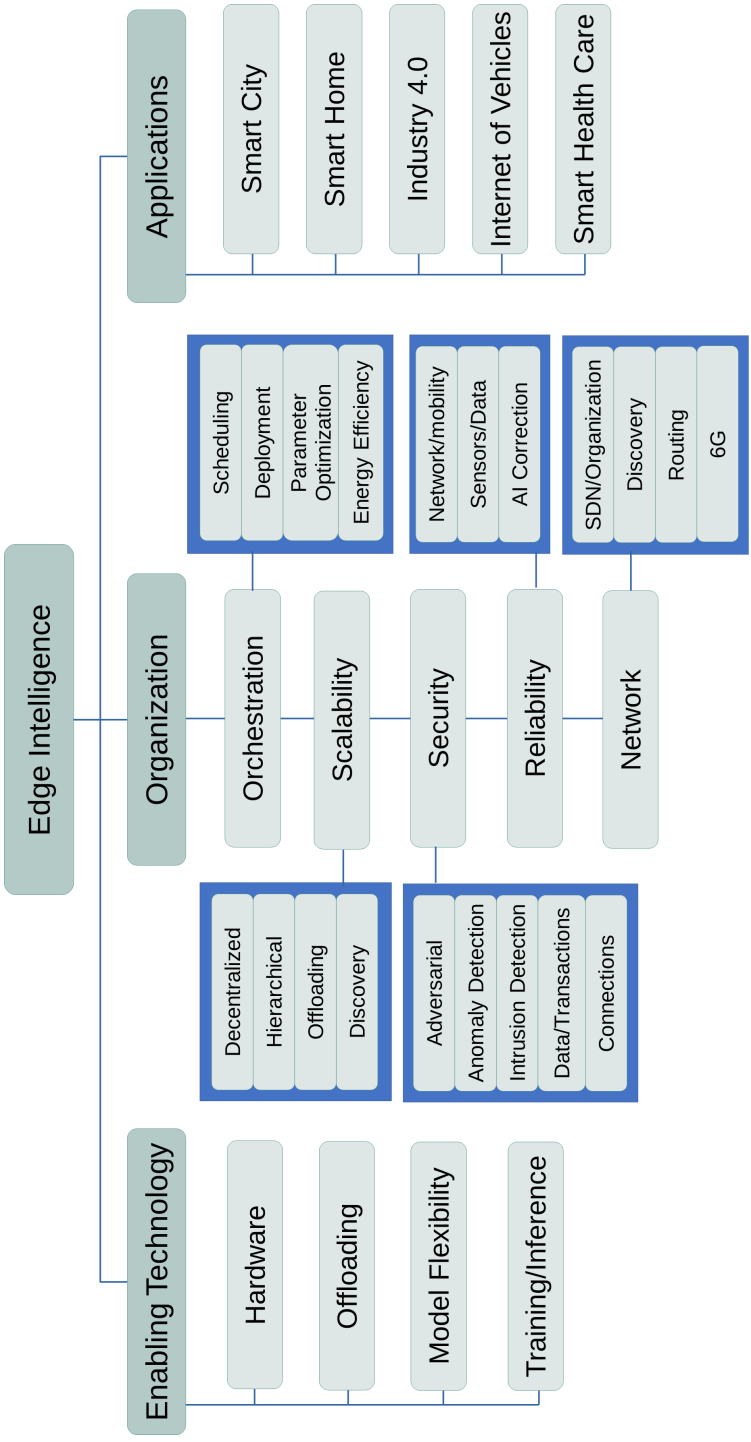


Figure 5.5: Taxonomy of the review.

improves the use of AI on an edge device. In other words, this category entails improvements to AI itself, rather than improvements in edge networks achieved through the application of AI.
There are four popular areas of research in this category:

- **Hardware** improvements or new specialized types of hardware generally increase the performance of AI, although they also indirectly result in new functionality and the ability to use more accurate AI models. Examples of this are the Neural Processing Unit (NPU) [80], which can be optimized for the type of repetitive calculation used in ANNs, and Field Programmable Grid Arrays (FPGA), which in their most basic version can be modified at the hardware level to quickly execute any algorithm without the need for software programming. Any edge device is capable of running neural networks without an NPU, but such a specialized processor can increase performance and practical model size by orders of magnitude without increasing the power requirements of a device.

- **Offloading** is not strictly an enabling factor of AI in the edge, but improves it nonetheless. Its original intent was to move certain well-delineated tasks from the cloud to the edge or vice versa. As such, a lot of research in offloading is related to being able to run AI in the edge in the first place.

- **Model flexibility** relates to different factors that allow the modification of AI models for low-resource edge devices. For example, model compression is used to reduce the size of a model, and to improve its performance, at the cost of a small loss in accuracy. Other approaches involve creating incrementally smaller but less accurate models for different classes of hardware, or using modular models, although the latter is closely related to offloading.

- **Training** and inference phases are often split up, as training a model is very computationally intensive and usually done in the cloud. Because all training data has to be gathered in the cloud, this approach is not scalable. Moreover, it enforces a single model for all devices, even if individual, localized learning may result in more accurate results. Existing techniques such as FL aim to solve this issue by integrating the changes learned by each device into a central model, but training in the edge is still affected by energy efficiency, scalability and processing power.

Organization This category entails studies that use AI to improve the infrastructure of the fog and edge. More specifically, this includes organizing software services, data and (software defined) networks, and ensuring their security and reliability. Note that many of the aspects of "Organization" are closely related. For example, a study may involve a novel method of combining service scheduling and aspects of SDN to improve service reliability or scalability.
There are a number of subcategories that studies can contribute to:

- **Orchestration** is the optimization of service scheduling and deployment, and the study of various parameters involved. Many studies focused on this involve optimizing QoS or end-user experience, balanced against a number of other factors such as energy efficiency, or minimization of resource use or network traffic. Effective algorithms for orchestration in the cloud exist, but in the fog and edge they are complicated by scale, heterogeneous hardware and the need for real-time adjustments due to mobile nodes. This problem can be further complicated by also taking data placement into account, although most studies focus on either data placement or service placement alone.

- **Scalability** focuses specifically on the problems imposed on service and network management by the geographical scale of the edge, and the sheer number of devices in it. Scalability can be achieved by decentralizing frameworks or algorithms, but also by organizing them hierarchically or through modularization. In the first case, self-organizing networks and service architectures can be designed, while in the others the cloud is usually employed as the highest, centralized level of the service architecture. Offloading, as discussed in "Enabling Technology", can be used to move parts of cloud workloads to the edge, and as such represents a limited form of scalability. Finally, automated discovery of nodes and (service) resources is an important step in effective and efficient self-organization on the scale of edge networks.

- **Security** of data and network traffic in the edge is complicated by the increased exposure compared to cloud data centers, and because of the scale of the edge. Research into adversarial attacks attempts to solve security issues with AI itself, in particular DNNs which can be "tricked" into incorrect classification. Anomaly detection and intrusion detection using AI are popular research topics in the security of edge networks, although there are other aspects in securing networks. Similarly, blockchain technologies are gaining a lot of attention for scalable and secure transaction systems, and where data in the edge is concerned, privacy is the most significant aspect of security. Finally,

AI can also be used to secure connections through authentication or authorization.

- **Reliability** of software, networks and data (integrity) ensures the continued and seamless functioning of fog and edge services from an end-user viewpoint. Reliability of software includes seamlessly failing over to other service instances when any instance becomes unreachable, in addition to taking steps that services do not end up in invalid states to begin with. Network reliability involves finding new routes around unreachable nodes or subnetworks, and discovering and maintaining redundant routes. Both network and service reliability require real-time monitoring of nodes and services to enable AI optimization, and they are often used in combination to ensure QoS targets. For data, reliability means not only availability and redundancy, but also the integrity of the data itself. This is different from data security in that data may become unintentionally corrupted due to hardware or software errors. The latter case can also be caused by problems with AI systems, in which case redundancy and correction are needed.

- **Network** organization in the edge is usually done through SDNs, imposing a virtual, software-controlled layer of IP addresses and network functions (e.g. NFV) on top of the physical networks comprising the fog and edge. Apart from SDNs, many studies focus on network resource discovery and application traffic routing in the edge, either as NFV or as part of a holistic approach to edge networking. Finally, 6G networking has recently emerged as a research topic, aiming to integrate AI directly into various aspects of next-generation network management and operation.

Applications "Applications" in the Intelligent Edge differ from the topics listed in "Organization" in that they are AI applications that interact with end-users, running on top of the AI-organized edge. As such, these applications represent the end goal of creating the Intelligent Edge: intelligent applications running autonomously on AI-managed infrastructure, enabled by AI specific technology.
The Intelligent Edge applications discussed in this review are:

- **Smart City** is a collective term for all applications using AI in the context of cities. In this review, only applications that employ EI are considered, although many can also be realized through AI in the cloud, albeit at the cost of increased communication overhead and higher response times. There are a number of popular research topics in this area, such as inner-city traffic and parking management.

Other topics include public health monitoring (e.g. fall detection), and security (e.g. surveillance of specific areas). Scaling Smart City applications to manage entire cities poses challenges in terms of service deployment, traffic routing, resource monitoring and real-time reaction to changes in service and network topology (e.g. movement of nodes, redistributing load).

- **Smart Home** applications aim to gradually improve all aspects of homes, from basic automation to fully AI-assisted living. Similar to Smart City applications, many recent Smart Home studies also focus on health monitoring and security, although there is less need for scaling and more focus on privacy and personalization. Scalability is also an important requirement, but only to be able to deploy the appropriate services to individual homes when required, rather than forming a collaborative service mesh across an entire city.
- **Industry 4.0** aims to improve various aspects of industry and manufacturing through AI. For example, blockchain and AI combinations can reliably log information, which can later be used to track down production chain issues related to faulty manufactured items. Other technologies such as digital twins promise to optimize manufacturing processes by setting up virtual duplicates and searching for ideal settings and parameters, either for each step or holistically.
- **Internet of Vehicles** or IoV has a wide range of applications. Some studies involve the detection of traffic problems and proactively managing the flow of traffic around affected areas, often using dedicated roadside units as computational nodes. Others focus on inter-vehicle communications for optimized traffic flow, or other network-related aspects of autonomous vehicles. In almost all cases, IoV applications need to work with large numbers of fast-moving, unpredictable vehicles, combining the latest in extremely low-latency communication (e.g. 5G or 6G) with highly flexible network and service management.
- **Smart Health Care** aims to combine IoT and AI for various health related purposes, most importantly preventive health care and efficient, personalized patient monitoring. Applications include, but are not limited to, fall prediction, general elderly care, preventive and chronic health care through monitoring, and epidemic monitoring.

5.5.2.2 Query parameters

The results of this review include both numbers on recent research trends, and selected studies, both of which are gathered by querying Google Scholar.

Table 5.2: Query keywords per taxonomy (sub)category.

(Sub)category	Keywords
Enabling Technology	GPU, NPU, FPGA, hardware acceleration, partitioning, inference, offloading
Orchestration	deployment, provisioning, scheduling, optimization, energy efficient
Scalability	scalability, decentralized, hierarchical, discovery, offloading
Security	security, anomaly detection, intrusion detection, adversarial, blockchain security
Reliability	reliability, resilience, fault tolerant
Network	networking, discovery, SDN, routing, 6G
Applications	smart city, smart home, industry, iiov, iov, vanet, iomt, health care

Table 5.3: Query keywords for types of AI.

AI type	Keywords
Regression	regression
Genetic algorithm	evolutionary, genetic
Unsupervised learning	unsupervised
Supervised learning	supervised
Neural network	neural
Federated learning	federated
Distributed learning	distributed learning
Swarm intelligence	swarm

This source is chosen because it is a well-maintained meta-index, linking to studies found in various other indexes. For trends, the following base query is used:

("edge network" OR "edge computing" OR "fog computing" OR "fog network") AND "(keyword(s))" AND "artificial intelligence"

where (*keyword*) is replaced by the keywords from Table 5.2. Note that the keywords are not always directly mapped to taxonomy categories. Rather, they are considered relevant topics which may yield studies that can be mapped onto the taxonomy. The query is crafted to return almost no false positives, and as little false negatives as possible. However, many keywords are mentioned only in passing in loosely related studies, especially as the popularity of any subject increases, so the apparent interest in some topics will be inflated compared to the actual interest.

Historical trends are given from 2015 to 2020. Before 2015, most keywords

yield either unreliable results (e.g. less than 5 studies, keywords not yet coined) or irrelevant results, and 2021 is excluded from the trends because extrapolating numbers from an incomplete year is unreliable. In all searches, the Google Scholar options "include patents" and "include citations" are disabled so that the results represent only studies in which the keywords were actually used in the text.
The trends are presented in Sections 5.5.3 through 5.5.5. Some keywords are inevitably more popular than others, either due to a focus of interest in their specific direction, or due to being often-quoted concepts in studies on EI. Because of this, the results will be presented in two forms; the absolute numbers to indicate the amount of research interest per keyword, and normalized numbers to determine the growth of research interest. In the latter case, the results are normalized to the amount of research interest in 2015 for each keyword. Finally, in the charts for relatively research interest, a "General" trend is added representing the average interest growth in EI.
In addition to the popularity of AI in research topics, the popularity of the types of AI used in the edge, as discussed in Section 5.2, is determined by compiling interest trends using the same methodology as for research topics. The keywords used to gather the data for these trends are shown in Table 5.3. The results are presented in Section 5.5.4.
Further requirements are introduced for the selection of referenced studies from the base query. The studies included in this review range only from 2019 to 2021, and their topics must explicitly relate to a novel application of AI in one or more aspects of edge computing as detailed in the taxonomy. They must also be effectively published in a peer-reviewed journal, barring a limited number of accepted studies from 2021 for which pre-print versions are used.

5.5.3 Enabling the Intelligent Edge

This section discusses recent work related to the "Enabling Technology" category of the taxonomy presented in section 5.5.2.1, including novel hardware solutions, innovations in offloading, AI model flexibility and important progress in (distributed) training and inference algorithms for EI.
Although this review aims to cover all types of AI, (deep) neural networks are currently the most computationally intensive type of AI, and the least suitable to run on general purpose low-resource edge devices. As such, most of this section covers technologies to improve the inference stage of DNNs in the edge.
Fig. 5.6 shows the number of studies that mention keywords related to enabling AI in the edge since 2015. In absolute terms, the most popular topics are offloading and optimization of inference in the edge, followed

closely by GPU acceleration. Considering relative interest, various hardware acceleration methods have gained a lot of interest since 2018, keeping pace with or outpacing interest growth for other keywords.

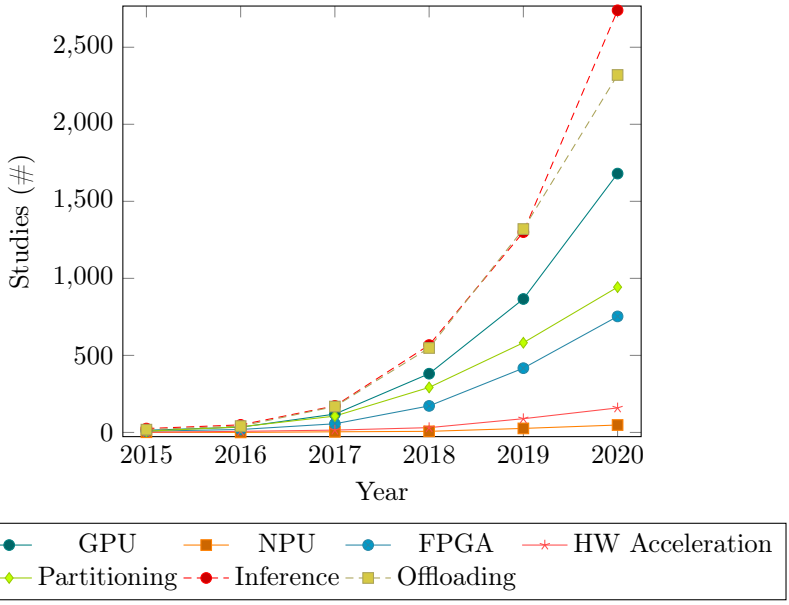


Figure 5.6: Number of studies mentioning AI enabling technology in the edge.

Various dedicated PUs aim to improve the performance of AI inference on low-powered edge devices. Commercial PUs include Google Edge TPU [81] and Nvidia Jetson Nano, both of which are designed for running DNNs in the edge. FPGAs are often used for the acceleration of repetitive but computationally intensive tasks. As an example, the use of an FPGA System-on-Chip (SoC) with OpenStack [82] allows the ARM CPU of the SoC to run a customized OpenStack worker and task planning, while the FPGA itself executes DNN inference. This particular solution uses Dynamic Partial Reconfiguration (DPR) to continually update the FPGA programming, enabling OpenStack to run a virtual machine on the FPGA. Memory is shared between the CPU and FPGA for performance reasons. This solution manages to run a YOLO implementation at 8fps using merely 6.57W of power, coming close to real-time video stream processing.

At the level of single devices, efficient management of different PUs can significantly improve AI performance. In particular, NeuroPipe [83] is aimed at improving the energy efficiency of DNN inference on edge devices by slicing each layer into chunks suited for the processing capacity of each available PU, and pipelining them independently. By parallelizing execution like this,

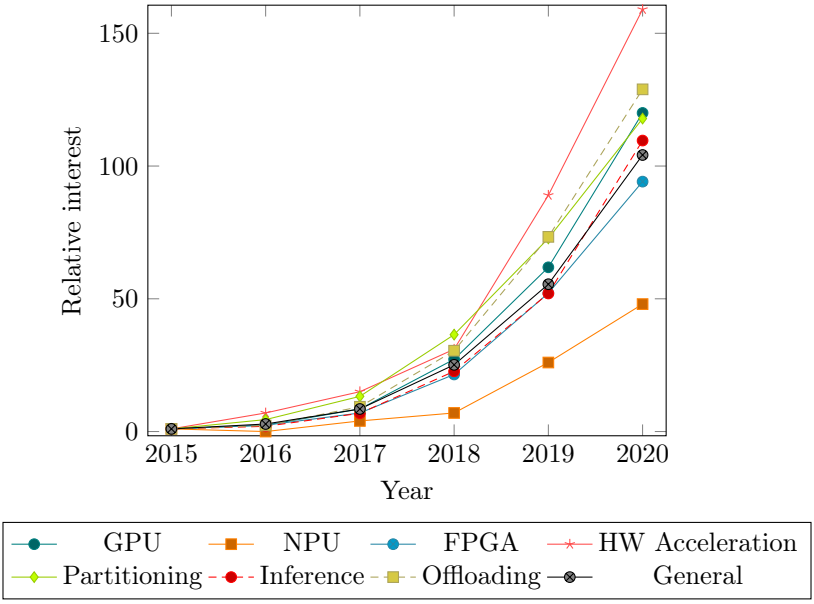


Figure 5.7: Relative interest in AI enabling technology in the edge, normalized to 2015.

NeuroPipe manages to reduce energy consumption by 11% compared to a normal inference run.

Moving up to the level of edge networks, efficiency and responsiveness can be improved by intelligent cooperation between devices. In an example of client-server cooperation, Edgent [84] aims to improve the performance of DNN inference on end-user devices by offloading to edge servers, while maintaining a high responsiveness through co-inference. During an offline stage, Edgent partitions a DNN using right-sizing to optimally divide the workload between devices, after which the partitions can be run on-demand on their respective target machines. The framework is optimized for communication efficiency to reduce the required traffic between edge device and server as they run their respective workloads. Another approach to this problem finds the optimal partitioning point in a DNN by considering latencies between devices and the amount of communication between each pair of layers [85]. The algorithm is evaluated using several CNN models, showing that its offloading results in better performance than local inference, given a sufficiently powerful edge server and at least 16Kbps of network traffic.

Instead of two-part co-inference, DNNs can also be divided into (sub)layer tasks. However, the distributed deployment of such tasks is an intractable scheduling problem (NP-hard). One possibility is to optimize task deploy-

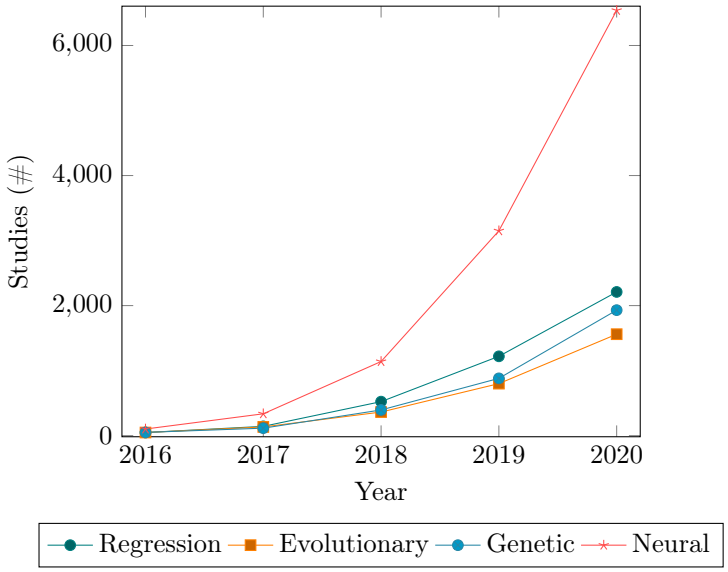


Figure 5.8: Number of studies mentioning various types of AI in the edge, "Evolutionary" and "Genetic" representing the same concept.

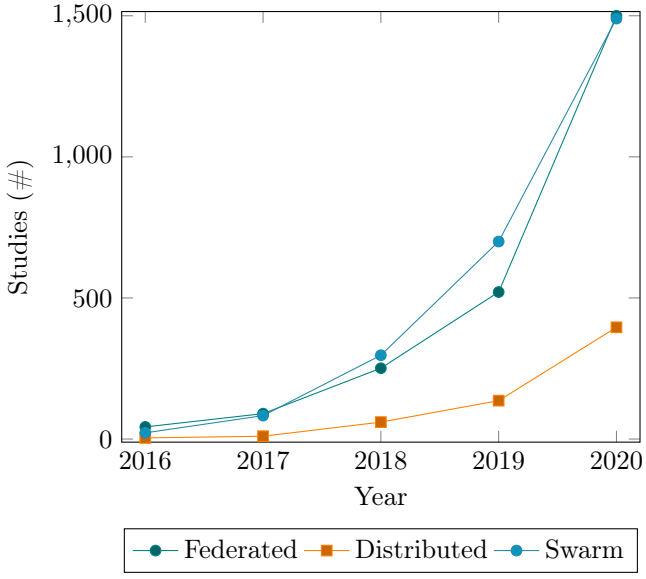


Figure 5.9: Number of studies mentioning various types of distributed AI and learning in the edge.

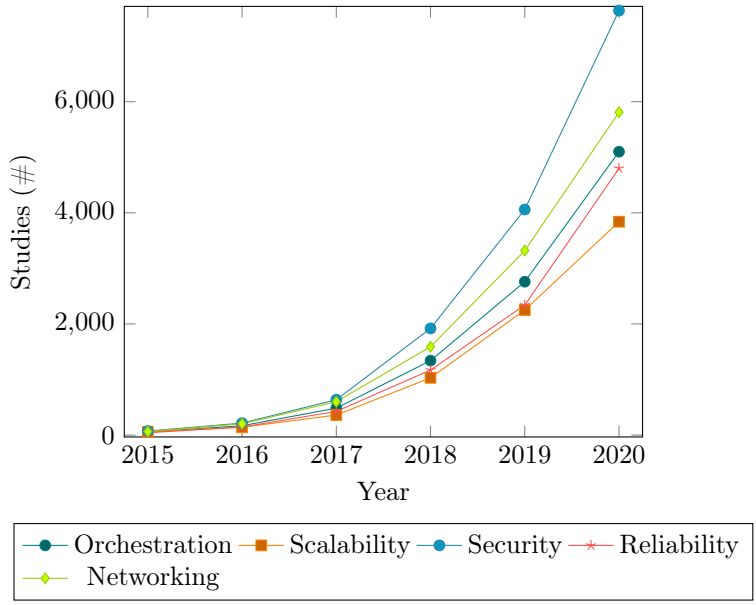


Figure 5.10: Number of studies mentioning organizational aspects of the Intelligent Edge.

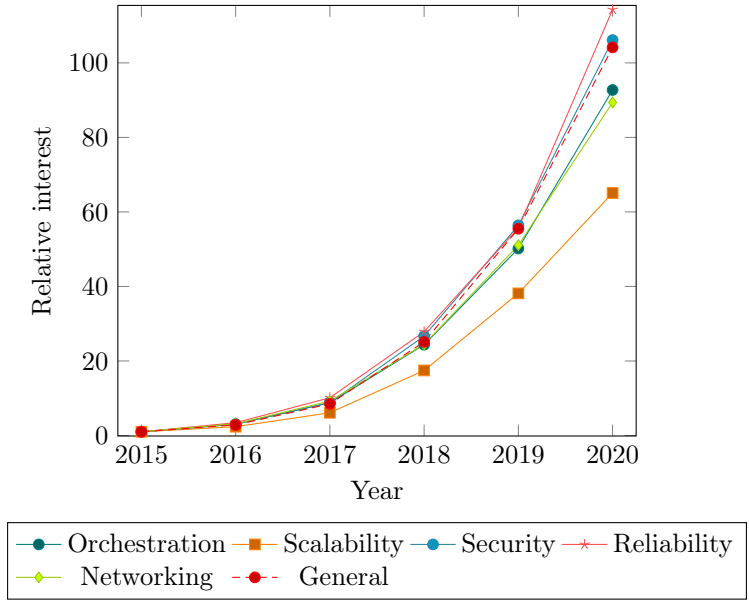


Figure 5.11: Time-relative interest in organizational aspects of the Intelligent Edge, normalized to 2015.

ment for minimal total task completion delays using Solution Space Tree Pruning (SSTP) [86]. This approach is shown to produce significantly lower delays than Edgent, while both perform better than a cloud-only inference model. The addition of partial execution of the inference phase to layer-wise partitioning and offloading of DNNs can result in lower overall inference delays and lower processing requirements, at the cost of reduced classification accuracy. However, this approach significantly improves the performance of real-time applications (e.g. video analysis) on resource-constrained embedded devices [87]. Another solution is to partition not only into layers, but into sub-units of layers, while using a scalable, distributed algorithm to handle the offloading [88]. The Matching Game-based DINA-O offloads each individual piece to different fog nodes based on factors such as queue length, communication delays and processing delays. This approach is shown to have 2.6 to 4.2 times lower total inference latencies than comparable algorithms.

The learning phase of DNNs is far more computationally intensive than inference, and thus more challenging to efficiently realize in the edge. The offloading of learning tasks from the cloud to the edge can be achieved using a graph-based task representation of a DNN [89]. In this approach, the learning task graph is requested on-demand from the cloud, and divided among nearby, suitable edge servers by the edge server that initiated the learning task using NSGA-II. The result is a collaborative learning scheme for DNNs in the edge with feedback of learned parameters to the cloud. Another solution is to remove the need for a cloud server entirely, by using a voting process to select an appropriate edge node as coordinator for a collaborative learning process [90]. The coordinator node is elected by all nodes through a democratic voting strategy, based on computational capacity and distance from the actual deployments. The learning process itself is twofold: a first training batch is executed on the coordinator node, after which the preliminary model is distributed to all other nodes for further training. The computational and energetic impact of the learning algorithm itself can be optimized by using a ternarized gradient [91]. Ternarized Back-Propagation (TBP) uses only the signs of weight differences to update the model weights, rather than calculating whole integer values. Additionally, this method uses L^2 regularization and a mutation rate for weight updates during the training process. The result is increased performance without reducing the accuracy of the resulting trained model, and evaluations show that compared to default backpropagation using 16 bit integers, this method is more energy efficient by two orders of magnitude.

5.5.4 Organizing the Intelligent Edge

Fig. 5.8 shows the interest in various types of AI since 2016. This chart does not include 2015 due to unreliable results for several categories. Neural networks are by far the most popular topic, being mentioned or used in around 50% of the studies in 2020. Around 30% of the studies mention genetic algorithms ("Genetic" + "Evolutionary"), while regression methods receive 20% of the total attention. Although neural networks are Turing complete [92], and can thus technically perform any type of calculation, the use of other AI methods makes sense in many situations, for example when the problem is more easily modeled for a different approach, or when hardware requirements are too stringent to run a neural network. The interest in various types of distributed AI is shown in Fig. 5.9. The number of mentions of all keywords has increased over 10 times in just 4 years, showing a strong interest in decentralized AI in the edge, although specific interest in FL and SI significantly outpaces general distributed algorithms. General research trends are presented in Fig. 5.10 and Fig. 5.11, showing that while there is some spread in the numbers of studies mentioning various aspects of organizing the Intelligent Edge, the relative growth is more or less equal for all keywords. The only exception is "scalability", which lags in both absolute and relative interest.

5.5.4.1 Orchestration

Fig. 5.12 and Fig. 5.13 show the number of studies and relative interest in edge orchestration for AI, or using AI. Optimization and energy efficiency attract the most research interest, while provisioning is least mentioned. Despite significant differences in absolute interest, all keywords have a comparable growth in relative interest, indicating significant research potential in any topic.

An example of decentralized AI task orchestration is Cognition-Centric Fog Computing Resource Balancing (CFCRB) [93], which uses a node exploration algorithm and distributed Q-learning to find the optimal nodes to offload computational tasks to. CFCRB consists of three main concepts; **sensing** involves knowledge of node resources and IoT data acquisition, **interacting** involves efficient communication and coordination, and **learning** finds the optimal strategies for dividing workloads over resources.

Self-Optimizing Swirly (SoSwirly) from Section 4.4 is another distributed edge-oriented orchestrator, using SI to let edge nodes and fog nodes find their own optimal service providers. Nodes run a discovery algorithm to find other nodes in their neighbourhoods, requesting services from other nodes based on their available resources and distance. Services are rede-

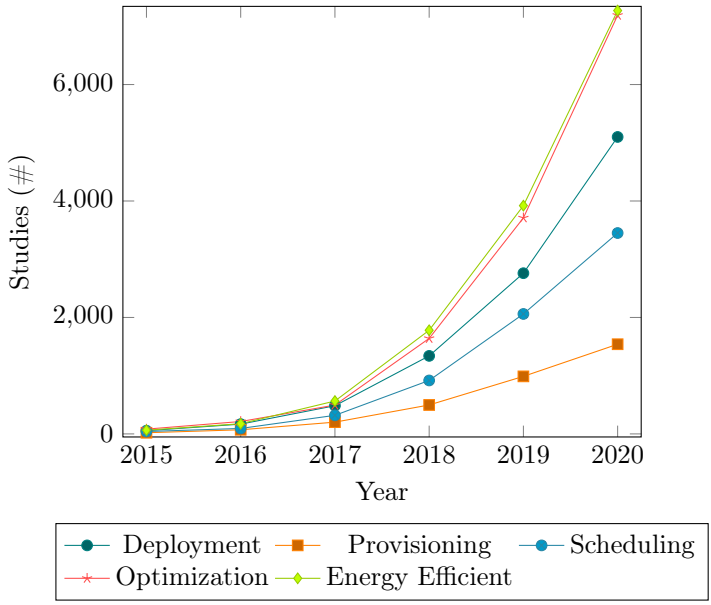


Figure 5.12: Number of studies mentioning software orchestration in the Intelligent Edge.

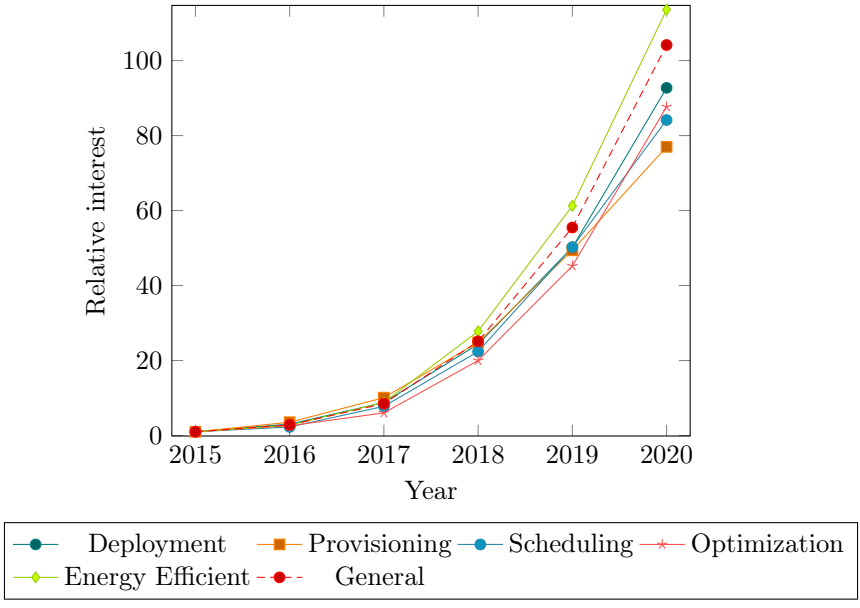


Figure 5.13: Time-relative interest in software orchestration in the Intelligent Edge, normalized to 2015.

ployed on-demand in real-time as node positions and resources change, and performance is shown to be two orders of magnitude faster than evolutionary algorithms (NSGA-II).

For a more fine-grained demand-oriented deployment strategy, MDPs and a Dueling-Deep Q-network can be combined [94] to orchestrate services in mobile edge networks based on patterns in end-user service requests. Additionally, the algorithm also decides whether to let an edge service instance handle any request, or to forward it to the cloud. The approach is compared to various other deep learning solutions, showing an improvement in both total response time, and the number of requests executed in the edge rather than forwarded to the cloud.

The decision of whether or not to offload tasks from an edge device to a computational node can also be modeled as an MDP, and optimized using ϵ-greedy Q-learning [95]. This approach takes into account available resources on nodes, as well as communication channel properties and task queue lengths. Evaluations indicate execution times in line with those of offloading everything to edge servers, but higher power requirements than computing everything locally.

Fuzzy Clustering Algorithm with PSO (FCAP) [96] is a combined algorithm in which both fog nodes and computational tasks are represented as standardized resource vectors. In a preliminary phase, Fuzzy Clustering is used to divide fog nodes into computational, storage and network nodes depending on their resources. PSO is used to avoid local optima during this clustering phase. In the scheduling phase, the task resource vector is used to find the best matching class, and the most suitable node to run the task on from that class.

A similar approach, I-FASC [97], clusters the tasks into categories rather than the computational nodes, using the same classes as FCAP. The tasks of each class are scheduled using a modified Fireworks Algorithm (FA), a crossbreed between SI and evolutionary algorithms. Evaluations show that I-FASC has lower execution times than comparable algorithms, while also providing a more stable load across nodes as the number of tasks increases.

Rather than using AI to directly determine the nodes to offload tasks to, AI-Based Task Distribution Algorithm (AITDA) [98] uses a neural network on each computational node to predict the execution time for potential tasks. The predictions are based on task type and task input data, and the results are combined with policies to determine if a task should be run on a fog node or in the cloud. The example policy optimizes both response time and network traffic, and results show a significant advantage over either completely cloud-based or completely-fog based processing.

The use of dual neural networks with RL aims to provide an integral cloud

to edge optimization [99]. In this approach, the first network predicts if a specified task is suitable for execution in the fog, while the second distributes fog-allocated tasks among computational nodes. The second network optimizes task placement for evenly distributed resource use and minimal communication, with the explicit goal of clustering interdependent tasks on the same nodes to further reduce network traffic.

LATA, an approach to jointly optimizing communication efficiency and end-user latency specifically for fog nodes connected by a wireless SDN [100], aims to balance the workloads of fog nodes to achieve better global response times. The algorithm itself is distributed over the SDN controller and the fog nodes, and evaluations show consistently lower latencies than comparable solutions.

Rather than focusing only on the optimization of latency versus communication efficiency, FairTS [101] uses a resource-centered approach to online task scheduling in the fog. This solution is based on Dominant Resource Fairness (DRF) to ensure that all types of resources are divided fairly among running tasks, using RL to learn the optimal assignments. Comparison to a greedy strategy shows similar average task completion times, but more stable execution times and thus potential QoS guarantees.

Applying a fairness policy to computational nodes rather than resource allocations, Fairness Cooperation Algorithm (FCA) [102] aims to fairly divide tasks between fog nodes based on their available resources, for the joint optimization of global minimal energy consumption and task processing time. To train FCA, an algorithm is presented which converges slower than either Newton Descent or Steepest Descent in early rounds, but results in smaller error rates after only 75 rounds.

In their work, Yang et al. [103] present an MDP-based model which attempts to optimize the use of FL in EI. Arguing that while FL preserves privacy, it also has a negative effect on battery-powered and low-resource devices, their algorithm aims to jointly optimize both privacy gains from FL, and resource use on edge devices.

Distributed Artificial Intelligence-as-a-Service (DAIaaS) is a different take on distributed AI task orchestration [104], aiming to provide a standardized framework for distributed intelligent services in Internet of Everything (IoE) environments. Deployment parameters considered by this framework are CPU requirements, network traffic and link latencies, and it is evaluated in terms of energy and financial costs for three distinct use cases.

FogBus [105] provides a Platform-as-a-Service (PaaS) approach to cloud-fog-IoT integration, allowing platform independent deployment of software services. A multi-tiered architecture is used to standardize communication and application behavior, separating IoT devices from communication gate-

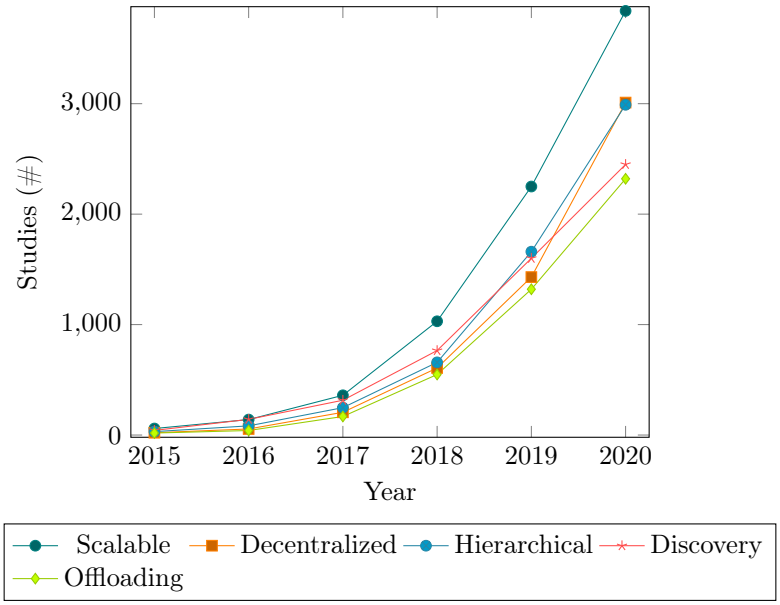


Figure 5.14: Number of studies mentioning scalability in the Intelligent Edge.

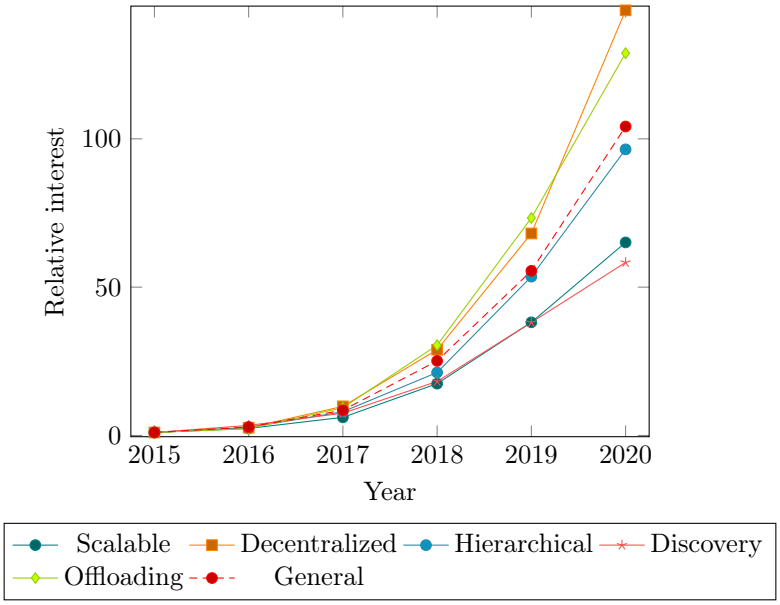


Figure 5.15: Time-relative interest in scalability in the Intelligent Edge, normalized to 2015.

ways, computational nodes and the cloud. A blockchain implementation is added in addition to other security features to ensure data integrity when transferring confidential data between nodes. The platform is evaluated in terms of energy efficiency, latency and resource use.
Some orchestrators are designed for specific domains, for example in Mobile Crowdsensing [106] using DRL with a CNN to organize the execution of tasks. The orchestrator is designed to schedule divisible computing tasks generated by edge devices, deploying each subtask in the fog or cloud depending on computational requirements. The scheduler aims to guarantee QoS for each task, and to minimize processing time and network traffic for each task.
Other orchestrators are aimed at databases [107] rather than computational tasks, using MDP as a probabilistic method to determine database placement and to guarantee freely definable QoS requirements for application developers. This database orchestrator is evaluated using a Kubernetes-based implementation, and compared to Analytic Hierarchy Process (AHP) in terms of QoS violations.

5.5.4.2 Scalability

The research interest in scalability in the Intelligent Edge is shown in Fig. 5.14 and Fig. 5.15. All keywords are more or less equally mentioned, with the umbrella term "Scalable" occurring more often, although interest in "Discovery" feathers off slightly in 2020. As growth in relative interest is concerned, "Offloading" and "Decentralized" have the fastest growing interest, while "Discovery" again lags.
Scalability and efficient orchestration have largely overlapping requirements. As a result, many of the studies listed in this section are similar to the ones discussed for "Orchestration", they have been specifically selected to illustrate one or more aspects of scalability for EI.
While offloading is mostly an enabling technology and requires new organizational algorithms, it can also be used as a tool for scalable AIoT (Artificial Intelligence of Things). Splitting neural networks layer-wise and offloading the initial layers to IoT devices [108] has the advantage of not only scaling part of the training process with the number of edge devices, but also that training occurs where the IoT sensor data is most readily available. For the higher layers, less data intensive learned features are communicated to the cloud for further training.
On the level of a single neural network model, scalability can be achieved through the offloading of each layer to different devices. Accelerated Artificial Intelligence for IoT (AAIoT) [109] is one such approach, optimizing the response time of inference versus network traffic and computational effort

through dynamic programming. Furthermore, the algorithm can operate in multi-layer IoT architectures, rather than a two-layer cloud-fog architecture.

Intelligent service discovery and integration are an essential part of functionality scaling, providing new functions without the need for manual implementation of suitable interfaces. One approach using Generative Adversarial Networks (GANs) [110] shows the potential of this type of neural network for self-learning service discovery in the edge, specifically in the context of 6G networks. The generators in this approach are trained to produce synthetic data associated with distinct service classes, based on captured data. The discriminators have the dual tasks of recognizing real data from fake data, and associating it with a specific generator. As such, specific traffic flows are discovered by adding generators, whereas the discriminator identifies and classifies them.

Decentralization is an important aspect of scaling EI, as no centralized or offloaded algorithm can scale to the load of exponentially growing edge networks. One approach enables distributed, cloud-cooperative intelligence by combining a Task Model Offloading Algorithm (TMOA) and Adaptive Task Scheduling Algorithm (ATSA) based on Ant Colony [111]. The former assigns nodes to tasks based on computational capacity, latency and energy efficiency, while the latter ensures load balancing of AI tasks between nodes. However, in this approach the scheduler algorithm itself remains a centralized instance. Evaluations show performance comparable to or better than state of the art alternatives.

A study by Lim et al. [112] considers the scale limiting bottleneck of communication inefficiency in FL, and resource allocation problems in the more efficient Hierarchical Federated Learning (HFL). They propose a two-level resource allocation solution for HFL. In the lower level, evolutionary game theory is used to model the process of data owners joining cluster heads, based on rewards given for data participation. In the upper level, a deep learning-based auction mechanism is used for cluster heads to service model owners. This added level of indirection is shown by evaluations to lead to stable resource allocation.

SoSwirly from Section 4.4 uses an approach based on SI to distribute the task and service orchestration process itself. Each end-user device is responsible for finding the nearest suitable fog node for the services it requires, switching to other fog nodes in real-time if QoS requirements are violated. Furthermore, SoSwirly can be layered for a hierarchical architecture, from the IoT sensors through various layers to the cloud.

Recently, the network edge has been used to distribute Data Stream Processing (DSP) [113] for intelligent applications, parallelizing stream processing and increasing scalability. For example, Aggregate End-to-End Latency

Strategy with Region Patterns and Latency Awareness [114] (AELS + RP + LA) aims to decrease the processing latency of DSP applications in geo-distributed cloud-edge architectures. The solution analyses DSP application graphs to determine the optimal offloading strategy, and is shown to scale up to 250.000 edge resources (edge nodes and IoT devices). Another edge DSP framework [115] optimizes energy efficiency by reducing network traffic in real-time through two components. The first is an energy-aware IoT data gathering component, using adaptive sampling to reduce its network traffic, while the second is a data prediction model which calculates future data for multiple sensor IoT environments. The data prediction model uses clustering to filter outliers and to generate reliable data, and the framework is shown to be up to 60% more energy efficient for IoT devices than continuous data streams. Finally, Processing Intelligent Agent Running on Fog Infrastructure (PIAF) [116] uses Time Petri Nets to model time-critical DSP in the context of industrial settings, using intelligent agents to distribute DSPs among the available edge nodes.

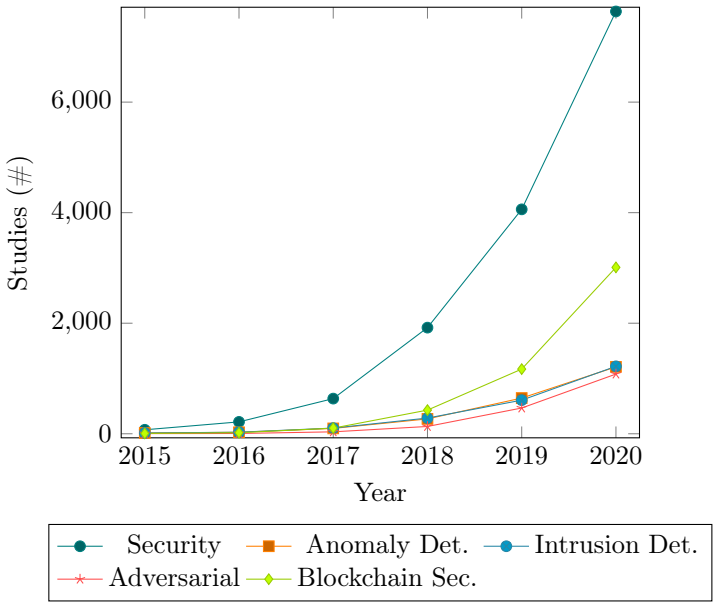


Figure 5.16: Number of studies mentioning AI security in the Intelligent Edge.

Another solution for the distributed management of cloud and edge resources for intelligent applications uses modified Virtual Infrastructure Managers (VIMs), specifically OpenStack in the cloud and Docker in the edge [117]. Both OpenStack and Docker are extended with a custom resource management API (DARK), and a Network Function Virtualization Orchestrator

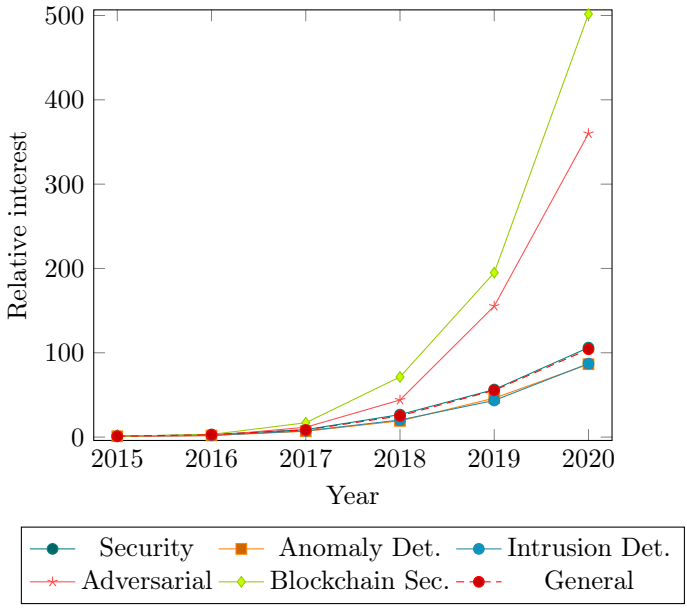


Figure 5.17: Time-relative interest in security in the Intelligent Edge, normalized to 2015.

(MORCH). The scheduling algorithm in DARK works in real-time, mapping incoming requests in the form of service graphs onto available resources and nodes using a greedy heuristic, taking into account network conditions between nodes. The MORCH component enables network-awareness for a multi-layer architecture.

The goal of scalability studies can also be limited to a single aspect of EI. For example, a scalable Intrusion Detection System (IDS) for Smart Cities [118] based on the distributed training and inference of neural networks. Two workflows are presented, a semi-distributed approach in which feature selection is distributed but final classification is performed by a central instance in the fog, and a fully distributed version. While the accuracy of the distributed approach is about 2% less than a centralized algorithm, the Time To Build Model (TTBM) is 64.82 times faster.

5.5.4.3 Security

The research interest in various aspects of security in the Intelligent Edge is shown in Fig. 5.16 and Fig. 5.17. While there is a great interest in security itself, more specific keywords are mentioned far less, possibly indicating that most studies focus on one specific topic, or that general security concerns are

a secondary topic of many loosely related studies. While there is a significant interest in challenges such as anomaly detection and intrusion detection, there has been an explosive growth in interest related to adversarial attacks and blockchain-based solutions since 2018.

SecOFF-FCIoT [119] is aimed specifically at secure offloading of computational tasks. The data is secured at the sensor level using a Neuro-Fuzzy Model which predicts device sensitivity to malicious data injection, and offloaded to appropriate fog nodes using PSO, taking into account the processing capacity and energy levels of nodes. Although some tasks are offloaded to the cloud, RL is used to ensure data privacy by offloading tasks with sensitive data only to private clouds. Evaluations show this approach has a significantly lower energy consumption and response latency than comparable solutions.

Secure Mobile Crowdsensing Protocol (SMCP) [120] provides a framework to secure data and ensure privacy for crowdsensing applications in the edge. The framework uses a cloud server to act as a registry for fog and edge nodes, using Extended Triple Diffie–Hellman Key Agreement (X3DHKA) and Advanced Encryption Standard (AES) as lightweight algorithms to secure traffic and enable the mutual authentication of nodes.

A general approach to anomaly detection in the fog is provided by Yang et al. [121], along with a concrete example of a Deep Network Analyzer (DNA) for 5G networks.

An unnamed holistic framework by Jararweh et al. [122] offers a distributed approach for trustworthy and reliable edge services. This framework incorporates custom algorithms which deploy services in the edge and guarantee user privacy. To ensure data and network traffic integrity, a neural network-based IDS is integrated. Evaluations show that the accuracy of this IDS is up to 99.3%, and that response times can be significantly reduced by scaling the number of edge servers.

Another solution for anomaly detection uses a collaborative/transfer learning approach in the fog, using Principal Component Analysis (PCA) for initial feature engineering and using a variety of models (e.g. RL, DNN, SVM) for each node, selecting the optimal one [123]. The fog enabled infrastructure supporting this distributed AI consists of standard software such as Hadoop[3] and Spark[4], using both batch and streaming modes.

SeArch [124] is a hierarchical IDS for SDN-based cloud IoT, deployed on edge gateways, fog SDN controllers and as a cloud application. Communication channels are restricted to the same level or one level higher in the architecture. The algorithms at each level are restricted by computatio-

[3]https://hadoop.apache.org/
[4]https://spark.apache.org/

nal power; SVM for node-level detection in the edge, Self-Organizing Maps (SOE) for network-level detection in the fog, and deep learning in the cloud. Evaluations to alternative solutions show that SeArch has similar accuracy, but significantly lower detection times.

In another approach to IDS, a framework using TA-Edge [125] uses Trusted Authority edge nodes to certify other edge devices in their domains, securing communication between them. The second component of the framework, SDN-ADS, is an SDN/Openflow based anomaly detection system which first discovers the topology and SDN data flows of the entire network. This topology is used by a malicious traffic detector to find packets with invalid properties or routed through anomalous flows.

In the context of Smart Homes, AI can be used to monitor smart devices, and to authenticate and authorize them to interact with the cloud [126]. This approach learns the specific behaviors of devices in the home network, which allows the creation of device profiles that can be authorized by end-users.

Adversarial attacks exploit the modelling properties of deep learning networks to cause misclassification or incorrect outputs. Small, but intentional perturbations in the input can reliably cause misclassification, whereas accidental, seemingly random input can sometimes be misinterpreted by a DNN, with high probability outputs. DeSVig [127] is a decentralized approach to correct such problems within milliseconds in Industrial AI systems, using Conditional GANs (CGAN) to verify the inputs and outputs of DNNs against attacks. The CGAN is trained to generate copies of the inputs supplied by DNNs, while a separate discriminator compares the generated copy against the actual input to determine whether it contains signals that indicate an attack. Evaluations indicate 96-99% accuracies for several datasets, and detection within 62ms.

Another strategy aims to construct and execute DNNs safely by ensuring data integrity during both the training (poisoning/backdoor attack) and inference (adversarial) phases, and the security and privacy of data transfers during training [128]. Secure training is achieved by simultaneously using active, pending and secure models for each application to detect suspected hostile data. These data are stored in a "hostile" dataset and used to update the pending model, while eventually a separate detector DNN will recognize the hostile features and integrate them into a new secure model. Security during inference is enforced through a punishment mechanism derived from a game model.

Blockchain technology is often used in conjunction with AI for the secure processing and distributed storage of transaction-like data in edge networks. For example, in Smart Healthcare [129] AI on edge devices can be leveraged

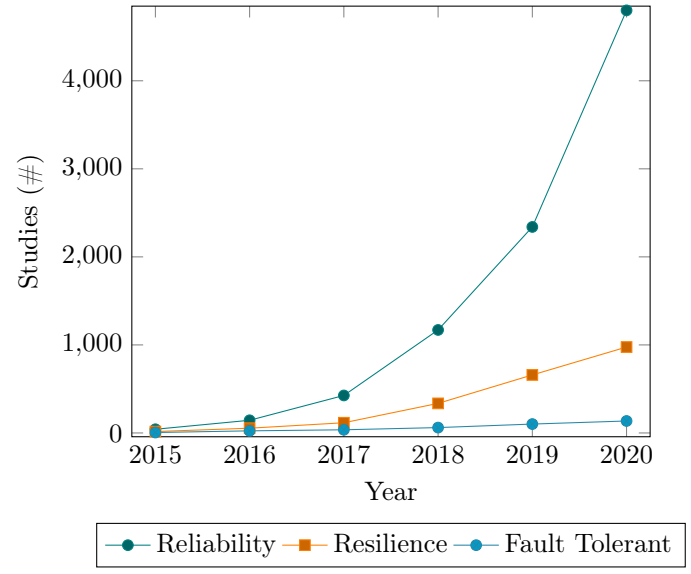


Figure 5.18: Number of studies mentioning reliability in the Intelligent Edge.

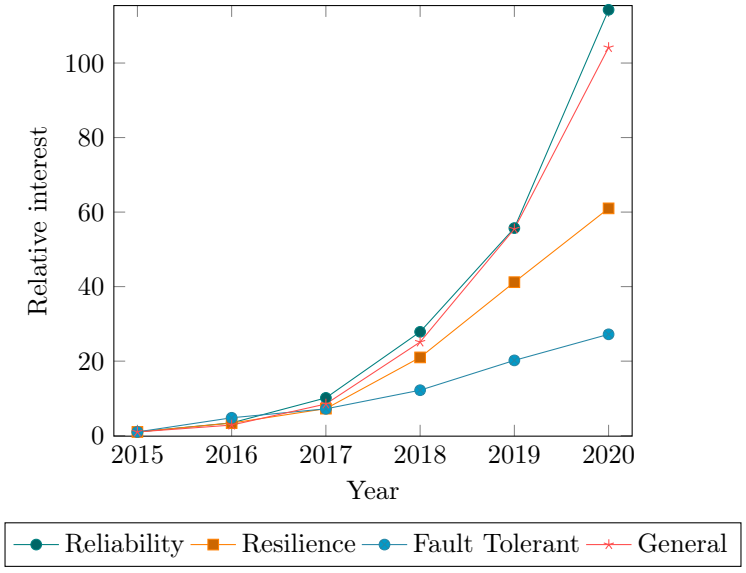


Figure 5.19: Time-relative interest in reliability in the Intelligent Edge, normalized to 2015.

for biometric data analysis and feature extraction, the results of which are stored in a blockchain, or enable the execution of smart contracts in the edge. A concrete implementation considers arrhythmia detection with a CNN in the edge, storing the resulting output along with device ID and other transactional metadata in an Ethereum chain.
Similarly, blockchain applications can aid with privacy concerning sensitive data in the edge by processing data locally using AI, and keeping track of all parties accessing the resulting features by using an Ethereum chain [130].
A different application combines blockchain-based smart contracts with trustless smart oracles for trust management in the fog computing platform of DECENTER [131]. This particular framework uses blockchains to register trusted components and users, while smart contracts use data provided by the smart oracles to verify QoS and trust requirements.
Finally, BlockSecIoTNet [132] provides an example of using blockchain technology as part of an IDS. An SDN based IoT network is used to continually monitor node traffic, allowing ubiquitous and decentralized IDS, while a blockchain ensures decentralized, trusted data storage and logging of the transactions between components. A similar approach can be applied to traffic in Vehicular Ad-Hoc Networks (VANETs) [133], in which the blockchain provides trust between actors and components.

5.5.4.4 Reliability

Fig. 5.18 and Fig. 5.19 show the research interest in reliability in the Intelligent Edge. The trends presented here are similar to those for the security; great interest in reliability in general, but far less interest in specific aspects, especially fault tolerance. However, the interest in reliability keeps almost perfect pace with the general interest in EI, indicating that it is consistently an important and pervasive topic in studies whose primary subject does not necessarily concern reliability.
As the reliability of systems starts with their input, reliable IoT sensor data is an important enabling factor of EI. One approach towards reliable sensor data uses fog-based validation by combining the output of several physically clustered sensors of different types to detect unreliable outputs [134]. The algorithm is applied to a scenario in which AI detects people through a security camera, showing that false negatives of the AI can be corrected through sensory substitution.
Moving up to the level of reliable AI using IoT data, deepFogGuard [135] is a DNN augmentation scheme which makes distributed inference resilient to failure. The main feature of this scheme is that it relies on skip hyperconnections, which function like residual connections in DNNs, except that they skip entire nodes rather than simply layers. By ensuring at least a min-

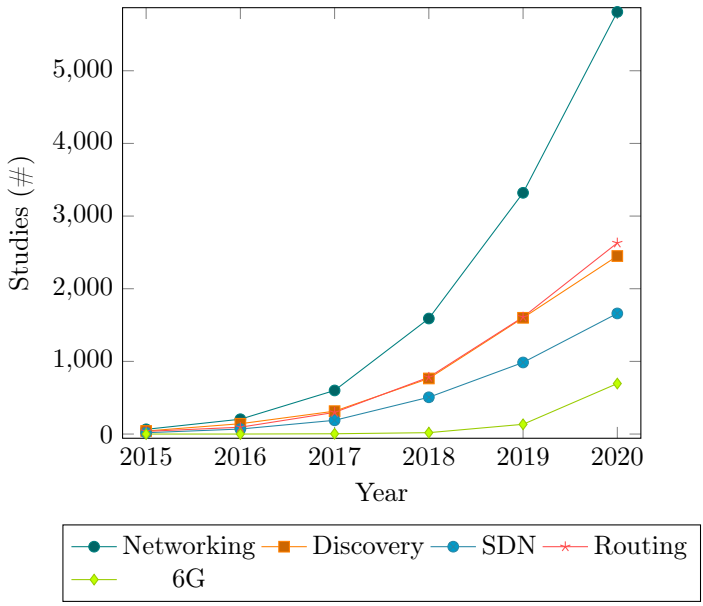


Figure 5.20: Number of studies mentioning networking in the Intelligent Edge.

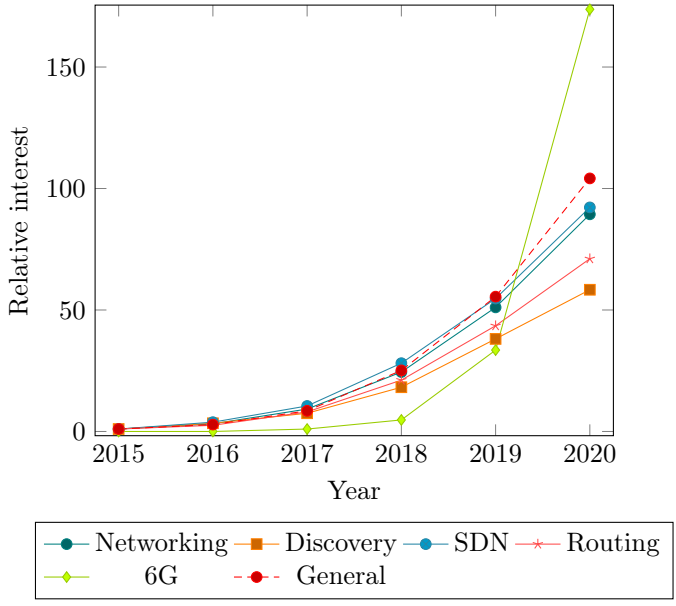


Figure 5.21: Time-relative interest in networking in the Intelligent Edge, normalized to 2015.

imal data flow from lower layers on node failure, deepFogGuard is shown to significantly improve inference accuracy over default DNN inference, especially for high node failure rates. The mobility of vehicles in IoV can be a detriment to timely and reliable inference, but the application of AI and coded computing can instead exploit this mobility through opportunistic offloading [136]. This solution uses a modified Multi-Armed Bandit (MAB) approach to learn the delay behavior of nodes in real-time, while coded computing is used for redundant offloading, accepting whichever results are received first.

Finally, on the scope of networks, work by Radanliev et al. [137] develops a risk-assessment framework for the purpose of creating secure and reliable networks in extreme environments, specifically in the context of edge computing and AI.

For holistic solutions aiming to enhance overall reliability, Elastic Intelligent Fog (EiF) [138] is a general, AI-enabled fog computing framework designed to enable distributed and reliable IoT systems. The approach is similar to offloading, but implemented as PaaS, offering APIs for network, IoT and AI functions for edge deployments. The framework itself uses real-time monitoring to enable Follow-me Moving Edge Cloud functionality, in which services "follow" users in the edge, employing FL to update the deployment strategy.

5.5.4.5 Networking

The interest in network-related aspects of the Intelligent Edge is shown in Fig. 5.20 and Fig. 5.21. In absolute terms, discovery and routing have attracted the most interest since 2015, although interest in 6G has skyrocketed since 2018 as the concept of the Intelligent Edge has grown, and at the current growth rate it will become one of the most discussed topics in EI within 2 years.

Work by Xia et al. [139] illustrates the effects of AI and Fog Radio Access Networks (F-RANs) on each other. It discusses how the deployment of distributed and hierarchical AI, especially DNNs, is enabled by the properties of F-RANs, while F-RANs themselves are organized more efficiently by AI. A concrete example is given through the use of MAB to solve a caching problem with unknown content popularity.

Offloading can be used to optimize network performance, for example by minimizing communication power consumption in wireless networks [140]. Unlike most similar approaches, this framework uses statistical learning, specifically iteratively reweighted L1 minimization with difference-of-convex functions regularization. Evaluations show that this approach results in a significantly lower power consumption than comparable algorithms.

By dividing large networks into cells and applying a CNNs, cell outages and congestion can be detected and traffic rerouted. The scalability and reliability of this approach can be increased by distributing the CNN over edge servers, each managing 100 cells [141]. Evaluations indicate that this distributed anomaly detection has up to 96% accuracy.

Inductive Content Augmented Network Embedding (ICANE) [142] uses a network embedding which preserves higher order (multi-hop) node proximity, aimed to facilitate service deployment in edge networks. The embedding is learned by sampling network nodes for neighbours up to k hops, and transforming proximities and node resources into feature vectors, which are fed to an LSTM based network. Evaluations show that ICANE has significantly higher F1 scores [143] than similar algorithms for various learning datasets.

Because of its virtual nature, SDN allows for new possibilities in ad-hoc network organization. For example, a self-adaptive SDN based solution can organize virtual topologies based on application demands, available resources and physical topologies [144]. A practical implementation uses ONOS SDN controllers and OpenFlow switches, deployed by a self-adaptive framework, to organize the SDN. While this particular approach does not yet employ AI, the authors plan to use machine learning to improve the organizational algorithm.

Another framework combines the flexibility of SDN with extra security [145], with a focus on Smart Healthcare. IoT devices are authenticated by edge servers using a lightweight probabilistic k-nearest neighbour (p-KNN) based algorithm. The edge servers are used for collaborative intelligence, offloading tasks to each other, while the SDN controller is responsible for load balancing and network optimization between them. The offloading algorithm uses a form of SI, with each edge server using Beacons to alert nearby servers if their task queue grows too long.

Intelligent real-time routing decisions can greatly improve network performance. As an example, Smart Edge Broker (SEB) [146] has a dual purpose. Its main purpose is routing Smart Home traffic in edge networks, acting as a broker to organize direct communication between edge nodes instead of routing through the cloud. By keeping all communication between nodes in the edge network, latency and traffic overhead are reduced. It also acts as an edge server, filtering and processing any incoming data instead of forwarding it to the cloud.

AI plays a critical role in most research on next-generation 6G networks. One architecture [147] defines four layers of AI in 6G; intelligent sensing, analytics, intelligent control and smart applications, examining which types of AI would be suitable for each purpose. Further topics discussed include communication spectrum management, AI-empowered MEC, and intelligent

mobility management.

5.5.5 Applications in the Intelligent Edge

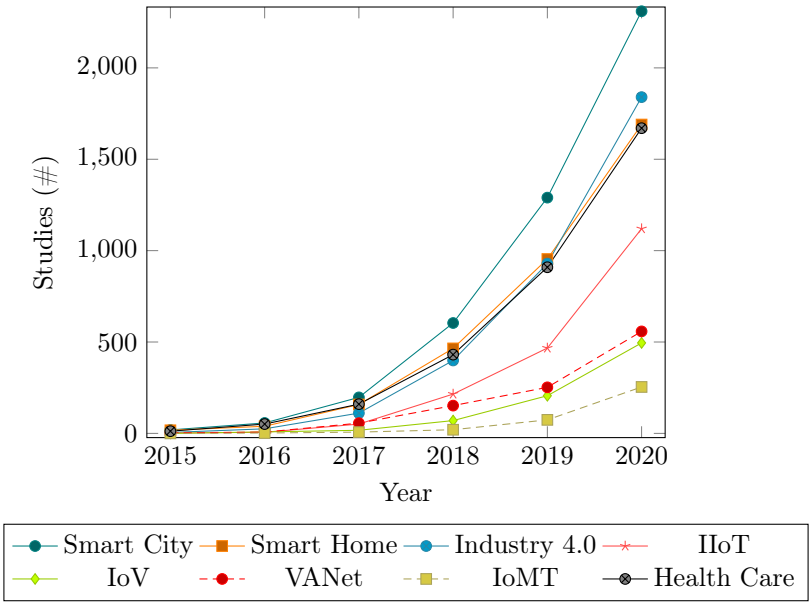


Figure 5.22: Number of studies mentioning Intelligent Edge applications.

Fig. 5.22 and Fig. 5.23 show the research interest in Intelligent Edge applications. In absolute numbers most domains are equally popular, but the relatively few and variable mentions of related abbreviations (e.g. IoMT, IIoT) indicate an uneven terminology. However, the recent relative growth of "IoMT" and "IIoT" may simply indicate some time is required for their widespread adoption. In relative terms, the interest in industrial applications is rising explosively, even compared to the significant growth of other domains.

5.5.5.1 Smart City

Apart from the specific domains of health care, IoV, IIoT and Smart Homes, the Smart City comprises a large number of topics and potential AI applications. This section discusses only some of the most recent, AI-based applications as an introduction.
Smart Grids, often regulated by Energy Management Systems (EMSs), play an important role in an ever more fine-grained energy grid, optimizing for demand and minimizing losses and overproduction. Improving EI paves

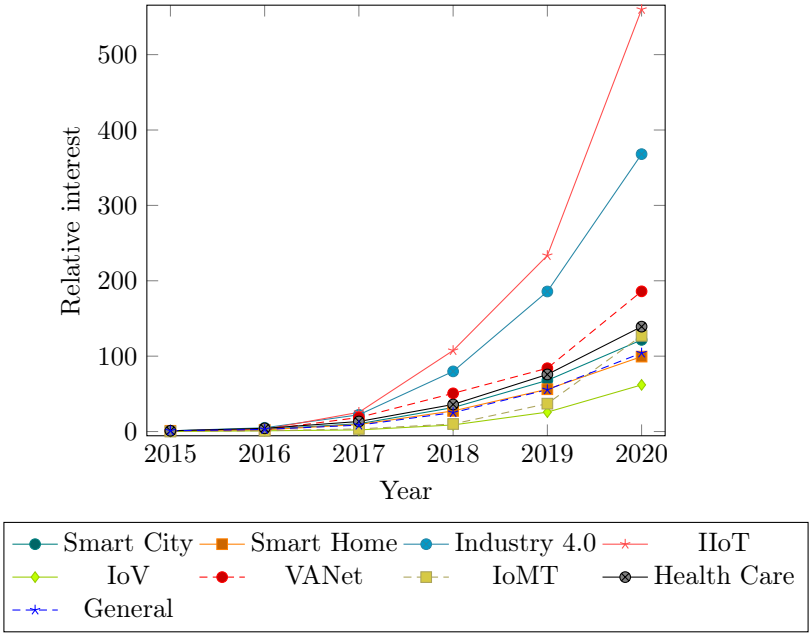


Figure 5.23: Time-relative interest in Intelligent Edge applications, normalized to 2015.

the way for the decentralization of Smart Grid functionality to the edge, such as an AI-oriented Smart Power Meter with edge analytics for use in a cloud-assisted EMS [148]. Another Smart Grid application is the detection of energy fraud using edge-based AI [149]. In this work, data from smart appliances and distributed power sources (e.g. solar panels) is pre-processed using Principal Component Analysis (PCA) and Missing Completely At Random (MCAR), and evaluations show that neural network-based regression shows promise for the classification phase.

Another popular Smart City topic is parking surveillance. The advent of deep learning on edge hardware has enabled real-time intelligent surveillance systems based in the edge. Such systems can combine processing power of IoT devices and edge servers [150], optimizing for performance while minimizing network traffic. The work in this study combines background subtraction with a Single Shot Detector (SSD) on IoT devices, and a tracking algorithm on edge servers to efficiently track multiple vehicles in poor lighting conditions. The work of Mittal et al. [151] summarizes the use of and challenges of deep learning in the edge for more general surveillance applications in the Smart City.

5.5.5.2 Smart Home

Edge-based surveillance systems similar to those in Smart Cities can be applied to crime prevention in Smart Home environments [152]. This work uses an event-driven approach, where edge devices use background subtraction for quick, naive motion detection. Upon motion detection, an edge device forwards video data to fog nodes, which use a CNN (VGGNet) to classify crime objects (e.g. guns). When an object is detected, labeled images tagged with location information can be forwarded to the relevant authorities.

ImPeRIum is a general, fog-enabled Smart Home solution [153]. In its architecture, data is gathered by sensor nodes and forwarded to nearby computation capable devices (defined as fog nodes, e.g. Smart TV, gateway device) for decision making. The decision process uses both an ensemble method and MLP, and the models are distributed over all fog nodes to avoid a single point of failure. Efficient dissemination of data to other devices is achieved through a Publish/Subscribe mechanism (MQTT), only publishing an event when it is dissimilar enough to the previously sent one.

The next section discusses some AI applications in Smart Health Care. Some of these can overlap with Smart Homes, for example a fog-based framework for predictive veterinary health care [154]. This framework uses FogBus [105] as a base platform, along with a WiSense mesh and a health sensor belt. The Probability of Health Vulnerability (PoHV) is calculated in the fog using sensory, environmental, behavioral and dietary data. The PoHV is further processed by a temporal ANN (t-ANN) which predicts a Temporal Sensitivity Measure (TSM), classified into alert levels. Finally, a Self Organized Map is used to create day-to-day visualizations for caregivers.

5.5.5.3 Smart Health Care

Fall detection is an example IoMT application which can benefit greatly from running in the edge [155]. This type of real-time application requires pervasive sensor and wireless networks, although these are often low-powered and have limited bandwidth. In the proposed architecture, sensor data is sent to a local edge gateway over low-power Bluetooth, where an LSTM RNN performs fall detection in real-time. In case a fall is detected, an event is sent over LoRa to a fog server, which sends the required notifications to caregivers.

One Smart Health Care framework is based on collaborative learning, distributing AI over the edge and fog [156]. A case study on arrhythmia detection has edge devices performing ECG signal pre-processing, feature extraction, and classification with a shallow neural network. If the proba-

bility of the classification is too low, a CNN in the fog layer takes over, using the full ECG image as input. Finally, ECG data is also streamed to the cloud, where it is combined with health provider data to train and improve (personalized) models. A similar application [157] uses EEG data to predict seizures in patients, but this approach uses Discrete Wavelet Transform (DWT) as an additional de-noising step and a Kriging model (Gaussian regression) as a classifier. Finally, deep learning can also be used for disease prediction. For example, biometric data from IoMT sensors combined with medical metadata, processed by a Deep Factorization Machine (DeepFM), can predict the presence of hepatitis [158].

5.5.5.4 Industry 4.0

A general study on distributed AI in IIoT by Queiroz et al. [159] lists key concerns for the synergy of distributed AI and Cyber-Physical Systems (CPS) in industrial settings. These concerns are fine-grained, covering every aspect from networking and embedded hardware to human-machine interaction, identifying cross-concerns for the successful cooperation of AI and CPS.

Infrastructurally, IIoT can use the same multi-tier architecture as used in various other applications of AI in the edge, with the edge interfacing with humans and machinery. One such architecture uses a cloud tier for training, fog/edge tiers for distributed inference and an SDN layer to seamlessly connect all devices [160]. An IIoT AI task scheduling algorithm is implemented on top of this architecture, optimizing for latency by taking into account computational capacity and the proximity of edge servers to manufacturing equipment. Evaluations show this approach is significantly more efficient than either cloud computing or ad-hoc, in-place execution.

Similar work also uses a multi-tier architecture, arguing how it solves latency, bandwidth and security problems compared to a purely cloud-based IIoT approach [161]. This approach uses the edge as an interfacing and control layer, and the fog as an information integration layer. The potential for a multi-tier approach to enable Digital Twin Shop-floor (DTS) is discussed, in which the virtual representation of the physical shop-floor can be used to intelligently manage and improve manufacturing processes.

A blockchain-edge framework for IIoT by Kumar et al. [162] does not directly involve AI, but is aimed at facilitating AI applications, and the potential for integrating FL into the blockchain is discussed.

Fogsy [163] is a holistic system for the training, deployment and management of AI in industrial settings. It operates as a fog/edge cluster management system, with facilities for data procurement and management in addition to AI model management. It also features AI pipeline management,

and explainability of models through causal graphs.
A study from the Smart Maintenance Living Lab presents an approach for Smart Predictive/Preventive Maintenance [164], using a three-tiered platform based on Obelisk, a fog- and cloud-based Smart City framework. Data from IIoT sensors is gathered in the Edge tier and pre-processed (e.g. feature extraction) by gateways, after which it is sent to Obelisk in the Platform tier for ingestion. A collection of machine learning algorithms act on the ingested data to generate dashboard data for a centralized Enterprise tier.
An example AI application in IIoT is smoke detection in foggy environments [165]. This approach uses an energy efficient residual CNN based on MobileNet V2, designed for deployment in Smart City and IIoT settings. Evaluations indicate both higher accuracy and better performance than state of the art solutions.
Another concrete application of AI in IIoT is real-time poultry monitoring using EI [166]. Data from sensors monitoring the atmospheric concentration of gases such as ammonia, methane, and carbon (di)oxide is fed into an RNN with GRUs on an Nvidia Jetson Nano, predicting the evolution of air quality around the poultry farm.

5.5.5.5 Internet of Vehicles

In F-RANs in an IoV context, the increased wireless network traffic caused by intelligent applications can cause interference on wireless channels. RIMMA (Reliable and Interference-free Mobility Management Algorithm) [167] solves this problem by managing channels based on their characteristics, over AI-driven F-RANs. Furthermore, RIMMA is combined with fog computing to optimize for mobility, reliability and packet loss.
A similar problem on a topological level is efficient caching and communication management in quickly changing topologies with moving nodes (cars) and RSUs (Road-Side Units), especially considering the severe latency constraints on IoV applications. One solution to this problem uses twin timescales for mobility-aware offloading/content requests [168]; a long-term strategy determined by PSO, and a short term strategy determined by deep Q-learning. These strategies consider not only resource use, but also hard deadlines for requests.
Another approach considers the energy efficiency of workload deployment in fog-cloud IoV applications [169]. The algorithm uses a Learning Classifier System (specifically XCS, genetic-based machine learning), optimizing for energy use and workload delay, taking into account battery status of battery powered nodes. Evaluations show that the approach generally results in higher average battery levels than comparable algorithms, and significantly lower execution times.

There is much untapped potential for higher-level IoV applications. For example, Seal et al. [170] recognize that in an IoV context, the flood of data from vehicles will soon outstrip the ability of the cloud alone to process it. They develop a benchmark for real-time traffic incidence identification and traffic control, and using a multi-tier testbed, determine that a deep learning approach using (tiny)YOLOv3 is up to 80% faster in an edge-cloud architecture than in the cloud alone.

5.5.6 Challenges and Vision

In this section, future research directions and challenges are presented for each of the main topics of this review. For a high-level vision, we can look at the highly cyclical history of computing [171, 172], which shows that eventually all functionality ends up as close to end-users as possible, in a complex interplay of productivity, end-user experience, evolving hardware and networking capabilities, and costs. The last decades have seen ever more functionality deployed closer to end-users on increasingly pervasive network infrastructures. As such, it is safe to assume that AI and other resource-intensive tasks will continue to move to the edge. Additionally, the next wave of innovation might very well emerge in a centralized form, but quickly take advantage of the infrastructure provided by the Intelligent Edge. Such next-generation applications could be highly tailored to ubiquitous user interaction, and their concepts far removed from physical systems due to increased virtualization and intelligent management.

5.5.6.1 Enabling Technology

While common AI frameworks such as TensorFlow are capable of offloading calculations to dedicated hardware, they only offload to one PU per task. In the edge, where there may be many nearby PUs, layer-based slicing provides better overall results, offloading individual layers to different PUs based on their capabilities. This approach can be further optimized through intelligent management of PUs, monitoring their performance for several types of AI tasks. At the local network level, computational tasks are often offloaded to individual devices. A synergy with PU level offloading, either hierarchical or by peer-to-peer sharing of PU details, could provide better performance. Another challenge at this level is the efficient integration of new types of PUs with computing frameworks. Considering the highly customized nature of most hardware (e.g. FPGAs), this will likely remain in the realm of manual work, rather than automated discovery.
Significant progress has been made in low-power inference in the edge, although improvements are still likely due to incremental gains in hardware

performance and model efficiency. AI training however still requires immense amounts of computation and power, often beyond the reach of individual devices in the edge, and increasing as neural networks grow deeper. To combat this issue, computational efforts can be offloaded to more powerful layers in the fog, multi-mode AI models can be deployed which trade accuracy for performance on resource-constrained devices, or the training workload can be spread out over many devices through FL or other cooperative strategies. While further research into truly distributed, cooperative strategies will certainly yield better performance for years to come, the learning process can also be greatly improved by reducing the amount of training required. Contributing factors for this may include improved regularization, fast-converging gradient descent strategies, and zero-shot learning.

5.5.6.2 Organizing the Edge

In terms of orchestration, important challenges are real-time redeployment of (AI) services in volatile network environments, and opening up new classes of devices for the flexible deployment of services and AI. For the former challenge, the ideal is to achieve optimal QoS (e.g. availability, latency) for all users at all times, while optimizing any number of other factors (e.g. resource use, network traffic). The latter ensures that more devices can contribute their processing power, and help optimize the general functionality of the Intelligent Edge. This can be partially solved through better hardware, but also through lightweight operating systems capable of suitable virtualization (e.g. containers, unikernels). Energy efficiency is of particular importance, as edge applications increasingly push intelligence to even the most limited IoT devices. Some devices have extremely restricted power supplies, while others are battery powered and may not be (easily) rechargeable. Challenges consist of optimizing network traffic and response times over low-powered protocols, and reducing total CPU use over the lifecycle of a device. As for strategies, some may involve offloading of workloads generated by the devices, while others try to divide a known workload over a pool of devices before their batteries run out.

Significant open challenges for scalable systems are true decentralization of orchestration, and self-organizing service meshes. Almost all recent work relies on a multi-tier architecture, using the cloud as a critical infrastructural component to some degree. However, a truly scalable and flexible architecture can not be dependent on any centralized, resource-bound component (e.g. the cloud) to support an unbounded collection of devices (e.g. the edge). Likely factors to enable new architectures are peer-to-peer weight updates for AI models, local discovery of functionality and resources, and

inverted deployment in the sense that an edge node primarily decides where to request/deploy a service, rather than being directed to an instance by load balancers or load balancing (distributed) DNS. As a combination of these factors, an Intelligent Edge could be envisioned in which (AI) services simply "follow" users through nearby computational nodes, pre-emptively moving to other nodes as they learn user behavioral patterns.

While progress in anomaly and intrusion detection is likely to continue, improvements are more likely to be in terms of performance and response time than accuracy, considering the high accuracy of current systems. Recent solutions for adversarial attacks are similarly effective, especially when combined with redundant systems, but adversarial attacks could be severely diminished by studying the fundamental properties of state of the art neural networks that give rise to these vulnerabilities in the first place. The increased popularity of blockchain solutions for distributed, secure transactional storage and smart contracts indicates their usefulness, but widespread adoption requires solving fundamental problems of blockchain technology related to energy consumption, fast consensus protocols, and security in privacy-sensitive applications. Privacy is a significant driving factor for the Intelligent Edge; if data is locally processed it can not be intercepted. However, privacy mechanisms are still important for processed data which is sent to the cloud, and for distributed architectures, particularly the aforementioned blockchain solutions. Some privacy concerns may be alleviated by using different types of sensors (e.g. environmental instead of cameras), encouraging (AI) services based purely on actions and behavior rather than learning properties of individuals.

The latter solution to privacy can aid with reliability; if several types of non-visual sensors are involved in a single decision, a system may be able to detect when a malfunction occurs in one of them. Network reliability in itself is important for many Smart applications, especially in IoV settings where extremely low latencies are required, despite fast-moving network nodes. Solutions to this challenge may include redundant offloading, more effective predictive offloading, and improving the reliability of layer-wise offloaded AI models. Similarly, the resilience of distributed neural networks can be improved through various means, leading to an indirect improvement in overall reliability of EI.

There are many opportunities for network-oriented research using AI in the edge. An important topic is intelligent network management through NFV and SDN to consolidate large edge networks, forming a logical, reliable topology for edge applications. Subtopics include automated discovery and integration of network resources, and redundant routing which adapts in real-time to discover optimal routes. Furthermore, initial work on 6G envi-

sions the integration of AI into every aspect of networking from hardware through connection management to support for applications. The wide-scoped, long-term effort required to form a new, EI oriented, next-generation set of networking standards will undoubtedly provide great opportunities for new forms of intelligent network management.

5.5.6.3 Applications

While there are many innovative AI-based applications in the edge, this aspect of the Intelligent Edge is still in its inception, with true Smart Cities still in the distant future. Most studies focus on a single, narrow application within their domain, using little in the way of standards and rarely considering future integration with other applications, but providing valuable proof of the potential of the Intelligent Edge. This is partly due to the constantly changing underlying technologies, which cause rapid obsolescence of existing applications, and give rise to many others. In such a rapidly changing area of research, there are many opportunities. For example, some studies present basic Smart City/Home/Industry EI management frameworks, but standards and integrated, greater scope frameworks (e.g. what TensorFlow did for neural networks) are mostly absent. Existing Smart City frameworks such as FiWare [173] and Obelisk [174] are mostly cloud-based, offering a broad support of IoT communication protocols and scalable data processing, but do not explicitly contain edge-oriented intelligent features. Ongoing IEEE standardization efforts, as presented in Section 5.3.1, are very likely to significantly improve this situation in the near future. However, because of the limited scope of most Smart setting applications, they can be deployed modularly, and this challenge poses no immediate restriction on future innovation. Smart Cities in a broad sense offer many interesting research topics, but traffic and security aspects are likely to receive most attention in the coming years, as they can drastically improve the safety and quality of life in cities through intelligent management. Other topics, such as Smart Grids, are driven by the necessity for intelligent management because of rapidly changing energy grid conditions. In Smart Homes, an interesting topic is the discovery and integration of services and devices, and imbuing discovered devices with (partial, co-operative) AI. Such AI capable networks can then form a solid basis on which to run Smart Home applications that improve the security (e.g. intruder/weapon detection), health (e.g. fall detection, IoMT monitoring at home) and general quality of life of inhabitants. While there are many opportunities in IoV, the research potential in this area is likely to shift from roadside monitoring and traffic flow management in the fog, to inter-vehicle communication and self-organizing traffic flows as vehicles are outfitted with more powerful computational hardware. However,

roadside monitoring and city-wide traffic management in the fog will remain important practical topics, especially as self-organizing traffic will remain largely impossible until intelligent, autonomous vehicles outnumber the rest. Smart Healthcare features many highly specialized applications for the prevention or monitoring of diseases and conditions. As such, an interesting challenge is to create a general monitoring and alerting framework, with AI plugins for any number of conditions and diseases. Such a framework could be integrated into the sensory network of hospitals or Smart Homes, taking into account the different types of sensors and networks present in both settings. AI models designed for this architecture should be able to flexibly handle partial or missing sensory information. Smart applications in industrial settings are, more than in any other Smart setting, highly dependent on the situation. However, digital twins are an interesting research topic, especially in terms of automated discovery and digital representation of physical industrial settings, and the subsequent optimization. Finally, human interaction with Smart applications can be used to augment AI, creating Social Edge Intelligence (SEI) [175]. SEI can drastically improve applications in which AI is used to analyze gathered data, but in which some steps benefit from higher cognitive abilities than the state of the art currently offers.

5.5.7 Discussion

The use of AI in Smart applications and in the organization of the edge presents a rapidly advancing research field, with a great variety of opportunities. In this review, an introduction is given to the technologies required to understand the state of the art in Edge Intelligence (EI), and the concept of EI is elaborated using a taxonomy with "Enabling Technology", "Organization" and "Applications" as its main topics. Research trend data from 2015 to 2020 is gathered from Google Scholar for subdivisions of these topics, and presented to show both absolute and relative interest in each subtopic. The "Organization" aspect, being the main focus of this review, has a more fine-grained subdivision, explaining all contributing factors in detail. Related work is presented, comparing it to the work in this review, and for each subdivision of the taxonomy a number of selected studies are gathered to illustrate the state of the art as completely as possible at a high level. From the research trends and selected studies, a number of short-term challenges and high-level visions for EI are formulated, providing a basis for future work.

5.6 Summary

This chapter introduces the various types of AI, explaining their underlying concepts and suitable applications. Although not technically AI, the concept of a blockchain is also explained, as it is a main component of many state of the art Smart City studies related to AI. After this short introduction, the Intelligent Edge was introduced as the synergy of edge computing and AI, with a quick overview of its various areas of application.
Some ideas are presented on how SoSwirly can be improved by ANN, and how SoSwirly can in turn decentralize ANN learning by propagating weight updates. Related work for both these approaches is provided, along with a theoretical framework for their implementation and properties. However, both ideas may yet have some disadvantages that limit their use cases. Depending on the frameworks used, integrating an ANN into SoSwirly may significantly increase resource requirements (e.g. memory required by TensorFlow and Python themselves), making it feasible only on certain classes of edge devices, and constraining the rest to the default SoSwirly mechanisms. On the other hand, using SoSwirly to decentralize ANN weight updates depends on finding suitable use cases through mathematical models (e.g. reconciling weight updates from nodes with potentially diverging models), so any implementation will likely start off with limited use cases.
To conclude, a review of the state of the art of AI in the Intelligent Edge is performed, showing a rapidly growing interest in this convergence of technologies, and Smart Cities as a whole. A taxonomy is presented, and trends from 2015 to 2020 show an exponential growth of research interest in all categories, from enabling technology through the various aspects of organization, to all Smart City application domains. An inquiry into the types of AI used shows that while ANN are increasingly popular, other types of AI remain important, either because they are more suitable to a specific problem, or because they provide better performance on edge hardware. Finally, a selection of papers from each taxonomy topic illustrates the most recent progress from 2019 to 2021.

References

[1] *Is-ought problem*, October 2021. Available from: https://en.wikipedia.org/wiki/Is%E2%80%93ought_problem.

[2] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. *Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing*. Proceedings of the IEEE, 107(8):1738–1762, aug 2019. doi:10.1109/jproc.2019.2918951.

[3] V. D. Hunt. *Introduction to Artificial Intelligence and Expert Systems*. In Artificial Intelligence & Expert Systems Sourcebook, pages 1–39. Springer US, 1986. doi:10.1007/978-1-4613-2261-0_1.

[4] G. Brewka. *Artificial intelligence—a modern approach by Stuart Russell and Peter Norvig, Prentice Hall. Series in Artificial Intelligence, Englewood Cliffs, NJ.* The Knowledge Engineering Review, 11(1):78–79, mar 1996. doi:10.1017/s0269888900007724.

[5] A. I. Schein and L. H. Ungar. *Active learning for logistic regression: an evaluation*. Machine Learning, 68(3):235–265, aug 2007. doi:10.1007/s10994-007-5019-5.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *Overview of Supervised Learning*. In The Elements of Statistical Learning, pages 9–41. Springer New York, dec 2008. doi:10.1007/978-0-387-84858-7_2.

[7] S. Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv:http://arxiv.org/abs/1609.04747v2.

[8] M. L. Puterman. *Chapter 8 Markov decision processes*. In Handbooks in Operations Research and Management Science, pages 331–434. Elsevier, 1990. doi:10.1016/s0927-0507(05)80172-0.

[9] C. C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing, 2018. doi:10.1007/978-3-319-94463-0.

[10] S. Mirjalili. *Evolutionary Algorithms and Neural Networks*. Springer International Publishing, 2019. doi:10.1007/978-3-319-93025-1.

[11] L. Kallel, B. Naudts, and C. R. Reeves. *Properties of Fitness Functions and Search Landscapes*. In Natural Computing Series, pages 175–206. Springer Berlin Heidelberg, 2001. doi:10.1007/978-3-662-04448-3_8.

[12] L. L. Mitsuo Gen, Runwei Cheng. *Network Models and Optimization: Multiobjective Genetic Algorithm Approach.* Springer London, 2008. doi:10.1007/978-1-84800-181-7.

[13] T. Murata and H. Ishibuchi. *MOGA: multi-objective genetic algorithms.* In Proceedings of 1995 IEEE International Conference on Evolutionary Computation. IEEE, 1995. doi:10.1109/icec.1995.489161.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. *A fast and elitist multiobjective genetic algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation, 6(2):182–197, apr 2002. doi:10.1109/4235.996017.

[15] D. A. Van Veldhuizen and G. B. Lamont. *Evolutionary Computation and Convergence to a Pareto Front.* In J. R. Koza, editor, Late Breaking Papers at the Genetic Programming 1998 Conference, pages 221–228, 1998.

[16] B. Karlik and A. Olgac. *Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks.* In International Journal of Artificial Intelligence and Expert Systems (IJAE), 2011.

[17] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi. *Learning Activation Functions to Improve Deep Neural Networks.* arXiv:http://arxiv.org/abs/1412.6830v3.

[18] T. Hastie, R. Tibshirani, and J. Friedman. *Unsupervised Learning.* In The Elements of Statistical Learning, pages 485–585. Springer New York, dec 2008. doi:10.1007/978-0-387-84858-7_14.

[19] A. Pavelka and A. Procházka. *Algorithms for initialization of neural network weights.* In In Proceedings of the 12th Annual Conference, MATLAB, pages 453–459, 2004.

[20] C. Louizos, M. Welling, and D. P. Kingma. *Learning Sparse Neural Networks through L_0 Regularization.* arXiv:http://arxiv.org/abs/1712.01312v2.

[21] W. Zaremba, I. Sutskever, and O. Vinyals. *Recurrent Neural Network Regularization.* arXiv:http://arxiv.org/abs/1409.2329v5.

[22] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. *On the importance of initialization and momentum in deep learning.* In S. Dasgupta and D. McAllester, editors, Proceedings of the 30th International Conference on Machine Learning, volume 28 of *Proceedings of Machine*

Learning Research, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR. Available from: http://proceedings.mlr.press/v28/sutskever13.html.

[23] D. Goldfarb, Y. Ren, and A. Bahamou. *Practical Quasi-Newton Methods for Training Deep Neural Networks.* arXiv:http://arxiv.org/abs/2006.08877v3.

[24] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. *Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey.* Proceedings of the IEEE, 108(4):485–532, apr 2020. doi:10.1109/jproc.2020.2976475.

[25] S. Albawi, T. A. Mohammed, and S. Al-Zawi. *Understanding of a convolutional neural network.* In 2017 International Conference on Engineering and Technology (ICET). IEEE, aug 2017. doi:10.1109/icengtechnol.2017.8308186.

[26] L. Medsker and L. C. Jain, editors. *Recurrent Neural Networks.* CRC Press, dec 1999. doi:10.1201/9781420049176.

[27] A. Sherstinsky. *Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network.* Physica D: Nonlinear Phenomena, 404:132306, mar 2020. doi:10.1016/j.physd.2019.132306.

[28] M. Sewak. *Deep Reinforcement Learning.* Springer, 2019.

[29] D. P. Anderson. *BOINC: A Platform for Volunteer Computing.* Journal of Grid Computing, 18(1):99–122, nov 2019. doi:10.1007/s10723-019-09497-9.

[30] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith. *Federated Learning: Challenges, Methods, and Future Directions.* arXiv:http://arxiv.org/abs/1908.07873v1, doi:10.1109/MSP.2020.2975749.

[31] R. C. Eberhart, Y. Shi, and J. Kennedy. *Swarm Intelligence.* Elsevier Science & Techn., 2001.

[32] J. Golosova and A. Romanovs. *The Advantages and Disadvantages of the Blockchain Technology.* In 2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE). IEEE, nov 2018. doi:10.1109/aieee.2018.8592253.

[33] D. Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER.* 2014.

[34] J. Sedlmeir, H. U. Buhl, G. Fridgen, and R. Keller. *The Energy Consumption of Blockchain Technology: Beyond Myth.* Business & Information Systems Engineering, 62(6):599–608, jun 2020. doi:10.1007/s12599-020-00656-x.

[35] *Data protection in the EU*, February 2022. Available from: https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en.

[36] R. Sánchez-Corcuera, A. Nuñez-Marcos, J. Sesma-Solance, A. Bilbao-Jayo, R. Mulero, U. Zulaika, G. Azkune, and A. Almeida. *Smart cities survey: Technologies, application domains and challenges for the cities of the future.* International Journal of Distributed Sensor Networks, 15(6):155014771985398, jun 2019. doi:10.1177/1550147719853984.

[37] *IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing.* doi:10.1109/ieeestd.2018.8423800.

[38] IEEE. *P2805.1 - Self-Management Protocols for Edge Computing Node*, 2019. Available from: https://standards.ieee.org/project/2805_1.html.

[39] IEEE. *P2805.3 - Cloud-Edge Collaboration Protocols for Machine Learning*, 2019. Available from: https://standards.ieee.org/project/2805_3.html.

[40] IEEE. *P2961 - Guide for an Architectural Framework and Application for Collaborative Edge Computing*, 2021. Available from: https://standards.ieee.org/project/2961.html.

[41] IEEE. *P2979 - Standard for Edge Intelligent Terminal for Expressway Cooperative Transportation*, 2021. Available from: https://standards.ieee.org/project/2979.html.

[42] *TensorFlow Lite*, January 2022. Available from: https://www.tensorflow.org/lite/guide.

[43] N. Kukreja, A. Shilova, O. Beaumont, J. Huckelheim, N. Ferrier, P. Hovland, and G. Gorman. *Training on the Edge: The why and the how.* In 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 899–903. IEEE, 2019.

[44] M. Hausknecht and P. Stone. *Deep recurrent q-learning for partially observable mdps.* In 2015 aaai fall symposium series, 2015.

[45] B. Magableh and M. Almiani. *A deep recurrent Q network towards self-adapting distributed microservice architecture.* Software: Practice and Experience, 50(2):116–135, nov 2019. doi:10.1002/spe.2778.

[46] P. T. Yamak, L. Yujian, and P. K. Gadosey. *A Comparison between ARIMA, LSTM, and GRU for Time Series Forecasting.* In Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence. ACM, dec 2019. doi:10.1145/3377713.3377722.

[47] B. Jang, M. Kim, G. Harerimana, and J. W. Kim. *Q-learning algorithms: A comprehensive classification and applications.* IEEE Access, 7:133653–133667, 2019.

[48] K. Potdar, T. S. Pardawala, and C. D. Pai. *A comparative study of categorical variable encoding techniques for neural network classifiers.* International journal of computer applications, 175(4):7–9, 2017.

[49] Y. Tokuyama, Y. Fukushima, and T. Yokohira. *The Effect of Using Attribute Information in Network Traffic Prediction with Deep Learning.* In 2018 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, oct 2018. doi:10.1109/ictc.2018.8539488.

[50] C. W. Misner, K. S. Thorne, and J. A. Wheeler. *Gravitation.* Princeton Univers. Press, 2017.

[51] A. G. Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger. *Braintorrent: A peer-to-peer environment for decentralized federated learning.* arXiv preprint arXiv:1905.06731, 2019.

[52] A. Lalitha, S. Shekhar, T. Javidi, and F. Koushanfar. *Fully decentralized federated learning.* In Third workshop on Bayesian Deep Learning (NeurIPS), 2018.

[53] I. Hegedűs, G. Danner, and M. Jelasity. *Gossip learning as a decentralized alternative to federated learning.* In IFIP International Conference on Distributed Applications and Interoperable Systems, pages 74–90. Springer, 2019.

[54] R. Ormándi, I. Hegedűs, and M. Jelasity. *Gossip learning with linear models on fully distributed data.* Concurrency and Computation: Practice and Experience, 25(4):556–571, 2013.

[55] L. Rokach. *Ensemble-based classifiers.* Artificial Intelligence Review, 33(1-2):1–39, nov 2009. doi:10.1007/s10462-009-9124-7.

[56] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. *A survey of model compression and acceleration for deep neural networks.* arXiv preprint arXiv:1710.09282, 2017.

[57] T. Goethals, B. Volckaert, and F. D. 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.

[58] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya. *Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence.* IEEE Internet of Things Journal, 7(8):7457–7469, aug 2020. doi:10.1109/jiot.2020.2984887.

[59] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief. *Communication-Efficient Edge AI: Algorithms and Systems.* IEEE Communications Surveys & Tutorials, 22(4):2167–2191, 2020. doi:10.1109/comst.2020.3007787.

[60] J. Zhang and D. Tao. *Empowering Things with Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things.* IEEE Internet of Things Journal, pages 1–1, 2020. doi:10.1109/jiot.2020.3039359.

[61] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen. *Convergence of Edge Computing and Deep Learning: A Comprehensive Survey.* IEEE Communications Surveys & Tutorials, 22(2):869–904, 2020. doi:10.1109/comst.2020.2970550.

[62] M. P. Véstias. *A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing.* Algorithms, 12(8):154, jul 2019. doi:10.3390/a12080154.

[63] M. Véstias. *Processing Systems for Deep Learning Inference on Edge Devices.* In Internet of Things, pages 213–240. Springer International Publishing, 2020. doi:10.1007/978-3-030-44907-0_9.

[64] Z. Zou, Y. Jin, P. Nevalainen, Y. Huan, J. Heikkonen, and T. Westerlund. *Edge and Fog Computing Enabled AI for IoT-An Overview.* In 2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE, mar 2019. doi:10.1109/aicas.2019.8771621.

[65] A. Nazir, R. N. Mir, and S. Qureshi. *Exploring compression and parallelization techniques for distribution of deep neural networks over Edge–Fog continuum – a review.* International Journal of Intelligent Computing and Cybernetics, 13(3):331–364, jun 2020. doi:10.1108/ijicc-04-2020-0038.

[66] B. K. Mohanta, D. Jena, U. Satapathy, and S. Patnaik. *Survey on IoT security: Challenges and solution using machine learning, artificial intelligence and blockchain technology.* Internet of Things, 11:100227, sep 2020. doi:10.1016/j.iot.2020.100227.

[67] M. Rihan, M. Elwekeil, Y. Yang, L. Huang, C. Xu, and M. M. Selim. *Deep-VFog: When Artificial Intelligence Meets Fog Computing in V2X.* IEEE Systems Journal, pages 1–14, 2020. doi:10.1109/jsyst.2020.3009998.

[68] C.-X. Wang, M. D. Renzo, S. Stanczak, S. Wang, and E. G. Larsson. *Artificial Intelligence Enabled Wireless Networking for 5G and Beyond: Recent Advances and Future Challenges.* IEEE Wireless Communications, 27(1):16–23, feb 2020. doi:10.1109/mwc.001.1900292.

[69] R. Gupta, D. Reebadiya, and S. Tanwar. *6G-enabled Edge Intelligence for Ultra -Reliable Low Latency Applications: Vision and Mission.* Computer Standards & Interfaces, 77:103521, aug 2021. doi:10.1016/j.csi.2021.103521.

[70] J.-H. Huh and Y.-S. Seo. *Understanding Edge Computing: Engineering Evolution With Artificial Intelligence.* IEEE Access, 7:164229–164245, 2019. doi:10.1109/access.2019.2945338.

[71] Z. Ullah, F. Al-Turjman, L. Mostarda, and R. Gagliardi. *Applications of Artificial Intelligence and Machine learning in smart cities.* Computer Communications, 154:313–323, mar 2020. doi:10.1016/j.comcom.2020.02.069.

[72] G. M. Gilbert, S. Naiman, H. Kimaro, and B. Bagile. *A Critical Review of Edge and Fog Computing for Smart Grid Applications.* In IFIP Advances in Information and Communication Technology, pages 763–775. Springer International Publishing, 2019. doi:10.1007/978-3-030-18400-1_62.

[73] S. Sepasgozar, R. Karimi, L. Farahzadi, F. Moezzi, S. Shirowzhan, S. M. Ebrahimzadeh, F. Hui, and L. Aye. *A Systematic Content*

Review of Artificial Intelligence and the Internet of Things Applications in Smart Home. Applied Sciences, 10(9):3074, apr 2020. doi:10.3390/app10093074.

[74] L. Greco, G. Percannella, P. Ritrovato, F. Tortorella, and M. Vento. *Trends in IoT based solutions for health care: Moving AI to the edge.* Pattern Recognition Letters, 135:346–353, jul 2020. doi:10.1016/j.patrec.2020.05.016.

[75] A. Angelopoulos, E. T. Michailidis, N. Nomikos, P. Trakadas, A. Hatziefremidis, S. Voliotis, and T. Zahariadis. *Tackling Faults in the Industry 4.0 Era—A Survey of Machine-Learning Solutions and Key Aspects.* Sensors, 20(1):109, dec 2019. doi:10.3390/s20010109.

[76] S. Singh, P. K. Sharma, B. Yoon, M. Shojafar, G. H. Cho, and I.-H. Ra. *Convergence of blockchain and artificial intelligence in IoT network for the sustainable smart city.* Sustainable Cities and Society, 63:102364, dec 2020. doi:10.1016/j.scs.2020.102364.

[77] R. Yang, F. R. Yu, P. Si, Z. Yang, and Y. Zhang. *Integrated Blockchain and Edge Computing Systems: A Survey, Some Research Issues and Challenges.* IEEE Communications Surveys & Tutorials, 21(2):1508–1532, 2019. doi:10.1109/comst.2019.2894727.

[78] Y. Wu, H.-N. Dai, and H. Wang. *Convergence of Blockchain and Edge Computing for Secure and Scalable IIoT Critical Infrastructures in Industry 4.0.* IEEE Internet of Things Journal, 8(4):2300–2317, feb 2021. doi:10.1109/jiot.2020.3025916.

[79] D. C. Nguyen, M. Ding, Q.-V. Pham, P. N. Pathirana, L. B. Le, A. Seneviratne, J. Li, D. Niyato, and H. V. Poor. *Federated Learning Meets Blockchain in Edge Computing: Opportunities and Challenges.* IEEE Internet of Things Journal, 8(16):12806–12825, aug 2021. doi:10.1109/jiot.2021.3072611.

[80] L. Fiack, L. Rodriguez, and B. Miramond. *Hardware design of a neural processing unit for bio-inspired computing.* In 2015 IEEE 13th International New Circuits and Systems Conference (NEWCAS). IEEE, jun 2015. doi:10.1109/newcas.2015.7181997.

[81] A. Yazdanbakhsh, K. Seshadri, B. Akin, J. Laudon, and R. Narayanaswami. *An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks.* arXiv:http://arxiv.org/abs/2102.10423v1.

[82] K. Karras, E. Pallis, G. Mastorakis, Y. Nikoloudakis, J. M. Batalla, C. X. Mavromoustakis, and E. Markakis. *A Hardware Acceleration Platform for AI-Based Inference at the Edge.* Circuits, Systems, and Signal Processing, 39(2):1059–1070, aug 2019. doi:10.1007/s00034-019-01226-7.

[83] B. Kim, S. Lee, A. R. Trivedi, and W. J. Song. *Energy-Efficient Acceleration of Deep Neural Networks on Realtime-Constrained Embedded Edge Devices.* IEEE Access, 8:216259–216270, 2020. doi:10.1109/access.2020.3038908.

[84] E. Li, L. Zeng, Z. Zhou, and X. Chen. *Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing.* IEEE Transactions on Wireless Communications, 19(1):447–457, jan 2020. doi:10.1109/twc.2019.2946140.

[85] S. Dey, J. Mondal, and A. Mukherjee. *Offloaded Execution of Deep Learning Inference at Edge: Challenges and Insights.* In 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). IEEE, mar 2019. doi:10.1109/percomw.2019.8730817.

[86] W. Jiang and S. Lv. *Inference Acceleration Model of Branched Neural Network Based on Distributed Deployment in Fog Computing.* In Web Information Systems and Applications, pages 503–512. Springer International Publishing, 2020. doi:10.1007/978-3-030-60029-7_45.

[87] S. Dey, A. Mukherjee, A. Pal, and B. P. *Embedded Deep Inference in Practice.* In Proceedings of the 1st Workshop on Machine Learning on Edge in Sensor Systems - SenSys-ML 2019. ACM Press, 2019. doi:10.1145/3362743.3362964.

[88] T. Mohammed, C. Joe-Wong, R. Babbar, and M. D. Francesco. *Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading.* In IEEE INFOCOM 2020 - IEEE Conference on Computer Communications. IEEE, jul 2020. doi:10.1109/infocom41043.2020.9155237.

[89] L. Zhang, J. Wu, S. Mumtaz, J. Li, H. Gacanin, and J. J. P. C. Rodrigues. *Edge-to-Edge Cooperative Artificial Intelligence in Smart Cities with On-Demand Learning Offloading.* In 2019 IEEE Global Communications Conference (GLOBECOM). IEEE, dec 2019. doi:10.1109/globecom38437.2019.9013878.

[90] T. Zhang, Z. Shen, J. Jin, X. Zheng, A. Tagami, and X. Cao. *Achieving Democracy in Edge Intelligence: A Fog-Based Collaborative Learning Scheme*. IEEE Internet of Things Journal, 8(4):2751–2761, feb 2021. doi:10.1109/jiot.2020.3020911.

[91] T. Kaneko, K. Orimo, I. Hida, S. Takamaeda-Yamazaki, M. Ikebe, M. Motomura, and T. Asai. *A study on a low power optimization algorithm for an edge-AI device*. Nonlinear Theory and Its Applications, IEICE, 10(4):373–389, 2019. doi:10.1587/nolta.10.373.

[92] H. Siegelmann and E. Sontag. *On the Computational Power of Neural Nets*. Journal of Computer and System Sciences, 50(1):132–150, feb 1995. doi:10.1006/jcss.1995.1013.

[93] S. Liao, J. Wu, S. Mumtaz, J. Li, R. Morello, and M. Guizani. *Cognitive Balance for Fog Computing Resource in Internet of Things: An Edge Learning Approach*. IEEE Transactions on Mobile Computing, pages 1–1, 2020. doi:10.1109/tmc.2020.3026580.

[94] Y. Zhai, T. Bao, L. Zhu, M. Shen, X. Du, and M. Guizani. *Toward Reinforcement-Learning-Based Service Deployment of 5G Mobile Edge Computing with Request-Aware Scheduling*. IEEE Wireless Communications, 27(1):84–91, feb 2020. doi:10.1109/mwc.001.1900298.

[95] X. Liu, Z. Qin, and Y. Gao. *Resource Allocation for Edge Computing in IoT Networks via Reinforcement Learning*. In ICC 2019 - 2019 IEEE International Conference on Communications (ICC). IEEE, may 2019. doi:10.1109/icc.2019.8761385.

[96] G. Li, Y. Liu, J. Wu, D. Lin, and S. Zhao. *Methods of Resource Scheduling Based on Optimized Fuzzy Clustering in Fog Computing*. Sensors, 19(9):2122, may 2019. doi:10.3390/s19092122.

[97] S. Wang, T. Zhao, and S. Pang. *Task Scheduling Algorithm Based on Improved Firework Algorithm in Fog Computing*. IEEE Access, 8:32385–32394, 2020. doi:10.1109/access.2020.2973758.

[98] M. Abedi and M. Pourkiani. *Resource Allocation in Combined Fog-Cloud Scenarios by Using Artificial Intelligence*. In 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC). IEEE, apr 2020. doi:10.1109/fmec49853.2020.9144693.

[99] M. K. Pandit, R. N. Mir, and M. A. Chishti. *Adaptive task scheduling in IoT using reinforcement learning*. International Journal of Intelligent Computing and Cybernetics, 13(3):261–282, jun 2020. doi:10.1108/ijicc-03-2020-0021.

[100] S. C. G., V. Chamola, C.-K. Tham, G. S., and N. Ansari. *An optimal delay aware task assignment scheme for wireless SDN networked edge cloudlets.* Future Generation Computer Systems, 102:862–875, jan 2020. doi:10.1016/j.future.2019.09.003.

[101] S. Bian, X. Huang, and Z. Shao. *Online Task Scheduling for Fog Computing with Multi-Resource Fairness.* In 2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall). IEEE, sep 2019. doi:10.1109/vtcfall.2019.8891573.

[102] Y. Dong, S. Guo, J. Liu, and Y. Yang. *Energy-Efficient Fair Cooperation Fog Computing in Mobile Edge Networks for Smart City.* IEEE Internet of Things Journal, 6(5):7543–7554, oct 2019. doi:10.1109/jiot.2019.2901532.

[103] W. Yang, Y. Zhang, W. Y. B. Lim, Z. Xiong, Y. Jiao, and J. Jin. *Privacy is not Free: Energy-Aware Federated Learning for Mobile and Edge Intelligence.* In 2020 International Conference on Wireless Communications and Signal Processing (WCSP). IEEE, oct 2020. doi:10.1109/wcsp49889.2020.9299703.

[104] N. Janbi, I. Katib, A. Albeshri, and R. Mehmood. *Distributed Artificial Intelligence-as-a-Service (DAIaaS) for Smarter IoE and 6G Environments.* Sensors, 20(20):5796, oct 2020. doi:10.3390/s20205796.

[105] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya. *FogBus: A Blockchain-based Lightweight Framework for Edge and Fog Computing.* Journal of Systems and Software, 154:22–36, aug 2019. doi:10.1016/j.jss.2019.04.050.

[106] H. Li, K. Ota, and M. Dong. *Deep Reinforcement Scheduling for Mobile Crowdsensing in Fog Computing.* ACM Transactions on Internet Technology, 19(2):1–18, apr 2019. doi:10.1145/3234463.

[107] P. Kochovski, R. Sakellariou, M. Bajec, P. Drobintsev, and V. Stankovski. *An Architecture and Stochastic Method for Database Container Placement in the Edge-Fog-Cloud Continuum.* In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, may 2019. doi:10.1109/ipdps.2019.00050.

[108] O. Debauche, S. Mahmoudi, S. A. Mahmoudi, P. Manneback, and F. Lebeau. *A new Edge Architecture for AI-IoT services deployment.* Procedia Computer Science, 175:10–19, 2020. doi:10.1016/j.procs.2020.07.006.

[109] J. Zhou, Y. Wang, K. Ota, and M. Dong. *AAIoT: Accelerating Artificial Intelligence in IoT Systems.* IEEE Wireless Communications Letters, 8(3):825–828, jun 2019. doi:10.1109/lwc.2019.2894703.

[110] Y. Xiao, G. Shi, Y. Li, W. Saad, and H. V. Poor. *Toward Self-Learning Edge Intelligence in 6G.* IEEE Communications Magazine, 58(12):34–40, dec 2020. doi:10.1109/mcom.001.2000388.

[111] S. Xu, Z. Zhang, M. Kadoch, and M. Cheriet. *A collaborative cloud-edge computing framework in distributed neural network.* EURASIP Journal on Wireless Communications and Networking, 2020(1), oct 2020. doi:10.1186/s13638-020-01794-2.

[112] W. Y. B. Lim, J. S. Ng, Z. Xiong, J. Jin, Y. Zhang, D. Niyato, C. Leung, and C. Miao. *Decentralized Edge Intelligence: A Dynamic Resource Allocation Framework for Hierarchical Federated Learning.* IEEE Transactions on Parallel and Distributed Systems, 33(3):536–550, mar 2022. doi:10.1109/tpds.2021.3096076.

[113] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito. *Pushing Intelligence to the Edge with a Stream Processing Architecture.* In 2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, jun 2017. doi:10.1109/ithings-greencom-cpscom-smartdata.2017.121.

[114] A. da Silva Veith, M. D. de Assuncao, and L. Lefevre. *Latency-Aware Strategies for Deploying Data Stream Processing Applications on Large Cloud-Edge Infrastructure.* IEEE Transactions on Cloud Computing, pages 1–1, 2021. doi:10.1109/tcc.2021.3097879.

[115] E. R. de Oliveira, F. Delicato, A. R. da Rocha, and M. Mattoso. *A Real-time and Energy-aware Framework for Data Stream Processing in the Internet of Things.* In Proceedings of the 6th International Conference on Internet of Things, Big Data and Security. SCITEPRESS - Science and Technology Publications, 2021. doi:10.5220/0010370100170028.

[116] I. Gharbi, K. Barkaoui, and B. A. Samir. *An Intelligent Agent-Based Industrial IoT Framework for Time-Critical Data Stream Processing.* In Mobile, Secure, and Programmable Networking, pages 195–208. Springer International Publishing, 2021. doi:10.1007/978-3-030-67550-9_13.

[117] B. Sonkoly, D. Haja, B. Németh, M. Szalay, J. Czentye, R. Szabó, R. Ullah, B.-S. Kim, and L. Toka. *Scalable edge cloud platforms for IoT services.* Journal of Network and Computer Applications, 170:102785, nov 2020. doi:10.1016/j.jnca.2020.102785.

[118] M. A. Rahman, A. T. Asyhari, L. Leong, G. Satrya, M. H. Tao, and M. Zolkipli. *Scalable machine learning-based intrusion detection system for IoT-enabled smart cities.* Sustainable Cities and Society, 61:102324, oct 2020. doi:10.1016/j.scs.2020.102324.

[119] A. A. Alli and M. M. Alam. *SecOFF-FCIoT: Machine learning based secure offloading in Fog-Cloud of things for smart city applications.* Internet of Things, 7:100070, sep 2019. doi:10.1016/j.iot.2019.100070.

[120] F. Concone, G. L. Re, and M. Morana. *SMCP: a Secure Mobile Crowdsensing Protocol for fog-based applications.* Human-centric Computing and Information Sciences, 10(1), jul 2020. doi:10.1186/s13673-020-00232-y.

[121] K. Yang, H. Ma, and S. Dou. *Fog Intelligence for Network Anomaly Detection.* IEEE Network, 34(2):78–82, mar 2020. doi:10.1109/mnet.001.1900156.

[122] Y. Jararweh, S. Otoum, and I. A. Ridhawi. *Trustworthy and sustainable smart city services at the edge.* Sustainable Cities and Society, 62:102394, nov 2020. doi:10.1016/j.scs.2020.102394.

[123] S. Xu, Y. Qian, and R. Q. Hu. *Data-Driven Network Intelligence for Anomaly Detection.* IEEE Network, 33(3):88–95, may 2019. doi:10.1109/mnet.2019.1800358.

[124] T. G. Nguyen, T. V. Phan, B. T. Nguyen, C. So-In, Z. A. Baig, and S. Sanguanpong. *SeArch: A Collaborative and Intelligent NIDS Architecture for SDN-Based Cloud IoT Networks.* IEEE Access, 7:107678–107694, 2019. doi:10.1109/access.2019.2932438.

[125] K. N. Qureshi, G. Jeon, and F. Piccialli. *Anomaly detection and trust authority in artificial intelligence and cloud computing.* Computer Networks, 184:107647, jan 2021. doi:10.1016/j.comnet.2020.107647.

[126] M. S. Zareen, S. Tahir, M. Akhlaq, and B. Aslam. *Artificial Intelligence/ Machine Learning in IoT for Authentication and Authorization of Edge Devices.* In 2019 International Conference on Applied and Engineering Mathematics (ICAEM). IEEE, aug 2019. doi:10.1109/icaem.2019.8853780.

[127] G. Li, K. Ota, M. Dong, J. Wu, and J. Li. *DeSVig: Decentralized Swift Vigilance Against Adversarial Attacks in Industrial Artificial Intelligence Systems.* IEEE Transactions on Industrial Informatics, 16(5):3267–3277, may 2020. doi:10.1109/tii.2019.2951766.

[128] C. Zhou, Q. Liu, and R. Zeng. *Novel Defense Schemes for Artificial Intelligence Deployed in Edge Computing Environment.* Wireless Communications and Mobile Computing, 2020:1–20, aug 2020. doi:10.1155/2020/8832697.

[129] T. N. Gia, A. Nawaz, J. P. Querata, H. Tenhunen, and T. Westerlund. *Artificial Intelligence at the Edge in the Blockchain of Things.* In Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 267–280. Springer International Publishing, 2020. doi:10.1007/978-3-030-49289-2_21.

[130] A. Nawaz, T. N. Gia, J. P. Queralta, and T. Westerlund. *Edge AI and Blockchain for Privacy-Critical and Data-Sensitive Applications.* In 2019 Twelfth International Conference on Mobile Computing and Ubiquitous Network (ICMU). IEEE, nov 2019. doi:10.23919/icmu48249.2019.9006635.

[131] P. Kochovski, S. Gec, V. Stankovski, M. Bajec, and P. D. Drobintsev. *Trust management in a blockchain based fog computing platform with trustless smart oracles.* Future Generation Computer Systems, 101:747–759, dec 2019. doi:10.1016/j.future.2019.07.030.

[132] S. Rathore, B. W. Kwon, and J. H. Park. *BlockSecIoTNet: Blockchain-based decentralized security architecture for IoT network.* Journal of Network and Computer Applications, 143:167–177, oct 2019. doi:10.1016/j.jnca.2019.06.019.

[133] J. Gao, K. O.-B. O. Agyekum, E. B. Sifah, K. N. Acheampong, Q. Xia, X. Du, M. Guizani, and H. Xia. *A Blockchain-SDN-Enabled Internet of Vehicles Environment for Fog Computing and 5G Networks.* IEEE Internet of Things Journal, 7(5):4278–4291, may 2020. doi:10.1109/jiot.2019.2956241.

[134] L. Russell, F. Kwamena, and R. Goubran. *Towards Reliable IoT: Fog-Based AI Sensor Validation.* In 2019 IEEE Cloud Summit. IEEE, aug 2019. doi:10.1109/cloudsummit47114.2019.00013.

[135] A. Yousefpour, S. Devic, B. Q. Nguyen, A. Kreidieh, A. Liao, A. M. Bayen, and J. P. Jue. *Guardians of the Deep Fog.* In Proceedings of the First International Workshop on Challenges in Artificial Intelligence

and Machine Learning for Internet of Things - AIChallengeIoT'19. ACM Press, 2019. doi:10.1145/3363347.3363366.

[136] S. Zhou, Y. Sun, Z. Jiang, and Z. Niu. *Exploiting Moving Intelligence: Delay-Optimized Computation Offloading in Vehicular Fog Networks.* IEEE Communications Magazine, 57(5):49–55, may 2019. doi:10.1109/mcom.2019.1800230.

[137] P. Radanliev, D. D. Roure, K. Page, M. V. Kleek, O. Santos, L. Maddox, P. Burnap, E. Anthi, and C. Maple. *Design of a dynamic and self-adapting system, supported with artificial intelligence, machine learning and real-time intelligence for predictive cyber risk analytics in extreme environments – cyber risk in the colonisation of Mars.* Safety in Extreme Environments, feb 2021. doi:10.1007/s42797-021-00025-1.

[138] J. An, W. Li, F. L. Gall, E. Kovac, J. Kim, T. Taleb, and J. Song. *EiF: Toward an Elastic IoT Fog Framework for AI Services.* IEEE Communications Magazine, 57(5):28–33, may 2019. doi:10.1109/mcom.2019.1800215.

[139] W. Xia, X. Zhang, G. Zheng, J. Zhang, S. Jin, and H. Zhu. *The interplay between artificial intelligence and fog radio access networks.* China Communications, 17(8):1–13, aug 2020. doi:10.23919/jcc.2020.08.001.

[140] K. Yang, Y. Shi, W. Yu, and Z. Ding. *Energy-Efficient Processing and Robust Wireless Cooperative Transmission for Edge Inference.* IEEE Internet of Things Journal, 7(10):9456–9470, oct 2020. doi:10.1109/jiot.2020.2979523.

[141] B. Hussain, Q. Du, A. Imran, and M. A. Imran. *Artificial Intelligence-Powered Mobile Edge Computing-Based Anomaly Detection in Cellular Networks.* IEEE Transactions on Industrial Informatics, 16(8):4986–4996, aug 2020. doi:10.1109/tii.2019.2953201.

[142] B. Yuan, J. Panneerselvam, L. Liu, N. Antonopoulos, and Y. Lu. *An Inductive Content-Augmented Network Embedding Model for Edge Artificial Intelligence.* IEEE Transactions on Industrial Informatics, 15(7):4295–4305, jul 2019. doi:10.1109/tii.2019.2902877.

[143] C. Goutte and E. Gaussier. *A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation.* In Lecture Notes in Computer Science, pages 345–359. Springer Berlin Heidelberg, 2005. doi:10.1007/978-3-540-31865-1_25.

[144] I. Bedhief, L. Foschini, P. Bellavista, M. Kassar, and T. Aguili. *Toward Self-Adaptive Software Defined Fog Networking Architecture for IIoT and Industry 4.0.* In 2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD). IEEE, sep 2019. doi:10.1109/camad.2019.8858499.

[145] J. Li, J. Cai, F. Khan, A. U. Rehman, V. Balasubramaniam, J. Sun, and P. Venu. *A Secured Framework for SDN-Based Edge Computing in IoT-Enabled Healthcare System.* IEEE Access, 8:135479–135490, 2020. doi:10.1109/access.2020.3011503.

[146] J. A. and. *Enhanced Smart Edge Broker using Fog Computing for Smart Homes.* International Journal of Artificial Intelligence and Applications for Smart Devices, 7(1):1–6, nov 2019. doi:10.21742/ijaiasd.2019.7.1.01.

[147] H. Yang, A. Alphones, Z. Xiong, D. Niyato, J. Zhao, and K. Wu. *Artificial-Intelligence-Enabled Intelligent 6G Networks.* IEEE Network, 34(6):272–280, nov 2020. doi:10.1109/mnet.011.2000195.

[148] Y.-Y. Chen, Y.-H. Lin, C.-C. Kung, M.-H. Chung, and I.-H. Yen. *Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in Demand-Side Management for Smart Homes.* Sensors, 19(9):2047, may 2019. doi:10.3390/s19092047.

[149] J. C. Olivares-Rojas, E. Reyes-Archundia, N. E. Rodriiguez-Maya, J. A. Gutierrez-Gnecchi, I. Molina-Moreno, and J. Cerda-Jacobo. *Machine Learning Model for the Detection of Electric Energy Fraud using an Edge-Fog Computing Architecture.* In 2020 IEEE International Conference on Engineering Veracruz (ICEV). IEEE, oct 2020. doi:10.1109/icev50249.2020.9289669.

[150] R. Ke, Y. Zhuang, Z. Pu, and Y. Wang. *A Smart, Efficient, and Reliable Parking Surveillance System With Edge Artificial Intelligence on IoT Devices.* IEEE Transactions on Intelligent Transportation Systems, pages 1–13, 2020. doi:10.1109/tits.2020.2984197.

[151] V. Mittal, A. Tyagi, and B. Bhushan. *Smart Surveillance Systems with Edge Intelligence: Convergence of Deep Learning and Edge Computing.* SSRN Electronic Journal, 2020. doi:10.2139/ssrn.3599865.

[152] T. Sultana and K. A. Wahid. *IoT-Guard: Event-Driven Fog-Based Video Surveillance System for Real-Time Security Management.* IEEE Access, 7:134881–134894, 2019. doi:10.1109/access.2019.2941978.

[153] G. P. R. Filho, R. I. Meneguette, G. Maia, G. Pessin, V. P. Gonçalves, L. Weigang, J. Ueyama, and L. A. Villas. *A fog-enabled smart home solution for decision-making using smart objects.* Future Generation Computer Systems, 103:18–27, feb 2020. doi:10.1016/j.future.2019.09.045.

[154] M. Bhatia. *Fog Computing-inspired Smart Home Framework for Predictive Veterinary Healthcare.* Microprocessors and Microsystems, 78:103227, oct 2020. doi:10.1016/j.micpro.2020.103227.

[155] J. P. Queralta, T. N. Gia, H. Tenhunen, and T. Westerlund. *Edge-AI in LoRa-based Health Monitoring: Fall Detection System with Fog Computing and LSTM Recurrent Neural Networks.* In 2019 42nd International Conference on Telecommunications and Signal Processing (TSP). IEEE, jul 2019. doi:10.1109/tsp.2019.8768883.

[156] B. Farahani, M. Barzegari, F. S. Aliee, and K. A. Shaik. *Towards collaborative intelligent IoT eHealth: From device to fog, and cloud.* Microprocessors and Microsystems, 72:102938, feb 2020. doi:10.1016/j.micpro.2019.102938.

[157] Y. Qiu, H. Zhang, and K. Long. *Computation Offloading and Wireless Resource Management for Healthcare Monitoring in Fog-Computing based Internet of Medical Things.* IEEE Internet of Things Journal, pages 1–1, 2021. doi:10.1109/jiot.2021.3066604.

[158] Z. Yu, S. U. Amin, M. Alhussein, and Z. Lv. *Research on Disease Prediction Based on Improved DeepFM and IoMT.* IEEE Access, 9:39043–39054, 2021. doi:10.1109/access.2021.3062687.

[159] J. Queiroz, P. Leitao, J. Barbosa, and E. Oliveira. *Distributing Intelligence among Cloud, Fog and Edge in Industrial Cyber-physical Systems.* In Proceedings of the 16th International Conference on Informatics in Control, Automation and Robotics. SCITEPRESS - Science and Technology Publications, 2019. doi:10.5220/0007979404470454.

[160] X. Li, J. Wan, H.-N. Dai, M. Imran, M. Xia, and A. Celesti. *A Hybrid Computing Solution and Resource Scheduling Strategy for Edge Computing in Smart Manufacturing.* IEEE Transactions on Industrial Informatics, 15(7):4225–4234, jul 2019. doi:10.1109/tii.2019.2899679.

[161] Q. Qi and F. Tao. *A Smart Manufacturing Service System Based on Edge Computing, Fog Computing, and Cloud Computing.* IEEE Access, 7:86769–86777, 2019. doi:10.1109/access.2019.2923610.

[162] T. Kumar, E. Harjula, M. Ejaz, A. Manzoor, P. Porambage, I. Ahmad, M. Liyanage, A. Braeken, and M. Ylianttila. *BlockEdge: Blockchain-Edge Framework for Industrial IoT Networks.* IEEE Access, 8:154166–154185, 2020. doi:10.1109/access.2020.3017891.

[163] Z. Ádam Mann and S. Schulte. *Fogsy: Towards Holistic Industrial AI Management in Fog and Edge Environments.* 2020. doi:10.34726/KUVS2020.

[164] P. Moens, V. Bracke, C. Soete, S. V. Hautte, D. N. Avendano, T. Ooijevaar, S. Devos, B. Volckaert, and S. V. Hoecke. *Scalable Fleet Monitoring and Visualization for Smart Machine Maintenance and Industrial IoT Applications.* Sensors, 20(15):4308, aug 2020. doi:10.3390/s20154308.

[165] K. Muhammad, S. Khan, V. Palade, I. Mehmood, and V. H. C. de Albuquerque. *Edge Intelligence-Assisted Smoke Detection in Foggy Surveillance Environments.* IEEE Transactions on Industrial Informatics, 16(2):1067–1075, feb 2020. doi:10.1109/tii.2019.2915592.

[166] O. Debauche, S. Mahmoudi, S. A. Mahmoudi, P. Manneback, J. Bindelle, and F. Lebeau. *Edge Computing and Artificial Intelligence for Real-time Poultry Monitoring.* Procedia Computer Science, 175:534–541, 2020. doi:10.1016/j.procs.2020.07.076.

[167] A. H. Sodhro, G. H. Sodhro, M. Guizani, S. Pirbhulal, and A. Boukerche. *AI-Enabled Reliable Channel Modeling Architecture for Fog Computing Vehicular Networks.* IEEE Wireless Communications, 27(2):14–21, apr 2020. doi:10.1109/mwc.001.1900311.

[168] L. T. Tan, R. Q. Hu, and L. Hanzo. *Twin-Timescale Artificial Intelligence Aided Mobility-Aware Edge Caching and Computing in Vehicular Networks.* IEEE Transactions on Vehicular Technology, 68(4):3086–3099, apr 2019. doi:10.1109/tvt.2019.2893898.

[169] M. Abbasi, M. Yaghoobikia, M. Rafiee, A. Jolfaei, and M. R. Khosravi. *Energy-efficient workload allocation in fog-cloud based services of intelligent transportation systems using a learning classifier system.* IET Intelligent Transport Systems, 14(11):1484–1490, nov 2020. doi:10.1049/iet-its.2019.0783.

[170] A. Seal, S. Bhattacharya, and A. Mukherjee. *Fog Computing for Real-Time Accident Identification and Related Congestion Control.* In 2019 IEEE International Systems Conference (SysCon). IEEE, apr 2019. doi:10.1109/syscon.2019.8836965.

[171] D. Peak and M. Azadmanesh. *Centralization/decentralization cycles in computing: Market evidence.* Information & Management, 31(6):303–317, jan 1997. doi:10.1016/s0378-7206(97)00002-5.

[172] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. *Edge-centric Computing.* ACM SIGCOMM Computer Communication Review, 45(5):37–42, sep 2015. doi:10.1145/2831347.2831354.

[173] V. Araujo, K. Mitra, S. Saguna, and C. Åhlund. *Performance evaluation of FIWARE: A cloud-based IoT platform for smart cities.* Journal of Parallel and Distributed Computing, 132:250–261, oct 2019. doi:10.1016/j.jpdc.2018.12.010.

[174] V. Bracke, M. Sebrechts, B. Moons, J. Hoebeke, F. D. Turck, and B. Volckaert. *Design and evaluation of a scalable Internet of Things backend for smart ports.* Software: Practice and Experience, apr 2021. doi:10.1002/spe.2973.

[175] D. Wang, D. Zhang, Y. Zhang, M. T. Rashid, L. Shang, and N. Wei. *Social Edge Intelligence: Integrating Human and Artificial Intelligence at the Edge.* In 2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI). IEEE, dec 2019. doi:10.1109/cogmi48466.2019.00036.

6 Overall Conclusion

Thank You And Goodnight - Culture GCU, Iain M. Banks

In Chapter 1, some high level topics were introduced, along with three topics derived from the title, specifically "Resource Efficiency and Flexibility", "Scaling Towards Smart Cities", and "Intelligence in Smart Cities", each of which is an important driving factor in the field of Smart Cities. An overview was presented of the publications written during the realization of this dissertation, some of which have been adapted and extended as content.

6.1 Resource Efficiency and Flexibility

Chapter 2 showed that virtualization is an important enabler for reliable and flexible software systems, discussing its use in both software processes and networks. A performance comparison of unikernels and containers was provided, which although slightly aged, illustrates their nuances and use cases. As these technologies continue to evolve, and alternatives regularly emerge, monitoring their strengths, relevance and relative performance remains an important topic for both researchers and software engineers.
The second part of Chapter 2 discussed network virtualization in terms of NFV and VPN. Although VPN are a slightly older technology, and NFV allows the management and evaluation of specific aspects of network virtualization, the collection of features presented by a VPN is still useful to securely federate private networks, or to overcome connectivity and security

issues when including edge nodes in a computational cluster. To that end, several VPN alternatives were benchmarked in the context of containerized micro-services, showing that WireGuard is the best option in most cases. However, future work in this field is more likely to fully pursue the path of NFV and SDN, separating the functions of service meshes, overlay networks, encryption, and monitoring for increased flexibility.
The fog and edge were properly introduced in Chapter 3, which also showed how virtualization transforms geographically distributed networks with a large, heterogeneous collection of devices into a uniform environment for software to run on. Introducing the concept of container orchestration, several recent platforms for edge orchestration were discussed and compared to FLEDGE. The construction of FLEDGE explored the various aspects of designing a resource-efficient container engine for edge devices, and the evaluations showed that it requires significantly less resources than alternatives, at the cost of some (cloud-oriented) functionality. However, follow-up evaluations seem to indicate that rather than optimizing for efficiency, most orchestrators tend to increase their resource use as they keep adding features, as newer, more edge-focused alternatives, take their place. To date, all the evaluated orchestrators are container-based, despite the requirement of a suitable Linux kernel and a container runtime to deploy containers, both of which continue to be impossible for many extremely low-resource devices to run. While there are options to enable microVM (e.g. unikernels, Firecracker) deployment in Kubernetes, a true virtualization agnostic orchestrator does not yet seem to exist.
Chapter 3 also introduced Swirly, a highly scalable real-time service scheduler. Whereas most orchestrators are designed for hundreds or thousands of nodes, Swirly was explicitly designed to handle immense service architectures in edge networks, enabling the real-time organization of services for up to hundreds of thousands of nodes. However, despite requiring minimal resources from each individual edge node, its Kubernetes-integrated centralized scheduler eventually saturates all available memory and network bandwidth on a single machine. Some strategies to avert this scenario are discussed, but the preferable solution was to decentralize the algorithm.

6.2 Scaling Towards Smart Cities

In Chapter 4, various types of scalability were discussed, along with their potential benefits. While FLEDGE and Swirly implicitly improve local scaling due to a focus on low resource requirements, the real goal of scalability in the edge is to run micro-services on many devices simultaneously as they are needed, rather than better vertical (local) scaling.

As an example of emergent functionality through federation, FUSE was introduced as a framework to quickly and securely federate private networks to pool computational resources in crisis situations. By relying on Docker-in-Docker and a micro-service approach, it can create a federation within minutes, and new resources can join in less than a minute, while using a VPN to create a secure separation between federated and non-federated resources in each individual network. However, this approach is quite resource intensive, and the master nodes can not reliably run on any current generation edge hardware. Although this is not an issue in private corporate networks, which generally contain at least a few powerful servers, the resource requirements and join times are not quite suitable for applications in the edge that need real-time responsiveness.
To that end, SoSwirly was presented as a decentralized, real-time, highly scalable service orchestrator. By decentralizing the original Swirly algorithm into a micro-service architecture, each edge node is responsible for mapping its own environment, and requesting services from suitable nearby nodes. As the performance of this approach only depends on configuration and the local density of nodes, it is shown to be highly scalable, and unhindered by the geographical expansion of networks. Combined with FLEDGE, the resource requirements of this solution are sufficiently low for edge gateways and many other devices. However, the solution is still reactive with respect to topology changes, and the integration of AI may further improve topology efficiency, depending on the performance impact of integrating suitable AI.

6.3 Intelligence in Smart Cities

Having presented and illustrated the importance of scalability and reliability in the edge in Chapters 2 through 4, the aspect of intelligence was introduced in Chapter 5. Common types of AI in the edge were elaborated, along with specific uses and how the synergy of intelligence and edge networks results in the Intelligent Edge.
Early stage work was introduced based on the combination of AI and SoSwirly, to further improve the effectiveness of SoSwirly, and to leverage it as a distributed weight update mechanism. While there are still open questions concerning the distributed weight update model, and whether efficiency improvements are worth the added load of an AI component, these ideas form a solid basis for future work.
Finally, the integration of the main topics of this dissertation was summarized in a review of AI in the Intelligent Edge and Smart Cities. This review shows an exponential increase in research interest in all aspects of AI in the

edge and Smart Cities from 2015 to 2020, with no immediate indication of slowing down, and while recent studies indicate that ANN and blockchain-based solutions are especially popular, other types of AI remain important, especially in edge scenarios with limited resources.

6.4 Future Challenges

While every chapter lists concrete ideas for extending the research presented in it, there are a number of trends and topics in the field of Smart Cities that merit attention.

Virtualization and orchestration
In Chapters 2 and 3, it was determined from updated information that the number of alternatives available in lightweight virtualization and edge container orchestration is evolving quickly. While older solutions tend to incorporate more functionality and grow out of their original target devices, alternative, highly targeted solutions are usually developed to take their place and solve some of their shortcomings. Monitoring the scope and benchmarking the performance of all these solutions will remain academically interesting for at least several more years, as edge computing continues to mature.
Whereas Docker containers themselves are relatively mature, upcoming technologies can be divided into three categories, each with their advantages and potential target devices in the edge:

- Running one or more containers in a microVM adds the security and isolation of a VM to the existing orchestration and networking potential of containers. Such architectures are relatively resource intensive and are highly suitable for fog servers or powerful edge gateway devices. An example of this category is Kata Containers [1].

- Executing a single process directly inside a microVM, which limits the attack surface of a software process and potentially improves performance. Depending on which hypervisors are supported (i.e. bare-metal vs Linux) and how the target process is integrated, these solutions may be suitable for resource-constrained edge devices. Examples include OSv unikernels [2] and Firecracker microVMs [3].

- Customized system interfaces are essentially a substantial layer of indirection at the kernel level, which can provide security benefits for the host kernel, and performance benefits for any process running on them. Solutions in this category can be highly experimental, ranging

from WASI [4], which is shown to be extremely fast but does not yet have any networking capabilities, to gVisor [3], which is more stable and merely aims to separate containers from their host kernel.

Novel methods of organizing software services in the edge should take into account the various feasible technologies, rather than containers alone. To that end, contributions to the upcoming technologies themselves could be researched. For example, the networking capabilities of microVM-based solutions are often highly compatible with CNI, and a novel CNI plugin or container runtime could transparently handle container, unikernel, and Firecracker deployments. WASI is an important development, as it does not rely on either a container runtime nor a hypervisor, and a basic socket implementation would allow further research into its performance for micro-services, as well as possible integration into service architectures and container networks.
Additionally, it is important to determine which factors and device properties determine the best technology to use for service deployment on a specific device. These factors are no longer limited to basic hardware resources, as environmental factors (e.g. physical location, nearby services and devices) and specialized hardware acceleration may be more important. It would also be useful to determine how software images can transparently support various virtualization technologies, similar to how Docker images can support multiple CPU architectures.
Finally, many low-grade sensors and IoT devices do not currently possess the resources for any type of virtualization, much less containers. Such devices usually only support a basic dialect of C or C++, but synergetic research involving a focus on truly minimal hardware and a primitive form of isolated, flexible software deployment could finally open up this class of hardware for limited pre-processing of data, or limited but immediate actuation.

Decentralization
Cloud computing has always been an inherently centralized concept, in which scaling is mostly limited to selecting optimal nodes and throwing more hardware resources at a problem. However, scalability is fundamentally different in edge networks, and by extension Smart Cities, which contain a variety of low-resource devices spread out over large geographical areas, and where networks and nodes are constantly in flux. Whereas network technology is already quite suitable for decentralized operation, service orchestration and Smart City applications are usually confined to small networks and limited scopes. For a true Smart City to develop, in which highly responsive services are transparently available wherever one goes,

more integration is required between various services and networks, so that applications are no longer features of single buildings or industrial yards. SoSwirly in Chapter 4 is a limited example of how entire service architectures can result from the emergent behavior of a collection of nodes, each with their own needs. However, much more remains to be done in terms of uniform runtime environments, decentralized discovery of functions and services, reliability and back-up service providers, pre-empting end-user needs, and predicting the short-term functional needs of any node through AI.

In terms of uniform runtime environments and decentralized discovery, further (IEEE) standardization is required to achieve a global level of cooperation between various solutions, much like OCI [5] has achieved for containers. SoSwirly currently only picks a single node for each service, and can make no guarantee for QoS in case of unexpected software or hardware failures. Additional research is required to determine how redundancy and timely failure detection can be implemented in decentralized architectures for a guaranteed QoS, and how the overhead of the resulting strategies can be minimized. Similarly, although the point of decentralization is to preserve privacy by not sending data to the cloud, additional research could be done into decentralized orchestration with hard limits on data availability (required by either the user or the law, i.e. GDPR [6]), and by extension, no option to move or possibly even store the data.

From an architectural point of view, SoSwirly can technically organize any service architecture that can be expressed as a Directed Acyclic Graph (DAG) through recursive deployment of agents. However, it may be useful to define service architectures in terms of functionality dependencies, rather than service dependencies. Combined with standardization efforts, this may result in devices exposing well-defined functionality, rather than software services with vendor-specific requirements or quirks, which in turn would turn the edge into a network of easily reusable building blocks. This behavior is generally already implicitly present in applications designed around a message bus or event bus, but such architectures are usually controlled by a centralized instance and limited in scope.

Intelligence

As shown in Chapter 5, there are many interesting ideas and potential applications around the topic of Smart Cities, but edge hardware is not always quite capable of running intensive workloads and complex AI models. The contradictory requirements of extremely small, cheap, low-powered and yet highly performant devices make hardware progress steady, but difficult. Flexible software and AI models can partially solve these difficulties by only enabling specific functionality as the required hardware is detected, or in

the case of ANN, by sacrificing accuracy for speed.

Innovations in GPU, TPU and FPGA development are likely to increase the size of feasible and trainable edge models by orders of magnitude in the near future. An Nvidia Jetson Nano [7] is relatively cheap, and comparable in power consumption to a Raspberry Pi 4, yet its GPU is capable of real-time inference of models with tens of thousands of parameters. As GPU industry predictions [8] indicate far faster performance increases in the context of neural networks than the legendary leaps predicted for CPUs by Moore's law, an era of proliferate Deep Learning on edge devices is at hand.

In any case, the use of AI in edge organization and applications is only just beginning, and will undoubtedly prove useful in workload orchestration, data placement, anticipating end-user needs, and managing network traffic as classical, hand-crafted algorithms become intractable or incapable of taking into account all necessary factors. Decentralized learning is an important enabler for running DNN entirely in the edge, but as Chapter 5 shows, entirely decentralized (online) learning introduces significant issues that need to be tackled.

As the size, and especially depth, of AI models increases, new algorithms for decentralized weight updates will be required. DNNs are sensitive to rounding errors, and straightforward merge mechanisms such as averaging weights may result in unacceptably high output divergence. Additionally, blindly sending large numbers of massive weight updates around a network of learning nodes will incur a severe overhead, easily in the range of gigabytes per device per day, necessitating more efficient dissemination algorithms. Finally, while privacy is not directly compromised by leaking data, the weight tensors of personally fitted edge AI models may still reveal personal habits and patterns by inference, and can potentially be abused by nodes that gather weight updates specifically for that purpose.

City of Things

Assuming these improvements in ubiquitous deployment, decentralized organization, standardized functionality and AI modelling are possible, a great many new Smart City applications will become possible. On a small scale, low-cost home automation will be able to use computer vision to identify items and keep track of fridge contents, combined with anonymized behavioral patterns, to suggest grocery shopping lists at convenient times. The same approach could be used at a larger scale in supermarkets to more efficiently guide employees in stocking shelves. Working at personal and city-wide scales simultaneously, roadside units and vehicular networks could predict traffic jams before they happen, and proactively redirect incoming traffic according to their preferences. Going one step further, behavioral

patterns could be combined with traffic predictions to suggest optimal times and modes for transportation. It is clear that in order for these applications to scale on-demand in real-time, and to absolutely preserve privacy, highly flexible edge computing with powerful AI engines will be required.
Last but not least, edge computing is also being rapidly adopted for Smart Grids and energy efficiency in general. The advent of Smart Meters and renewable energy, as well as the availability of cheap, easy power monitoring, opens up the potential for edge devices that optimally regulate their own power use depending on available local renewable power and past trends, optionally negotiating with nearby devices. Although this strategy is clearly not possible for some devices (e.g. a TV), such functionality would, in general, significantly reduce grid load and help achieve renewable energy goals, rather than wasting or disabling excess local production, at no extra effort to end-users. At a higher level, edge and fog computing can balance excess production and demand within neighbourhoods, before routing anything over larger distances.

References

[1] A. Randazzo and I. 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.

[2] C. Systems. *OSv*, July 2018. Available from: https://github.com/cloudius-systems/osv.

[3] T. Caraza-Harter and M. M. Swift. *Blending containers and virtual machines: a study of firecracker and gVisor.* In Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pages 101–113, 2020.

[4] *WASI - The WebAssembly System Interface*, October 2021. Available from: https://wasi.dev/.

[5] *About OCI.* Available from: https://www.opencontainers.org/about.

[6] *Data protection in the EU*, February 2022. Available from: https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en.

[7] A. A. Suzen, B. Duman, and B. Sen. *Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN.* In 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA). IEEE, jun 2020. doi:10.1109/hora49412.2020.9152915.

[8] T. S. Perry. *Move over, Moore's law. Make way for Huang's law [Spectral Lines].* IEEE Spectrum, 55(5):7–7, may 2018. doi:10.1109/mspec.2018.8352557.