Accelerating Orchestration with In-Network Offloading

Stefanos Sagkriotis

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

May 2023

Abstract

The demand for low-latency Internet applications has pushed functionality that was originally placed in commodity hardware into the network. Either in the form of binaries for the programmable data plane or virtualised network functions, services are implemented within the network fabric with the aim of improving their performance and placing them close to the end user. Training of machine learning algorithms, aggregation of networking traffic, virtualised radio access components, are just some of the functions that have been deployed within the network. Therefore, as the network fabric becomes the accelerator for various applications, it is imperative that the orchestration of their components is also adapted to the constraints and capabilities of the deployment environment.

This work identifies performance limitations of in-network compute use cases for both cloud and edge environments and makes suitable adaptations. Within cloud infrastructure, this thesis proposes a platform that relies on programmable switches to accelerate the performance of data replication. It then proceeds to discuss design adaptations of an orchestrator that will allow in-network data offloading and enable accelerated service deployment. At the edge, the topic of inefficient orchestration of virtualised network functions is explored, mainly with respect to energy usage and resource contention. An orchestrator is adapted to schedule requests by taking into account edge constraints in order to minimise resource contention and accelerate service processing times. With data transfers consuming valuable resources at the edge, an efficient data representation mechanism is implemented to provide statistical insight on the provenance of data at the edge and enable smart query allocation to nodes with relevant data.

Taking into account the previous state of the art, the proposed data plane replication method appears to be the most computationally efficient and scalable in-network data replication platform available, with significant improvements in throughput and up to an order of magnitude decrease in latency. The orchestrator of virtual network functions at the edge was shown to reduce event rejections, total processing time, and energy consumption imbalances over the default orchestrator, thus proving more efficient use of the infrastructure. Lastly, computational cost at the edge was further reduced with the use of the proposed query allocation mechanism which minimised redundant engagement of nodes.

Acknowledgements

First and foremost, I would like to express my gratitude to Prof. Dimitrios Pezaros who patiently shaped my academic criterion by introducing me to state of the art challenges, transferring his experience and knowledge, and guiding me on how to conduct impactful research. His support made this PhD possible, and his advice and feedback turned it into an invaluable experience.

This PhD would also not be possible without the help and support of my partner, Evangelia Nakou. She has been my strongest ally, tirelessly offering solutions to every hurdle by sacrificing her time and energy. I cannot thank her enough for helping me pursue my goals.

I am also grateful to my family: Georgios Sagkriotis, Elpida Mitropoulou, Angeliki Sagkrioti, for their care and encouragement. They have always provided a safety net behind each of my decisions.

I would also like to thank my second supervisor Dr Christos Anagnostopoulos for his feedback and guidance during the early stages of my PhD. I am also thankful to Dr Kostas Kolomvatsos for his advice and collaboration. From BT, I would like to thank my industrial supervisor, Peter Willis, for sharing his experience and thoughts to help me shape my PhD topic.

For his encouragement to pursue a PhD and his belief in me, I would like to thank Prof. Ioannis Moscholios. His mentorship during this journey greatly helped me form decisions and pursue my dreams.

To the fellow PhD students and collaborators: Dr Mircea-Mihai Iordache-Sica, Dr Kyle Simpson, Dr Alejandro Llorens Carrodeguas, Dr Irian Leyva-Pupo, Haruna Umar Adoga, Dr Marco Cook, Yiliyasi Sulaiman, Dimitrios Barkas, I am thankful for the engaging discussions, encouragements, and occasional banter. You have all assisted me greatly in unique ways over the previous years.

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 12, 20, 35, 36

ASIC Application-Specific Integrated Circuit. 2, 17, 24, 25, 45, 47, 60, 119

BPF Berkeley Packet Filter. 12

cBPF classic BPF. 12

CLI Command-Line Interface. 20

CNI Container Network Interface. 64–67, 74, 77

CPU Central Processing Unit. 10, 11, 13, 14, 16, 24, 31–33, 36, 75, 83–85, 89, 91, 92, 94, 97, 99, 102

CRAQ Chain Replication with Apportioned Queries. 45, 49, 50, 52, 53, 57

DMA Direct Memory Access. 14, 16

DPDK Intel Data Plane Development Kit. 11–13

eBPF extended Berkeley Packet Filter. 12, 13, 16

ECAS Energy Capacity-Aware Scheduler. 83, 92, 94, 97–100, 102, 103, 114

EN Edge Node. 104–110

ETSI European Telecommunications Standards Institute. 26, 27, 30, 35

FPGA Field-Programmable Gate Array. 17, 68, 119

GPU Graphics Processing Unit. 14, 24, 41, 119

GUI Graphical User Interface. 20

HDL Hardware Description Language. 17

IaaS Infrastructure as a Service. 34

IETF Internet Engineering Task Force. 9

ILP Integer Linear Programming. 29, 31, 32

IoT Internet of Things. 3–5, 7, 15, 32, 34, 40, 42, 69, 71, 72, 75, 86, 96, 98, 99, 102–104, 106, 111, 112, 114, 116–118

KPI Key Performance Indicator. 1, 2, 27, 31

KS Kubernetes Scheduler. 97–100, 102

KV Key-Value. 5, 37, 39, 44–47, 49, 50, 52, 54, 56, 57, 61, 62, 64–68, 103, 117, 118

KVS Key-Value Store. 2, 3, 5, 17, 36, 38, 39, 44–47, 49, 51, 54, 57, 60, 64, 65, 67, 68, 104, 116, 118, 119

LLS Least Loaded Scheduler. 97–100, 102, 114

MANO Management and Orchestration. 26, 28, 35

NAT Network Address Translation. 30

NF Network Function. 17, 26, 28, 30, 35, 36, 69

NFV Network Function Virtualisation. 2, 4, 26–29, 34, 35, 42, 44, 69, 74, 79, 115, 116

NIB Network Information Base. 21, 22

NIC Network Interface Card. 10–14, 16, 118

NPU Network Processing Unit. 16, 24, 25, 38, 47

ONAP Open Network Automation Platform. 34

OS Operating System. 13, 15, 27, 28, 72, 79

OSM Open Source Mano. 35

OVSDB Open vSwitch Database Management Protocol. 20

P4 Programming Protocol-Independent Packet Processor. 12, 16, 17, 25, 26, 36, 42, 45, 47, 52, 56, 57, 60, 64, 68, 115, 119

PaaS Platform-as-a-Service. 15

PCA Principal Components Analysis. 106, 108, 109, 118

PDP Programmable Data Plane. 2, 5, 8, 37, 38, 40, 44–47, 50–52, 58, 64–67, 103, 115, 117

PISA Protocol-Independent Switch Architecture. 24, 25, 45, 67

QC Query Controller. 70, 104–107, 109, 110

QoS Quality of Service. 19, 30–32, 85, 86

QP Query Processor. 104–107, 109–111

QPS Queries Per Second. 58, 60

REST Representative State Transfer. 20

RFC Request For Comments. 9

RMSE Root Mean Square Error. 89

RMT Reconfigurable Match-Action Table. 23, 24

RSS Receive-Side Scaling. 11, 14

RTT Round-Trip Time. 45, 47, 48, 57, 62, 65, 67

SBC Single Board Computer. 3, 4, 15, 33, 70–72, 75, 79, 83, 89, 93, 110, 116

SDN Software Defined Networking. 1, 2, 4, 6, 18, 19, 22, 27, 44, 116

SLA Service Level Agreement. 32, 82

SLO Service Level Objective. 20, 22

smartNIC smart Network Interface Card. 16, 23, 24, 68, 115, 118

SoC State of Charge. 33, 72–74, 77, 83, 85, 88, 89, 91–94, 100, 102

SRAM Static Random-Access Memory. 24, 44, 51, 65, 117

TCAM Ternary Content-Addressable Memory. 24

TCP Transmission Control Protocol. 9, 14

UDP User Datagram Protocol. 45, 47, 52, 53, 116

USV Unmanned Surface Vehicle. 110, 111

VM Virtual Machine. 27, 28, 30–32, 34, 35

VNE Virtual Network Embedding. 28, 29

VNF Virtualised Network Function. 5, 26–35, 42–44, 47, 69, 70, 79–81, 98, 114, 118, 119

XDP eXpress Data Path. 12, 13

Chapter 1

Introduction

1.1 Overview

Over the past few years, computer networks have proved to be essential infrastructure by providing services that are used daily from billions of users. Data centres and end-user networks are interconnected through backbone infrastructure with methods resilient enough to support growing numbers of users at constantly increasing access speeds, requesting new types of online services. Even during the series of challenges imposed (or amplified) by the Covid-19 pandemic, critical sectors like healthcare, education, banking, commerce, were in periods relying entirely on network infrastructure. Network traffic increases of around 20% had to be rapidly tamed (within a week) with the yearly estimated increase prior to the pandemic being approx. 30% [56]. The growing reliance in networking infrastructure brings expectations of high availability and stable performance while the room for configuration and management errors is shrinking.

To achieve this resilience and performance, the Internet structure has changed since its early days with the majority of Internet traffic now directed to a few cloud providers and content delivery networks like Google and Meta, also known as Hyperscalers or Hypergiants [72]. To minimise the distance between the end user and their infrastructure and meet latency requirements, they are peering directly at Internet Exchange Points and, in some cases, lay their own fibre networks. Hyperscalers bear the burden of ensuring availability of their services over the Internet with minimum latency overhead on top of backbone networks. They have to manage services spanning across multiple data centres in different locations while maintaining availability and consistency [36]. It is apparent that they cannot settle for eventual convergence of networking protocols or best-effort delivery. Hyperscalers strive to achieve intent-based network management so that network configuration is optimised towards meeting certain Key Performance Indicators (KPIs). This is done by employing Software Defined Networking (SDN) to obtain a centralised view of the network, monitor traffic and failures, and perform routing decisions shaped

around certain KPIs [57]. SDN technologies have been at the forefront of research during the previous years with multiple efforts from both industry and academia to deliver a sustainable separation of the network control plane from the forwarding plane. This separation is not trivial. Even one of the most prominent SDN specifications in-use, OpenFlow, presents multiple shortcomings: it supports a limited number of protocols, it fails to perform stateful traffic processing, and it is limited to a fixed set of possible actions against traffic flows.

Some of these technical challenges have been addressed through advances in programmable networking hardware which enabled stateful per-packet processing at line rate, with similar throughput to fixed-function switches. Programmable packet parsing allows experimentation with new protocols without the need for specification revisions. These developments have created momentum in the area of data plane programmability with multiple efforts to offload applications, either partially or completely, in Programmable Data Plane (PDP) hardware, like programmable Application-Specific Integrated Circuits (ASICs). With their hardware design optimised for certain computations, programmable network devices showcase accelerated performance of the offloaded services when compared to legacy implementations. Designing applications for partial or complete offloading of computation within network devices is also known as in-network compute.

Implementations of Key-Value Stores (KVSs) within network devices have been particularly promising, not only due to the integral role of KVSs in mainstream data centre applications (e.g., Google Spanner [36], Facebook Memcache [150, 7]), but also because of the close proximity of network devices to the source of KVS queries. The hops between the client and the corresponding server are minimised if the network device can accommodate queries instead of just forwarding them. Within the scope of this thesis is the design of in-network compute applications accommodating KVS workloads within programmable data planes to deliver performance enhancements. A replication algorithm is designed and implemented in PDP and through evaluation it is shown to improve the state of the art in aspects like scalability, throughput, and latency.

In parallel, both within and outside data centres, network administrators have to deal with device heterogeneity. The co-existence of different switching hardware generations can be the consequence of upgrades to achieve continuous growth [163]. Processing nodes might also be updated in phases within a data centre in order to be on par with modern processing capabilities. To deploy and manage network functions and services in heterogeneous hardware and be able to repurpose hardware, network operators employ Network Function Virtualisation (NFV) technologies. In the case of 5G and 6G, network functions need to converge between the cloud and the Radio Access Network [60, 144]. The management and orchestration of virtualised functions need to be agile and optimal in order to satisfy the availability and scale requirements of the workloads with minimum

excess resource consumption. These requirements are satisfied with complex orchestration platforms that gather measurements and health statistics from the deployed services and store them in distributed KVS. This thesis studies the characteristics of this functionality and proposes partial offloading of this functionality in programmable network devices in order to accelerate decision-making and provide scalability for large-scale orchestration.

In the case of telecommunication providers, the dependence of modern applications on low-latency responses from the network has pushed virtualised network functionality outside the cloud and close to the edge, especially for certain 5G and 6G use cases like Internet of Things (IoT) and augmented reality [20, 104, 38]. Storage and analysis of large amounts of data at the edge offers reduced link bandwidth usage that would otherwise be required for data transfers at the cloud, resulting in an overall computationally efficient infrastructure. This work explores data representation techniques that can describe the type of information held by the participating devices. Through data representation, queries can be directed to nodes based on the statistical relevance between the query and the information stored at the nodes. Data traversals can be minimised by executing data analysis tasks at the parts of the network that store relevant data. A methodology that can be used on resource-constrained nodes holding low-dimensional data is presented.

Single Board Computers (SBCs) have become a mainstream choice for IoT environments because of their small form factor, their connectivity with a wide variety of sensors and adaptors, and their improved hardware capabilities that demonstrate increasing processing capacity, higher network speeds, and growing memory size. These changes have increased the adoption rate of SBCs as computational nodes for Fog Computing and Edge Computing, either as standalone devices or in clusters [103]. This work explores this direction further by developing a cluster of SBCs that is used as a testbed for the development of a virtualised, scalable, and fault-tolerant application that is able to hold sensor data in local KVSs. Developed using lightweight containers to deploy virtualised services, the application proved that cloud-native virtualisation methods are usable at the edge and can run on resource-constrained SBCs. To assess the impact of virtualisation in IoT devices, the energy overhead of different tasks running within the cluster is measured and compared.

Mainstream virtualisation platforms like Kubernetes are usable in IoT clusters without however adapting their orchestration approach to the constrains of such environments. Sensor devices are usually running on limited energy resources that enables them to be mobile and easily deployable in diverse environments. Deploying and managing virtualised services without taking this into consideration can result in failed deployments due to insufficient resources or underutilisation of the available ones. Part of this thesis is dedicated in performing a list of changes in Kubernetes to make energy-aware decisions for service orchestration in energy-constrained IoT clusters. To achieve this, a regression

model that is able to establish the relationship between battery consumption and hardware utilisation is implemented. A new scheduling algorithm is integrated in Kubernetes to make it capable of assigning services to nodes based on the expected battery consumption and the available resources. The above are implemented and tested using a testbed of energy-constrained SBCs, indicative of edge infrastructure constraints and capabilities.

1.2 Thesis Statement

This work considers the capability of modern network hardware to perform per-packet stateful processing at line rates as a potential accelerator of computations within the network fabric. It asserts that, through stateful packet processing, data replication services can be implemented *within* the network in a scalable manner that outperforms legacy data replication methods. This work identifies limitations of previous in-network replication platforms and proposes design changes that offer better scalability, throughput and latency over the previous state of the art, without harming consistency or fault-tolerance. Because of the central role of Key-Value Stores as a coordination platform for widely deployed controllers and orchestrators and their workload characteristics, offloading this functionality in programmable hardware is proposed to reduce reaction times to network events and promote scalable orchestration.

This work also recognises the proliferation of virtualised network functions, able to formulate advanced network services in both data centres and edge infrastructure. It asserts that Edge IoT infrastructure can host mainstream orchestration platforms despite the existing resource constraints, further reducing latency at the end user by placing computation in low proximity. It proceeds to examine the energy impact of running an orchestration platform in IoT devices and obtains energy profiles for different roles and workloads within an IoT cluster. This work affirms that a mainstream orchestrator can be adapted to perform energy-aware scheduling and operate on a cluster of edge devices, using real time sensor readings to offer efficient use of the available resources. Finally, it asserts that statistical matching between queries arriving at the cloud and data existing at the edge can reduce excess computations by minimising data transfers and using only nodes storing statistically relevant data.

1.3 Contributions

The contributions of this thesis are presented alongside the produced publications:

- A study of the technological advancements leading to in-network compute, inclusive of NFV, SDN, and the respective applications existing in the literature (Chapter 2). The study is inclusive of works in the area of accelerated computing through

in-network offloading, both in PDP hardware designed for data centre environments and resource-constrained IoT environments. Particularly, motivating factors and existing limitations in the area of *data storage* technologies in data plane hardware and edge devices are traced through the literature.

- The thesis expands on the impact of design decisions in PDP environments (Chapter 3) (Publication 3). It addresses weaknesses and performance limitations of Key-Value (KV) platforms that operate in PDP by proposing a new in-network KVS platform that adopts a primary-backup replication method with major (in the orders of magnitude) latency, throughput, and scalability improvements over existing solutions (Publication 2). The thesis, by analysing the workloads generated from mainstream orchestration platforms, expands on the potential benefits of using in-network data storage to deliver scalable orchestration (Publication 1).

- Motivated by the benefits of the Edge Computing paradigm, one of the first testbeds showcasing a mainstream orchestration platform running over resource-constrained IoT devices is implemented to validate the proliferation of locally-managed virtualised computing close to the end-user (Chapter 4) (Publication 6).

- The design and implementation of a virtualised application able to obtain the energy usage profiles of the various cluster nodes (Chapter 4). The results are used to identify the energy overhead of virtualised services running in IoT but also measure the energy overhead of routine functions of the orchestration platform (Publication 6).

- An energy-aware scheduling implementation that uses real-time sensor data to place Virtualised Network Functions (VNFs) within an IoT cluster (Chapter 4). The scheduler is integrated in a mainstream orchestration platform. It fully utilises the energy resources of the cluster by placing computations even at the master node, without sacrificing the systems reliability. Compared to the default scheduling approach, it increases the average acceptance ratio and reduces the total time events spend within the cluster (Publication 4).

- A mechanism that, by using in-network compute at the edge, generates statistical representations (signatures) of data stored in IoT nodes (Chapter 4). Data representations are used to help cloud nodes direct queries at the edge and reduce the participation of nodes holding irrelevant data. Due to statistical matching, the variance of the provided responses appears to be lower, thus improving the quality of the responses (Publication 5).

1.4 Publications

The work in this thesis has led to the following publications:

1. Stefanos Sagkriotis and Dimitrios Pezaros. Scalable data plane caching for kubernetes. In *2022 18th International Conference on Network and Service Management (CNSM)*, pages 345–351, 2022

2. Stefanos Sagkriotis and Dimitrios Pezaros. Scale-friendly in-network coordination. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 5747–5752, 2022

3. Stefanos Sagkriotis and Dimitrios Pezaros. Accelerating kubernetes with in-network caching. In *Proceedings of the SIGCOMM '22 Poster and Demo Sessions*, SIGCOMM '22, page 40–42, New York, NY, USA, 2022. Association for Computing Machinery

4. Alejandro Llorens-Carrodeguas, Stefanos G. Sagkriotis, Cristina Cervelló-Pastor, and Dimitrios P. Pezaros. An energy-friendly scheduler for edge computing systems. *Sensors*, 21(21), 2021

5. Stefanos Sagkriotis, Kostas Kolomvatsos, Christos Anagnostopoulos, Dimitrios P. Pezaros, and Stathes Hadjiefthymiades. Knowledge-centric analytics queries allocation in edge computing environments. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2019

6. Stefanos Sagkriotis, Christos Anagnostopoulos, and Dimitrios P. Pezaros. Energy usage profiling for virtualized single board computer clusters. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2019

1.5 Organisation of the Thesis

The remaining of this thesis is structured as follows:

- **Chapter 2** describes the evolution of network softwarisation: the path towards a centralised view of the network with SDN; the key contributions that have enabled data plane programmability and use cases; and topics on the management and orchestration of virtualised network functions for both edge and cloud environments.

- **Chapter 3** reviews state of the art work on data replication using the programmable data plane. It identifies shortcomings and addresses key design limitations by proposing a new data replication platform that operates entirely in data plane. It discusses

the potential integration of the platform for the acceleration of container orchestration.

- **Chapter 4** explores the use of container deployment over clusters of edge devices as a way to offer virtualised services close to the end user. It identifies the challenges of container deployment in resource-constrained nodes, like IoT devices. With energy capacity being the limiting factor in such environments, an energy-aware scheduler is presented as a component that will efficiently allocate the available energy resources to incoming requests. Lastly, this chapter provides a mechanism that allows queries from the cloud to be allocated to the appropriate edge device, based on statistical similarity between the query and the data existing at the device.

- **Chapter 5** summarises the work by revisiting the contributions and the thesis statement. It also discusses future research directions.

Chapter 2

Background

2.1 Overview

This Chapter critically discusses a sequence of efforts and ideas that have shaped modern networking into offering enhanced programmability, interoperability, and ease of management. These works are not presented in a linear timeline and instead are grouped into sections based on their relevance. For example, technologies that attempt to enhance end-host packet processing are grouped together, the same happens for bespoke hardware technologies, etc. The reasoning behind each technological shift is presented, leading to current challenges in each respective area, with a focus on PDP technologies, in-network compute, and orchestration of virtualised network functions, which are analysed in more detail to help the reader assess the contributions of this thesis.

2.2 Challenges in Network Evolution

The networking community was challenged with tremendous growth of Internet traffic and network users after the 1990s. The increasing popularity of applications like media streaming, online gaming and peer-to-peer networking has resulted in a permanent increase of demand and pushed networking infrastructure to accommodate greater amounts of generated traffic. New types of applications surface daily with requirements for low-latency, high-bandwidth, and high availability. More recently, critical society domains, e.g., healthcare, banking, commerce, anticipate high quality of experience and reliability for their day to day activities that are running over the network infrastructure. A series of changes and adaptations of infrastructure have shaped modern networking and have made it capable of surpassing previous bottlenecks while growing reliance on the infrastructure and coping with an ever-growing demand for bandwidth.

Efforts to satisfy demand revealed important flaws in the operation of networks. One such flaw was the interaction between manufacturers of silicon for network devices and

the networking protocol standardisation community – the Internet Engineering Task Force (IETF). Silicon manufacturers developed closed hardware that implemented existing versions of protocols. On the other end, IETF produced Requests For Comments (RFCs) that patched issues existing in proprietary protocols with the aim of increasing stability, security and performance. Switches existing in networks could not be reprogrammed due to their closed hardware, which resulted in security and performance issues existing in the wild. This caused reliance on outdated protocols – an issue also known as protocol ossification [158, 160].

Protocol ossification proved particularly costly in the case of Transmission Control Protocol (TCP) which is still widely used as the main transport protocol. In TCP, the use of large buffers could heavily impact its probing mechanism and increase queuing times – a problem that is known as bufferbloat [71]. Although numerous proposals were made to mitigate this, e.g., Active Queue Management methods that dynamically change the buffer size (like RED and CODEL [59, 149]), switches with locked hardware slowed the adoption of these methods. Exogenous technological changes can also introduce novel factors of degraded performance. An example is the use of of mmWave wireless links with TCP where rapid bandwidth variations in the physical layer can lead to bufferbloat problems [204].

2.3 Early Efforts for Network Programmability

The coupling of network hardware and implemented protocols was disrupting network evolution and slowing down adaptation to external changes and demand. Extensibility was limited and experimentation with new ideas was constrained. The networking community was aware of the limited flexibility of the networking architectural model since the '90s and suggested alternatives before the manifestation of large scale consequences. Early efforts focused on addressing these fundamental limitations by proposing programmable network nodes able to perform arbitrary computations on traversing packets and modify them accordingly. This body of work is known as active networking [189, 85].

In the context of this thesis, active networks are examined as an early conception of the idea of network programmability. Programmable switches, a currently popular research topic, were proposed as part of a framework that allows network operators to dynamically change the code that runs within their routers, thus enabling extensibility and coping with ossification. A more passive alternative was capsules – packets that entail code fragments and embedded data that can be processed by switches to either activate built-in primitives or re-program the switch using their payload [189].

Due to the significant technical challenges and security concerns that their implementation involves, these proposals remained mostly theoretical until recently. The idea of

switches that are able to perform arbitrary computations at line rates could not be supported by the available hardware. Innovation in silicon manufacturing and data plane programming languages made programmable switches available. Yet, contrary to what was portrayed in active networking papers, the tools to program network devices restrict certain operations that can impact packet processing performance (e.g., loops) to ensure time-bounded execution of the compiled code. Moreover, completely repurposing a switch is not possible without significant downtime.

On the other hand, the security and performance implications of capsules rendered their use prohibitive in modern networks. One such limitation is the authentication of the encapsulated code which requires privileged access to networking resources. Recent research examines programmable hardware as a potential enabler for the deployment of capsules and suggests modern authentication approaches as a way to overcome the security limitations of the original proposal [41]. However, there is not adequate motivation for the deployment of capsules as the benefits over existing methods of network programmability are yet unclear.

A more pragmatic approach to resolve the early problems of networking envisioned the separation of the control from the data plane. An example of such effort to introduce an open interface between the control and the data plane is detailed in the ForCES standard [199]. The standard attempts to provide logically centralised control by allowing the installation of forwarding-table entries in switches through an external controller. The lack of adoption of the standard from router vendors resulted in little deployment and halted its evolution [55]. Similarly, works like RCP [54] and and SoftRouter [119] attempted to deliver a programmable control plane that is able to adapt to traffic loads. These works were faced with criticism on the fault-tolerance properties of the controller and the violation of the fate sharing principle of traditional network protocols [55].

2.4 Programmable Networking in Commodity Hardware

With fixed-function switches not permitting programmability, other alternatives were explored. Network researchers turned to general purpose processing hardware to build software routers based on commodity PC components. The extensive list of programming tools available for x86 Central Processing Units (CPUs) ensures easy adaptation of new protocol features and customisation in packet processing. It is reasonable to assume that the performance achieved by a software router that lacks all hardware-level optimisations is not on par with a fixed-function switch. Software packet processing entails multiple processing overheads that occur from the various components of a commodity PC that have to exchange data in order to process a packet: packets from the Network Interface Card (NIC) are transferred to memory, which are subsequently processed by the CPU and then

sent to NIC's ring buffer. Packets are finally transferred from memory to NIC in order to be transmitted. The upper bound of a processor's capacity and bus speeds remains a big factor of routing performance for software implementations, even with today's available hardware [157].

2.4.1 User Space Packet Processing

The network stack transactions of a general-purpose Linux kernel can introduce processing delays, for example, multiple system calls that facilitate the kernel to user space transition of packets introduce delays [28]. Other disadvantages are the restricted in-kernel programmability and processing performance that, while it is higher over user space, is proven to be inadequate for multiple 10 Gbps interfaces [14]. This is also affected from network stack features that are enabled by default but are unnecessary for certain environments and negatively affect processing times, e.g., traffic filtering at an end host device of a data centre. To circumvent overheads and maintain programmability, various methods of accelerating user space packet processing have been developed:

- One of the simplest methods is transferring raw packets to user space by **bypassing the kernel** completely.
- Transferring packets from NIC in **batches**, as a way to minimise the total amount of system calls required to access the interface.
- A buffer pool that is visible from both the NIC and user space, therefore avoiding memory-to-memory copies of buffers between kernel space and user space. This method is known as **zero-copy**.
- Supporting **multiple hardware queues in** modern **NICs** to improve parallelism by offloading packets to multiple cores. Therefore, incoming traffic can be load-balanced among the available cores.

Frameworks that combine the aforementioned methods in order to provide a coherent environment that enables packet processing have been proposed. One of the most prominent ones is Intel Data Plane Development Kit (DPDK) [164]. It provides a set of drivers and libraries that mainly support packet processing programmability for x86 and ARM CPUs. DPDK incorporates all of the aforementioned features (kernel bypass, zero-copy, etc.) to enhance processing performance and enables multi-core handling of packets. Packets are transferred either in run-to-completion mode or pipeline mode. Under run-to-completion, packets are distributed to multiple cores using Receive-Side Scaling (RSS) [142]: a packet distribution method that hashes packet header fields and distributes them evenly across multiple hardware RX queues. In pipeline mode, one core is responsible for transferring packets from NIC and another one for processing and transmitting them.

Another widely adopted framework is Netmap [169]. Its Application Programming Interface (API) receives packets directly from NIC whose rings are partially disconnected from the network stack (which would be the default setup). A shared memory provides buffer rings between the host stack and the NIC and enables Netmap to exchange packets with both. It enables programmability through lightweight metadata representations of packet processing definitions. Through this, device-specific features are hidden and batch processing is enabled. It supports zero-copy features through a buffer pool which is statically allocated upon device initialisation. The lack of dynamic definition of this buffer can force memory-to-memory copies upon buffer overflow, which in turn can degrade performance. Netmap, while not as feature-rich as DPDK, displays similar performance when integrated in software routing platforms [14].

2.4.2 In-Kernel Packet Processing

The network stack of the Linux kernel is mostly restrictive in terms of programming flexibility. There have been efforts to introduce programmability to some of the network layers. A well-regarded framework that provides an instruction set and an execution environment that enables programmability in Linux kernel is the extended Berkeley Packet Filter (eBPF) [97]. More specifically, instructions and data are passed at the lowest level of the Linux kernel – called eXpress Data Path (XDP). A restricted version of C or Programming Protocol-Independent Packet Processor (P4) (see Section 2.5.4) can be used as the programming language which is then compiled into object code. The code is then verified and translated before being offloaded to the processor or a compatible NIC. Data can be exchanged between user space and the eBPF executable through generic key-value stores called maps (defined in user space). Maps enable data to persist between different executions of a program, can share info between different programs, and interact with user space in real time [193].

eBPF origins date back to 1992 and the Berkeley Packet Filter (BPF) framework, developed to apply packet filters within the kernel. In the original BPF (also known as classic BPF (cBPF)), packet filters were described with the use of 22 instructions and two 32-bit registers. In eBPF, the instructions are extended to implement arithmetic and logic instructions, function calls, and table operations. The 32-bit registers are also swapped for 64-bit width registers and their number grows from 2 to 11. The pipeline that was followed in BPF to deploy filters in-kernel ensured high performance, crash-free execution, and Just-In-Time compilation. The instructions describing the filters were passed to the kernel in the form of bytecode which was then verified and compiled. A similar pipeline was followed in eBPF, as described above, to retain the proven performance benefits. eBPF's programmability would be expected to bear performance costs in throughput and processing latency, but its reliance on XDP displays improvements over default Linux:

it outperforms default Linux in packet drop throughput by achieving five times greater performance while consuming fewer CPU resources for the same amount of traffic [87]. It is worth noting that both eBPF and Linux are outperformed by DPDK with significant latency improvement and throughput increases but with increased CPU usage [87, 193].

Unlike DPDK, eBPF does not require a re-implementation of structures used to process packets, like routing tables. By using existing structures found in the kernel, implementation becomes simpler and stability is ensured by using the default XDP kernel module [193]. eBPF does not require exclusive control over the NIC and permits its use from other Operating System (OS) applications. Taking into account the reduced CPU usage it shows when compared to DPDK, it provides an efficient way to utilise hardware resources. eBPF has been used to deliver diverse functionality over large-scale environments, like Facebook's load-balancer Katran [52], Netflix's traffic monitoring tool Vector [96], or widely adopted software like the Suricata network monitor and Intrusion Prevention System [62].

2.4.3 Software Routers

One of the first attempts to deliver a software router was Click [111]. The developers focused mostly on providing an easily extensible router based on a modular architecture that combines building blocks called "elements". Each building block constitutes a different feature in the packet processing pipeline (e.g., queues, scheduling) and they are written in C++. Its design contributed significantly in materialising a programmable and extensible router. Fine grained latency measurements were obtained for each proposed "element", but the results concern deprecated hardware and cannot be assessed in a meaningful manner for today's standards. Click was later adapted to use both DPDK and Netmap with great performance improvements in both cases.

In order to obtain the best possible performance it is sensible to optimise for processing parallelism. In recent years, CPUs have greatly improved their parallelism capabilities through multi-core architectures. One of the first implementations that optimised for parallelism across all stages of packet processing is RouteBricks [48]. The developers aimed to deliver performance comparable to fixed-function routers by utilising parallelism not only within a single server but also across multiple servers. Packets are distributed among the servers with the use of load balancing. However, efforts to utilise all the available hardware features did not yield real switch performance. The main bottleneck was identified to be the CPU's speed and the shared bus between the memory and the CPU.

2.4.4 GPU Offloading

The integration of NIC hardware features in packet processing allowed higher throughput and better utilisation of CPU resources (e.g., multi-queuing traffic offloaded to multiple cores) while maintaining the flexibility of user space environment. This motivated experimentation with other PC components with unique hardware features, like Graphics Processing Units (GPUs), that could be used as a co-processor for offloading certain workloads. GPUs host a lot of processing cores, up to orders of magnitude more when compared to a CPU, and therefore present great potential for parallel processing. What can be challenging, is providing adequate volume of data to make use of all the cores. GPU cores tend to be slower than CPU ones, making their use inefficient for small workloads [185].

Snap, a platform based on Click, utilised GPUs for fast path computations and managed to increase the platform's maximum attainable throughput up to four times [185]. Snap's main goal is to preserve parts of Click's packet processing pipeline in CPU for flexibility and only transfer the parts that present degraded performance in GPU. Snap enabled multiple packets to be transferred between the platform's elements (previously prohibited in Click). A new type element was added, called PacketBatch, that is able to pass batches of packets between Click's elements, thus utilising the parallelism potential of GPUs. A shortcoming of parallel processing is packet re-ordering, which can negatively affect TCP performance. To avoid out-of-order packet transfers between the GPU and the CPU, a new queue element was developed that would pass packets sequentially back to the CPU. Other optimisations concern the elimination of packet divergence occurring from packets following different processing paths and the use of packet slicing to reduce the amount of data that is copied between CPU and GPU. Snap's evaluation indicated meaningful forwarding performance improvements over Click, with approx. 2× higher throughput for smaller packet sizes (64-256 Bytes) and similar performance for bigger packet sizes.

PacketShader [81] is another routing platform that faced similar challenges to adapt packet processing for GPUs. The default Linux network stack implements a mechanism that consumes a lot of CPU cycles to allocate and deallocate buffer memory for incoming packets. The allocations concern two buffers: `skb`, used for packet metadata that is passed between network layers, and a buffer for the packet data. PacketShader replaced the dynamic buffers with two fixed-size buffers, with fixed-sized cells which are reused upon RX queue wrap-up. This removed the per-packet buffer allocation costs and the Direct Memory Access (DMA) mapping costs (DMA: the transfer of packet data from NIC to memory through PCIe). Apart from this, batch processing was also implemented at the application level (instead of just using batching for NIC in Click [111]) in order to fully utilise GPU capabilities. RSS was used to deliver core-aware NIC RX and TX queues.

Forwarding Performance appeared improved for small packet sizes (64 - 512 Bytes) of IPv6 traffic.

2.4.5 Single Board Computers

SBCs are small form-factor devices that provide all of the components of common computers using a single circuit board. SBCs are, in most cases, low-cost devices using low energy (compared to common computers). Their OS is typically based on Linux, which inherits a well-known networking stack and some of the packet processing features explained in the previous sections. SBCs can therefore support custom packet processing pipelines in edge environments, albeit with reduced performance due to hardware limitations. Additionally, SBCs offer connectivity with a wide range of sensors and devices [103]. An extensive survey by Johnston et al. performs an in-depth analysis of the state of the art use cases for these devices [103]. They explain the main characteristics of SBCs and detail the different device models available in the market. In addition, the authors identify the broad domains where the SBCs might be deployed. The authors confirm that these devices present great potential for service deployment at the edge because of their power requirements and size. SBCs are characterised as computational game changers that can bring computation closer to the data-generating parts of the network [103].

SBCs present a wide range of characteristics that satisfy the requirements posed by modern edge applications, either as standalone devices or through formulation of clusters. The works of Basford et al and Pahl et al further expand on the topic of cluster formulation using SBCs [15, 156]. A cluster can be created either by coupling physical elements together or by using the concept of Platform-as-a-Service (PaaS) to create and manage the cluster. Basford et al. present a new method for creating physical clusters of SBCs, called the Pi Stack [15]. This method minimises the amount of cabling required to create a cluster by reusing some elements of an SBC's physical construction as a communication channel for both power and management. The researchers compare three different SBC clusters using the proposed technique. The clusters are composed of nodes from several vendors. Their results reinforce how important SBC clusters are in improving resilience and performance in IoT deployments. Likewise, Pahl et al. [156] have built an SBC cluster using the PaaS paradigm. However, they deploy their own dedicated tool to manage and configure the cluster. The authors do not compare against other management platforms. Thus, further evaluations needs to be done to demonstrate the feasibility and benefits of their approach.

2.5 Network Programmability over Bespoke Hardware

Efforts to achieve network programmability in commodity hardware, while successful, provide limited performance in terms of throughput and latency. The research community has been exploring alternatives in the form of bespoke hardware, able to accelerate packet processing while remaining programmable. The latter is increasingly difficult when moving away from generic processors. Recent advancements in programmable networking devices and programming languages, both for end-host and switching devices, have (to a certain degree) addressed previous limitations. This Section discusses such changes with an emphasis on switching hardware which is within the scope of this thesis' contributions.

2.5.1 End-Host Programmability

Network Processing Units

To cope with the continuous expansion of supported network protocols and applications, a special type of hardware was incorporated in network topologies circa 2000 known as network processors or Network Processing Units (NPUs). These were usually substituting traditional NICs in the network. NPUs integrated application-specific instruction processors which provide packet processing at relatively fast line rates. Their design was based on processors with reduced instruction sets, mainly developed for instructions like bit manipulation and lookup operations. They also integrated specialised hardware components for fast DMA between RX and TX ports and internal memory [107]. A special packet scheduler was responsible for distributing packets across the processor's cores according to their incoming order [126]. Some representative devices, like the Cavium LiquidIO and the Intel X520 appear to have a low port density, and therefore reduced throughput [27, 94]. Programming these devices is dependent on the vendor's support and requires high expertise, which made them a less popular choice for network operators [100].

Smart Network Interface Cards

Similar to NPUs, smart Network Interface Cards (smartNICs) can perform packet processing in specialised hardware in order to avoid the use of CPU resources. The term "smart" is attributed to the extended programmability that these devices offer when compared to legacy NICs. They are equipped with a fully programmable multi-core System-on-Chip processor which, like NPUs, is designed for fast network-specific operations. They can be programmed with open tools like eBPF (Section 2.4.2), P4 (Section 2.5.4), and C, minimising the requirement for specialised knowledge of the vendor's tools. Their attainable throughput is up to 400 GbE with the port count remaining fairly low (up to four ports) [152]. Their memory and processor specifications are the main bottlenecks to the types of

programs they can host. Example applications include load balancing [141], distributed denial of service mitigation [140], and the deployment of a KVS [182].

Field-Programmable Gate Array Cards

Another type of programmable network cards has become available. Field-Programmable Gate Array (FPGA) attached to network cards offer a conceptual array of programmable logic blocks and memories that can be combined using Hardware Description Language (HDL). Their hardware enables runtime reconfiguration while using only the required processing blocks for the specified functionality, offering fast processing speeds. FPGA cards have shown comparable performance to ASICs by achieving processing speeds of up to 400 Gbps, making them a very capable end-host processing device. An influential open-source implementation, targeted towards educational purposes, has been made available with the work of Lockwood et al., [129]. A HDL might require a significant time investment to master and optimise for specific network operations. Alternatively, high-level synthesis tools can be used with syntax similar to C and the output can then be converted to HDL. This workflow does not offer the maximum attainable speeds from hardware and therefore is not optimal for commercial deployments. Another workflow which allows describing the data plane behaviour of the device with P4 and then compile it in HDL without performance reductions has been recently proposed [92].

2.5.2 Middleboxes

To keep up with network adaptations required to satisfy performance and security requirements, bespoke hardware appliances known as middleboxes have been extensively used in operational networks. Middleboxes contain fixed-function ASICs that process packets at high rates according to the function they implement. Examples of middlebox functions are: load balancing, Network Address Translation, flow monitoring, firewall, or intrusion detection. Significant upfront cost might be necessary to acquire and configure middleboxes, with many of the aforementioned network functions requiring placement at the edge of the network and therefore great capacity provisioning which translates to increased cost.

Middleboxes require careful planning as the same Network Function (NF) might need to be applied on multiple traffic flows which have to be redirected to the same middlebox. In such cases, redirection can incur high bandwidth consumption and latency increases [29]. Middleboxes can also have traffic-changing effects, with certain functions generating or reducing the amount of traffic based on its characteristics, leading to unpredictable hot spots in the network [132]. In other cases, there might be dependencies between middleboxes, e.g., traffic that first has to be processed by an Intrusion Detection System before reaching a load-balancer [137]. Middlebox placement in general topologies has

been proved to be NP-hard [29]. Their fixed-function characteristics also result in limited extensibility and reliance on vendors for patches and updates.

2.5.3 Software Defined Networking

Since the era of active networking, the execution of custom computations in the data plane was identified as a key enabler for expanding the list of supported protocols and introducing in-network services. Network engineers, in most cases, strive for fast integration of new features to the equipment they operate instead of relying to vendors for delayed implementations of these features. From an investment perspective, owners of network equipment expect adequate programmability that will allow future integration of new services and features.

SDN was designed to address these rigid requirements. Under SDN, the traffic forwarding functionality – data plane, is differentiated from the decision making around forwarding rules – control plane. In this manner, the control plane remains agnostic about the implementation of data plane code and hardware that forwards traffic. The two planes communicate using a specification that details the different types of messages required for establishing rules. The first and most prominent specification is OpenFlow, originally designed for deployment over a university campus and later adopted by many commercial switch vendors and network operators [135].

OpenFlow

Under OpenFlow, the software that is able to manipulate the forwarding rules of compatible switches is called the controller (details in Section 2.5.3). OpenFlow specifies a controller-switch protocol that is executed to establish a secure channel between the two entities which is subsequently used to exchange switch features, like supported optional instructions, which are sent to the controller and the desired rules are forwarded back to the switch. Traffic monitoring and device statistics can be exchanged in order to adapt decision-making in the controller. Packets can also be forwarded to the controller in order to process unexpected entries and introduce new rules.

In OpenFlow switches, packet header fields are matched against tables of rules that define a corresponding action. This is known as the Match-Action approach. A rule can be matched against a subset of traffic called a flow and the tables that contain multiple rules are called flow tables. A packet that goes through the ingress port has to enter the ingress processing stage and (after OpenFlow v1.5.0) optionally an egress processing stage. Within each stage, multiple numbered flow tables define actions that are applied based on the matched packet fields. Actions are written in a per-packet action set that, when not containing a GoTo-Table instruction, the processing stops and all of the actions

within are executed on the packet. Packets cannot return to previous flow tables and the outcome of a table needs to be forward-pointing. Between the ingress and egress stages, a list of action buckets is defined in the form of a table which can decide which bucket will be executed. Tables within the egress stage are not allowed to change the output port that was originally assigned to the packet and can only forward packets to other egress flow tables.

Through the Open Networking Foundation, a non-profit body constituted of multiple networking equipment vendors, this specification has been revised multiple times through the years with the aim of enhancing the level of programmability offered. A sequence of the most significant of these changes is presented here in order to offer a notion of the standard's evolution relative to the requirements of the networking environments that adopted it. The standard moved from a single table to multiple tables per switch in Version 1.1. Tables could be linked to formulate sophisticated processing pipelines. Version 1.2 enabled extensible matching and header rewriting by defining custom descriptors for existing header fields, and simultaneous connection to multiple controllers. In Version 1.3, Quality of Service (QoS) was enabled with the addition of per-flow meters used to control the rate of packets. This version also included several quality improvements by enabling more options in handling table misses and allowing simultaneous connections with a single controller. Version 1.4 added improved support for monitoring across multiple controllers, support for optical ports, and bundled control for easier rule updates in multiple switches. Finally, Version 1.5 added egress table processing per port, port recirculation for service chaining, better support for multiple controllers with more customisable bundles, and packet-aware pipeline processing by substituting the need for all packets to be layered over Ethernet.

Changes happening in the specification can provide a good insight of the deployment characteristics and requirements of the environments that adopted OpenFlow. A move towards more complex processing pipelines is apparent, with the aim to support a wider gamut of services in the network. Flow metrics are introduced in parallel with enhanced scalability which captures the need for automated decision-making to facilitate larger scale deployments with less human involvement. Concurrent connections to multiple controllers aim to increase fault tolerance and minimise potential downtime as OpenFlow starts to be deployed in more mainstream networks with stringent availability requirements.

OpenFlow Control Plane

Multiple SDN controllers have been developed in order to cover the different performance requirements of various deployments. Identifying some common elements among different controller designs can help assess the contribution of this thesis and link content from the following chapters. An overview of the main components that commonly exist in

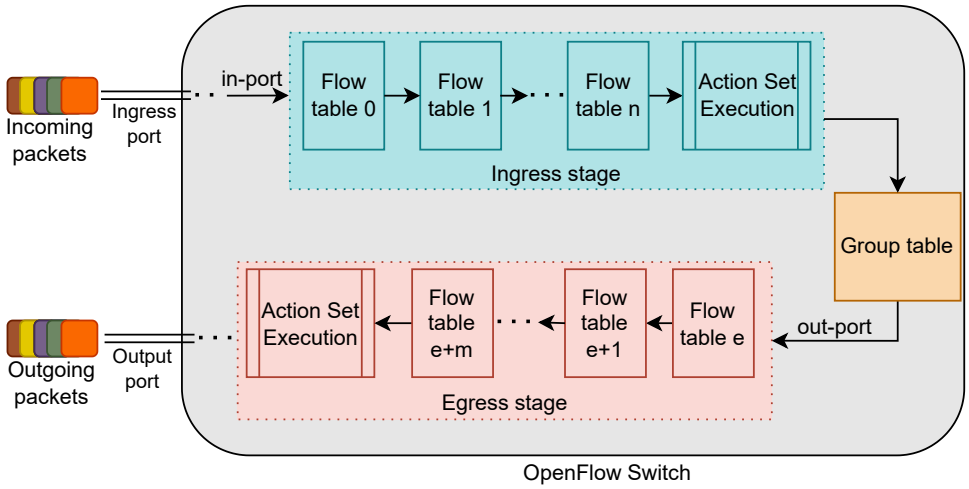


Figure 2.1: The processing pipeline of OpenFlow switches.

a controller is presented in Figure 2.2. The controller design is divided between the southbound and northbound interfaces that interact with its core modules.

The southbound interface is responsible for mediating communication between the control and the data plane. Flow rules can be disseminated through the OpenFlow specification that is implemented in the southbound interface. Similarly, virtual switch instances, like Open vSwitch, can be configured through an Open vSwitch Database Management Protocol (OVSDB) implementation [159]. Device configuration parameters can be managed through NETCONF [50]. The list of supported southbound protocols can vary for different controller implementations.

The northbound interface provides an API that developers can use to interact with the controller in order to implement their functionality. The interface can interact with the necessary controller modules to satisfy application requirements, providing a level of abstraction to the application developer. Depending on the controller implementation, this interface can be standardised with the use of a Representative State Transfer (REST) API to promote interoperability [208], e.g., YANG. Graphical User Interface (GUI) and Command-Line Interface (CLI) can also be implemented, delivering common functionality for debugging and monitoring.

The core modules are responsible for all internal controller operations. For example, the installation and management of flow rules is an integral functionality that is implemented as a core module. The rules installed in each network device need to be visible and manageable by the controller to perform further decision-making and rule updates. Other core controller modules include: the device drivers that are required to communicate with network devices or families thereof and manage their resources, the storage of authentication keys for all the connected entities, and the definition of Service Level Objectives (SLOs) that are used to automate configuration according to application intents.

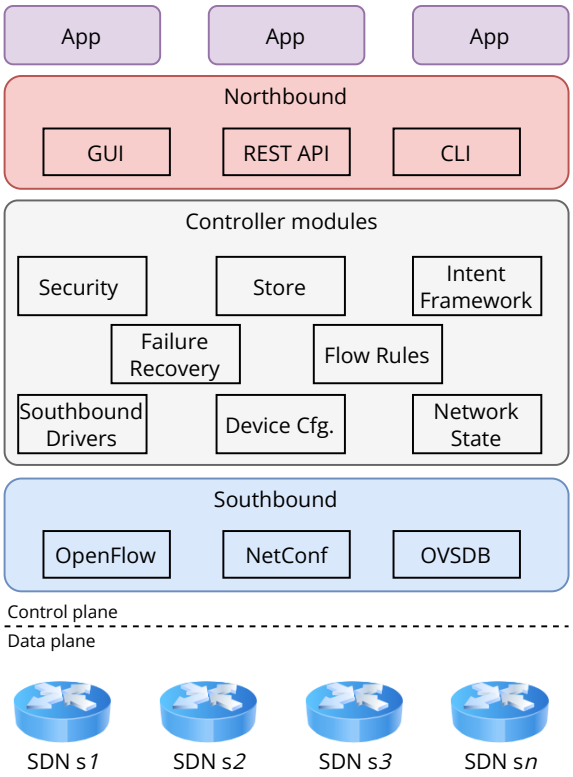


Figure 2.2: Overview of controller design and modules.

The maintenance of a consistent and up to date view of the network state (e.g., link status, capacities, port counters) within the controller – also known as Network Information Base (NIB), is instrumental to achieve reliability and scalability [116]. It is also an important assumption for the development of applications that rely on NIB accuracy to manage rules, e.g., load balancing. Mismatches between the perceived network state and the actual state leads to suboptimal rule updates [122].

This becomes an increasingly difficult problem for environments with a large number of data plane devices. A centralised controller entity (e.g., Ryu [34], Floodlight [165]) is inadequate for the amount of forwarding decisions required [2]. It is a scalability bottleneck and a potential single point of failure. A physically distributed controller is used instead in order to cope with demand by deploying multiple controller instances [13]. Such controllers are either operating in:

- a flat manner: physically distributed instances controlling forwarding only in their respective area (e.g., ONOS [117], Onix [116]). They still maintain a logically centralised view of the network by synchronising the network state.

- a hierarchical manner: the controller is vertically separated in layers. Lower layers handle local events. Events that require a wider view of the network are forwarded to the higher controller layers.

A flatly distributed controller requires frequent exchanges between all participating instances in order to obtain a consistent view of the network. The notion of a hierarchically distributed controller allows frequent events to be handled locally, resulting in fewer updates to the higher-tier controller instances. The latter is used in state of the art controllers, like Google's Orion, as it is found to significantly reduce the number of events forwarded to hierarchically higher instances without harming availability. In both cases, a consistent NIB is usually achieved through an in-memory hash table.

OpenFlow Achievements and Limitations

OpenFlow materialised a big portion of the SDN vision by defining a specification that could offer a logically centralised view of the network. This was an impactful step towards network softwarisation that in many ways resolved problems related to legacy networks (see Section 2.2). OpenFlow has been adopted in numerous data centres, allowing protocol extensibility over existing hardware in a pragmatic manner, contrary to active networking. The suggested changes over legacy networks maintained the previous link rates while introducing forwarding control.

An OpenFlow controller can configure multiple data plane devices, offering a scalable model for large deployments. It rendered per-device, vendor-specific, configuration unnecessary. Dynamic changes in forwarding rules were enabled, allowing adaptation to workload requirements. Different types of traffic were abstracted with the use of flows, grouping traffic in easier to manage groups and making batch changes possible. Internal protocol implementations are abstracted from the controller's perspective, adding flexibility for deployment in heterogeneous environments.

Overall, the separation of the data from the control plane improved manageability through achieving intent-based networking instead of eventual convergence of the underlying protocols – a big step forward from legacy networks. Thanks to OpenFlow, network administrators could gracefully handle network failures through the controller, enabling datacentre operation under measurable SLOs. OpenFlow moved from experimental deployment in campus networks to hierarchical deployment for hyper-scale networks, like Google's Jupiter [57]. More recently, it was even adapted to support the transition to optical circuit switches within the Jupiter network [163].

While the wide adoption of the OpenFlow specification from network vendors and operators brought SDN to fruition in data centre environments, being hardware agnostic is a key assumption in OpenFlow's design and remains a big constraint for the evolution of SDN. An issue that stems from this design decision is that local storage of data in the switch is not permitted in OpenFlow, making all data plane processing stateless. Flows that require storage of local metadata for their operation could require traversals to middleboxes and the controller. This results in increased flow processing overheads and a

missed opportunity for partial controller functionality to be offloaded in the data plane. Despite stateless forwarding, the packet processing pipeline requires multiple match-action stages that are protocol-dependent. Parallel processing is not available and all of the stages are sequentially executed, resulting in high overall packet processing complexity. Data plane customisation is not permitted, leaving potential optimisations unutilised.

OpenFlow supports a limited number of protocols by assuming a fixed parser, making the deployment and testing of new protocols a slow process. Even if a new protocol gets adopted by the specification, the controller as well as the participating networking devices need to be updated to the version that supports it. This recreates, at a certain extent, reliance to vendors for supporting their devices with specification updates.

2.5.4 Programmable Application-Specific Integrated Circuits

All aforementioned methods of introducing data plane programmability: purely software based, hybrid hardware-software designs, and smartNICs, either lack in performance or fine-grained programmability. OpenFlow reinstated that data centre traffic and scalable control can only be achieved through application-specific hardware. A significant breakthrough, by Bosshart et al. [22], was achieved in this direction which provided an architecture and chip design that allows the definition of tables that match against arbitrary header fields, as described by the programmer. It can be considered fundamental because it proved that Reconfigurable Match-Action Table (RMT) switching is feasible, which formulated the basis for many innovations in the area of programmable switches. To deliver RMT, the packet processing pipeline that was defined in OpenFlow was revisited and extended:

- The parser could be reprogrammed with new fields and modifications of previous ones.
- The output of the parsing stage produces a vector of the received header fields alongside a metadata struct containing router state variables and information like the ingress port. These are input arguments to the ingress processing stage.
- Within the ingress stage, a pipeline of tables of arbitrary number, topology, widths, and depths can be defined, so long as their total size abide to the memory limitations of the device.
- New, more complex types of actions like protocol encapsulations or addition of new header fields can be defined and executed as single-cycle operations.
- Header fields and action data occurring from the match stage can be arguments for action functions. Meters and counters are also available and can impact the processing of future packets without the involvement of control plane.

- Packets can be output in queues with varying queuing disciplines that lead to any subset of ports.
- The pipeline of match-action tables can be dynamic, contrary to OpenFlow which only allowed a forward-pointing table pipeline.

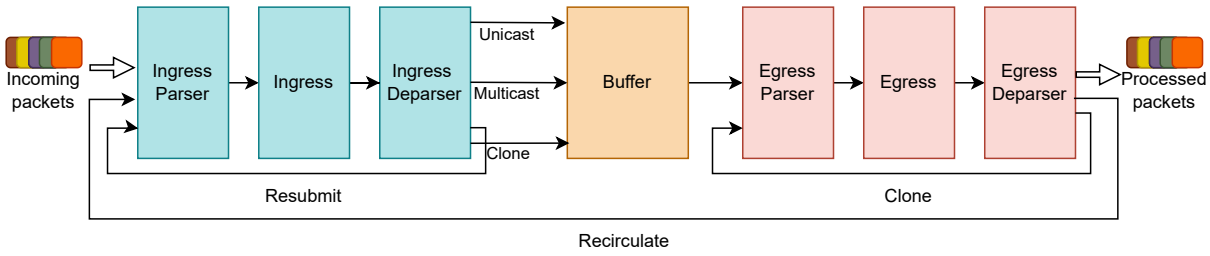


Figure 2.3: The Protocol-Independent Switch Architecture.

RMT was implemented in a switch that supported 64 ports of up to 10Gb/s each, which at the time the paper was released was orders of magnitude higher than CPU, GPU, and NPU router implementations. The obtained throughput was the result of parallel processing of incoming traffic with multiple parsers, pipelines, and action units. Important was also the role of restrictions in the physical pipeline stage to achieve terabit speeds: only a fixed number of match stages can be defined, both because of physical match stage restrictions tied to hardware but also because each stage requires an amount of memory hardware resources (e.g., Ternary Content-Addressable Memory (TCAM), Static Random-Access Memory (SRAM), Hash Units, etc.) which are limited and expensive; one instruction per field is permitted for each stage of processing, with the instructions being simple arithmetic, logical, and bit manipulations; the packet header vector produced after parsing has to be limited in size. RMT was later generalised in a single pipeline forwarding architecture that became known as Protocol-Independent Switch Architecture (PISA). Around 2016, the first PISA-based ASIC, Tofino [93], became available from Barefoot networks (later acquired by Intel). It offered line-rate execution of compiled binaries, proving that programmability does not always result in reduced performance. Commercial implementations hosting Tofino are able to accommodate 100 GbE over 8 ports. The latest iteration of the chip, Tofino 3, promises 64 ports with up to 400 GbE each. More recently, PISA programmable switches have also been available in smaller form factor (half-width rack switches) and can be used in couples for redundancy purposes or be deployed at the edge of the network to potentially facilitate end-to-end programmability [146]. The same stands for end-host devices like smartNICs [145].

Programming Protocol-Independent Packet Processors

With existing hardware architectures offering terabit throughput using the PISA architecture, a common language that can describe packet processing pipelines in high level is necessary for easy definition of new protocols. Low-level programming of the chip requires expertise and increases the complexity of implementing network applications (as demonstrated by NPU implementations). P4 (Programming Protocol-independent Packet Processors) was invented to address this as a language able to describe packet processing pipelines regardless of target hardware [21]. The authors originally saw OpenFlow's limited number of supported protocols as an important obstacle for the deployment of novel protocols and the encapsulation requirements of modern data centres. P4 was originally envisioned to be the extensible data plane component that would complement a newer version of OpenFlow and offer protocol independence and reconfigurable switches. A version of OpenFlow that can interact with P4 has not been available till now and OpenFlow switches are mostly controlled through scripts relying on manufacturer-provided runtime libraries.

P4 is employing a forwarding model that is similar to the PISA architecture to ensure hardware compatibility. Each of the pipeline blocks becomes programmable through the language's keywords. Headers can be specified through an ordered set of lists. A programmable parser implements a state machine operating on header values. Ingress and egress match-action pipelines are separated with a buffer and populated through the control plane. Different types of matching (e.g., exact, ternary) happens against header fields. Actions can be composed out of protocol-independent primitives supported through hardware. Serial or parallel execution of tables is determined by analysing a table dependency graph based on header fields. P4 evolved (from $P4_{14}$ to $P4_{16}$) by removing many core language features and instead placing them in libraries, resulting in a smaller language. In its current version, P4 allows the description of new library elements through the "extern" construct suggesting that new, hardware-specific functionality can be added to the language in a modular manner [63]. Currently, a great variety of use cases make use of P4 to apply stateful per-packet processing in line rates with great success – offering in some cases the fastest available implementation. Example use cases include: in-network telemetry [18], key-value stores [100], and consensus protocols [187]. Some of them were accelerated beyond legacy implementations through using P4 programmable ASICs to offload computations.

P4 introduced helpful abstractions to program a processing pipeline for different types of hardware but manufacturers still have an important role in the deployment of P4 code by providing essential tools to compile and run software on each device. The software development framework, the definition of the device's architecture, and the compiler for the target device are all provided by the manufacturer and constitute essential tools in

the programming process. Match-action table entries in most cases are edited through the manufacturer-provided runtime libraries. Therefore, while P4 development and prototyping can happen independently with software like BMv2 [35], deployment in hardware can be dependent on the idiosyncrasies of the provided software which might introduce unexpected bugs or obscure error codes.

2.6 Network Function Virtualisation

The deployment of middleboxes incurred a significant capital expenditure for network operators and increased complexity in managing the deployed functions due to their fixed-function characteristics. Traffic within the network in some cases has to be redirected to multiple middleboxes in order to apply various NFs, which in turn alters the traffic characteristics and consumes bandwidth. The lack of flexibility in middleboxes makes adaptation to external changes cumbersome and the implementation of new services slow. To address these shortcomings, NFV was proposed. NFV aims to enable elastic service deployment in heterogeneous infrastructure through virtualisation of NFs. By deploying VNFs on general purpose hardware (e.g., x86 servers), general programming tools can be utilised to develop new services and support can be provided for existing ones. Horizontal and vertical scalability can be facilitated through resource allocation to various VNFs based on traffic demands. Co-location of different functions can happen even within the same server, minimising bandwidth consumption and latency. Without the fixed-function characteristics of middleboxes, NFV promises a wide set of deployed services including the ones traditionally deployed in middleboxes (e.g., firewalls, deep packet inspection services, content caches, etc.), scalable deployment that aligns with demand, cost minimisation through hardware repurposing, and decreased configuration complexity compared to middleboxes [29, 32].

2.6.1 Reference Architecture

Network function instantiation, placement and migration decisions, and monitoring of deployed services are accomplished through the NFV orchestration framework. The aforementioned operations are known as Management and Orchestration (MANO) operations. The code base to achieve this functionality can be quite large, with all of the subcomponents being developed as standalone programs with failure-recovery mechanisms. European Telecommunications Standards Institute (ETSI), the NFV standardisation body, defined a reference MANO architecture comprised of the following components:

1. NFV Orchestrator: it accepts the desired policy for certain VNFs (or chains thereof) and is responsible for their high-level control and monitor of their health. It dis-

seminates instructions to other sub-modules to retrieve the state of health for the deployed VNFs and react to events in order to maintain the KPIs of the specified policy.

2. VNF Manager: deploys the desired services within the virtualised environment and configures them according to the parameters defined by the orchestrator. A link to a VNF catalogue can be achieved for standardising the core of deployed services and only configuring certain parameters upon deployment.

3. Virtualised Infrastructure Manager: it monitors and manages the underlying network infrastructure. It is responsible for the allocation of physical or virtual resources that will be used by the deployed VNFs. ETSI acknowledged the importance of SDN in managing the underlying network infrastructure and relied on OpenFlow for rule installation and management.

4. Data repositories: hold structured data that describe the state and health of running VNFs, the infrastructure resources, and the deployed services.

Network operators relying on NFV frameworks, can utilise the flow of information between the components to perform real-time monitoring of the deployed services and define policies that maintain certain KPIs. This is known as intent-based networking and is one of the most fundamental differences between NFV and traditional networking.

2.6.2 Virtualisation Technologies

NFV achieves separation of the executed binary from the underlying infrastructure through virtualisation. To preserve this property, an isolated environment with dedicated resources needs to be defined for application binaries. This can be achieved through various technologies that allow hardware resources to be allocated to different isolated instances. Virtualisation can be primarily offered in two levels:

1. Hardware-level virtualisation: server resources can be managed by a hypervisor which exposes them to virtual machines, each with a complete OS and the necessary libraries to implement the desired application. This has been the predominant virtualisation method across mainstream data centres.

2. OS-level virtualisation: server resources can be managed through the OS and a software engine running above it can provide access to resources for encapsulated applications running above it. Application dependencies and code are encapsulated in a single software unit that is known as container.

Virtual Machines (VMs) rely on hardware-level virtualisation to deploy a full OS alongside the necessary libraries and binaries. They offer great flexibility in terms of software

stacks and provide good isolation for the binaries they host. Apart from data centre applications, they have been extensively used for VNFs deployed in 5G, with slices being usually dependent on a VM to host the desired functionality.

Containers belong to the category of OS-level virtualisation. By encapsulating only the required dependencies, they operate on top of a software engine that is responsible to interact with the OS to allocate resources and ensure isolation. Containers using the Docker system, the most popular system to date, were originally designed for deployment over the Linux kernel (using `cgroups`, namespaces, etc.) but a separate software engine was later developed for other OSs, like Windows. Containers offer significantly smaller footprint compared to full VMs which allows for quick deployment across various host machines. The performance of containers has been shown to be similar to bare-metal deployments, with the main source of performance degradation for containers being multi-tenancy in large scales [180]. They can be executed in various types of hardware, so long as the engine is supported by the host OS.

The small footprint and quick deployment of containers led to their integration in agile software development. Services deployed using this level of virtualisation are also known as microservices. Deployment of VNFs in the form of microservices has some architectural advantages over monolithic VM deployment: it minimises redundancy by separating core VNF functionality in microservices; each microservice can be scaled separately based on demand [32]; redundancy costs are minimised and fine-grained MANO operations can be conducted over the deployed functions [32].

2.6.3 Placement & Resource Allocation

NFV presents various algorithmic challenges with open-ended questions. One such topic which covers multiple optimisation problems is that of resource allocation. In NFV, resource allocation requires taking into account both physical and virtual resources. The NFV framework is responsible to map virtual to physical resources and dynamically change the ratio between the two types. Physical processing resources can host multiple VNFs, either chained together or by forming layers of virtualised resources [58]. This type of problem bears resemblance to the Virtual Network Embedding (VNE) problem, where a set of virtual resources has to be allocated to a set of physical resources in an offline or online manner and the decision has to be optimised for cost, link bandwidth, energy efficiency, etc. However, for resource allocation within a NFV framework, VNE is not directly applicable. The orchestrator deals with placement requests for chains of interdependent VNFs that form NFs, instead of requests for deployment of single NFs in the case of VNE. Moreover, the topology changes dynamically and resource demands also change over time even for NFs that have been already deployed [131]. In most cases, traffic flow requirements have to be mapped to VNFs and not vice versa.

The clear advantages of NFV have driven research on the placement and resource allocation problem. Placement decisions have to be mostly dynamic, with VNF requests describing arbitrary resources for random time periods. The orchestrator itself can be distributed or centralised, depending on the framework capabilities. Redundancy of deployed VNFs can also heavily impact resource allocation decisions as the need for resources scale drastically with high redundancy requirements. Balancing the aforementioned requirements in an online, efficient manner is a challenging task for the VNF orchestrator. The work of Herrera and Botero was among the first to define the three stages of NFV resource allocation [73]:

1. Chain composition: the formulation of chains of VNFs by taking into account dependencies between them and their behaviour (e.g., potential traffic gains). Chain formulation can impact throughput, traffic cost, latency, etc., and can therefore be optimised towards these aspects [138].

2. Forwarding Graph Embedding: processing the Forwarding Graph produced in the chain composition stage as input and subsequently matching the available network resources to satisfy chain requirements. This problem can be considered as a generalisation of the VNE problem that includes a broader spectrum of available resources, like storage, processing capacity, etc. It is therefore an NP-hard problem as well.

3. Scheduling: arranging the order of execution of VNFs to minimise total processing time based on the set of available resources and their processing requirements.

To be able to mathematically formulate problems for these stages and generate solutions, it is common that a linear objective function is formulated which, when minimised (or maximised) around a certain variable, it provides the best mapping of the available resources to the set of functions that need to be deployed. This is known as the Integer Linear Programming (ILP) optimisation method. ILP models, including mixed ILP and binary ILP, provide accurate solutions for resource allocation and placement problems but by being NP-hard they require a lot of resources and scale poorly with network size. Similarly, non-linear programming models, i.e., containing non-linear constraints, present NP-hardness and are therefore not scalable enough to be integrated in real-world orchestration frameworks. This is why most of the research on VNF placement is using ILPs as an accuracy comparison baseline for a proposed heuristic alternative, which significantly reduces the execution time but only provides near-optimal solutions. The proposed heuristics are novel and specialised for the problem in question. They follow different approaches to reduce complexity, usually through ranking the solutions of the results space and picking the best one [44].

Luizelli at el. provided one of the first ILP formulations of the placement problem considering VNF chains and the performance impact of allocating more resources (e.g.,

CPU, memory) per VNF [131]. Through the proposed model, authors managed to reduce end-to-end latency and resource over-provisioning. This work considered some basic NFs types: load-balancer, firewall, Network Address Translation (NAT), without differentiating placement based on their characteristics. The placement problem has since been further contextualised to consider the type of deployed network function and the environment in which it is deployed. For example, Ali et el., by extending the work of Basile et al. [16], consider security VNFs in the context of a Cloud data centre [4]. Unlike generic VNFs, security VNFs in the cloud showcase intricacies such as custom traffic processing requirements which are differentiated for each tenant based on their individual requirements from a security function [4]. To address this, security VNFs are classified using traffic requirements and then placement decisions are trying to maximise residual resources.

Lately, Machine Learning methods have been proposed as the optimisation technique for placement problems [109, 184]. Machine Learning methods present certain strengths in the context of VNF placement. Their ability to create statistical correlations of dynamic traffic changes and particularly hidden patterns of traffic variations is particularly important for prediction of demand and reallocation of resources [79]. Furthermore, trained machine learning models present low computation time, especially when compared with numerical methods [184]. For each deployment scenario, a machine learning model that will satisfy the deployment requirements and provide the statistical accuracy that is required has to be picked. Then it has to be parameterised accordingly. The process of selecting and evaluating the best model for different scenarios is beyond the scope of this work, however relevant papers are cited to inform the reader of potential solutions to the placement problem, add insight towards what is currently feasible, and help assess the contributions of this thesis.

Flow-Aware Placement

Through research, arguments have been formulated on the benefits of including QoS awareness and flow handling in the placement problem. It has been proposed that placement decisions should be optimised around a given traffic matrix instead of placing VNFs and then making traffic routing decisions. Sahhaf et al. proposed a novel, flow-aware NF decomposition model based on a heuristic that allows scaling of VNFs based on demand, successfully increasing the acceptance ratio of incoming service requests [176]. Kuo et el. attempt to balance path length and VM usage, i.e., make use of shorter paths without spanning multiple new VMs. By doing so, they maximise the amount of available resources and remaining link capacity [118]. Similarly, Mechtri et al. aim to offer joint VNF placement and chaining that is able to adapt to complex use cases proposed by ETSI. The work integrates concepts like multi-tenancy and resource heterogeneity, being among the first to consider the combination of physical and virtual nodes [136]. They propose an eigende-

composition of the requested graph and the infrastructure graphs to solve the problems of VNF placement and traffic distribution. Future models have become even more inclusive of real-world requirements, taking into account: vertical services, VM setup times, and KPI requirements. One such example is the work of Golkarifard et al [74], which provides a mixed ILP that is able to make joint decisions on a number of different deployment parameters: VMs activation, VNF placement, CPU resources, and routing.

Iordache-Sica et al. [95] integrate a flow matrix in a context-aware placement problem of security-oriented VNFs to minimise path latency when compared to a non flow-aware ILP. However, due to the dynamic nature of networks, which is emphasised at the edge, frequent re-calculation of the placement decisions is required to adjust to changes. A traffic matrix makes this process more complex. This was effectively addressed in the work of Cziva et el. with the use of optimal stopping theory which informs the best moment to re-execute the placement algorithm in order to preserve optimality despite changes in the network [39].

Placement at the Edge

Placement algorithms have also been adjusted for the network edge where processing resources are limited, VNFs need to accommodate bursty traffic and be optimised for user mobility. Whereas, functions placed at the core of the network usually process steady, high-throughput traffic and can use more processing resources. Apart from withstanding volatile behaviour and optimising resource expenditures, placement approaches need to provide guarantees on delay for the end user, as this is the primary goal of edge computing. The aforementioned work of Cziva et al. developed a latency-optimal VNF placement method for the edge [39]. It can dynamically schedule containerised VNFs with the best possible QoS. Zhang et al., motivated by the variety of 5G use cases (like autonomous driving, 4K video transmission, etc.), acknowledged the differences in latency and throughput requirements per slice [206]. They suggested a placement model that is using each slide's latency and throughput characteristics as an input. They proceeded to study the impact of VNF consolidation on performance degradation by employing a demand-supply model to quantify the interference in terms of throughput reductions. VNFs are gradually consolidated after deployment without violating throughput requirements. The topic of performance interference between co-located VNFs has been further investigated by Zeng et al., without however any proposals for interference-aware placement [203].

The body of work for placement at the edge has been extended by capturing the need to deploy and manage chains of VNFs without voiding latency limitations. In doing so, there is effort to best utilise all the available resources. Jin et al. have developed a mixed ILP that is able to solve the VNF chain deployment problem while minimising the total resource consumption of edge devices [99]. To achieve this, they use a two-stage placement

approach: a constrained depth-first search algorithm for path selection, and a path-based greedy algorithm for VNF placement with minimum resource consumption. The approach demonstrated near-optimal performance during evaluation. Other work by Nguyen et al. addresses the VNF placement problem in multi-cloud deployments with interconnected IoT gateways [148]. The authors formulated a non-convex ILP to establish the optimal solution for the problem. Furthermore, they proposed a Markov approximation technique and a heuristic-based approach for near-optimal solving with faster convergence times. They proved significant reductions in bandwidth and computation cost by considering the IoT topologies during VNF placement.

Energy-Aware Placement

Placement approaches primarily attempt to optimise towards a network-oriented goal, which is usually latency, throughput, etc. In the literature however, there have been placement algorithms with optimisation goals that are peripheral to core networking metrics, like energy efficiency – a matter well regarded both in data centre networks and edge environments. Placement approaches that target energy efficiency often aim to minimise the number of active nodes in the network, or enforce policies that reduce the power consumption. Tajiki et al. have been among the first to propose an ILP and a heuristic approach to minimise energy consumption without voiding defined QoS Service Level Agreements (SLAs) [186]. Their placement method tries to minimise the number of required servers to run VNFs without harming QoS in order to minimise energy consumption. The results show moderate reductions in energy consumption for most cases when compared with standard ILP solvers. Xu et al., apart from server energy consumption (based on CPU usage), they also consider link energy consumption through its on/off state and its bandwidth utilisation during placement [198]. Varasteh et. al propose a framework to solve the power-aware and delay-constrained joint VNF placement and routing problem [192]. In the framework's first phase, a centrality-based ranking method maps the VNFs to physical nodes. In a second stage, the delay budget between consecutive VNFs is split, and the shortest path through the selected nodes is found using the Lagrange Relaxation Aggregated Cost algorithm.

The works of Abd et al. propose energy efficient scheduling algorithms in a cloud computing environment to place tasks and reduce energy consumption [1]. They try to minimise VM migrations and strike the right balance of machine utilisation before bottlenecking performance using pre-defined thresholds. Evaluation with the use of synthetic data and Google trace logs showed that their placement algorithm reduces total resource utilisation and energy consumption when compared to static allocation or scheduling strategies. Marahatta et al. also propose a placement algorithm for cloud environments that includes a fault-tolerant scheduling scheme alongside minimisation of energy consumption

[134]. The results showed decreases in rejection ratio and energy consumption. However, the number of migrations might have a negative impact on energy consumption and it is not included in the proposed placement method. In a similar manner, Ding et al. have proposed a Q-Learning algorithm to schedule tasks using an energy-efficient method [47]. Their approach aims to minimise the task response time and maximise the utilisation of a node's CPU therefore improving resource utilisation and energy consumption for the infrastructure.

This thesis explores energy-aware VNF placement for edge computing, which is a combination of technologies usually found in cloud infrastructure with environmental challenges found at the edge, like limited processing and energy resources. This combination creates new optimisation problems related to resource allocation but also reveals challenges related to the systems aspect, like the integration of sensor measurements in a reliable way in the orchestrator. In the case of edge nodes managing wireless sensor networks and simultaneously hosting VNFs, the placement problem becomes more complex as energy measurements need to be taken into account for both the edge nodes and the wireless sensors in a way that will maximise the total lifetime of the network and allow discrepancy between device errors and battery depletion.

Several works aim to estimate battery State of Charge (SoC) by using different methods. Hu et al. present the Extended Kalman Filter technique as the SoC estimation algorithm [89]. The researchers evaluate the proposed estimator using two types of Lithium-ion batteries under different loading profiles and temperatures. The optimal model parameters used in the Extended Kalman Filter are obtained from generic functions for battery modelling that combine several degrees of polynomials [89].

Soon et al. have proposed an enhanced coulomb counting method for estimating the SoC and the State of Health of lithium-ion batteries [147]. They improve the estimation accuracy by considering the correction of the operating efficiency and the impact in the State of Health. The proposed method can be easily implemented in all portable devices, such as SBCs, due to simple calculations and low hardware requirements. Pop et al. propose a new SoC algorithm by combining direct measurement of the Electro-Motive Force and Coulomb counting [161]. They demonstrate the effectiveness of their approach by improving the SoC and accuracy of the remaining run-time.

In contrast to the previous work, the authors of [196] have presented two methods for actual bias modelling of batteries. They have proposed a polynomial and Gaussian process regression model using a typical battery circuit model to examine the bias modelling and the SoC estimation. The results of their model show a significant improvement in comparison with the baseline models (i.e., first- and second-order resistance-capacitance models) while being able to maintain similar computational efficiency.

For the management of the lifetime of wireless sensors, early work by Lim et al. pro-

posed an adaptive algorithm that is able to determine the sampling schedule of wireless sensors [125]. The algorithm is trying to adjust data gathering based on user accuracy goals, network connectivity, and preliminary collected data. Results show substantial improvements over baseline algorithms in terms of network lifetime. The proposed scheduler covers only the lifetime of wireless sensors and does not consider energy consumption of the base station.

Gazori et al developed a Deep Reinforcement Learning approach that tackles the task scheduling problem in fog IoT applications considering both edge and cloud infrastructure [69]. The main function of the proposed scheduler is to decide whether to process the task in a fog node or send it to the cloud data centre. The authors include an energy consumption model in the scheduler's proposal to guarantee selection of the most appropriate VM in terms of power consumption.

2.6.4 Network Function Virtualisation Platforms

In this Section, an overview of representative orchestration platforms is presented. The reference architecture, presented in Section 2.6.1, has been adapted (or changed) by the various platforms to favour certain properties (performance, robustness, scalability) based on the intended use cases. The presented platforms have been successfully deployed in different scales by mainstream operators and have proved the feasibility of relying on NFV for deploying virtualised services.

Open Network Automation Platform

The Linux Foundation, alongside mainstream telecommunication providers have developed the Open Network Automation Platform (ONAP) framework [67]. ONAP is designed to support the deployment and management of VNFs in commercial deployments, currently used to manage deployments for Nokia, Ericson, and Huawei [19]. ONAP's hardware-agnostic architecture is comprised of the Management Framework – which follows a microservices-based approach to monitors other ONAP components, the Design Framework – which leverages a declarative modeling language for the definition of resource requirements of each service, and the Run-time Framework – which manages the deployment of and deletion of microservices and allows real-time monitoring of the infrastructure. ONAP offers modularity through northbound and southbound interfaces, used to extend its functionality with the use of other operating systems and platforms.

Openstack

With a primary focus on ease of deployment, Openstack [64], is a widely used Infrastructure as a Service (IaaS) platform that allows the deployment of VMs, containers, and bare

metal workloads. For the management of containers, it supports orchestrators like Docker Swarm or Kubernetes. It relies on an extensible, open-source architecture that enables flexibility for deployment in heterogeneous infrastructures [207]. However, Openstack's forwarding performance appears to be impacted in large-scale deployments due to either layer-2 flooding or oversize routing tables and poor synchronisation [205].

Open Source Mano

Open Source Mano (OSM) is a NFV platform developed by ETSI that follows the reference architecture defined by the same body. OSM is one of the first NFV platforms to be released. OSM's scope is to provide an interoperable, open-source, and scalable framework to manage VNFs for the purposes of 5G networks. OSM was initially oriented towards VM orchestration but the appearance of lighter virtualisation methods, like containers [49], and the proliferation of physically deployed NFs contributed to changes in the scope of the platform. Its community now plans to incorporate all of the available NF deployment methods and make OSM a high-level orchestrator that can manage services regardless of the technology they rely on. Its architecture is envisioned to support unified orchestration across cloud and edge, providing abstractions that are ideal for heterogeneous operator environments and offering automation in service monitoring and management. As such, OSM is more popular in 5G environments where various verticals are anticipated to be supported under one common infrastructure. To make this possible, 5G utilises network slicing through softwarisation in both the Core Network and the Radio Access Network [60]. NFV MANO and OSM are key components to this vision [104]. This vision aligns well with hot topics in the research frontier, where end-to-end programmability becomes an increasingly important and complex topic.

Kubernetes

Kubernetes is a container orchestration platform that has been extensively used in production environments. It manages Docker containers enclosed in pods – an overlay structure that enables MANO operations and network handling by Kubernetes. The source code is open and extensible, making it ideal for experimentation with various deployment scenarios. It offers a fault-tolerant control plane through replication and failover mechanisms. Kubernetes relies on a modular architecture with each of its components running within stateless pods that can be restarted upon failure.

Kubernetes' architecture is split between control plane components (frequently referred as Master node) and worker node components. An overview is shown in Figure 2.4. Central to Kubernetes' control plane is its API component which handles front end traffic and interacts with all other components to manage pod deployments. Kubernetes relies on etcd for fault-tolerant storage of the control plane data. Etcd is a standalone platform that offers

consistent, distributed, consensus-based KVS storage [51]. By default, Kubernetes runs etcd within a pod that can be replicated and accessed from the API in order to store lease information, pod health status, etc. Etcd can also be installed as a standalone application that is linked with Kubernetes using authentication keys allowing further customisation of its setup like high-availability replication, setup on custom hardware, etc. Pods are assigned to nodes and scaled through the scheduler component. The scheduler first filters the nodes based on their available resources and the defined deployment requirements like deadlines, hardware, software, data locality, etc. It then ranks the candidate nodes based on a score that describes their availability of resources with the top candidates being selected for hosting the pod.

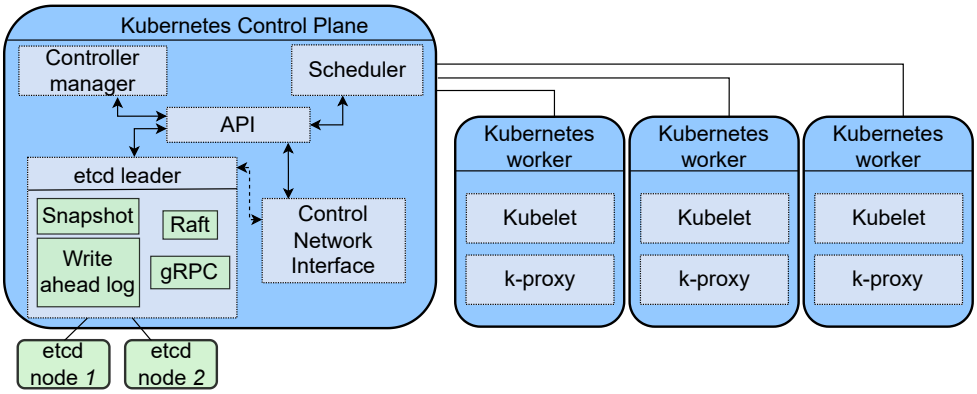


Figure 2.4: Kubernetes architecture overview [9].

By default, worker nodes host two types of services: kubelet – ensures that the defined pod specifications are executed through health monitoring and restarts of the failed pods, and kube-proxy which manages network rules to facilitate communication between pods, the master node(s), and outside traffic. Worker nodes also interface with the underlying container runtime (usually Docker).

2.7 In-network Offloading

The proliferation of bespoke networking hardware able to perform stateful packet processing at increasingly high throughput and the appearance of virtualised NFs have created new possibilities for compute within the network fabric. Stateful processing in traditional networks required access to an end-host CPU or fixed-function middleboxes. Instead, modern hardware deliver programmability directly within the network fabric. Advances in the area of programmable switches, both in programming tools (P4 [21], PSA [78]) and in hardware implementations (Intel Tofino [93], Broadcom Trident [24]) have allowed line-rate performance for compiled binaries executed on programmable switches. These enablers have driven innovation in the area of in-network compute. Researchers have used

such technology to offload computation primitives in PDP, significantly improving the performance of the applications that rely upon these primitives [123, 162, 197, 178]. New protocols can be developed and previous ones can be extended with new features without traffic rerouting or acquiring new hardware.

2.7.1 Data storage

With the growth in demand for online services like cloud, web services, online gaming, etc., the performance and scalability of storage solutions has become a critical design aspect for the developers and administrators of replicated storage platforms. For example, in the case of Facebook, the total photo storage grew from 1.5 petabytes to 20 petabytes just between 2009 and 2010 [17]. At the same time, the traffic generated by users doubled, making time-efficient retrieval of the requested data a challenging task. To deal with performance requirements, data engineers employed various solutions both in terms of data structures and hardware.

In order to respond to workload demand with minimum upfront investment, data centres followed a scaling out strategy by acquiring and deploying more commodity servers in response to projected demand increases. In this manner, adequate amount of redundant resources can be budgeted towards availability targets to ensure fault tolerance and fault recovery. However, relational data models incur increased overhead in data retrieval among distributed normalised data [86]. In order to increase query capacity and performance, scaling up is required through upgrading the used computational nodes. This requires significant upfront investment and a higher risk of downtime in case of failures. It was evident that relational databases were not an efficient tool for the growth strategy of data centres and big-data applications.

Furthermore, the scale and throughput requirements of modern applications posed software development challenges. Modern applications required the conception of data as a single, highly available entity with low latency access [86]. This development abstraction alongside the aforementioned performance, availability, and scalability requirements led to the emergence of NoSQL data stores: schemaless data models that would allow relaxed consistency in favour of performance without harming the application's functionality. NoSQL data stores did not constitute the silver bullet for all of the above [42]. They simply offered design flexibility by promoting specific design goals, like availability and consistency with trade-offs in partition tolerance, as explained in the CAP theorem [23].

KV replication methods are commonly separated between quorum-based and primary-backup methods. Quorum-based methods require the majority of the participating nodes to conduct an operation in order to render it successful (e.g., Paxos [120]). Primary-backup methods operate by appointing one of the participating nodes as the primary (or reference) node. Operations are considered successful when they are executed on this reference node.

The rest of the nodes act as replicas of the primary one. E.g., Chain Replication [191].

KVSs have quickly become a popular NoSQL data store due to their simplicity and efficiency. They store data based on a key-value tuple: keys are used to perform lookup operations on the values. They constitute a lightweight, highly-scalable storage and coordination service and provide design flexibility that enhances the performance of relational database implementations [17]. KVSs have hosted a range of use cases, from graph data caching to storage pipelines involving machine learning [181, 43, 25].

Different types of hardware were used for writing commits to balance persistence with performance in NoSQL. To favour persistence, direct writes to Hard Disk Drives or Solid State Drives were used (e.g., Oracle NoSQL [154]). On the other hand, performance could be prioritised by writing directly to memory (e.g., memcached [139]). Hybrid platforms are also available in order to achieve the desired balance by first using memory cells and subsequently transferring them to disk for persistence (e.g., [130].

Transferring Key Value Stores in the Data Plane

By examining a query's path in KVS implementations, network devices appear to be reached first in order to transfer the query to the designated server. By generating responses as close to the client as possible, and in this case the closest network device, an opportunity to significantly reduce the transfer time of the query to the server is presented. Latency is expected to decrease analogously, given that responses are transferred with the least possible delay back to the client. Furthermore, there are potential improvements in the attainable throughput of generated replies, with programmable switches able to generate packets in data rates that are orders of magnitude higher than the throughput obtained from conventional KVS implementations.

The IncBricks paper was among the first to capture the potential of PDP devices because of their location in a query's message path [126]. The message path for a query in a legacy KVS would start from the client, go through network devices, and finally it would reach the servers that host the KVS. IncBricks substituted the need to reach coordination servers for some types of queries and suggested a shorter path for cached values which stays between the client and the network devices. Replication among the participating switches was based on a hierarchical directory-based coherence algorithm. Network programmability was achieved with the use of NPUs which offered limited instructions and were tied to proprietary programming tools provided by the manufacturer. Despite this limitation, the realisation of this new role for network devices greatly reduced the query response time and increased the attainable throughput over legacy KVS deployments.

NetChain and NetCache followed up as a couple of the most prominent works in the area of in-network caching [100, 101]. By accommodating queries entirely in the data plane, NetChain is effectively the fastest in-network KVS platform that exists today. Other

important works perform offloading of certain KV processes, such as conflict detection [187, 209], which offer performance improvements over legacy storage. NetChain is explained in more detail in Section 3.2 where its functionality is analysed and certain limitations of its design are identified.

In-Network Conflict Detection

Other platforms for network-accelerated data storage offer partial offloading of data storage functionality within the network fabric, like conflict detection operations. For example, Harmonia attempts to provide scalable replication without sacrificing linearizability, i.e., correctness is preserved upon concurrent modifications of an object [209]. Harmonia treats the memory of a programmable switch as storage space for the set of contended objects. It uses a shim layer to communicate with mainstream KVS platforms (e.g., Redis), which act as the main storage. Through performing in-network conflict detection, Harmonia can direct queries to any replica with the latest version of the object, improving scalability of legacy replication methods. It reduces the need for multiple message exchanges to identify an up-to-date replica, replacing it with look-ups in the line-rate accessible memory of the switch and subsequent routing to up-to-date replicas. One key assumption is that workloads are read-mostly, which is expected for data centre KVS traffic. By monitoring write requests, for each object it stores the following metadata in switch memory: a monotonically increasing sequence number, a set of dirty objects and their highest sequence number, and the latest write committed by the switch. Switch memory, being a scarce resource, can be a limiting factor to the amount of simultaneous writes Harmonia can deal with. Through evaluation, NetChain appears to have up to three orders of magnitude higher throughput than what Harmonia can achieve [209]. This is expected since KV replies can be generated directly in data plane, eliminating the need to contact a server. Despite this, Harmonia still manages meaningful performance improvements (up to millions of queries per second) over legacy replication methods.

Similar goals and practices are adopted by Flair, which is designed to work on top of quorum-based protocols (e.g., Raft [153], Zookeeper [91], etc.). Differently from Harmonia, Flair focuses on quorum-based approaches and deploys modules (shim layer) on top of all participating followers (storage nodes). Updates on followers are subsequently committed to the leader node which then stores them in a central programmable switch. The switch is able to direct queries to storage nodes based on the list provided by the leader. This centralised allocation of queries allows load balancing among up-to-date storage nodes and awareness of the load of each node. The switch can also drop replies with suspected stale data, in case there are pending writes concurrently with a read reply. Through evaluation with synthetic workloads (generated with 95:5 read-write ratio using YCSB [102]), it is shown to achieve superior performance to legacy replication methods: Flair

achieved higher throughput than legacy alternatives and appeared to be less susceptible to diminishing performance due to workload skewness. Similar to Harmonia, Flair relies on external storage nodes and it is therefore adopting the performance limitations of this design decision which are not present in the case of fully offloaded platforms in PDP.

Data Storage at the Edge

Service function chains extend networking functionality to the network edge and therefore devices operating at the edge also perform computations necessary for traffic routing, provisioning of security services, storage of configuration data, etc. In parallel, programmable switching hardware becomes available in smaller form factors, like the APS2112D switch which is the smallest form-factor Tofino switch available [146]. The synergy of programmable networking devices and IoT devices being able to execute accelerated user space packet processing places in-network compute applications closer to the end-user. To this end, in-network storage at the edge is examined as a key technology that can aid the convergence of cloud and edge infrastructure.

An example of work that considers data management at the edge is the Dragon scheme, able to identify nodes that can reply to users' requests based on criteria describing nodes themselves and their data [112]. Huacarpuma et al. propose a distributed data service providing functionality for data collection and processing [37]. The objective is to enable multiple IoT middleware systems to share common data services covering interoperability issues. The parallel execution of queries increases the performance of the applications and at the same time, analytics are retrieved by different nodes aggregating them to deliver the final outcome. In edge environments, it is of high importance to obtain a view on the data statistics on each node since the relevance between the data and analytics queries can be estimated [113], [115], [114]. Multiple efforts handle the problem of allocating data to specific nodes. Balkesen et al. provide a mechanism to partition data streams on-the-fly taking into consideration the query semantics [12]. Cao et al. propose a multi-route optimiser exploiting intra- and inter-stream correlations to produce effective data partitions [26]. Other schemes propose the separation of streams into sets of sub-streams over which queries are executed in parallel [70, 202].

Query engines for the IoT domain provide results in real time and data are processed on the devices at the network edge [195]. Such edge-centric processing is important in real-time, mission-critical applications such as self-driving vehicles [31]. Multiple works utilise edge-centric processing in the literature for various applications [3, 46, 143, 168, 84, 88]. Some efforts deal with the automated separation of queries into two sets: queries processed on edge devices and queries processed in the cloud [75]. Quoc et al. adopt statistical learning to recommend a previously generated query plan to the optimiser for a given query [167]. The objective is to predict the query execution time for workload management

and capacity planning. The delivery of edge analytics involves communication efficient predictive modelling within the edge network [83]. Analytics are derived by models dealing with dynamic optimal decisions for data delivery in light of communication efficiency [84, 30]. Several schemes exploit the computational capability of edge nodes to launch algorithms directly at the data sources [5, 68, 105].

2.7.2 Machine Learning

The work of Li et al. explored the impact of communication delays in distributed Reinforcement Learning [124]. By examining state of the art distributed Reinforcement Learning systems, they observed that: either workers had to exchange local gradients with the centralised parameter server – incurring all-to-all communication for updating the weights of each worker, or the weights had to be exchanged in a circular, distributed manner among the workers. They have calculated that message exchanges for gradient aggregation occupy up to 83.2% of the time for each training session. By using stateful packet processing and simple arithmetic operations, they accelerate the training process through in-network aggregation of gradients and distribution of updated weights to the workers, effectively repurposing the switch to a Reinforcement Learning accelerator. According to the evaluation they conducted, the training time was sped up by up to 3.66× for synchronous and up to 3.71× for asynchronous training. However, the implementation does not cover multi-rack topologies nor does it include a mechanism to prevent lossy traffic.

In a similar manner, SwitchML identifies all to all communication patterns and suggests a more generic aggregation mechanism for Machine Learning [178]. In SwitchML, computation is partitioned between end hosts and switches in order to best utilise the computational capacity of both systems. End hosts perform quantisation of floating numbers before submitting them for aggregation in order to enable arithmetic operations in the switch. End hosts are also responsible for failure recovery. The switch performs fixed point aggregation and dispatching of updates. The authors managed a 2.27× speedup in training times over a 100 Gbps network, with training times improving when using faster GPUs due to the minimisation of computation/communication ratio.

2.7.3 Aggregation

Another type of in-network computation which, similarly to data storage, is common between both edge and data centre environments is data aggregation. At the edge, the data obtained from sensor readings that require transferring to Wide Area Networks and processing nodes are usually low-dimensional and therefore small-sized. As a result, sensor data encapsulated to packets consume a small amount of the total packet size with most

of the packet size allocated to layered headers responsible for transferring data. For example, Wang et al. calculate that in Sigfox (a prominent IoT protocol) a 12-byte payload over 42-byte header consumes just 28% of the network bandwidth – with 78% used for transferring header fields [194]. The solution proposed by Wang et al. suggests aggregating multiple IoT packets under a single transfer packet to minimise processing overheads. The solution is implemented in P4 and consists of one aggregation switch transferring traffic to Wide Area Network and one disaggegation switch responsible for unpacking the large payloads of transfer packets back to single IoT packets. A similar work has been presented by Madureira et al., performing data aggregation at layer-2 and confirming that data aggregation can reduce communication costs in IoT-cloud communication [133, 110].

In data centre environments, applications that distribute data across multiple worker nodes, like distributed big data analytics, machine learning, stream processing, appear to be bottlenecked on communication cost, with message exchanges between the participating entities consuming a major portion of their execution time [177]. These are also the types of applications that would yield the best results from data aggregation, as the communication cost can significantly be reduced [178]. Data aggregation usually requires simple arithmetic operations which current version of programmable switches are capable of conducting at line rate. SwitchML used programmable switches to offload machine learning model updates [178] in an all-reduce manner (as mentioned in Section 2.7.3). A similar proof-of-concept implementation has been conducted by Sapio et al. [177], proposing a hierarchical aggregation tree using the map-reduce principle over programmable switches.

2.8 Summary

This chapter presents a series of technological and theoretical shifts in computer networks that support the assumptions and the contributions of this thesis. It explains the route towards network programmability as a series of efforts to efficiently utilise the available hardware: starting from attempts to use commodity hardware for packet processing, the proliferation of middleboxes, the contributions of OpenFlow in delivering intent-based networking and, finally, the use of bespoke programmable hardware architectures and VNFs. At the same time, this chapter presents work that shows how network expansion brought heterogeneity, pushed service deployment at the edge, and increased management complexity by requiring balanced allocation of diverse resources to dynamic processing requirements.

This chapter shows that managing such dynamic environments requires storing and updating metrics-related information in various parts of the network. In the data centre, NFV orchestrators require distributed data replication for failure recovery and hierarchical management. Similarly, distributed control plane deployments require real-time updates

on the status of the participating devices across multiple control plane instances. At the edge, placement decisions involve the processing of resource utilisation metrics. The following chapters are shaped by these findings and discuss research outcomes on: distributed data replication within programmable switches; the efficient orchestration of VNFs at the edge; and the synergy of cloud and edge computing for efficient data representation.

Chapter 3

Replicated Storage in the Data Plane

3.1 Overview

Distributed data replication has been primarily used as a way to deliver coordination services for many mainstream applications, as explained in Section 2.7.1. KVSs have been used as the primary storage solution for hyperscale web applications like Google Ads [181], mainstream social media like Facebook [150], data centre software like Amazon AWS [43] etc. Originally, KVSs were used to accelerate MySQL database systems as an in-memory caching solution for frequent queries. This is apparent in the case of Facebook where memcached servers constituted 33% of the total amount of storage servers [155].

For KVSs, computation offloading in PDP not only enables line-rate generation of replies but effectively reduces the amount of hops necessary in a query's path. The Round-Trip latency is reduced in half by generating a reply from the first network device in the path instead of traversing all the way to the server that stores KV pairs. Given the instrumental role of KVSs in providing configuration management [51], locking mechanisms [11], and web-service related operations in large-scale data centres [181], the potential for performance improvement from PDP deployment is significant.

Distributed storage is also at the core of network configuration frameworks, responsible for state sharing between entities of the control plane. As shown in Section 2.5.3, state needs to be shared among hierarchical SDN controller entities for consistent network configuration. This is also true in the case of distributed or microservice-based NFV orchestrators. Rotsos et al. classified data replication and consistency among distributed VNF orchestration entities as an important challenge for network evolution [170]. It is evident that distributed data stores are a backbone tool for large-scale services but are also used for network configuration and VNF management.

This chapter presents NetCRAQ, a new data replication platform that can be placed within programmable data plane devices by leveraging their protocol-independent processing pipeline. The Static Random-Access Memory (SRAM) memory of programmable

switches is used to store state-related data and make them retrievable in line rates. The necessary protocols that deliver KVS transactions and consistency among the participating devices are also presented.

NetChain, transferred the original idea of IncBricks to accommodate queries in PDP and delivered an in-network KVS that is able to generate responses to queries with the line-rate performance of programmable ASICs. Using PISA and the P4 language, the authors manipulated the registers of programmable switches to store KV pairs and generate replies based on a custom protocol layered over User Datagram Protocol (UDP). NetChain's implementation had provided the fastest in-network KVS which, however, presents limitations that make it less appealing for large-scale deployments in a data centre environment and make it slower compared to NetCRAQ. Its Chain Replication mechanism requires full chain traversals to fetch values from the appointed reference node in order to maintain per-item consistency among the participating nodes. In a data centre environment, this results in generating packets that require multiple hops between switches to fetch a value, and then return this value back to the source of the query. A repercussion of this is the generation of high volumes of traffic directed towards a single node, which can cause traffic hot-spots within the topology and eventually lead to link saturation. Moreover, NetChain's packets place the IPs of participating nodes inside the header, rendering its size dependent on the number of participating nodes. This design can make the total packet size arbitrarily large and can cause increases in parsing times which, in combination with repeated chain traversals, can results in performance losses and inefficient use of resources [82].

NetCRAQ is able to accommodate queries entirely in the data plane in order and maintains the sub-Round-Trip Time (RTT) latency demonstrated in NetChain. The design limitations like high traffic generation and full chain traversals are addressed by adapting a different replication method, Chain Replication with Apportioned Queries (CRAQ) [190], to work under the PSA architecture. Packet control logic complexity and hardware memory are traded for performance and to provide better scalability.

The scalability achieved by NetChain is reassessed to show that increasing chain lengths are deteriorating its attainable latency and throughput. This work promotes higher scalability and offers lower average latency over increasing chain lengths and higher average throughput: up to 9.46× higher throughput for a chain of 8 nodes, and 4 orders of magnitude lower latency. The routing mechanism is designed to achieve high parsing efficiency and low overhead over the underlying transfer protocol. NetCRAQ operates under strong consistency, while the replication method can be adapted to work with relaxed consistency in favour of performance.

Overall, this work contributes by:

- Identifying weaknesses and performance limitations of KV platforms that operate in

PDP.

- Stateful packet processing at link rate involves numerous programming constraints to ensure that the compiled binary can achieve the attainable rates. This introduces a series of design decisions that need to be made in order to strike a balance between the environment constraints, performance, and the functionality of the selected use case. A replicated storage fully implemented in PDP allows experimentation with various design choices: algorithm selection, elements maintained in PDP, protocol structure, failure handling, etc. This work discusses such design decisions in detail and provides reasoning behind design choices for the case of replicated storage in PDP.

- Using the insight acquired from examining the various design choices of previous works in the area (as mentioned above), this work contributes by proposing a new in-network KV platform with design elements that combine a different replication algorithm alongside various engineering improvements (like reduced packet size, implicit KV state representation, etc.). Through evaluation, the new platform appears to offer major performance and scalability improvements over the state-of-the-art.

- Providing the accessory routing and processing mechanisms to efficiently deliver strong consistency and flexibility.

3.2 Existing Limitations

Research in the workload characteristics of deployed KVS shows that they accommodate read-mostly workloads: the read-write ratio is 30:1 for Facebook's Memcache [7], 380:1 for Google F1 [181], and 500:1 for Facebook TAO [25]. Primary-backup variations, like Chain Replication, are specifically designed for read-mostly workloads, making them an appealing candidate for the underlying replication method of data centre KVSs. In the context PDP devices, it has been shown that the simple design of primary-backup protocols and their reduced computation complexity requirements, ensure integration without performance compromises. For this reason, most approaches focus on primary-backup methods (instead of consensus-based methods) that are designed for read-mostly workloads.

3.2.1 NetChain

Capturing the need of modern data centre applications for fast transactions between various distributed entities, NetChain created a KVS platform for the programmable data plane, able to accommodate the need for distributed, consistent data replication at the processing speed of programmable switches (going beyond what IncBricks had achieved

IncBricks) [100, 126]. Modern VNF orchestration platforms perform regular reads of KV pairs concerning configuration variables, health metrics, and scheduling variables. Offloading such values in network devices that are co-located in the same infrastructure as the orchestrator enables sub-RTT responses and therefore quick recoveries in case of failures or changes within the network. There is also a large range of data centre applications and network functionalities that rely on data caching and could be greatly accelerated with the use of an in-network replication platform, e.g., distributed Machine Learning, value exchanges for protocol convergence, etc. For the purposes of this work, NetChain is considered as an important platform that expands in-network compute to include frequently used services.

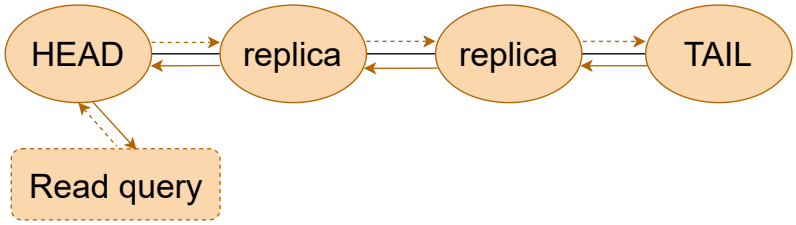
Under Chain Replication, the replication method used by NetChain, the participating nodes/programmable switches form a chain and each has a distinct role: head, tail, or replica. All of the participating nodes hold the same KV pairs. Write queries originate from the head and then propagate across all replica nodes until they reach the tail. The tail issues a response which acts as an acknowledgement for the write. A tail node is also responsible for responding to read queries. Only the tail is considered to be up-to-date with the latest commit for a value and acts as a reference point for the entire chain. In Figure 3.1a, we show the path of a read query (dashed arrows) and its response (solid arrows). Replicas can replace the head or the tail in case of a failure.

Defining the tail as the reference point allows per-key consistency for the entire chain to be achieved. When a write query reaches the tail, it has certainly been processed by all previous chain nodes. Therefore, all chain nodes are updated with its latest version. If the write query is lost before reaching the tail, then all subsequent reads will be replied with the previous version for this object. This ensures consistency in replies.

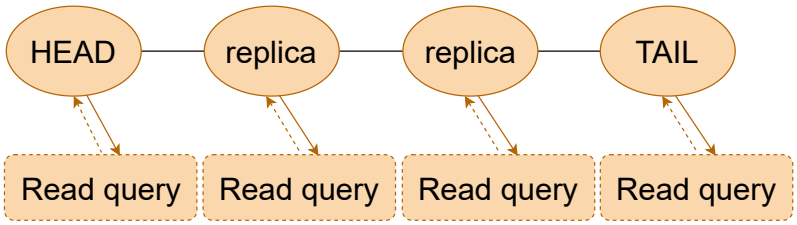
NetChain [100], with the use of P4 [63] managed to deploy an in-network KVS in high-performance ASICs, instead of NPUs in the case of IncBricks, and therefore achieved greater performance than IncBricks. Its design realised that queries can be processed in PDP with minimal interactions with the control plane, as part of a fully deployed replication method.

The query-response mechanism employed relies heavily on the incoming packets that are processed using the match-action pipeline [78]. A custom packet format was used (shown in Figure 3.5a), layered over UDP transport, which contained the following fields: `OP` - the type of operation (read, write), `KEY` - the ID of the object in question, `VALUE` - its value, `SEQ` - a monotonically increasing sequence number that mitigates out-of-order deliveries, `SC` - the number of chain nodes in the header, $\texttt{S}_\texttt{k}$ - IP of the k^{th} participating node. Storing the IPs of the nodes in the header aims at reducing the amount of stored data per switch and allowing dynamic mapping of data to chains.

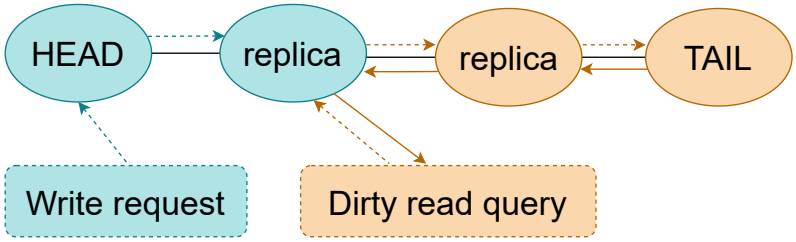
We notice that the suggested packet structure can add significant overhead bytes,



(a) Message path for a read query in Chain Replication.



(b) Message path for read queries to clean objects in CRAQ.



(c) Message path for read queries to dirty objects in CRAQ.

Figure 3.1: Comparison of message path for a read query in Chain Replication vs CRAQ.

especially for larger chains where all the participating node IPs have to be added in the header. For a 4-node chain, NetChain's header is 58 bytes, and grows by 32 bits with every node addition. The linear growth of the packet size with the chain length can cause increased parsing times and adds complexity when fields need to be added or removed [82], which according to the platform's design happens each time a query is processed. In a data centre topology, the overhead computations result in scalability loss and wasted resources. This design choice forces the administrator to choose between performance and redundancy. Another issue arises from the use of a monotonically increasing value in the `SEQ` packet field, which is 16 bits by default. This size allows just $65,536$ operations before the field overflows.

The reasoning behind the choice of Chain Replication as the main replication method for NetChain reflects the limitations and features of the deployment environment and provides important lessons. Firstly, Chain Replication has small redundancy requirements to achieve fault tolerance: to survive f node failures, it requires $f + 1$ nodes. This is significant reduction when considering that it translates to the amount of programmable switches in use. Secondly, Chain Replication presents low implementation complexity by requiring a simple commit-and-forward pipeline to execute a write query among the chain nodes. A quorum-based approach would require several RTTs to reach consensus on a

successful write or respond to a read, which would increase implementation complexity: the chain has to remain small for performance to remain unaffected from increased packet sizes and chain traversals. In which case, the offered redundancy is limited and the burden to monitor and maintain multiple small chains grows with the number of KV pairs.

While the choice of Chain Replication as an in-network replication method displayed superior performance over legacy KVSs, we observe some performance-limiting factors. Based on the principle that only the tail can reply to read queries, the amount of generated packets is substantial: for n participating nodes, $2n$ packets are required for read queries and $n+1$ for write queries. NetChain, by employing Chain Replication, adopts this design which, in the context of a data centre environment, has the following limitations:

1. generating messages for the tail results in high packet gain for the platform. It requires network resources for extensive parsing and forwarding;

2. the chain's reply rate is limited to the throughput that can be provided by the tail node, being the only one responsible to reply. This heavily harms scalability;

3. directing all queries to a certain node can also be root cause for hot-spots within the topology;

4. the response latency increases linearly with the chain length because of the increasing number of hops.

NetChain attempts to spread the load among nodes and minimise downtime in case of failures by adopting a variation of consistent hashing [106] in combination with virtual groups [40]. In other words, the KV pairs are partitioned among different hash rings with each switch holding different continuous segments of the ring. The parameters for these methods are defined when the nodes are initiated and remain static. There is no mechanism to re-adjust at a later stage. To make a manual change in the deployed KV pairs, each switch has to be updated with the appropriate match-action rules while an in-network KVS is live and accommodates millions of queries, resulting in potential failure to serve the queries. Even then, the idiosyncrasies of Chain Replication are not resolved and performance is not guaranteed to be improved.

3.3 CRAQ

Another primary-backup replication method, CRAQ [190], employs a different design but operates in a similar manner to Chain Replication: the nodes form a chain and each node can be a head, tail, or replica. The key differences with Chain Replication are: CRAQ's ability to handle load across all chain nodes – effectively enhancing scalability, and its ability to operate under relaxed consistency guarantees to benefit performance.

In CRAQ, each KV pair can be either clean, in which case the there are no pending commits for its value, or dirty, which means that the most recent commit is yet to be acknowledged by the tail. Therefore, multiple versions of a value can correspond to a key. CRAQ places this information inside each participating node. Upon receiving a read query, each node can either: respond to it, if the version is clean (cf. Figure 3.1b), or redirect the query to the tail in order to fetch the latest version (cf. Figure 3.1c). Writes operate similarly to Chain Replication: a node has to propagate a write down the chain until it reaches the tail and then be acknowledged as the latest clean version. Once this happens, the rest of the chain nodes are notified and can delete previous versions of this object.

The performance limitations of Chain Replication, as identified in Section 3.2, are revisited here to examine how CRAQ's design addresses them.

1. Packet gain is reduced, given that **only** dirty read requests need to be redirected to the tail. Considering that workloads are read-mostly, the ratio of clean reads over dirty reads should be high, resulting in a small amount of queries transferred to the tail.

2. The chain's throughput is not limited by the throughput of the tail. Replica nodes are able to reply to clean requests and contribute to the total throughput of the chain, promoting scalability.

3. While the notion of the tail node as the reference point remains, the amount of traffic directed to it is reduced, making the rising of hot-spots less probable.

4. Response latency is consistently lower than Chain Replication for the case of clean objects, regardless of chain size. Only a single hop is required to fetch the reply.

Furthermore, read performance can be further enhanced using CRAQ's feature to operate under relaxed consistency. CRAQ has two operation modes that support relaxed consistency. The first supports eventual consistency: dirty versions of an object can be used for replies meaning that monotonic read consistency is only locally maintained, and not throughout the chain. The second supports eventual consistency with maximum-bounded inconsistency: replies use dirty values that are above a certain time or version threshold.

It appears that CRAQ offers various improvements over Chain Replication with advantages in performance and scalability while maintaining strong consistency. In the following Section, we examine how CRAQ can be adapted to work in PDP, adhering to the posed constraints of such devices while preserving its advantages over Chain Replication.

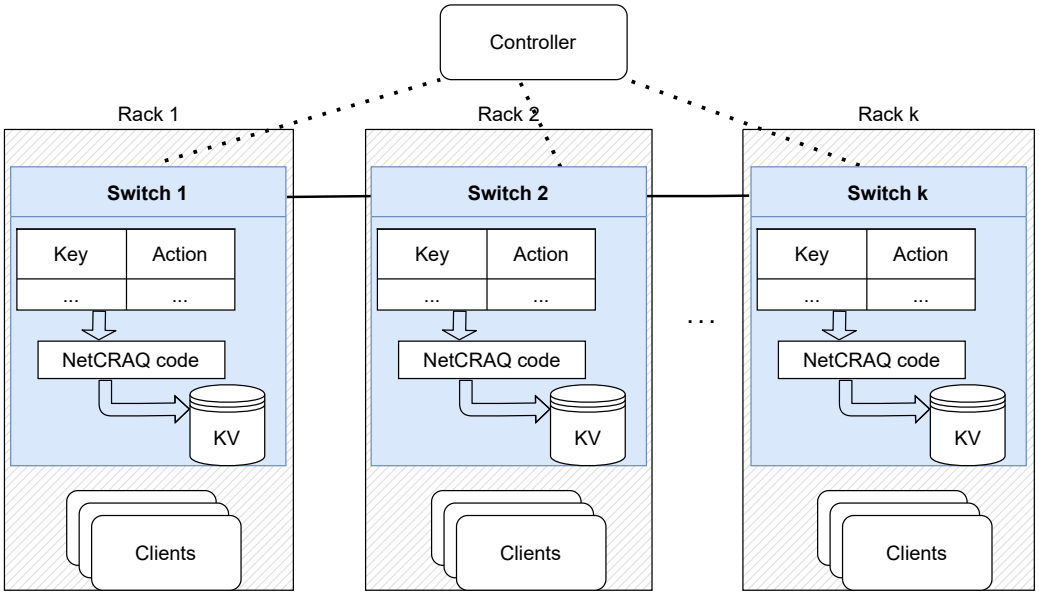


Figure 3.2: Overview of NetCRAQ.

3.4 NetCRAQ Design

NetCRAQ's design delivers a fault-tolerant, in-network KVS with focus on high throughput and scalability, minimum packet gain, and strong consistency. The design allocates functionality between the control and the data planes according to the advantages and disadvantages of each. Time-critical computations are placed in the data plane to utilise line-rate performance. The control plane is responsible for network-wide operations, like failure detection and recovery. An overview of the design is shown in Figure 3.2. NetCRAQ supports multi-level topologies and is able to direct queries to nodes using IP forwarding. This could be utilised to formulate different chains within a topology, using the available resources in the most efficient way. However, for demonstrating the platform's functionality in a simple way, its deployment is presented in top-of-rack switches in a linear topology.

We leverage the line-rate performance of PDP devices to offload all the query processing tasks in the data plane. This ensures that fast responses are generated even when retrieving values from other switches cannot be avoided. The time necessary to retrieve a local key-value pair is minimised by placing the necessary data structures within each switch's registers. The registers are located at the switch's SRAM to ensure line-rate accesses [22]. Moreover, we keep all coordination messages within the data plane to ensure minimum delay and consistency.

The control plane is used for less time-critical operations of the platform. Forwarding rules are generated and installed through the control plane upon initialisation or failure. The roles of the switches are also initialised through the control plane. They are installed across all switches to make sure role-based forwarding does not involve retrieving data from the controller, which would introduce delays in the packet processing pipeline. The

control plane is also entirely responsible for reacting to failures. In the case of a failure, it initially activates the failover mechanism, which redirects traffic from the failed node to a working switch. This is done to minimise traffic loss while the node is down. Once a recovery node is booted, the control plane installs the latest KV pairs in its registers and then replaces the failed node with the recovery one.

On the client side, NetCRAQ requests are layered over UDP, using a reserved port number. The requests can be easily integrated in any modern API using a simple script to transform them to NetCRAQ's packet format. The same format is followed by reply packets.

3.4.1 Data Plane

NetCRAQ's data plane design is responsible for two main operations: storing/retrieving the KV pairs in the relevant data structures and processing and forwarding queries and coordination messages. We present the different elements that enable these operations.

To adhere to the constraints of PDP devices and program under a solid set of rules that will make our implementation transferable to programmable switch chips, we choose the P4 programming language [63] and the Portable Switch Architecture (PSA) [78]. P4 is specifically designed to be compatible with programmable switches and ensures that the compiled binary can be executed with near line-rate performance. This choice also allows for a direct comparison against NetChain, which is developed with the same tools.

Implicit KV State in PDP

A key difference between Chain Replication and CRAQ is the way that new writes are processed. In CRAQ, for each object k, there are potentially multiple versions, n. In Chain Replication, appending multiple values for an object is not required and instead the only "clean" version exists in the tail. In the context of programmable switches, to satisfy CRAQ's requirement, n register cells need to be available to commit writes. For this reason, the switch is initialised with $k \times n$ sequential register cells reserved, forming an array. We call this the `objects_store` array. Figure 3.3 shows the format of the array. Each object consumes n number of cells in the array to allow dirty commits to be appended at the end of the last commit.

The state of each object (clean/dirty) has to be retrieved to determine the future control logic operations. We implicitly define the state as clean iff the latest committed value exists in the first cell of the object's space within the array. The location of the latest committed value for each object in the `read_index` array (cf Figure 3.4). Similarly, the location of the next available cell to commit a write is stored in the `write_index` array. The latter is also used to prevent out-of-bound writes. This implicit definition is based on

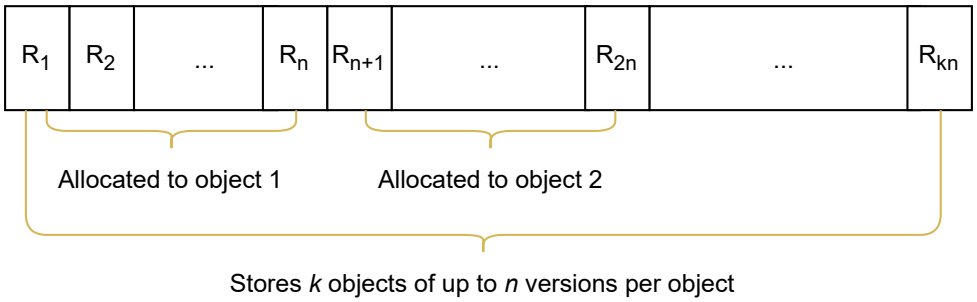


Figure 3.3: Multiple versions per object in a single array – `objects_store`.

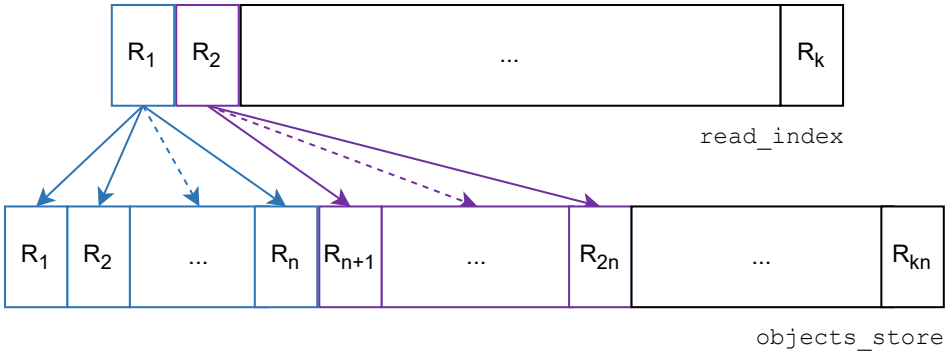
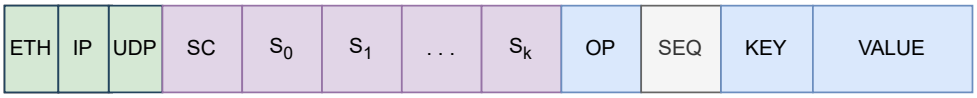


Figure 3.4: Auxiliary data structure used to determine clean/dirty KV pairs – `read_index`.

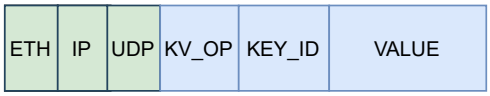
the principle that every previous value is deleted upon a successful write of an object, i.e., when this is acknowledged by the tail node. To give an example of how this mechanism differentiates between clean and dirty values, assume $n = 30$ objects with $k = 10$ versions each are stored in a switch consuming 300 register cells in total. Assume a query arrives for object with id=1. To determine if the latest version of this object is clean or dirty, its read index has to be examined. If the read index for this object points to `objects_store[10]` then there are no pending commits and the version is clean. If the read index for this object points anywhere between `objects_store[11]` and `objects_store[19]`, then the version is dirty. This mechanism is presented in the form of pseudo-code as part of the complete control logic in Algorithm 1.

Packet Format

In Section 3.2, NetChain's packet structure is described. We reiterate its variability according to the chain length and the large amount of overhead bytes that can be added. Since extensive packet parsing cannot be avoided when messages have to traverse the entire chain, even by employing CRAQ, truncating the packet size and reducing packet modifications on each switch should enable faster forwarding between the participating nodes. NetCRAQ's packet format (seen in Figure 3.5b) follows a simpler approach by having just three fields layered over UDP:



(a) NetChain's packet format.



(b) NetCRAQ's packet format.

Figure 3.5: Comparison of packet format between NetChain and NetCRAQ.

- `KV_OP`: defines the type of the operation: read request/reply, write request, acknowledgement. (2 bit)
- `KEY_ID`: contains the key id. (32 bit)
- `VALUE`: the field containing the value for the specific key. (128 bit)

NetChain's design placed information tied to the functionality of the platform, like the number of participating nodes and their IPs, within the packet. We follow a different approach, where such information is omitted from the packet and instead placed within the match-action rules of the switch, thus minimising the header size and reducing the parsing time for all KV operations and coordination messages. The control plane is responsible for updating the roles according to changes, instead of relying on the incoming packets for information that concerns the network infrastructure. This way, when changes occur within the network, recalculation of forwarding rules and switch role allocations can happen in a single entity that is responsible for such operations.

Ingress Control Logic

The control logic, executed by all participating switches, entails all the necessary operations for interacting with the KV pairs and managing the network traffic. All operations of the KVS are atomic to protect the values from simultaneous accesses. NetCRAQ's control logic relies heavily on the match-action concept, while values obtained from parsing the NetCRAQ header are matched against a pre-defined table that dictates the action that is executed when a match occurs. These match-action pairs are computed at the control plane.

Metadata fields, used for branching decisions, are also filled by the control plane in advance using the same mechanism. These metadata fields contain values that need to be regularly retrieved to manage incoming traffic. For example, the role of each switch or the IP of the switch appointed to be the tail of the chain. This constitutes a key design difference with NetChain, since the aforementioned information is already stored and maintained in the switches instead of being passed through incoming packets (in the

Algorithm 1: Control Logic

```
1  objects_store = register[k * n] ; /* main storage of n objects with k
     versions */
2  read_index = register[n] ;          /* defines read pointer array */
3  write_index = register[n] ;        /* defines write pointer array */
4  if kv_op == READ then
5  |  get_read_index(KEY_ID);
6  |  get_my_role();
7  |  if meta.read_index == 0 then
8  |  |  clean_read(KEY_ID);
9  |  |  generate_reply();
10 |  else if meta.my_role == TAIL then
11 |  |  dirty_read(KEY_ID);
12 |  |  generate_reply();
13 |  else
14 |  |  forward_to_tail();
15 else if kv_op == WRITE then
16 |  get_write_index(KEY_ID);
17 |  get_my_role();
18 |  if meta.write_index == 0 then
19 |  |  clean_write(KEY_ID);
20 |  |  forward_to_tail();
21 |  else
22 |  |  if meta.write_index >= k then
23 |  |  |  drop();
24 |  |  else
25 |  |  |  dirty_write(KEY_ID);
26 |  |  |  forward_to_tail();
27 |  if meta.my_role == TAIL then
28 |  |  clean_write(KEY_ID);
29 |  |  generate_acknowledgement();
30 |  |  multicast();
31 else if kv_op == ACKNOWLEDGEMENT then
32 |  clean_write(KEY_ID);
```

case of NetChain). Having this information stored instead of parsed enables faster overall parsing and forwarding [82].

The control of packets that contain the NetCRAQ header is primarily dictated by the `KV_OP` field. The allowed operations for this field are: `READ`, `READ_REPLY`, `WRITE`, and `ACKNOWLEDGE`. Deletes happen in the form of a `WRITE` operation, since the memory is statically managed and cannot be freed upon removal of KV pairs. Packets requesting a `READ` operation need to contain the value 100 in this field. Replies are matched with 101. Writes contain the value 200 and acknowledgements the value 300.

Algorithm 1 shows the complete control logic. If the identified operation is a `READ`, the next decision is based on the position of the value within the register. If a value exists in the first position of the object's register space, we know that the version is clean, otherwise it is dirty. The next stage includes checking the role of the node. Only a tail node can reply to a read with a dirty version, while the rest can only reply with clean versions of an object. Writes in the tail node may be committed but not yet acknowledged by all nodes, therefore replying with the latest value is not voiding consistency.

3.4.2 Control Plane

The control plane is responsible for installing all the match-action rules related to forwarding, KV operations, and failure recovery. The control plane allocates a different IP in each switch. This IP is stored within the metadata of each switch and determines whether a query will be replied on the arriving node or forwarded. We use the IP protocol for this and modify the header accordingly to forward to tail or generate acknowledgements. This provides flexibility for multi-level topologies, offering integration with load-balancing protocols, like Equal-Cost Multi-Path. Furthermore, the control plane determines and allocates the roles of the switches within the chain. Based on the number of participating nodes and the distance between them, different role allocation techniques can be used for a more flexible deployment. For example, the control plane can integrate the `meter` extern, offered by P4, to identify potential hot spots within the topology and re-adjust the chain lengths and the KV pairs within each register.

The multicast rules of the tail switch are maintained through the control plane, ensuring that nodes are added/removed as changes happen within the chain. Registers can also be accessed and managed from the control plane, avoiding packet generation in the chain during initialisation or addition of a node. All things considered, NetCRAQ's control plane design emphasises reconfiguration of rules according to changes within the chain and the network. Ensuring a smooth adaptation to network changes can be critical in dynamic data centre environments where failures are common. That said, beyond the failure recovery mechanism, the evaluation of the reconfiguration properties is beyond the scope of this work and scenarios like dynamic role changes or hot-spot detection are not

examined.

Handling Failures

Failure mitigation happens in two phases: 1. immediate redirection of traffic to a failover node to reduce the traffic loss; and 2. complete recovery with a replacement node and re-installation of forwarding rules and KV pairs.

When a node remains unresponsive for a certain amount of time, the client can automatically direct requests to a different chain node. This time can be adjusted based on what is considered as a prolonged lack of response according to the average response rate of the network. Once the failure is noticed by the control plane, the forwarding rules are updated by removing the node from the forwarding tables and the multicast group.

In the second phase, a new node re-enters the chain. To maintain consistency, we follow CRAQ's approach to identify which node will be used to copy KV pairs from. The control plane, depending on the position of the failed node, decides the node that will be used to copy the KV pairs to the new node. The reader can refer to the original CRAQ paper for the complete list of scenarios [190]. The recovery node remains offline while the control plane copies the KV pairs from an online node. During this phase, the control plane also disables any writes across the chain in order to preserve consistency. When the copy is complete, the node is added in the forwarding tables and the multicast group of the chain.

3.5 NetCRAQ Performance (vs NetChain)

NetChain was compared against legacy KVS and demonstrated the clear performance benefits from generating sub-RTT responses using fast data plane memory accesses [100]. NetChain's findings allow us to safely infer that, if NetCRAQ's performance is equal or better than NetChain, then it is also faster than legacy methods. Therefore, we directly compare NetCRAQ's performance against NetChain in a series of tests concerned with: throughput, latency, mixed workloads, and scalability.

3.5.1 Evaluation Setup

The testbed used for evaluation runs a bare-metal installation of Ubuntu 18.04 (kernel: 4.15.0-140-generic) on Intel Core i7-4790 CPU and 16GB of DDR3 RAM. P4 behaviour is emulated using the reference BMv2 switch [35] - compiled using performance flags. The topology is generated and managed using Mininet [121] and P4-utils [76]. The control plane is written in Python and communicates with the Mininet switches using Thrift [61] and the P4-utils API.

Typically, BMv2 is used for prototyping purposes. There is a big gap between BMv2 performance and the performance of programmable switch chips. However, by closely following the documentation and compiling it using performance flags, we ensured a stable behaviour that enabled reproducible results. BMv2 documentation testing reports a median throughput of 1047 Mbps and our testbed achieved a median of 1176 Mbps for the same tests, which ensures that the testbed in use has adequate resources and its performance is on par with the expected performance. We were able to demonstrate the design differences between the two platforms and showcase the impact these have on performance. The measured performance differences stem from the optimisations that reduce the number of computations and hops required to generate a reply. In the case of implementing the platform in PDP hardware, such optimisations should be prevalent regardless of the implementation details of the hardware.

Each experiment uses the same topology for both platforms. The same holds true for the number of objects and their sizes, the changes of which did not impact performance during evaluation. All topologies are linear and have the same number of switches and hosts. Since one host is able to generate enough queries to saturate the link with the switch, there is no need to use more than one host per switch. To compare raw performance, no load balancing methods were used despite that both platforms can support them.

3.5.2 Throughput

We evaluate the throughput of both platforms based on the maximum attainable rate at which they can provide responses to queries. The measurements are in Queries Per Second (QPS). We direct millions of packets to each switch while increasing the packet rate. The maximum attainable response rate is considered the rate at which the response rate starts to decrease and the response latency rises.

Figure 3.6 shows that NetCRAQ's throughput is not impacted by distance when the queried object is clean. The reduction in required hops and computations create a big performance difference in favour of NetCRAQ: 4.08× higher throughput for queries directed to the head of the chain. In case of dirty objects, throughput is still higher than NetChain with the difference being attributed to the smaller packet size used by NetCRAQ, 72 overhead bytes for NetChain (`SEQ` field set to 128 bits to allow continuous traffic) vs 22 bytes for NetCRAQ. This difference results in smaller parsing times, which when the number of hops increases is less apparent. When dirty queries are generated directly at the tail, the amount of processing required to generate a reply is the only factor impacting performance since hops are minimum. In this case, NetCRAQ shows 22% higher throughput than NetChain, proving higher overall computation efficiency. When queries are directed to the head, the amount of nodes between the source of the queries and the tail is introducing limitations in throughput. Here, NetCRAQ is 10.5% faster. Overall, in terms of

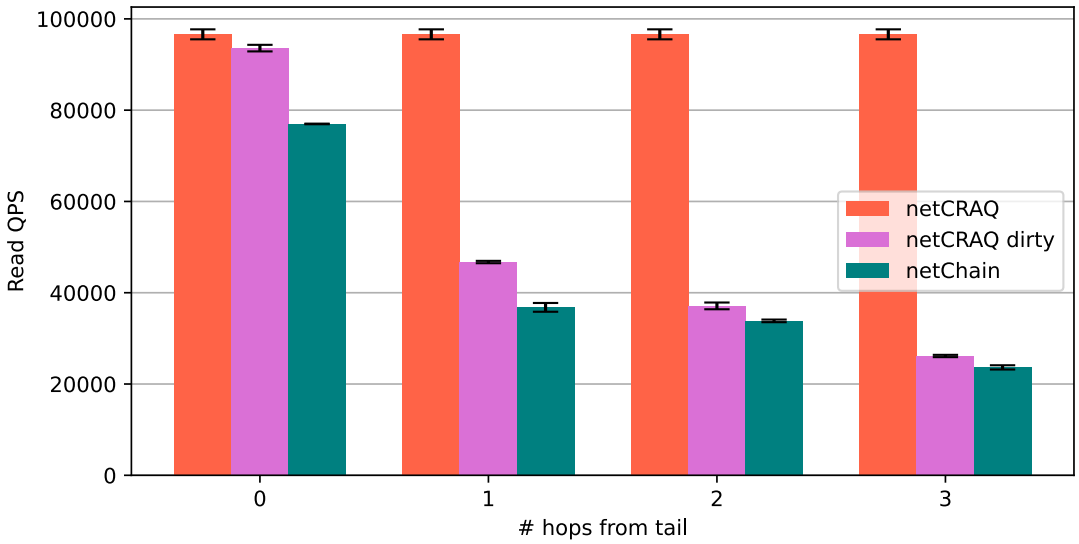


Figure 3.6: Max read QPS vs distance from tail.

throughput, NetCRAQ shows superior performance irrespective of the state of the object and the distance from tail.

We examine how the two platforms operate in an environment with limited resources in order to determine the impact of traffic gain and the overall computation efficiency. A platform with fewer redundant operations will be able to utilise the available resources to generate replies instead of performing chain traversals and packet parsing. This is indicative of the wasted processing cycles and link strain that would occur in a data centre environment. To evaluate these properties, we create congestion in an increasing number of switches across the chain and assess the impact this has in throughput. To make the comparison fair, all different combinations of clients are averaged, irrespective of their distance from the tail. Figure 3.7 shows the outcome of this experiment. Once more, NetCRAQ achieves better utilisation of the testbed resources and sustains higher throughput under intense workload scenarios: 2.25× higher throughput for 25% of clients generating queries, 2.8× higher throughput for 50% of clients generating queries, and 4.73× higher throughput for 100% of clients generating queries.

3.5.3 Latency

As the rate of queries rises, sustaining the same response latency becomes increasingly harder with continuous atomic operations causing contention of the available resources. The number of hops and the ingress processing pipeline are the two main factors to impact the latency of a response. To investigate the latency of the two platforms under different loads, we use the same 4-node chain and generate an increasing number of read queries.

Figure 3.8 displays the obtained measurements for this scenario. In the displayed

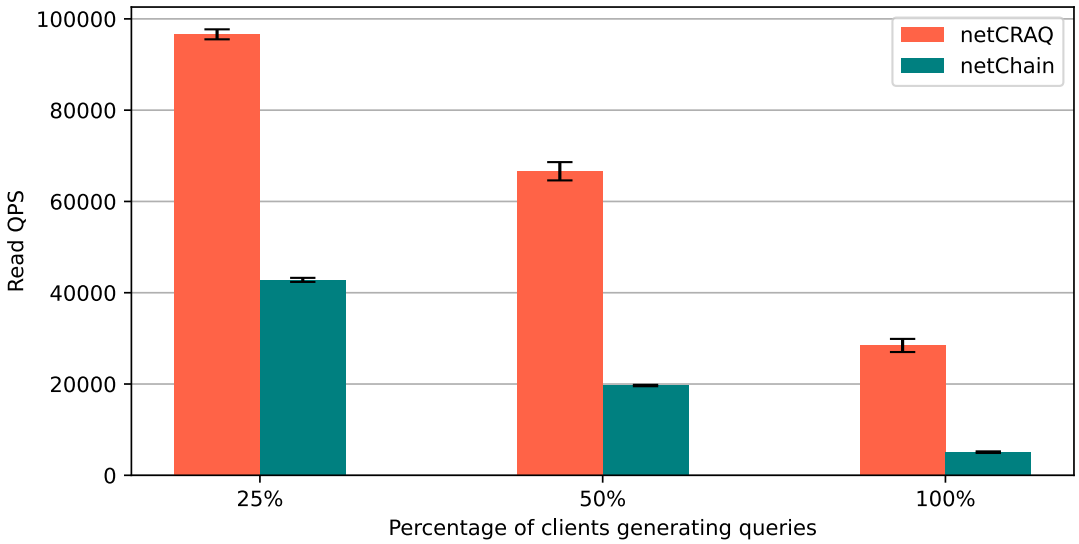


Figure 3.7: Sustained read throughput vs percentage of congestion.

measurements, we include metrics from all participating nodes, regardless of distance from tail. Providing consistent latency with different chain lengths adds flexibility to the platform and the ability to adapt to the requirements of the KVS and the network. NetCRAQ shows a steady latency response that rises marginally with the number of read queries. NetChain presents a big variance in response latency which is related to the varying distance from the tail. The difference becomes more significant as the number of QPS rises: two orders of magnitude faster responses for 5k and 10k QPS, and three orders of magnitude for 20k QPS. It is worth noting that the latency profile observed in our evaluation setup is different than the one in the original publication [100]. The reason is two-fold: 1. we are using emulation instead of an ASIC, which is also the main reason for any latency discrepancies, and 2. we use a larger chain (4 nodes) than the one used in the original paper

Figure 3.9 shows the relevant graph for write queries. NetCRAQ is able to accommodate writes faster for query rates up to 10k QPS. For 15k QPS, the multicast operation of NetCRAQ is congesting the tail link causing a rise in latency. To avoid unintended congestion, rate-limiting can be applied through P4 externs. We explore the impact of slower writes in mixed read/write workloads below. It is worth noting that these platforms are targeted towards read-mostly workloads and a scenario with continuous writes would be an extreme use case.

3.5.4 Mixed Workloads

We evaluate the platforms under realistic workloads containing a mix of reads and writes. The behaviour of both platforms under such workloads is shown in Figure 3.10. Starting

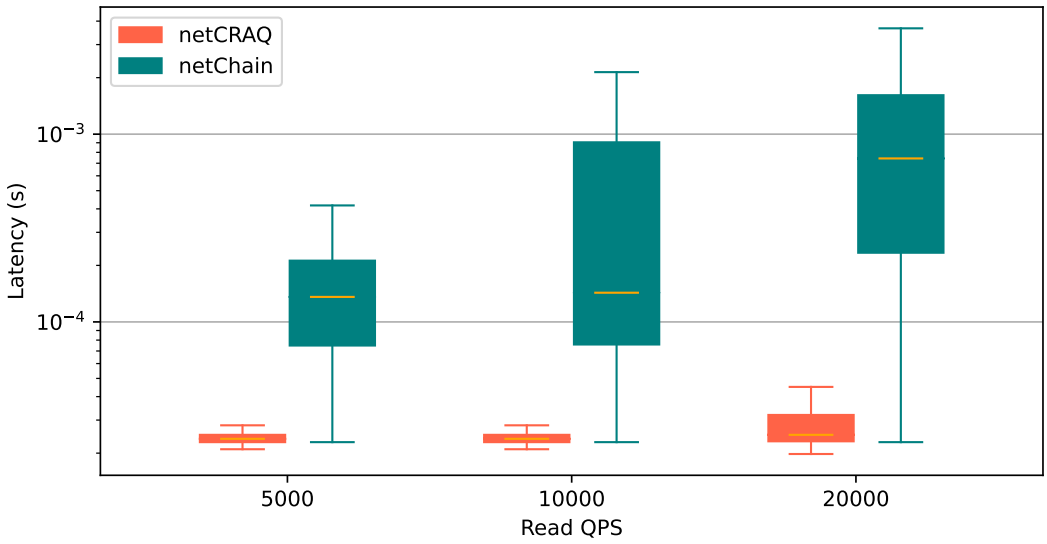


Figure 3.8: Response latency vs QPS.

from a read-only workload, we gradually increase the percentage of writes with a step of 25%. The performance of the platforms is judged by their attainable response rate. NetCRAQ achieves more than double the read throughput for all write percentages. With 75% of the queries being writes, both platforms show a decrease of around 85% of their read-only workload performance. Nonetheless, the read efficiency of NetCRAQ enables higher throughput, despite the write latency measurements observed in Figure 3.9. Adequate register cells need to be budgeted to maintain all dirty versions before they can be acknowledged by the tail. This is depicted by the increasing amount of dirty commits as write percentage rises, observed in the right y axis of Figure 3.10.

Although we evaluate both platforms with workloads with a higher percentage of write queries, these are not considered typical workloads for these platforms. Both the algorithms and their data-plane implementations are designed for workloads with a high percentage of read queries, as shown in the representative examples of Section 3.2. Applications that would generate a high percentage of write queries would have to be able to withstand a large amount of dirty commits. This means that they should either be able to operate under relaxed consistency or face delays in convergence times.

3.5.5 Scalability

NetCRAQ is also able to operate over longer chains with smaller throughput and latency losses over NetChain. We validate this in Figure 3.11. Here, the comparison is between read queries directed to the head of the chain. We vary the chain length from 4 to 8 nodes. There are no intermittent writes and therefore all KV pairs are clean. The results showed that in a chain of 8 nodes, the throughput of NetChain was reduced in half. On that

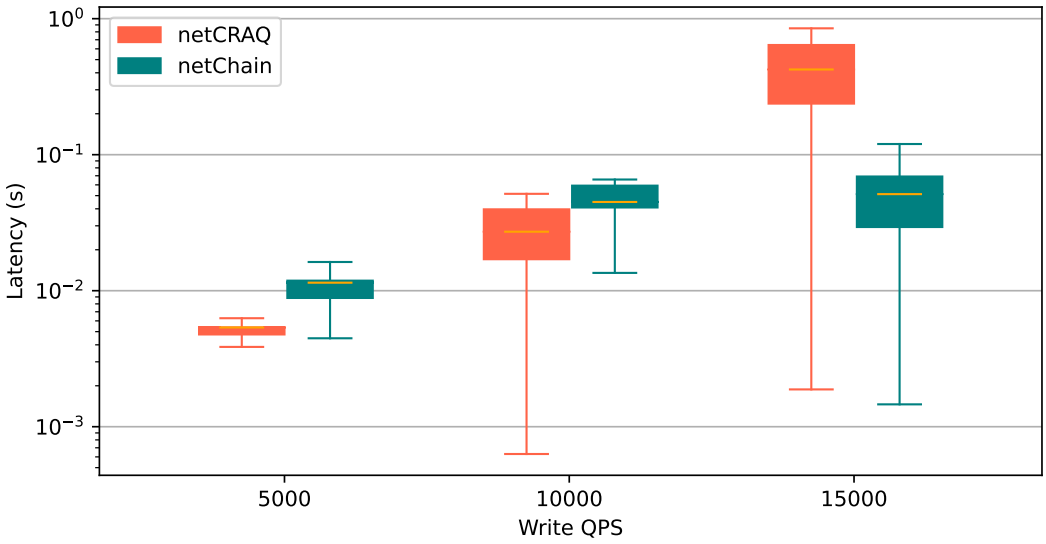


Figure 3.9: Write latency vs increasing QPS.

account, we did not proceed to further increasing the chain length as the performance gap was already representative of the throughput losses that occur. The performance difference is indicative of the potential gap between the two platforms when accommodating read-mostly workloads in varying chain lengths. The throughput difference can be up to 9.46× in favour of NetCRAQ, in the case of 8 chain nodes. This difference stems mainly from the ability of nodes to respond directly to read queries, thus reducing unnecessary forwarding and header parsing and processing. NetCRAQ requires only a single RTT to respond to a read query for a clean KV pair. Because of this, the performance remains the same regardless of chain size or the role of the node that generates the response (head, tail, or replica). For the case of NetChain, the amount of computations necessary in order to respond grows with the number of participating nodes.

3.6 Discussion

This section discusses challenges and limitations of the designed replication platform. It also expands on potential use cases that would benefit from using the platform, based on their workload characteristics.

3.6.1 State Preservation

Current programmable switches hold a limited amount of register memory which is accessible at line rates. On that account, registers are considered a scarce resource. Moreover, the number of registers that will be used can only be reserved during the initialisation of

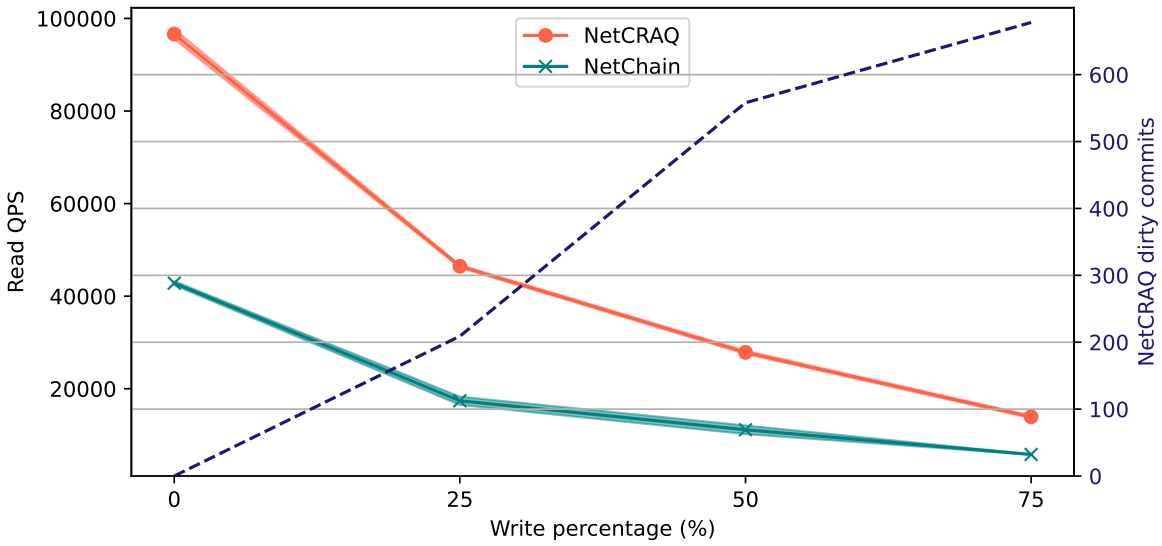


Figure 3.10: Performance under mixed read/write workloads.

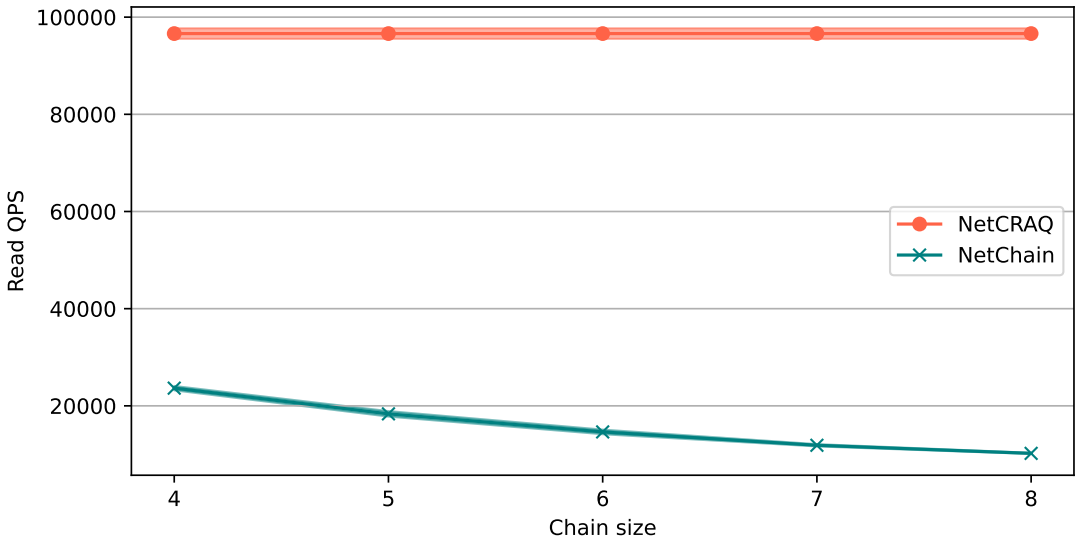


Figure 3.11: Read throughput vs chain length.

the switch. Therefore, their use should be done with careful consideration.

While NetCRAQ outperforms the state-of-the-art in many aspects, it also comes with a higher memory footprint. Namely, multiple object versions need to be stored per key. The exact amount is dependent on the application requirements and the link rates which will determine the required buffering between commits of a value. The chain-related information is also placed within the switch, minimising packet size but reserving space for storing the chain. This is a platform requirement that needs to be examined according to the deployment environment and the workload characteristics. By examining the Tofino iterations, we notice that they tend to grow in terms of available register memory [93]. We expect that this trend will also propagate to other programmable network devices and this requirement will become less constraining with time.

3.6.2 Technical Challenges

As explained on 3.6.1, a significant amount of information has to be maintained in switches for the platform to function correctly. Namely, the implicit definition of clean or dirty KV pairs, the IPs and roles of the participating switches, and a consistent KVS. The control and the data planes need to interoperate dynamically over network and platform changes. This can be a challenging task for the developer who needs to ensure that the hosted KVS as well as the accessory information remains coherent in a variety of scenarios. Even though P4 enables the development of custom layers in a fairly straight-forward manner, the final control logic relies mostly on conditionals and can expand quickly. The P4 code base becomes rather monolithic and complex with little options for debugging prior to deploying. The control plane can be developed using a variety of tools but evaluation of its responsiveness to PDP events requires real workloads, making development slow. All in all, while the technology has matured enough to support such implementations, in its current state the development becomes challenging, time consuming, and error prone.

3.6.3 Example Use Case

Apart from standalone deployments, KVS operate as part of other orchestration and network configuration frameworks, e.g., Kubernetes. Central to Kubernetes' architecture is etcd [51], a fault-tolerant, consistent KVS that provides coordination services and is used as the backup store for all of Kubernetes' control-plane components. Most importantly, crucial parameters for the operation of the cluster are stored in etcd, such as Container Network Interface (CNI) information like lease times and health status of participating nodes. However, it has been shown that etcd presents scalability bottlenecks[98]. Etdc relies on a consensus-based approach to ensure consistency among the participating nodes. This approach requires a growing amount of time to confirm that changes have been

committed in the majority of the participating nodes, which in turn creates increased response times. In this work, we examine the workload imposed to etcd by Kubernetes to examine its suitability for deployment in PDP. Based on this analysis, a suggestion is made to extend the Kubernetes architecture and offload part of the KV traffic to PDP devices.

Kubernetes Requests

To find out the type of requests directed to etcd, we operate a Kubernetes cluster with four nodes while monitoring various metrics of the generated requests. Stateless services are deployed in the form of containers which in Kubernetes are further enclosed within pods. We then proceed to scale these pods equally among the participating nodes. The pods are originally 2 and then scale up to 25. After the deployment has finished successfully we proceed to delete all of them.

The results are the average metrics as obtained over multiple runs of the same experiment. The majority of the requests were read queries (known as ranges in etcd) of a single KV pair – approx. 15.3k requests. The write requests were just 2.9k, which is 16% of the total number of queries. Most of the read requests, 55%, were directed to just 25 KV pairs which concerned health requests, leases, and scheduler values. There were 3.1k consensus proposals, all of which were successfully conducted.

To establish the best performance that can be achieved by etcd, an etcd benchmark is set up with representative characteristics of a typical Kubernetes deployment (as found on Kubernetes' documentation): 3 nodes, 128bit values, 1 client with parallel connections. The default benchmark tool provided by etcd was used to establish these parameters. The results showed an average write duration of 0.21s and an average read query duration of 0.7ms. These numbers are orders of magnitude higher that what can be offered by PDP platforms like NetChain.

The workload measurements reveal a *read-mostly* workload, *skewed* towards a small subset of KV pairs. The amount of consensus proposals appears to be significant considering that each consensus involves multiple RTTs to be conducted. Moreover, the response time of etcd is orders of magnitude lower than the existing in-network implementations. A PDP implementation of a KVS is not impacted by skewed workloads, given that all SRAM memory registers are accessed at line rate. The combined performance characteristics can ensure faster response times of Kubernetes to events such as changes within the network like health status updates and CNI transactions. Given that a big portion of the workload is generated during deployment, deployment times can also be reduced after allocation of the relevant KV pairs in PDP. These results compile a good use case for in-network KV offloading and integration of programmable devices in Kubernetes' design.

To this end, a new design for the integration of NetCRAQ within Kubernetes is pro-

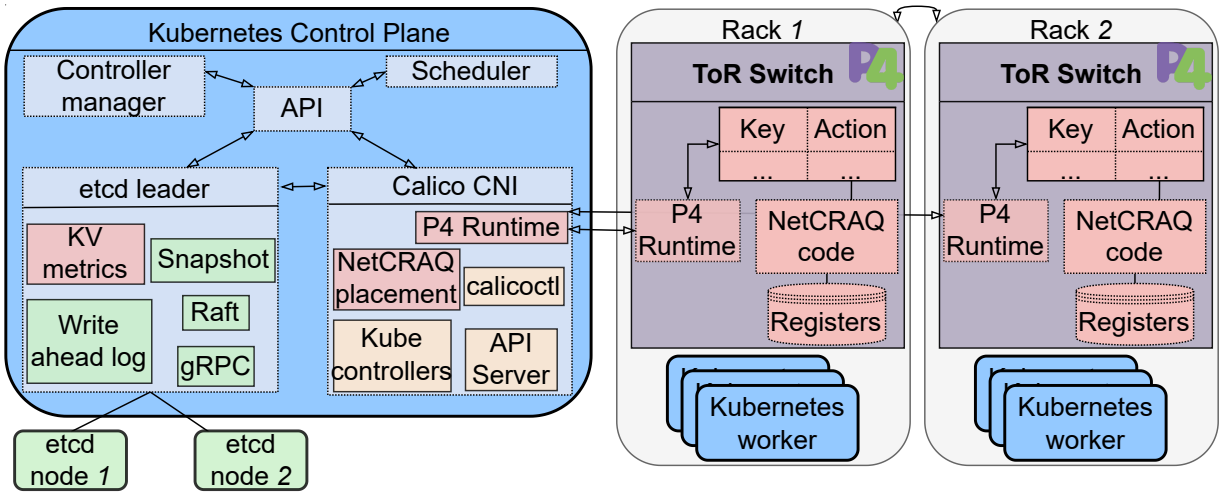


Figure 3.12: Overview of proposed design.

posed. The design can be differentiated in two main parts: the aforementioned PDP components of NetCRAQ; and the Kubernetes framework changes to support offloading KV pairs to PDP based on real-time metrics. An overview of the design is shown in Figure 3.12. Pink blocks represent the extended NetCRAQ design including blocks that are integrated in Kubernetes to extend its functionality. The rest of the depicted blocks show Kubernetes and etcd architecture in blue and green, respectively.

Kubernetes Control Plane Extension

Kubernetes' control plane comprises 5 components: the API server which is the front end for the control plane; the controller manager that monitors jobs, endpoints, and distributes tokens; etcd; the CNI that establishes network routing among the participating nodes; and the scheduler which allocates pods to workers by comparing the requested resources with the available worker resources [9]. In our proposed design, two Kubernetes components have been extended to support the integration of PDP: etcd and CNI.

A monitoring component has been added to etcd in order to identify most commonly accessed KV pairs. It uses the integrated Prometheus endpoint to read metrics [8]. The most frequent KV pairs are selected as candidates for deployment in PDP. The number of pairs is decided based on the available memory of the network device and the total size of the candidate KV pairs.

Monitoring is also in place for the values already in the PDP, which have counters for access frequency. These statistics are obtained through P4Runtime which is added as an extended part of the CNI [77]. Through the NetCRAQ placement scheduler, also located within the CNI, these metrics are compared and a decision on which values will be transferred to PDP is made. The most frequently read values are placed in the data plane. The pairs are transferred through packets that are generated in the CNI.

The P4Runtime CNI component is also used for less time-critical control plane operations. Forwarding rules are generated and installed through the control plane upon initialisation or failure. Equally, the roles of the switches are initialised through the control plane. They are installed across all switches to make sure role-based forwarding does not involve retrieving data from the controller, which would introduce delays in the packet processing pipeline. The CNI is entirely responsible for reacting to failures. In the case of a failure, it initially activates the failover mechanism, which redirects traffic from the failed node to a working switch. This is done to minimise traffic loss while the node is down. Once a recovery node is booted, the CNI installs the latest KV pairs in its registers and then replaces the failed node with the recovery one.

3.7 Summary

In this chapter, the capability of programmable network devices to perform per-packet stateful processing at line rate is leveraged to perform in-network replication. Literature review indicates that previous work contributed by implementing a KVS replication mechanism in PDP that delivers orders of magnitude throughput and latency improvements over traditional KVS platforms. By examining the design elements of previous in-network replication platforms I have identified key limitations that harm the performance and scalability: the selected replication algorithm can become a factor of performance degradation for longer chains leading to link saturation, traffic hot spots, and bottlenecked performance; the packet format used for managing the deployed chain introduces higher parsing times as the number of participating nodes increases; sequence control limits the maximum number of writes of a KV pair to the size of a single header field.

To address these limitations, a new design has been proposed. The key assumptions of previous work are adopted to ensure a fair comparison: 1. the platform accommodates read-mostly workloads as they are the most frequent type of workload within a data centre; 2. replies to queries are generated solely in data plane by manipulating the processing pipeline of PISA programmable hardware to minimise RTT. A new replication mechanism is adjusted to address previous design limitations and enable scalable generation of query replies among all participating nodes for clean KV pairs. A smaller packet format is adopted, transferring the duty of maintaining chain-related information at the control plane and practically reducing parsing and forwarding time. Furthermore, a series of ingress control logic changes allows implicit definition of clean and dirty KV pairs directly at the data plane, maintaining consistency and line rate processing performance without the need to consult the control plane or introduce consistency errors. The proposed platform design tries to address previous design limitations without compromising fault-tolerance

The evaluation of the aforementioned design choices shows improvements in scalability, throughput, and latency for the proposed platform at the cost of higher memory consumption – used to store committed values before they are acknowledged by the reference node. The workload imposed at a KVS from mainstream orchestration platforms, like Kubernetes, is examined as a potential use case. This can be part of a discussion for further integration of in-network replication within orchestration frameworks. This integration can offer faster deployment times and faster reactions to network events like node failures, link changes, etc. Additionally the co-location of data plane management elements with management elements of virtualised computing can unify network configuration in a singular control plane instance, allowing hybrid orchestration and management of services in both data plane and host servers.

This work can be further extended to dynamically integrate end-host programmable hardware, like smartNICs and FPGAs, in the replication chain. Through existing mechanisms (like P4 counters), sources of traffic can be identified and linked to popular KV pairs. In a multi-rack deployment, this information can be used to extend the chain of participating nodes to include end-host programmable hardware located at the source of the traffic, and therefore provide responses to client queries at reduced latency while reducing bandwidth consumption that would be spent on query/response transfers. To achieve this, there are many intermediate steps that need to be in place before committing to expand the chain of participating nodes. Certain metrics need to be adjusted to inform the decision of expanding the chain, e.g., what is the amount of generated traffic from a certain host that will trigger the expansion of the chain?, what is the maximum chain size that can be supported without compromising performance? etc. Such questions can expand the current scope of this work and constitute avenues for further research.

Chapter 4

In-Network Storage and Processing at the Edge

4.1 Overview

With edge computing being the main driver for deploying IoT applications, orchestration frameworks have to manage increasingly complex infrastructures with diverse constraints. More specifically, works in the area of cooperative sensing, augmented reality and Industry 4.0 have stringent requirements in terms of latency, availability, resilience and scalability [201, 108, 166]. Several edge nodes that inter-operate formulate clusters that enable failure recovery and accumulate processing capacity. The management of the deployed NFs needs to be agile and offer unique optimisations for each deployment environment.

To achieve edge-cloud convergence and scalable management of the deployed clusters, edge computing is envisioned to work in conjunction with NFV to deliver a dynamic, scalable computation environment at the edge [183]. However, mainstream VNF orchestration frameworks are primarily designed for use in cloud infrastructures and therefore do not integrate mechanisms that adapt to the constraints of edge nodes. For example, in edge environments with limited energy resources, the orchestrator remains oblivious about the energy levels of each processing node which results to suboptimal placement decisions that harm the network's lifetime. Furthermore, responses for queries that arrive in cloud infrastructure require processing of aggregated data that are gathered at the edge. This results in excessive bandwidth and energy consumption for transferring and processing these data at the cloud.

This chapter treats energy-constrained IoT clusters as a representative edge deployment environment and identifies existing limitations. It also contributes to the discussion of whether virtualised NFs can be hosted at the edge, what is the energy impact, and what would be the minimum viable computing specifications. The main limitations that are identified and discussed further in this chapter are:

- Absence of an in-network storage mechanism for energy usage profiling that is based on real-time measurements.
- Lack of energy awareness of the orchestrator leading to suboptimal placement decisions.
- High-cost edge-cloud communication for data transferring and analysis.

To demonstrate these limitations, one of the most popular orchestration platforms, Kubernetes, is selected as a representative framework for deployment of containerised VNFs at the edge. Using a real world cluster of SBCs running on batteries and managed by Kubernetes, energy consumption measurements are collected. Based on the collected measurements, I build the energy profile of Kubernetes in order to further examine the feasibility of such deployments in terms of energy footprint. This work further contributes with a new placement method that uses the obtained energy measurements to deliver real-time energy-aware VNF placement in Kubernetes. Mainstream orchestration platforms like Kubernetes are developed to provide robust software components and ensure trustworthy behaviour and interoperability. These aspects are maintained in our case by extending this platform and only altering the scheduling component. Additionally, a unified edge-cloud infrastructure is promoted through reliance on the same orchestration platform.

The issue of expensive edge-cloud communications is further addressed for the aforementioned environment. By leveraging the data stored at the edge and the processing capabilities of edge nodes that make them able to respond to analytics queries of simple sensor data, a mechanism that is able to direct queries to the relevant edge node is built. This is achieved by building statistical signatures of data held by edge nodes and storing them in a Query Controller (QC). The QC determines the relevance between queries and stored data and directs queries to nodes with relevant data where they are processed and responses are generated. This allows the deployment of a hierarchical query allocation mechanism that can direct queries to relevant nodes and therefore save the bandwidth and energy cost of data transferring to the cloud.

4.2 Motivation

With edge computing being a core component of modern services, the need to perform fine-grained management of virtualised services at the edge, i.e., aware of local constraints and demands, becomes apparent. Work reviewed in Section 2.6.3 reveals lack of research on the computational and energy footprint of virtualised in-network computations at the edge is not examined. Moreover, the energy footprint of running an orchestration platform at the edge is not measured and the virtualisation overheads are not established. This work

is motivated from the above and deploys a testbed managed by a mainstream orchestration platform running on typical IoT devices to monitor the energy footprint of different roles of cluster nodes and different types of computations typically found within the cluster, like data gathering and data storage.

Section 2.6.3 analyses a plethora of works in the areas of resource allocation, placement, and scheduling. In the literature, the placement problem has been adjusted to various parameters like latency, communication cost, etc. However, practical implementations of scheduling algorithms are scarce, especially for edge environments. The proposed algorithms mostly use simulated environments and testbeds with abundance of computational resources to test the proposed solutions. This work is motivated by the lack of real-world implementations of schedulers that deal with the placement problem for energy-constrained environments and addresses this gap by deploying the proposed scheduler in the aforementioned testbed. A representative cluster of off-the-shelf SBC devices is managed entirely using the proposed scheduler, proving its feasibility and technical soundness. It extends previous work by introducing real-time energy measurements in the placement process in order to minimise the avoid service disruption due to battery depletion.

As explained in Section 2.7.1, past research dealt with the allocation of queries to a set of nodes; either through a time-optimised scheme for selecting the appropriate nodes based on the *odds algorithm*, or a reinforcement learning model for query allocation [113, 115]. A statistical learning processes for query load balancing has previously been implemented [114]. The query allocation mechanism shown in this work extends previous schemes by providing a statistics-based efficient mechanism responsible to deliver the minimum sufficient information of the data for query allocation in edge computing environments. Such mechanism is *exposed* to the Cloud infrastructure being part of the schemes proposed in [113, 115, 114].

4.3 Energy Monitoring on Clusters of Single Board Computers

This section describes a virtualised, scalable, and fault-tolerant energy monitoring mechanism for SBC clusters that can derive and store energy measurements for the deployed services based on sensor readings. This mechanism is designed to be integrated into Kubernetes and allow network planning and real-time optimisation of energy consumption through placement and migration. Furthermore, it showcases the potential of SBC devices to formulate clusters and host containerised functions and services at the edge. This is among the first demonstrated deployments of SBC clusters running Kubernetes in the literature and providing some insights on the energy usage requirements of Kubernetes running over SBCs.

4.3.1 Design

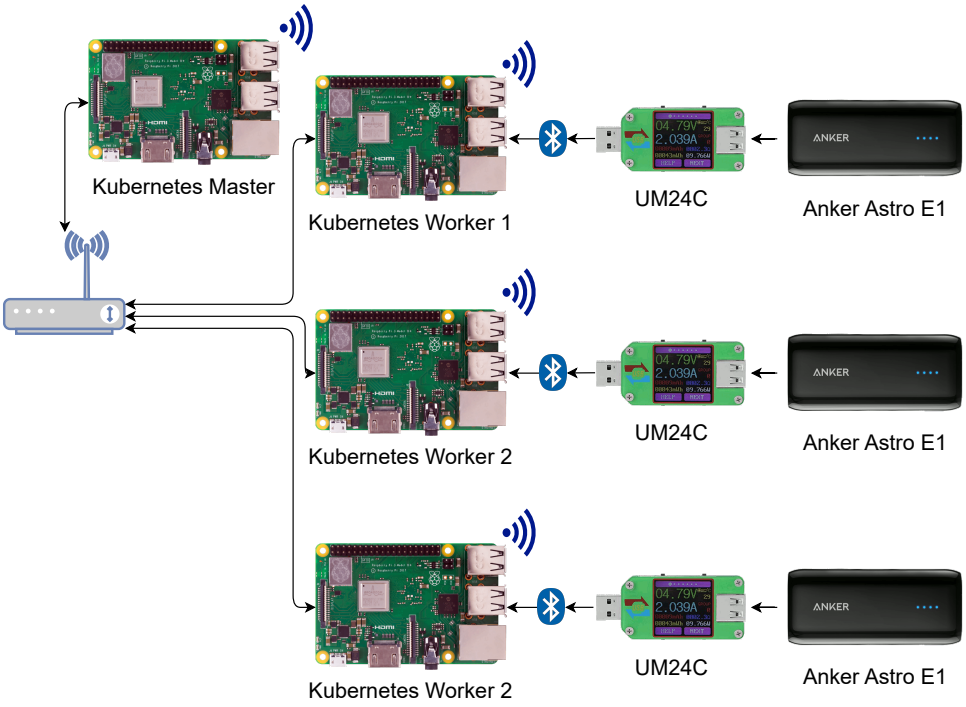


Figure 4.1: Overview of cluster's architecture.

To explore the potential of an edge SBC cluster, a testbed comprised of four Raspberry Pi 3 Model B devices is deployed [65]. These devices are a typical representation of an IoT device, commonly used as the core devices for many SBC clusters built in the past [103]. By being compatible with a wide range of sensor devices, the Raspberry Pi can function both as a sensor device and a processing node. The OS of choice is Raspbian, a variation of Debian Linux that is officially supported by the manufacturer of Raspberry Pis.

The energy source for each device is a battery rated at the capacity of 5200 mAh [127]. The sensor device for the electric measurements of the power consumption for each node is the UM24C module [188], which connects with the Raspberry Pi via Bluetooth. Connectivity via non standard methods can prove challenging when using containers or other virtualisation methods. However, IoT devices are likely to rely on such connectivity. To address this, such use cases are explored using this testbed. A diagram depicting the connections among the devices can be seen in Figure 4.1.

The energy monitoring is handled by a virtualised application that will communicate with the UM24C module to obtain energy measurements. The application components need to be designed to recover from failures, given that they can be common in IoT environments. The application must also process the measurements and calculate an estimation of the remaining capacity of the battery. This metric, called State of Charge (SoC), will be updated on a database table separately for each of the nodes. The database

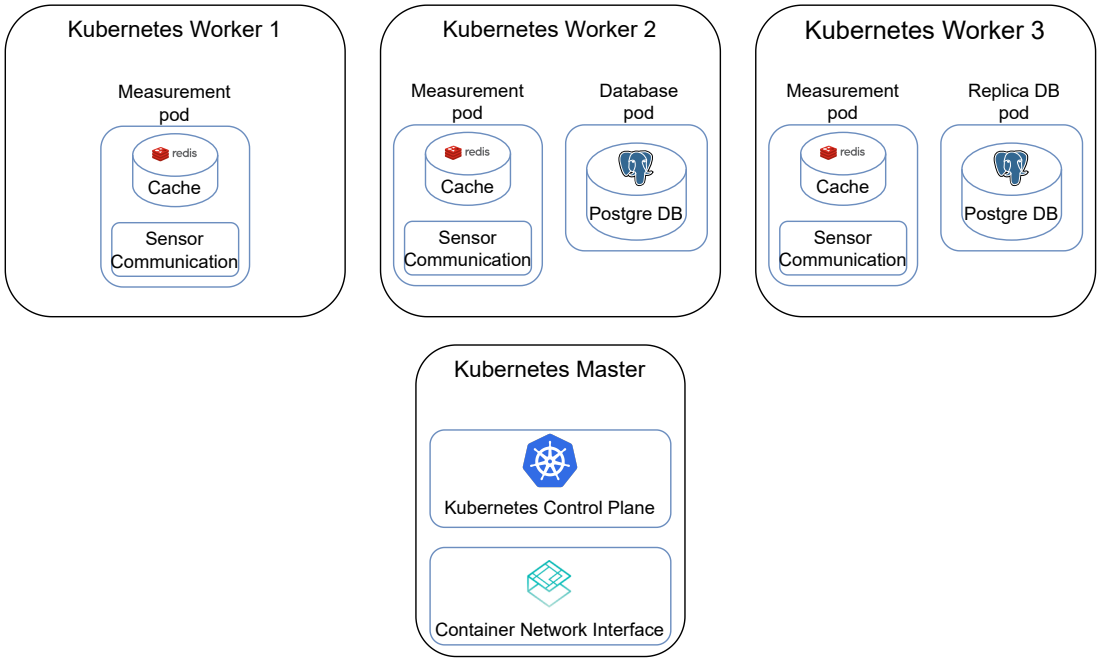


Figure 4.2: Overview of application architecture

must be maintained in more than one nodes for data recovery in case of battery depletion or failure of the pods.

Communication with the UM24C device was achieved through serial-over-Bluetooth connectivity. There is an initial message which the application must send to UM24C in fixed time intervals to trigger the broadcast of the measurements. The required measurements are acquired by investigating the 130-byte reply message. The SoC estimation of a battery, depending on the estimation model in use, can be dependent on factors like the state of health of the battery, the specific battery model, etc. [90]. These factors are typically hardware related and estimations that integrate them have to be considered coupled with the hardware under use. In such cases, they have to be re-estimated upon hardware change. Such complications are less present in the coulomb counting method [147]. By using the coulomb counting method, the total electric charge that a battery absorbs while being charged is monitored, and the same stands for the total charge that is released when the battery is used to depletion. In principle, the SoC is estimated according to the percentage of the electric charge that exited the battery over the electric charge that entered the battery:
With $Q_{releasable}$ denoting the released capacity when the battery is completely discharged, and Q_{rated} depicting the rated capacity of the battery, SoC percentage is given by:

$$SoC = \frac{Q_{releasable}}{Q_{rated}} \tag{4.1}$$

Because of its simple yet accurate approach, the coulomb counting method is followed

for providing SoC updates in the application. To get more accurate results, we decided to evaluate the Q_{rated} capacity by examining the actual electric charge that the battery is able to deliver over a number of charge-discharge cycles. We followed a process similar with the coulombic efficiency estimation that authors proposed in [147]. Coulombic efficiency, denoted with η, was found to be $\approx 57.69\%$ of the rated capacity, or around 3000 mAh in total capacity. To increase the accuracy of the SoC estimation, Q_{max} was used to denote the maximum releasable capacity and eq. 4.1 was adjusted to:

$$SoC = \frac{Q_{max}}{\eta \cdot Q_{rated}} = \frac{Q_{max}}{0.576923 \cdot 5200mAh} \tag{4.2}$$

Estimated SoC values are stored in a database after being written in a Redis cache. The pods responsible for these operations are always considered prone to failure. Therefore, the application must be designed according to a resilient architecture. Replicas of the database have to be stored in multiple nodes, so that it can be recovered in case of unexpected failures. Equally, a caching mechanism is used to ensure that services are able to restore their last state after failure. If the pod hosting the cache and the application container fails, the application uses the database to restore its last state. The final architecture of the application can be seen in Figure 4.2. Each black rectangle represents a node. Inside each node there are pods, represented by blue rectangles. The collection of pods necessary for the execution of CNI – in this case Weave-net, are omitted. The rectangles inside the pods depict the containers running within.

The Measurement pod is the main component of the application. Inside it is a Redis container, which functions both as a cache for the data and a safety measure in case of failure. Data can be requested from the Redis cache of each node in case of high traffic on the database. This can help reduce load from the database at times of congestion, when for example an administrator performs demanding tasks, and save the database pod from failure.

Within the Measurement pod is also the Sensor Communication container which performs data gathering, SoC estimation, and saves the measurements in the Redis container and in the Postgres database (located in the Database pod). The Measurement pod is deployed in every worker node and measures their energy requirements. The database is replicated in the Replica DB pod. The Replica DB pod can easily scale to allow higher redundancy.

The redundancy characteristics of our architecture are not bound to energy monitoring applications and could be equally used for other applications that make use of sensing devices and need redundant storage of the measurements. These fault-tolerant characteristics can be combined with the scalable properties of the design to make it appealing for applications with dynamic node numbers. Other NFV services can also be deployed

in parallel, alter the processing pipeline of obtained measurements, and introduce new routines, e.g., anomaly detection.

4.3.2 Measurements

Our goal is to obtain measurements from real-world, energy-constrained deployments of IoT hardware that host virtualised computing workloads. To do so, we mounted energy measurement devices that obtain voltage and current measurements to containerised services.

Setup

More specifically, four UM24Cs were mounted to pods through privileged execution of `volumeMounts` within Kubernetes. The same was done for Docker containers. This was adequate to obtain a fully functional cluster, showcasing that combinations of SBCs and sensors can be used to host virtualised applications at the edge. However, a way to support device mounting in pods without granting generic rights, like rights for manipulation of the network stack, is an addition that would enhance security control.

At the time of testing, Kubernetes was still in early development stages with minimum support for the processing architecture of Raspberry Pi. Despite this, the application presented a small amount of fails. On more than 40 hours of testing, pods failed 3 times and were able to recover without loosing any data. The resilient design seems mandatory as pod failures are likely to happen. The observed failures were not linked to development mistakes. Instead, platform related errors were the main source of pod failures.

Stress test

To obtain an insight on the energy requirements of a standalone Raspberry Pi, we performed a CPU stress test, spawning four workers of the `stress` Debian package, and measured the voltage and current drained from the battery until depletion. The voltage graph of Figure 4.3 reveals a sawtooth function whose period gets shorter near the depletion time of the battery. This sawtooth behaviour is due to the voltage regulator circuit that is used in the Anker Astro E1, which ensures that the voltage never drops bellow a certain value.

The graph of current, in Figure 4.4, displays a peak current of $\approx 0.750A$ at the beginning of the measurements subsequently remaining between $0.6A$ - $0.68A$. This comes as a result of the thermal throttling that the Raspberry Pi experiences. As CPU temperatures raise above 83.5 degrees Celsius, the frequency of the CPU drops from 1.2GHz to 600MHz which results in the aforementioned average of power consumption. Behaviour is likely to change with active cooling.

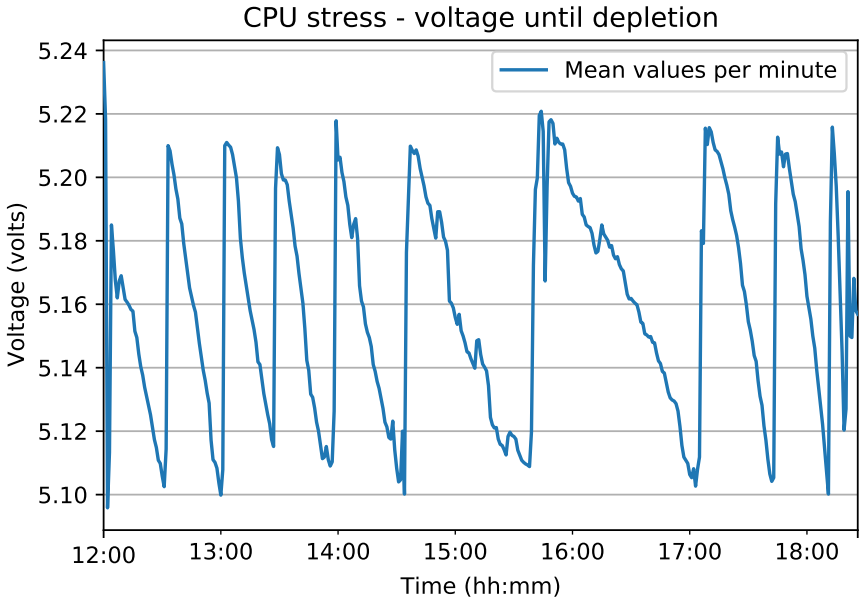


Figure 4.3: Voltage measurements under CPU stress.

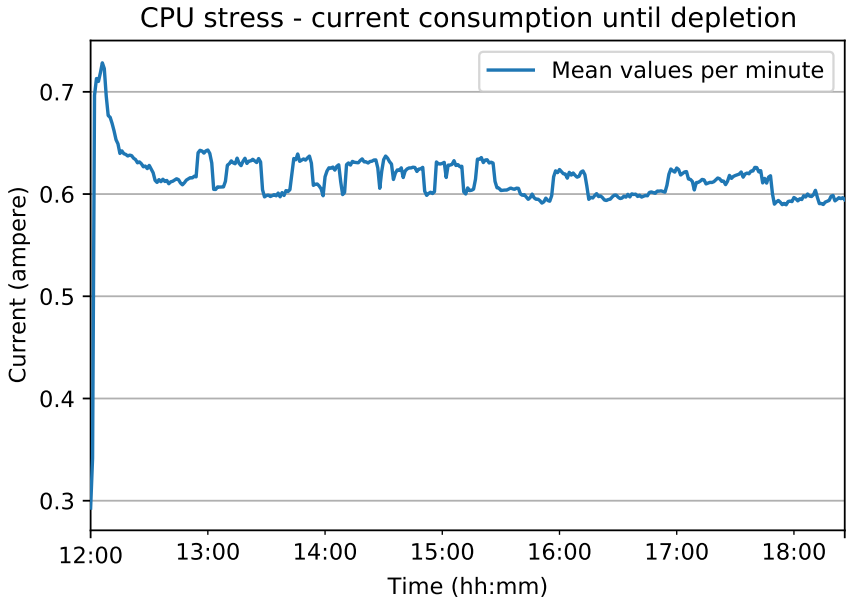


Figure 4.4: Current measurements under CPU stress.

State of Charge estimation

The SoC estimation method is evaluated by checking for a synchronisation between the estimated remaining capacity and battery depletion. This was found to be accurate in most cases, an example of which is shown in Figures 4.5 and 4.6. Differentiation occurred when batteries with different states of health and age were tested.

To demonstrate the SoC estimation method, we test it against unsteady current values, expecting a change of behaviour in accordance to the fluctuation of current. The input values of current are shown in Figure 4.5 and the corresponding output in Figure 4.6. Opposite to the expected output, the SoC estimation is linear without presenting any major changes that correspond to the varying values of current. The hypothesis behind this observation is that the changes in the current are neither big nor long enough to cause such change when compared with the capacity of the battery.

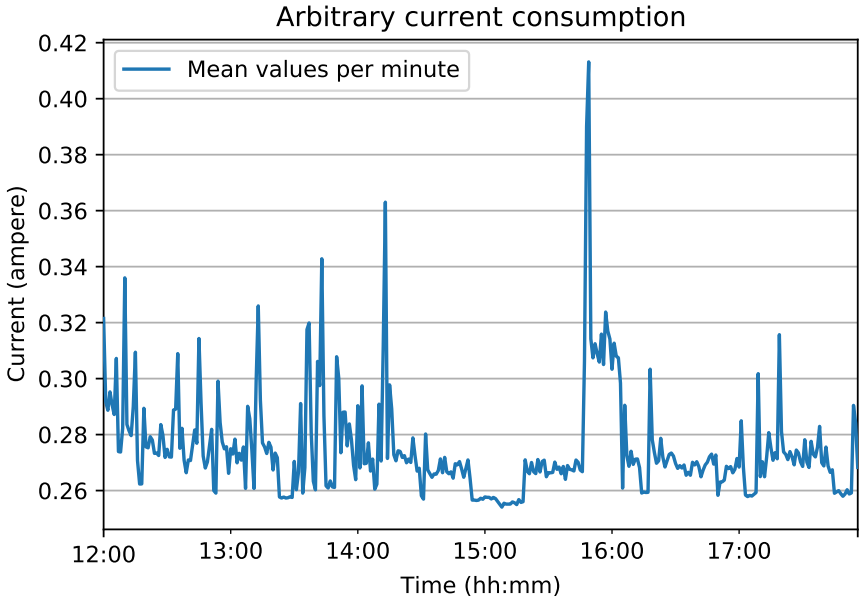


Figure 4.5: Input of current values to the SoC estimation algorithm.

Consumption per Task

Another area of investigation is the average energy consumption of nodes depending on the tasks that are executed on them. The measurements for the nodes were obtained while the monitoring application was at full scale for 1.5 hours on the testbed. The task allocation for each node was: the master node executing Weave-net (CNI) pods and managing the running pods, a worker performing energy measurements and logging, another worker performing measurement and logging while running the database where measures are stored. The results are shown in Figure 4.7.

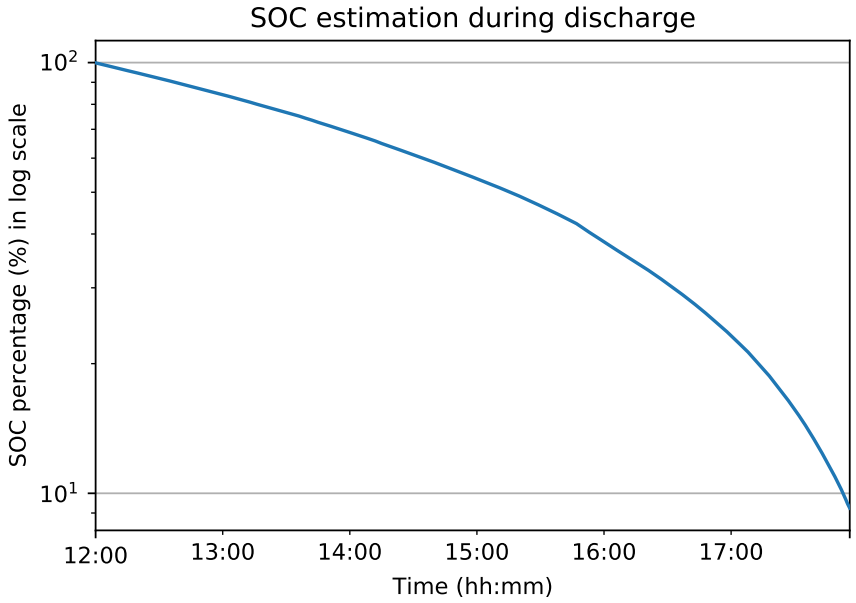


Figure 4.6: Response of the coulomb counting method to the input of Figure 4.5

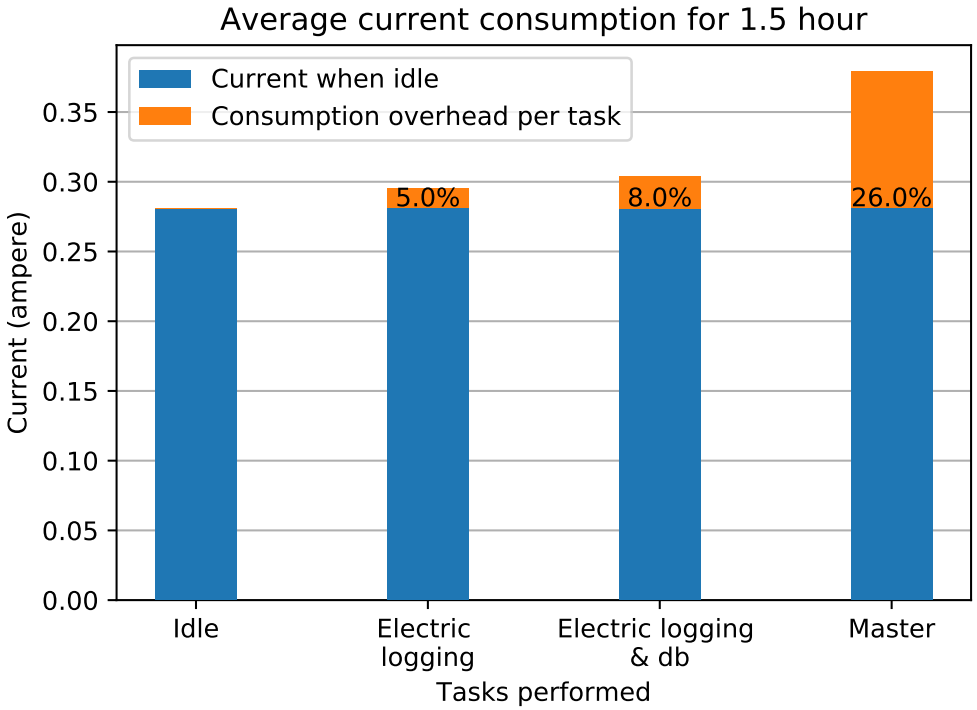


Figure 4.7: Average current measurement per activity.

Our comparison basis for energy needs is an idle Raspberry Pi which was found to consume an average of $0.281A$. The node responsible for the logging of energy measurements displayed an increase of $\approx 5\%$, reaching $0.295A$. The combination of database and logging services was found to consume an average of $0.304A$, or $\approx 8\%$ growth from the baseline measurements. The master node was found to be the most energy-demanding node, with an average consumption of $0.379A$ which translates to $\approx 26\%$ increase in power consumption compared to our basis. The explanation behind this result is that the master node is responsible for all management and orchestration processes, i.e., monitoring pod health status, monitoring the resources of worker nodes, scheduling, maintaining forwarding rules, handling of API requests. These processes have a constant demand for computation and networking resources resulting in a high power draw. This concludes that the master node is the most affected node from Kubernetes' execution in terms of energy consumption.

Findings

Overall, the successful deployment of the testbed proved that SBC clusters are ready to be used in conjunction with NFV platforms to host custom-made applications interconnected with sensor hardware. The lack of an official method to mount host devices to VNFs is not a limiting factor, at least for the development stage of applications. Additionally, the overall stability of the hardware, the OS, and the NFV platform is sufficient to claim that SBC clusters have the potential to be considered as trustworthy edge devices when a resilient application design is employed. Even when hosting NFV platforms they retain their small energy footprint. This justifies their suitability for applications that run on batteries and have mobility or distant location characteristics. The NFV support makes the management of such networks even easier, especially if resilient and scalable monitoring applications ensure the normal operation of the network. All these contribute towards the use of SBC clusters as the infrastructure to develop edge-oriented applications.

4.4 Energy-Aware Placement

This Section shows how the integration of the mechanism presented above (Section 4.3) in the VNF placement process is achieved in order to perform placement decisions based on energy measurements. The goal is to maximise the attainable lifetime of the edge cluster infrastructure without compromising the interoperability and fault-tolerant features of the orchestrator.

4.4.1 Problem Definition & Notation

To capture the aforementioned environment characteristics and formally define the problems we investigate, we consider the reference software architecture depicted in Figure 4.8. It builds upon the previous hardware and software architectures shown on Figures 4.1 & 4.2 by using the same cluster and orchestration framework. The previous architectures are expanded to enable energy-aware placement. It also defines the types of events that can be scheduled within the cluster.

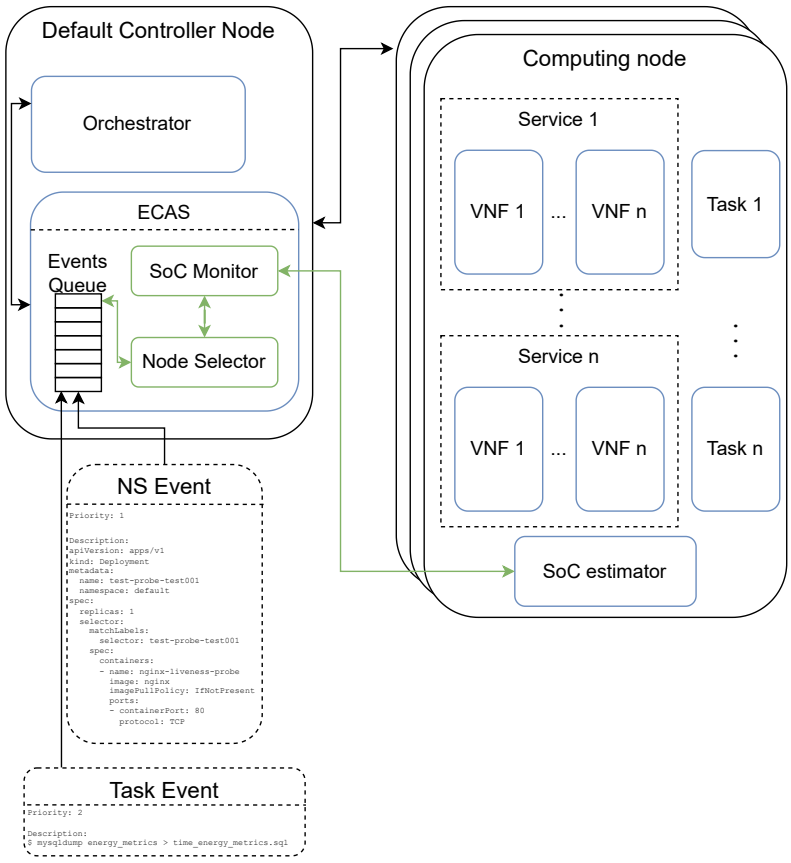


Figure 4.8: Software architecture showing distribution of tasks and services.

The controller is responsible for distributing incoming event requests and schedule them according to their deadline and requirements for resources. The cluster is treated as a pool of available resources which has to be managed in a way that does not violate deadlines or energy constraints. Two types of events are considered: **tasks** and network **services**. The former is a set of instructions that require a fixed amount of time for their execution. Examples of tasks are: log rotation (compressing log files older than a particular time and deleting deprecated ones), processing rows of a database table, backing up a service database before its deletion, etc. The resources required for the execution of a task are released upon completion. A service event is composed of a set of VNFs that can reserve resources for arbitrary amounts of time that is not known in advance.

The incoming events are examined by the scheduler in an online manner, i.e., as they arrive and by examining the current resource capacity of the cluster. The events are scheduled to a set of deployable units within computing nodes (P). P represents all the virtual nodes (p) – in this case Kubernetes pods, created over physical nodes (N). Each virtual node $p \in P$ is identified with an ID. The parameter p_i^n indicates that virtual node p_i is placed in physical node n, where $n \in N$.

Any given network service (S) is formed by a sequence of VNFs (F), where each function f must be processed on a set of physical nodes. These functions must be scheduled one after the other in a specified sequence. Each created virtual node p can only process one function at a time. Similar to the network functions, any given task (T) will be processed on the selected physical node where a virtual node p will be created to execute the requested task. Each event has a requested rate (r_e) that must be met by the selected node. In addition, the processing capacity of the node (c_n) has to cover the requested rate by the event (i.e., $c_n \geq r_e$).

All network functions and tasks have a running time parameter (t_r) that denotes the amount of time that must pass before an event can be considered completed. When $t_r > 0$, the event runs during the specified time. In the case that this parameter is 0, or not specified, the event will be executed during the whole life cycle of the system. Both types of events also have a starting time (t_s) and a completion time (t_c). Thus, we can calculate the execution time (t_e) of an event through the following equation:

$$t_e = t_c - t_s \tag{4.3}$$

In addition, the event's time of arrival (t_a) is registered to calculate the entire time that an event is in the system (t_t). Thus, the first parameter (t_a) denotes the time when the request for scheduling was received by the controller node. The second parameter (t_t) is defined as the total time, starting from when an event arises until its completion, and it can be calculated as follows:

$$t_t = t_c - t_a \tag{4.4}$$

As our system will receive event requests following a Poisson process, we introduce a priority queue to the controller node. The events can have different priorities according to user demands. The events are ordered in the queue according to various criteria including priority. The scheduler takes the highest ranking element in the list and chooses the best physical node to deploy a virtual node that will run the event (task or network function). The priority queue introduces certain delay in the node assignment process since the controller node schedules events one by one. In the case of the arrival event rate becoming higher than the scheduling rate, we define the waiting time (t_w) of an event as the time

from its arrival until its execution is started. This parameter can be obtained from the following equation:

$$t_w = \begin{cases} t_s - t_{a_{f=1}} & \text{if the event is a service} \\ t_s - t_{a_T} & \text{if the event is a task} \end{cases} \tag{4.5}$$

From Equation (4.5), we obtain the waiting time for each event. In the case of a service, we need the arrival time of the first network function, as this function represents the beginning of the service. Both the waiting and total time parameters are considered evaluation metrics that capture the performance of the scheduler.

Finally, a deadline (d) is defined for processing a given event. In the case of a network service, the processing of its last function must be completed before this deadline. Otherwise, the scheduler incurs a SLA violation. A list of notations related to the system model is provided in Table 4.1.

Notation	**Description**
P	Set of deployable units of computing nodes
N	Set of physical nodes where events can be scheduled
S	Network service request arriving to controller node
F	Sequence of VNFs compounding a network service request
T	Task request arriving to controller node
p	Each virtual node created on the physical nodes to run the events
n	Each physical node where virtual nodes are created
p_{f_i}	Indicates the virtual node where function f_i is running
p_T	Indicates the virtual node where task T is running
f_i	Each network function forming part of a network service
r_e	Demanded rate of each task and network function
c_n	Processing capacity of each physical node ($n \in N$)
t_r	Running time of an event before considering it completed
t_s	Starting time of an event when being processed in the selected node
t_c	Completion time of an event in the selected node
t_e	Execution time of an event in the assigned node
t_a	Arrival time of an event request in the controller node
t_t	Total time of an event in the system
t_w	Waiting time of an event in the priority queue
d	Deadline for processing a given event

Table 4.1: Descriptions of used notation.

4.5 The ECAS Scheduler

A new scheduler is proposed to replace Kubernetes' default scheduler. Energy Capacity-Aware Scheduler (ECAS) is able to process event requests and determine the best node in a cluster of SBCs to allocate them based on the remaining battery of each node, the predicted power requirements of the event, and the processing resources of each node. The proposed scheduler, shown in Figure 4.8 is formed by three main elements: the SoC Monitor, the Node Selector, and the Events Queue. The SoC Monitor and Node Selector components will be described in detail in the following sections. The Events Queue component was described in Section 4.4.1. External to ECAS is the SoC estimator component that runs constantly in the computing nodes as an agent and provides measurements to the SoC Monitor part of ECAS. It is a simplified version of the mechanism described in Section 4.3 that omits database replication and instead feeds measurements directly in the SoC Monitor component. The remaining Controller components are depicted in the Orchestrator block to simplify presentation and allow focusing on the novel parts of the proposed Controller.

4.5.1 SoC Monitor

This module is responsible for gathering and storing CPU and memory usage measurements from virtual nodes that have been created and assigned an event. It also monitors the status and power usage of the physical nodes and tracks the number of rejected events and violations due to insufficient resources.

It starts by obtaining the status of the nodes that form the resources pool for the cluster, as shown in Procedure "Update Status of Nodes". This procedure updates the

Procedure Update Status of Nodes

1 **forall** $n \in N$ **do**
2 Update node usage by comparing the accumulative CPU and memory use of all the virtual nodes placed in physical nodes
3 **if** $(n_{usage_{CPU}} \geq usage_{CPU_{max}}$ ***or*** $n_{usage_{RAM}} \geq usage_{RAM_{max}})$ ***and*** n_{status} *is available* **then**
4 $n_{status} \leftarrow$ unavailable
5 **if** $(n_{usage_{CPU}} < usage_{CPU_{max}}$ ***or*** $n_{usage_{RAM}} < usage_{RAM_{max}})$ ***and*** n_{status} *is unavailable* **then**
6 $n_{status} \leftarrow$ available

resource utilisation of each node within the SBC cluster. It determines a node's resource utilisation by calculating the whole usage of its virtual nodes in terms of CPU and memory. Considering all the cluster nodes as candidates to place a created virtual node by default,

this procedure checks if a current node's utilisation has not reached its defined maximum capacity (line 3). If the maximum capacity has been reached, the node's status is marked as *unavailable*, and it is excluded from the candidate selection process in the scheduling algorithm (line 4). Line 5 checks for the opposite condition. It verifies that the current node's usage is below its maximum value. The node's status is set to *available* in line 6 if it was previously marked as *unavailable*. The node updating procedure is used by the monitor block and its behaviour is described in Algorithm 2.

Algorithm 2: Update Event Metrics

```
1  Event_rejected ← 0 (Amount of rejected events)
2  Event_violations ← 0 (Amount of deadline violation in events)
3  while True do
4      forall p ∈ P do
5          if p_status is Running then
6              Get CPU and Memory usage from metrics server
7              Save CPU and Memory values in p_usage
8          else
9              if p_status is Succeeded and p_tc > p_d then
10                 Event_violations = Event_violations + 1
11             if p_status is Failed then
12                 Event_rejected = Event_rejected + 1
13             Delete the virtual node running the event to release its resources
14             Remove p from P
15             Remove S or T from l_S or l_T accordingly
16     Procedure: Update Status of Nodes
```

The monitor process begins by initialising two parameters (lines 1-2) that run throughout the component's execution (line 3). It checks several parameters (e.g., p_{status}, p_{t_c}, p_d) for all the created virtual nodes ($p \in P$) to verify if certain conditions have been satisfied. If a virtual node is running, the monitor block gathers its CPU and memory usage from a metrics server (e.g., Prometheus [8]) and records these values (lines 5-7). Otherwise, the event is determined to be in one of two possible states: *succeeded* or *failed*. In the case that an event has completed its execution, the virtual node where it was running is marked as succeeded. If the event has been completed past its deadline (line 9), the algorithm updates the amount of deadline violations in line 10. The other state is related to the failed virtual nodes (line 11). When this condition is satisfied, the amount of rejected events is updated (line 12). Afterwards, the algorithm releases the used resources and updates the respective parameters (lines 13-15). Finally, the algorithm calls the updating nodes procedure in line 16. The rejected events and deadline violations parameters are later used as evaluation metrics of the proposed scheduler.

Finally, the monitoring block receives the SoC information sent by the SoC estimator block in a process parallel to Algorithm 2. The received SoC information is saved in $n_{usage_{SoC}}$. The Node Selector component is able to obtain node utilisation metrics such as CPU usage, memory usage, and SoC by reading the stored values in n_{usage}.

4.5.2 Node Selector

This module is responsible for allocating events from the queue to nodes. To determine the best node for an event, it takes into account the node's remaining energy resources, its processing resources, and the event's requirements. It uses a regression model to estimate if the event can be hosted in the node or it will fail due to insufficient energy resources. The selection of the model and its evaluation is presented below. Typically the controller node is not included in the node selection process, due to the importance of the workloads it is already executing. This work re-evaluates this decision and assesses the behaviour of the system with and without deployment of events in the controller.

Impact of Controller Participation

To study the impact of this decision, we ran several experiments that include/exclude the controller from the scheduling problem. The cluster's ability to schedule more events successfully by using controller resources will be evaluated. Another aspect that will be evaluated is the saturation of controller processes when computational workloads coincide with them.

Experimental Setup

The testbed that is used is similar to the one used to obtain the energy monitoring measurements in Section 4.5.1. It is presented in more detail in Section 4.5.3. Generated events follow a Poisson distribution with requirements defined in Table 4.2. The results are presented with a 95% ci. Figure 4.9 depicts the average number of successfully scheduled and rejected events as well as deadline violations for different rates of event generation. An event is considered successfully scheduled when it did not exceed its deadline. On par with other works in the area, we assume that QoS is a soft constraint and therefore surpassing a deadline does not lead to service interruption [69].

Findings on Controller Participation

Controller participation in the resource pool did not affect the number of successfully scheduled events, which was relatively similar for both cases for all event generation rates. However, the number of event rejections appears to be decreased for generation rates of 8, 10, and 12 with a reduction of 50%, 61% and 66% respectively. On the contrary, deadline

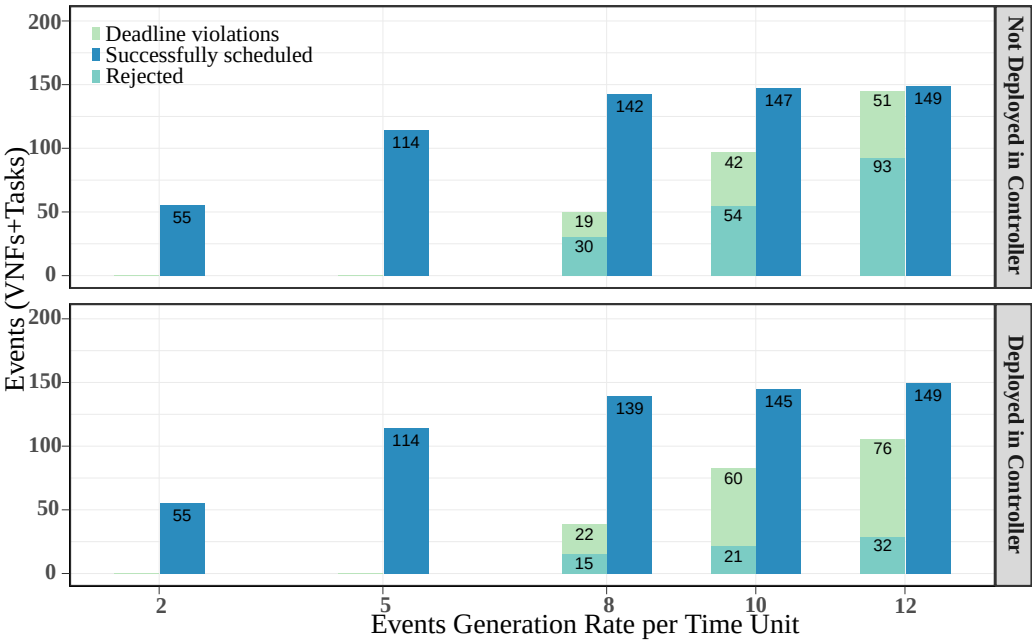
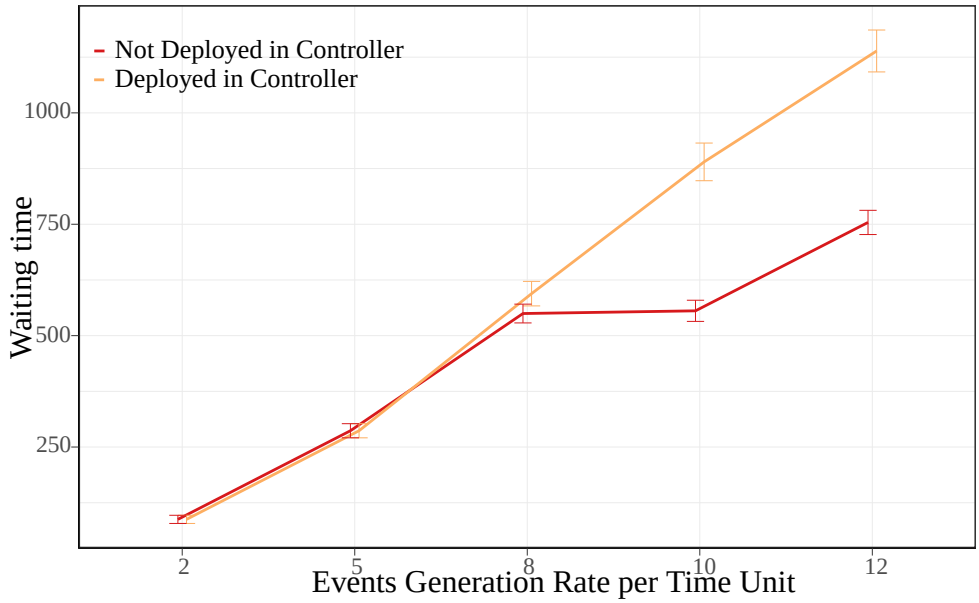


Figure 4.9: Assessment of controller participation by measuring event rejections and deadline violations over different event generation rates.

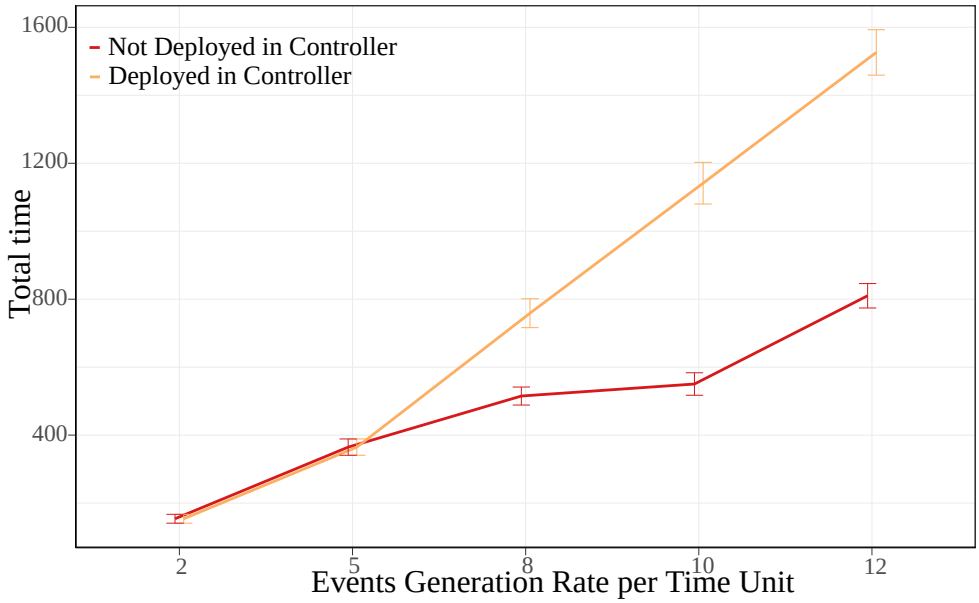
violations appear increased compared to a non-participating controller, with rates of 16%, 42%, and 49% for the same generation rates. This increase reveals two things: (i) the controller takes more time to remove events from the queue when it acts as a computing node and as a result events exceed their deadlines; (ii) an increase of the total amount of resources does not guarantee a faster performing cluster and caution should be taken when allocating roles at each node in the cluster during network planning. Assuming QoS is a soft constraint for IoT workloads (as explained in the experimental setup), deadline violations are not severely compromising performance. If this assumption changes, then we can infer that QoS will be affected proportionally to deadline violations.

Figure 4.10a shows the average waiting time of the events for both criteria. The waiting time for events that are scheduled in the controller appears increased (soft orange line) when compared with events scheduled in regular compute nodes (strong red line). This is the result of the scheduling algorithm which limits the number of deployed events in the controller to just one, in order to ensure that there is no saturation of the system components because of excessive load in the controller. Similar results were obtained for the average total time of the events, which includes the waiting time, as seen in Figure 4.10b. Deployment of events in compute nodes does not involve this constraint, and therefore happens at a faster pace. For 8 and 10 events per Time Unit it is clear that waiting time is minimally impacted. This shows that the specific arrival rates are matching the testbed resources which is able to quickly perform the ranking of nodes and allocate events.

A reduced number of rejections is assumed to be a better overall outcome for our



(a) Waiting time.



(b) Total time.

Figure 4.10: Average metrics time while deploying or not events in the controller node.

cluster since it satisfies a hard constraint. In Figure 4.11, we show the average acceptance ratio of events for both criteria. Events are generated following Poisson distribution and using randomly selected values values from Table 4.2 following a uniform distribution. For a generation rate of 2 and 5 events per time unit, both criteria have an acceptance rate of 100%. However, for generation rates greater than 5 events per time unit, performance was different. Specifically, when deploying events in the controller node, the events acceptance ratio was increased by 11%, 28% and 30% for generation rates of 8, 10 and 12 events per time unit, respectively.

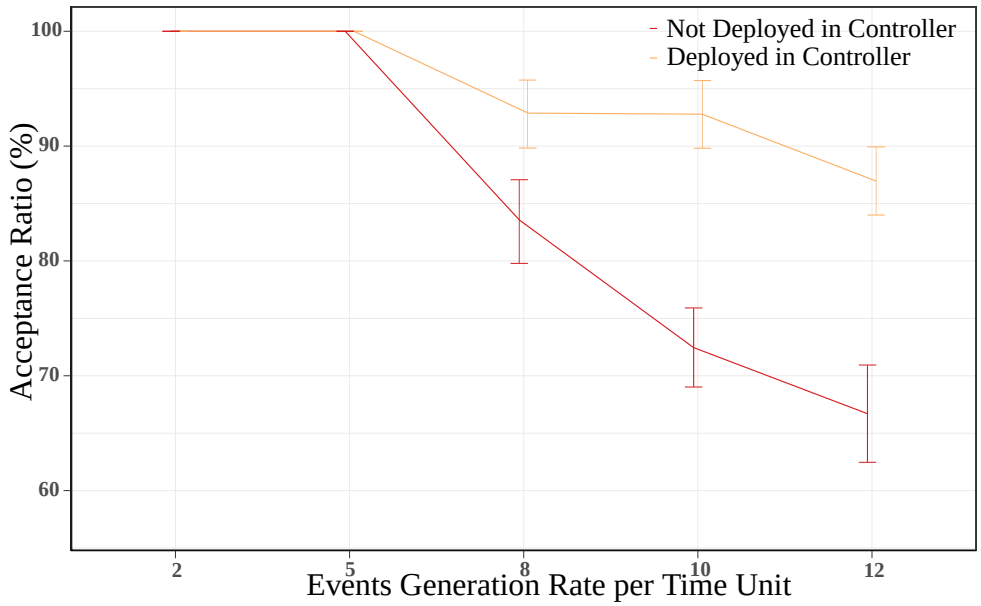


Figure 4.11: Average events acceptance ratio with and without deploying events in the controller node.

Based upon these results, we can confirm that using the controller node to deploy specific events guarantees a higher event acceptance ratio than using its resources solely for scheduling tasks. The higher acceptance ratio was evidenced by a significant reduction in the number of rejected events, although, it was at the cost of greater deadline violations. Overall, the advantages outweigh the drawbacks when deploying services and tasks on the controller node in a resource-constrained environment. Thus, the proposed scheduler has been implemented by taking these results into account.

SoC Regression Model

Data driven methods can derive the statistical relevance between usage of resources and SoC measurements. Such methods are autoregression moving average, artificial neural networks, support vector machine, etc. [45]. These methods can be computationally expensive to train, especially when applied over large training data. Additionally, they must be trained in advance before their hyper-parameters can be adjusted. These methods

are thus deemed unsuitable for resource-constrained environments like an edge cluster.

A simpler method that is able to estimate future values based on previous trends is regression. In the case of regression, the coefficients are determined from available training data by minimising the Root Mean Square Error (RMSE) between the predicted and real values. The RMSE represents the standard deviation of the prediction errors, thus showing how concentrated the data is around the line of best fit [53]. In general, regression models can be classified into two types: polynomial and linear regression models. The former may include higher powers of one or more predictor variables and are defined in Equation (4.6). The latter may include the interaction effects of two or more variables, and a typical example is represented in Equation (4.7).

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ...\beta_k x_k^n \tag{4.6}$$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + ...\beta_{mn} x_m x_n \tag{4.7}$$

Several regression models were studied to choose the one that best fit our case of study. Figure 4.12 shows three polynomial models of first, second and third order which use CPU usage as a predictor to estimate the SoC in a compute node. To describe the accuracy of the models, we included the adjusted R^2 parameter which reflects the variation of predictors. Additionally, the adjusted R^2 parameter does not automatically increase when more predictors are added to the model. According to this metric, the best model was the third-order model, which is represented by the blue line. However, given the negligible accuracy difference of the third-order model and the computation overhead it requires, the second-order polynomial was used instead to strike a balance between accuracy and computation cost.

The controller node has different behaviour compared to compute nodes since its main function is to schedule service and task requests which demand less CPU usage than hosting services and tasks. Figure 4.13 depicts the controller's behaviour when CPU usage reaches its highest value of around 1900 milliCPU (milliCPU: fraction of CPU resources used in Kubernetes [10]) and starts to decrease while the SoC begins to decline. From the adjusted R^2 values, we can see that none of the analysed models, based on CPU usage, fit the data correctly because the values were below 0.80. We have hence analysed the number of incoming packets in the controller node to measure user requests and worker messages. The studied polynomial models are shown in Figure 4.14. This figure shows that all tested models fit the data as their adjusted R^2 was 1.

The CPU usage represents a boundary parameter for the SBCs because when a node reaches its maximum value it cannot process any new requests. As a result, CPU usage must also be considered in the regression model for the controller node. In this regard, figure 4.15 depicts a three-dimensional representation of the regression model for the con-

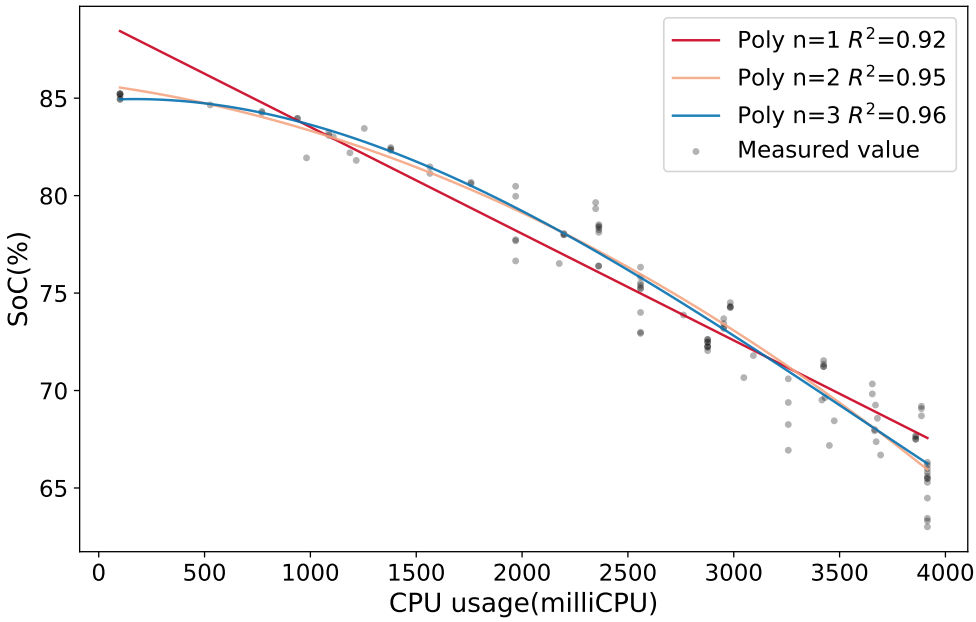


Figure 4.12: Multiple regression models for SoC estimation based on CPU usage for compute nodes.

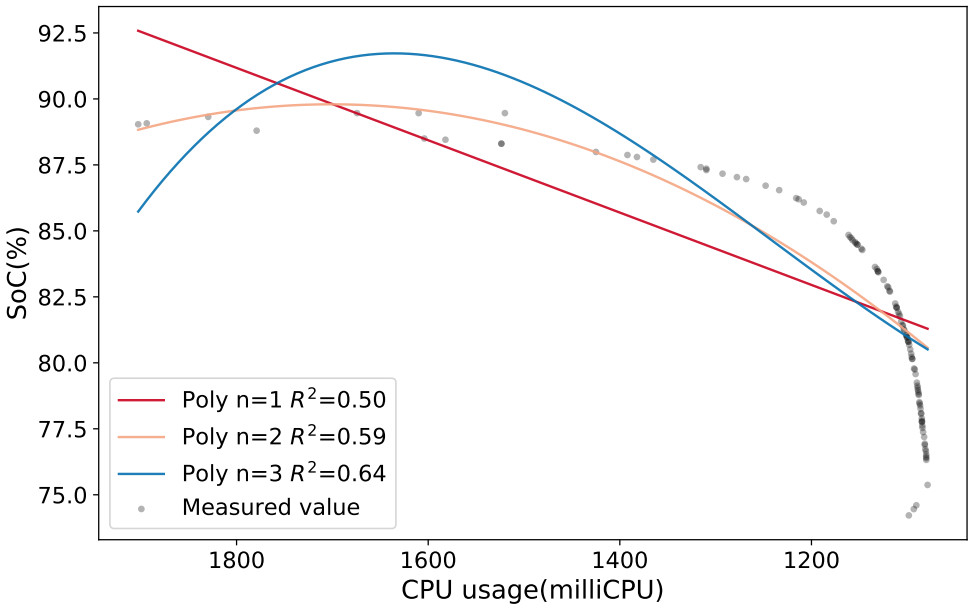


Figure 4.13: SoC regression based on CPU usage for the control plane node.

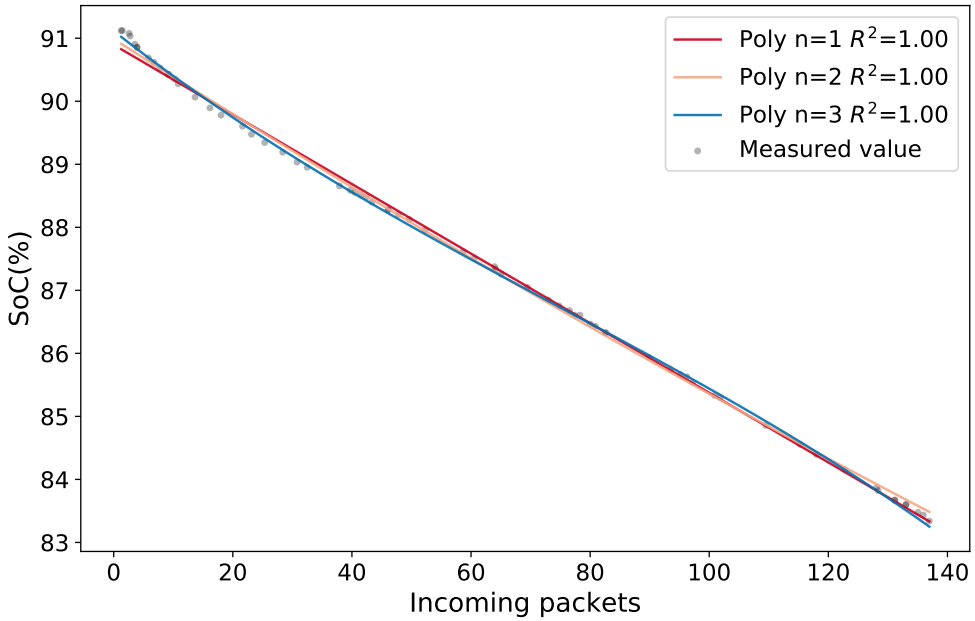


Figure 4.14: SoC regression based on incoming packets at the control plane node.

troller node based on incoming packets and CPU usage. When using two predictors, the least square regression line becomes a plane with two estimated slope coefficients. The model's coefficients are estimated by finding the minimum sum of squared deviations between the blue plane and the measured values. With consideration of the adjusted R^2, this model fits our data since it has the highest possible value.

Finally, the SoC regression model for the SBC cluster can be expressed as follows:

$$SoC_{pred_{model}} = \delta_0 + \delta_1 \cdot pkt_{in} + \delta_2 \cdot cpu + (\delta_{3_i} \cdot compute_i) \cdot cpu^2 + \delta_{4_i} \cdot compute_i \quad (4.8)$$

The model coefficients (δ) were obtained from the available training data set. Categorical variables $compute_i$ are introduced to represent each compute node in the model. These variables are binary. The node whose SoC is to be predicted takes a value of 1 and the others take a value of 0 (e.g., $compute_1$=1, $compute_2$=0, ..., $compute_n$=0). Notice that in the case of the controller node, all categorical variables are 0 (e.g., $compute_1$=0, $compute_2$=0, ..., $compute_n$=0). Thus, the SoC regression model is transformed into a linear model with two predictors (i.e., pkt_{in} and cpu). In contrast, for the compute nodes, a second-order polynomial model based on CPU usage was used since the pkt_{in} term was discarded due to its low impact when compared with CPU usage.

Algorithmic Components of Node Selector

Procedure SoC estimation summarises the SoC estimation methodology. It uses the aforementioned trained regression model to estimate the SoC value that a node will have if a specific virtual node was assigned to it (explained in more detail below) [53]. Furthermore,

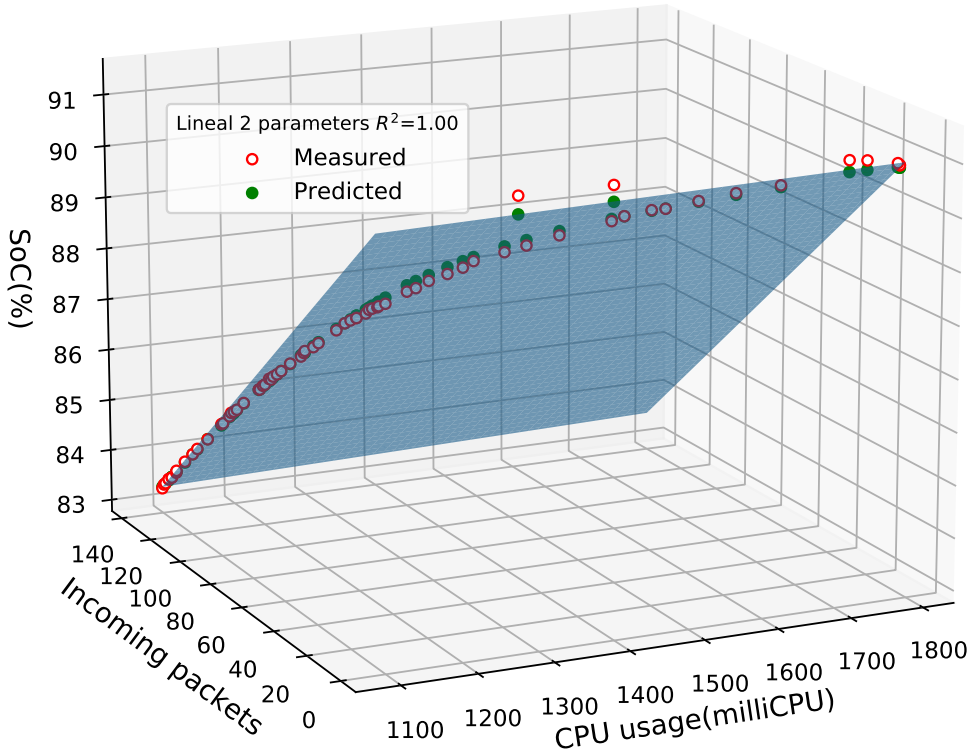


Figure 4.15: Lineal regression model with CPU usage and incoming packets as predictors for the controller node.

by using the outcome of the controller participation experiments it integrates the controller in the list of candidate nodes in a way that increases the total amount of resources but also prevents system saturation due to extensive workload deployment.

The SoC estimation procedure uses a physical and a virtual node as input arguments. Then, using the trained regression model, it predicts the SoC value if the allocation of virtual node to the physical node proceeds. The first step in the SoC prediction procedure is to initialise the output variable and create an empty set to store the data used by the prediction model (line 1). Next, the procedure checks if the input node is the controller node in order to adjust the model arguments and provide accurate predictions using network measurements (line 2). Lines 3-6 determine the values required by the model and store them in the *data* variable. In line 3, the expected CPU usage of the node is calculated by adding the current CPU usage and the CPU usage in case the event is allocated in the physical node. Lines 4 and 5 determine the expected overall number of exchanged packets between the controller and the computing nodes once the virtual node is scheduled. The obtained values are then stored in the data set (line 6). In the case of computing nodes, the procedure only factors in the expected CPU usage and saves this value in the data set (lines 7-9). Finally, the prediction value is obtained from the SoC regression model using the stored data as input (line 10).

Algorithm 3 is constantly running within ECAS and it is triggered by events entering

Procedure SoC estimation

Input: p^0, n

Output: SoC_{value}

1 $SoC_{value} \leftarrow 0$, $data \leftarrow \varnothing$
2 **if** n *is controller* **then**
3 $\quad cpu \leftarrow n_{usage_{CPU}} + p^0_{CPU_{req}}$
4 $\quad pkt_{in} \leftarrow n_{pkt_{in}} + \frac{n_{pkt_{in}}}{\sum_{p \in P}}$
5 $\quad pkt_{out} \leftarrow n_{pkt_{out}} + \frac{n_{pkt_{out}}}{\sum_{p \in P}}$
6 $\quad data \leftarrow cpu, pkt_{in}, pkt_{out}, n$
7 **else**
8 $\quad cpu \leftarrow n_{usage_{CPU}} + p^0_{CPU_{req}}$
9 $\quad data \leftarrow cpu, n$
10 $SoC_{value} \leftarrow SoC_{pred_{model}}(data)$
11 **return** SoC_{value}

the priority queue (line 1). The algorithm fetches the resource utilisation metrics for each physical node from the SoC Monitor component (line 2). It then takes the first element in $l_{priority}$ and initialises the list of candidates that can host the new virtual node (lines 3-5). In the next steps, a list of potential candidate SBCs for the virtual node is formulated. Each physical node with a battery percentage above a minimum predefined value and status set to "available" is added to $l_{candidate}$ (lines 6-8). If there are more than one available candidates for deployment and the controller belongs to the list, then it is removed to prioritise allocation to compute nodes (lines 9-10). This is done to reduce unnecessary use of the controller's resources and avoid unnecessary processing that can delay core functions of the cluster. In this manner, controller resources are only used when the rest of the cluster resources are reserved. If the previous condition is not met (line 9), two scenarios are examined (line 11):

- There are no available resources to host a virtual node.
- The only available node is the controller and the event has arbitrary execution time.

In both cases, the associated event is rejected (lines 12-22) and the $Event_{rejected}$ metric is incremented. The created virtual node for that event is removed and P, l_S and l_T are updated. In the case of a service event, the algorithm checks each participating network function: if the function has already been deployed, it is deleted and the associated resources are released; if the function is in the priority queue, it gets removed to save scheduler resources. After not meeting the aforementioned conditions (lines 9 and 11), there is at least one node in $l_{candidate}$ (line 23). Notice that the controller node can be in $l_{candidate}$ when there are no more available computing nodes and the events to be placed

have a specified running time ($t_r > 0$). ECAS in this case is trying to minimise the number of rejected events. At this stage, nodes in $l_{candidate}$ are ranked using equation 4.9 to find the best one to deploy the event (lines 24-32). In line 26, the algorithm uses the SoC predictor model to calculate the SoC of the current candidate node using Procedure SoC estimation (line 27).

The output of the SoC prediction procedure is used in Algorithm 3 to calculate the node score through Equation (4.9) (line 28). With this equation, the algorithm tries to maximise the node score by selecting the node with the highest SoC and the minimum CPU usage.

$$n_{score} = \alpha_1 \cdot (SoC/100) + (1 - \alpha_1) \cdot (1 - \mathbb{E}_{CPU} / Usage_{CPU_{max}}) \tag{4.9}$$

In Equation (4.9), the value α_1 is an adjustable positive weight with values between 0 and 1. SoC is a value between 0 and 100, as obtained from the monitoring component. ($\mathbb{E}_{CPU}$) represents the expected CPU usage of the node if the assignment of the event proceeds. This value is derived by adding the current CPU usage of the node and the CPU usage described in the event requirements. After calculating n_{score} for all nodes, Algorithm 3 selects the one with the highest score (line 31). Finally, by communicating with the orchestrator, ECAS binds p^0 in n^{best} (line 33).

Algorithm 4 shows an overview of the sequence of the different procedures and algorithmic components of ECAS. It initialises the set of virtual nodes, which can be updated by both the node selector and the monitor block and the event lists (lines 1-3). In addition, it initialises the SoC prediction model used in Algorithm 3 and the data set to train the regression model (line 4). Algorithm 4 listens for event requests while the node selector component is running (line 8). When a request arrives, the algorithm adds it to the corresponding list in line 9. Then, it creates a virtual node with the specified requirements and adds it to the set of virtual nodes that need to be scheduled (line 10). Events are ranked in the priority queue ($l_{priority}$) (lines 11-13) which is determined by a process that considers two factors: $delay(p)$ and $wait_{queue}(p)$. The former represents the amount of time that the scheduler can delay the execution of an event without missing its deadline (see Equation (4.10)). The latter is the waiting time of the event before being processed by the scheduler (see Equation (4.11)). In both equations, we denote the current system time as t_{now}. Note that the smaller the $delay(p)$, the faster the created virtual node will be executed.

$$delay(p) = p_d - t_{now} - p_{t_r} \tag{4.10}$$

$$wait_{queue}(p) = t_{now} - p_{t_a} \tag{4.11}$$

Algorithm 3: Event Scheduler

while $len(l_{priority}) > 0$ **do**
  Get node's capacity information
  $p^0 \leftarrow$ First element of $l_{priority}$
  Update $l_{priority}$
  $l_{candidate} \leftarrow \emptyset$
  forall $n \in N$ **do**
    if $n_{usage_{SoC}} > SoC_{min_{threshold}}$ ***and*** n_{status} *is scheduled* **then**
      $l_{candidate} \leftarrow l_{candidate} + n$
  if $l_{candidate} > 1$ ***and*** $n^{controller} \in l_{candidate}$ **then**
    Remove $n^{controller}$ from $l_{candidate}$
  else if $l_{candidate} == 0$ ***or*** $(l_{candidate} == 1$ ***and*** $n^{controller} \in l_{candidate}$ ***and*** $p^0_{t_r} == 0)$ **then**
    if p^0_{event} *is Service* **then**
      forall $f \in F$ **do**
        if p_{f_i} *is deployed* **then**
          Delete p_{f_i} to release its resources
        if $p_{f_i} \in l_{priority}$ **then**
          Remove p_{f_i} from $l_{priority}$
    else
      Delete p_T to release its resources
    Remove p from P
    Remove S or T from l_S or l_T accordingly
    $Event_{rejected} = Event_{rejected} + 1$
  else
    $n^{best} \leftarrow$ First element in $l_{candidate}$
    forall $n \in l_{candidate}$ **do**
      if $SoC_{pred_{model}} \neq \varnothing$ **then**
        $n_{SoC_{pred}} \leftarrow$ **Procedure SoC estimation:** Predict SoC(p^0, n)
        Calculate n_{score} using Equation (4.9) with $n_{SoC_{pred}}$
      else
        Calculate n_{score} using Equation (4.9) with $n_{usage_{SoC}}$
      if $n_{score} > n^{best}_{score}$ **then**
        $n^{best} \leftarrow n$
    Bind p^0 to n^{best}

Algorithm 4: Main process

1 $P \leftarrow 0$
2 $l_S \leftarrow 0$ (List of running services)
3 $l_T \leftarrow 0$ (List of running tasks)
4 $SOC_{pred_{model}} \leftarrow \varnothing$, $training\_data \leftarrow \varnothing$
5 **Algorithm 2:** Update Status of Events
6 **Algorithm 3:** Event Scheduler
7 Training of regression model
8 **while** *True* **do**
9 Add S or T to l_S or l_T according to event requests
10 Create p for S or T and add it to P
11 Determine maximum delay to process p through Equation (4.10)
12 Determine the time before putting p into priority queue through Equation (4.11)
13 Calculate p_{rank} using Equation (4.12)
14 Add p to $l_{priority}$
15 Sort $l_{priority}$ by virtual node ranking

Based on the previous definitions, we calculate the ranking score for the virtual node where an event runs, denoted by p_{rank}, as follows:

$$p_{rank} = \beta_1 \cdot delay(p) - (1 - \beta_1) \cdot wait_{queue}(p), \tag{4.12}$$

where β_1 is an adjustable positive weight with values between 0 and 1. A virtual node with the lowest ranking must be executed first. Thus, the algorithm updates the priority list and sorts the queue by taking into account the calculated ranking of the virtual node (lines 14-15).

4.5.3 Evaluation

Here we demonstrate the performance of the design decisions explained in the previous sections when combined together and integrated within Kubernetes.

4.5.4 Evaluation Setup

A similar cluster to that of Section 4.5.1 has been used, with a few upgrades. Raspberry Pi 3 Model B were replace by Raspberry Pi 4 Model B [66]. While both devices are representative of modern IoT nodes, in terms of cluster capabilities, Model 4 has higher processing capacity and therefore evaluation can conducted using a more diverse set of events. A higher capacity battery was also used as the energy source, raising the capacity from 5200 mAh to 10000 mAh. The sensor device that obtains energy measurements remained the same. Kubernetes 20.04 was deployed as the management framework for

Table 4.2: Evaluation parameter ranges based on testbed.

Number of VNFs in a Service	5 - 10
Processing Capacity per Node (MIPS)	500 - 3000
CPU Capacity per Node (milliCPU)	4000
Memory Capacity per Node (Ki)	7998464
Required Processing Rate per Event (MIPS)	100 - 500
Required CPU per Event (milliCPU)	150 - 250
Required Memory per Event (Ki)	200 - 500

virtualised services and tasks. Services and tasks are virtualised using Docker containers within pods. The deployed events are placed into the devices and utilise their available capacity according to predefined requirements. ECAS was implemented using Python 3.6.8 and deployed within Kubernetes, replacing the baseline scheduler.

In our evaluation scenarios, the services and tasks to be scheduled arrive one at a time following a Poisson distribution. We explore different event arrival rates that range from 2 to 12 events per time unit. The main parameters used for creating the services and tasks are selected randomly from the list of values shown in Table 4.2 following a uniform distribution. The evaluation parameters are defined based on typical workloads derived from the literature. CPU usage is measured in *CPU* units and is expressed as an absolute quantity. Thus, 100 milliCPU and 0.1CPU are approximately the same amount of CPU usage in a single-core, dual-core, or 48-core machine. This conforms with Kubernetes' design and allows portability of the designed platform to different clusters [10].

The proposed scheduling component, ECAS, is compared against the popular greedy Least Loaded Scheduler (LLS), and the native Kubernetes Scheduler (KS). LLS aims to allocate the different events to the node with the highest available buffer capacity. Thus, the node whose CPU is least utilised is ranked first. KS first filters the nodes that can host the event by checking the requirements of the event and the node's specifications. It then ranks all of the candidate nodes based on the remaining processing resources they will have if the assignment proceeds and selects the one with the highest amount of remaining processing resources. There are more scoring criteria in the process of node scoring (affinity, anti-affinity of pods, data locality, etc) for KS, but to ensure a fair comparison they have been disabled. The main metrics used to evaluate their performance were: scheduled and rejected events, deadline violations, waiting time and total time, acceptance ratio and battery consumption. To ensure the reliability of the obtained results, for each generation rate the experiment is run multiple times for all scheduling algorithms to obtain a 95% confidence interval.

Scheduled, Requested, and Rejected Events

Figure 4.16 depicts the obtained results in terms of requested, scheduled and rejected tasks and services. We can observe that all the schedulers, except for the KS, achieved similar results when the generation rate was small (i.e., 2 and 5) and managed to deploy all the requested services (rejected services were 0). The KS had 2 rejected services (red bar) for a generation rate of 5 events per unit time. We also notice that for high generation rates (i.e., 8, 10 and 12), KS rejected 2 tasks (salmon bar). In contrast, the other algorithms deployed all the requested tasks (mid blue bar).

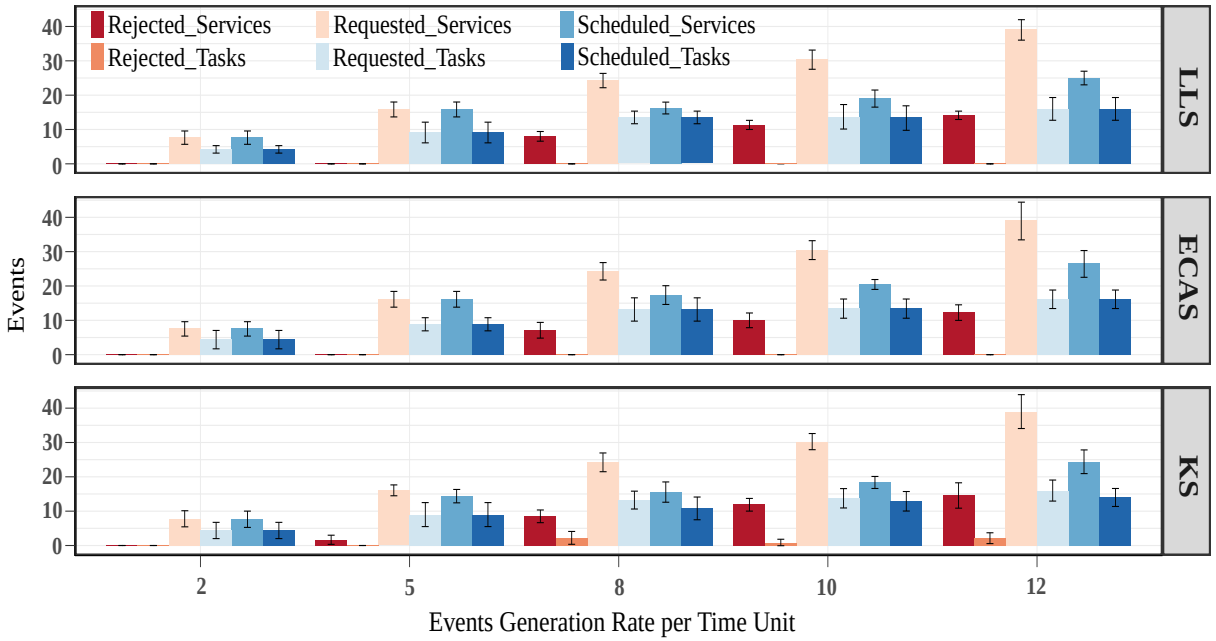


Figure 4.16: Number of requested, scheduled and rejected events (i.e., Services and Tasks) for each scheduling algorithm.

Since generated services are formed by several VNFs, figure 4.17 shows the obtained results for their constituent VNFs. By looking at rejected VNFs for low-value generation rates, KS appears to have rejected 8 VNFs (apricot bar) for the rate of 5 events per time unit. By contrast, the other schedulers were able to schedule all requested VNFs for low generation rates. In terms of scheduled VNFs (windows blue bar), ECAS outperformed LLS and KS by 19% and 17%, 8% and 7%, respectively, for low event generation rates. For higher generation rates the differences are incremental with 5% and 2%, respectively. Additionally, the ECAS reduced the rejected VNFs w.r.t LLS by 11%, 12% and 12% for 8, 10 and 12 events per unit time, respectively. For the same generation rates, reductions were higher when compared against KS, with values up to 16%, 23% and 18%. For low event generation rates, which are more typical for an IoT cluster, ECAS demonstrates that events are less likely to be initially accepted and then evicted due to low energy resources. This is reflected in the increased number of scheduled events but also with fewer event

rejections on average.

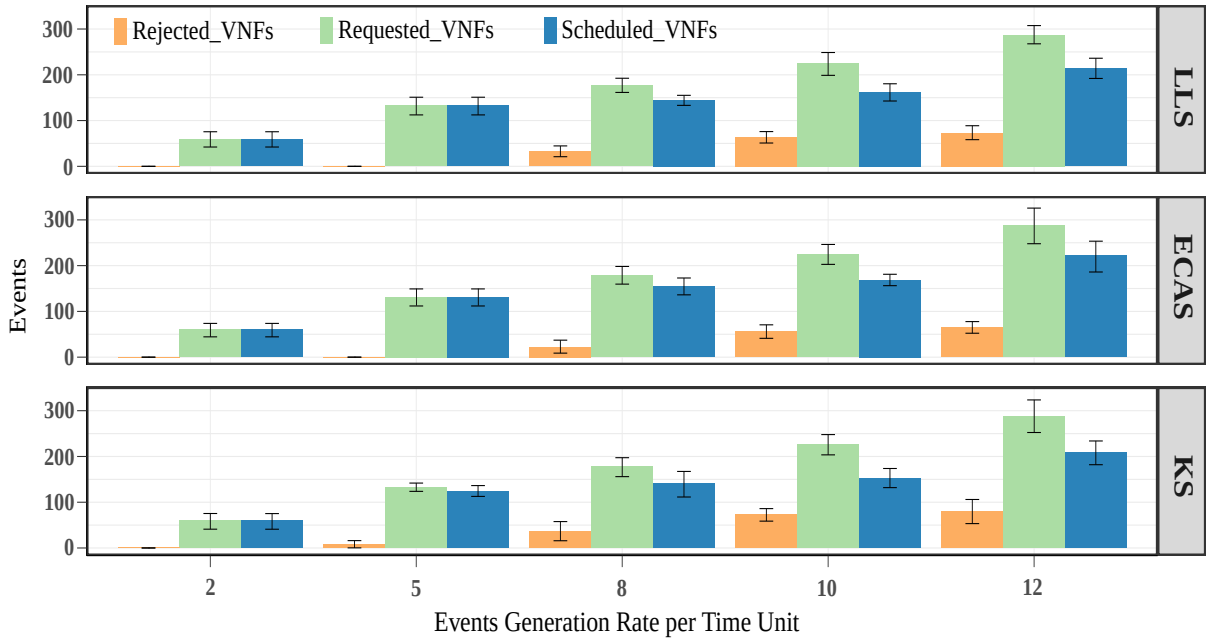


Figure 4.17: Number of requested, scheduled and rejected VNFs per scheduling algorithm.

Average Acceptance Ratio

The acceptance ratio represents the proportion between scheduled and requested events. Figure 4.18 shows the average acceptance ratio for various generation rates for all considered schedulers. KS had the worst performance across all generation rates with its lowest point being around 72% for 12 events/time unit. LLS and ECAS showed similar performance for low generation rates. For arrival rates of 8 or more events per unit time ECAS showed increased acceptance ratios by around 2% with respect to LLS. The performance of KS shows that the default ranking approach, originally designed for commodity servers, ignores energy resources and fails to adapt to low-end devices which operate well under reduced resources. In contrast, LLS – by relying primarily on CPU resources captures the potential of IoT devices better. ECAS further improves this by including energy resources.

Average Number of Scheduled Events and Deadline Violations

Figure 4.19 illustrates a deeper insight into the number of scheduled events, since it separates the events with deadline violations from the successful ones. In general, ECAS showcased reduced deadline violations compared to the others for all the generation rates. For the highest generation rate (i.e., 12 events per unit time), ECAS decreased the number of deadline violations by 75% and 83% w.r.t LLS and KS, respectively. Only for the case of 5 events per unit time, LLS had the lowest value of all. This indicates that events

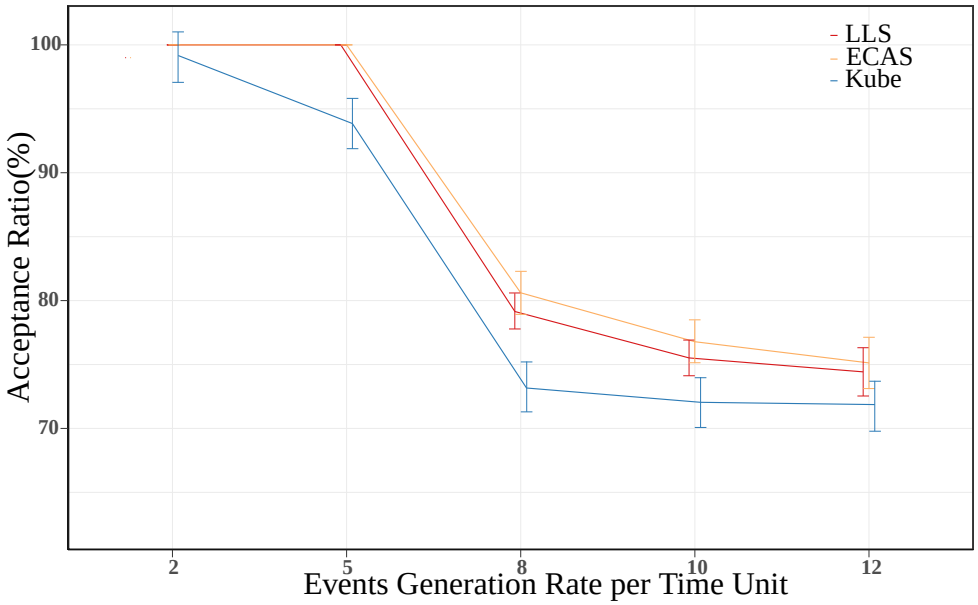


Figure 4.18: Acceptance ratio of events per scheduling algorithm.

arriving at lower generation rates are less likely to cause resource contention and a greedy approach is adequate to ensure timely fulfilment of the requests.

Average Waiting and Total Time

As seen in Subsection 4.5.2, the values of the waiting and total times are higher when the scheduler deploys events in the controller node. From Figures 4.20, 4.21, it is affirmed that ECAS had the lowest increment for event waiting time and total time. The waiting time is reduced by 42% and 53% w.r.t to LLS and KS for high event generation rates, respectively (Figure 4.20). Additionally, ECAS decreased the total up to 34% and 53% in comparison with LLS and KS, respectively. KS is particularly slow at managing the queue of events, with various processes delaying the deployment of events to perform ranking and parsing of the queue. This is addressed with LLS and ECAS, with the latter allowing faster processing of events by deploying them in less congested nodes with adequate energy capacity.

Average Battery Consumption

To measure battery consumption, the difference between the initial and final measured SoC was compared for various event generation rates. Figure 4.22 shows the average battery consumption for each node for all studied scheduling algorithms. The results show that increases in arrival rate translate to higher battery consumption. KS appears to have the highest battery consumption for all generation rates when compared to the rest. In contrast, the node selection criteria of ECAS ensured the lowest battery consumption. This is attributed to two main aspects of the scheduler:

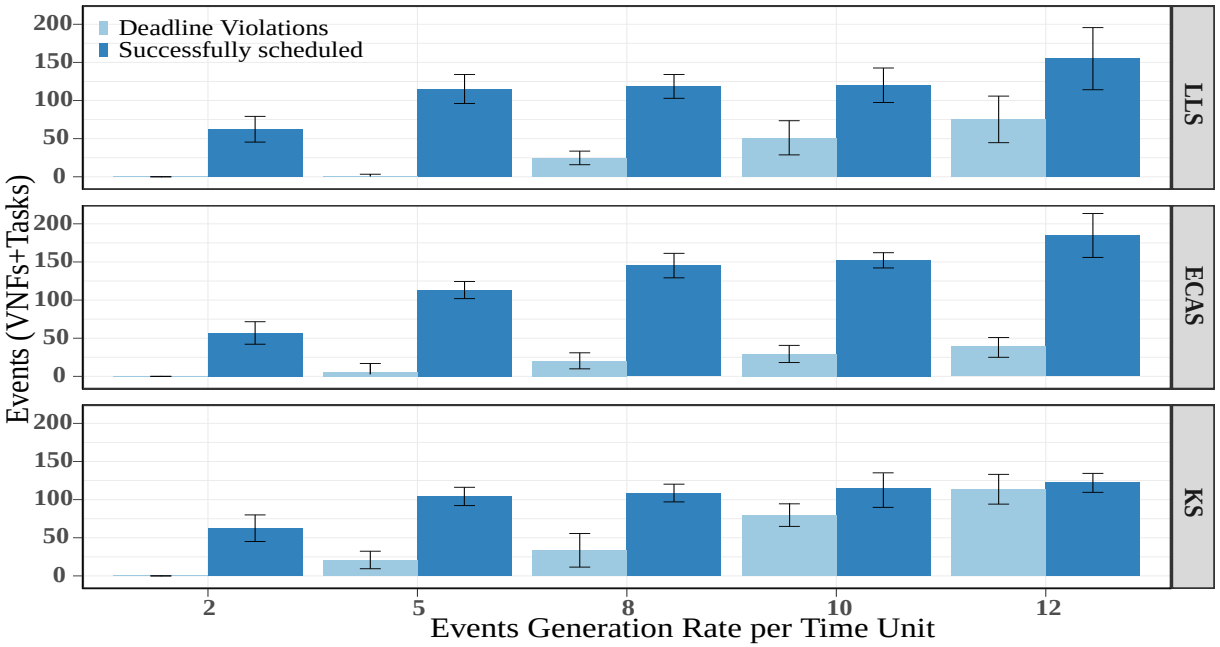


Figure 4.19: Number of successfully scheduled events and deadline violations for each scheduling algorithm.

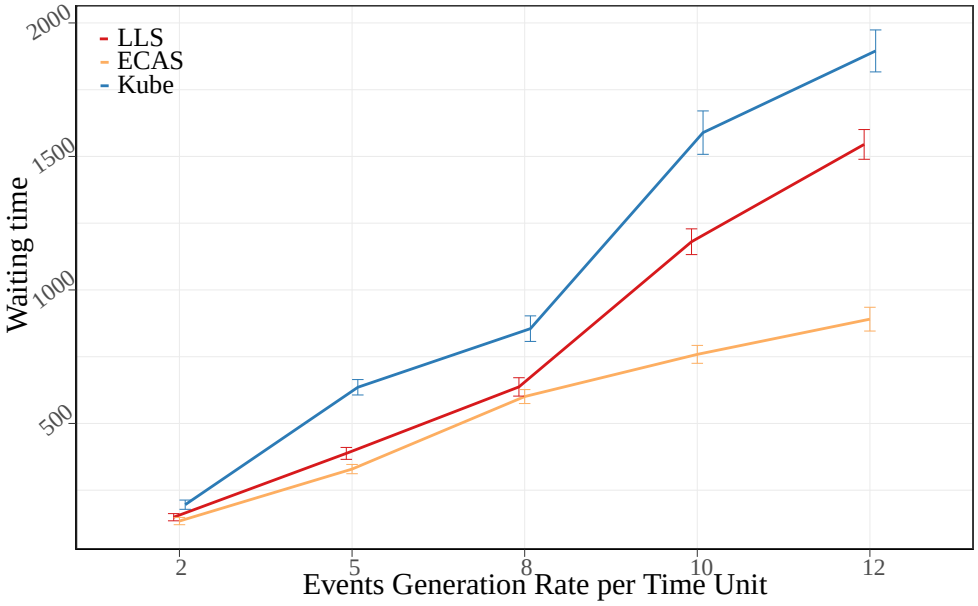


Figure 4.20: Waiting time for all scheduling algorithms.

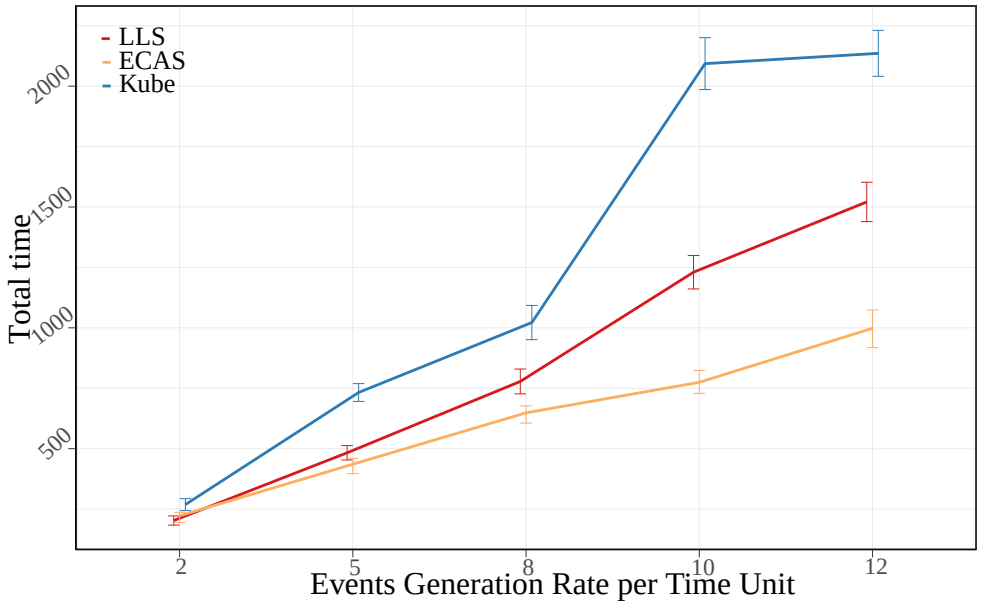


Figure 4.21: Total processing time for all scheduling algorithms.

1. It always selects nodes with high SoC and low CPU utilisation, reducing resource contention.

2. The reduced number of deadline violations reduce the time events spend in the system, and therefore lower the impact in battery consumption when compared to the other schedulers.

By taking a closer look at the highest generation rate, our scheduler is shown to have saved up to 39% and 59% of in battery resources for the master node w.r.t LLS and KS, respectively. Likewise, it decreased the consumption in Worker1 by 36% and 51% in comparison. It was also found that ECAS reduced imbalances in power consumption among workers: for a generation rate of 12 events per time unit, ECAS presented an imbalance of 0.25 between the maximum and minimum average battery consumption. On the contrary, for the same generation rates, LLS and KS scored 2.82 and 3.61, respectively.

4.5.5 Discussion

Apart from the advantages of energy-aware placement shown over a series of tests, there is also another subtle observation that relates to the absolute values in terms of deployment time and processing time for various event generation rates. The reader can easily notice that the waiting time for an event can vary from a few minutes up to a substantial duration of time (close to 30 minutes) depending on the rate of generation. Our analysis shows that waiting time constitutes a significant percentage of the total processing time for an event. The devices used in the previous experiments are indicative of IoT equipment, limited in processing resources and unable to mitigate this problem. Offloading parts of

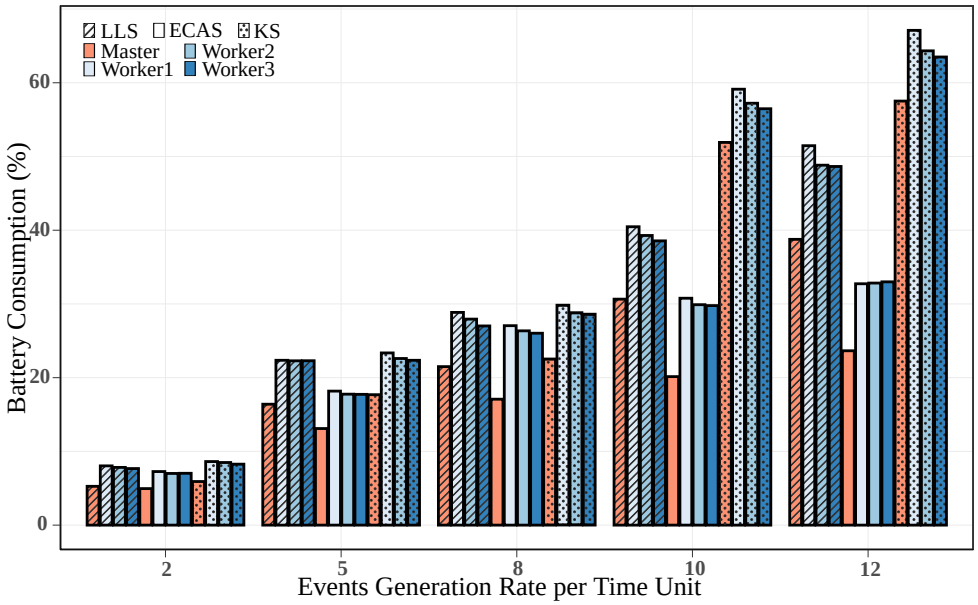


Figure 4.22: Battery consumption for each node while running different scheduling algorithms.

the computation within the network fabric can be accelerate certain processes such as deployment times and reaction time to events. In the Chapter 3, an architecture that allows offloading KV pairs within the PDP has been presented. This mechanism can be adapted to work in edge environments with the use of small form-factor programmable switches. For example, devices like the APS 2112d can be deployed in close proximity to clusters of IoT devices and enable line-rate access to replicated storage or other in-network compute services [146]. Offloading of ECAS parameters can also happen in data plane to provide faster updates and allow access from other network devices or higher-tier nodes, operating in close proximity to the cloud or data centre. This is another example of how offloading at the data plane can facilitate real-time sharing of information that leads to synergy between different network segments.

4.6 Query allocation at the Edge

In the previous sections, cluster formulation at the edge and local decision making optimised service deployment around available resources. However, with infrastructure spanning across both data centre and the network edge there is need to scale management and provide a converged management architecture. This thesis has shown that deployment of data in PDP supports high-throughput responses to queries (Chapter 3). A converged architecture needs to place management information across all available infrastructure in a way that relieves heavy processing from edge nodes and transfers it to cloud infrastructure. An important question arises from this endeavour: in a scenario where a cluster of data-

gathering IoT devices is interconnected to the cloud (possibly through a programmable switch), what type of information should be placed at the gateway to the cloud to enhance data representation and query processing for the whole infrastructure? This section presents a new mechanism that allows data to remain stored in edge nodes while simultaneously maintaining meaningful representations of these data in the cloud infrastructure (e.g., a programmable switch). This information can be used to direct queries to statistically relevant nodes which can process the queries and transfer the replies back to the cloud. It contributes by minimising the amount of data transferred over the network, reducing bandwidth, energy, and usage of processing resources.

Edge Nodes are regarded as distributed data repositories, holding a KVS with sensor readings, where queries can be executed using the available processing resources. The efficient management and allocation of incoming analytics queries as well as the provided results characterise the success of the supported applications. Usually, applications demand a response in the minimum time to provide high quality services to end users. Hence, the Edge Nodes (ENs)/Query Processors (QPs) should adopt query allocation and execution plans that limit the time required for obtaining the final analytic result.

Should we desire to significantly reduce the time required for delivering final results of incoming analytics queries, the selection of the appropriate query execution plan is the first step of the process. Then, one should involve the selection of an efficient query allocation plan, i.e., selecting the most appropriate subset of nodes that will deliver the appropriate results according to queries semantics and in the minimum time. Then, an efficient aggregation mechanism on the partial results should be invoked. This work focuses on the allocation process of analytics queries arriving at a QC, either located in the Cloud or in a master node at the edge, taking into consideration the sufficient statistics (*statistical signatures*) of the data present in each EN. The underlying nodes are considered to be logically clustered based, for example, on geospatial criteria imposed by the analytics applications. A decision making mechanism is introduced that exploits the statistical signatures of nodes' data sets and concludes on the most appropriate subset of nodes for allocation and execution. Based on this approach the time for getting responses is reduced and the quality of the responses is improved as nodes with irrelevant data (compared to queries semantics) are excluded from the query allocation and execution process. The ENs regularly send their statistical signatures to the QC (at pre-defined intervals) involving the elimination of outliers and the reduction of the data dimensionality. This way, ENs send reliable statistics for their dataset that consequently support the QC's decision making.

4.6.1 Definitions & Problem Formulation

A set of (N) ENs (e.g., Raspberry Pis) is considered: $\mathcal{E} = \{n_1, n_2, \ldots, n_N\}$, placed at various locations, e.g., in a city. IoT sensors are connected with ENs to deliver their

contextual data. At the upper layer, e.g., in the Cloud, there is a set of QCs responsible to receive and execute analytics queries $q_1, q_2, \ldots$ defined by analytics applications and/or end users (data analysts). Such queries are then allocated to the appropriate QPs for execution functioning in the available ENs. Consider the set of N QPs $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$, each one corresponding to an EN. After receiving an analytics query a QC invokes the most appropriate subset $\mathcal{P}' \subset \mathcal{P}$ of QPs to get their query results and return the final *aggregated* result to the requesting applications depending on how *relevant* is the query to the underlying data of each EN. The determination of the subset $\mathcal{P}'$ is achieved from certain statistics of data that each EN delivers to the back-end infrastructure, i.e., QC. Such statistics support the QC with the necessary view on what data are present in each EN used for the statistical matching with each incoming analytics query. Figure 4.23 illustrates the considered architecture.

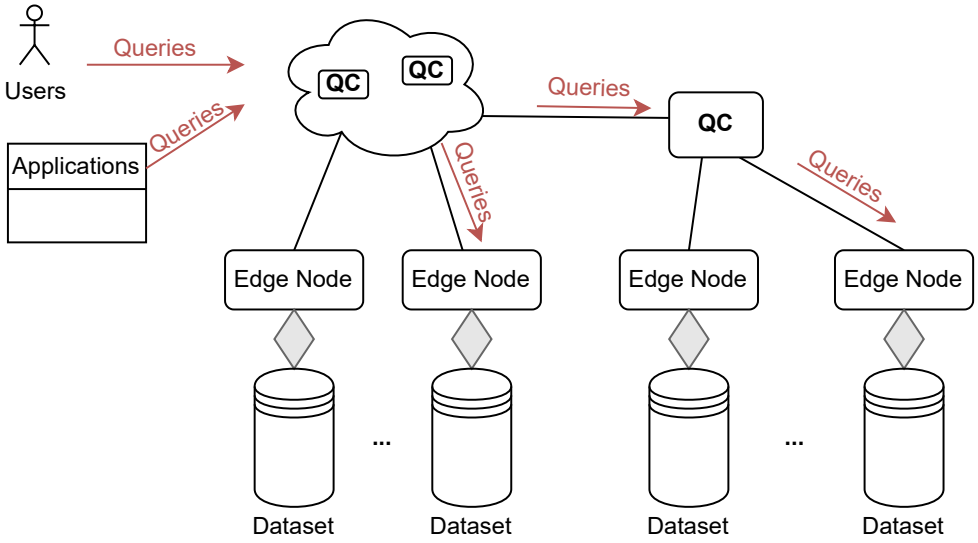


Figure 4.23: An architecture that distributes queries through QCs to edge nodes.

Definition 1 *A data set $D_i = \{\mathbf{x}_j\}_{j=1}^{m_i}$ of the EN i is a set of m_i row data vectors $\mathbf{x} = [x_1, \ldots, x_d] \in \mathbb{R}^d$ with real attributes $x_k \in \mathbb{R}$.*

Analytics queries are issued over a d-dimensional data space and bear two key characteristics: First, they define a subspace of interest, using various predicates on attribute (dimension) values. Second, they perform aggregate functions over said data subspaces (to derive key statistics over the subspace of interest). A general vectorial representation is adopted for modelling a query over any type of data storage/processing system. Predicates over attributes define a data subspace over a data set D formed by a sequence of logical conjunctions using (in)equality constraints $(\leq, \geq, =)$. A *range-predicate* restricts an attribute x_k to be within range $[l_k, u_k]$: $x_k \geq l_k \wedge x_k \leq u_k$, $k = 1, \ldots, d$. A range query is modelled over a dataset D through conjunctions of predicates, i.e., $\bigwedge_{k=1}^{d}(l_k \leq x_k \leq u_k)$ represented as a vector in $\mathbb{R}^{2d}$.

Definition 2 *A (range) row analytics query vector is defined as* $\mathbf{q} = [l_1, u_1, \ldots, l_d, u_d] \in \mathbb{R}^{2d}$ *corresponding to the range query* $\bigwedge_{k=1}^{d}(l_k \leq x_k \leq u_k)$.

For instance, consider an analytics range query asking for extracting the correlation between temperature x_1 and humidity x_2 in the 2-dim subspace $[5, 10] \times [80, 100] \subset \mathbb{R}^2$. If such a query is executed over a dataset where the pairs (temperature, humidity) are outwith the above-mentioned 2-dim subspace, then the corresponding node will waste computational resources for executing this range query. In addition, such results will affect the final response due to aggregation.

Every EN i at pre-defined intervals calculates certain statistics of its dataset D_i forming the statistical signature: $\mathcal{S}_i$. $\mathcal{S}_i$ contains sufficient statistics of the underlying data vectors in D_i. The following statistics are adopted for the signature: the mean row vector $\boldsymbol{\mu} = [\mu_1, \ldots, \mu_d]$, the variance row vector $\boldsymbol{\sigma} = [\sigma_1, \ldots, \sigma_d]$ and the eigenbase $\mathbf{W}_{d\times K}$ of the first $K \leq d$ column eigenvectors $\mathbf{w}_k \in \mathbb{R}^d$ (Principal Components) produced by the Principal Components Analysis (PCA) [80] over the data in D_i (or a sample); see Section 4.6.2. The signature of EN i is:

$$\mathcal{S}_i \quad = \quad \{\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i, \mathbf{W}_{i,d\times K}\}. \tag{4.13}$$

Note that, the signature should be constructed and incrementally updated in limited time, thus, the performance of nodes is not affected. The extraction of the signature $\mathcal{S}_i$ is based on a multidimensional outliers elimination model. The outliers elimination model is based on an aggregation scheme over two known statistical measures of χ^2 and the Grubb's test in order to decide about the presence of outliers in D_i. The outcomes of the two outliers techniques are combined and then the signature is constructed over *outliers-free* data.

Each dataset D_i is updated over the time as data streams are produced by IoT devices at high rates. In our context, the QC does not have any view on the data present in every dataset since data are not delivered to the QC; *only* their corresponding statistical signatures are delivered and updated regularly.

The QC receiving a query $\mathbf{q}$ should conclude on a matching degree between the query and the available signatures $\{\mathcal{S}_i\}_{i=1}^{N}$. Based on this matching, the QC should decide on the most appropriate subset $\mathcal{P}' \subset \mathcal{P}$ of QPs referring to those ENs that will selectively execute the query q. Based on this partial engagement of the QPs, irrelevant ENs are excluded from the query execution thus avoiding providing results that do not match the query predicates. Hence, irrelevant nodes are not involved in the execution of queries whose data are not matched with the queries semantics (represented by predicates). By analogy with the previous sections, a task that invokes fewer nodes allows a more energy-efficient response than a task that invokes all of the nodes. The QC becomes responsible for selecting the appropriate nodes and therefore removes the computational burden from

energy-constrained devices. The problem is formalised as follows:

Problem 1 *Given an analytics query* $\mathbf{q}$ *to the QC and a set of* N *statistical signatures* $\{\mathcal{S}_i\}_{i=1}^{N}$ *derived from* N *ENs, seek the most appropriate subset of QPs* $\mathcal{P}' \subseteq \mathcal{P}$ *which will be engaged for executing the query* $\mathbf{q}$.

4.6.2 Scaling-out the Assignment of Queries

This section describes the methodology for finding the relevant subset $\mathcal{P}'$ of the QPs given a random range analytics query $\mathbf{q}$ in the QC. First, an aggregation mechanism is introduced for removing outliers from the ENs' data sets before constructing the statistical signatures. Then, we provide the construction of the signatures and elaborating on the methodology of selecting the most relevant ENs for query execution based on the query semantics and the signatures.

Aggregation-Based Outliers Elimination

For detecting multivariate outliers in a dataset D_i, we can rely on widely adopted techniques, which could be categorised to: (i) statistical-based (parametric or non-parametric approaches), (ii) nearest neighbour-based, (iii) clustering-based, (iv) classification-based (Bayesian network-based and support vector machine-based approaches), and (v) spectral decomposition-based approaches. In this work, we focus on the statistical methods that require less computational resources. An aggregation scheme of the χ^2 metric and the Grubb's test [80] for the final outliers outcome is introduced. A data vector $\mathbf{x} \in D_i$ is considered as an outlier if the χ^2-statistic exceeds a specific threshold, which is defined as:

$$\chi^2(\mathbf{x}) = \sum_{k=1}^{d} \frac{(x_k - \mu_k)^2}{\mu_k} \tag{4.14}$$

where μ_k is the k-th mean value dimension of the mean vector $\boldsymbol{\mu}$ of the dataset D_i. According to the central limit theorem, when d is large ($\gg 30$), the χ^2 has approximately a Gaussian distribution [200]. The Grubb's test is also adopted for outlier elimination providing the so-called z-score for data vector $\mathbf{x} \in D_i$ defined as: $z(\mathbf{x}) = \frac{\max(\|\mathbf{x}-\boldsymbol{\mu}\|)}{\max_{k=1,\ldots,d}(\sigma_k)}$. The vector $\mathbf{x} \in D_i$ is considered as an outlier when

$$z(\mathbf{x}) \geq \frac{|D_i|-1}{\sqrt{|D_i|}} \sqrt{\frac{t^2_{\alpha/(2|D_i|),|D_i|-2}}{|D_i|-2+t^2_{\alpha/(2|D_i|),|D_i|-2}}} \tag{4.15}$$

where $t^2_{\alpha/(2|D_i|),|D_i|-2}$ is retrieved by the t-distribution at the significance level of $\alpha/(2|D_i|)$.

This outlier aggregation mechanism is based on a conjunction of the outlier result determined by $\chi^2(\mathbf{x})$ and Grubb $z(\mathbf{x})$. Specifically, let the outlier indicator functions be

$I_\chi(\mathbf{x}) = 1$ and $I_G(\mathbf{x}) = 1$, respectively denoting that $\mathbf{x}$ is an outlier based on the above-mentioned statistics. Then, the vector $\mathbf{x}$ is considered as an outlier in the dataset D_i if $I_\chi(\mathbf{x}) \wedge I_G(\mathbf{x}) = 1$; otherwise the data vector is not an outlier. This means that both methods should agree on the result. If there is a disagreement, $\mathbf{x}$ is not an outlier and is included in the construction of the statistical signature $\mathcal{S}_i$.

Statistical Signature

The statistical signature $\mathcal{S}_i$ of EN i is based on the data vectors in D_i, which are not considered outliers based on the above-mentioned methodology, i.e., $\tilde{D}_i = \{\mathbf{x} \in D_i : I_\chi(\mathbf{x}) \wedge I_G(\mathbf{x}) = 0\}$. The basic statistics of the mean vector $\boldsymbol{\mu}$ and the variances vector $\boldsymbol{\sigma}$ are directly determined and efficiently incrementally updated from the outliers-free dataset $\tilde{D}_i$. They are both used for matching with the query predicates, as will be discussed later. Now, for establishing the minimum sufficient statistics that can reflect the basis of the underlying data, we use the first K_d principal components of the data vectors in $\tilde{D}_i$ that explain the α percentage of the inherent variance (normally $\alpha = 0.9$). Specifically, we seek the eigenbase of the outlier-free data $\tilde{D}_i$ such that given an random data vector $\mathbf{y} \in \mathbb{R}^d$ we can efficiently determine if this vector can be reconstructed (derived from) from the eigenvectors of those data $\mathbf{x} \in \tilde{D}_i$. This is the rationale behind the concept of the signature where we extract the sufficient synopsis of the data deriving the most representative eigenvectors of $\tilde{D}_i$. If the vector $\mathbf{y}$ can be explained by the eigenbase of $\tilde{D}_i$ then we draw the conclusion that $\mathbf{y}$ belongs (can be projected onto) to the subspace of $\tilde{D}_i$. Otherwise, $\mathbf{y}$ is considered statistically irrelevant to $\tilde{D}_i$. In order to come up with this reasoning, we need first to derive the K PCs of $\tilde{D}_i$ by adopting (incremental) PCA over the dataset $\tilde{D}_i$.

In PCA over the $\tilde{D}_i$, we seek the $d \times K$ matrix $\mathbf{W}_i$ of K column (eigen)vectors $\{\mathbf{w}_k\}_{k=1}^K$ that minimizes the objective:

$$\min_{\mathbf{W}_i \in \mathbb{R}^{d \times K} : \mathbf{W}_i^\top \mathbf{W} = \mathbf{I}} \sum_{\mathbf{x}_j \in \tilde{D}_i} \|\mathbf{x}_j - \mathbf{W}_i \mathbf{W}_i^\top \mathbf{x}_j\|^2, \tag{4.16}$$

where $\|\mathbf{x}\|$ is the Euclidean norm of the vector $\mathbf{x}$. That is, we try to find those K PCs in the eigenbase $\mathbf{W}_i$ such that when we project a d-dim vector onto the subspace defined by those PCs, the error of the projection vector $\tilde{\mathbf{x}} = \mathbf{W}_i \mathbf{W}_i^\top \mathbf{x}$ and the actual vector $\mathbf{x}$ are minimised. Hence, we argue that if $\mathbf{x}$ belongs to this subspace then the projection error is the minimum w.r.t. the K PCs. We select the first K PCs, which are ordered by their eigenvalues $\lambda_k, k = 1, \ldots, K$, such that they explain $\alpha(\%)$ of the variances in $\tilde{D}_i$. This is achieved by selecting the first K PCs such that: $K = \min\{K' : \frac{\sum_{k=1}^{K'} \lambda_k}{\sum_{k=1}^{d} \lambda_k} \geq \alpha\}$. Given a *projection error tolerance* $\epsilon > 0$ and the eigenbase $\mathbf{W}_i$ reflecting the sufficient eigenvectors from $\tilde{D}_i$ we infer that a random vector $\mathbf{y} \in \mathbb{R}^d$ belongs to the subspace

defined by the PCs of $\tilde{D}_i$ iff its projection error $\|\mathbf{y} - \mathbf{W}_i\mathbf{W}_i^\top\mathbf{y}\| \leq \epsilon$; otherwise, the vector $\mathbf{y}$ is considered statistically irrelevant (cannot be explained from) to $\tilde{D}_i$, or in other words it is highly unlikely to be observed in EN i. The statistical signature $\mathcal{S}_i = \{\boldsymbol{\mu}_i, \boldsymbol{\sigma}_i, \mathbf{W}_i\}$ is then delivered to the QC. Note, the eigenbase $\mathbf{W}_i$ can be incrementally updated with trivial computational complexity adopting well-known incremental PCA methods. The EN i regularly updates the QC with an updated signature $\mathcal{S}_i' = \{\boldsymbol{\mu}_i', \boldsymbol{\sigma}_i', \mathbf{W}_i'\}$ iff there is a significant difference in $\|\boldsymbol{\mu}_i - \boldsymbol{\mu}_i'\|$, $\|\boldsymbol{\sigma}_i - \boldsymbol{\sigma}_i'\|$ and $\|\mathbf{W}_i - \mathbf{W}_i'\|$; otherwise, there is no meaning for an update.

Signature-based Query Assignment

The QC receives the signatures $\{\mathcal{S}_i\}_{i=1}^N$ from all N ENs in order to reason about the most relevant subset of QPs to engage for each analytics query. This means that, for each query $\mathbf{q}$ a different subset $\mathcal{P}'$ of QPs is determined engaging the corresponding ENs. Given an analytics query $\mathbf{q} = [l_k, u_k]_{k=1}^d \in \mathbb{R}^{2d}$, the QC derives its d-dim centre query vector $\mathbf{y} = [\frac{l_k+u_k}{2}]_{k=1}^d = [y_k]_{k=1}^d \in \mathbb{R}^d$, where each k-th component y_k refers to the centre of k-th range predicate, i.e., $y_k = \frac{l_k+u_k}{2}$. The centre query $\mathbf{y}$ is then projected onto each eigenbase $\mathbf{W}_i$ of EN i in order to judge whether the centre predicate semantics are projected over the data subspace defined by each $\tilde{D}_i$. If the vector $\mathbf{y}$ is approximately considered to belong to the eigenbase $\mathbf{W}_i$ based on the error tolerance $\|\mathbf{y} - \mathbf{W}_i\mathbf{W}_i^\top\mathbf{y}\| \leq \epsilon$, then the associated QP i is a candidate to be engaged for the execution of this query. Otherwise, the dataset D_i is not relevant for providing analytics results for the query $\mathbf{q}$. In the case where the QP i is candidate for query $\mathbf{q}$, we further examine if the underlying data are statistically sufficient (in number) to support the query. This is examined by using the ratio of data dimensions with $y_k \in [\mu_k - \sigma_k, \mu_k + \sigma_k]$, $k = 1, \ldots, d$. If we let $I(y_k) = 1$ if $|y_k - \mu_k| \leq \sigma_k$; otherwise 0, we can then introduce the *degree of engagement* $I(\mathbf{q}, \mathcal{S}_i)$ of an analytics query to a QP i over the dataset D_i of EN i as:

$$I(\mathbf{q}, \mathcal{S}_i) = \begin{cases} 0 & \text{if } \|\mathbf{y} - \mathbf{W}_i\mathbf{W}_i^\top\mathbf{y}\| > \epsilon, \\ 1 + \frac{1}{d}\sum_{k=1}^d I(y_k) & \text{otherwise} \end{cases} \tag{4.17}$$

Hence, given a query $\mathbf{q}$ at the QC, we first derive its centre query vector $\mathbf{y}$ and then for each QP i, we check its corresponding degree of engagement $I(\mathbf{q}, \mathcal{S}_i)$ based on the signature $\mathcal{S}_i$. Then, the QC determines the subset $\mathcal{P}'(\mathbf{q}) \subseteq \mathcal{P}$ of those QPs with $I(\mathbf{q}, \mathcal{S}_i) > 0$, i.e.,

$$\mathcal{P}'(\mathbf{q}) = \{p_i \in \mathcal{P} : I(\mathbf{q}, \mathcal{S}_i) > 0\}. \tag{4.18}$$

The QC assigns the analytics query $\mathbf{q}$ to the QPs belonging in $\mathcal{P}'(\mathbf{q})$ to execute that query over their corresponding ENs.

4.6.3 Experimental Evaluation

This section describes the experimental setup that has been employed to evaluate the proposed mechanism presented in Section 4.6.2. Furthermore, the performance metrics that have been used are presented alongside the comparison baseline – query allocation without the use of our mechanism. The outcome of the performance assessment is discussed at the end of this section.

Setup

To evaluate our approach, we simulated a deployment scenario that involves four ENs, represented by SBC devices, that gather two-dimensional data over time. To make the experiments reproducible, a publicly available dataset has been chosen to simulate the data repositories of the ENs [6]. The dataset is composed of sensor readings that originate from a cluster of SBCs, each one attached to an Unmanned Surface Vehicle (USV). The sensors gather temperature and humidity readings over time at the sea surface, hence, the two dimensional data vectors of our scenario. However, different nodes stop gathering data at different times. To compensate for that inconsistency, we chose the time at which the first USV node stops gathering data to indicate the end of the dataset. In this way, we maintain consistency through all nodes for data gathered before that time.

With the use of Python libraries, like Scikit-Learn [179], NumPy [33], Pandas [151], over the aforementioned data, we simulate the behaviour of QC and the underlying nodes. Various queries and their respective responses are generated for a varying error tolerance values and batch sizes. When the simulation commences, no data is available to the USV nodes. Instead, data is parsed gradually according to typical node behaviour. Once a predefined amount of data is available in an EN/USV, the method that builds the statistical signature for this last amount of data is triggered and builds the relevant statistical signature $\mathcal{S}_i$ as defined in Equation 4.13. All USV/nodes transmit their latest statistical signatures to the QC. In turn, the QC receives a four-dimensional (two temperature values and two humidity values) analytics query randomly generated according to a uniform distribution which ranges between $[0, 40]$ for humidity and $[0, 60]$ for temperature. The QC, based on the initial query $\mathbf{q}_k$, calculates the two-dimensional centre query vector $\mathbf{y}_k$. Decisions on which nodes will accommodate the query are based on the mechanism described in Section 4.6.2. The error tolerance ϵ defined in Equation 4.17 is pre-determined to a certain value for each run of the experiment.

Performance Metrics

After queries are allocated to QPs, the responses are based on the available contextual data on each node. Prior to using the proposed method, the query would be allocated

to *every* QP in the network, i.e., the baseline solution engaging *all* USVs. By using the proposed query allocation and execution mechanism, the amount of nodes that receive the query is expected to decrease because of the statistical irrelevance of their data. As a result, the data that will formulate the response to the query will be fewer and more *relevant* to the query, which is our goal. In order to evaluate this rationale, we calculate the total variance of the data existing in nodes after a query allocation: $V'_{total} = \sum_{i=1}^{N'} \sigma_i$, for $N' = |\mathcal{P}'(\mathbf{q})|$ and compare it against the total variance of data across *all* nodes (prior to applying the proposed mechanism) $V_{total} = \sum_{i=1}^{N} \sigma_i$, for $N = |\mathcal{P}|$, where $|\mathcal{P}|$ is the cardinality of set $\mathcal{P}$.

After the query allocation phase is complete, the nodes that have been found to hold *relevant data for the issued queries* are marked in order to examine the total node-involvement ratio per query. The *involvement ratio* is defined as the number of nodes that were identified to hold relevant data over the total number of nodes $r = \frac{N'}{N}$, with $0 \leq N' \leq N$. Such involvement ratio provides a high-level metric of the resources that are utilised per query execution. By utilising a portion of the nodes and thus having a small involvement ratio, the message exchanges are reduced and fewer query processing transactions are required. Maintaining a small involvement ratio r while accommodating the needs of queries translates to more efficient use of the available resources: this is an important aspect when considering scenarios involving resource-constrained edge nodes, like the UxV environments.

Performance Assessment

A comparison baseline is set by examining the the naive query allocation mechanism. In that case, the queries are allocated to every QP available in the edge network. Naive allocation is selected as the default mechanism for query allocation in IoT. Comparison with existing methods that have similar goals (e.g., Dragon's scheme [112], Huarcapuma et al. [37]) is not straightforward due to subtle differences in data modeling and processing. It is therefore left for future work. After conducting experiments for this use case and obtain the aforementioned metrics, we repeat the experiments with the query allocation mechanism enabled. To ensure convergence of the obtained results, we conducted repeated experiments and used the average of the outcome. The query allocation mechanism was executed 1000 times over the previously described setup. The error tolerance was fixed, $\epsilon = 40$ for the execution of the experiments. Since the data are gathered in batches before a statistical signature is built, to ensure synchronisation between the queries we fixed the size of the batches to 40 rows, resulting in statistical signatures for data of 40 rows and 2 columns.

The results in Fig. 4.24 shows that the V'_{total} values obtained through the use of the query allocation mechanism are significantly lower when compared with the values

occurring when the mechanism is not in use. The observed variance reduction fluctuates between $30\% - 60\%$, depending on the query. The reduced variance indicates a *higher quality of the query response* as it will be based on data with smaller variance not including irrelevant contextual information.

Average involvement ratio r			
	Batch size		
tolerance ϵ	30	40	50
20	0.225	0.21667	0.20834
30	0.425	0.38334	0.4375
40	0.6625	0.58334	0.47916
50	0.7625	0.88334	0.85416
60	0.9875	1	0.875

Table 4.3: Impact of error tolerance ϵ on expected involvement ratio r.

In order to examine the relation of the reduced variance and the involvement ratio, we compared the two metrics in Fig. 4.25. The plot shows that variance reduction is, for most of the time, inversely proportional to the involvement ratio. This shows that nodes are only involved when they hold relevant information to the query. This increases the quality of the response by excluding irrelevant information (and nodes). By using statistical signatures to describe the underlying data, irrelevant nodes are filtered. The statistical relevance of the data stored on each node is examined *before* assigning a query, and therefore our mechanism avoids the use of unnecessary resources (nodes and links).

In Table 4.3, we included a summary of measurements of the average involvement ratio for different error tolerance values and batch sizes. Our aim is to determine the impact that the error tolerance ϵ has on the involvement ratio. We found that regardless of the batch size, increasing values of the error tolerance lead to higher average involvement ratio as more and more nodes are found to be within this tolerance threshold and therefore are considered relevant to the query. By adjusting the value of the error tolerance, the mechanism can be tuned to provide strict correlation between the query and the signatures (low tolerance) or more relaxed correlation (higher tolerance).

4.7 Summary

The findings of the previous chapter motivated an attempt to extend in-network compute for edge infrastructure and explore the challenges of deploying virtualised computations at resource-constrained environments. This work studies the computational and energy impact of deploying a mainstream orchestration platform at the edge by using an energy constrained IoT cluster as an indicative deployment environment. A virtualised application is built to measure energy consumption using wireless sensors. The gathered data is studied

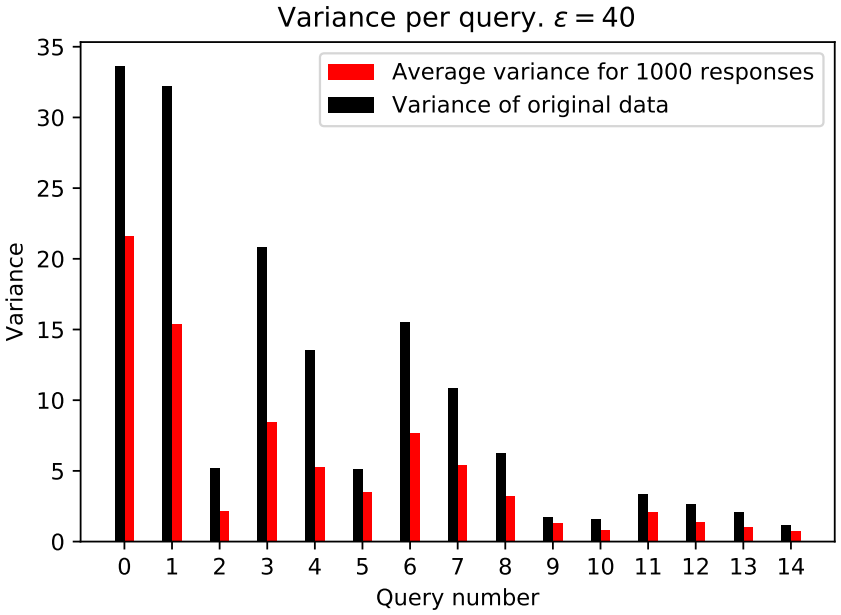


Figure 4.24: Comparison between the variance that a query is exposed to w.r.t. the baseline solution and our proposed mechanism.

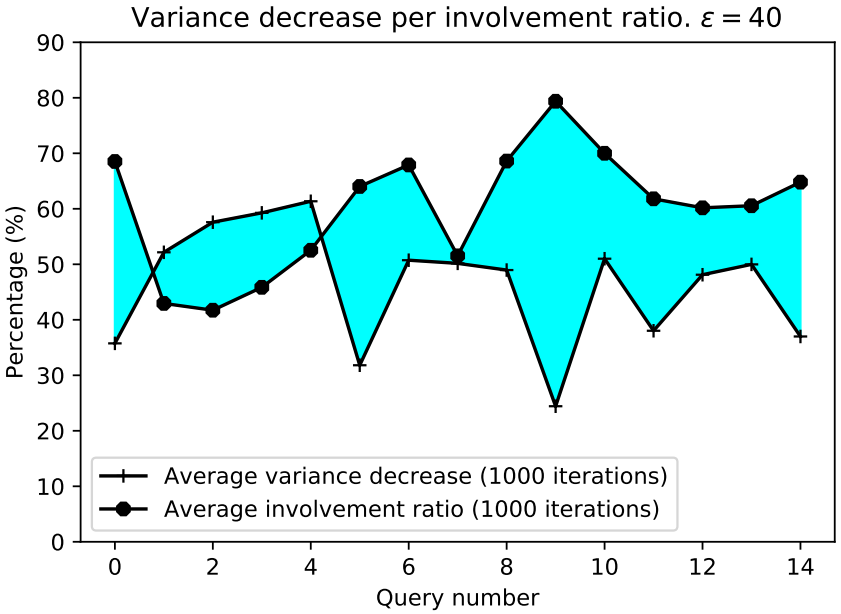


Figure 4.25: The involvement ratio r is compared to variance decrease that occurs from our query allocation mechanism.

to discover the energy impact of the different roles of nodes within the cluster (worker or master) and how battery consumption changes based on different computations. The findings show that the master node consumes more energy on average, due to monitoring and networking processes running constantly to ensure the cluster is alive and healthy. Overall, virtualised services posed insignificant overheads in energy consumption (5% - 8%) compared to idle power draw, suggesting that using virtualised services at the edge is attainable under adequate energy provisioning.

The results of deploying the orchestration framework and the app revealed an imbalanced but overall modest energy consumption, which motivated the conception of a master node that is able to integrate real-time energy measurements in the scheduling process with the aim of reducing imbalances and minimise rejections. It is apparent that the inability of the default master node to allocate VNFs to IoT nodes based on the required energy resources makes the cluster underperform by evicting VNFs before their completion and failing to distribute the available resources equally. A series of design adaptations were made to introduce real-time measurements within the scheduling process and predict the energy usage of various requests based on their requirements in processing and network resources. The resources of the master node were allocated to the general pool of cluster resources to increase the total processing and energy capacity of the cluster. This was done for workloads with low processing requirements that will not corrupt the master node functionality. The proposed energy-aware scheduler, ECAS, outperformed the default Kubernetes scheduler and the greedy LLS algorithm in a series of tests. It minimised resource contention at the cluster nodes which was also reflected in evenly-balanced energy consumption. By matching energy requirements to the available capacity, it minimised event rejections and average deadline violations.

The use of a cloud orchestrator at the network edge revealed that cloud-edge convergence can be achieved in a technical level with the use of interoperable and extensible software. Cloud-edge convergence; however, involves distributing information between edge and cloud, which, considering the vast data gathering of sensor data at the edge, becomes a demanding task. The last part of this chapter proposes a mechanism that relies on compute nodes in the cloud to hold representations of data existing at the edge. These representations are formulated using statistical signatures generated at the edge, which are subsequently stored and updated at the cloud gateways. Using these representations, queries generated in the cloud can be directed at edge nodes with relevant data in order to be processed. The mechanism is capable of providing responses with decreased variance over naive query allocation. In doing so, it involved fewer nodes for generating a response, therefore minimising computation and energy usage. The results serve as an indicator of how cloud gateways, or similarly-located programmable networking hardware, can be used to facilitate cloud-edge communication.

Chapter 5

Conclusion

5.1 Overview

This chapter revisits the thesis statement and the contributions described in Chapter 1 with the aim of explaining how the conducted work validates these claims. It also discusses future research directions of the presented work.

5.2 Contributions

The work conducted in this thesis is motivated by a study of the literature that spans across various disciplines, technologies, and methods. This thesis contributes by analysing and critically discussing the findings of this study in Chapter 2. The main focus is on network programmability tools that allow service offloading in both cloud and edge infrastructure. The study is inclusive of tools operating in commodity hardware (e.g., user-space packet processing, kernel space), high port-count devices like programmable switches, and even end-host interfaces like smartNICs. It then proceeds to analyse the potential and limitations of modern service provisioning tools, like NFV, and the integration of services in the network fabric with the use of P4. The most prominent use cases are presented for both cloud and edge environments.

Having discussed the importance of distributed data replication for the orchestration of services within the network, Chapter 3 of this thesis contributes by analysing previous efforts for service deployment in PDP (e.g., NetChain, NetCache, IncBricks). The fastest available platform, NetChain, is presented in depth with the goal of identifying the main design aspects that bottleneck its performance. A new data replication platform is proposed that utilises the previous lessons from the deployment of NetChain and the performance benefits of deploying this service in PDP. The platform is able to achieve great performance improvements over NetChain: close to linear scalability for read-mostly workloads and higher throughput across all read/write ratios over NetChain. The plat-

form's design supports easy integration with other data centre applications by providing simple query-response messages over UDP. The offloading of Kubernetes' KVS workload is examined as a potential use case due to its high percentage in read queries over write queries and bursty traffic during deployment of services.

With service provisioning expanding closer and closer to the end user, this thesis contributes by reporting findings on edge IoT infrastructure as the host of virtualised services and the benefits of synergy between cloud gateways and IoT clusters. In Chapter 4, the feasibility of managing IoT clusters of SBC devices using Kubernetes is assessed and measurements on the energy impact are gathered. The findings reveal skewed energy consumption occurring from the operation of the master node and imbalances on the allocation of resources. This work contributes by adapting the design of Kubernetes' scheduler to integrate real-time energy measurements, used to predict the remaining capacity alongside resource utilisation after deploying services to nodes. This work achieves this and contributes by presenting a scheduler integrated in Kubernetes that is able to minimise consumption imbalances and resource contention. Therefore, the scheduler maintains compatibility with cloud by relying on the same orchestration software while improving its operation at the edge by adapting it to local constraints.

Driven by the ability to use programmable networking at the edge, either as small form-factor programmable switches or commodity hardware, and simultaneously use a common orchestration platform, Chapter 4 of this thesis contributes by combining edge and cloud infrastructure to improve the allocation of queries to nodes. More specifically, this work proposes a mechanism that generates statistical signatures of data stored at the edge and stores them in cloud gateways. Using this mechanism, queries arriving at the cloud can be directed to nodes based on the statistical relevance between them. This mechanism provides selective invocation of nodes by only directing queries to statistically relevant nodes, hereby improving energy consumption by minimising redundant operations and reducing the average variance in responses to queries.

The work conducted in Chapters 3, 4, appears to be interconnected in many regards. Data storage within the network, either in data centre or the edge, and efficient data retrieval appear to be a common denominator of the proposed frameworks. Furthermore, this thesis formulates proposals by relying on the paradigms of SDN and NFV, that span across both edge and cloud infrastructure and make service deployment and management possible. The various proposals that are made either rely or extend these paradigms. These are considered to be integral elements to achieve edge-cloud continuum and the delivery of end-to-end service deployments.

5.3 Thesis Statement Revisited

The thesis statement of Section 1.2 is restated here in order to be revisited afterwards:

"This work considers the capability of modern network hardware to perform per-packet stateful processing at line rates as a potential accelerator of computations within the network fabric. It asserts that, through stateful packet processing, data replication services can be implemented within the network in a scalable manner that outperforms legacy data replication methods. This work identifies limitations of previous in-network replication platforms and proposes design changes that offer better scalability, throughput and latency over the previous state of the art, without harming consistency or fault-tolerance. Because of the central role of Key-Value Stores as a coordination platform for widely deployed controllers and orchestrators and their workload characteristics, offloading this functionality in programmable hardware is proposed to reduce reaction times to network events and promote scalable orchestration.

This work also recognises the proliferation of virtualised network functions, able to formulate advanced network services in both data centres and edge infrastructure. It asserts that Edge IoT infrastructure can host mainstream orchestration platforms despite the existing resource constraints, further reducing latency at the end user by placing computation in low proximity. It proceeds to examine the energy impact of running an orchestration platform in IoT devices and obtains energy profiles for different roles and workloads within an IoT cluster. This work affirms that energy-aware scheduling can be implemented at a cluster of edge devices, using real time sensor readings, offering efficient use of the available resources. Finally, it asserts that statistical matching between queries arriving at the cloud and data existing at the edge can reduce excess computations by minimising data transfers and using only nodes storing statistically relevant data."

The work presented in Chapter 3 expands on the feasibility of performing data replication in PDP by manipulating the SRAM registers available in programmable switches. In a wider context, implementing in-network replication means repurposing the hardware that enables stateful packet processing to store values and update them consistently across multiple programmable switches. This thesis presents the previous state of the art which, by adopting the design aspects that enabled the generation of query responses entirely in PDP and addressing limitations that impact performance and scalability, is further improved. The evaluation of the proposed platform revealed great performance improvements without harming consistency or fault tolerance. However, a potential downside is that the total number of required registers for the implementation of the proposed method is higher in total compared to previous state of the art. Kubernetes is examined as a potential use case of the proposed platform. With its workload comprised of read-mostly queries concerning a limited number of KV pairs, it conforms to the constraints of modern programmable switches and presents great potential for accelerating the deployment times

of services and its reaction to network events.

Chapter 4 expands on the topic of VNF offloading at the edge. It experiments with the deployment of Kubernetes in a cluster of representative IoT devices (Raspberry Pis). It is shown that an IoT cluster is capable of hosting virtualised services in the form of Kubernetes pods and orchestrate them effectively, despite the constraints of the environment. The aspect of energy consumption is also examined, as a way to examine the feasibility of such deployments under more realistic constraints. Energy measurements are obtained, showing that the infrastructure can support this deployment using moderate energy resources, albeit with consumption imbalances. This is addressed in the later part of this work, which implements an energy-aware scheduler that is able to minimise these imbalances and improve the usage of the available resources by demonstrating fewer event rejections and deadline violations. Lastly, this thesis proposes a mechanism that allows data existing at the edge to be represented at the cloud. Using PCA, statistical signatures of the data are built and subsequently stored at cloud gateways, which are able to use them and direct queries to statistically relevant nodes. The mechanism has been evaluated using real-world sensor data and has demonstrated reduced allocation of queries to nodes holding irrelevant data. Therefore, only the statistical signature of the data is necessary for query allocation which translates to fewer transactions compared to transferring all of the data existing at the edge. It also means that nodes can remain inactive as long as queries are irrelevant, further decreasing the amount of redundant processing cycles.

5.4 Future Research Directions

Each work segment of this thesis can be extended individually or in conjunction with the rest of them to produce novel research since many of the problems presented in this thesis are still open-ended and gather attention from both academia and industry. Starting with the topic of in-network data replication, the presented work could be extended by evaluating various Kubernetes deployments with: 1. the use of in-network KV replication instead of etcd; 2. hybrid use of in-network replication and etcd (with the latter storing less frequently accessed pairs). Another future research direction is the integration of smartNICs in the chain of participating nodes for the acceleration of KVS transactions close to the end-user, as explained in Section 3.7. In attempting to do so, various optimisation problems are created. For example, how different workloads impact the duration that a KV pair should remain cached at the end-host NIC? What are some general types of use cases that can benefit the most from this type of data replication? Can the management of KV pairs be done dynamically to be optimal for each workload based on real-time measurements?

In the broader topic of network programmability, which is anticipated to be an active

area of research for the following years, P4 has successfully become a language that allows the definition of packet processing pipelines, regardless of the target hardware. This pushes P4 towards becoming a reference language for the definition of protocols and distributed applications. Although, closed hardware architectures currently make development tied to a vendor's provision of board support packages and software development kits, including compilers. This creates inertia in the development and evaluation of new use cases from the networking community. It also causes ideas to be bounded around hardware limitations, harming creativity and resulting in a narrow research scope.

I remain optimistic that future hardware releases and community efforts will address these problems, enabling the use of a common programming language across different architectures and vendors with minimal adjustment of the source code. Therefore, the area of network programmability will continue to grow with efforts directed towards interoperable languages for the data plane. This will fuel research in the programmability of multiple networking devices under a common framework, able to synthesise the end binary using all the available network resources. Eventually, I hope to see the network infrastructure conceptualised as a unified distributed system that is end-to-end programmable under a common framework. A diverse infrastructure, able to utilise heterogeneous hardware, e.g., GPUs, FPGAs, ASICs, to accelerate different types of computations based on each device's characteristics would present great potential performance improvements. Research efforts towards chain synthesis, hybrid placement of network computations across both commodity and networking hardware, real-time reprogramming of data plane hardware, will greatly support the next generations of networking. It is apparent that in-network data replication, by offering line-rate all-to-all communication, will have an instrumental role in the programmability of such an infrastructure. An in-network KVS can supply all of the necessary configuration parameters across the infrastructure, e.g., security keys, link health states, congestion metrics etc. Therefore, in-network replication has the potential of being a key enabler for large-scale configuration of next-generation networks.

The work conducted in the domain of energy-aware scheduling can be further evaluated over infrastructure that combines both cloud and edge devices. The scheduler can be adapted to take into account energy measurements of data centre devices, resulting in novel optimisation problems related to placement decisions, energy usage prediction of a more diverse set of requests, interconnected requests between edge and cloud, etc. By including infrastructure with higher energy consumption and therefore varying energy data, it would be beneficial to associate them with carbon emissions and create a reference metric on how placement decisions impact the overall emissions of the infrastructure. This can also be integrated in the process of VNF scheduling to allow trade-offs between performance and emissions to be assessed and integrated in the configuration of the infrastructure. As networks grow and VNF deployment becomes more standardised, such adjustments to the

way networks are configured can have an impactful change in the energy footprint of the infrastructure.

5.5 Concluding Remarks

This work has enhanced the previous state of the art by improving the design of use cases in the area of in-network compute. It has improved the management of resource-constrained edge infrastructure, with the aim of best utilising resources located close to the end user for service deployment. It produced work on cloud-edge integration that improved data representation across the infrastructure. By utilising the epitome of current technologies in the area, it has delivered novel research ideas on the respective topics and motivated future research directions.

References

[1] Sura Khalil Abd, Syed Abdul Rahman Al-Haddad, Fazirulhisyam Hashim, Azizol BHJ Abdullah, and Salman Yussof. An effective approach for managing power consumption in cloud computing infrastructure. *journal of computational science*, 21:349–360, 2017.

[2] Suhail Ahmad and Ajaz Hussain Mir. Scalability, consistency, reliability and security in sdn controllers: a survey of diverse sdn controllers. *Journal of Network and Systems Management*, 29(1):1–59, 2021.

[3] Noura Al-Hoqani, Shuang-Hua Yang, Daniel P. Fiadzeawu, and Ross J. Mcquillan. In-network on-demand query-based sensing system for wireless sensor networks. In *2017 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, 2017.

[4] Abeer Ali, Christos Anagnostopoulos, and Dimitrios P. Pezaros. Resource-aware placement of softwarised security services in cloud data centers. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–5, 2017.

[5] Christos Anagnostopoulos. Quality-optimized predictive analytics. *Appl. Intell.*, 45(4):1034–1046, 2016.

[6] Dr Christos Anagnostopoulos. Gnfuv dataset. `https://archive.ics.uci.edu/ml/datasets/GNFUV+Unmanned+Surface+Vehicles+Sensor+Data+Set+2`. Accessed: 2022-01-29.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Prometheus Authors. Prometheus. `https://prometheus.io/`. Accessed: 2022-05-30.

[9] The Kubernetes Authors. Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/`. Accessed: 2022-05-30.

[10] The Kubernetes Authors. Kubernetes resource management. `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/`. Accessed: 2022-01-29.

[11] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013.

[12] Cagri Balkesen and Nesime Tatbul. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *International workshop on data management for sensor networks (DMSN)*, 2011.

[13] Fetia Bannour, Sami Souihi, and Abdelhamid Mellouk. Distributed sdn control: Survey, taxonomy, and challenges. *IEEE Communications Surveys & Tutorials*, 20(1):333–354, 2018.

[14] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, page 5–16, USA, 2015. IEEE Computer Society.

[15] Philip J Basford, Steven J Johnston, Colin S Perkins, Tony Garnock-Jones, Fung Po Tso, Dimitrios Pezaros, Robert D Mullins, Eiko Yoneki, Jeremy Singer, and Simon J Cox. Performance analysis of single board computer clusters. *Future Generation Computer Systems*, 102:278–291, 2020.

[16] Cataldo Basile, Christian Pitscheider, Fulvio Risso, Fulvio Valenza, and Marco Vallini. Towards the dynamic provision of virtualized security services. In Frances Cleary and Massimo Felici, editors, *Cyber Security and Privacy*, pages 65–76, Cham, 2015. Springer International Publishing.

[17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[18] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for*

Computer Communication, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Leonardo Bonati, Michele Polese, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. Open, programmable, and virtualized 5g networks: State-of-the-art and the road ahead. *Computer Networks*, 182:107516, 2020.

[20] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[22] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernado Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.

[23] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[24] Broadcom. Broadcom Trident4. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series`. Accessed: 2022-01-29.

[25] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, June 2013. USENIX Association.

[26] Lei Cao and Elke A. Rundensteiner. High performance stream query processing with correlation-aware partitioning. *Proc. VLDB Endow.*, 7(4):265–276, 12 2013.

[27] Cavium. Cavium liquidio. `https://web.archive.org/web/20170709170234/http://www.cavium.com/LiquidIO_Server_Adapters.html`. Accessed: 2022-10-18.

[28] Danilo Cerović, Valentin Del Piccolo, Ahmed Amamou, Kamel Haddadou, and Guy Pujolle. Fast packet processing: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):3645–3676, 2018.

[29] Yang Chen and Jie Wu. Nfv middlebox placement with balanced set-up cost and bandwidth consumption. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Nan Cheng, Ning Lu, Ning Zhang, Tingting Yang, Xuemin Shen, and Jon W. Mark. Vehicle-assisted device-to-device data delivery for smart grid. *IEEE Transactions on Vehicular Technology*, 65(4):2325–2340, 2016.

[31] Aakanksha Chowdhery, Marco Levorato, Igor Burago, and Sabur Baidya. *Urban IoT Edge Analytics*, pages 101–120. Springer International Publishing, Cham, 2018.

[32] S. R. Chowdhury, M. A. Salahuddin, N. Limam, and R. Boutaba. Re-architecting nfv ecosystem with microservices: State of the art and research challenges. *IEEE Network*, 33(3):168–176, 2019.

[33] NumPy Community. Numpy. `https://numpy.org/`. Accessed: 2022-01-29.

[34] Ryu SDN Framework Community. Ryu controller. `https://ryu-sdn.org/index.html`, 2014. Accessed: 2022-01-29.

[35] Behavioral Model Contributors. Bmv2. `https://github.com/p4lang/behavioral-model`. Accessed: 2022-01-29.

[36] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.

[37] Ruben Cruz Huacarpuma, Rafael Timoteo De Sousa Junior, Maristela Terto De Holanda, Robson De Oliveira Albuquerque, Luis Javier García Villalba, and Tai-Hoon Kim. Distributed data service for data management in internet of things middleware. *Sensors*, 17(5), 2017.

[38] R. Cziva and D. P. Pezaros. Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 6 2017.

[39] Richard Cziva, Christos Anagnostopoulos, and Dimitrios P. Pezaros. Dynamic, latency-optimal vnf placement at the network edge. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 693–701, 2018.

[40] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 202–215, New York, NY, USA, 2001. Association for Computing Machinery.

[41] Rajdeep Das and Alex C. Snoeren. Enabling active networking on rmt hardware. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 175–181, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on nosql stores. *ACM Comput. Surv.*, 51(2), 4 2018.

[43] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[44] Sedef Demirci and Seref Sagiroglu. Optimal placement of virtual network functions in software defined networks: A survey. *Journal of Network and Computer Applications*, 147:102424, 2019.

[45] Zhongwei Deng, Xiaosong Hu, Xianke Lin, Yunhong Che, Le Xu, and Wenchao Guo. Data-driven state of charge estimation for lithium-ion battery packs based on gaussian process regression. *Energy*, 205:118000, 2020.

[46] Ousmane Diallo, Joel J.P.C. Rodrigues, and Mbaye Sene. Real-time data management on wireless sensor networks: A survey. *Journal of Network and Computer Applications*, 35(3):1013–1021, 2012. Special Issue on Trusted Computing and Communications.

[47] Ding Ding, Xiaocong Fan, Yihuan Zhao, Kaixuan Kang, Qian Yin, and Jing Zeng. Q-learning based dynamic task scheduling for energy-efficient cloud computing. *Future Generation Computer Systems*, 108:361–371, 2020.

[48] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 15–28, New York, NY, USA, 2009. Association for Computing Machinery.

[49] Docker. Use containers to build, share and run your applications. `https://www.docker.com/resources/what-container`. Accessed: 2022-10-29.

[50] Rob Enns. NETCONF Configuration Protocol. RFC 4741, December 2006.

[51] etcd Authors. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. `https://etcd.io/`. Accessed: 2022-01-29.

[52] Facebook. Katran. `https://github.com/facebookincubator/katran#readme`. Accessed: 2022-10-18.

[53] Ludwig Fahrmeir, Thomas Kneib, Stefan Lang, and Brian Marx. Regression models. In *Regression*, pages 21–72. Springer, 2013.

[54] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The case for separating routing from routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, page 5–12, New York, NY, USA, 2004. Association for Computing Machinery.

[55] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.

[56] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. A year in lockdown: How the waves of covid-19 impact internet traffic. *Commun. ACM*, 64(7):101–108, 6 2021.

[57] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98. USENIX Association, April 2021.

[58] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann de Meer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[59] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[60] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine*, 55(5):94–100, 5 2017.

[61] Apache Software Foundation. Thrift api. `https://thrift.apache.org/docs/`. Accessed: 2022-01-29.

[62] Open Information Security Foundation. Suricata. `https://suricata.readthedocs.io/en/latest/index.html`. Accessed: 2022-10-18.

[63] Open Networking Foundation. P4_16. `https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html`. Accessed: 2022-01-29.

[64] OpenInfra Foundation. Opentack. `https://www.https://www.openstack.org/`. Accessed: 2022-01-29.

[65] Raspberry Pi Foundation. Raspberry Pi 3. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`. Accessed: 2018-10-20.

[66] Raspberry Pi Foundation. Raspberry pi 4. `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/`. Accessed: 2022-10-18.

[67] The Linux Foundation. Onap. `https://www.onap.org/`. Accessed: 2022-01-29.

[68] Moshe Gabel, Daniel Keren, and Assaf Schuster. Monitoring least squares models of distributed streams. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 319–328, New York, NY, USA, 2015. Association for Computing Machinery.

[69] Pegah Gazori, Dadmehr Rahbari, and Mohsen Nickray. Saving time and cost on the scheduling of fog-based iot applications using deep reinforcement learning approach. *Future Generation Computer Systems*, 110:1098–1115, 2020.

[70] Bugra Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDB J.*, 23(4):517–539, 2014.

[71] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet: Networks without effective aqm may again be vulnerable to congestion collapse. *Queue*, 9(11):40–54, 11 2011.

[72] Petros Gigis, Matt Calder, Lefteris Manassakis, George Nomikos, Vasileios Kotronis, Xenofontas Dimitropoulos, Ethan Katz-Bassett, and Georgios Smaragdakis. Seven years in the life of hypergiants' off-nets. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 516–533, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Juliver Gil Herrera and Juan Felipe Botero. Resource allocation in nfv: A comprehensive survey. *IEEE Transactions on Network and Service Management*, 13(3):518–532, 2016.

[74] Morteza Golkarifard, Carla Fabiana Chiasserini, Francesco Malandrino, and Ali Movaghar. Dynamic vnf placement, resource allocation and traffic routing in 5g. *Computer Networks*, 188:107830, 2021.

[75] Nithyashri Govindarajan, Yogesh Simmhan, Nitin Jamadagni, and Prasant Misra. Event processing across edge and the cloud for internet of things applications. In Srikanta Bedathur, Divesh Srivastava, and Satyanarayana R. Valluri, editors, *20th International Conference on Management of Data, COMAD 2014, Hyderabad, India, December 17-19, 2014*, pages 101–104. Computer Society of India, 2014.

[76] ETH Networked Systems Group. P4-utils. `https://github.com/nsg-ethz/p4-utils`. Accessed: 2022-01-29.

[77] The P4.org API Working Group. P4runtime specification. `https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html`. Accessed: 2022-05-30.

[78] The P4.org Architecture Working Group. Portable switch architecture. `https://p4lang.github.io/p4-spec/docs/PSA-v1.1.0.html`. Accessed: 2022-01-29.

[79] Lav Gupta, M Samaka, Raj Jain, Aiman Erbad, Deval Bhamare, and Chris Metz. Colap: A predictive framework for service function chain placement in a multi-cloud environment. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–9, 2017.

[80] Jiawei Han, Micheline Kamber, and Jian Pei. 3 - data preprocessing. In Jiawei Han, Micheline Kamber, and Jian Pei, editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 83–124. Morgan Kaufmann, Boston, third edition edition, 2012.

[81] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 195–206, New York, NY, USA, 2010. Association for Computing Machinery.

[82] Hasanin Harkous, Michael Jarschel, Mu He, Rastin Pries, and Wolfgang Kellerer. P8: P4 with predictable packet processing performance. *IEEE Transactions on Network and Service Management*, pages 1–1, 2020.

[83] Natascha Harth and Christos Anagnostopoulos. Quality-aware aggregation & predictive analytics at the edge. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 17–26, 2017.

[84] Natascha Harth and Christos Anagnostopoulos. Edge-centric efficient regression analytics. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 93–100, 2018.

[85] H. Hashim, J.A. Manan, and M. Samad. Active network implementations. In *Student Conference on Research and Development*, pages 371–374, 2002.

[86] Pat Helland. Life beyond distributed transactions: An apostate's opinion. *Queue*, 14(5):69–98, 10 2016.

[87] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Liang Hu, Rui Sun, Feng Wang, Xiuhong Fei, and Kuo Zhao. A stream processing system for multisource heterogeneous sensor data. *J. Sensors*, 2016:4287834:1–4287834:8, 2016.

[89] Xiaosong Hu, Shengbo Li, Huei Peng, and Fengchun Sun. Robustness analysis of state-of-charge estimation methods for two types of li-ion batteries. *Journal of power sources*, 217:209–219, 2012.

[90] F Huet. A review of impedance measurements for determination of the state-of-charge or state-of-health of secondary batteries. *Journal of power sources*, 70(1):59–69, 1998.

[91] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[92] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4->netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 1–9, New York, NY, USA, 2019. Association for Computing Machinery.

[93] Intel. Intel Tofino. `https://www.intel.co.uk/content/www/uk/en/products/network-io/programmable-ethernet-switch/tofino-series.html`. Accessed: 2022-01-29.

[94] Intel. Intel x520. `https://www.intel.co.uk/content/www/uk/en/products/docs/network-io/ethernet/network-adapters/server-adapter-x520-da1-da2-for-ocp-brief.html`. Accessed: 2022-10-18.

[95] Mircea M. Iordache-Sica, Christos Anagnostopoulos, and Dimitrios P. Pezaros. Towards qos-aware provisioning of chained virtual security services in edge networks. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 178–186, 2021.

[96] Koch Jason, Spier Martin, Gregg Brendan, and Hunter Ed. Extending vector with ebpf to inspect host and container performance. `https://netflixtechblog.com/extending-vector-with-ebpf-to-inspect-host-and-container-performance-5da3af4c584b`. Accessed: 2022-10-18.

[97] Schulist Jay, Borkmann Daniel, and Starovoitov Alexei. Linux socket filtering aka berkeley packet filter (bpf). `https://www.kernel.org/doc/Documentation/networking/filter.txt`. Accessed: 2022-10-18.

[98] Andrew Jeffery, Heidi Howard, and Richard Mortier. Rearchitecting kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 7–12, New York, NY, USA, 2021. Association for Computing Machinery.

[99] Panpan Jin, Xincai Fei, Qixia Zhang, Fangming Liu, and Bo Li. Latency-aware vnf chain deployment with efficient resource reuse at network edge. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 267–276, 2020.

[100] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[101] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[102] Ren Jinglei, Kjellqvist Chris, and Deng Long. Ycsb. `https://github.com/basicthinker/YCSB-C`. Accessed: 2022-01-29.

[103] Steven J Johnston, Philip J Basford, Colin S Perkins, Herry Herry, Fung Po Tso, Dimitrios Pezaros, Robert D Mullins, Eiko Yoneki, Simon J Cox, and Jeremy Singer. Commodity single board computer clusters and their applications. *Future Generation Computer Systems*, 89:201–212, 2018.

[104] A. Kaloxylos. A survey and an analysis of network slicing in 5g networks. *IEEE Communications Standards Magazine*, 2(1):60–65, 3 2018.

[105] Goutham Kamath, Pavan Agnihotri, Maria Valero, Krishanu Sarker, and Wen-Zhan Song. Pushing analytics to the edge. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2016.

[106] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[107] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 67–81, New York, NY, USA, 2016. Association for Computing Machinery.

[108] Irene Keramidi, Panagiotis Kardaras, Ioannis Moscholios, Panagiotis Sarigiannidis, and Michael Logothetis. A study on the impact of service time distributions in a vehicular ad hoc network. In *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, pages 407–412, 2021.

[109] Irene Keramidi, Dimitris Uzunidis, Marinos Vlasakis, Panagiotis G. Sarigiannidis, and Ioannis Moscholios. Exploiting machine learning for the performance analysis of a mobile hotspot with a call admission control mechanism. In *2022 IEEE 27th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 77–82, 2022.

[110] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155, 2021.

[111] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 8 2000.

[112] Roman Kolcun and Julie A. McCann. Dragon: Data discovery and collection architecture for distributed iot. In *2014 International Conference on the Internet of Things (IOT)*, pages 91–96, 2014.

[113] Kostas Kolomvatsos. An intelligent scheme for assigning queries. *Appl. Intell.*, 48(9):2730–2745, 2018.

[114] Kostas Kolomvatsos and Christos Anagnostopoulos. Reinforcement learning for predictive analytics in smart cities. *Informatics*, 4(3), 2017.

[115] Kostas Kolomvatsos and Stathes Hadjiefthymiades. Learning the engagement of query processors for intelligent analytics. *Appl. Intell.*, 46(1):96–112, 2017.

[116] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[117] Ayaka Koshibe. Onos system components. `https://wiki.onosproject.org/display/ONOS/System+Components`. Accessed: 2022-01-29.

[118] Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, and Ming-Jer Tsai. Deploying chains of virtual network functions: On the relation between link and server usage. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[119] TV Lakshman, T Nandagopal, Ramachandran Ramjee, K Sabnani, and T Woo. The softrouter architecture. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*, 2004.

[120] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[121] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[122] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized? state distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, page 1–6, New York, NY, USA, 2012. Association for Computing Machinery.

[123] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 279–291, 2019.

[124] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 279–291, 2019.

[125] Jong Chern Lim and Chris Bleakley. Adaptive wsn scheduling for lifetime extension in environmental monitoring applications. *International Journal of Distributed Sensor Networks*, 8(1):286981, 2011.

[126] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 795–809, New York, NY, USA, 2017. Association for Computing Machinery.

[127] Fantasia Trading LLC. Anker Astro E1. `https://web.archive.org/web/20210306110603/https://www.anker.com/products/variant/astro-e1/A1211012`. Accessed: 2022-12-20.

[128] Alejandro Llorens-Carrodeguas, Stefanos G. Sagkriotis, Cristina Cervelló-Pastor, and Dimitrios P. Pezaros. An energy-friendly scheduler for edge computing systems. *Sensors*, 21(21), 2021.

[129] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 160–161, 2007.

[130] Redis Ltd. Redis. `https://redis.io/`. Accessed: 2022-01-29.

[131] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. Piecing together the nfv

provisioning puzzle: Efficient placement and chaining of virtual network functions. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 98–106, 2015.

[132] Wenrui Ma, Oscar Sandoval, Jonathan Beltran, Deng Pan, and Niki Pissinou. Traffic aware placement of interdependent nfv middleboxes. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[133] André Luiz R. Madureira, Francisco Renato C. Araújo, and Leobino N. Sampaio. On supporting iot data aggregation through programmable data planes. *Computer Networks*, 177:107330, 2020.

[134] Avinab Marahatta, Youshi Wang, Fa Zhang, Arun Kumar Sangaiah, Sumarga Kumar Sah Tyagi, and Zhiyong Liu. Energy-aware fault-tolerant dynamic task scheduling scheme for virtualized cloud data centers. *Mobile Networks and Applications*, 24(3):1063–1077, 2019.

[135] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 3 2008.

[136] Marouen Mechtri, Chaima Ghribi, and Djamal Zeghlache. A scalable algorithm for the placement of service function chains. *IEEE Transactions on Network and Service Management*, 13(3):533–546, 2016.

[137] Sevil Mehraghdam, Matthias Keller, and Holger Karl. Specifying and placing chains of virtual network functions. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 7–13, 2014.

[138] Sevil Mehraghdam, Matthias Keller, and Holger Karl. Specifying and placing chains of virtual network functions. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 7–13, 2014.

[139] memcached contributors. memcached. `https://memcached.org/`. Accessed: 2022-01-29.

[140] Sebastiano Miano, Roberto Doriguzzi-Corin, Fulvio Risso, Domenico Siracusa, and Raffaele Sommese. Introducing smartnics in server-based data plane processing: The ddos mitigation use case. *IEEE Access*, 7:107161–107170, 2019.

[141] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In

Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, page 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[142] Microsoft. Scalable networking: Eliminating the receive processing bottleneck—introducing rss. `https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fdownload.microsoft.com%2Fdownload%2F5%2Fd%2F6%2F5d6eaf2b-7ddf-476b-93dc-7cf0072878e6%2Fndis_rss.doc`. Accessed: 2022-10-18.

[143] Shangfeng Mo, Hong Chen, Xiaoying Zhang, and Cuiping Li. Tinyqp: A query processing system in wireless sensor networks. In Jianyong Wang, Hui Xiong, Yoshiharu Ishikawa, Jianliang Xu, and Junfeng Zhou, editors, *Web-Age Information Management*, pages 788–791, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[144] Ioannis D. Moscholios, Mariusz Głąbowski, Panagiotis G. Sarigiannidis, and Michael D. Logothetis. A special issue on modeling, dimensioning, and optimization of 5g communication networks, resources, and services. *Applied Sciences*, 12(4), 2022.

[145] Netronome. Agilio cx smartnics. `https://www.netronome.com/products/agilio-cx/`. Accessed: 2022-01-29.

[146] APS Networks. APS Networks 2112d. `https://www.aps-networks.com/products/aps2112d/`. Accessed: 2022-01-29.

[147] Kong Soon Ng, Chin-Sien Moo, Yi-Ping Chen, and Yao-Ching Hsieh. Enhanced coulomb counting method for estimating state-of-charge and state-of-health of lithium-ion batteries. *Applied energy*, 86(9):1506–1511, 2009.

[148] Duong Tuan Nguyen, Chuan Pham, Kim Khoa Nguyen, and Mohamed Cheriet. Placement and chaining for run-time iot service deployment in edge-cloud. *IEEE Transactions on Network and Service Management*, 17(1):459–472, 2020.

[149] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, 7 2012.

[150] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[151] Inc. NumFOCUS. Pandas. `https://pandas.pydata.org/`. Accessed: 2022-01-29.

[152] NVIDIA. NVIDIA Connectx-7. `https://nvdam.widen.net/s/srdqzxgdr5/connectx-7-datasheet`. Accessed: 2022-01-29.

[153] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[154] Oracle. Oracle nosql. `https://www.oracle.com/database/nosql/technologies/nosql/`. Accessed: 2022-01-29.

[155] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), 8 2015.

[156] Claus Pahl, Sven Helmer, Lorenzo Miori, Julian Sanin, and Brian Lee. A container-based edge cloud paas architecture based on raspberry pi clusters. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 117–124. IEEE, 2016.

[157] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 194–206, New York, NY, USA, 2021. Association for Computing Machinery.

[158] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic, and Simone Mangiante. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

[159] Ben Pfaff and Bruce Davie. The Open vSwitch Database Management Protocol. RFC 7047, December 2013.

[160] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. A survey on recent advances in transport layer protocols. *IEEE Communications Surveys & Tutorials*, 21(4):3584–3608, 2019.

[161] V Pop, HJ Bergveld, PHL Notten, JHG Op het Veld, and Paulus PL Regtien. Accuracy analysis of the state-of-charge and remaining run-time determination for lithium-ion batteries. *Measurement*, 42(8):1131–1138, 2009.

[162] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 209–215, New York, NY, USA, 2019. Association for Computing Machinery.

[163] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.

[164] Linux Foundation Projects. Dpdk. `https://www.dpdk.org/`. Accessed: 2022-10-18.

[165] Ryan Izard Qing Wang, Geddings Barrineau. Floodlight controller. `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview`, 2018. Accessed: 2022-01-29.

[166] Moritz Quandt, Benjamin Knoke, Christian Gorldt, Michael Freitag, and Klaus-Dieter Thoben. General requirements for industrial augmented reality applications. *Procedia CIRP*, 72:1130–1135, 2018. 51st CIRP Conference on Manufacturing Systems.

[167] Hoan Nguyen Mau Quoc, Martin Serrano, John G. Breslin, and Danh Le Phuoc. A learning approach for query planning on spatio-temporal iot data. In Krzysztof Janowicz, Werner Kuhn, Federica Cena, Armin Haller, and Kyriakos G. Vamvoudakis, editors, *Proceedings of the 8th International Conference on the Internet of Things, IOT 2018, Santa Barbara, CA, USA, October 15-18, 2018*, pages 1:1–1:8. ACM, 2018.

[168] Amir M. Rahmani, Tuan Nguyen Gia, Behailu Negash, Arman Anzanpour, Iman Azimi, Mingzhe Jiang, and Pasi Liljeberg. Exploiting smart e-health gateways at the edge of healthcare internet-of-things: A fog computing approach. *Future Generation Computer Systems*, 78:641–658, 2018.

[169] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, 6 2012. USENIX Association.

[170] Charalampos Rotsos, Daniel King, Arsham Farshad, Jamie Bird, Lyndon Fawcett, Nektarios Georgalas, Matthias Gunkel, Kohei Shiomoto, Aijun Wang, Andreas Mauthe, Nicholas Race, and David Hutchison. Network service orchestration standardization: A technology survey. *Computer Standards & Interfaces*, 54:203–215, 2017. SI: Standardization SDN&NFV.

[171] Stefanos Sagkriotis, Christos Anagnostopoulos, and Dimitrios P. Pezaros. Energy usage profiling for virtualized single board computer clusters. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2019.

[172] Stefanos Sagkriotis, Kostas Kolomvatsos, Christos Anagnostopoulos, Dimitrios P. Pezaros, and Stathes Hadjiefthymiades. Knowledge-centric analytics queries allocation in edge computing environments. In *2019 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2019.

[173] Stefanos Sagkriotis and Dimitrios Pezaros. Accelerating kubernetes with in-network caching. In *Proceedings of the SIGCOMM '22 Poster and Demo Sessions*, SIGCOMM '22, page 40–42, New York, NY, USA, 2022. Association for Computing Machinery.

[174] Stefanos Sagkriotis and Dimitrios Pezaros. Scalable data plane caching for kubernetes. In *2022 18th International Conference on Network and Service Management (CNSM)*, pages 345–351, 2022.

[175] Stefanos Sagkriotis and Dimitrios Pezaros. Scale-friendly in-network coordination. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 5747–5752, 2022.

[176] Sahel Sahhaf, Wouter Tavernier, Matthias Rost, Stefan Schmid, Didier Colle, Mario Pickavet, and Piet Demeester. Network service chaining with optimized network function embedding supporting service decompositions. *Computer Networks*, 93:492–505, 2015. Cloud Networking and Communications II.

[177] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, page 150–156, New York, NY, USA, 2017. Association for Computing Machinery.

[178] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 785–808. USENIX Association, 2021.

[179] scikit-learn developers. Scikit-learn. `https://scikit-learn.org/stable/`. Accessed: 2022-01-29.

[180] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery.

[181] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[182] Giuseppe Siracusano and Roberto Bifulco. Is it a smartnic or a key-value store? both! In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, page 138–140, New York, NY, USA, 2017. Association for Computing Machinery.

[183] Nina Slamnik-Kriještorac, Haris Kremo, Marco Ruffini, and Johann M Marquez-Barja. Sharing distributed and heterogeneous resources toward end-to-end 5g networks: a comprehensive survey and a taxonomy. *IEEE Communications Surveys & Tutorials*, 22(3):1592–1628, 2020.

[184] Tejas Subramanya and Roberto Riggio. Machine learning-driven scaling and placement of virtual network functions at the network edges. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 414–422, 2019.

[185] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with gpus and click. In *Architectures for Networking and Communications Systems*, pages 25–35, 2013.

[186] Mohammad M. Tajiki, Stefano Salsano, Luca Chiaraviglio, Mohammad Shojafar, and Behzad Akbari. Joint energy efficient and qos-aware path allocation and vnf placement for service function chaining. *IEEE Transactions on Network and Service Management*, 16(1):374–388, 2019.

[187] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. FLAIR: Accelerating reads with consistency-aware network routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 723–737, Santa Clara, CA, February 2020. USENIX Association.

[188] Hangzhou Ruideng Technology. UM24C. `https://www.mediafire.com/folder/0jt6xx2cyn7jt`. Accessed: 2022-12-20.

[189] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[190] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, June 2009. USENIX Association.

[191] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 91–104. USENIX Association, 2004.

[192] Amir Varasteh, Basavaraj Madiwalar, Amaury Van Bemten, Wolfgang Kellerer, and Carmen Mas-Machuca. Holu: Power-aware and delay-constrained vnf placement and chaining. *IEEE Transactions on Network and Service Management*, 18(2):1524–1539, 2021.

[193] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), 2 2020.

[194] Shie-Yuan Wang, Chia-Ming Wu, Yi-Bing Lin, and Ching-Chun Huang. High-speed data-plane packet aggregation and disaggregation by p4 switches. *Journal of Network and Computer Applications*, 142:98–110, 2019.

[195] Putu Wiramaswara Widya, Yoga Yustiawan, and Joonho Kwon. A onem2m-based query engine for internet of things (iot) data streams. *Sensors*, 18(10), 2018.

[196] Zhimin Xi, Modjtaba Dahmardeh, Bing Xia, Yuhong Fu, and Chris Mi. Learning of battery model bias for effective state of charge estimation of lithium-ion batteries. *IEEE Transactions on Vehicular Technology*, 68(9):8613–8628, 2019.

[197] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25–33, New York, NY, USA, 2019. Association for Computing Machinery.

[198] Zhichao Xu, Xiaoning Zhang, Shui Yu, and Ji Zhang. Energy-efficient virtual network function placement in telecom networks. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–7, 2018.

[199] Lily Yang, Todd A. Anderson, Ram Gopal, and Ram Dantu. Forwarding and Control Element Separation (ForCES) Framework. RFC 3746, April 2004.

[200] Nong Ye and Qiang Chen. An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. *Quality and Reliability Engineering International*, 17(2):105–112, 2001.

[201] 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, 2019.

[202] Erik Zeitler and Tore Risch. Scalable splitting of massive data streams. In Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, and Chiemi Watanabe, editors, *Database Systems for Advanced Applications, 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II*, volume 5982 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010.

[203] Chaobing Zeng, Fangming Liu, Shutong Chen, Weixiang Jiang, and Miao Li. Demystifying the performance interference of co-located virtual network functions. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 765–773, 2018.

[204] Menglei Zhang, Michele Polese, Marco Mezzavilla, Jing Zhu, Sundeep Rangan, Shivendra Panwar, and Michele Zorzi. Will tcp work in mmwave 5g cellular networks? *IEEE Communications Magazine*, 57(1):65–71, 2019.

[205] Qianyu Zhang, Gongming Zhao, Hongli Xu, Zhuolong Yu, Liguang Xie, Yangming Zhao, Chunming Qiao, Ying Xiong, and Liusheng Huang. Zeta: A scalable and robust East-West communication framework in Large-Scale clouds. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1231–1248, Renton, WA, April 2022. USENIX Association.

[206] Qixia Zhang, Fangming Liu, and Chaobing Zeng. Adaptive interference-aware vnf placement for service-customized 5g network slices. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 2449–2457, 2019.

[207] Naweiluo Zhou, Huan Zhou, and Dennis Hoppe. Containerization for high performance computing systems: Survey and prospects. *IEEE Transactions on Software Engineering*, 49(4):2722–2740, 2023.

[208] Wei Zhou, Li Li, Min Luo, and Wu Chou. Rest api design patterns for sdn northbound api. In *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, pages 358–365, 2014.

[209] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376–389, November 2019.