Master of Science in Telecommunication Systems
January 2019

Optimization of Packet Throughput in Docker Containers

Anusha Ginka
Satya Sameer Salapu Venkata

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Telecommunication Systems. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):

Anusha Ginka
E-mail: anusha.ginka@gmail.com

Satya Sameer Salapu Venkata
E-mail:sameer.salapu@gmail.com

External advisor:
Maysam Mehraban
Developer
Ericsson, Lindholmen, Gothenburg

University advisor:
Dr. Patrik Arlos
Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

abstract

Container technology has gained popularity in recent years, mainly because it enables a fast and easy way to package, distribute and deploy applications and services. Latency and throughput have a high impact on user satisfaction in many real-time, critical and large-scale online services. Although the use of microserivces architecture in cloud-native applications has enabled advantages in terms of application resilience, scalability, fast software delivery and the use of minimal resources, the packet processing rates are not correspondingly higher. This is mainly due to the overhead imposed by the design and architecture of the network stack. Packet processing rates can be improved by making changes to the network stack and without necessarily adding more powerful hardware.

In this research, a study of various high-speed packet processing frameworks is presented and a software high-speed packet I/O solution i.e., as hardware agnostic as possible to improve the packet throughput in container technology is identified. The proposed solution is identified based on if the solution involves making changes to the underlying hardware or not. The proposed solution is then evaluated in terms of packet throughput for different container networking modes. A comparison of the proposed solution with a simple UDP client-server application is also presented for different container networking modes. From the results obtained, it is concluded that packet mmap client server application has higher performance, when compared with simple UDP client server application.

Keywords: Microservices, Networking, Packet Throughput, Performance

Acknowledgements

We would like to express my gratitude to my university advisor Patrik Arlos for giving me an opportunity to work with this thesis under his guidance and providing support throughout the period of thesis work.

We would also like to thank Maysam Mehraban and Jan Lundkvist A at Ericsson R&D Gothenburg for giving me this opportunity and for their valuable suggestions, guidance and support during the thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

UDP User Datagram Protocol

VM Virtual Machine

DPDK Data Plane Development Kit

API Application Program Interface

OS Operating System

CNM Container Networking Model

IP Internet Protocol

RTT Round Trip Time

NIC Network Interface Card

TCP Transmission Control Protocol

BSD Berkeley Software Distribution

GPU Graphics Processing Unit

IEEE Institute of Electrical and Electronics Engineers

eBPF Extended Berkeley Packet Filter

QUIC Quick UDP Internet Connections

Chapter 1

Introduction

With the advent of the microservices, architecture businesses across industries like telecommunications and other IT companies are choosing microservices to develop, maintain and scale new applications. Each microservice can be deployed, upgraded, scaled, and restarted independently of other containers in the application, typically as part of an automated system, enabling frequent updates to live applications without impacting end customers. Today, with microservices architecture, apps are being built as a distributed collection of services, which aligns with the distributed nature of the cloud. This is a much more granular means of deploying the minimum resources necessary to reliably maintain performance. With the rising applications of microservices architecture in cloud infrastructures, the overhead imposed by the design and architecture of the network stack, quickly becomes a bottleneck for packet processing rates as the infrastructure grows in size.

The initial applications of cloud computing relied on the Infrastructure as a Service that replaced on-premise infrastructure with virtual machines running in cloud data centers. In the cloud, however, servers and databases are distributed. While the IaaS was sufficient for small applications such as basic web services, it was still very difficult to scale and manage security at the same time [1]. Cloud-native is an approach to building and running applications that fully exploit the advantages of the cloud computing delivery model. Cloud-native is about how applications are created and deployed, not where [2].

The idea behind microservices architecture is to break down the application into smaller and more manageable independent services [3]. This architecture enables application resilience, scalability, fast software delivery and the use of minimal resources. Services need their own allotment of resources for computing, memory, and networking. However, both from a cost and management standpoint, it's not feasible to scale the number of VMs to host each service of your app as you move to the cloud. Containers are used for the deployment of services developed with microservice architecture. Containers are light-weight, portable and scalable. Containers [4, 5] perform better in terms of execution time, la-

tency, throughput, power consumption, CPU utilization and memory usage when compared to virtual machines. Containers have less flexibility in operating systems, to run containers with the different operating system you need to start a new server, whereas in VMs any kind of the operating system can live next to each other on the same server. Virtual machines provided by hypervisor-based virtualization techniques are claimed to be more secure than the containers as they add an extra layer of isolation between the applications and the host. An application running in a VM is only able to communicate with VM kernel, not the host kernel. Consequently, in order for the application to escalate out of a VM, it must bypass the VM kernel and the hypervisor before it can attack the host kernel. On the other hand, containers can directly communicate with the host kernel, thus allowing an attacker to save a great amount of effort when breaking into the host system[6].

There are many cloud-native applications, which uses the microservices architecture, that we are all familiar with. The Netflix movie streaming service is certainly one.[7] Many bigger companies, such as Google, Facebook, Twitter, Microsoft, Amazon are now trying to migrate to cloud-native architectures for greater scalability and ease of use.

1.1 Motivation

Since 2014, we have seen an increase in the use of containers for cloud-native applications. The TCP/IP networking stack has factors like interrupt overhead [8], receive livelock [9] and context switching contributing to the degradation in packet throughput in containers. These factors refer to hardware resources limitations and the way that packet processing is organized based on the network stack. This impedes the operation of network I/O intensive applications. For better network performance, the interrupt overhead and the number of context switches should be as low as possible. Containers (docker) require tuning to support I/O intensive applications [5].

To address this issue many high-speed packet I/O frameworks have been proposed. Some of the popular high-speed packet I/O frameworks are DPDK(Data Plane Development Kit)[10] , OpenOnload [11], Netmap [12] and PacketShader [13]. Most of the high-speed packet I/O frameworks requires special hardware or is designed for specific hardware only. The main motivation of the thesis is to develop a software high-speed packet processing solution, which is as hardware independent as possible, in a cloud-native application. The test application is deployed using Docker containers in an experimental testbed, and the packet throughput is evaluated, for varying parameters, for this new solution.

1.2 Problem Statement

The problem when scaling the microservices architecture is that, as the number of containers increases beyond a number, the interrupt overhead and the number of context switches increases[14], as a result, latency increases and the packet throughput degrades. Packet throughput is defined as the number of successful packets delivered over the communication channel per second. The problem with a large number of context switches can be overcome by kernel bypass methods [15, 16]. Containers require tuning to support I/O intensive applications [5].

1.3 Aims and Objectives

The main aim of this research is to investigate different high-speed packet I/O methods, to propose a software high-speed packet I/O solution in a hardware-agnostic way and improve the packet throughput.

The objectives of the thesis are:

- Understanding the Linux network stack.
- Investigate different high-speed packet I/O methods.
- Propose a solution independent of the underlying hardware.
- Evaluate the performance, in terms of packet throughput, of the proposed solution for different networking modes.
- Compare the performance, in terms of packet throughput, of the proposed solution with the performance of simple UDP client server.

1.4 Research Questions

The research questions are as follows:

1. How can the packet throughput be improved in docker containers?
2. What is the effect of the implemented method on the performance of networking in docker containers?

The performance metric packet throughput is the main focus of this thesis.

1.5 Research Method

The research method followed in this thesis is a Qualitative research methodology for identifying and proposing a solution to answer the research questions in a systematical procedure. A study of various packet processing frameworks is done. Based on the study conducted a solution is proposed and the implemented solution is then tested for different test scenarios and the effect of various parameters on packet throughput is evaluated. The different steps followed in this research method are:

- Literature Study
- Experimental design
- Implementation
- Data collection

1.6 Thesis Outline

The Thesis is organized as follows:

- Chapter 2 gives a brief background on Containers, docker container networking and OpenVswitch.
- Chapter 3 provides related works done on this area of research.
- Chapter 4 presents a study of various packet processing frameworks.
- Chapter 5 presents and describes the research method followed in this research, the experimental test bed and the test scenarios used to carry out the experimental performance evaluation.
- Chapter 6 presents the results and provides an analysis of the results obtained.
- Chapter 7 provides the conclusion of our performance evaluation and future work.

Chapter 2

Background

2.1 Containers

Containers are lightweight operating system virtualization that allows the users to run an application and its dependencies in a resource-isolated environment [17]. The containers share the same kernel of the host operating system. All the dependencies, libraries, configuration files needed to run it are bundled into a single package. It also ensures that the applications deploy, quickly, reliably regardless of the deployment environment. [18] There are three important things that developers, architects, and systems administrators, need to know:

1. All applications, inclusive of containerized applications, rely on the underlying kernel
2. The kernel provides an API to these applications via system calls
3. Versioning of this API matters as it's the "glue" that ensures deterministic communication between the user space and kernel space.

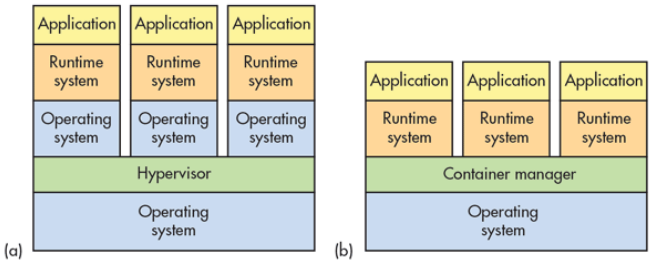

Figure 2.1: container vs VM

As shown in figure 2.1, VMs execute on top of hypervisors and Containers execute on the host OS through the container engine. Hypervisor runs the VMs that have their own OS using hardware VM support. Container engine merges into the kernel of the host OS, it can differentiate between the containerized execution and user-level processes. Due to low overhead of containers, they can start new containers at a fast speed, whereas while starting a VM underlying

OS takes time, memory and space needed for disk storage. (1) Containers are lightweight in nature, which makes them shareable i.e., building and shipping the images consisting of applications along with their dependencies. Since VMs are heavyweight in nature, they are not easily transferred from one machine to another because of their own kernel. Containers are useful when there is a microservice architecture, as it allows to use the container functions properly. VMs are used when a full platform should run on a single system.

While containers are sometimes treated like virtual machines, it is important to note, unlike virtual machines, the kernel is the only layer of abstraction between programs and the resources they need access to. All processes make system calls from Userspace to Kernel space[18], in the same way as containers are processes they also make system calls from Userspace to kernel space.
Advantages of containers:

1. As containers are lightweight, it enables less overhead in terms of performance and size.

2. They can be created much faster than the Virtual Machine instances.

3. As containers are in an isolated environment, any upgrade or changes in one container will not affect another container.

Disadvantages of containers:

1. Containers can directly communicate with the host kernel, thus allowing an attacker to save a great amount of effort when breaking into the host system.

2. As the containers share the same kernel of the host OS, they have root access which makes them less isolated when compared to the VMs.

3. Less flexibility in operating systems, a new server is needed to start to run containers with different operating systems. Whereas in the virtual machine any kind of OS can live next to each other on the same server.

2.1.1 Docker containers

Docker is the company driving the container movement and the only container platform provider to address every application across the hybrid cloud [19]. Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Users interact with Docker through Docker client which is used to run for manipulating containers. Docker Daemon is the component which manages the Docker images and containers on the local machine. It manages and manipulates containers using commands received from Docker client. Docker has been used in some popular applications, such as Spotify, Yelp and eBay. Using docker

in the cloud has advantages such as, first it provides interfaces to create docker container in a simple and safe manner. Docker also works well with third-party tools such as Puppet[20] , ansible [21] and Vagrant [22], thus making docker containers easily deployable in cloud. Many orchestration tools such as Mesos[23] and kubernetes [24] also support Docker containers. These tools provide an abstract layer of resources management and scheduling over Docker. Docker consists of two major components: the Docker engine and Docker Hub. The former is an open source virtualization solution, while the latter is a Software-as-a-Service platform for sharing Docker images.

2.1.2 Linux Containers

Linux containers (LXC) is an opensource, lightweight OS-level virtualization software that helps to run multiple Linux systems on a single host. It provides a Linux environment as close as to a standard Linux installation without the need for a separate kernel. LXC uses kernel namespaces to enforce isolation [25]. It also uses cgroups (control groups) functionality, this limits the CPU isolation, memory, disk I/O and network usage of one or more processors.

LXD is an extension on the LXC project by providing a command line interface (CLI) management interface and a REST API for network-based management [26]. LXD can work with large scale deployments by integrating with OpenStack. LXD works on any Linux distribution.

2.1.3 Rkt

Rkt is an application container engine which is developed for modern production cloud-native environments[27]. Rkt is developed by CoreOS. Rkt uses the pod-native approach which is a collection of applications which can be executed in a shared context. Pods are the core execution unit of Rkt.

2.1.4 Windows Server containers

Windows server containers reply on the windows server kernel which uses process and namespace isolation to create separate space for each container. All containers that run on the host system shares the kernel. Windows server containers don't provide strong security, it is best to avoid using untested and untrusted containers in this mode [28].

2.2 Docker Container Networking

Containers need a network to communicate. Docker networking sits between the application and the network. It is also known as Container Network Model or

CNM. CNM brings connectivity for the docker containers. Multiple networks can be attached to a container [29]. Container, when attached to two networks, can communicate with the members of either container. There are five possible network setups in Docker:

1. **None**: Container will have its own isolated network stack and it is left unconfigured. The user needs to set up the network stack.

2. **Host**: When host network mode is attached to the container, network stack is not isolated. It shares the same network stack with the host i.e., the container has full access to the network interfaces of the host. It uses the host port space to expose the services running inside the container[30].

3. **Bridge**: Newly created Docker container connects to the default bridge network [31]. Instead of using default bridge users can create a user-defined bridge network with a unique name. Containers which are attached to the same bridge can intercommunicate. It allows the container running on the same host without port clashing. The bridge driver is a local scope driver i.e., it only provides service discovery and connectivity on a single host [29]. Each container will have its own namespace and it provides isolation from containers which are not connected to the bridge network. The container gets a private IP address and Docker daemon generates random MAC address from the allocated IP address. The bridge is connected to the external network and provides access via Network Address Translation(NAT)[32].

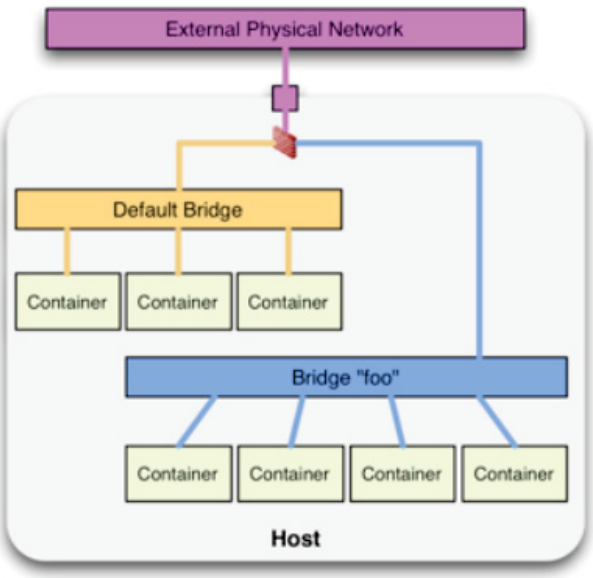

Figure 2.2: Docker Bridge networking mode.

4. **Overlay**: When a container is in the overlay network, the network driver creates a distributed network among multiple Docker daemon hosts [33]. It removes the need to do OS-level routing between the containers. In host and bridge network mode does the communication they don't provide a solution for container communication across the hosts. Overlay network sits on top of host-specific networks, allowing secure communication between the containers connected to it. Containers can be connected to more than one network at a time, they can communicate across networks they are connected to.

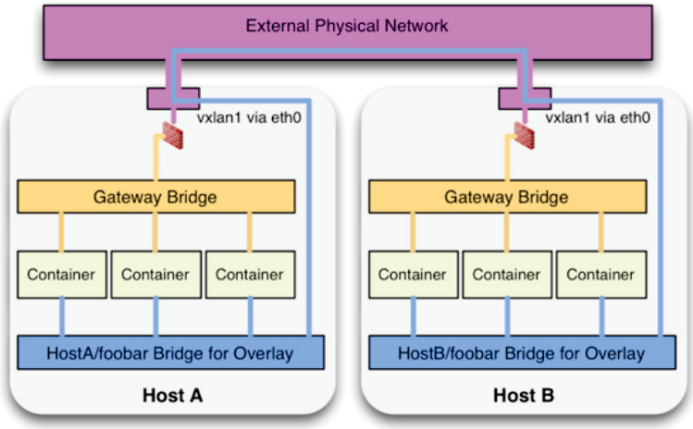

Figure 2.3: Docker Overlay networking mode.

5. **Macvlan**: Macvlan network allows to assign a MAC address to a container, by which it can appear as a physical device on the network [34]. It connects the container interfaces directly to host interfaces. Containers can be addressed with routable IP addresses that are on the subnet of the external network.

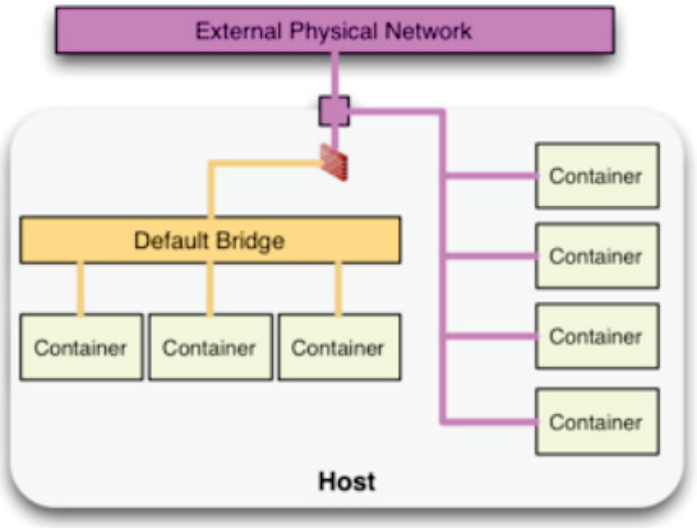

Figure 2.4: Docker Macvlan networking mode.

2.3 Open vSwitch

Open vSwitch is a multilayer software switch licensed under the open source Apache 2 license. The goal is to implement a production quality switch platform that supports standard management interfaces and opens the forwarding functions to programmatic extension and control [35]. The rise of server virtualization has brought with it a fundamental shift in data center networking. A new network access layer has emerged in which most network ports are virtual, not physical – and therefore, the first hop switch for workloads increasingly often resides within the hypervisor. Reconfiguring and preparing a physical network for new workloads slows their provisioning, and coupling workloads with physical L2 segments severely limit their mobility and scalability to that of the underlying network. These pressures resulted in the emergence of network virtualization [36]. In network virtualization, virtual switches become the primary provider of network services for VMs, leaving physical data center networks with transportation of IP tunnelled packets between hypervisors. This approach allows the virtual networks to be decoupled from their underlying physical networks, and by leveraging the flexibility of general purpose processors, virtual switches can provide VMs, their tenants, and administrators with logical network abstractions, services and tools identical to dedicated physical networks.

Network virtualization demands a capable virtual switch – forwarding functionality must be wired on a per virtual port basis to match logical network abstractions configured by administrators. Implementation of these abstractions,

across hypervisors, also greatly benefits from fine-grained centralized coordination. This approach starkly contrasts with early virtual switches for which a static, mostly hard-coded forwarding pipelines had been completely sufficient to provide virtual machines with L2 connectivity to physical networks. It was this context: the increasing complexity of virtual networking, the emergence of network virtualization, and limitations of existing virtual switches, that allowed Open vSwitch to quickly gain popularity. Today, on Linux, its original platform, Open vSwitch works with most hypervisors and container systems, including Xen, KVM, and Docker. Open vSwitch also works "out of the box" on the FreeBSD and NetBSD operating systems and ports to the VMware ESXi and Microsoft Hyper-V hypervisors are underway.

Chapter 3

Related Work

This section deals with the literature work that has been done previously which motivated and guided us towards implementing and completing this thesis work.

Chinna Venkanna Varma et al. [14] concludes that as the number of containers increases beyond a number, the RTT increases and throughput decreases. A performance evaluation is carried out and factors that affect the performance are discussed. Barbette et al. [15] concludes that using a fast packet processing framework does increase network performance. Different packet processing software's are used. Rajesh et al., [16] Shanmugalingam [37] and Zhao et al.[38] all conclude that using a fast packet processing framework like Intel DPDK improves the throughput and reduces the RTT and Latency. The problem with vendor-specific solutions is that it works only with specific hardware or is accustomed to a proprietary hardware vendor. This paper also concludes that context switching is the major reason for performance degradation in high-speed networks.

In [39] the following parameters are considered to affect the performance of packet processing applications: 1) CPU speed and inadequate utilization, 2) interrupt overhead, 3) limited bus bandwidth in comparison to a fast processing unit, 4) memory latency, 5) I/O latency. In [40], there is a more detailed description of the factors related to the first parameter mentioned above, meaning the CPU, that restrict high-speed packet processing. More specifically these overheads originate from: 1) performing memory allocation and deallocation for each packet (per-packet memory allocation), 2) overhead of copying data between kernel and user-space, 3) expensive cache misses, 4) per-packet system calls, because of the CPU-intensive context switch between kernel and user-space, and 5) the transformation of the parallelized processing of packets by the queues of multi-queue NICs to a serialized one. This happens because all packets must converge to one single point, thus creating a bottleneck.

Interrupt overhead of Gigabit network devices can have a significant negative impact on system performance [8]. Interrupt-driven systems tend to perform very badly under such heavy load conditions. Interrupt-level handling, by definition, has absolute priority over all other tasks. If the interrupt rate is high enough, the system will spend all of its time responding to interrupts, and

nothing else will be performed; and hence, the system throughput will drop to zero. This situation is called receive livelock [9]. In this situation, the system is not deadlocked, but it makes no progress on any of its tasks, causing any task scheduled at a lower priority to starve or not have a chance to run. At low packet arrival rates, the cost of interrupt overhead and latency for handling incoming packets are low. However, interrupt overhead cost directly increases with an increase of packet arrival rates, causing receive livelock.

Chapter 4

Kernel Bypass methods

This chapter provides a description of the various high-speed packet processing frameworks. As mentioned in the section 1.5, each framework is studied and the proposed solution is identified based on if the method involves making hardware specific changes to the kernel.

4.1 Packet_mmap

Packet mmap [41] is a feature added to the standard UNIX sockets in the Linux kernel, using packet buffers in a memory region shared (mmaped, hence its name) between the kernel and the user space.

The advantages of using packet_mmap to enhance the network I/O performance are:

- Bypassing the TCP/IP network stack also known as native Linux network stack, i.e. the packets are only processed by the processing framework and by the applications running on top of them.
- Using polling instead of interrupts to receive packets.
- Pre-allocating packet buffers at the start of an application with no further allocation or deallocation of memory during execution of an application.
- No copying of data between user and kernel memory space as a packet is copied once to memory by the NIC and this memory location is used by processing framework and applications alike.
- Processing batches of packets with one API call on reception and sending.
- More importantly, packet mmap can be used without making any changes to the NIC driver and does not need special hardware.

4.1.1 Netmap

Netmap [12] is a framework that performs packet processing at high speeds by reducing the avoidable costs that slow the processing down. It enables the applications to gain fast access to the packets. Netmap is independent of specific devices and hardware and is built upon existing operating system characteristics and applications.

ioctl() function is used to synchronize the state of the rings between kernel and userspace. Ioctl() tells the kernel which buffers have been read by user space and informs user space of any newly received packets. On the TX side, ioctl() tells the kernel about new packets to transmit and reports to userspace how many free slots are available [42]. When the Netmap application is not active, the driver works normally for the OS and the traditional applications. In Netmap mode i.e., after starting the Netmap enabled application, the NIC is disconnected from the host stack and made directly accessible to applications. The operating system, however, still believes that the NIC is present and available, so it will try to send and receive traffic from it. Netmap attaches two software Netmap rings to the host stack, making it accessible using the Netmap API. No packets are delivered to the standard OS interfaces and traditional applications. When the Netmap application is stopped, the NIC driver switches back to its original functionality. Maintaining this compatibility in the driver allows for easy integration into general-purpose operating systems. Netmap has already been integrated into the FreeBSD kernel [43].

Although Netmap has significantly better performance compared to the standard networking stack or the TCP/IP stack. The disadvantage is that applications that are not Netmap enabled cannot reach the outside world and Netmap is NIC driver specific i.e., for the Netmap to work a patch to the NIC driver should be installed depending on the type of NIC the machine is running. As the main goal of this thesis is to develop a solution, without making any changes to the underlying hardware or without the help of dedicated hardware to enhance the network I/O performance, this method is not suitable for our solution.

4.2 Data Plane Development kit (DPDK)

DPDK [10] The Intel Data Plane Development Kit is somehow comparable to Netmap but provides more user level functionalities such as a multi-core framework with enhanced NUMA-awareness, and libraries for packet manipulation across cores.

The DPDK driver does not feature a transparent mode, i.e. when this driver is loaded, the NIC becomes available to DPDK but is made unavailable to the

Linux kernel regardless of whether any DPDK-enabled application is running or not. The DPDK implements a run to completion model [44] for packet processing, where all resources must be allocated prior to calling Data Plane applications, running as execution units on logical processing cores. The model does not support a scheduler and all devices are accessed by polling. The primary reason for not using interrupts is the performance overhead imposed by interrupt processing. In addition to the run-to-completion model, a pipeline model may also be used by passing packets or messages between cores via the rings. This allows work to be performed in stages and may allow the more efficient use of code on cores[45].

Data Plane Development Kit (DPDK) greatly boosts packet processing performance and throughput, allowing more time for data plane applications [46]. DPDK can improve packet processing performance by up to ten times[47]. Although there is a great increase in performance compared to the standard networking stack or the TCP/IP stack, the DPDK works only on Intel hardware. The DPDK was designed specifically for Intel hardware. As the main goal of this thesis is to develop a solution, without making any changes to the underlying hardware or without the help of dedicated hardware to enhance the network I/O performance, this method is not suitable for our solution.

4.3 OpenOnload

OpenOnload® [11] is a high-performance network stack from Solarflare that dramatically reduces latency and CPU utilization and increases message rate and bandwidth. OpenOnload runs on Linux and supports TCP/UDP/IP network protocols with the standard BSD sockets API and requires no modifications to applications to use. It achieves performance improvements in part by performing network processing at user-level, bypassing the OS kernel entirely on the data path [48]. Networking performance is improved without sacrificing the security and multiplexing functions that the OS kernel normally provides.

An important feature of this model is that applications do not get direct access to the networking hardware and so cannot compromise system integrity[49]. Onload can preserve system integrity by partitioning the NIC at the hardware level into many, protected 'Virtual NICs' (VNIC). An application can be granted direct access to a VNIC without the ability to access the rest of the system (including other VNICs or memory that does not belong to the application). Thus, Onload with a Solarflare NIC allows optimum performance without compromising security or system integrity. OpenOnload is comparable to DPDK but made by Solar Flare, only for their products.

4.4 NetSlice

NetSlice [50] is an operating system abstraction that attempts to provide high-speed packet processing and runs in the user-space. NetSlice also determines the path that packets must follow from NICs to applications and vice versa. Packets are slightly processed on the k-peer CPU core, and then they are directed to the user-space application, where they are processed in a pipeline. NetSlice is based on the conventional socket API. More specifically, it uses the operations write, read and poll for the distinct data flows of the different NetSlices. NetSlice extends the API via the ioctl mechanism. One difference of the NetSlice extended API is the batched system calls that it can support. Due to the batching, NetSlice achieves a reduced number of system calls, and thus minimizes the delays of system calls. Batching results also in a reduction of the overheads that per-packet processing poses. NetSlice does not exploit the advantages of zero-copy techniques, and it does copy the packets from kernel to the user-space.

4.5 PF_RING

PF_Ring [51] is a framework that provides fast packet capturing and processing. PF_Ring implements zero-copy by avoiding data copy between the kernel and the user-space. It achieves that with the use of packet buffers that are found in a memory region common to the kernel and the users' applications [12]. The packet buffers are pre-allocated. Due to this packet buffer schema, it avoids the cost of per-packet memory allocation and deallocation. PF_RING ZC features a driver with capabilities like those of Netmap, i.e. the driver is based on a regular Linux driver acting transparently if no special application is started. During the time such an application is active regular applications cannot send or receive packets using the respective default OS interfaces.

The PF_Ring [52] can be used with a specialized type of device driver, namely Direct NIC Access (DNA), for achieving even faster processing without the intervention of CPU and the use of system calls. PF_Ring provides a mapping between the NIC memory and the user-space memory and permits the explicit communication between applications and NICs. The PF_RING ZC version is not free and costs per mac address.

4.6 PacketShader

PacketShader [13] constitutes a user-space framework for fast packet processing that takes advantage of the GPU characteristics. The GPU part of PacketShader is not publicly available, only the code of the packet engine was released, which can be used on its own. However, this engine is not developed as actively as Netmap,

PF_RING ZC, or DPDK leading to a low number of updates in the repository [53]. Highly parallel applications can take advantage of the ample computation cycles and the ample memory bandwidth of GPUs, higher than that of CPUs, which enables access to different datasets [54]. PacketShader makes use of GPU to enhance the network I/O performance.

Chapter 5

Methodology

This chapter mainly describes the research method, experimental design and procedure followed the experimental test bed and the experimental scenarios used in the performance evaluation. The different steps taken in this thesis are briefly explained in Section 1.5.

5.1 Literature Study

The first step begins with the study of the network stack to understand the path a packet takes while going out/in of a system and to identify the bottlenecks in TCP/IP network stack. Articles and research papers from the IEEE explore associated with the TCP/IP bottlenecks are studied to do so. Based on the study of previous related work, research papers, reports and high-speed packet processing frameworks a solution is proposed to overcome the problem stated in Chapter 1. A detailed explanation of the studied high-speed packet processing frameworks is presented in Chapter 4. The identification of a solution is based on the method adopted in packet processing. The networking stack has factors like interrupt overhead [8], receive livelock [9] and context switching contributing to the degradation in packet throughput in containers. These factors refer to hardware resources limitations and the way that packet processing is organized based on the network stack. This impedes the operation of network I/O intensive applications[5]. For better network performance, the interrupt overhead and the number of context switches should be as low as possible. Using the packet processing frameworks the network performance can be improved in containers. The main motivation in this thesis is to find a solution, which does not make hardware specific changes to improve network performance. Based on the study of different frameworks and the previous related works, the proposed solution uses the packet mmap method. The main criteria to propose a solution was if the solution required to make any hardware specific changes. The advantages of packet mmap are listed in section 4.1.

5.2 Experimental Design

The design consists of a study and implementation of the proposed solution, to evaluate the performance of the packet throughput in different scenarios. The performance evaluation is done for different docker networking modes. The implementation of different networking modes is briefly explained in Section 5.3. In each networking mode, the packet throughput is evaluated by varying the number of packet sent, the packet size and the transmission rate. Packet mmap is used to develop a software kernel bypass solution, mainly because the solution is independent of making hardware specific changes to the Operating system the application is running on. The client-server is written in C. Refer to appendix A.1,A.2 for the full source code of mmap client and server application. The applications are deployed as two different docker containers. After the networking is set up, the containers are deployed. The UDP is a simple client-server written in C. Refer to appendix B.1,B.2 for the full source code of UDP client and server application. The UDP client - server is deployed in an identical manner as the mmap solution and both the solution are compared for packet throughput. The containers are deployed on the Ubuntu 16.04.3 LTS machine. The System specifications are listed in Table 5.1.

Table 5.1: System Specification

System Component	Description
OS	Ubuntu 16.04.3 LTS
kernel version	4.15.2-42-generic
CPU	Intel(R) Core (TM) i7-4610M CPU @ 3.00 GHz
Memory	16 GB
Memory Type	DDR3
Memory Speed	1600 MHz
CPU(s)	4
Thread(s) per core	2
Network card(s)	Intel Corporation
network driver version	24

The network interface is given as an input to the client script. The client should know the network interface to be used for sending the packets to the server. Binding the socket to the network interface is mandatory to know the header size of frames used in the circular buffer. The network interface must be binded to the socket to achieve zero-copy. A packet socket allows the user to set the data link information like source MAC address and the destination MAC address. Mmap is the function used to create the TX and RX ring buffer in a

shared memory space between the Userspace and the Kernel space. The circular buffer is compound of several physically discontiguous blocks of memory, they are contiguous to the user space, hence just one call to mmap is needed. This has the advantage of reducing the syscalls, interrupts and context switching, whenever a packet is sent/received.

In this research, 1024 frames are initialized in the tx ring buffer. Each frame size is set to 8192 Bytes. All the headers are constructed in the Userspace. Initially, the kernel initializes all the frames in ring buffer to TP STATUS AVAILABLE. After the packet is constructed, it is put in the shared memory space and the status of the frame in the ring buffer is set to TP STATUS SEND REQUEST. The kernel updates each status of sent frames with TP STATUS SENDING until the end of the transfer. At the end of each transfer, buffer status returns to TP STATUS AVAILABLE. Multiple buffers can be filled with data and set the status to TP STATUS SEND REQUEST. When the send() function is called, all the buffers with status set to TP STATUS SEND REQUEST are forwarded to the network device. For each send request, 1000 frames are filled with data and set status to TP STATUS SEND.

5.3 Implementation

The Experimental setup used to evaluate the proposed solution is described in this section. The proposed solution is evaluated in a Linux machine. A list of the system specifications is provided in Table 5.1. The proposed solution is initially tested by running both the client and server applications without docker containers. The proposed solution is then tested for different docker networking modes, by deploying the client and server as two different containers on the same host. The impact of running both the client and server applications on the same host is that the system will suffer from a high CPU load. Different docker networking modes are described in section 2.1.1. In this research, the proposed solution is evaluated for the following testbed setups:

- Native app to app on same host
- Docker bridge mode using Open vSwitch
- Docker Macvlan bridge networking mode
- Docker Macvlan trunk bridge networking mode

For each networking mode, the proposed solution has a client and server as two containers. The proposed solution is evaluated for packet throughput and compared with a simple client-server UDP application. The UDP client-server are deployed in an identical way as the proposed solution. After the testbed is set up,

the experiments to be used in the performance evaluation are executed. By using the MMAP client-server architecture, the packet throughput measurements will be collected as the total number of received packets divided by the time difference between the arrival time of first and last packets. The packet throughput values will be collected in an identical way using the UDP client-server architecture. The packet throughput for the mmap client server is compared with UDP client-server architecture. The following subsections present a brief explanation of the experimental testbed used in each scenario for the performance evaluation.

5.3.1 Native app to app on same host

The client and server application is initially tested by running both the applications without docker containers, as shown in figure 5.1 . This experiment can be used as a baseline for the experiments carried out in Docker containers. The performance evaluation is done for three different scenarios for the mmap and udp application. The metric packet throughput is compared for mmap and udp application in each scenario. The three scenarios are explained in the section 5.4.

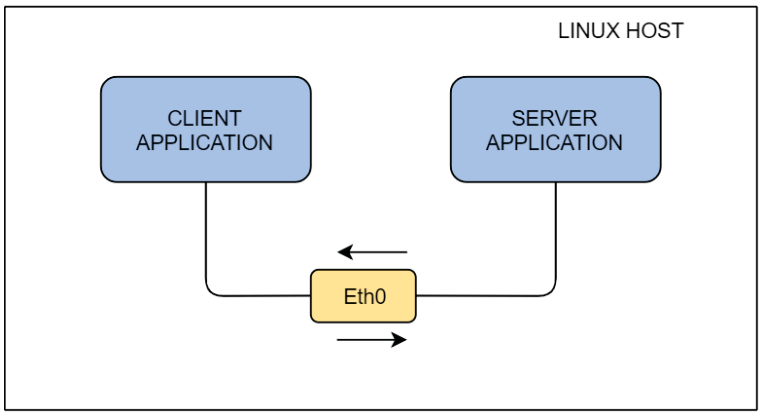

Figure 5.1: Native app to app

5.3.2 Docker bridge networking using Open vSwitch

An Open vSwitch bridge is created and the two containers server and client are connected to the ovs bridge, as shown in the figure 5.2. The Open vSwitch cli is used to create the bridge and connect the containers to the bridge. The performance evaluation is done for three different scenarios for mmap application and UDP application. The packet throughput is compared for mmap application and UDP application for each scenario. The three scenarios are explained in the section 5.4.

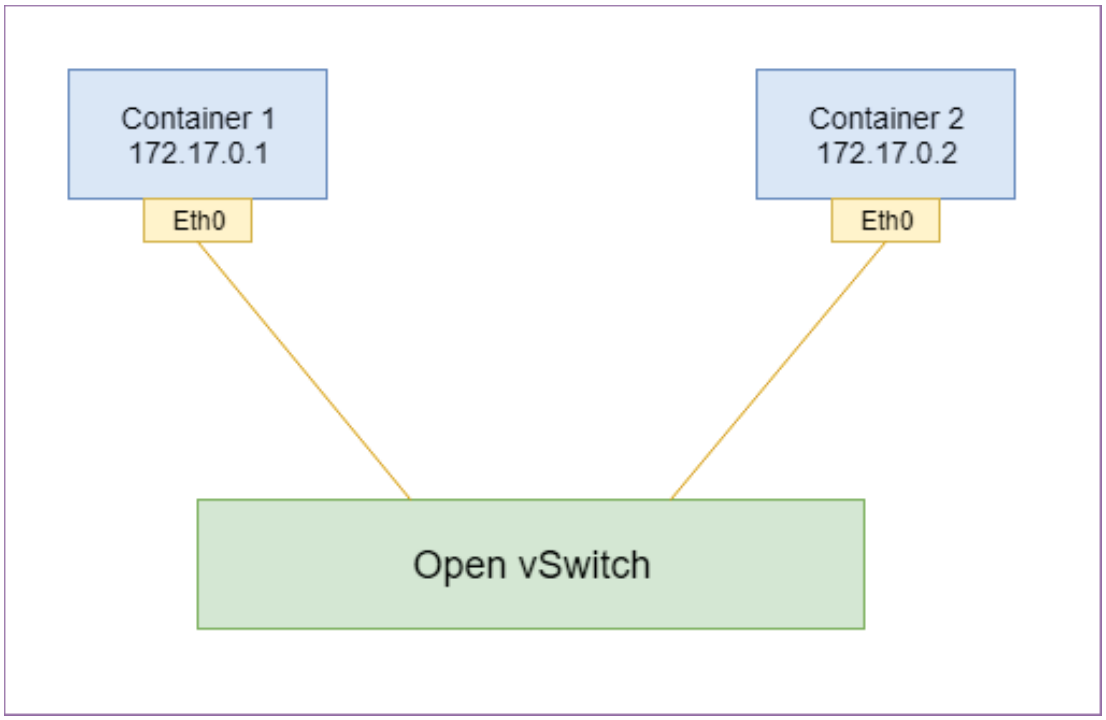

Figure 5.2: Docker bridge mode using Open vSwitch

5.3.3 Docker MACVLAN networking

When you create a Macvlan network, it can either be in bridge mode or 802.1q trunk bridge mode.

- In bridge mode,Macvlan traffic goes through a physical device on the host.
- In 802.1q trunk bridge mode, traffic goes through an 802.1q sub-interface which Docker creates on the fly. This allows you to control routing and filtering at a more granular level.

To create a Macvlan network which bridges with a given physical network interface, use –driver macvlan with the docker network create command. You also need to specify the parent, which is the interface the traffic will physically go through on the Docker host. If you specify a parent interface name with a dot included, such as eth0.50, Docker interprets that as a sub-interface of eth0 and creates the sub-interface automatically [34]. The performance evaluation is done for three different scenarios for mmap application and UDP application. The packet throughput is compared for mmap application and UDP application for each scenario. The three scenarios are explained in the section 5.4. The setup is shown in figure 5.3 and 5.4.

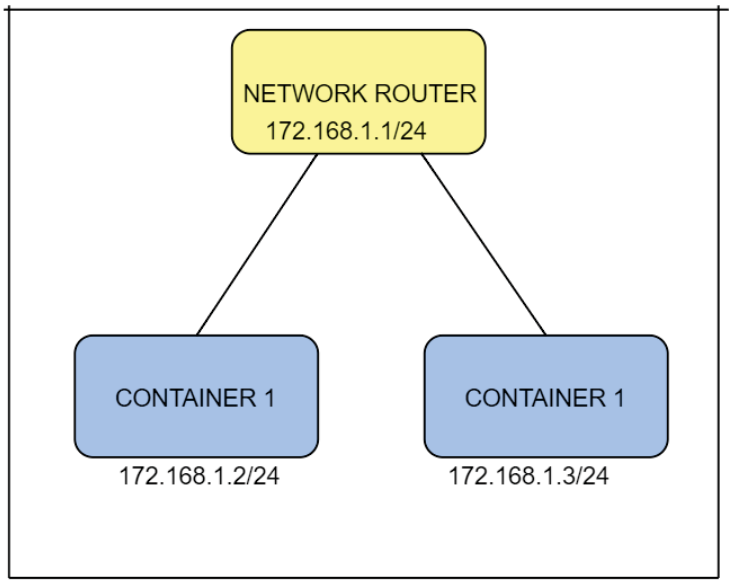

Figure 5.3: Docker MACVLAN bridge networking mode

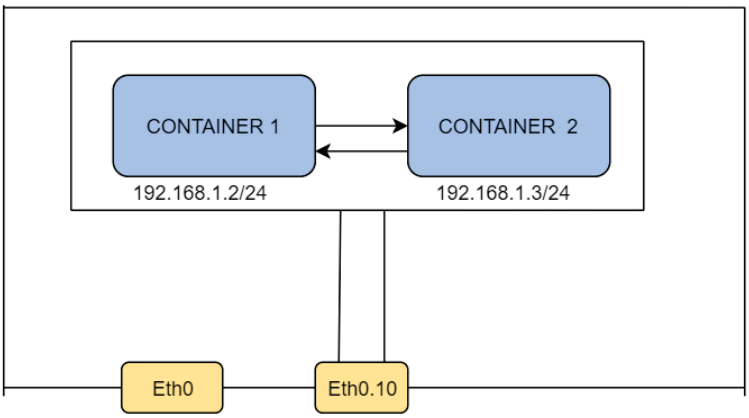

Figure 5.4: Docker MACVLAN trunk bridge networking mode

5.4 Experimental Scenarios and Data Collection

This section explains the experimental scenarios for which the performance evaluation is done. After the testbed is set up, experiments are carried out for different scenarios. In each scenario, the metric packet throughput is collected as the number of packets received by the server per second. The time when the first packet is received is taken as the start time for experiment and the end time is the time the last packet arrived. The times are collected using the function gettimeofday(). After the networking is set up as shown in section 5.3, the two containers are de-

ployed. In each setup, 3 different parameters are varied. The packet throughput values are noted down in a case by varying one parameter and keeping the other two parameters at a constant value. So the effect of each parameter on packet throughput is evaluated. The number of packets sent, the packet size and the tx rate are all given as inputs to the client. On each run, the varied parameter value is changed and the other two parameter values are kept at a constant value. For controlling the data rate, the rate of packets sent is calculated after each batch of packets are sent. If the rate calculated is more than the intended send rate, the usleep function is used to sleep for a period of time. The time to sleep is calculated as the difference between the total number of sent packets by the actual send rate and the duration the experiment has run till then. If the calculated packet rate is less than the actual send rate, then the experiment continues without waiting.

5.4.1 Scenario I: By varying the number of packets sent

In this case, the containers are deployed for the client and server. First, the mmap application is deployed and the packet throughput values are collected by varying the number of packets sent while having the packet tx rate and the packet size as constant. The number of packets sent is 2 million, 4 million, 6 million, 8 million and 10 million. The packet transmission rate is kept constant at 500 Mbps and the data size is kept constant at 64 Bytes. After collecting the values for mmap application, new containers for UDP application is deployed and the experiment is done similar to the mmap application.

5.4.2 Scenario II: By varying the transmission rate of packets sent

In this case, the containers are deployed for the client and server. First, the mmap application is deployed and the packet throughput values are collected by varying the transmission rate of packets sent while having the number of packets sent and the packet size as constant. The varying transmission rates are 200Mbps, 400Mbps, 600Mbs and 800 Mbps. The number of packets sent is kept constant at 1 million and the data size is kept constant at 64 Bytes. After collecting the values for mmap application, new containers for UDP application is deployed and the experiment is done similar to the mmap application.

5.4.3 Scenario III: By varying the packet size

In this case, the containers are deployed for the client and server. First, the mmap application is deployed and the packet throughput values are collected by varying the packet size while having the number of packets sent and the transmission rate as constant. The varying packet sizes are 64 bytes, 128 bytes, 256 bytes, 512

bytes and 1024 bytes. The packet transmission rate is kept constant at 500 Mbps and the number of packets sent is kept constant at 1 million. After collecting the values for mmap application, new containers for UDP application is deployed and the experiment is done similar to the mmap application.

Chapter 6

Results and Analysis

In this chapter, the results collected from the experiments conducted for the experimental setup mentioned in section 5.3 are presented. The results are collected for different testbed scenarios as explained in the section 5.4. The results from the proposed solution are then compared with the results obtained from a simple UDP client-server architecture. The metric evaluated is packet throughput.

6.1 Native app to app on same host

Scenario I: By varying the number of packets sent

In this scenario, the number of packets sent by the client is 2 million, 4 million, 6 million, 8 million and 10 million. The packet transmission rate is kept constant at 500 Mbps and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In Figure 6.1, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of the mmap are shown against the values of simple udp client-server application. From the graph, it is observed that, with an increase in the number of packets sent, the packet throughput values for the mmap application are significantly higher compared to the values collected for the udp application. The packet throughput for the UDP application has the highest value of 170.755 kpps with an average value of 165 kpps. The packet throughput for the mmap application has the highest value of 592.086 kpps with an average value of approximately equal to 585 kpps, which is 3.5 times higher when compared to the UDP application. The experiment statistics are shown in figure 6.2.

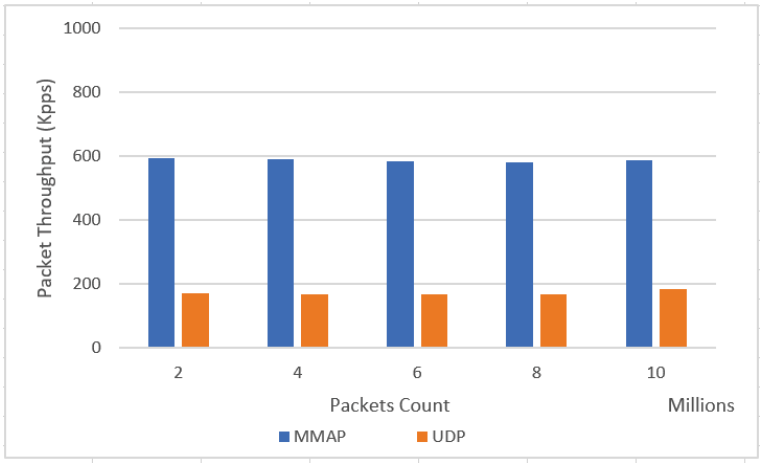

Figure 6.1: varying number of packets sent for native app to app test bed

Number of Packets Sent	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
2000000	2000000	1998274	1547324875.82	1547324878.24	3.067	1547335284.32	1547335293.23	8.4402
4000000	4000000	3999872	1547325309.32	1547325315.32	6.068	1547335420.43	1547335436.46	16.881
6000000	6000000	5985774	1547325350.34	1547325359.13	9.189	1547335487.43	1547335513.16	25.866
8000000	8000000	7996165	1547325443.52	1547325456.34	12.695	1547335814.12	1547335848.83	33.259
10000000	10000000	9991528	1547325534.12	1547325549.12	15.197	1547335944.51	1547335986.56	41.912

Figure 6.2: Data for varying packet count

Scenario II: By varying the transmission rate of packets sent

In this scenario, the transmission rate of the packet sent from the client is 200Mbps, 400Mbps, 600Mbs and 800 Mbps. The number of packets sent is kept constant at 1 million and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.3, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the transmission rate of packets sent from the client increases the packet throughput values for the mmap application are significantly higher than the values collected for the udp application. The packet throughput for the UDP application has the highest value of 325.334kpps. The packet throughput for the mmap application has the highest value of 433.152 kpps. The experiment statistics are shown in figure 6.4.

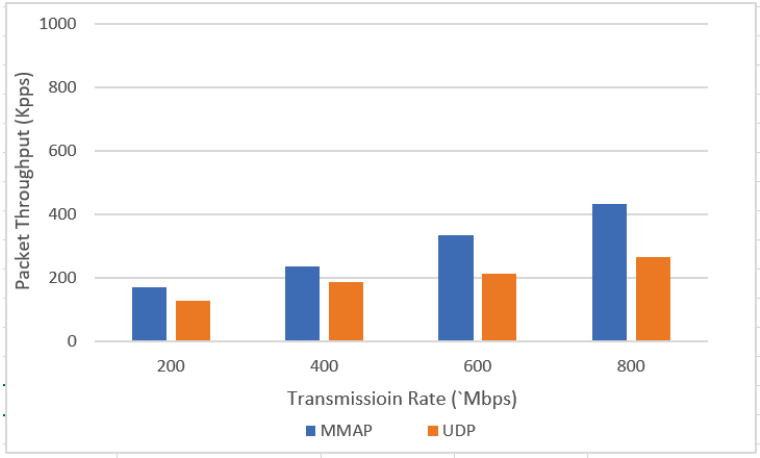

Figure 6.3: varying tx rate of packets sent for native app to app test bed

Transmission Rate	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
200	1000000	998534	1547326075.03	1547326080.31	5.582	1547386798.12	154738680.62	7.379
400	1000000	999345	1547326140.35	1547326144.27	4.567	1547386871.36	1547386877.83	5.177
600	1000000	998979	1547326265.92	1547326268.04	2.976	1547386939.05	1547386943.23	4.294
800	1000000	998993	1547326343.90	1547326345.23	2.180	1547387023.05	1547387026.15	3.194

Figure 6.4: Data for varying tx rate

Scenario III: By varying the packet size

In this scenario, the packet size is varied by the client and the packet sizes for which values are collected are 64 bytes, 128 bytes, 256 bytes, 512 bytes and 1024 bytes. The packet transmission rate is kept constant at 500 Mbps and the number of packets sent is kept constant at 1 million. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure **??**, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the packet size increases the packet throughput values for the mmap application are significantly higher when compared to the values collected for the udp application. As the packet size increases the packet throughput values gradually decreases, with the highest value of 599.361 kpps for 64 Bytes and with the lowest value of 132.734 kpps for 1024 Bytes for mmap application. The udp application has the highest value of 180.780 kpps for 64 Bytes and the lowest value of 60.916 kpps for 1024 bytes. The experiment statistics are shown in figure **??**.

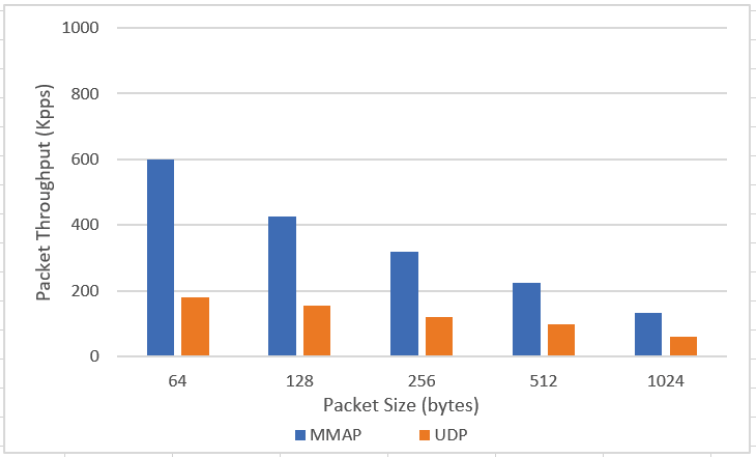

Figure 6.5: varying packet size of packets sent for native app to app test bed

Packet Size	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
64	1000000	999345	1547325699.43	1547325700.13	1.539	1547386335.02	1547386340.19	5.153
128	1000000	999624	1547325766.12	1547325768.90	2.097	1547386444.22	1547386450.24	6.401
256	1000000	998534	1547325830.03	1547325833.49	3.568	1547386516.19	1547386525.98	8.337
512	1000000	999893	1547325922.24	1547325926.22	4.439	1547386578.21	1547386589.09	10.223
1024	1000000	999970	1547326000.35	1547326013.23	7.541	1547386669.23	1547386686.32	16.415

Figure 6.6: Data for varying packet size

6.2 Docker Containers

6.2.1 Setup I: Bridge networking mode

Scenario I: By varying the number of packets sent

In this scenario, the number of packets sent by the client is 2 million, 4 million, 6 million, 8 million and 10 million. The packet transmission rate is kept constant at 500 Mbps and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In Figure 6.7, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, with an increase in the number of packets sent, the packet throughput values for the mmap application are significantly higher compared to the values collected for the udp application. The slight packet drops in the udp application are due to the high CPU load, on account of running the client and server on the same host. The packet throughput for the UDP application has the highest value of 236.152 kpps with an average value of 225 kpps. The packet throughput for the mmap application has the highest value of 773.665 kpps with an average value of approximately equal to 760 kpps, which is 3.5 times higher when compared to the UDP application. The experiment data is shown in figure 6.8.

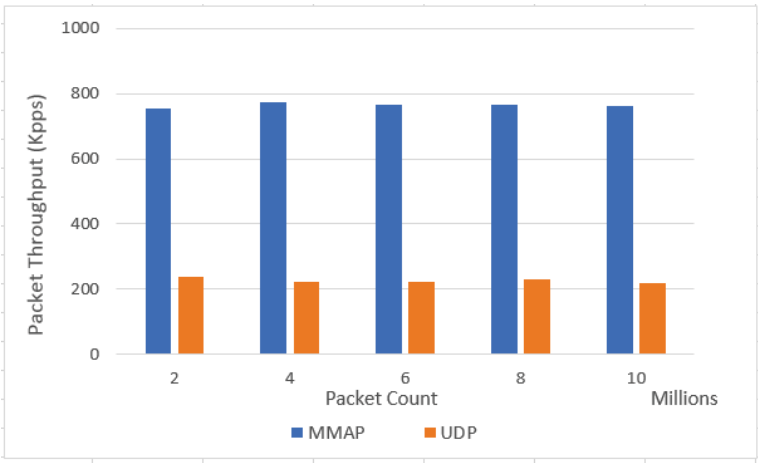

Figure 6.7: varying number of packets sent for bridge networking

Number of Packets Sent	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
2000000	2000000	1999839	1546736291.57	1546736293.67	2.623	1546737311.78	1546737319.45	8.469
4000000	4000000	3999901	1546736310.67	1546736315.78	5.17	1546737324.05	1546737344.18	18.054
6000000	6000000	5999892	1546736325.98	1546736332.07	7.817	1546737352.96	1546737378.69	26.999
8000000	8000000	7999283	1546736345.19	1546736355.24	10.457	1546737382.02	1546737416.30	34.886
10000000	10000000	9999289	1546736363.05	1546736376.97	13.089	1546737422.48	1546737477.78	45.496

Figure 6.8: Data for varying packets sent

Scenario II: By varying the transmission rate of packets sent

In this scenario, the transmission rate of the packet sent from the client is 200Mbps, 400Mbps, 600Mbs and 800 Mbps. The number of packets sent is kept constant at 1 million and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.9, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the transmission rate of packets sent from the client increases the packet throughput values for the mmap application are significantly higher than the values collected for the udp application. The slight packet drops in the udp application are due to the high CPU load, on account of running the client and server on the same host. The packet throughput for the UDP application has the highest value of 455.362 kpps. The packet throughput for the mmap application has the highest value of 721.041 kpps. The experiment data is shown in figure 6.10.

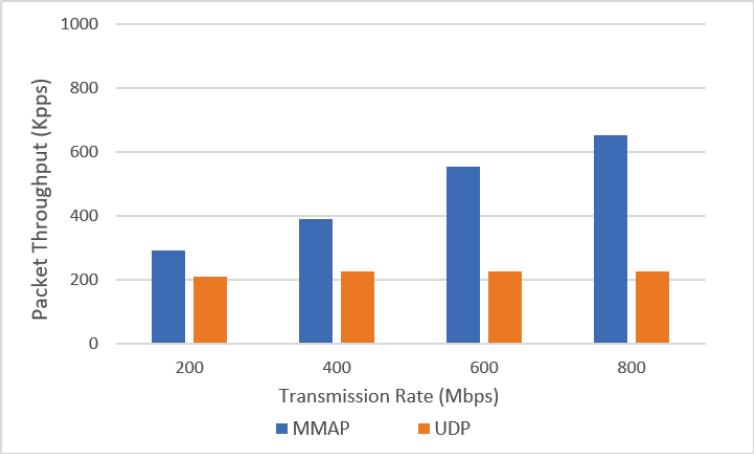

Figure 6.9: varying tx rate of packets sent for bridge networking

Transmission Rate	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
200	1000000	999623	1546738330.40	1546738331.12	1.302	1546738380.18	1546738385.02	4.812
400	1000000	999748	1546738342.54	1546738343.10	1.306	1546738391.65	1546738395.17	4.273
600	1000000	999245	1546738355.09	1546738357.03	1.296	1546738410.78	1546738415.65	4.84
800	1000000	999893	1546738367.09	1546738369.91	1.3	1546738426.74	1546738430.89	4.429

Figure 6.10: Data for varying tx rate

Scenario III: By varying the packet size

In this scenario, the packet size is varied by the client and the packet sizes for which values are collected are 64 bytes, 128 bytes, 256 bytes, 512 bytes and 1024 bytes. The packet transmission rate is kept constant at 500 Mbps and the number of packets sent is kept constant at 1 million. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.11, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the packet size increases the packet throughput values for the mmap application are significantly higher when compared to the values collected for the udp application. As the packet size increases the packet throughput values gradually decreases, with the highest value of 773.142 kpps for 64 Bytes and with the lowest value of 293.197 kpps for 1024 Bytes for mmap application. The udp application has the highest value of 232.948 kpps for 64 Bytes and the lowest value of 60.916 kpps for 1024 bytes. The performance degrades gradually as the packet size increases. The experiment data is shown in figure 6.12.

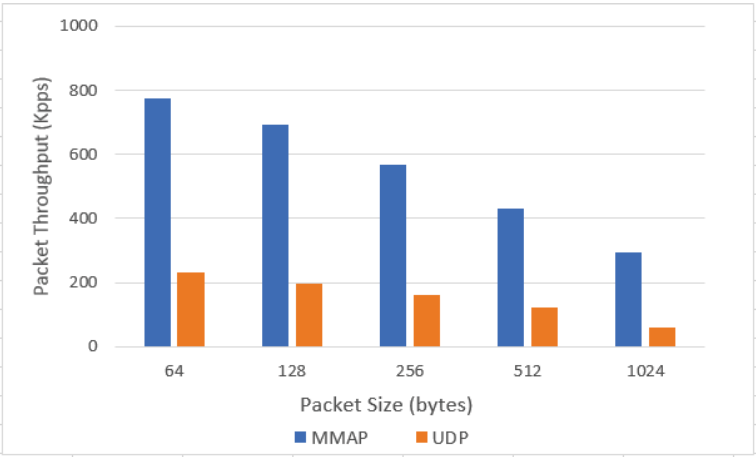

Figure 6.11: varying packet size of packets sent for bridge networking

Packet Size	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
64	1000000	999828	1546737618.49	1546737619.70	1.293	1546737692.48	1546737697.19	4.292
128	1000000	999739	1546737628.79	1546737629.04	1.44	1546737710.29	1546737715.78	5.133
256	1000000	999838	1546737640.84	1546737642.02	1.755	1546737722.41	1546737728.29	6.134
512	1000000	999873	1546737654.16	1546737655.10	2.309	1546737738.48	1546737746.09	8.224
1024	1000000	999807	1546737667.97	1546737670.58	3.417	1546737755.20	1546737766.73	16.416

Figure 6.12: Data for varying packet size

6.2.2 Setup II: Macvlan bridge networking mode

Scenario I: By varying the number of packets sent

In this scenario, the number of packets sent by the client is 2 million, 4 million, 6 million, 8 million and 10 million. The packet transmission rate is kept constant at 500 Mbps and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.13, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, with an increase in the number of packets sent, the packet throughput values for the mmap application are significantly higher compared to the values collected for the udp application. The slight packet drops in the udp application are due to the high cpu load, on account of running the client and server on the same host. The packet throughput for the UDP application has the highest value of 212.626 kpps with an average value of 205 kpps. The packet throughput for the mmap application has the highest value of 821.066 kpps with an average value of approximately equal to 820 kpps, which is 4 times higher when compared to the UDP application. The experiment data is shown in figure 6.14.

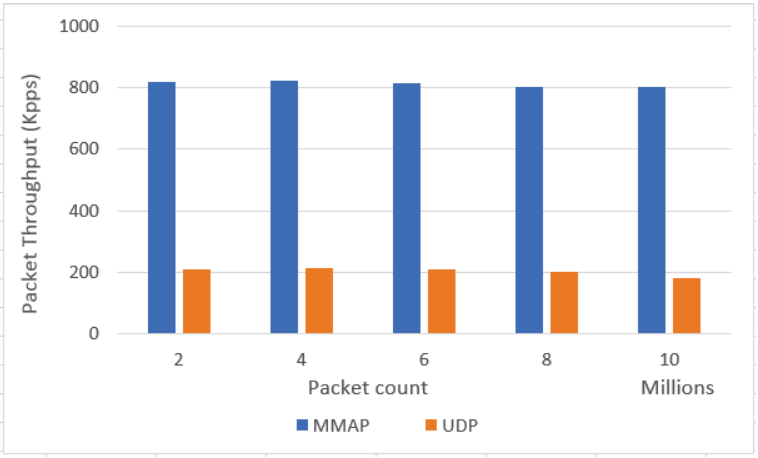

Figure 6.13: varying number of packets sent for macvlan bridge

Number of Packets Sent	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
2000000	2000000	1999829	1546717943.3	1546717945.06	2.44	15467183345.8	15467183354.6	9.567
4000000	4000000	3999890	1546718121.1	1546718126.7	4.868	15467184460.5	15467184478	18.812
6000000	6000000	5999876	1546718156.7	1546718165.3	7.382	15467184486.5	15467184516.1	28.557
8000000	8000000	7999072	1546718232.4	1546718242.6	9.654	15467185320.5	15467185360.4	40.095
10000000	10000000	9999893	1546718270.5	1546718282.7	12.487	15467185689.3	15467185745.2	55.392

Figure 6.14: Data for varying packet count

Scenario II: By varying the transmission rate of packets sent

In this scenario, the transmission rate of the packet sent from the client is 200Mbps, 400Mbps, 600Mbs and 800 Mbps. The number of packets sent is kept constant at 1 million and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.15, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the transmission rate of packets sent from the client increases the packet throughput values for the mmap application are significantly higher than the values collected for the udp application. The slight packet drops in the udp application are due to the high cpu load, on account of running the client and server on the same host. The packet throughput for the UDP application has the highest value of 489.874 kpps . The packet throughput for the mmap application has the highest value of 821.131 kpps. The experiment data is shown in figure 6.16.

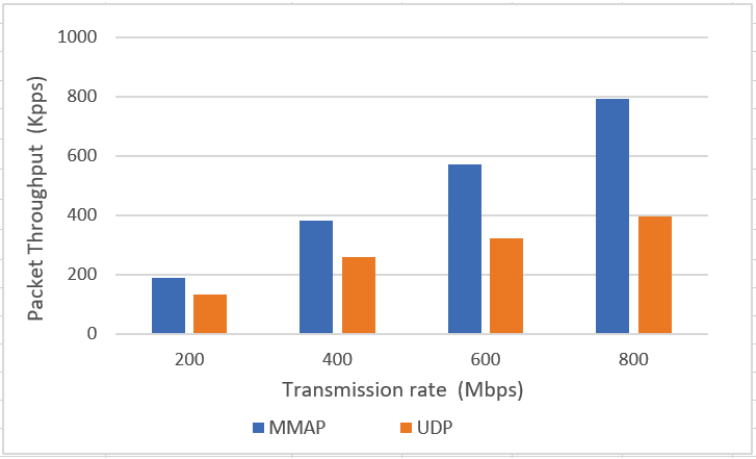

Figure 6.15: varying the tx rate of packets sent for macvlan bridge

Transmission Rate	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
200	1000000	999892	1546726271.63	1546726272.57	1.234	1546726576.79	15467265581.45	5.162
400	1000000	999779	1546726282.46	1546726283.78	1.258	1546726592.25	1546726594.31	4.724
600	1000000	999837	1546726290.82	1546726291.36	1.253	1546726603.80	1546726607.52	4.456
800	1000000	999839	1546726308.72	1546726309.29	1.249	1546726628.11	1546726632.51	4.767

Figure 6.16: Data for varying tx rate

Scenario III: By varying the packet size

In this scenario, the packet size is varied by the client and the packet sizes for which values are collected are 64 bytes, 128 bytes, 256 bytes, 512 bytes and 1024 bytes. The packet transmission rate is kept constant at 500 Mbps and the number of packets sent is kept constant at 1 million. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.17, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the packet size increases the packet throughput values for the mmap application are significantly higher when compared to the values collected for the udp application. As the packet size increases the packet throughput values gradually decreases, with the highest value of 796.908 kpps for 64 Bytes and with the lowest value of 297.062 kpps for 1024 Bytes for mmap application. The udp application has the highest value of 241.421 kpps for 64 Bytes and the lowest value of 60.916 kpps for 1024 bytes. The performance degrades gradually as the packet size increases. The experiment data is shown in figure 6.18.

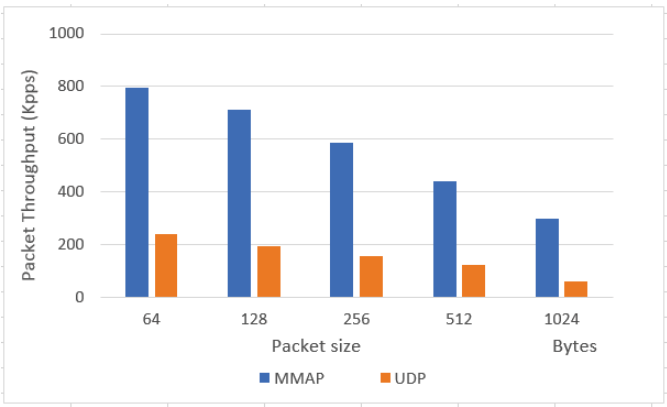

Figure 6.17: varying packet size of packets sent for macvlan bridge

Packet Size	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
64	1000000	998765	1546725190.32	1546725191.45	1.252	1546725970.53	1546725974.23	4.142
128	1000000	999237	1546725200.25	1546725202.98	1.404	1546725985.54	1546725991.86	5.142
256	1000000	998937	1546725220.75	1546725222.02	1.709	1546726001.85	1546726007.56	6.343
512	1000000	997387	1546725240.54	1546725243.62	2.262	1546726020.96	1546726029.02	8.224
1024	1000000	999345	1546725250.25	1546725254.74	3.3	1546726043.76	1546726059.64	16.154

Figure 6.18: Data for varying packet size

6.2.3 Setup III: Macvlan trunk bridge networking mode

Scenario I: By varying the number of packets sent

In this scenario, the number of packets sent by the client is 2 million, 4 million, 6 million, 8 million and 10 million. The packet transmission rate is kept constant at 500 Mbps and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.19, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, with an increase in the number of packets sent, the packet throughput values for the mmap application are significantly higher compared to the values collected for the udp application. The slight packet drops in the udp application are due to the high cpu load, on account of running the client and server on the same host. The packet throughput for the UDP application has the highest value of 240.270 kpps with an average value of 240 kpps. The packet throughput for the mmap application has the highest value of 850.413 kpps with an average value of approximately equal to 840 kpps, which is 3.5 times higher when compared to the UDP application. The experiment data is shown in figure 6.20.

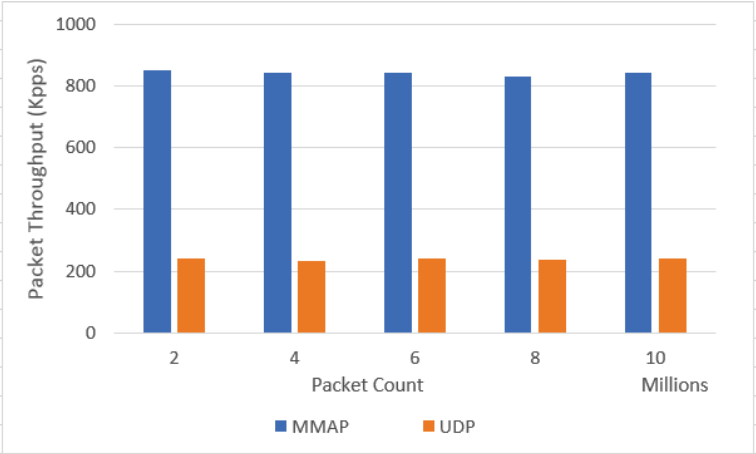

Figure 6.19: varying number of packets sent for macvlan trunk bridge

Number of Packets Sent	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
2000000	2000000	1999898	1546732159.52	1546732161.64	2.350	1546732875.45	1546732884.49	8.324
4000000	4000000	3999797	1546732175.39	1546732179.02	4.752	1546732895.09	1546732912.70	17.054
6000000	6000000	5999893	1546732190.31	1546732197.86	7.135	1546732958.65	1546732983.41	24.956
8000000	8000000	7999908	1546732209.97	1546732229.78	9.611	1546733301.47	1546733334.39	33.457
10000000	10000000	9999721	1546732246.48	1546732257.07	11.568	1546733349.67	1546733397.79	48.308

Figure 6.20: Data for varying packet count

Scenario II: By varying the transmission rate of packets sent

In this scenario, the transmission rate of the packet sent from the client is 200Mbps, 400Mbps, 600Mbs and 800 Mbps. The number of packets sent is kept constant at 1 million and the data size is kept constant at 64 Bytes. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.21, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the transmission rate of packets sent from the client increases the packet throughput values for the mmap application are significantly higher than the values collected for the udp application. The packet throughput for the UDP application has the highest value of 484.835 kpps. The packet throughput for the mmap application has the highest value of 811.940 kpps. The experiment data is shown in figure 6.22.

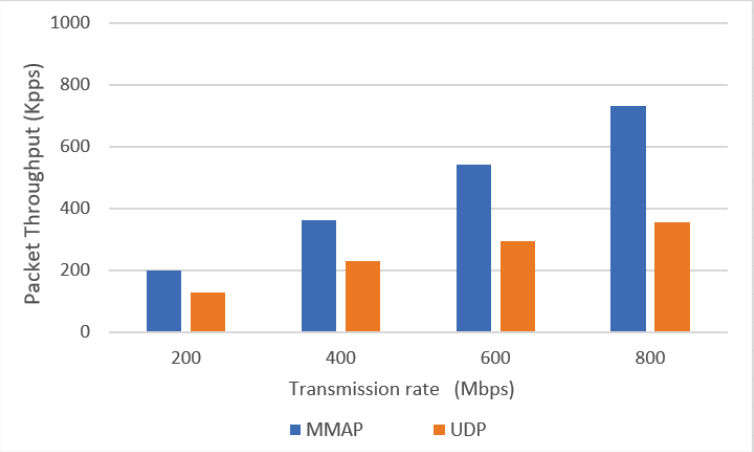

Figure 6.21: varying tx rate of packets sent for macvlan trunk bridge

Transmission Rate	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
200	1000000	999679	1546734552.87	1546734553.09	1.239	1546734791.74	1546734795.598	4.364
400	1000000	999739	1546734569.52	1546734570.62	1.228	1546734810.35	1546734815.548	4.244
600	1000000	998979	1546734582.49	1546734583.48	1.216	1546734830.59	1546734834..010	4.438
800	1000000	999587	1546734597.02	1546734598.51	1.228	1546734861.02	1546734865.597	4.258

Figure 6.22: Data for varying tx rate

Scenario III: By varying the packet size

In this scenario, the packet size is varied by the client and the packet sizes for which values are collected are 64 bytes, 128 bytes, 256 bytes, 512 bytes and 1024 bytes. The packet transmission rate is kept constant at 500 Mbps and the number of packets sent is kept constant at 1 million. The experiment is first done with the mmap client-server and is then repeated with a simple UDP client-server architecture. In figure Figure 6.23, a comparison of results obtained from the experiment is presented. The metrics collected in the experiment is packet throughput. The packet throughput values of mmap are shown against the values of a simple udp client-server application. From the graph, it is observed that, as the packet size increases the packet throughput values for the mmap application are significantly higher when compared to the values collected for the udp application. As the packet size increases the packet throughput values gradually decreases, with the highest value of 804.564 kpps for 64 Bytes and with the lowest value of 289.867 kpps for 1024 Bytes for mmap application. The udp application has the highest value of 233.519 kpps for 64 Bytes and the lowest value of 60.915 kpps for 1024 bytes. The performance degrades gradually as the packet size increases. The experiment data is shown in figure 6.24.

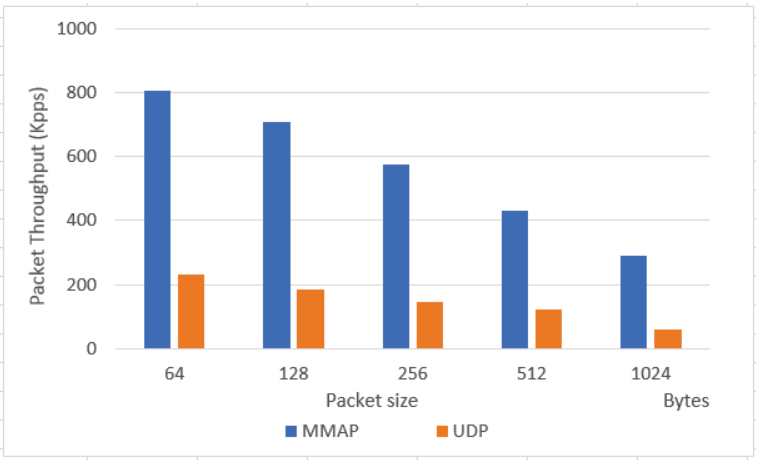

Figure 6.23: varying packet size of packets sent for macvlan trunk bridge

Packet Size	Received Packet Count		MMAP (Arrival Time in seconds)			UDP (Arrival Time in seconds)		
	MMAP	UDP	First packet	Last packet	Time Difference	First packet	Last packet	Time Difference
64	1000000	999238	1546733597.79	1546733598.98	1.242	1546733592.89	1546733596.03	4.276
128	1000000	999797	1546733610.05	1546733611.07	1.414	1546733620.48	1546733625.87	5.175
256	1000000	999379	1546733624.70	1546733625.89	1.740	1546733632.84	1546733638.79	6.921
512	1000000	999749	1546733636.89	1546733638.97	2.231	1546733657.91	1546733664.08	8.224
1024	1000000	999846	1546733651.58	1546733653.47	3.447	1546733679.84	1546733695.48	16.416

Figure 6.24: Data for varying packet size

6.3 Summary

This section gives a brief summary, based on the observations from the above graphs and data. As observed from the above graphs, it can be concluded that:

- By varying number of packets sent: From figure 6.1, 6.4, 6.7 as the number of packets sent increases the packet throughput for the implemented mmap application performs 4 times better when compared to the simple UDP client-server application.
- By varying transmission rate: From figure 6.2, 6.5, 6.8 as the transmission rate increases the packet throughput for the implemented mmap application performs 3.5 times better when compared to the simple UDP client-server application.
- By varying the packet size: From figure 6.3, 6.6, 6.9 as the packet size increases the packet throughput for the implemented mmap application performs 3.3 times better when compared to the simple UDP client-server application. It is also observed that the performance degrades as the packet sizes increases.

Chapter 7

Conclusions and Future Work

It is possible to improve packet processing rates using kernel bypass methods in a cloud-native infrastructure. From the study of various packet processing frameworks presented in Chapter 4, It can be concluded that packet mmap can be used to implement a kernel bypass solution which does not involve making hardware specific changes. Based on the results and analysis presented in Chapter 6, it can be concluded that the proposed solution performs significantly better in terms of packet throughput when compared with a simple UDP client-server application.

Furthermore, It can be concluded that from figure 6.19, 6.13, 6.7 as the number of packets sent from client increases the packet throughput for mmap application performs approximately 4 times better than the simple UDP client - server application. From figure 6.21, 6.15, 6.9 as the transmission rate increases the packet throughput for mmap application performs approximately 3.5 times better than the simple UDP client - server application. From figure 6.23,6.17,6.11 it can be concluded that as the packet size increases packet throuhghput gradually decreases and mmap application performs 3.3 times better that the simple UDP client - server application.

7.1 Research Questions and Answers

RQ 1: How can the packet throughput be improved in Docker containers?

A: This question is answered by doing a literature survey of various packet processing frameworks. Packet throughput can be improved in Docker containers by using packet processing frameworks. The identified packet processing frameworks are listed below:

- Packet mmap
- Netmap
- Data Plane Development Kit(DPDK)

- Open Onload
- NetSlice
- PF_RING
- Packet Shader

Based on the study of various packet processing frameworks presented in Chapter 4, the proposed solution uses packet mmap. Advantages of packet mmap are listed in Section 4.1. The main criteria in using the packet mmap method are that the solution did not involve making any hardware specific changes to the kernel. The main aim of implementing a solution independent of the underlying hardware is possible by using the packet mmap method.

RQ 2: What is the effect of the implemented method on the performance of networking in Docker containers?

A: A performance evaluation is done for the packet mmap and udp. The performance metrics packet throughput is evaluated for mmap and simple udp client-server architecture. The performance evaluation is initially carried out by running both client and server application on the same host. The performance evaluation is then carried out in three networking modes, as explained in Section 5.3. For different test bed setups, the experiment is done by varying 3 different parameters, varying the number of packets sent, varying the packet size and varying the tx rate. The packet throughput values are noted down in each case by varying one parameter and keeping the other two parameters at a constant value. So the effect of each parameter on packet throughput is evaluated. From the graphs in Chapter 6 and the summary of results provided in Section 6.3, it can be concluded that the proposed solution, packet mmap client-server application has better packet processing rates in the docker containers, compared to a simple UDP client-server application.

7.2 Future work

Integration with eBPF
As a future work it would be interesting to, study the **eBPF**(extended Berkeley Packet Filter) and integrate with the proposed solution. eBPF is used for debugging the kernel and carrying out performance analysis; programs can be attached to tracepoints, kprobes, and perf events. Because eBPF programs can access kernel data structures, developers can write and test new debugging code without having to recompile the kernel. eBPF can also be used to do high-performance packet processing by running eBPF programs at the lowest level of the network stack, immediately after the packet is received.

Integration with QUIC protocol
As a future work it would be interesting to, study the **QUIC** (**Q**uick **U**DP **I**nternet **C**onnections) protocol and integrate with the proposed solution. Using the QUIC protocol in cloud-native applications using high-speed networks, better performance can be achieved. QUIC is a new transport which reduces latency compared to that of TCP. On the surface, QUIC is very similar to TCP+TLS+HTTP/2 implemented on UDP. Because TCP is implemented in operating system kernels, and middlebox firmware, making significant changes to TCP is next to impossible. However, since QUIC is built on top of UDP, it suffers from no such limitations.

References

[1] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-Native Applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, Sep. 2017.

[2] P. Software, "Cloud-Native," Jan. 2017. [Online]. Available: https://pivotal.io/cloud-native

[3] M. Sneps-Sneppe and D. Namiot, "Micro-service Architecture for Emerging Telecom Applications," *International Journal of Open Information Technologies*, vol. 2, no. 11, pp. 34–38, Nov. 2014. [Online]. Available: http://injoit.org/index.php/j1/article/view/161

[4] J. Zhang, X. Lu, and D. K. Panda, "Performance Characterization of Hypervisor-and Container-Based Virtualization for HPC on SR-IOV Enabled InfiniBand Clusters," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1777–1784. [Online]. Available: doi.ieeecomputersociety.org/10.1109/IPDPSW.2016.178

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ISPASS.2015.7095802

[6] T. Bui, "Analysis of Docker Security," *arXiv:1501.02967 [cs]*, Jan. 2015, arXiv: 1501.02967. [Online]. Available: http://arxiv.org/abs/1501.02967

[7] "Case Studies." [Online]. Available: https://kubernetes.io/case-studies/

[8] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, Aug. 1997. [Online]. Available: http://doi.acm.org/10.1145/263326.263335

[9] K. K. Ramakrishnan, "Performance considerations in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203–219, Feb. 1993.

[10] "DPDK." [Online]. Available: https://dpdk.org/

[11] "OpenOnload." [Online]. Available: http://www.openonload.org/

[12] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 101–112. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342830

[13] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40. ACM, 2010, pp. 195–206.

[14] P. China Venkanna Varma, K. Venkata Kalyan Chakravarthy, and V. Valli Kumari, "Analysis of a Network IO Bottleneck in Big Data Environments | Apache Hadoop | Process (Computing)," *ACM*, vol. 3, no. c, pp. 24–28, Apr. 2016. [Online]. Available: https://www.scribd.com/document/330518106/Analysis-of-a-Network-IO-Bottleneck-in-Big-Data-Environments

[15] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2015, pp. 5–16.

[16] R. Rajesh, K. B. Ramia, and M. Kulkarni, "Integration of LwIP Stack over Intel(R) DPDK for High Throughput Packet Delivery to Applications," in *2014 Fifth International Symposium on Electronic System Design*, Dec. 2014, pp. 130–134.

[17] P. Rubens, "What are containers and why do you need them?" Jun. 2017. [Online]. Available: https://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html

[18] "Architecting Containers Part 1: Why Understanding User Space vs. Kernel Space Matters," Jul. 2015. [Online]. Available: https://rhelblog.redhat.com/2015/07/29/architecting-containers-part-1-user-space-vs-kernel-space/

[19] "What is a Container," Jan. 2017. [Online]. Available: https://www.docker.com/what-container

[20] "Chef and Puppet." [Online]. Available: https://www.chef.io/puppet/

[21] A. Hat, Red, "Ansible is Simple IT Automation." [Online]. Available: https://www.ansible.com

[22] "Introduction." [Online]. Available: https://www.vagrantup.com/intro/index.html

[23] "Apache Mesos." [Online]. Available: http://mesos.apache.org/

[24] "Production-Grade Container Orchestration - Kubernetes." [Online]. Available: https://kubernetes.io/

[25] "Linux Containers - LXC - Introduction." [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[26] "Linux Containers - LXD - Introduction." [Online]. Available: https://linuxcontainers.org/lxd/introduction/

[27] "rkt, a security-minded, standards-based container engine." [Online]. Available: https://coreos.com/rkt/

[28] "About Windows Containers | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/

[29] "Understanding Docker Networking Drivers and their use cases - Docker Blog." [Online]. Available: https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/

[30] "Use host networking," May 2018. [Online]. Available: https://docs.docker.com/network/host/

[31] "Use bridge networks," Apr. 2018. [Online]. Available: https://docs.docker.com/network/bridge/

[32] "How Docker Networking Works and the Importance of IPAM Functionality," Jun. 2016. [Online]. Available: https://community.infoblox.com/t5/Community-Blog/How-Docker-Networking-Works-and-the-Importance-of-IPAM/ba-p/6871

[33] "Use overlay networks," May 2018. [Online]. Available: https://docs.docker.com/network/overlay/

[34] "Use Macvlan networks | Docker Documentation." [Online]. Available: https://docs.docker.com/network/macvlan/

[35] "What Is Open vSwitch? — Open vSwitch 2.10.90 documentation." [Online]. Available: http://docs.openvswitch.org/en/latest/intro/what-is-ovs/

[36] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang, "Network Virtualization in Multi-tenant Datacenters," p. 15.

[37] S. Shanmugalingam, A. Ksentini, and P. Bertin, "DPDK Open vSwitch performance validation with mirroring feature," in *2016 23rd International Conference on Telecommunications (ICT)*, May 2016, pp. 1–6.

[38] Y. Zhao, T. Liu, G. Zhang, and J. Cui, "Optimization Research on Processes I/O Performance in Container-level Virtualization," in *2010 Sixth International Conference on Semantics, Knowledge and Grids*, Nov. 2010, pp. 113–120.

[39] K. Tsiamoura, W. Florian, and G. R. Daniel, "A survey of trends in fast packet processing," *Seminar Future Internet*, vol. 41, 2014.

[40] J. L. García-Dorado, F. Mata, J. Ramos, P. M. Santiago del Río, V. Moreno, and J. Aracil, "Datatraffic monitoring and analysis," E. Biersack, C. Callegari, and M. Matijasevic, Eds. Berlin, Heidelberg: Springer-Verlag, 2013, ch. High-Performance Network Traffic Processing Systems Using Commodity Hardware, pp. 3–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555672.2555674

[41] Packet_mmap. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt

[42] "The Netmap Project." [Online]. Available: http://info.iet.unipi.it/~luigi/netmap/

[43] "FreeBSD man page NETMAP(4)." [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=netmap&sektion=4

[44] "Overview — Data Plane Development Kit 18.05.0 documentation." [Online]. Available: https://dpdk.org/doc/guides/prog_guide/overview.html

[45] "Introduction — Data Plane Development Kit 18.05.0 documentation." [Online]. Available: http://dpdk.org/doc/guides/prog_guide/intro.html

[46] "Data Plane Development Kit: Performance Optimization Guidelines | Intel® Software." [Online]. Available: https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper

[47] "DPDK Boosts Packet Processing, Performance, and Throughput." [Online]. Available: https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html

[48] S. Pope and D. Riddoch, "Introduction to OpenOnload," no. 1, p. 8, 2011.

[49] "OpenOnload_user_guide.pdf." [Online]. Available: http://www.smallake.kr/wp-content/uploads/2015/12/SF-104474-CD-20_Onload_User_Guide.pdf

[50] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: Scalable multi-core packet processing in user-space," in *2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Oct. 2012, pp. 27–38.

[51] "DNA vs netmap," Jan. 2012. [Online]. Available: https://www.ntop.org/pf_ring/dna-vs-netmap/

[52] PF_ring Documentation — PF_ring 7.0 documentation. [Online]. Available: https://www.ntop.org/guides/pf_ring/

[53] "Packet-IO-Engine: A high-performance and batching-oriented device driver for Intel 82598/82599-based network interface cards, the work is done in cooperation with ANLAB and NDSL," May 2018, original-date: 2011-12-09T03:46:05Z. [Online]. Available: https://github.com/ANLAB-KAIST/Packet-IO-Engine

[54] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," p. 14.

Appendix A

Source code for mmap

A.1 mmap client source code

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <net/ethernet.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>
#include <netinet/udp.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>
#include <poll.h>
#include <pthread.h>
#include <netdb.h>
#include <sys/time.h>
#include <math.h>

/* params */
static char *str_devname = NULL;
static int c_packet_sz = 150;
static int c_packet_nb = 1000;
```

```
37 static int c_buffer_sz = 1024 * 8;
38 static int c_buffer_nb = 1024;
39 static int c_sndbuf_sz = 0;
40 static int c_mtu = 0;
41 static int mode_loss = 0;
42 static int mode_verbose = 0;
43 static int c_batch_nb = 1000;
44 static struct sockaddr_in dst_addr;
45 static long send_rate = 500; //in Mbps
46
47 /* globals */
48 volatile int fd_socket;
49 volatile int data_offset = 0;
50 volatile struct sockaddr_ll *ps_sockaddr = NULL;
51 volatile struct tpacket_hdr *ps_header_start;
52 volatile int shutdown_flag = 0;
53 struct tpacket_req s_packet_req;
54 struct sockaddr_in src_addr;
55 char ether_src[ETH_ALEN];
56 char ether_dst[ETH_ALEN];
57 struct sockaddr addr;
58
59 short src_port = 6666;
60
61 uint8_t eth_addr[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
62 int values[6];
63 int i_index = 0;
64 int tot_pkt = 0;
65 unsigned long long tot_byt = 0;
66 int bat_num = 0;
67
68 int task_send(int blocking);
69 ssize_t sendudp(char *data, ssize_t len, const struct sockaddr_in *
      dest);
70
71 static void usage()
72 {
73     fprintf(stderr,
74             "Usage: ./packet_mmap [OPTION] [INTERFACE]\n"
75             " -h\tshow this help\n"
76             " -r\t set tx rate in Mbps\n"
77             " -y\tset packets per batch\n"
78             " -s\tset packet size\n"
79             " -c\tset batch count\n"
80             " -m\tset mtu\n"
81             " -b\tset buffer size\n"
82             " -n\tset buffer count\n"
83             " -z\tset socket buffer size\n"
84             " -a\tset destination IP address\n"
85             " -p\tset destination port\n"
86             " -l\tdiscard wrong packets\n"
```

```
87              " -v\tbe verbose\n");
88  }
89
90  void getargs(int argc, char **argv)
91  {
92      int c;
93      opterr = 0;
94      while ((c = getopt(argc, argv, "e:s:m:b:B:n:c:z:y:r:j:a:p:M:
        vhgtl")) != EOF)
95      {
96          switch (c)
97          {
98          case 'r':
99              send_rate = strtoul(optarg, NULL, 0);
100             break;
101         case 'y':
102             c_batch_nb = strtoul(optarg, NULL, 0);
103             break;
104         case 's':
105             c_packet_sz = strtoul(optarg, NULL, 0);
106             break;
107         case 'c':
108             c_packet_nb = strtoul(optarg, NULL, 0);
109             break;
110         case 'b':
111             c_buffer_sz = strtoul(optarg, NULL, 0);
112             break;
113         case 'n':
114             c_buffer_nb = strtoul(optarg, NULL, 0);
115             break;
116         case 'z':
117             c_sndbuf_sz = strtoul(optarg, NULL, 0);
118             break;
119         case 'm':
120             c_mtu = strtoul(optarg, NULL, 0);
121             break;
122         case 'a':
123         {
124             struct hostent h_dent;
125             memcpy(&h_dent, gethostbyname(optarg), sizeof(h_dent));
126             memcpy(&dst_addr.sin_addr, h_dent.h_addr, sizeof(
        dst_addr.sin_addr));
127             break;
128         }
129         case 'p':
130             dst_addr.sin_port = atoi(optarg);
131             break;
132             break;
133         case 'l':
134             mode_loss = 1;
135             break;
```

```
136         case 'v':
137             mode_verbose = 1;
138             break;
139         case 'h':
140             usage();
141             exit(EXIT_FAILURE);
142             break;
143         case '?':
144             if (isprint(optopt))
145             {
146                 fprintf(stderr,
147                         "ERROR: unrecognised option \"%c\"\n",
148                         (char)optopt);
149                 exit(EXIT_FAILURE);
150             }
151             break;
152         default:
153             fprintf(stderr, "ERROR: unrecognised command line option
    \n");
154             exit(EXIT_FAILURE);
155             break;
156         }
157     }
158     /* take first residual non option argv element as interface name
    . */
159     if (optind < argc)
160     {
161         str_devname = argv[optind];
162     }
163 
164     if (!str_devname)
165     {
166         fprintf(stderr, "ERROR: No interface was specified\n");
167         usage();
168         exit(EXIT_FAILURE);
169     }
170 
171     printf("CURRENT SETTINGS:\n");
172     printf("str_devname:       %s\n", str_devname);
173     printf("c_batch_nb :       %d\n", c_batch_nb);
174     printf("c_packet_sz:       %d\n", c_packet_sz);
175     printf("c_buffer_sz:       %d\n", c_buffer_sz);
176     printf("c_buffer_nb:       %d\n", c_buffer_nb);
177     printf("c_packet_nb count: %d\n", c_packet_nb);
178     printf("c_mtu:             %d\n", c_mtu);
179     printf("c_sndbuf_sz:       %d\n", c_sndbuf_sz);
180     printf("mode_loss:         %d\n", mode_loss);
181 }
182 
183 struct sockaddr get_dest_mac(const struct sockaddr_in *ip, char *dev
    )
```

```
184 {
185     struct arpreq req;
186     bzero(&req, sizeof(req));
187     req.arp_pa = *(struct sockaddr *)ip;
188     int sock = socket(AF_INET, SOCK_DGRAM, 0);
189     if (sock < 0)
190     {
191         perror("socket");
192         exit(1);
193     }
194     strncpy(req.arp_dev, dev, sizeof(req.arp_dev));
195     if (ioctl(sock, SIOCGARP, &req) < 0)
196     {
197         perror("SIOCGARP");
198         exit(1);
199     }
200     close(sock);
201     if (req.arp_flags & ATF_COM)
202         return req.arp_ha;
203     fprintf(stderr, "arp request flags: %d\n", req.arp_flags);
204     exit(1);
205     return req.arp_ha;
206 }
207 
208 struct sockaddr get_if_addr(char *dev)
209 {
210     struct ifreq if_mac;
211     memset(&if_mac, 0, sizeof(struct ifreq));
212     strncpy(if_mac.ifr_name, dev, IFNAMSIZ - 1);
213     int sock = socket(AF_INET, SOCK_DGRAM, 0);
214     if (sock < 0)
215     {
216         perror("socket");
217         exit(1);
218     }
219     if (ioctl(sock, SIOCGIFHWADDR, &if_mac) < 0)
220     {
221         perror("SIOCGIFHWADDR");
222         exit(1);
223     }
224     close(sock);
225     return if_mac.ifr_hwaddr;
226 }
227 
228 struct sockaddr_in get_if_ip(char *dev)
229 {
230     struct ifreq if_ip;
231     memset(&if_ip, 0, sizeof(struct ifreq));
232     if_ip.ifr_addr.sa_family = AF_INET;
233     strncpy(if_ip.ifr_name, dev, IFNAMSIZ - 1);
234     int sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```
235     if (sock < 0)
236     {
237         perror("socket");
238         exit(1);
239     }
240     if (ioctl(sock, SIOCGIFADDR, &if_ip) < 0)
241     {
242         perror("SIOCGIFADDR");
243         exit(1);
244     }
245     close(sock);
246     return *(struct sockaddr_in *)&if_ip.ifr_addr;
247 }
248 
249 // Function for checksum calculation. From the RFC,
250 // the checksum algorithm is:
251 //  "The checksum field is the 16 bit one's complement of the one's
252 //  complement sum of all 16 bit words in the header.  For purposes
    of
253 //  computing the checksum, the value of the checksum field is zero
    ."
254 unsigned short csum(unsigned short *buf, int nwords)
255 { //
256     unsigned long sum;
257     for (sum = 0; nwords > 0; nwords--)
258         sum += *buf++;
259     sum = (sum >> 16) + (sum & 0xffff);
260     sum += (sum >> 16);
261     //printf("checksum is : %hu\n",(unsigned short)(~sum));
262     return (unsigned short)(~sum);
263 }
264 
265 int main(int argc, char **argv)
266 {
267     uint32_t size; // opt_len;
268     int ec;
269     struct sockaddr_ll my_addr, peer_addr;
270     struct ifreq s_ifr; /* points to one interface returned from
    ioctl */
271     //int len;
272     int i_ifindex;
273     int mode_socket;
274     int tmp;
275     int i_nb_error;
276     /* get configuration */
277     getargs(argc, argv);
278 
279     send_rate *= pow(10, 6);
280 
281     src_addr = get_if_ip(str_devname);
282     struct sockaddr hwaddr = get_if_addr(str_devname);
```

```
283     memcpy(ether_src, hwaddr.sa_data, sizeof(ether_src));
284     dst_addr.sin_family = AF_INET;
285 
286     printf("\nSTARTING TEST:\n");
287 
288     if (mode_dgram)
289     {
290         printf("DGRAM Socket!!\n");
291         mode_socket = SOCK_DGRAM;
292     }
293     else
294     {
295         printf("RAW Socket!!\n");
296         mode_socket = SOCK_RAW;
297     }
298     fd_socket = socket(PF_PACKET, mode_socket, htons(ETH_P_ALL));
299     if (fd_socket == -1)
300     {
301         perror("socket");
302         return EXIT_FAILURE;
303     }
304 
305     /* start socket config: device and mtu */
306 
307     /* initialize interface struct */
308     strncpy(s_ifr.ifr_name, str_devname, sizeof(s_ifr.ifr_name));
309 
310     /* Get the index of the network interface. */
311     ec = ioctl(fd_socket, SIOCGIFINDEX, &s_ifr);
312     if (ec == -1)
313     {
314         perror("iotcl");
315         return EXIT_FAILURE;
316     }
317     /* update with interface index */
318     i_ifindex = s_ifr.ifr_ifindex;
319 
320     /* new mtu value */
321     if (c_mtu)
322     {
323         s_ifr.ifr_mtu = c_mtu;
324         /* update the mtu through ioctl */
325         ec = ioctl(fd_socket, SIOCSIFMTU, &s_ifr);
326         if (ec == -1)
327         {
328             perror("iotcl");
329             return EXIT_FAILURE;
330         }
331     }
332 
333     /* set sockaddr info */
```

```
334     memset(&my_addr, 0, sizeof(struct sockaddr_ll));
335     my_addr.sll_family = AF_PACKET;
336     my_addr.sll_protocol = htons(ETH_P_ALL);
337     my_addr.sll_ifindex = i_ifindex;
338 
339     /* bind port */
340     if (bind(fd_socket, (struct sockaddr *)&my_addr, sizeof(struct
    sockaddr_ll)) == -1)
341     {
342         perror("bind");
343         return EXIT_FAILURE;
344     }
345 
346     /* prepare Tx ring request */
347     s_packet_req.tp_block_size = c_buffer_sz;
348     s_packet_req.tp_frame_size = c_buffer_sz;
349     s_packet_req.tp_block_nr = c_buffer_nb;
350     s_packet_req.tp_frame_nr = c_buffer_nb;
351 
352     /* calculate memory to mmap in the kernel */
353     size = s_packet_req.tp_block_size * s_packet_req.tp_block_nr;
354 
355     /* set packet loss option */
356     tmp = mode_loss;
357     if (setsockopt(fd_socket, SOL_PACKET, PACKET_LOSS,
358                    (char *)&tmp, sizeof(tmp)) < 0)
359     {
360         perror("setsockopt: PACKET_LOSS");
361         return EXIT_FAILURE;
362     }
363 
364     /* send TX ring request */
365     if (setsockopt(fd_socket, SOL_PACKET, PACKET_TX_RING,
366                    (char *)&s_packet_req, sizeof(s_packet_req)) < 0)
367     {
368         perror("setsockopt: PACKET_TX_RING");
369         return EXIT_FAILURE;
370     }
371 
372     /* change send buffer size */
373     if (c_sndbuf_sz)
374     {
375         printf("send buff size = %d\n", c_sndbuf_sz);
376         if (setsockopt(fd_socket, SOL_SOCKET, SO_SNDBUF, &
    c_sndbuf_sz,
377                        sizeof(c_sndbuf_sz)) < 0)
378         {
379             perror("getsockopt: SO_SNDBUF");
380             return EXIT_FAILURE;
381         }
382     }
```

```
383
384     /* get data offset */
385     data_offset = TPACKET_HDRLEN - sizeof(struct sockaddr_ll);
386     printf("data offset = %d bytes\n", data_offset);
387
388     /* mmap Tx ring buffers memory */
389     ps_header_start = mmap(0, size, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd_socket, 0);
390     if (ps_header_start == (void *)-1)
391     {
392         perror("mmap");
393         return EXIT_FAILURE;
394     }
395
396     //ping the dest address for arp entry
397     char cmd[50];
398     strcpy(cmd, "ping -c 1 ");
399     strcat(cmd, inet_ntoa(dst_addr.sin_addr));
400     printf("%s \n", cmd);
401     int sys = system(cmd);
402
403     if (sys == -1)
404     {
405         perror("ping");
406         return EXIT_FAILURE;
407     }
408
409     addr = get_dest_mac(&dst_addr, str_devname);
410
411     struct timeval start, now, btw;
412     gettimeofday(&start, NULL);
413
414     int i;
415     char buf[c_packet_sz];
416     //char *str;
417     memset(buf, 1, c_packet_sz);
418     for (i = 0; i < c_packet_nb; i++)
419     {
420         sendudp(buf, sizeof(buf), &dst_addr);
421
422         //rate control
423         gettimeofday(&btw, NULL);
424         double duration_btw;
425         unsigned long long rate_btw;
426         duration_btw = (btw.tv_sec - start.tv_sec);
427         duration_btw *= 1000000;
428         duration_btw += btw.tv_usec - start.tv_usec;
429         duration_btw /= 1000000;
430         rate_btw = (c_packet_sz + 42) * c_batch_nb * (i + 1) * 8;
431         rate_btw = rate_btw / duration_btw;
432         if (rate_btw > send_rate)
```

```
433         {
434             double int_delay = (c_packet_sz + 42) * c_batch_nb * (i
    + 1) * 8;
435             int_delay = int_delay / send_rate;
436             int_delay = int_delay - duration_btw;
437             if (int_delay > 0)
438                 usleep(int_delay * 1000000);
439         }
440     }
441
442     gettimeofday(&now, NULL);
443     double rate_now, duration_now, pkt_thr;
444     duration_now = (now.tv_sec - start.tv_sec);
445     printf("duration: %lf\t start: %ld\t now: %ld\n", duration_now,
    start.tv_sec, now.tv_sec);
446
447     duration_now *= 1000000;
448     duration_now += (now.tv_usec - start.tv_usec);
449     duration_now /= 1000000;
450
451     rate_now = tot_byt * 8;
452     printf("rate:%lf\tduration: %lf\t start: %ld\t now: %ld\n",
    rate_now, duration_now, start.tv_sec, now.tv_sec);
453
454     rate_now = rate_now / duration_now;
455
456     pkt_thr = tot_pkt / duration_now;
457     printf("[FINAL: ]sent %d packets, bytes: %llu, txrate: %lf Mbps,
     pkt throughput: %lf pps\n", tot_pkt, tot_byt, rate_now /
    1000000, pkt_thr);
458
459     i_nb_error = 0;
460     for (i = 0; i < c_buffer_nb; i++)
461     {
462         struct tpacket_hdr *ps_header;
463         ps_header = ((struct tpacket_hdr *)((void *)ps_header_start
    + (c_buffer_sz * i)));
464         switch ((volatile uint32_t)ps_header->tp_status)
465         {
466         case TP_STATUS_SEND_REQUEST:
467             printf("A frame has not been sent %p\n", ps_header);
468             i_nb_error++;
469             break;
470
471         case TP_STATUS_LOSING:
472             printf("An error has occured during transfer\n");
473             i_nb_error++;
474             break;
475
476         default:
477             break;
```

```
478         }
479     }
480     printf("END (number of error:%d)\n", i_nb_error);
481 
482     if (munmap((void *)ps_header_start, c_buffer_sz * c_buffer_nb))
483     {
484         perror("munmap");
485         return 1;
486     }
487 
488     if (close(fd_socket))
489     {
490         perror("close");
491         return 1;
492     }
493 
494     /* display header of all blocks */
495     return EXIT_SUCCESS;
496 }
497 
498 /* This task will call send() procedure */
499 int task_send(int blocking)
500 {
501     int ec_send;
502 
503     if (blocking)
504         printf("start send() thread\n");
505 
506     do
507     {
508         /* send all buffers with TP_STATUS_SEND_REQUEST */
509         /* Wait end of transfer */
510         if (mode_verbose)
511             printf("send() start\n");
512         ec_send = sendto(fd_socket,
513                          NULL,
514                          0,
515                          0,
516                          (struct sockaddr *)ps_sockaddr,
517                          sizeof(struct sockaddr_ll));
518         if (mode_verbose)
519             printf("send() end (ec=%d)\n", ec_send);
520 
521         if (ec_send < 0)
522         {
523             perror("send");
524             break;
525         }
526         else
527         {
528             tot_byt += ec_send;
```

```
529             fflush(0);
530         }
531
532     } while (blocking && !shutdown_flag);
533
534     if (blocking)
535         printf("end of task send()\n");
536
537     return ec_send;
538 }
539
540 void *get_free_buffer()
541 {
542     int i;
543     for (i = 0; i < c_buffer_nb; i++)
544     {
545         struct tpacket_hdr *ps_header = ((struct tpacket_hdr *)((
    void *)ps_header_start + (c_buffer_sz * i_index)));
546         char *data = ((void *)ps_header) + data_offset;
547
548         switch ((volatile uint32_t)ps_header->tp_status)
549         {
550         case TP_STATUS_AVAILABLE:
551             /* fill data in buffer */
552             return (void *)data;
553             break;
554
555         case TP_STATUS_WRONG_FORMAT:
556             printf("An error has occured during transfer\n");
557             exit(EXIT_FAILURE);
558             break;
559
560         default:
561             //usleep(0);
562             break;
563         }
564     }
565     return NULL;
566 }
567
568 struct iphdr *construct_ip(struct iphdr *ip,
569                            const struct sockaddr_in *dest, ssize_t
    packet_len)
570 {
571     memset(ip, 0, sizeof(*ip));
572     ip->ihl = 5;
573     ip->version = 4;
574     ip->tos = 4; // Low delay
575     ip->tot_len = htons(packet_len);
576     ip->id = htons(54321);
577     ip->ttl = 64;       // hops
```

```
578     ip->protocol = 17; // UDP
579     ip->saddr = src_addr.sin_addr.s_addr;
580     ip->daddr = dest->sin_addr.s_addr;
581     ip->frag_off |= htons(IP_DF);
582     return ip;
583 }
584 
585 struct udphdr *construct_udp(struct udphdr *udp,
586                              short dst_port, ssize_t packet_len)
587 {
588     memset(udp, 0, sizeof(*udp));
589     udp->source = htons(src_port);
590     // Destination port number
591     udp->dest = htons(dst_port);
592     udp->len = htons(packet_len);
593     return udp;
594 }
595 
596 struct ether_header *construct_ether(struct ether_header *ether,
    const struct sockaddr_in *dest_ip)
597 {
598     memcpy(ether->ether_shost, ether_src, sizeof(ether_src));
599 
600     ether->ether_dhost[0] = addr.sa_data[0];
601     ether->ether_dhost[1] = addr.sa_data[1];
602     ether->ether_dhost[2] = addr.sa_data[2];
603     ether->ether_dhost[3] = addr.sa_data[3];
604     ether->ether_dhost[4] = addr.sa_data[4];
605     ether->ether_dhost[5] = addr.sa_data[5];
606     ether->ether_type = htons(ETH_P_IP);
607 
608     return ether;
609 }
610 
611 /* send a UDP packet. */
612 ssize_t sendudp(char *payload, ssize_t len, const struct sockaddr_in
      *dest)
613 {
614 
615     for (int i = 0; i < c_batch_nb; i++)
616     {
617         /* get free buffer */
618         void *data = get_free_buffer();
619         while (data == NULL)
620         {
621             usleep(1);
622             data = get_free_buffer();
623         }
624 
625         struct ether_header *ether = (struct ether_header *)data;
626         struct iphdr *ip = (struct iphdr *)(data + sizeof(*ether));
```

```
627         struct udphdr *udp = (struct udphdr *)(((char *)ip) + sizeof
    (struct iphdr));
628         char *payload_ptr = ((char *)udp) + sizeof(*udp);
629         memcpy(payload_ptr, payload, len);
630         short packet_len = (short)len + sizeof(*ip) + sizeof(*udp);
631         construct_ether(ether, dest);
632         construct_ip(ip, dest, packet_len);
633         construct_udp(udp, dest->sin_port, packet_len - sizeof(
    struct iphdr));
634         // Calculate the checksum for integrity
635         ip->check = csum((unsigned short *)ip, packet_len);
636
637         struct tpacket_hdr *ps_header = (struct tpacket_hdr *)(data
    - data_offset);
638         /* update packet len */
639         ps_header->tp_len = packet_len + sizeof(*ether);
640         /* set header flag to USER (trigs xmit)*/
641         ps_header->tp_status = TP_STATUS_SEND_REQUEST;
642
643         i_index++;
644         if (i_index >= c_buffer_nb)
645         {
646             i_index = 0;
647         }
648         tot_pkt++;
649     }
650     bat_num++;
651     int ec_send = task_send(0);
652     if (ec_send == 0)
653     {
654         fprintf(stderr, "ERROR: error in data transfer.....");
655         exit(EXIT_FAILURE);
656     }
657     return len;
658 }
```

Listing A.1: mmap client

A.2 mmap server source code

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <poll.h>
#include <unistd.h>
#include <signal.h>
#include <inttypes.h>
#include <sys/socket.h>
#include <sys/mman.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <signal.h>

#include <netinet/udp.h>

#ifndef likely
#define likely(x) __builtin_expect(!!(x), 1)
#endif
#ifndef unlikely
#define unlikely(x) __builtin_expect(!!(x), 0)
#endif

struct block_desc
{
    uint32_t version;
    uint32_t offset_to_priv;
    struct tpacket_hdr_v1 h1;
};

struct ring
{
    struct iovec *rd;
    uint8_t *map;
    struct tpacket_req3 req;
};

static unsigned long packets_total = 0, bytes_total = 0;
static sig_atomic_t sigint = 0;
static unsigned long bytes = 0;
static double start_time;
static double end_time;

static int iphdrlen;
```

```
50
51 static void sighandler(int num)
52 {
53     sigint = 1;
54 }
55
56 static int setup_socket(struct ring *ring, char *netdev)
57 {
58     int err, i, fd, v = TPACKET_V3;
59     struct sockaddr_ll ll;
60     unsigned int blocksiz = 1 << 22, framesiz = 1 << 11;
61     unsigned int blocknum = 64;
62
63     fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
64     if (fd < 0)
65     {
66         perror("socket");
67         exit(1);
68     }
69
70     err = setsockopt(fd, SOL_PACKET, PACKET_VERSION, &v, sizeof(v));
71     if (err < 0)
72     {
73         perror("setsockopt");
74         exit(1);
75     }
76
77     memset(&ring->req, 0, sizeof(ring->req));
78     ring->req.tp_block_size = blocksiz;
79     ring->req.tp_frame_size = framesiz;
80     ring->req.tp_block_nr = blocknum;
81     ring->req.tp_frame_nr = (blocksiz * blocknum) / framesiz;
82     ring->req.tp_retire_blk_tov = 60;
83     ring->req.tp_feature_req_word = TP_FT_REQ_FILL_RXHASH;
84
85     err = setsockopt(fd, SOL_PACKET, PACKET_RX_RING, &ring->req,
86                      sizeof(ring->req));
87     if (err < 0)
88     {
89         perror("setsockopt");
90         exit(1);
91     }
92
93     ring->map = mmap(NULL, ring->req.tp_block_size * ring->req.
    tp_block_nr,
94                      PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
95     if (ring->map == MAP_FAILED)
96     {
97         perror("mmap");
98         exit(1);
99     }
```

```
100
101     ring->rd = malloc(ring->req.tp_block_nr * sizeof(*ring->rd));
102     assert(ring->rd);
103     for (i = 0; i < ring->req.tp_block_nr; ++i)
104     {
105         ring->rd[i].iov_base = ring->map + (i * ring->req.
    tp_block_size);
106         ring->rd[i].iov_len = ring->req.tp_block_size;
107     }
108
109     memset(&ll, 0, sizeof(ll));
110     ll.sll_family = PF_PACKET;
111     ll.sll_protocol = htons(ETH_P_ALL);
112     ll.sll_ifindex = if_nametoindex(netdev);
113     ll.sll_hatype = 0;
114     ll.sll_pkttype = 0;
115     ll.sll_halen = 0;
116
117     err = bind(fd, (struct sockaddr *)&ll, sizeof(ll));
118     if (err < 0)
119     {
120         perror("bind");
121         exit(1);
122     }
123
124     return fd;
125 }
126
127 static void display(struct tpacket3_hdr *ppd, int buflen)
128 {
129
130     struct timeval time;
131     gettimeofday(&time, NULL);
132     double timenow;
133     timenow = time.tv_sec * 1000000;
134     timenow += time.tv_usec;
135     timenow /= 1000000;
136
137     struct ethhdr *eth = (struct ethhdr *)((uint8_t *)ppd + ppd->
    tp_mac);
138     struct iphdr *ip = (struct iphdr *)((uint8_t *)eth + ETH_HLEN);
139     char sbuff[NI_MAXHOST], dbuff[NI_MAXHOST];
140
141     iphdrlen = ip->ihl * 4;
142
143     if (eth->h_proto == htons(ETH_P_IP))
144     {
145         struct sockaddr_in ss, sd;
146
147         memset(&ss, 0, sizeof(ss));
148         ss.sin_family = PF_INET;
```

```
149         ss.sin_addr.s_addr = ip->saddr;
150         getnameinfo((struct sockaddr *)&ss, sizeof(ss),
151                     sbuff, sizeof(sbuff), NULL, 0, NI_NUMERICHOST);
152 
153         memset(&sd, 0, sizeof(sd));
154         sd.sin_family = PF_INET;
155         sd.sin_addr.s_addr = ip->daddr;
156         getnameinfo((struct sockaddr *)&sd, sizeof(sd),
157                     dbuff, sizeof(dbuff), NULL, 0, NI_NUMERICHOST);
158     }
159 
160     struct udphdr *udp = (struct udphdr *)(struct udphdr *)(((char
    *)ip) + sizeof(struct iphdr));
161     if (ntohs(udp->dest) == 9000)
162     {
163 
164         if (packets_total == 0)
165         {
166             start_time = timenow;
167         }
168         end_time = timenow;
169         bytes = ppd->tp_snaplen;
170         bytes_total += bytes;
171         packets_total++;
172     }
173 }
174 
175 static void walk_block(struct block_desc *pbd, const int block_num)
176 {
177     int num_pkts = pbd->h1.num_pkts, i, buflen;
178 
179     struct tpacket3_hdr *ppd;
180 
181     ppd = (struct tpacket3_hdr *)((uint8_t *)pbd + pbd->h1.
    offset_to_first_pkt);
182     for (i = 0; i < num_pkts; i++)
183     {
184 
185         buflen = ppd->tp_len;
186         display(ppd, buflen);
187 
188         ppd = (struct tpacket3_hdr *)((uint8_t *)ppd + ppd->
    tp_next_offset);
189         fflush(0);
190     }
191 }
192 
193 static void flush_block(struct block_desc *pbd)
194 {
195     pbd->h1.block_status = TP_STATUS_KERNEL;
196 }
```

```
197
198 static void teardown_socket(struct ring *ring, int fd)
199 {
200     munmap(ring->map, ring->req.tp_block_size * ring->req.
    tp_block_nr);
201     free(ring->rd);
202     close(fd);
203 }
204
205 int main(int argc, char **argp)
206 {
207     int fd, err;
208     socklen_t len;
209     struct ring ring;
210     struct pollfd pfd;
211     unsigned int block_num = 0, blocks = 64;
212     struct block_desc *pbd;
213     struct tpacket_stats_v3 stats;
214     double duration, pkt_thr = 0;
215
216     if (argc != 2)
217     {
218         fprintf(stderr, "Usage: %s INTERFACE\n", argp[0]);
219         return EXIT_FAILURE;
220     }
221
222     signal(SIGINT, sighandler);
223
224     memset(&ring, 0, sizeof(ring));
225     fd = setup_socket(&ring, argp[argc - 1]);
226     assert(fd > 0);
227
228     memset(&pfd, 0, sizeof(pfd));
229     pfd.fd = fd;
230     pfd.events = POLLIN | POLLERR;
231     pfd.revents = 0;
232
233     while (likely(!sigint))
234     {
235         pbd = (struct block_desc *)ring.rd[block_num].iov_base;
236
237         if ((pbd->h1.block_status & TP_STATUS_USER) == 0)
238         {
239             poll(&pfd, 1, -1);
240             continue;
241         }
242
243         walk_block(pbd, block_num);
244         flush_block(pbd);
245         block_num = (block_num + 1) % blocks;
246     }
```

```
247
248     len = sizeof(stats);
249     err = getsockopt(fd, SOL_PACKET, PACKET_STATISTICS, &stats, &len
        );
250     if (err < 0)
251     {
252         perror("getsockopt");
253         exit(1);
254     }
255
256     fflush(stdout);
257     duration = end_time - start_time;
258     pkt_thr = packets_total / duration;
259
260     printf("\nReceived %lu packets, %lu bytes, %u dropped,
        freeze_q_cnt: %u\npkt thr: %lf (pps),duration: %lf (s)\n",
261            packets_total, bytes_total, stats.tp_drops,
262            stats.tp_freeze_q_cnt, pkt_thr, duration);
263
264     teardown_socket(&ring, fd);
265     return 0;
266 }
```

Listing A.2: mmap server

Appendix B

Source code for UDP

B.1 udp client source code

```
/*
 * udpclient.c - A simple UDP client
 * usage: udpclient <host> <port> <BUFSIZE> <pkt count> <send rate
    in Kbps>
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>
#include <math.h>

/*
 * error - wrapper for perror
 */
void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char **argv)
{
    int sockfd, portno, n, pkt_count = 0;
    int BUFSIZE;
    int serverlen;
    int tot_pkt = 0;
    unsigned long long tot_byt = 0;
    struct sockaddr_in serveraddr;
    struct hostent *server;
    char *hostname;
    long send_rate;
```

```
36
37      /* check command line arguments */
38      if (argc != 6)
39      {
40          fprintf(stderr, "usage: %s <hostname> <port> <BUFSIZE> <pkt
    count> <send rate in Kbps>\n", argv[0]);
41          exit(0);
42      }
43      hostname = argv[1];
44      portno = atoi(argv[2]);
45      pkt_count = atoi(argv[4]);
46      BUFSIZE = atoi(argv[3]);
47      send_rate = atoi(argv[5]);
48
49      send_rate *= pow(10, 3);
50
51      printf("send rate is : %ld\n", send_rate);
52
53      char buf[BUFSIZE];
54      /* socket: create the socket */
55      sockfd = socket(AF_INET, SOCK_DGRAM, 0);
56      if (sockfd < 0)
57          error("ERROR opening socket");
58
59      /* gethostbyname: get the server's DNS entry */
60      server = gethostbyname(hostname);
61      if (server == NULL)
62      {
63          fprintf(stderr, "ERROR, no such host as %s\n", hostname);
64          exit(0);
65      }
66
67      /* build the server's Internet address */
68      bzero((char *)&serveraddr, sizeof(serveraddr));
69      serveraddr.sin_family = AF_INET;
70      bcopy((char *)server->h_addr,
71            (char *)&serveraddr.sin_addr.s_addr, server->h_length);
72      serveraddr.sin_port = htons(portno);
73
74      struct timeval start, now, btw;
75      gettimeofday(&start, NULL);
76
77      memset(buf, 1, BUFSIZE);
78      for (int i = 0; i < pkt_count; i++)
79      {
80
81          /* send the message to the server */
82          serverlen = sizeof(serveraddr);
83          n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&
    serveraddr, serverlen);
84          if (n < 0)
```

```
85              error("ERROR in sendto");
86
87          //rate control
88
89          gettimeofday(&btw, NULL);
90
91          double duration_btw;
92          long double rate_btw;
93          duration_btw = (btw.tv_sec - start.tv_sec);
94          duration_btw *= 1000000;
95          duration_btw += btw.tv_usec - start.tv_usec;
96          duration_btw /= 1000000;
97          rate_btw = n * 8;
98          rate_btw = rate_btw / duration_btw;
99          rate_btw *= (i + 1);
100         if (rate_btw > send_rate / (i + 1))
101         {
102             double int_delay = n * 8;
103             int_delay = int_delay / send_rate;
104             int_delay *= (i + 1);
105             int_delay = int_delay - duration_btw;
106             if (int_delay > 0)
107             {
108                 usleep(int_delay * 1000000);
109             }
110         }
111
112         tot_pkt++;
113         tot_byt += n;
114     }
115
116     gettimeofday(&now, NULL);
117     unsigned long long rate;
118     double rate_now, duration_now, pkt_thr;
119     duration_now = (now.tv_sec - start.tv_sec);
120     duration_now *= 1000000;
121     duration_now += (now.tv_usec - start.tv_usec);
122     duration_now /= 1000000;
123
124     rate = tot_byt * 8;
125     printf("rate:%llu\tduration: %lf\t start: %ld\t now: %ld\n",
    rate, duration_now, start.tv_sec, now.tv_sec);
126
127     rate_now = rate / duration_now;
128
129     pkt_thr = tot_pkt / duration_now;
130     printf("[FINAL: ]sent %d packets, bytes: %llu, txrate: %lf Mbps,
     pkt throughput: %lf pps\n", tot_pkt, tot_byt, rate_now /
    1000000, pkt_thr);
131
132     return 0;
```

```
133 }
```

Listing B.1: udp client

B.2 udp client source code

```
/*
    Simple udp server
*/
#include <stdio.h>  //printf
#include <string.h> //memset
#include <stdlib.h> //exit(0);
#include <arpa/inet.h>
#include <sys/socket.h>
#include <signal.h>
#include <netdb.h>
#include <unistd.h>
#define BUFLEN 1024 * 8 //Max length of buffer
#define PORT 8000       //The port on which to listen for incoming
    data

static volatile int sigint = 0;
static int pkt_tot = 0;
double timenow, start_time, end_time;
double duration, pkt_thr = 0;

void die(char *s)
{
    perror(s);
    exit(1);
}

void sighandler(int num)
{
    sigint = 1;
    duration = end_time - start_time;
    pkt_thr = pkt_tot / duration;
    printf("received: %d packets\npkt_thr: %lf (pps)\nduration: %lf(
    s)", pkt_tot, pkt_thr, duration);
    exit(1);
}

int main(void)
{
    struct sockaddr_in si_me, si_other;

    int s, slen = sizeof(si_other), recv_len;
    unsigned char buf[BUFLEN];

    //create a UDP socket
    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
    {
        die("socket");
    }

```

```
48      // zero out the structure
49      memset((char *)&si_me, 0, sizeof(si_me));
50
51      si_me.sin_family = AF_INET;
52      si_me.sin_port = htons(PORT);
53      si_me.sin_addr.s_addr = htonl(INADDR_ANY);
54
55      //bind socket to port
56      if (bind(s, (struct sockaddr *)&si_me, sizeof(si_me)) == -1)
57      {
58          die("bind");
59      }
60
61      printf("Waiting for data...\nListening on port %d\n", PORT);
62
63      signal(SIGINT, sighandler);
64      //keep listening for data
65      while (1)
66      {
67          if ((recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr
        *)&si_other, (socklen_t *)&slen)) == -1)
68          {
69              die("recvfrom()");
70          }
71
72          struct timeval time;
73          gettimeofday(&time, NULL);
74          timenow = time.tv_sec * 1000000;
75          timenow += time.tv_usec;
76          timenow /= 1000000;
77
78          if (pkt_tot == 0)
79          {
80              start_time = timenow;
81          }
82
83          end_time = timenow;
84
85          pkt_tot++;
86      }
87
88      close(s);
89      return 0;
90  }
```

Listing B.2: udp server