Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Markus Teivo

Evaluation of low latency communication methods in a Kubernetes cluster

Master's Thesis
Espoo, June 13, 2023

Supervisor:	Senior University Lecturer Vesa Hirvisalo, Aalto University
Advisor:	M.Sc. (Tech.) Jari Karppinen

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Markus Teivo		
Title: Evaluation of low latency communication methods in a Kubernetes cluster			
Date:	June 13, 2023	**Pages:**	70
Major:	Security and cloud computing	**Code:**	SCI3084
Supervisor:	Senior University Lecturer Vesa Hirvisalo		
Advisor:	M.Sc. (Tech.) Jari Karppinen		

Evolving technologies and the aim to support the ever-growing data traffic in mobile networks present a constantly increasing trend in latency and reliability requirements. With the need to respond rapidly to varying traffic by scaling provided services, movement from static and specialized hardware towards scalable cloud computing has been inevitable. Containers now form the core of cloud computing, and enable rapid deployment and scaling of applications by leveraging their inherent virtualization. Naturally, there is a constant search for more efficient methods of inter-container communication.

The use of the operating system kernel and related excessive copies of data is a major resource sink in the traditional Internet protocol suite (TCP/IP). Techniques such as Remote Direct Memory Access (RDMA) can be used to completely bypass the kernel involvement in communication and to eliminate unneeded data copies. However, solutions to bring RDMA communication for containerized clusters, while preserving the advantages of virtualization, are still in the prototyping stage. The main problems are the virtualization of the required hardware, and the different programming interface of RDMA compared to traditional TCP/IP.

This thesis investigates Freeflow, a proposed RDMA virtualization prototype, and compares the networking performance between TCP/IP communication and RDMA with regard to containerized clusters.

The results indicate that RDMA outperforms TCP/IP by an order of magnitude in latency compared to bare metal. It is shown that the Freeflow prototype has potential in virtualizing RDMA, improving performance of the container network on the performed TCP/IP testing. Additionally, the measured performance of an existing solution to translate TCP/IP into RDMA communication shows negligible difference to native RDMA.

Keywords:	Cloud Computing, Kubernetes, Container, Cluster, Remote Direct Memory Access, Virtualization
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Markus Teivo		
Työn nimi: Alhaisen viiveen kommunikaatiometodien arviointi Kubernetes-klusterissa			
Päiväys:	13. kesäkuuta 2023	**Sivumäärä:**	70
Pääaine:	Tietoturva ja pilvilaskenta	**Koodi:**	SCI3084
Valvoja:	Vanhempi yliopistonlehtori Vesa Hirvisalo		
Ohjaaja:	Diplomi-insinööri Jari Karppinen		

Kehittyvät teknologiat ja jatkuvasti kasvava dataliikenne mobiiliverkoissa samalla laajentavat viive- ja toimintavarmuusvaatimuksia. Tarve vastata nopeasti vaihtelevaan liikenteen määrään skaalaamalla tarjottuja palveluita on tuonut väistämättömän siirroksen staattisesta ja erikoistuneesta laitteistosta skaalautuvaan pilvilaskentaan. Ohjelmistokontit muodostavat nyt pilvilaskennan perustan, ja mahdollistavat nopean sovellusten käyttöönoton ja skaalauksen virtualisaatiota hyödyntämällä. Ohjelmistokonttien väliseen kommunikaatioon etsitään luonnollisesti yhä tehokkaampia ratkaisuja.

Käyttöjärjestelmän ytimen eli kernelin käyttö ja siihen liittyvät ylimääräiset datan kopio-operaatiot käyttävät suuren osan resursseista perinteisessä Internet-protokollajärjestelmässä (TCP/IP). Käyttämällä tekniikoita kuten Remote Direct Memory Access (RDMA), käyttöjärjestelmän ydin voidaan ohittaa ja ylimääräiset datan kopio-operaatiot estää. Ratkaisut RDMA-tekniikan käyttöön ohjelmistokontteihin pohjautuvissa klustereissa siten, että virtualisaatiosta saatu hyöty säilytetään, ovat kuitenkin vielä prototyyppiasteella. Tärkeimmät ongelmat ovat RDMA-laitteiston virtualisaatio ja RDMA-kommunikoinnissa käytettävä erityyppinen ohjelmointirajapinta verrattuna TCP/IP-järjestelmään.

Tässä diplomityössä tarkastellaan Freeflow-prototyyppiä RDMA-tekniikan virtualisaatiossa, ja kokeellisessa osuudessa verrataan RDMA-tekniikan suorituskykyä TCP/IP-tekniikkaan liittyen ohjelmistokonttipohjaisiin klustereihin.

Tuloksena havaitaan, että RDMA-tekniikan suorituskyky viiveen suhteen on kymmenkertainen TCP/IP-tekniikkaan verrattuna. Nähdään, että Freeflow-prototyyppi tarjoaa potentiaalisen lähtökohdan RDMA-tekniikan virtualisaatioon, parantaen ohjelmistokonttiverkon suorituskykyä TCP/IP-kokeessa. Lisäksi, tutkittu ratkaisu TCP/IP-kommunikaation kääntämiseen RDMA-kommunikaatioksi suoriutuu vähäisellä erolla natiiviin RDMA-tekniikkaan.

Asiasanat:	Pilvilaskenta, Kubernetes, ohjelmistokontti, klusteri, Remote Direct Memory Access, virtualisaatio
Kieli:	Englanti

Acknowledgements

This thesis was conducted at Nokia Oyj in Espoo, Finland.

I wish to thank professor Vesa Hirvisalo and Jari Karppinen for their guidance and valuable feedback, and the opportunity to perform research on this interesting topic.

This thesis work has been an invaluable learning experience. I am grateful for my advisor, supervisor, but also myself, for patience and perseverance during this thesis work, which went through long hibernation periods. I am also grateful for my colleagues for proofreading, and for my fellow thesis workers for their valuable peer support.

Finally, I would like to thank my family for supporting me in my studies, and all fellow students met along this journey to make it eventful and unforgettable.

Espoo, June 13, 2023

Markus Teivo

Abbreviations and Acronyms

5G	Fifth Generation Mobile Network
Telco	Telecommunications
RAN	Radio Access Network
NF	Network Function
K8s	Kubernetes
CPU	Central Processing Unit
RAM	Random Access Memory
I/O	Input/Output
VM	Virtual Machine
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
TCP/IP	Internet protocol suite
CNI	Container Network Interface
OCI	Open Container Initiative
OSI	Open Systems Interconnection
RoCE	RDMA over Converged Ethernet
OFED	Open Fabrics Enterprise Distribution
CEE	Converged Enhanced Ethernet
IBTA	Infiniband Trade Association
RDMA	Remote Direct Memory Access
DPDK	Data Plane Development Kit
NIC	Network Interface Controller
RNIC	RDMA-capable NIC
HPC	High Performance Computing
SR-IOV	Single-Root I/O Virtualization
API	Application Programming Interface
HCA	Host Channel Adapter
QP	Queue Pair

Contents

Abbreviations and Acronyms **5**

1 Introduction **8**
1.1 Problem statement . . . 9
1.2 Scope and methodology . . . 9
1.3 Contributions . . . 10
1.4 Structure of the Thesis . . . 10

2 Background **12**
2.1 Virtualization of telco applications . . . 12
2.2 Containerization in cloud computing . . . 13
2.2.1 Container orchestration . . . 16
2.2.2 Container networking . . . 17
2.3 Interconnect technologies . . . 20
2.3.1 Ethernet . . . 21
2.3.2 Infiniband . . . 22

3 Low-latency interconnect communication **24**
3.1 Latency and throughput in networks . . . 25
3.2 Kernel bypass and zero-copy networking . . . 27
3.3 RDMA . . . 28
3.3.1 iWARP . . . 30
3.3.2 RoCE . . . 31
3.4 RDMA in containerized clusters . . . 32
3.4.1 Freeflow . . . 33

4 Experimentation setup **36**
4.1 Hardware . . . 36
4.1.1 Raspberry Pi 4 setup . . . 37
4.1.2 Nokia AirFrame server setup . . . 37
4.2 Software tools . . . 38

4.2.1 K3s, Lightweight Kubernetes . . . 38
4.2.2 Mellanox VMA . . . 39
4.3 Testing tools . . . 39
4.3.1 wrk and Nginx . . . 40
4.3.2 iperf3 . . . 40
4.3.3 Sockperf . . . 41
4.3.4 RDMA performance tools . . . 41
4.4 Test cases . . . 42
4.4.1 Test case 1: TCP/IP benchmark . . . 42
4.4.2 Test case 2: RDMA performance benchmark . . . 43
4.4.3 Test case 3: Mellanox VMA bare-metal TCP RDMA acceleration . . . 43
4.4.4 Test case 4: Freeflow TCP in cluster . . . 44

5 Results **45**
5.1 Test case 1: TCP/IP benchmark . . . 45
5.2 Test case 2: RDMA performance benchmark . . . 47
5.3 Test case 3: Mellanox VMA bare-metal TCP RDMA acceleration . . . 49
5.4 Test case 4: Freeflow TCP in cluster . . . 51
5.5 Summary . . . 53

6 Discussion **55**
6.1 Future work . . . 56

7 Conclusions **58**

A Additional graphs **66**

B Freeflow router Kubernetes deployment **69**

Chapter 1

Introduction

The amount of data transmitted in mobile networks, and networks in general, is continuously increasing. With 5G here and 6G in the horizon, the amount of data to process is not to be expected to stop growing. Throughput and latency requirements of mobile networks take large leaps in between generations, and growing amount of devices will use more of the capacity the network has to offer.

Similarly to for example the Internet of Things (IoT), factory automation and cloud robotics [22], as well as entertainment and healthcare [24], the telecommunications industry is becoming much more latency-critical when development continues.

Radio access networks (RAN) used in telecommunications (telco) specifically have strict requirements in latency and availability, while new generation technologies constantly tighten the requirements further. Traffic load in RANs additionally varies according to time of day, and possibly different events where large amounts of people are using the network at the same time or additionally at the same location. Therefore the network has to scale quickly to accommodate different loads. [64]

Advances in radio technology require innovations in network architecture as well [71]. Already for 4G LTE networks, scalability, reliability and timeliness were the main requirements for telecommunications services [7]. The vision for 5G networks was to support 1000 times more traffic compared to 4G [68], while increasing the supported amount of connected devices by 100 times, and decreasing latency by 5 times [2].

Cloudification of telco resources has already been in progress for the past years. Cloud technologies have adopted largely container-based structure, because of its scalability, ease of deployment and automated orchestration possibilities. In this structure, applications may be deployed in a microservice architecture, where parts of an application are deployed as containers,

providing robustness and scalability.

When communicating between machines in data centers, there is always an overhead of the traditional Internet protocol suite (TCP/IP) networking stack in kernel resulting from data copies and kernel access itself. By using zero-copy and kernel bypass technologies, this stack can be bypassed, resulting in better latencies and throughput. This thesis analyses the networking implications of traditional kernel networking, and technologies to bypass the kernel stack, while examining their viability in containerized cloud environments. It is determined that the implementation of kernel bypass and zero-copy networking is problematic in containerized clouds for various reasons, and this paper also reviews proposed solutions.

1.1 Problem statement

This thesis aims to research what are the problems of utilizing Remote Direct Memory Access (RDMA) technology on containerized clouds and how do the proposed solutions preserve the benefits of virtualization when implementing the technology. Additionally, this thesis evaluates the general performance of RDMA and the implications of running traditional Ethernet-based applications with RDMA technology.

1.2 Scope and methodology

This thesis focuses on the performance in throughput and, in particular, latency of communication methods between cluster pods, in the same node or separate nodes. While the motivation of this thesis is based on the need of the virtualization of telecommunications (telco) application parts with high performance, this thesis does not focus further on telco applications, but rather on cloud computing in general. Although the performance of a radio access network can be improved by upgrading the radio equipment, the focus of this thesis is on the networking hardware and interconnection performance. More specifically, we analyse the performance improvements that methods bypassing steps in the networking stack could give. In this regard, we research the proposed solutions for using these methods in cloud networks.

On our testing, we focus on creating a benchmark using the traditional networking stack, to compare to Remote Direct Memory Access (RDMA) benchmarks on bare metal. We perform a TCP/IP benchmark on both bare metal and on a Kubernetes cluster using the Flannel container network. We

have two test setups of lightweight and high performance hardware, but focus on the high performance hardware and compare the TCP/IP benchmark to two different RDMA benchmarks. We benchmark RDMA in a bare metal setting by using purpose-built RDMA performance tools. We additionally accelerate Ethernet-based performance measurement tools with RDMA by using a translation library. Finally, we also test the performance of Freeflow TCP, a solution accelerating a container network to bare metal performance, in a containerized cluster.

1.3 Contributions

The key contributions of this thesis is as follows. The authors perform a background research of proposed RDMA virtualization solutions in containerized clusters, and determine that Freeflow, a similar solution to existing TCP/IP virtualization based on virtual switches, is a viable approach.

The measurements show that RDMA achieves microsecond-level latency, performing an order of magnitude better than TCP/IP. The RDMA latencies are additionally very stable, whereas TCP/IP measurements have large variance due to varying processor usage. It is determined that when using a translation library to accelerate TCP/IP-based applications with RDMA, the performance difference to native RDMA is mostly only a few microseconds, and thus programming applications specifically for RDMA use may not be needed. The tested TCP branch of the Freeflow prototype accelerates the container network near bare metal performance, suggesting that its RDMA implementation should also incur little overhead.

The authors present a possible issue with the translation libraries to be studied on future work. It is additionally determined that the required driver installation process to containers is not straightforward, and only bare metal RDMA is thus benchmarked in this thesis.

1.4 Structure of the Thesis

The structure of this thesis is as follows. The background of the evolution of mobile networks towards cloud computing, the containerization in cloud computing, container management and networking, and relevant interconnect technologies are explained in Chapter 2. In Chapter 3 we examine what generates latency in interconnects and the relation of latency and throughput. We move on to explain the meaning and performance improvements of kernel bypass and zero-copy technologies, focusing on RDMA, and concluding in the

problems and recent proposals of bringing RDMA to containerized clusters. In Chapter 4 we explain our test setups, testing tools and test cases, and evaluate the results for our test cases in Chapter 5. We discuss our results, our success in evaluating the research questions, and possible future work in Chapter 6. Finally, we summarize and conclude the thesis in Chapter 7.

Chapter 2

Background

In this chapter, we explain the relevant background of mobile networks, cloud computing and machine interconnection. We begin by stating how the telecommunications (telco) industry has evolved to heavily utilize virtualization and cloud computing. Next, we examine the evolution of the deployment of virtualized applications, and see how containers have become the de facto virtualization solution in microservice-type cloud application deployments. We move on to explain the need of orchestration in the management of large amounts of containers, and the software technologies needed in container networking. Finally, we give insight to hardware technologies in machine interconnection, and in particular, two major interconnect technologies that are integral in this thesis.

2.1 Virtualization of telco applications

Multiple generations of technologies, with evolution from first generation mobile networks (1G) to present 5G form together the Radio Access Network (RAN). The Public Switched Telephone Network (PSTN) was the first in a series of revolutions in communication networks, and Global System for Mobile (GSM) became the dominant mobile communications system, with General Packet Radio Service (GPRS) first providing packet switching and thus enabling the sharing of a single link for multiple transmitters and receivers. [24]

The development of 3G and 4G networks focused on delivering better coverage and throughput [24]. Serrano et al. [61] presented that the 3G network had a typical throughput of 1Mbps and latency of 100ms. This amount of latency would be the most noticeable for small transmissions, such as 1 kilobyte, as the data transmission time is considerably faster than the

network latency. Small data transmissions would therefore not benefit from increasing bandwidth in the network, but rather from addressing the network latency. As an example, a latency range of at most 200-250 milliseconds is required for natural human conversation in voice over IP (VoIP) [24].

Now, emergent 5G networks focus on new visions, ubiquituous computing, and services ranging from Internet of Things (IoT), to latency critical telesurgery, virtual reality, autonomous vehicles, and more. In particular, 5G aims for ultra-reliable low-latency communication (URLLC), operating in below-millisecond latency and still employing high reliability. Latency performance is the determining factor in the business success of a wide range of services, including cloud services and video streaming. [24]

For a telco RAN, it is essential to be able to provide the mobile network to users at all times. This implies responding rapidly to load differences, which may vary hourly or daily, by scaling the required services. [64] Telco applications consist of Network Functions (NF), which have in the past generations been hardware components where network traffic passes through. The problem of this type of implementation is the requirement of specialized hardware dedicated for the specific network tasks, and future upgrades of said hardware when performance improvements and architectural changes are needed. [66]

5G specifically was defined as a service-based architecture and introduced the concept of network function virtualization [39]. Virtual Network Functions (VNF) enable rapid instantiation, termination and updating compared to physical implementations, and thus enable rapid response to load requirements while substantially easing the modification of NFs [17]. VNFs can be deployed without specializing the hardware for each different NF, and multiple VNFs can be deployed on a single piece of hardware because of their virtualized nature [66].

To process large amounts of data and to provide more reactive services, emergent *edge cloud* concept aims to deploy parts of the service at the edge, near the end user. Naturally, lowest possible latency is preferred between the service components to communicate as efficiently as possible. [3] Telco applications, being now alike other cloud applications, have been moving towards *container*-based structure in the deployment and packaging of applications to better benefit from virtualization. [64]

2.2 Containerization in cloud computing

In traditional application deployment, such as the deployment of hardware NFs, applications are deployed directly on top of the operating system (OS)

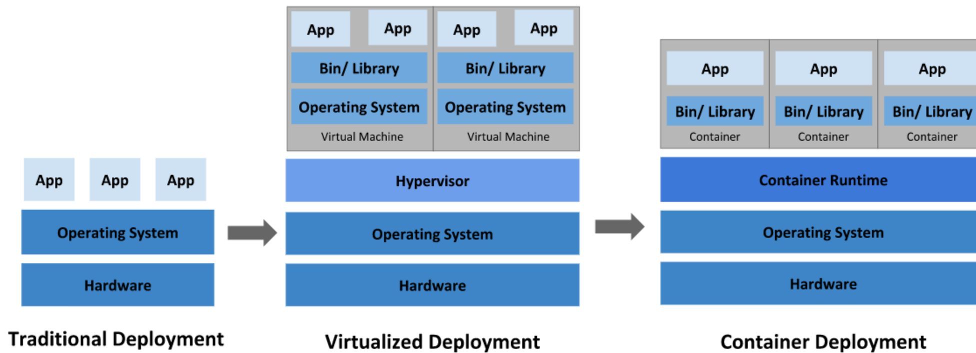

Figure 2.1: Evolution of software deployment from traditional means, to virtual machines, and containers. [31]

as seen in the left in Figure 2.1. Using this solution, to ensure that each application has access to sufficient resources, the scaling of applications would require the deployment of each application on different physical servers, and thus usually leaving resources underutilized overall [31].

Before recent movement towards containers, Virtual Machines (VM) have been in the core of cloud computing [22]. Virtual Machines use *hypervisor*-based virtualization, shown in the middle of Figure 2.1. A hypervisor software may be set up directly on hardware, or on the host operating system. The hypervisor software provides a VM moderated access to existing hardware through *emulation*. Because of this, VMs are well isolated between themselves and the host operating system, and enable even the emulation of foreign devices, Central Processing Unit (CPU) architectures, or operating systems for the host system. VMs can additionally be easily migrated to different systems because of the abstraction from the hardware. In turn, a VM image needs to include an operating system, and thus the start-up of its own operating system and its components is required on deployment. Additionally, hardware emulation incurs additional performance overhead, and all resources must be reserved before VM start-up. Well known VM solutions include Microsoft Hyper-V, VMware and Oracle VirtualBox. [16]

Containers, in contrast, are a form of operating system (OS) level virtualization. There is no need to include a full operating system inside a container, and further, there is no need for a hypervisor. [64] Container virtualization leverages OS kernel features in the isolation of processes. A container is essentially a process itself. Linux containers (LXC) project implements containers using two kernel features, *namespaces* and *control groups*. Namespaces may be used to create an isolated environment for processes, process

groups or even complete subsystems such as the kernel network stack. The kernel manages the passing and synchronization of required resources into these namespaces. Control groups are not mandatory in process isolation, but they provide tools for resource limitation and control. [16]

Software running on containers utilizes the existing kernel and resources such as the CPU on host system, but needs in turn to be compatible with them. Container images do not need to contain drivers, kernel or other vital tools to run an operating system, making them very small in size compared to VM images. A container can still be packaged with all the required libraries with correct versions, easing the deployment of applications that require specific dependencies. In contrast to heavy VMs, lightweight containers can be deployed in as fast as a few milliseconds. [16]

Containers and their images are managed by a container runtime, which is a software that performs the necessary operations to start up and run the container processes, usually according to specifications provided by the Open Container Initiative (OCI). Two well known container runtimes are `containerd` and CRI-O. [19] The deployment of containers is visualized on the right in Figure 2.1.

Containerized deployment is a natural fit for deploying applications in a *microservice* architecture, where different components of an application are separated to multiple containers [29]. In microservice architecture, a single application component should be independently replaceable and upgradeable, performing a single function while communicating with other application components. A microservice-based application is easy to scale in multiple different ways. The application may be scaled vertically by further decomposing the application into as small pieces as possible. Horizontal scaling may be performed by running multiple copies of a service, and balancing the load equally for each copy. Data can be additionally partitioned such that each server holds a subset of the data. [50]

Microservice architecture is opposed by monolithic architecture, where all functionality is combined in a single application. Even the development of a large monolithic application may prove more difficult compared to an application broken down to small microservices, but more importantly, the scaling of a monolithic application is only possible through duplication of the entire application. [64]

The main drive for containers becoming the de facto solution in the management and deployment of large cloud applications stems from three main benefits: isolation, controllability and portability. Containers isolate from each other, eliminating any conflicts in resource usage, including network ports, interfaces and routes. Portability ensures that the container functions identically in any host machine the container is deployed on, and retains

the same network address. Controllability enables container orchestration systems to monitor container health and performance, and enforce resource limits and network settings. [29]

2.2.1 Container orchestration

Container runtime engines are only responsible for running single containers. Container orchestrators are developed to run and control large amounts of containers across multiple connected machines. The most widely known container orchestrator platforms are Kubernetes and Docker Swarm [6]. In this thesis, we focus on Kubernetes container orchestration. Kubernetes, abbreviated as K8s, was open-sourced by Google in 2014 [31].

A container orchestrator engine generally performs three types of functions, namely resource management, scheduling and service management functions. The orchestrator manages access and limits to resources such as memory, processors, disk space and the interaction with the host file system via volumes. Additionally, remote cloud file system access can be granted via persistent volumes, and networking configuration can be managed. [4]

With scheduling functions, the orchestrator controls the deployment of containers across the cluster according to deployment settings, and restarts containers when they are crashed or in need of updates. A scheduler can perform automatic horizontal scaling by replicating containers. [4]

Various fine-tuning options for the building and management of complex applications are provided via service management capabilities. Containers may be labeled, grouped together or further isolated, dependencies between services may be set, and service readiness can be checked before making it available online. [4]

The machines involved in a Kubernetes cluster are named *nodes*. In the cluster, the nodes belong in two categories, *master* and *worker* nodes. The master node or nodes form the *control plane*, which performs all decisions in the cluster orchestration [6]. A cluster may involve multiple master nodes, distributing the control plane to a wider area and in the event of one node failure, another master node takes the orchestration tasks. The application payload is preferably deployed across the worker nodes. [32]

In a Kubernetes cluster, the smallest component managed by the orchestrator is a *pod*. Contrary to containers, which can be compared to processes, a pod is an environment for containers. Inside a pod, a set of one or multiple containers are run according to specifications. The containers within a pod share storage and networking resources. [32]

On each worker node, three components are run: a `kubelet` agent, the container runtime, and `kube-proxy`. The `kubelet` agent is responsible of run-

ning all containers within a pod and monitoring their health. The `kube-proxy` is responsible for maintaining network rules, enabling network communication inside or outside of the cluster. This component implements the Kubernetes Services abstraction, further described in the end of Section 2.2.2. Finally, a container runtime engine compliant with the Container Runtime Interface (CRI) specification runs the containers. The CRI is a set of requirements allowing the container runtime present in each node to be accessed and used by the `kubelet` agent. [32]

On master nodes, four components are present: `kube-apiserver`, `etcd`, `kube-scheduler` and `kube-controller-manager`. The `etcd` key-value store is used as a database for all data related to orchestration of the cluster. The `kube-scheduler` is responsible for assigning pod deployment to nodes based on their specifications and restrictions. [32]

The control plane `kube-controller-manager` component runs different types of controller processes. An Endpoint controller creates a link between the a Kubernetes Service and the actual pods. Some other types of controllers are involved in fault tolerance policies and scaling, such as Node controller and Replication controller. The Node controller will reschedule pods in other available worker nodes when one becomes unreachable. The Replication controller is responsible for maintaining the required number of scaled pods. [32]

The Kubernetes Application Programming Interface (API) is exposed via the control plane component `kube-apiserver`, and functions as the front-end for the control plane. It is controlled using the `kubectl` command, which sends Hypertext Transfer Protocol (HTTP) requests to the Kubernetes API. Via the API, it is possible to query and manipulate for example any pod or configuration setting within the cluster. The API can also be used to perform operations such as *rolling updates*. Rolling updates can be used to gradually change application versions without downtime, leaving always a portion of pods running while another portion is restarted with a different version. [32]

2.2.2 Container networking

Networking is an important part of a Kubernetes cluster, allowing containers to communicate within the cluster, or exposing them to the outside [55]. Containers and pods in orchestrated systems are networked using plug-ins based on the Container Network Interface (CNI) specification [10]. Widely used CNI plug-ins include Flannel, Calico, and Weave.

In regular *host mode* networking, the physical network interface of the host along with its IP address and port space are used, and thus the container communicates like any other process. This raises the problems of

possible conflicting ports, and the need to change both IP addresses and ports for different hosts. Proper inter-container communication, especially in large orchestrated clusters, requires the use of *virtual mode* networking in order to achieve isolation and portability. This mode implies the isolation of container network namespaces, and the use of virtual networks including software virtual switches. All traffic will go through the virtual switches, and the routes to the virtual IP addresses of containers or pods can be controlled with the virtual switches to ensure portability. [29]

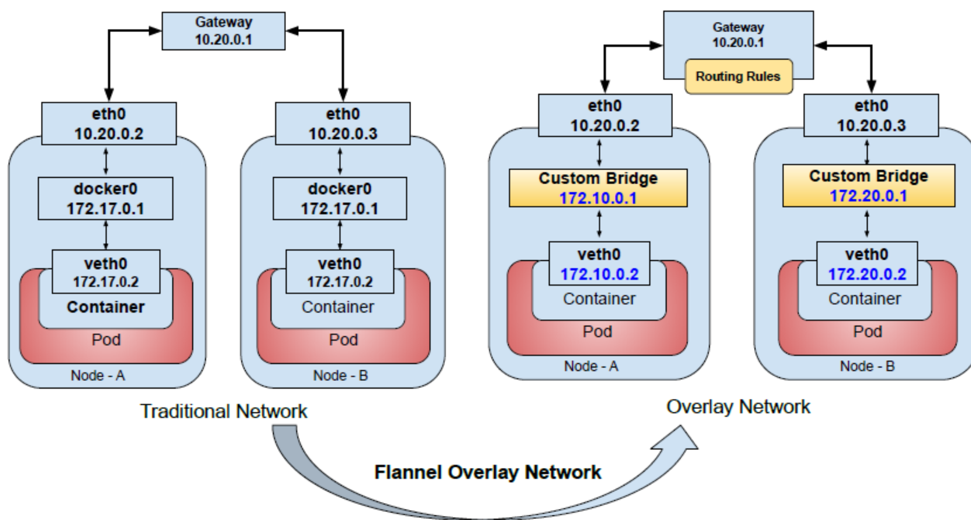

Figure 2.2: Comparison between two virtual mode container networks. [6]

Figure 2.2 shows the difference between two *virtual mode* networks. A traditional container network, which is used when deploying containers on a single machine using tools such as Docker [14], is compared to an overlay network, which is commonly used in orchestrated clusters. When deploying containers locally using Docker, a local virtual network interface such as `docker0` is used as a bridge in the host's private network. Every container will additionally be given its own virtual network interface, `veth`, enabling isolated container networking within a machine. This does not allow connections with containers situated in other nodes, as the virtual IP addresses given to containers within a node can conflict with containers within other nodes, and are not known to other containers outside the internal network of a node. [6]

An overlay network is essentially a virtual network laid on top of another network [66]. With an overlay network, portable and isolated communication

can be achieved between containers and pods residing in different nodes. An overlay network sets up networking tunnels between endpoints by encapsulating the packets using different methods. One common tunneling method is Virtual Extensible Local Area Network (VXLAN), performing encapsulation into UDP packets. [1] The encapsulating packet is used to route the packet to the correct node, where it is decapsulated and may continue to its final destination at the virtual network interface on another pod [66].

On overlay networks, a virtual bridge connects with the virtual interfaces of the pods on Layer 2. An outgoing packet first arrives at the virtual bridge, from where it is forwarded to an overlay tunnel endpoint via host protocol stack Layer 3 IP forwarding for encapsulation. Finally, the packet is sent via the host physical interface, `eth0`. In some CNI designs, the packet is forwarded directly from the pod's virtual interface to the tunnel endpoint without a bridge. [55]

A CNI plug-in daemon sets up the virtual network devices, tunneling options and network policies. Some CNI daemons additionally perform the packet encapsulation or decapsulation themselves instead of the kernel driver. When a pod is created, the CNI binary configures a pod's network by creating its virtual network interface and address. [55]

Instead of an overlay network, many CNI plug-ins support an underlay networking mode as an alternative in inter-host communication. Underlay networking mode uses the host protocol stack as an IP forwarder on Layer 3, if the underlying infrastructure supports routing using the IP addresses of the pods. The underlay networking model does not include a tunnel endpoint and thus packet encapsulation. Calico can fully operate on Layer 3 using Border Gateway Protocol (BGP). [55]

Kapocius et al. [26] measured that the overlay network adds additional compute overhead because of added encapsulation headers, while underlay-based CNI implementations performed closer to bare metal. Flannel was deemed the best in terms of latency when comparing CNI's using overlay networks. The overhead from the use of overlay networks stems from the need to traverse the TCP/IP networking stack two times: once for the physical network interface, and once for the virtual switch [1]. The terms "virtual bridge" and "virtual switch" are essentially interchangeable in this context.

In Kubernetes, the virtual network interface is allocated per each pod, and all containers within the pod will use that same interface. Kubernetes Service abstraction may be used to expose groups of pods to be accessed within the cluster using `ClusterIP`, or from outside the cluster using `NodePort` or `LoadBalancer`. They are abstractions that define the interface to access pods from a specific port and address, redirecting requests to the appropriate pods. The Kubernetes Service abstraction provides permanence of IP addresses

regardless of the pods' actual virtual IP addresses designated by the CNI. A load balancer offers a single IP to then route to the pods according to load. [32]

2.3 Interconnect technologies

In this section, we present the layer model with which different parts of interconnection technologies can be represented, and explore two relevant technologies used to connect machines and resources in data centers.

Networking performance is a major factor in the overall performance of a data center, and thus a reliable and fast interconnect is required. Some proposals have even been made to *disaggregate* resources, meaning that the utilization of e.g. the CPU or memory is performed via a network *fabric* instead of a bus. [21] This can be related to microservices, which in a way represent disaggregated resources.

Innovations such as Message Passing Interface (MPI), Myrinet and Infiniband in high-performance computing (HPC) applications have been motivated to reach microsecond-level latency and address the limitations of Ethernet, but in turn introduce a completely different architecture. Comparing Infiniband or Myrinet to Ethernet, the hardware supports all handling of payload data in user space, without accessing kernel space to perform memory copies or complex pointer redirections. On the other hand, TCP/IP is flexible and protocols such as the Remote Direct Memory Access (RDMA) protocol, which is further examined in this paper, have been shown to work on the Ethernet hardware. [33]

Each part of an interconnection architecture operates on a specific layer in the network. Layers in computer networks are used to represent the communication functions that each play a part in delivering messages from the sender to the recipient. Each layer is only connected with its upper and lower layer, and modifications to one are possible without affecting the other layer functions. The layer model is named the Open Systems Interconnection (OSI) model. [72]

In the following, we explain the seven layers of the OSI model as presented by Zimmermann et al. [72]: the physical, data link, network, transport, session, presentation, and application layer.

The lowest layer, referred to as **Layer 1**, is the physical layer which consists of e.g. the physical wiring, switch and Network Interface Controller (NIC) hardware. The physical layer is responsible for transmitting raw data as digital bits, using electrical, radio, or optical signals.

The data link layer, **Layer 2**, establishes and maintains the links between

machines in a network. The machines can be in a wide or a local area network. It is also possible to send Layer 2 data units, *frames*, to a single node, multiple nodes, or all network nodes. These are referred to as unicast, multicast, and broadcast, respectively. In TCP/IP, the data link layer contains two sublayers, Logical Link Control (LLC) for communication links and Media Access Control (MAC) for device identification.

Layer 3, the network layer, functions when transferring packets between machines, specifically in different networks. It handles routing through intermediate nodes using node addresses.

Layer 4, the transport layer, controls the transportation of the data from source to destination, with the possible use of flow and error control mechanisms. It manages the efficient message transportation for the session.

Layer 5, the session layer, binds and unbinds two presentation layer sessions.

Layer 6, the presentation layer, consists of services needed by the application layer to interpret the exchanged data.

Layer 7, the application layer, is the highest layer and provides the information to the end user, and acts as the data source and sink. Each layer wraps the lower layers, and isolates from the higher layers.

2.3.1 Ethernet

Ethernet is the most widely used technology in the world, with Ethernet networks found everywhere from data centers to homes [27].

The Ethernet TCP/IP suite works using the OSI networking model layers. In the TCP/IP protocol suite, Transmission Control Protocol (TCP) is the protocol for the transport layer, and Internet Protocol (IP) for the network layer. IP carries the TCP data to its destination, including functions such as routing, network address translation (NAT), packet encryption and decryption and returning possible error messages to the sender. IP carries all traffic and routing is performed by hopping through a series of routers, each aware of the next hop router and its interface for the packet. Internet Control Message Protocol (ICMP) message carried by an IP datagram is used to carry various error messages, and in addition used by utilities such as ping. [62]

The TCP protocol is the most widely used Ethernet protocol for its reliability in connection. Data must be intact and in correct order. The TCP protocol is connection-oriented, focusing on maintaining the integrity of a connection between endpoints. Loss of data in transmission is detected and the data resent, by keeping track of each byte received. Additionally, CPU or memory resource shortages are communicated, so that data transmission

can be slowed down if required. TCP congestion control mechanism allows reliable, stable and controlled way of connection, while making decisions whether to send data in smaller or bigger chunks. [62] The User Datagram Protocol (UDP), in contrast, is a connectionless protocol that does not verify the reception of the messages, being simple and lightweight in comparison and allowing other use cases such as broadcasting [70].

Maximum Transmission Unit (MTU) dictates the largest packet sent by TCP. Larger packets than this unit are broken down to smaller chunks by IP, and reassembled at the destination. [62]

The standard interface used by software to manage Ethernet communications is called Berkeley Socket API, allowing all required functions such as connection establishment and sending and receiving data [62].

2.3.2 Infiniband

Infiniband architecture development was first initiated to replace standard busses in input/output (I/O) systems. The busses were deemed inefficient compared to growing processing power. Infiniband introduces a complete packet-switched networking architecture of switches, wires, links, packet formats and other network components. Infiniband is developed by the Infiniband Trade Association (IBTA) founded in 1999. [54]

In high-performance computing (HPC), Infiniband is a widely used interconnect. Of the top 500 supercomputers, in 2009, 30% used Infiniband or Myrinet interconnects [33]. In 2014, Infiniband was the most used interconnect, although 10 gigabit Ethernet gained the lead by 2016 [65].

The Infiniband architecture uses message passing, where the unit of transfer is a message, instead of byte streams. Therefore there is no need to track the boundaries of a message within a byte stream. Infiniband features direct memory-to-memory transfers, direct access to the networking hardware bypassing kernel, and asynchronous transfers. [38]

An Infiniband network consists of end nodes, switches, a Subnet Manager (SM), and an optional router. The Infiniband connection management is performed via a management system, in contrast to being part of TCP specification in TCP/IP. Nodes are attached to the network via a Host Channel Adapter (HCA), which is comparable to Ethernet Network Interface Controllers (NIC). [38]

The Infiniband stack is divided into physical, link, network and transport layers [54]. The transport layer is implemented in the Infiniband HCA hardware in contrast to software implementation in TCP/IP. The services are visible to software via the *Verbs* API, which was introduced as the Open Fabrics Enterprise Distribution (OFED) interface between the HCA and ap-

plication in Infiniband architecture. It is essentially the correspondent of the Socket API of TCP/IP in Infiniband. [38]

The Verbs API supports both channel and memory semantics. Channel semantics resemble the functionality of TCP, as in the programs in both endpoints must explicitly participate in data sending and reception. Memory semantics model in contrast supports a technique named Remote Direct Memory Access (RDMA). This model means that only one side performs an operation, while the other endpoint remains passive. The transfer is performed directly to the memory of the receiving endpoint. The passive endpoint merely grants permission to the transfer by transferring the required information and a registration key of the memory area. [38] We look into RDMA itself, and the queue-based communication integral to the Verbs API, in more detail in Section 3.3.

Chapter 3

Low-latency interconnect communication

A large contributor in data center networking performance is the interconnect used between machines. The latency performance of the interconnects are vital when running more and more latency critical applications. Sensitivity to latency changes differs per different use cases. Three categories can be modeled for systems that are sensitive to microsecond-level latency. Parallel computing relies on fast synchronization of different threads of an application distributed to multiple compute nodes. Distributed memory usage requires fast access latency of the memory of another machine in a cluster. Finally, storage media, when accessing solid state drives and memory caches, also expects microsecond-level latency. [33]

TCP/IP communication includes multiple copy operations in the networking stack. This involves significant CPU and memory usage. The Linux kernel handles a large portion of the communication process, and thus multiple CPU cycles are consumed in kernel to handle the networking tasks. [8]

Ultimately, the aim in interconnect communication is to achieve the lowest possible latency, the highest possible bandwidth, and the lowest CPU utilization. When eliminating computing costs, the network link may be more efficiently utilized. In very high performance needs, commonly solutions aiming to bypass the typical networking stack and to move networking functions from kernel to user space have been used, removing unnecessary copy operations, and easing the overall CPU and memory utilization [26].

In this chapter, methods to achieve very low latency by bypassing the OS kernel and eliminating excessive data copies are explored. We begin by analysing the sources of latency. We move on to examine the meaning of kernel bypass and zero-copy networking, and existing methods to perform this

type of networking. Then our main focus is on Remote Direct Memory Access (RDMA) technology. We explore different RDMA implementations, and finally examine the implications of implementing RDMA into containerized clusters, along with the proposed Freeflow prototype.

3.1 Latency and throughput in networks

In contrast to transferring packets over long distances, where the data transfer in wire alone would impose latency in the ten-millisecond scale, inside a data center environment the latency variation of few milliseconds could impact application performance significantly. Throughput-wise, greater throughput implies a greater number of messages transferred, and throughput variations do not necessarily affect latency in itself. Whether an application is more dependent on latency or throughput is very dependent on the application itself. If an application needs to stall waiting on the response to a network request, it is latency dependent. If other threads can continue running in place of the waiting thread, the application may be more throughput dependent. [33]

The latency for a network packet can be decomposed to several components in the TCP/IP network stack. Namely, queue latency, processing latency, access latency, transmission time, and routing latency form the overall latency. [24]

Queue latency consists of the packet waiting for transmission, and of its waiting time in the buffer when received, waiting for processing. Priority queue and quality of service aware scheduling mechanisms, if supported, may lower the queue latency for higher priority packets. [24]

Processing latency includes signal and packet processing in a node, in physical, link, and hardware layers. Access latency consists of the time taken to get access to a wired or wireless medium in the network, including control signalling. For example, 802.11 protocol request-to-send and clear-to-send messaging, or 4G network processing scheduling grant messaging. Then, transmission of the header and the packet occurs between network nodes. Multiple hops in the network inflict multiple additional access and transmission latencies. Additionally, if a network node performs routing, it adds routing latency. [24]

Inherently, higher transmission rate implies lower transmission time. Techniques at physical, MAC, network, and transport layers may be designed for low-latency networks to reduce the latency components. [24]

On TCP/IP, the software Socket API imposes a significant portion to the overall latency. On message arrival, processor must be interrupted, protocol

stacks are used to find the application to process the message, and the application requires a *context switch* for the data to be copied to application buffer, before finally the message can be processed by the application. [33]

The memory space in an operating system is divided to so called user and kernel space. Some operations, such as networking, are typically isolated to kernel space. The overhead of the kernel network stack divides into three categories: system call overhead, extra data copies, and per-packet processing. System calls cause a switch from user to kernel space when an application interacts with a network Socket. When receiving or sending packets, packets are copied between user and kernel space. Finally, each packet causes an interruption, memory allocation, and the construction of a heavy `sk_buff` data structure, which holds the packet along with all attached headers. [8]

After a packet arrives in the NIC, the incoming packet is first copied to NIC's ring buffer with Direct Memory Access (DMA). From there, a hardware interrupt is sent by the NIC to the processor notifying about the packet. After the interruption, appropriate buffers are allocated, and the packet will be transferred to the kernel protocol stack for processing. When the processing completes, the packet is transferred from kernel space to user space, where an application may use a call to read the packet from Socket. [8]

Traditionally, network transfer includes using two system calls, *read()* and *write()*. The process includes four copies of the data to be sent: DMA copy from initial data location to kernel memory, CPU copy to user memory, CPU copy to Socket buffer in kernel memory, and finally, DMA copy to the NIC. In a CPU copy, the CPU reads a memory location and writes the result again to a new location. While the DMA copies use little CPU resources, the associated system calls still cause context switching. [63]

Additionally, L2 cache misses contribute to memory access latencies, as in a cache miss, the data has to be fetched from memory. CPU checks its cache levels from first cache (L1) onwards, and if the required data is not located there, the next cache levels are checked, and if all caches miss, the data has to be fetched from the Random Access Memory (RAM). Lower cache levels have smaller memory than higher levels, but are faster to use. [15] In the case of nine functions with a single cache miss with memory latency of 50-100 ns, overall latency impact would be 450-900 ns [33].

Variation of latencies per packet, jitter, may occur when packets are grouped for more efficient processing, rather than performing an interrupt for each packet. The interruptions are moderated differently for different operating systems such as Windows and Linux. If the size of a packet is greater than the MTU, processing is required for multiple packets instead of one. This effect may be dampened if the packets will be processed in groups, but latency will increase linearly as the byte count increases. [33]

In cloud environments, when multiple applications are sharing the same NIC on a machine, they may effect the latencies of each other even with dedicated CPU cores. [22]

3.2 Kernel bypass and zero-copy networking

While the kernel network stack performs multiple functions in the packet transfer process, it has fundamental limitations, as we explored in Section 3.1. Overall, kernel usage shows the mismatch between fast hardware and slow software [8]. Most of the data copies between kernel and user space are redundant, taking unnecessary CPU cycles and memory bandwidth, and *zero-copy* techniques aim to eliminate the unnecessary memory accesses [63]. It may also refer to removing the CPU involvement in data copies between memory areas, which can be achieved by directly placing data into memory by e.g. the NIC [27]. *Kernel bypass* refers to techniques with similar impact, by performing the networking stack functions fully in user space, reducing context switching [8].

Zero-copy techniques aim to reduce data copies between the kernel buffers and either the devices such as the NIC, or the application buffers. One technique to reduce the data copies is Direct Memory Access (DMA). It allows hardware with DMA support to perform writes or reads on memory without CPU involvement. Other techniques include dynamically mapping the application buffer to the kernel buffer, modifying the `sk_buff` structure, and buffer sharing between user and kernel memory, which enables data referencing via pointers rather than copies. [63]

Some Linux system calls support zero copy such as `sendfile()` and `splice`. Using `sendfile()` removes the need to use `read()` or `write()` altogether, thus reducing context switches from four to two. Still, a CPU copy and two DMA copies are needed. It keeps the data transfer only between kernel memory and the disk and NIC devices, meaning that the user space is bypassed. `splice()` sets up a data transfer path between kernel memory buffers by creating a pipe between file descriptors. This reduces the context switches to two, and requiring only two DMA copies. [63]

The approach of the designs of different user space network stacks share the key principle of bypassing the kernel and shortening the data path. The traditional networking stack can be expressed as two layers, with a packet I/O layer and a TCP/IP stack layer. Chen et al. [8] divide kernel bypass techniques to three categories based on the layer the kernel bypass mechanisms are located in: user space packet I/O, user space TCP/IP stack, and hybrid network stack.

A user space packet I/O technique eliminates packet transfer between the NIC and kernel. Examples of this are Netmap [60], which exposes the DMA memory of the NIC to user space, and Data Plane Development Kit (DPDK) [35], which constantly polls the NIC memory for messages. Netmap allows applications to access the NIC directly by using a shared ring buffer with the NIC. Netmap still utilizes the kernel NIC driver, whereas DPDK additionally runs the NIC driver in user space. [8]

Core libraries and drivers of the DPDK are Poll Mode Drivers (PMD), which means that it uses one full CPU core of resources per NIC to poll the NIC for messages [1]. The NIC can also be switched to listen to events if no packets are expected, but upon an event, poll mode must be used again to receive the packets. DPDK uses *hugepages* and maps the NIC memory to user space. [22]

An user space TCP/IP stack is based on a user space packet I/O library, and further bypasses the kernel by running an user space TCP processing thread along with user level Sockets. However, custom user space implementations of the TCP/IP stack may inevitably be outdated or limited compared to the kernel TCP/IP stack. A hybrid approach combines the previous approaches and ports the kernel TCP/IP stack into a user space packet I/O framework. [8]

Other technologies such as TCP Offload Engine (TOE) have been developed to move the full TCP/IP stack processing to the NIC, thus freeing CPU cycles required for TCP processing. Still, with TOE, only the TCP/IP protocol stack is offloaded while data itself is still moved through the kernel, and thus the processor load is still significantly affected [12].

Next we examine Remote Direct Memory Access (RDMA), which inherently bypasses the kernel via hardware offload, and also performs zero copy operations to kernel space.

3.3 RDMA

Remote Direct Memory Access (RDMA) is a technology to transfer data remotely between the memory of different machines without the involvement of an operating system. RDMA uses a specialized Network Interface Controller (NIC) to transfer the data to or from memory without the need of a kernel call. This type of hardware is named RDMA enabled NIC, or RNIC. To send a message, the CPU commands the NIC directly. [27]

As explained in Section 2.3.2, the Verbs API was introduced as the Open Fabrics Enterprise Distribution (OFED) high-level interface to the Infiniband technology. In 2006, OpenFabrics Alliance (OFA) generalized OFED to in-

clude all RDMA standards: Infiniband, RoCE, and iWARP. [38] A Verb is defined as an operation for the RNIC to execute [58]. The Verbs represent a similar set of operations as Create, Read, Update, Delete (CRUD). The RDMA Verbs are write, read, send, and receive. They can be classified to memory Verbs and messaging Verbs. The memory Verbs consist of reads, writes and atomic operations, and specify the remote address, bypassing the recipient CPU. The messaging Verbs consist of the send and receive Verbs, which involve the recipient CPU by writing to an address specified by the recipient in a previous message. [25]

As explained in Section 3.1, Sockets copy data to send or receive between user and kernel space buffers per each operation. Verbs do not use buffering, instead the hardware RNIC or HCA transfers data directly to the wire from user space and vice versa, providing zero copy. Whereas Sockets require kernel interaction during data transfer, Verbs interact directly with the Infiniband HCA or RNIC, providing kernel bypass. [38]

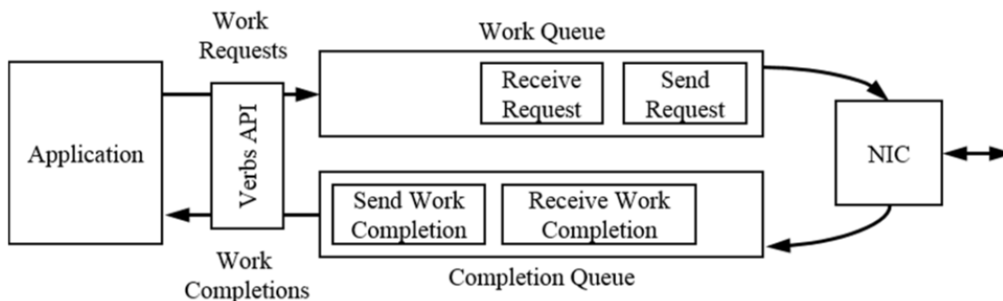

Figure 3.1: RDMA communication with queues through Verbs API. [22]

RDMA operates using Queue Pairs (QP), which consist of Work Queues (WQ) and Completion Queues (CQ) [22]. Operations to these queues are posted using the Verbs API. A Queue Pair represents a communications endpoint, comparable to a Socket. Memory regions are specified for communications data storage for local and remote network hardware, so that they can read from and write to this region. [13]

Figure 3.1 shows the usage of queues using the Verbs API between the application and the NIC. RDMA operations are initiated by creating elements into the Work Queue (WQ) in memory. An element signaling the completion of a Verb may be written to the CQ. The destination host will then read the CQ to determine a completed Verb. The Verb can be sent without a CQ entry by setting a flag in the request, and in this case the completion must be detected by other means. [25]

It can be seen as a disadvantage of RDMA communication that an incoming message must be explicitly handled on the receiver side [37]. To check for completed operations, the CQ can be polled periodically, or via a file descriptor triggering events when operations are completed [29].

The performance of Verbs differ based on them being inbound or outbound Verbs. Memory Verbs and send Verbs are outbound at the sender and inbound at the destination, while receive Verbs are always inbound. The receive Verbs are slower due to a DMA into the CQ, making the read and write Verbs preferred. [25]

RDMA transfers are either reliable or unreliable. Additionally, they are either connected or unconnected. Reliable transfer uses NIC acknowledgements, while unreliable transfers do not. Still, unreliable transfers very rarely drop packets in modern RDMA implementations such as Infiniband, which uses a lossless link layer. Unconnected transfers are also called datagrams, and connected and unconnected transfers are similar to TCP and UDP. Different RDMA transfer types include Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). Of these, UC does not support read Verbs, and UD does not support memory Verbs. [25]

The use of RDMA requires modifications to a TCP/IP based application. Programming for RDMA Verbs needs to take into account the creation of the data structures for queue pairs. In addition, they need to get the connected RDMA devices and allocate the messages with the device, with a send buffer. Completion queue polling must also be implemented. [23] Further, the memory buffers currently in use by RDMA communication cannot be reused before a work completion is confirmed. Rewriting applications for RDMA use may therefore prove expensive. [67]

Some solutions, such as *rsockets* [23] have been presented to enable the usage of RDMA via a Socket-like API. Further, other solutions that provide native Socket API for RDMA use, removing the need of any modifications to the source code have emerged, such as Mellanox VMA (NVIDIA/Mellanox Voltaire Messaging Accelerator) [45] and *vSocket* [67]. Both vSocket and VMA intercept the Socket API calls and translate them to Verbs via a preloaded library [67]. Using such a translation or Socket-like API would reduce the need for programmers to actually learn RDMA [23].

We next explore two existing Ethernet RDMA specifications, before looking into RDMA implementation in containerized clusters.

3.3.1 iWARP

iWARP is a protocol set consisting of multiple protocol specifications by the Internet Engineering Task Force (IETF). It provides an RDMA imple-

mentation on top of existing TCP or SCTP (Stream Control Transmission Protocol) internet protocols. [11]

iWARP uses its own Verbs interface similar to Infiniband Verbs at the top layer, and similarly uses Queue Pairs (QP). Additionally, a TCP connection is established between the endpoints at a lower layer. The iWARP Verbs require buffers to be locked before data transfer. [57]

iWARP consists of three protocol layers built on top of TCP/IP, and a Verbs API layer at the top layer connecting the applications to the protocol stack. The protocol layers consist of RDMA Protocol (RDMAP), Direct Data Placement (DDP), and Marker PDU Aligned Framing (MPA) layers. The RDMAP layer coordinates RDMA transfers and passes them down to the DDP layer, or receives them with a message header specifying the local destination. The DDP layer places data directly into application buffers without additional copies. The DDP layer additionally splits messages to fit into transport frames, and reassembles received messages. The MPA layer is responsible of keeping the received data in correct order with markers and a checksum. [12]

Some proposals have introduced partial software implementations of iWARP, such as the papers by Dalessandro et al. [12] [11], where the authors enable the client without an RNIC to communicate with a server equipped with an RNIC through iWARP.

3.3.2 RoCE

RDMA over Converged Ethernet (RoCE) is a protocol that enables the use of Infiniband protocols on existing Ethernet infrastructures [30]. RoCE implements the Infiniband layer in the same stack area as iWARP [40]. RoCE standard is specified by Infiniband Trade Association (IBTA) and it utilizes the Open Fabrics Enterprise Distribution (OFED) Verbs software interface [28]. The first implementation of the standard was done by Mellanox Technologies, providing 100GB/s throughput, and is now supported by multiple network adapter vendors [40].

Instead of the complex network layers of iWARP, RoCE uses an RDMA protocol purposefully built for Ethernet [40]. RoCE uses the upper layers of Infiniband architecture, including the transport layer, on top of Converged Enhanced Ethernet (CEE) [27]. CEE is a set of standards for Ethernet in data centers by Data Center Bridging task group. CEE enhances Ethernet by providing link level flow control and improved congestion control, aiming to enhance all data center traffic by lossless communication. [5] RoCE replaces Infiniband physical and link layers with Ethernet correspondents, while still using Infiniband Verb implementation. RoCE traffic does not

carry an IP header, and thus cannot be routed beyond subnets using traditional IP routers. RoCEv2 is an extension to the protocol, providing the IP header by replacing Infiniband Global Route Header (GRH). The traffic is encapsulated in an UDP header. [1]

RoCE can be implemented in both hardware and software. The software implementation of RoCE is a Linux driver called Soft RoCE. The driver still requires a hardware RoCE implementation in the connected machine. [28] When using soft RoCE, packet processing is again implemented in the operating system kernel, which affects latency and CPU usage [1].

3.4 RDMA in containerized clusters

Existing container overlay networks subject the traffic to a deeper software stack to achieve portability in the expense of performance. To achieve more efficient inter-container communication, the key is to take advantage of the fact that containers are essentially processes, by utilizing inter-process communication (IPC) on hosts and offloading network operations to hardware. [1] A de facto standard for the usage of RDMA in container networking is still yet to be defined. Still, multiple proposals on the topic have been presented.

As RDMA requires hardware support, its cloud-based implementation proves problematic. A cluster needs to include three properties vital to container networking: isolation, portability and controllability. Using an RNIC interferes with isolation, as containers have to use the host namespace to interact with the RDMA NIC. Portability is additionally limited, as the RNIC may have different addressing between hosts. As network processing is offloaded to the RNIC with RDMA, it is difficult to modify control plane states such as routes, or to control the data path, when data is placed directly into memory. To achieve all required properties, the networking is typically fully virtualized via a software switch, therefore presenting the need to virtualize RDMA communication. [29]

A few solutions exist for cluster pods to establish RDMA connection between each other, such as enabling the use of an Infiniband device via Kubernetes Device Pods in addition to the default container network [6]. Containers in these pods need the installation of OFED drivers for RDMA use. This type of setup, while effective, is still heavy to maintain with driver management and the assignment of Device Pods.

A Kubernetes plugin has been designed by Mellanox [43] to virtualize RNICs for containers using Single-Root I/O Virtualization (SR-IOV). However, use of plug-ins such as SR-IOV lacks the ability of control plane management, as in, reconfiguration is required when migrating containers. Further,

a virtualized interface is shared across a pod, preventing isolation for multiple containers within it. [34]

Abranches et al. [3] proposed a system that utilizes shared memory channels for inter-container communication. These shared memory regions would function in local communication or be synchronized via RDMA. The implementation relies on a separate broker part, which communicates with Kubernetes API, the shared memory, and the nodes. The authors presented a proof-of-concept prototype, while a full Kubernetes implementation was left for future work.

Another proposal by Kim et al. [29] aims to bring an RDMA container orchestration solution to clusters. In their solution, named Freeflow, an intermediary router container accesses the NIC hardware on each node, which provides better controllability than the usage of e.g. SR-IOV. Freeflow authors provide an open source prototype, which we next examine in further detail.

3.4.1 Freeflow

Freeflow, developed by Microsoft Research and Carnegie Mellon University, is a prototype of a software RDMA virtualization framework for containerized clouds. FreeFlow addresses the problem that RDMA NIC access does not support isolation or portability, two key advantages of containerization. According to the authors, Freeflow achieves close to bare-metal RDMA performance. [29]

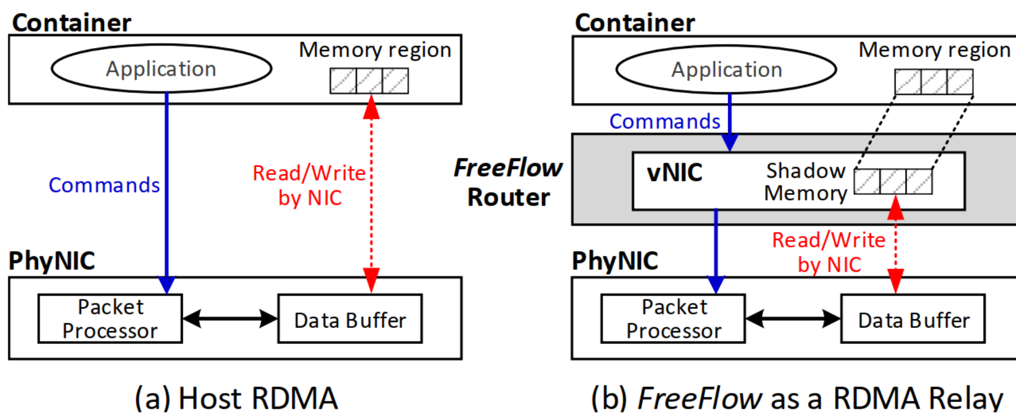

Figure 3.2: Freeflow router as an RDMA relay for containers, making the physical NIC transparent to applications. [29]

As done in TCP/IP virtualization, the authors deem the solution of RDMA network virtualization is the use of a software switch. A software switch provides isolation and portability by performing all addressing and routing functions, and leaving only the packet delivery for the physical network. [29]

The main component of Freeflow is its router container, which is the only component the RNIC performs memory operations on. This is illustrated in Figure 3.2. Freeflow is completely transparent to the application, intercepting the Verb calls between the application and the physical NIC. The transparency presents a design challenge regarding RDMA use, as the RNIC can modify memory silently. This is solved by the authors by noting that containers may be treated as processes, and thus resources such as memory and file descriptors can be shared between Freeflow components. Thus the modifications by the physical NIC may be automatically visible inside the application containers. [29]

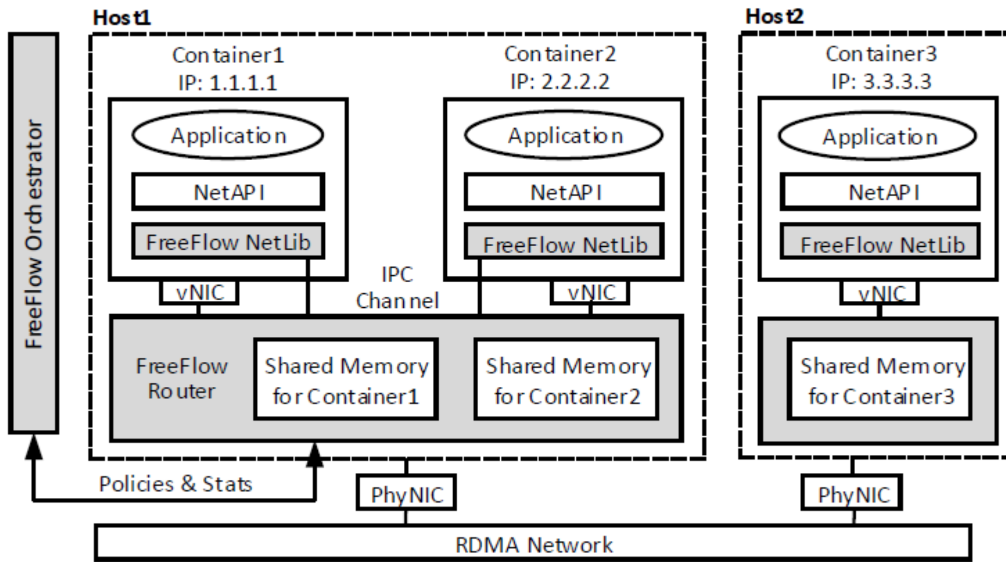

Figure 3.3: Freeflow architecture, showing the different parts of Freeflow added to the cluster. [29]

Freeflow architecture contains two additional integral parts, as illustrated in Figure 3.3. A single Freeflow orchestrator is responsible for configurations and monitoring, as well as maintaining centralized memory maps. A Freeflow network library (Netlib) is located in each application container, and is the part that makes Freeflow transparent to applications. It does this by pre-

senting the standard Verbs API to the application, including virtual QPs and CQs, and then coordinating with the Freeflow router to forward the requests. This essentially virtualizes the NIC for the application container. The Freeflow router is a single container run on each node, coordinating with the Netlib on all containers on the node. [29]

As the Freeflow Netlib intercepts all Verbs calls, and the calls still must travel through the Freeflow router, the authors propose two methods to efficiently forward the calls between the two Freeflow components. The first method is to forward the Verb request to the Freeflow router using the router's file descriptor instead of the NIC's. The Freeflow router then forwards the request using the actual NIC file descriptor. This method incurs additional latency, and as a low latency alternative, the second method proposed was for the Freeflow router to use a full CPU core to constantly check the shared memory for requests by the Freeflow Netlib. [29]

Freeflow was tested by the authors using RDMA performance benchmarking tools, and additionally the Socket to Verbs API translation tool *rsocket*. The authors determined that the translation induces overhead, causing worse throughput compared to bare metal TCP/IP, but outperformed even bare metal by up to 98% lower latency on small message sizes. With the RDMA performance benchmarking tools, Freeflow caused only up to 1.5 μs additional delay compared to bare metal RDMA. [29]

The Freeflow prototype is open-sourced on GitHub [47], and its TCP implementation will be tested in this paper.

Chapter 4

Experimentation setup

In this chapter, we explain our test setups and tools to perform TCP/IP and RDMA benchmarking, and evaluate the results in Chapter 5. The purpose of these measurements is to see the possible performance impact of RDMA, and to evaluate if TCP/IP-based applications can be efficiently used as-is with RDMA. Initially, we were aiming to measure the performance of RDMA within the Kubernetes cluster and compare to TCP/IP benchmarks on both bare metal and container network, but ultimately failed to implement it properly within the container setting. Thus, RDMA benchmarks remain in bare metal. Instead, we additionally included the measurements using a Socket to Verbs API translation tool, Mellanox VMA.

We include Kubernetes cluster benchmarks of TCP/IP with the Flannel container network, and note its overhead compared to bare metal. In addition, we test the TCP branch of Freeflow within the container network to see its acceleration performance compared to Flannel and its overhead compared to bare metal TCP/IP. This can give insight if its RDMA implementation would also perform with low overhead. For RDMA testing, we benchmark with dedicated RDMA performance tools, and compare to the VMA framework, seeing the efficiency of Socket to RDMA Verbs translation.

4.1 Hardware

In the testing implementation, we use two different setups. At first, a Raspberry Pi 4 cluster of three devices was used purely for initial benchmarking. However, the Raspberry Pi 4 does not support technology such as RDMA due to its incompatible NIC. We had access to two Nokia Airframe OpenEdge servers as the second test setup to perform RDMA benchmarking.

4.1.1 Raspberry Pi 4 setup

As our first test setup, we use three Raspberry Pi 4 computers running the Ubuntu Server 20.04.2 LTS operating system. Each contains a quad-core ARM Cortex-A72 @ 1.5 GHz processor with 1 MB shared L2 cache, and 4 GB RAM.

The computers are directly connected to a D-Link GO-SW-5G switch, and an Asus RT-N18U router is connected behind the switch as the main gateway. The Ethernet cables used are Cat.5e between the router and switch and Cat.6 between the switch and the Raspberry Pi computers.

One of the computers functions as the master node, and the other two as worker nodes. We use the worker nodes for our measurements and the master node only for orchestration purposes, to minimize CPU usage interference from cluster orchestration tasks.

4.1.2 Nokia AirFrame server setup

Our second test setup consists of two Nokia AirFrame OpenEdge [51] machines connected directly with 25GBps Direct Attach Copper (DAC) cables.

The servers are both equipped with Mellanox ConnectX-5 MT27800 NIC, and Intel Xeon Gold processors. One machine has Intel Xeon Gold 6230N @ 2.30GHz CPU with 40 logical processors, 20MB L2 cache, 27.5MB L3 cache, and 96GB RAM. The other machine has Intel Xeon Gold 6212U @ 2.40GHz CPU with 48 logical processors, 24MB L2 cache, 35.75MB L3 cache, and 192GB RAM. The ConnectX-5 cards are Intelligent RDMA (RoCE) enabled NICs [42].

The operating system version in the first machine was CentOS 7.9.2009 (Core), and in the second machine CentOS 7.6.1810 (Core), with kernel versions `3.10.0-1160.42.2.el7.x86_64`, and `3.10.0-957.el7.x86_64`, respectively. ConnectX-5 EN (Ethernet) drivers were used for the NICs, with Mellanox OFED driver version used in both machines being `MLNX_OFED_LINUX-5.3-1.0.0.1`.

In our Kubernetes deployment, both servers functions as worker nodes, while one also performs the master node functions. The master node tasks should incur little effect on our measurements, as all CPU cores will never be used to capacity in our testing.

When testing RDMA, additional performance gains could possibly have been achieved by tuning the BIOS for high performance via `lowlatency` kernel headers, but we used the `generic` package.

4.2 Software tools

In this section, we explain the two software tools integral to our tests. Lightweight Kubernetes was selected as our cluster software, and Mellanox VMA library is integral for our test on Socket to Verbs acceleration.

4.2.1 K3s, Lightweight Kubernetes

As our container orchestrator, we use lightweight Kubernetes [9], abbreviated as K3s. K3s was originally developed by Rancher, and is now maintained by Cloud Native Computing Foundation (CNCF). It is packaged in a single binary under 40 MB in size with reduced dependencies [56]. It is better suitable for constrained hardware, such as Raspberry Pi computers, which we use on one of our test setups, and supports its ARM-based processor. We use K3s version 1.20.5.

As the Container Runtime Interface (CRI), we use `containerd`, which is the default in K3s. Docker was used for image building and pushing to a private registry, from where they could be pulled into the test cluster, or imported directly to the cluster.

As the base image in the Raspberry Pi containers, we use Alpine. It is the most lightweight Linux distribution, with a size of only 5 MB. For the Nokia Airframe servers, we use Centos 7.9 as the base image and install tools with `yum` package manager. We compile the necessary tools such as wrk and use multi-stage building with Docker to package the applications in small images. We then upload the images to the private registry, from where Kubernetes pulls them.

As our container network, we use Flannel CNI as it is the default network in K3s. It was additionally shown to have the better performance of overlay networks, especially on small message sizes [26]. Two different networking modes are available for the Flannel CNI. The default mode is VXLAN, which is its overlay networking mode, and the other option is `host-gw`, its underlay networking mode.

On the Nokia AirFrame servers, we set Flannel to use host-gw backend using the `--flannel-backend=host-gw` flag for starting k3s, as there was no connection with the default VXLAN settings between nodes. This is likely due to some problem with UDP, possibly routing, and the VXLAN using UDP encapsulation. The host-gw setting also provides performance improvements, as IP routes to subnets are created via host machine IP addresses, and VXLAN encapsulation is not used. The use of this setting is possible on our test setups, as we have a direct Layer 2 connection between

the nodes.

Our clusters were configured such that each server application in a pod was attached to a ClusterIP service, which the client would connect to during tests.

4.2.2 Mellanox VMA

Mellanox/NVIDIA Voltaire Message Accelerator (VMA) [45] is a dynamically linked user space library able to translate Socket calls to Verbs calls. We can utilize this to compare RDMA communication directly to TCP/IP by using the same Socket-based measurement tools. We included the VMA library installation along with our OFED driver package.

The VMA library transparently presents the standard Socket API to the application, and implements the operations on RDMA Verbs API, fully from the user space [44]. The VMA internal thread uses a CPU core to poll the Completion Queue (CQ) constantly for messages [53].

The command `LD_PRELOAD`, prepended to another command, *preloads* a library before any other when running a program, modifying existing library behaviour by replacing functionality. This allows non-invasive modifications to applications, in our case the interception of Socket calls. Here, the preload enables VMA library to handle the Socket calls in user space, and translate them to RDMA Verbs. In our tests we run the different tools prepending the commands with

```
LD_PRELOAD=libvma.so
```

The output of VMA provided a warning about memory allocation, and according to the instructions in the warning message, we set the machines to use system hugepages with the value 800 set to

```
/proc/sys/vm/nr_hugepages
```

with an additional very large `shmmax` setting.

4.3 Testing tools

Various testing tools were used for for traffic generation and measurement in different test cases. Some of our tests include CPU usage data during the test run. We measure CPU usage using the Linux `top` tool, by getting a breakdown of CPU usage percentage each second. The tool gives, among others, the CPU usage percentage of user space, kernel space and software

interrupts, which we will focus on. Comparing CPU time spent in kernel space and software interruptions versus user space shows the benefits of kernel bypass.

Our test parameters are latency, as in usually round-trip time (RTT) divided by half for one way latency, and throughput. RTT consists of sent packet and its response, and by halving it we have the one-way latency. The throughput similarly is measured one way, from server to client.

We started our benchmarks by using traffic generation to a Nginx web server, but later moved on to pure network performance testing tools and focus most of the results on the Sockperf tool. The wrk tool, for example, reports full RTT for latency, whereas Sockperf reports one-way latency. Additionally, Sockperf worked in all our test cases and is designed for measurements with low overhead, and was therefore selected as the main testing tool.

4.3.1 wrk and Nginx

`wrk` [69] is a latency and bandwidth measuring tool from generated HTTP traffic, supporting the use of multiple threads and connections. HTTP communication consists of requests and responses. The performance measured by wrk is two-sided. The wrk tool sends requests to the server, which responds with the requested data payload. The round-trip time of sent request and received response is the measured latency. We use a script to format the results with detailed latency percentile statistics in a comma-separated format for graphing.

We set up a Nginx [20] web server to serve files of different sizes, which the wrk tool requests during our testing. The files were created by using the `dd` command to fill files with null bytes from */dev/zero*. For example, we created the 1 byte file using the command:

```
dd if=/dev/zero of=/usr/share/nginx/html/1B.bin bs=1 count=1
```

We use the Nginx web server unmodified, with `sendfile` enabled. This skips an additional buffer copy, and uses a file descriptor instead, sparing CPU cycles. `worker-processes` is set to `auto`, creating one worker process per CPU core.

4.3.2 iperf3

Iperf3 [18] is a network bandwidth measurement tool ideal for determining the maximum available bandwidth of a network. The recent version is named iperf3, which is a redesign of previous `iperf` versions. Iperf3 is developed by ESnet and Lawrence Berkeley National Laboratory.

We use the tool mainly to perform comparisons to the throughput data gathered by the Sockperf tool. We used iperf3 version 3.7 on the Raspberry Pi setup, and version 3.6 on the Nokia Airframe setup.

Iperf3 normally works by sending an array of bytes for a set time. We use the `-l` option in our tests to determine the size of the array to send, similar to our other tests. We measure one way throughput, and do not see the need to test reverse direction, as the network works identically for both directions.

Iperf3 uses Nagle's algorithm [49] by default, combining multiple small packets that are under the MTU in size and sending them as one large packet. The `no-delay` option disables this algorithm.

4.3.3 Sockperf

Sockperf [41] is a TCP and UDP network benchmarking tool by Mellanox. The tool provides the options for both throughput and latency measurements for a certain run time and message size. Sockperf is advertised to perform latency measurements for each discrete packet with very low overhead, by counting CPU ticks. Its output provides a detailed latency breakdown with multiple latency percentiles without affecting the benchmark. The measured one-way latency is given by dividing the round-trip time by two [52].

We installed Sockperf on the containers using the native package manager, and for bare metal use on the AirFrame setup, the tool was included with the Mellanox OFED driver package. We used Sockperf version 3.7-7 on the Airframe setup, and version 3.6 on the Raspberry Pi setup.

The tool tests the performance of UDP by default, and TCP by setting. We will test the TCP protocol as with other tools. Tests include under-load, ping-pong, playback, and one-way throughput. Of these we will omit under-load test and playback test, which sends predefined traffic, as again we are only interested in raw network performance. We set up a Sockperf client and a server for both endpoints.

The maximum message size the sockperf server can receive is 1048575 bytes, so we will use that size for the final megabyte size.

4.3.4 RDMA performance tools

The RDMA `perftest` tools are a collection of bandwidth and latency benchmark tests, written over `uverbs`. They are Open Fabrics Enterprise Distribution (OFED) performance tests, available from `linux-rdma` GitHub [36].

These tools can be used to gather a base benchmark of the performance of the RDMA Verbs send, write and read.

The performance tools use a CPU cycle counter, which measures time stamps without context switches. Half RTT is reported as one way latency, and the client measures the unidirectional bandwidth from server to client.

4.4 Test cases

We have four different test cases in which we run various benchmarks both on bare metal and on Kubernetes cluster. All tests are ran from client to server between two machines. First, we measure the performance of traditional TCP/IP communication. Then, we benchmark bare metal RDMA performance. Third, we compare bare metal VMA to the RDMA benchmark. Finally, we test the Freeflow TCP prototype in a cluster.

MTU is set to default 1500 bytes for all TCP/IP tests. The packets larger than the MTU will be split. The TCP protocol may interfere with the results on small message sizes by delaying e.g. one byte transfers to combine multiple messages to one, as per *Nagle's algorithm* [49].

To get an overall view for latency, we run tests for several different file or data sizes. Each test run uses a different message size, varying from 1 byte to a megabyte. The total amount of different data sizes varies per each testing tool used, generally from 13 to 16. Our latency tests were generally ran for three minutes at a time per data size. The throughput tests were generally short, running for 10 seconds per data size.

To gather the test data, we save the console output of a test run to a file and build a graph from several runs using these files.

4.4.1 Test case 1: TCP/IP benchmark

We perform the base benchmark for our other test cases by testing both test environments using wrk to Nginx load generation. This is tested both on bare metal and in Kubernetes Flannel network. We compare our two test environments and decide to focus our other test cases on the Nokia Airframe test setup.

The wrk tool reports latency as the full RTT in contrast to our other testing tools. Therefore other test cases compare results to a similar Sockperf tool benchmark. The `sendfile` option is set to on in the Nginx web server.

In this test case, we additionally determine the difference in performance between the usage of the Flannel host-gw backend setting compared to the default VXLAN setting. This is done on the Raspberry Pi test setup, as we were forced to use the host-gw backend on the Nokia Airframe setup.

4.4.2 Test case 2: RDMA performance benchmark

This test uses the performance test tools explained in Section 4.3.4, giving a benchmark for bare metal RDMA communication. Each tool will give us information of the performance of different types of RDMA Verbs. The testing is performed on the Nokia Airframe test setup, which is equipped with RNICs.

In each test, we use Reliable Connected (RC) type of connection, which is more comparable to TCP than the other types. MTU of the link in these tests is set to 1024 bytes. The transport type is Infiniband, but the data exchange method is Ethernet. Essentially we test run RoCE.

With the `-a` flag, data sizes from 2 bytes to 8 megabytes are tested, each with 1000 iterations. We focus on the results under 1 megabyte size, as with other tests.

One example of a command used for the write Verb on the client side is

```
ib_write_lat -d mlx5_2 -a -F 15.15.15.5
```

where the address corresponds to the address set to the server RDMA interface `mlx5_2`. The IP addresses for the machines are set to 15.15.15.5 and 15.15.15.6.

4.4.3 Test case 3: Mellanox VMA bare-metal TCP RDMA acceleration

In this test case, we run the Sockperf and iperf3 tools with the VMA library preloaded on both client and server to determine the efficiency of Socket acceleration towards RDMA. The testing is performed on the Nokia Airframe test setup.

To test VMA in our Kubernetes cluster, we would have needed to install the OFED drivers on the containers. We tried this, but we failed to install the correct kernel drivers required for the OFED driver installation. We could have used Kubernetes plug-ins such as SR-IOV here, but our limited time using the test setup lead to us only testing VMA on bare metal setting. Still, the usage of plug-ins such as SR-IOV very likely lead to near bare metal performance. We leave the cluster testing of this tool to future work.

We use the host IP addresses of the machines, without opening additional interfaces for the NICs, which support four interfaces.

4.4.4 Test case 4: Freeflow TCP in cluster

In this test case, we determine the efficiency of the TCP branch of Freeflow in its advertised acceleration of a container network towards bare metal. This testing is, similarly to the two previous test cases, performed on the Nokia Airframe setup.

We first attempted to set up the full Freeflow RDMA implementation by following the steps on the Freeflow GitHub page [47]. However, each Freeflow router required compilation and the installation of the correct RDMA OFED libraries and drivers. We failed to compile the Freeflow router, and our time-limited access to the testbed finally prevented us to get to the root of the problem.

We then moved on to test the Freeflow TCP branch [48], which should use the Socket API in communication. The Freeflow TCP branch installation steps worked simply by using the image `freeflow/freeflow:tcp` for each Freeflow router in our cluster. The flags for host network and privileged mode needed to be set to true.

We have limited information of the TCP branch of Freeflow outside the Readme file. The Readme file advertises acceleration to the same latency, throughput and CPU overhead as bare metal Socket. We assume the functionality of Freeflow TCP bypasses the Flannel CNI via the traffic interception by the Freeflow router, and performs networking transparently over bare metal.

We deployed the TCP implementation of Freeflow into our Kubernetes cluster according to the information in the Freeflow TCP branch Github page [48]. One pod per node acts as the Freeflow router, deployed into the host network. Then, pods are deployed to the nodes, sharing `libfsocket.so` file with the router using volumes. The applications ran on the pods use `LD_PRELOAD` for the shared file to intercept their Socket calls to certain networks. We used a `10.40.0.0/13` network mask for the environment variable `VNET_PREFIX` so that Freeflow will intercept both the Kubernetes and the Kubernetes service virtual networks, in subnets 10.43.x.x and 10.42.x.x. For the Freeflow router, the `HOST_IP_PREFIX` was set according to the host IP address of the server machines.

Each node contains a Kubernetes ClusterIP service exposing the required ports to the cluster. A Freeflow router is deployed to each node. This router mounts the `freeflow` folder and shares the shared object (`.so`) file for other services and pods. Each pod using Freeflow also mounts this same folder. The `yaml` file used when deploying a Freeflow router pod for each node is shown in Appendix B.

Chapter 5

Results

Our measurements were graphed using Python and its graphing library, `matplotlib`. For latencies, we measure maximum, minimum, mean with standard deviation, 99th and 99.999th percentile latencies where applicable. The CPU usage graphs show both client and server node CPU usage in this order for each data size test.

Generally, we present the latency or throughput values on the Y-axis, and in all graphs, the X-axis combines the test runs performed for each data size. We use logarithmic scale on the Y-axis in most measurements.

5.1 Test case 1: TCP/IP benchmark

Our first evaluations used the wrk tool for HTTP traffic generation to Nginx server on the Raspberry Pi setup. We first tested how a large amount of simultaneous connections from the wrk tool affect both the server and client. The results were that the CPU usage increases quickly when the amount of connections are increased, and the latencies start to have large variations, as could be seen from the standard deviation growing larger when multiple connections are introduced. As we test the network performance rather than the performance of Nginx server itself, we will use only a single connection for latency measurements.

Figure 5.1 shows the overhead coming from the usage of Flannel VXLAN mode compared to Flannel host-gw mode. There is an approximate 25% increased latency across all results when using VXLAN. We use the same host-gw setting in both hardware setups for more consistent comparison and to eliminate this additional overhead in our results.

We note the latencies for the maximum and 99.999th percentile are very high compared to other latencies. This may be because of the interruptions

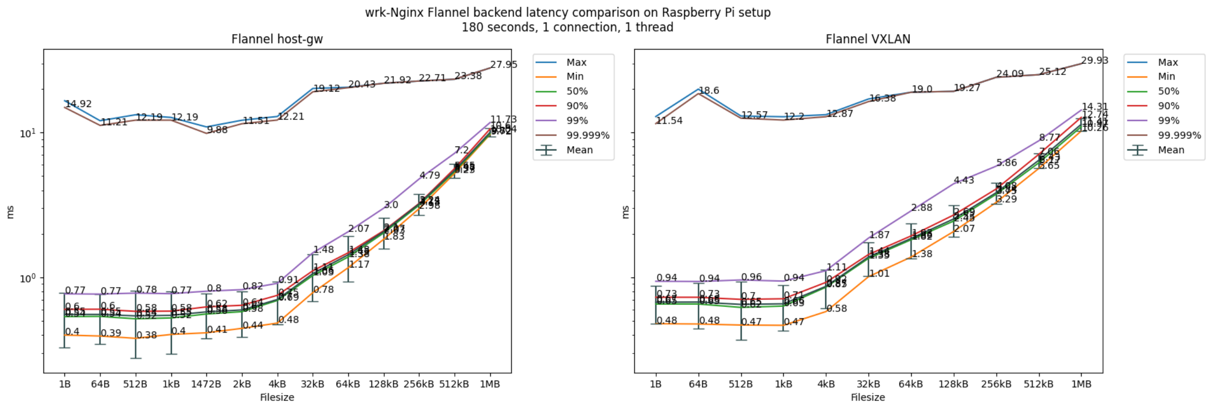

Figure 5.1: Latency comparison of Flannel host-gw and VXLAN settings on the Raspberry Pi setup

in the processor by other applications on the Raspberry Pi, and would likely not be as noticeable on our high-performance server setup.

In Figure 5.2 we compare the wrk-Nginx latencies of both test setups using bare metal and Flannel host-gw. The latencies are measured in milliseconds. For mean latencies, the overhead caused by Flannel can be determined as around 30 microseconds on the Nokia Airframe setup, and around 100 microseconds on the Raspberry Pi setup.

In the bare metal test, Raspberry Pi setup shows 99th percentile latencies 3 times and mean latencies 4 times larger than AirFrame setup on file sizes smaller than 4 kB, and up to 15 times larger latencies on large file sizes. Comparing the Flannel latencies on both setups shows similar relation with a slightly smaller difference of up to 13 times larger latency on large file sizes on Raspberry Pi setup.

The differing test setups have considerably different CPU frequencies and core amounts. Additionally, the Intel Xeon processors have considerably larger caches than the ARM Cortex. The processor along with the more efficient NIC contribute to the observed performance difference. We note the large difference between the test setups, and focus further measurements on the higher performance Airframe setup, where also RDMA can be evaluated.

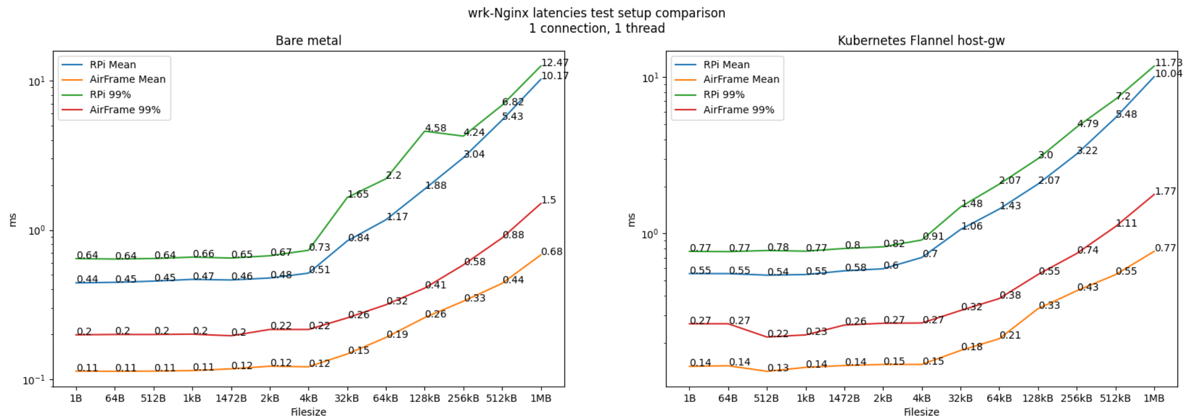

Figure 5.2: Comparison of latencies on both test setups using wrk and Nginx on bare metal and Flannel

5.2 Test case 2: RDMA performance benchmark

In this test case, we compare the latency and throughput of bare metal RDMA to bare metal TCP/IP. The RDMA operations tested are send, write and read. We did not gather CPU usage data from this test.

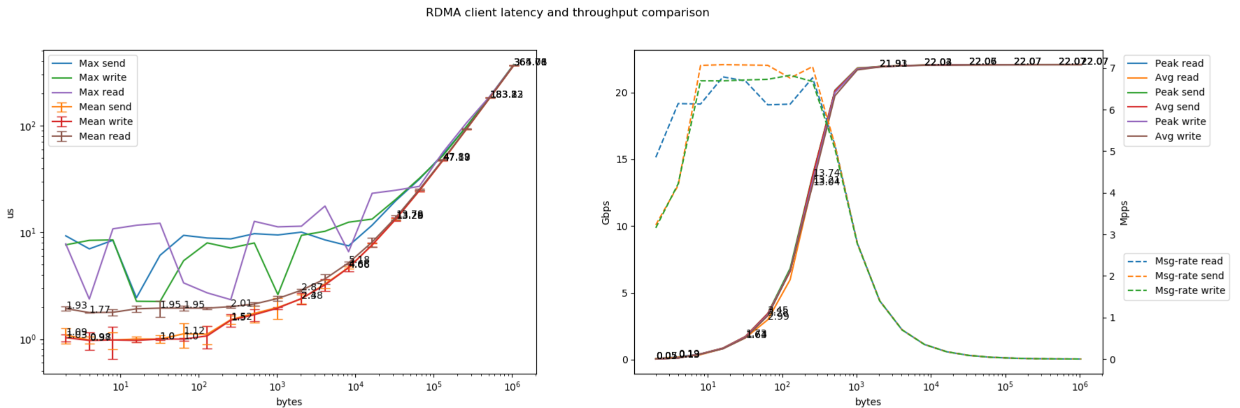

Figure 5.3: Client side latency and throughput measurements of RDMA Verbs

The latency graph on the left of Figure 5.3 shows the maximum and mean latencies of each RDMA operation across data sizes up to 1 megabyte. The

latencies are measured from client side. It can be seen that the latencies for all RDMA operations are very low across all message sizes. The latencies are as low as a few microseconds for small messages, and rise in magnitude only after 10 kilobytes, with a latency of 365 microseconds for a megabyte transfer. There is generally a very small standard deviation of the mean latencies, as specified by a line per each data size. The graph shows the read Verb having around double the latency of other Verbs on small messages. This is in line to the other Verbs generating exactly one RDMA message, while RDMA read generates two [59].

The throughput graph on the right of Figure 5.3 shows the throughput peaking immediately when reaching the MTU of the link, which on the RDMA tests is 1024 bytes. All Verbs give similar throughput.

On the right axis, millions of packets per second (Mpps) are measured in dashed line, while solid lines show throughput in Gbps. A very high message rate is achieved even for the smallest message sizes, as the messages can be sent immediately without filling a buffer first. The maximum throughput is around 22 Gbps of the 25 Gbps link, in line with maximum TCP/IP throughput as seen on Figure 5.8.

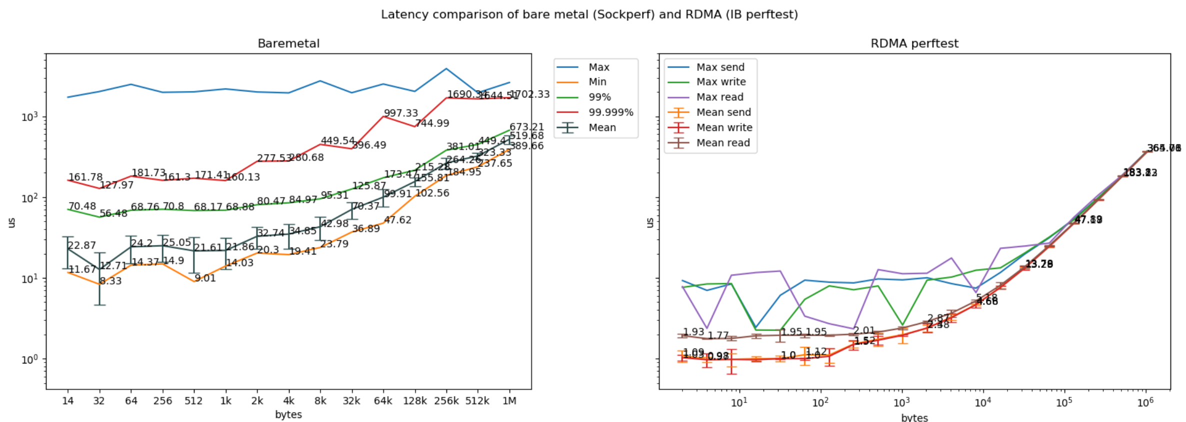

Figure 5.4: Bare metal Sockperf and RDMA latency comparison

Figure 5.4 presents the latencies of the Sockperf benchmark on bare metal compared to the RDMA benchmark. Both graphs are on the same scale, measured in microseconds. An order of magnitude difference to even the minimum latencies on TCP/IP can be seen on the RDMA mean latencies. Only the maximum latencies of the RDMA results measured reach some of the minimum latencies of bare metal. The RDMA test shows little devia-

tion of latencies, whereas there are large, an order of magnitude differences between maximum and mean latencies on bare metal. The large latency variations on TCP/IP can be attributed to variations in CPU usage. RDMA performs on microsecond level on small data sizes.

5.3 Test case 3: Mellanox VMA bare-metal TCP RDMA acceleration

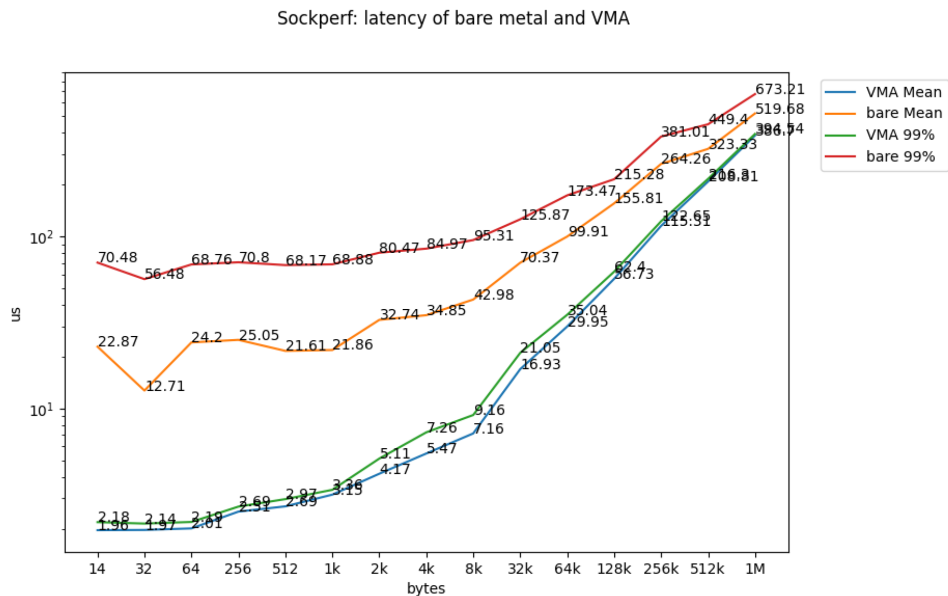

Figure 5.5: Bare metal to VMA latency comparison

Figure 5.5 shows the clear difference in latency using the Sockperf tool with VMA compared to bare metal, with VMA test showing an order of magnitude lower latencies on small data sizes. The latency difference is noticeably lower on the one megabyte data size. We can see that the VMA latencies remain relatively constant, while the 99th percentile latencies are largely 2 to 3 times higher than the mean latencies on bare metal.

Figure 5.6 shows the comparison between RDMA performance tests from Section 5.2 and VMA. It can be determined that the Socket acceleration using this tool is very efficient. VMA performs very close to native RDMA with a difference of a few microseconds on small data sizes. The difference increases

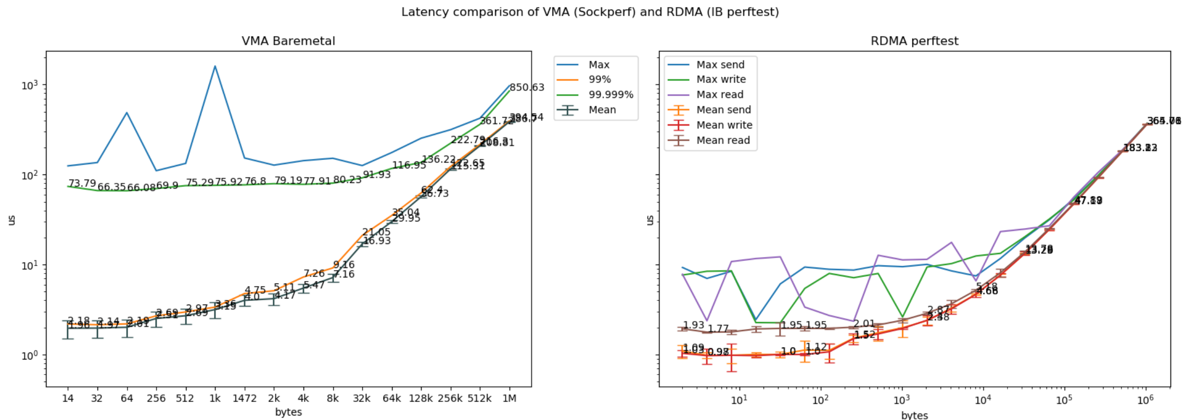

Figure 5.6: VMA Sockperf latencies compared to RDMA performance tests

to ten-microsecond scale on larger data sizes. There is more deviation on VMA on maximum and 99.999% latencies, but most messages stay stable near the mean latencies.

Latency results using Sockperf with corresponding mean CPU usage can be seen in Appendix A, Figure A.1. Comparison of Sockperf CPU performance between different test cases can be seen in Appendix A, Figure A.2. The graphs show the CPU usage clearly shifting from kernel to user space on VMA, although consuming around one full CPU core due to constant polling. As we have 40 cores in one machine, one core at 100% usage means 2.5% overall usage in our CPU graph. Similarly, 2.1% usage in the 48 core CPU corresponds to one full core. Due to an error in our graphing code, on the X axis, only the last CPU graph shows the correct data sizes for the CPU data, while the first CPU graphs show data indices.

Figure 5.8 shows the throughput performance of all our test cases using the Sockperf tool. We observe that VMA achieves a message rate as high as 17.5 million messages per second, immediately at the smallest data sizes. This can be related to the fact that small RDMA messages are immediately sent, whereas Nagle's algorithm on TCP may wait for the buffer to fill before transfer. The throughput achieves its peak already at data sizes of 1 kB and greater, performing very similarly as bare metal RDMA throughput.

The iperf3 throughput graph shown in Figure 5.7 compares bare metal, Kubernetes Flannel CNI, and VMA both from node to node, and locally by connecting to iperf3 server running on the same node. The node-to-node graph shows stagnation of the VMA throughput. As we did not gather CPU

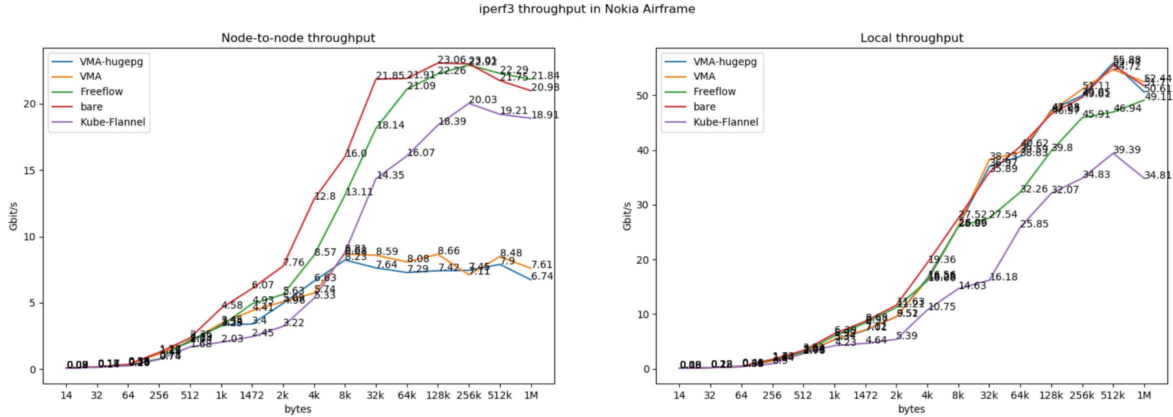

Figure 5.7: Iperf3 throughput measurements of VMA, Freeflow, Flannel and bare metal

data for the iperf3 test, possible CPU bottlenecking is inconclusive. According to an issue in VMA library Github [46], VMA has an issue of exhaustion of the transmission buffer and waiting for acknowledgements when running with data sizes greater than MTU. This could be the problem, although it is unclear why the Sockperf throughput test shows no issues. The problem may stem from the Nagle's algorithm used by iperf3, and we could, in hindsight, have tested if the problem persists when using the `no-delay` option, disabling the algorithm used. In a locally ran iperf3 throughput test, this VMA stagnation does not show. In addition, the test shows the throughput capability of the network card when not constrained by the 25Gbps link used, rising up to 55 Gbps bare metal.

5.4 Test case 4: Freeflow TCP in cluster

Our results show Freeflow outperforming the Flannel container network, and results relate closely to bare metal, especially on throughput testing.

Figure 5.8 shows the Sockperf throughput performance of multiple of our test cases. The throughput of Freeflow is seen very close to bare metal, with a clear improvement comparing to the Flannel container network across all data sizes. While Kim et al. [29] noted that the Socket to Verbs translation method *rsocket* used in their test of Freeflow, on large data sizes, showed inferior throughput compared to bare metal, on our test we note that the throughput using Freeflow sees no throughput degradation on TCP/IP. As we

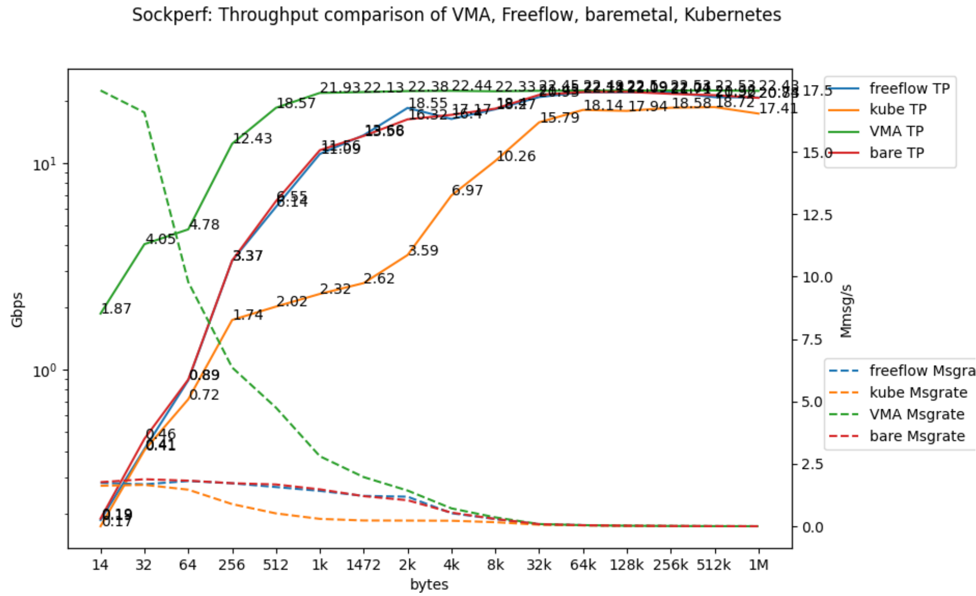

Figure 5.8: Sockperf throughput comparison between VMA, Freeflow, Kubernetes Flannel, and bare metal

did note a throughput performance degradation on large data sizes during the testing of the VMA library using the iperf3 tool in Section 5.3, this degradation could be originating from an issue in multiple Socket to Verbs translation libraries. Indeed, the Freeflow authors used the iperf tool during their measurement.

The throughput overall achieves in the best case around 22 Gbps of the 25 Gbps link. The throughput data shows that when passing MTU, the IP fragments start to rise, and the peak packets per second in the link with all TCP/IP tests is around 1 to 2 million. The message rate then drops when message size is increased, whereas packet rate stays the same as the bandwidth reaches its peak. On smaller packets, the packet header is comparatively larger to data payload size, making larger packets more efficient.

Figure 5.9 shows the mean and 99.999 percentile latency comparison between bare metal and the Flannel container network on the left, and comparison between bare metal and Freeflow on the right. The Flannel overhead is most noticeably seen on data sizes under 4 kB. The graph shows that the latencies of Freeflow generally perform close to the bare metal latencies, most noticeably on data sizes of under 4 kB. The Freeflow 99.999 percentile latencies show improvement even over bare metal. We determine that Freeflow

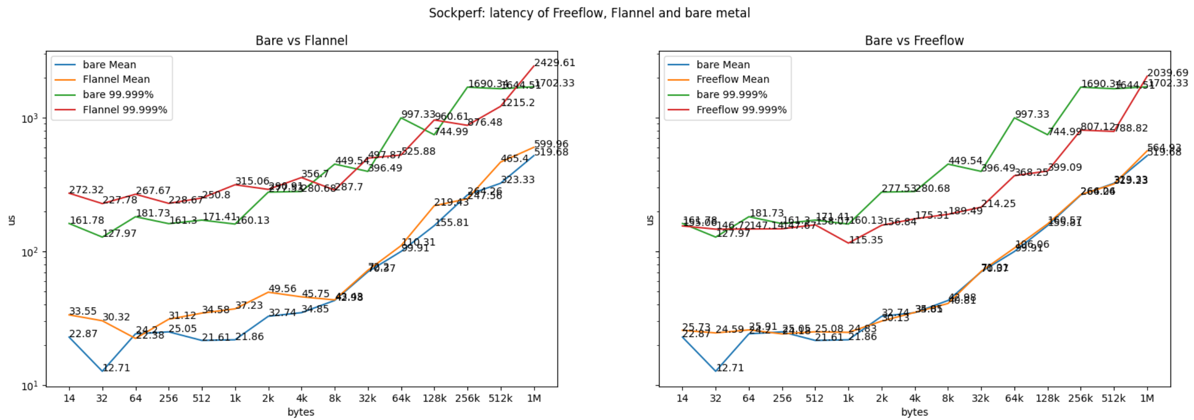

Figure 5.9: Sockperf bare metal latency comparison to Flannel and Freeflow on Nokia Airframe

TCP shows clear improvement over the Flannel latencies for all data sizes.

Appendix A, Figure A.2 shows the CPU usage of all test cases on the Nokia Airframe setup. It is shown that Freeflow matches bare metal in CPU usage as well, without the software interruptions of a container network and showing largely kernel usage. We determine that Freeflow efficiently achieves the expected results to accelerate containers towards bare metal.

5.5 Summary

In this chapter, we presented our results of both TCP/IP and RDMA measurements within four different test cases. First, we determined that there is a large difference between our two test setups, and a large overhead stems from the Flannel VXLAN option. With the host-gw option, the Flannel underlay networking mode can be used and that gives a latency overhead only on the ten-microsecond scale. We decided to focus our testing on the high-performance Nokia Airframe hardware.

The RDMA benchmark measured using native RDMA performance tools concludes that at least an order of magnitude lower latencies than even the minimum latencies of TCP/IP can be acieved with RDMA. The RDMA latencies are additionally very stable, whereas the TCP/IP measurement latencies vary greatly due to varying processor usage.

The RDMA measurements achieve a very high message transfer rate immediately from the smallest data sizes, as there is no need to fill a buffer

before sending messages. Thus RDMA achieves a good throughput even with very small message sizes.

When accelerating native Socket-based applications with RDMA by using the Mellanox VMA translation library, the added latency compared to native RDMA is largely only a few microseconds. Thus converting applications to use RDMA may not be required. Still, we presented an issue with the translation which should show particularly on throughput when the Nagle's algorithm is in use. The issue presumably exists on multiple different translation libraries, as we noted a similar trend on the *rsocket* library testing performed by the authors of Freeflow.

The CPU usage data shows, as expected, that the TCP/IP-based tests inflict largely kernel space processing, with a portion consumed on software interruptions when using the container network. With VMA, mainly user space usage is recorded, although the library consumes a CPU core to perform constant polling of the memory.

Finally, we measured that the TCP branch of the Freeflow prototype accelerates the container network roughly to bare metal performance, with negligible additional overhead. This indicates that the RDMA branch of the prototype should also be viable for RDMA use.

Chapter 6

Discussion

While in this thesis, we were only able to test RDMA on bare metal, and not specifically on containerized clusters, the performance impact of RDMA is clear. From bare metal comparison alone, it can be determined that the possible usage of RDMA between containers could provide perfomance advantages even to the extent of an order of magnitude over TCP/IP.

We examined the current possible tools for RNIC virtualization, and emerging prototype solutions for the cluster use of RDMA. These are very promising solutions that could provide clear benefits in inter-container communication when further developed and tested.

We note that the Socket to Verbs translation libraries may still contain issues on translating some TCP/IP networking cases. In particular, we theorize the splitting of packets above the size of MTU causes the Nagle's algorithm to function incorrectly and exhaust the transmission buffer, as seen affecting our iperf3 throughput results using VMA. This should be examined further. Specifically determining messaging to perform with no delay could function as a workaround.

Overall, our testing could have been planned better in multiple ways. Specifically, we should have planned a larger time window to have access to the Nokia Airframe setup equipped with RDMA-capable NICs. Our limited time using the test setup prevented extensive troubleshooting, effecting our ability to gather conclusive RDMA test data specifically for containers and pods. It is also clear that we were unprepared in our ability to set up the required RNIC and kernel drivers for cluster pods. On the other hand, our RDMA test data on bare metal suggests similar benefits can be assumed if close to bare metal performance level can be established in a containerized setup.

Our test cases could have included multiple other technologies for more conclusive results. Technologies such as DPDK could have been tested and

compared to RDMA performance. For more extensive testing, especially of VMA, we could have used Kubernetes plugins such as SR-IOV to attach the RNIC into containers. This would have provided data on the performance of Socket to Verbs translation tools regarding cluster implementations specifically.

Our Raspberry Pi setup remained largely as the starting point for our tests, while the relevant results were gathered from the Nokia Airframe setup. We could have investigated the software implementations of RDMA further. It would be interesting to see if software implementations of RDMA, such as Soft RoCE, would provide noticeable performance benefits on lower performance hardware, such as the Raspberry Pi, when connecting to hardware RDMA on the other end.

6.1 Future work

Further testing and comparison of different Socket to Verbs translation libraries, such as VMA and *rsocket*, is needed to create a more broad picture of possible performance differences. The translation libraries should be tested using various traffic generation tools determining their robustness in all communication scenarios. The root cause of possible issues within the libraries, such as the throughput stagnation using the iperf tool described in this paper, should be determined.

RDMA and Socket to Verbs translation tools should be more extensively tested specifically in cluster environments, by attaching the RNIC to pods via plug-ins such as SR-IOV. For Kubernetes, a plugin similar to SR-IOV could be developed to properly assign RDMA hardware to containers in the case of container migration. Further, it would be interesting to research if a container overlay network could be fully accelerated with a technology such as VMA.

The installation of correct drivers for RNICs in containers and pods should be specifically looked into. The driver installation was a crucial step preventing us from extensive testing at this time. It would be interesting to see research into a streamlined driver installation process in this specific case. The drivers could possibly be located in a special container within a pod environment.

The method of intercepting network calls transparently by an intermediary container is a promising option for container networking with RDMA. Focus should be on the overall utilization and improvement of the Freeflow library and similar solutions.

Overall, ideal development direction in the future could be that Ethernet-

based devices and the Socket API itself would incorporate the RDMA operations, making now specialized hardware the standard.

This thesis did not assess the security implications for RDMA communication, where the recipient memory is directly accessed. If the use of RDMA communication would become widespread, this aspect should be looked into.

Chapter 7

Conclusions

The amount of data transmitted in systems such as mobile networks is continuously increasing. The widely used microservice architecture may provide services increasingly closer to end users, and yet, calls for efficient communication between parts of the application. Technologies such as RDMA are used in some data centers to greatly improve performance compared to TCP/IP. Still, the usage of RDMA for communication in containerized clusters has no de facto solution. Mainly, proper hardware virtualization is difficult, and the solutions are in the prototyping stage.

This thesis provided insight into low latency communication techniques by using zero copy and kernel bypass methods, and how they could be used in containerized clusters. We focused on RDMA and tested it on bare metal, concluding that its container implementation is worth developing further in future work. We confirmed RDMA performance to surpass TCP/IP performance by an order of magnitude.

While we did not assess the cluster performance of RDMA completely in this thesis, we concluded that the bare metal performance benefit is significant. While often sacrificing a CPU core for polling, the use of RDMA still might free computing resources for other uses overall due to efficient communication. Our test results also show that the translation of traditional Socket calls to use RDMA can achieve performance very close to bare metal RDMA, reducing the need to repurpose applications for RDMA use.

Bibliography

[1] Abbasi, U., Bourhim, E. H., Dieye, M., and Elbiaze, H. A performance comparison of container networking alternatives. *IEEE Network 33*, 4 (2019), 178–185.

[2] Abdelwahab, S., Hamdaoui, B., Guizani, M., and Znati, T. Network function virtualization in 5G. *IEEE Communications Magazine 54*, 4 (2016), 84–91.

[3] Abranches, M., Goodarzy, S., Nazari, M., Mishra, S., and Keller, E. Shimmy: Shared memory channels for high performance inter-container communication. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)* (2019).

[4] Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., and Palopoli, A. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)* (2019), IEEE, pp. 1–6.

[5] Beck, M., and Kagan, M. Performance evaluation of the RDMA over Ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching* (2011), pp. 9–15.

[6] Beltre, A. M., Saha, P., Govindaraju, M., Younge, A., and Grant, R. E. Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)* (2019), IEEE, pp. 11–20.

[7] Bosch, P., Duminuco, A., Pianese, F., and Wood, T. L. Telco clouds and virtual telco: Consolidation, convergence, and beyond. In

12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops (2011), IEEE, pp. 982–988.

[8] Chen, R., and Sun, G. A survey of kernel-bypass techniques in network stack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence* (2018), pp. 474–477.

[9] Cloud Native Computing Foundation. K3s - Lightweight Kubernetes GitHub. `https://github.com/k3s-io/k3s`, Accessed 25.5.2023.

[10] Cloud Native Computing Foundation. Container Network Interface GitHub. `https://github.com/containernetworking/cni`, Accessed 5.6.2023.

[11] Dalessandro, D., Devulapalli, A., and Wyckoff, P. Design and implementation of the iWARP protocol in software. In *IASTED PDCS* (2005), pp. 471–476.

[12] Dalessandro, D., Devulapalli, A., and Wyckoff, P. iWARP protocol kernel space software implementation. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium* (2006), IEEE, pp. 8–pp.

[13] Deming, D. A. InfiniBand software architecture and RDMA. In *Storage Developer Conference* (2013).

[14] Docker Inc. Docker website. `https://www.docker.com/`, Accessed 6.6.2023.

[15] Drepper, U. What every programmer should know about memory. *Red Hat, Inc 11*, 2007 (2007), 2007.

[16] Eder, M. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM) 1* (2016).

[17] Erfanian, J., et al. Network operator perspectives on NFV priorities for 5G. In *NFV# 17 Plenary Meeting* (2017).

[18] ESnet. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. `https://github.com/esnet/iperf`, Accessed 1.6.2023.

[19] Espe, L., Jindal, A., Podolskiy, V., and Gerndt, M. Performance evaluation of container runtimes. In *CLOSER* (2020), pp. 273–281.

[20] F5 Inc. Nginx website. `https://www.nginx.com/`, Accessed 25.5.2023.

[21] Gao, P. X., Narayan, A., Karandikar, S., Carreira, J., Han, S., Agarwal, R., Ratnasamy, S., and Shenker, S. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (2016), pp. 249–264.

[22] Géhberger, D., Balla, D., Maliosz, M., and Simon, C. Performance evaluation of low latency communication alternatives in a containerized cloud environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), IEEE, pp. 9–16.

[23] Hefty, S. Rsockets. In *2012 OpenFabrics International Workshop, Monterey, CA, USA* (2012), vol. 5, p. 52.

[24] Jiang, X., Shokri-Ghadikolaei, H., Fodor, G., Modiano, E., Pang, Z., Zorzi, M., and Fischione, C. Low-latency networking: Where latency lurks and how to tame it. *Proceedings of the IEEE 107*, 2 (2018), 280–306.

[25] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), pp. 437–450.

[26] Kapočius, N. Performance studies of Kubernetes network solutions. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)* (2020), IEEE, pp. 1–6.

[27] Kaur, G., and Bala, M. RDMA over converged Ethernet: A review. *International Journal of Advances in Engineering & Technology 6*, 4 (2013), 1890.

[28] Kaur, G., Kumar, M., and Bala, M. Comparing Ethernet & soft RoCE over 1 gigabit Ethernet. *International Journal of Computer Science & Information Technolo 5*, 1 (2014), 323.

[29] Kim, D., Yu, T., Liu, H. H., Zhu, Y., Padhye, J., Raindel, S., Guo, C., Sekar, V., and Seshan, S. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 113–126.

[30] Kissel, E., and Swany, M. Evaluating high performance data transfer with RDMA-based protocols in wide-area networks. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (2012), IEEE, pp. 802–811.

[31] Kubernetes. What is Kubernetes? `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`, Accessed 17.5.2021.

[32] Kubernetes. Kubernetes Documentation. `https://kubernetes.io/docs/home/`, Accessed 27.5.2023.

[33] Larsen, S., Sarangam, P., Huggahalli, R., and Kulkarni, S. Architectural breakdown of end-to-end latency in a tcp/ip network. *International journal of parallel programming 37*, 6 (2009), 556–571.

[34] Link, C., Sarran, J., Grigoryan, G., Kwon, M., Rafique, M. M., and Carithers, W. R. Container orchestration by kubernetes for rdma networking. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)* (2019), IEEE, pp. 1–2.

[35] Linux Foundation. Data Plane Development Kit. `www.dpdk.org`, Accesssed 1.6.2023.

[36] linux-rdma GitHub. Infiniband Verbs Performance Tests. `https://github.com/linux-rdma/perftest`, Accessed 28.5.2023.

[37] Liu, J., Wu, J., and Panda, D. K. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming 32*, 3 (2004), 167–198.

[38] MacArthur, P., Liu, Q., Russell, R. D., Mizero, F., Veeraraghavan, M., and Dennis, J. M. An integrated tutorial on InfiniBand, verbs, and MPI. *IEEE Communications Surveys & Tutorials 19*, 4 (2017), 2894–2926.

[39] Mademann, F. The 5G system architecture. *Journal of ICT Standardization 6*, 3 (2018), 77–86.

[40] Mellanox. RoCE vs. iWARP competitive analysis. Mellanox white paper, 2017.

[41] Mellanox. Sockperf Network Benchmarking Utility GitHub. `https://github.com/Mellanox/sockperf`, Accessed 28.5.2023.

[42] Mellanox. ConnectX-5 ethernet adapter cards user manual. https://img-en.fs.com/file/user_manual/connectx-5-ethernet-adapter-cards-user-manual.pdf, Accessed 8.1.2023.

[43] Mellanox. Kubernetes IPoIB/Ethernet RDMA SR-IOV networking with ConnectX4/ConnectX5. https://enterprise-support.nvidia.com/s/article/kubernetes-ipoib-sriov-networking-with-connectx4-connectx5, Accessed 8.1.2023.

[44] Mellanox. VMA Architecture. https://github.com/Mellanox/libvma/wiki/Architecture, Accessed 8.1.2023.

[45] Mellanox. VMA GitHub. https://github.com/Mellanox/libvma, Accessed 8.1.2023.

[46] Mellanox. VMA GitHub, Issue 778. https://github.com/Mellanox/libvma/issues/778, Accessed 8.1.2023.

[47] Microsoft. Freeflow GitHub. https://github.com/microsoft/Freeflow, Accessed 20.11.2021.

[48] Microsoft. Freeflow TCP GitHub. https://github.com/Microsoft/Freeflow/tree/tcp, Accessed 20.11.2021.

[49] Mogul, J. C., and Minshall, G. Rethinking the TCP Nagle algorithm. *ACM SIGCOMM Computer Communication Review 31*, 1 (2001), 6–20.

[50] Namiot, D., and Sneps-Sneppe, M. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014), 24–27.

[51] Nokia. Airframe Open Edge server. https://www.nokia.com/networks/data-center/airframe-data-center/open-edge-server/, Accessed 8.1.2023.

[52] NVIDIA. Appendix: Sockperf - UDP/TCP Latency and Throughput Benchmarking Tool. https://docs.nvidia.com/networking/pages/viewpage.action?pageId=15042073, Accessed 28.5.2023.

[53] NVIDIA. VMA Library Architecture. https://docs.nvidia.com/networking/display/VMAv931/VMA+Library+Architecture, Accessed 8.1.2023.

[54] Pfister, G. F. An introduction to the Infiniband architecture. *High performance mass storage and parallel I/O 42*, 617-632 (2001), 102.

[55] Qi, S., Kulkarni, S. G., and Ramakrishnan, K. Understanding container network interface plugins: design considerations and performance. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN* (2020), IEEE, pp. 1–6.

[56] Rancher. Lightweight certified Kubernetes with Rancher. `https://www.rancher.com/products/k3s`, Accessed 25.5.2023.

[57] Rashti, M. J., and Afsahi, A. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g. In *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), IEEE, pp. 1–8.

[58] Recio, R. RDMA enabled NIC (RNIC) Verbs Overview. *dated Apr 29* (2003), 1–28.

[59] Recio, R., Metzler, B., Culley, P., Hilland, J., and Garcia, D. A remote direct memory access protocol specification. Tech. rep., RFC 5040, October, 2007.

[60] Rizzo, L. netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 101–112.

[61] Serrano, C., Garriga, B., Velasco, J., Urbano, J., Tenorio, S., and Sierra, M. Latency in broad-band mobile networks. In *VTC Spring 2009-IEEE 69th Vehicular Technology Conference* (2009), IEEE, pp. 1–7.

[62] Seth, S., and Venkatesulu, M. A. *TCP/IP Architecture, Design and Implementation in Linux.* Wiley New York, NY, USA, 2008.

[63] Song, J., and Alves-Foss, J. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS) 6*, 4 (2012), 256.

[64] Toimela, S., et al. Containerization of telco cloud applications. Master's thesis, Aalto University, 2017.

[65] TOP500. Highlights - june 2016. `https://www.top500.org/lists/top500/2016/06/highlights/`, Accessed 25.5.2023.

[66] Trani, R. G. *Integrating VNF Service Chains in Kubernetes Clusters*. PhD thesis, Politecnico di Torino, 2020.

[67] Wang, D., Fu, B., Lu, G., Tan, K., and Hua, B. vsocket: Virtual socket interface for rdma in public clouds. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2019), pp. 179–192.

[68] Wang, X., Cavdar, C., Wang, L., Tornatore, M., Chung, H. S., Lee, H. H., Park, S. M., and Mukherjee, B. Virtualized cloud radio access network for 5G transport. *IEEE Communications Magazine 55*, 9 (2017), 202–209.

[69] wg. wrk - a HTTP benchmarking tool GitHub. `https://github.com/wg/wrk`, Accessed 27.5.2023.

[70] Wikipedia. User Datagram Protocol. `https://en.wikipedia.org/wiki/User_Datagram_Protocol`, Accessed 25.5.2023.

[71] Ziegler, V., Viswanathan, H., Flinck, H., Hoffmann, M., Räisänen, V., and Hätönen, K. 6g architecture to connect the worlds. *IEEE Access 8* (2020), 173508–173520.

[72] Zimmermann, H. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications 28*, 4 (1980), 425–432.

Appendix A

Additional graphs

Additional graphing including the CPU usage for various measurements are shown here. Our graphing code for the CPU usage contains a bug, showing the indices of the data on X-axis. The actual data sizes used can be seen from the X-axis of the last CPU usage graph.

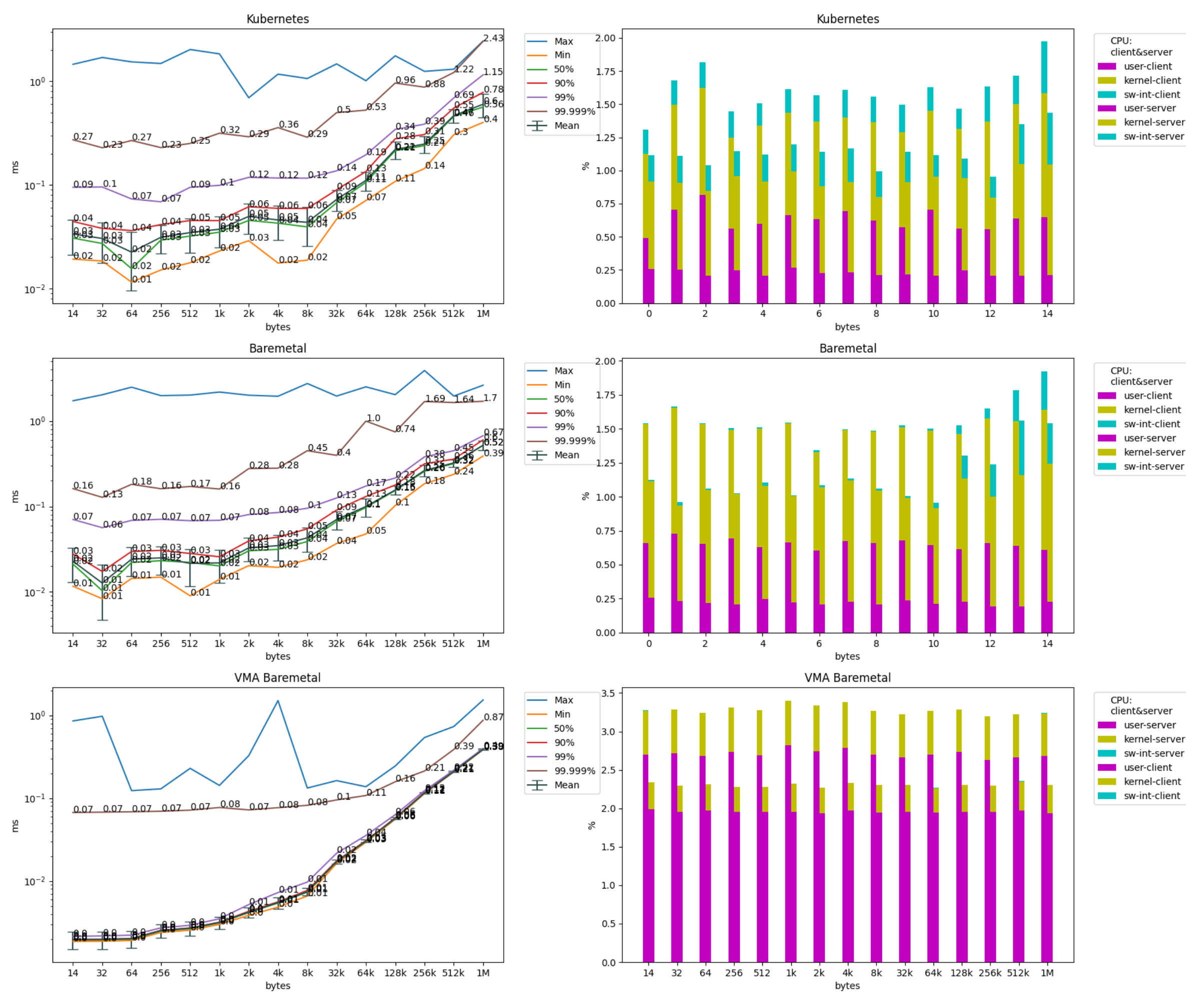

Figure A.1: Sockperf latencies of Kubernetes Flannel, bare metal, and VMA with CPU usage on Nokia Airframe setup.

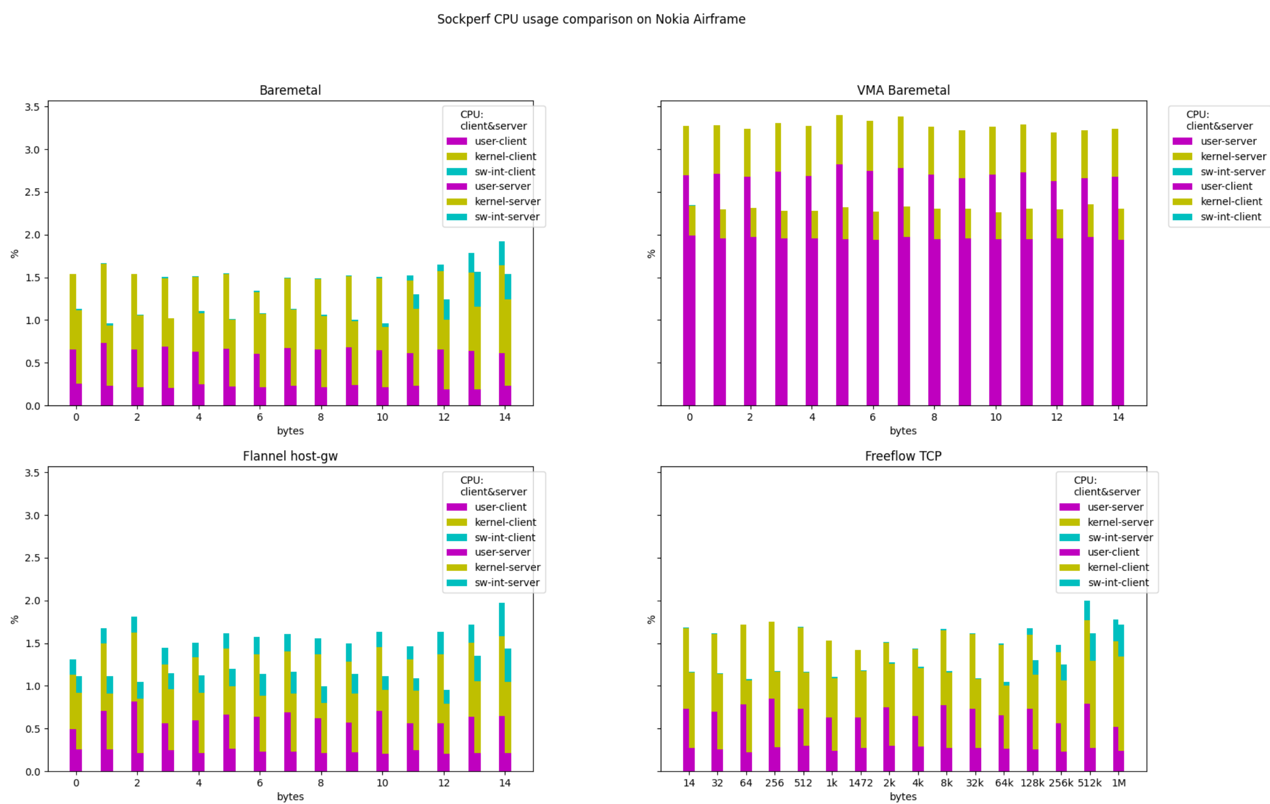

Figure A.2: CPU usage comparison of Sockperf latency tests on bare metal, Kubernetes Flannel, VMA and Freeflow TCP on Nokia Airframe setup.

Appendix B

Freeflow router Kubernetes deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ffrouter
spec:
  selector:
    matchLabels:
      app: ffrouter
  replicas: 2
  template:
    metadata:
      labels:
        app: ffrouter
    spec:
      hostNetwork: true
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - ffrouter
            topologyKey: "k3s.io/hostname"
      containers:
```

```
      - name: ffrouter-c
        image: freeflow/freeflow:tcp
        env:
        - name: HOST_IP_PREFIX
          value: 10.37.215.0/24
        securityContext:
          privileged: true
        volumeMounts:
          - mountPath: /freeflow/
            name: freeflow
    volumes:
      - name: freeflow
        hostPath:
          path: /freeflow/
          type: DirectoryOrCreate
```