T.C.

BAHÇEŞEHİR ÜNİVERSİTESİ

CONTAINERS MULTI-HOST NETWORKING PERFORMANCE INVESTIGATION ACCORDING TO TRAFFIC LOAD

Master's Thesis

GÜLSÜM ATICI

İSTANBUL, 2020

T.C.

BAHÇEŞEHİR UNIVERSITY

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING

CONTAINERS MULTI-HOST NETWORKING PERFORMANCE INVESTIGATION ACCORDING TO TRAFFIC LOAD

Master's Thesis

GÜLSÜM ATICI

Thesis Supervisor: ASSIST. PROF. DR. PINAR SARISARAY BÖLÜK

İSTANBUL, 2020

THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING

Name of the thesis: Containers Multi Host Networking Performance Investigation According to Traffic Load
Name/Last Name of the Student: Gülsüm ATICI
Date of the Defense of Thesis: 25.04.2020

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Assist. Prof. Dr. Yücel Batu SALMAN
Graduate School Director

I certify that this thesis meets all the requirements as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Tarkan AYDIN
Program Coordinator

This is to certify that we have read this thesis and we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Science.

Examining Comittee Members	Signatures
Thesis Supervisor Assist. Prof. Dr. Pınar SARISARAY BÖLÜK	----------
Member Assist. Prof. Dr. Tarkan AYDIN	----------
Member Assist. Prof. Dr. Yusuf YASLAN	----------

ACKNOWLEDGEMENTS

First, I would like to thank my thesis Supervisor, Assist. Prof. Dr. Pınar SARISARAY BÖLÜK, who have given me the opportunity to work on this thesis. I'm very thankful for their support, apprehension and precious help during the preparation of this thesis.

I would like to thank my thesis committee, consisting of Assist. Prof. Dr. Tarkan AYDIN and Assist. Prof. Dr. Yusuf YASLAN. Their feedbacks increased the quality of thesis study. I also would like to thank my lecturers who encouraged me during my Master program, and on my Master thesis.

My intimate thanks to my family for their endless support and love all through this work, also through my life. They provided immense moral support which strengthened me during my Master journey. Thanks to mother, father and my brother.

İstanbul, 2020 Gülsüm ATICI

ABSTRACT

CONTAINERS MULTI-HOST NETWORKING PERFORMANCE INVESTIGATION ACCORDING TO TRAFFIC LOAD

Gülsüm ATICI

Computer Engineering Master Program

Thesis Supervisor: Assist. Prof. Dr. Pınar SARISARAY BÖLÜK

April 2020, 110 pages

5G technology uses virtual network functions (VNF) in order to perform several operations based on NFV. Traditional VNFs (virtual machines) have some shortcomings such as consuming hardware resources excessively in high traffic situations and difficulty in operating dynamically pursuant to traffic load by causing inadequate user experience or even service interruptions. Cloud-native VNFs (containers) overcome the limitations of traditional VNFs by facilitating automated scaling according to dynamic traffic requirements, self-healing or fault tolerance together with automated capacity, fault and performance management by the help of container orhestrator platforms. Simplified management with reduction of unnecessary allocated resources, reusability together with sharing of processes by reducing power consumption and hardware resources are very beneficial especially for throttled resource situations. However, requirements of VNFs for telecommunications applications are different than any cloud native IT applications by performing data plane packet processing functions together with control, signalling and media processing with critical processing requirements such as very low latency, nearly zero downtime availability together with high traffic handle within thousand times larger throughput than ordinary IT applications. Consequently, CNFs in telecommunications should be resilient, low latency, ultra-high performance and robust under high traffic situations.

Previous researches put out that overlay networks give an overhead on network performance because of encapsulation, decapsulation or routing. Besides, container networking solutions can give different performances by using different implementation methods. Additionally, descrying the best performing container networking solution according to application type is critically important for specific VNF operations. Furthermore, traffic demand for VNF applications can fluctuate and boost depend on several situations. Consequently, multi host container networking performance for VNF applications may vary depend on container networking implementation methods, application types and traffic volume.

This study aims to discover high performing container networking solutions by considering workload, traffic type and traffic volume comprehensively. The behaviour of several container cluster networking solutions - Flannel, Weave, Libnetwork, Open

Virtual Networking for Open vSwitch and Calico - are explored with regard to the most commonly-used Container Network Functions in the form of MongoDB and Web access. Hence, this research is filling the gap of performance and reliability analysis on multi-host container networking under high traffic load within the mostly used NFV workloads such as database (MongoDB) and web application access which are not investigated before.

This thesis gives the contribution by evaluating the container networking solutions from two perpectives; single thread behaviour and behaviour according to traffic load. Initially, basic performance tests are performed within single thread to measure container networking solutions' regular behaviour by considering throughput, latency, bandwidth and reliability. Afterwards, heavy load situations has imitated with several parallel threads; performance and reliability has measured by using different workloads that are web access and MongoDB operations under different traffic loads in terms of throughput, latency, bandwidth, response times, jitter, retransmitted TCP segments and lost datagrams. Practically, tested workloads are selected because of their common usage in VNF's and container networking solutions conformity according to workloads are clearly represented.

In the evalutions with single thread, baremetal hosts native performance is also compared with container networking solutions in terms of throughput and latency. Baremetal hosts outperformed than all container networking solutions as container networking solutions have packet routing or encapsulation overheads. Suprisingly, Calico's throughput within 4096 bytes message sizes passed the bare-metal host throughput in TCP_CRR tests which can be explained with the hardware offloading features effect of Docker. This study is also supporting the offloading amendment in container networking solutions. In single thread evalutions, OVN does not provide satisfied results for throughput, bandwidth and latency for TCP traffic together with low bandwidth in UDP traffic, which can be explained by lack of Geneve offloading, bigger header size, first hit misses because of OVS long datapath. Weave is the second worst performing solution after OVN with fair and low bandwidth respectively in small and large window sizes together with having low reliability on all window sizes. Hereby, Weave could not operate adequately compared with other tested solutions in single thread operations. Libnetwork single thread performance is unsatisfactory compared with Calico as it has some drawbacks similar to other overlay solutions. Flannel is not opportune in single thread performance evaluations as giving low bandwidth and high latency outputs. Calico is best performing solution by considering all single thread test operations. This result is reasonable as Calico is not an overlay solution. Calico which uses BGP and pure L3 routing, is not impacted with any encapsulation overheads.

In multithreaded evalutions, OVN gives comparatively low multithreading gain for all type of workload operations. Weave offers the high multithreading gain for all database operations however it render quite low gain for web access operations. Libnetwork gives the highest multithreading gain for all workloads except database update operations. However, Libnetwork's performance has most deteriorated under heavy traffic load amongst all container networking solutions. Flannel gives the low multithreading gain for all database operations while offering average multithreading gain for web access operations. Flannel is most unreliable solution in terms of jitter, lost UDP datagrams and retransmitted TCP segments. Calico gives low

multithreading gain for all type of workload operations. Calico could not benefitted from multithreading as much as other solutions such as Libnetwork and Weave. However, it is performance ranking is not much impacted under heavy traffic load.

Multithreaded evaluations show that none of the solutions provide high throughput for all type of workloads under optimum or heavy load situations. Hence, this research presents the view that traditional container networking implementation methods may not fulfill the Container Network Functions' networking performance requirements. This is because container networking performance changes dynamically, depending on traffic load and application types. To overcome this problem, a new smart container networking architecture is proposed which uses the Smart Container Network Interface Manager in conjunction with container monitoring tools. The proposed architecture allows containers to use several container networking solutions dynamically, depending on the desired performance requirements.

Keywords: Container Cluster Networking, Performance, Reliability, Multithreading, Heavy Traffic Load.

ÖZET

TRAFİK YÜKÜNE GÖRE KONTEYNER ÇOKLU HOST AĞ ÇÖZÜMÜ PERFORMANS İNCELEMESİ

Gülsüm ATICI

Bilgisayar Mühendisliği Yüksek Lisans Programı

Tez Danışmanı: Dr. Öğr. Üyesi Pınar SARISARAY BÖLÜK

Nisan 2020, 110 sayfa

5G teknolojisi, NFV'ye dayalı çeşitli işlemleri gerçekleştirebilmek için sanal ağ işlevlerini (VNF) kullanır. Geleneksel VNF'ler (sanal makineler), trafik yüküne bağlı olarak dinamik çalışma zorluğu gibi yetersizliklere sahiptir. Bu sebeple yüksek trafik durumlarında donanım kaynaklarını aşırı tüketerek yetersiz kullanıcı deneyimi ve hatta hizmet kesintisine neden olabilirler. Konteyner yönetim platfomlarının otomatik kapasite, hata ve performans yönetim işlevleri de kullanılarak, konteyner olarak çalışan sanal ağ işlevleri (CNF), dinamik trafik gereksinimlerine uygun şekilde kolaylıkla iyileştirilip, ölçeklendirilebilir ve geleneksel VNF'lerin karşılaştığı sorunların üstesinden gelinebilir. CNF kullanımı ile donanım kaynakları paylaşımlı olarak kullanılır, gereksiz tahsis edilmiş kaynakların ve güç tüketiminin azaltılması sağlanır; konteynerların basitleştirilmiş yönetim kabiliyetleri ile kısıtlı kaynak durumlarının üstesinden kolaylıkla gelinir. Bununla birlikte, telekomünikasyon uygulamalarındaki sanal ağ işlevlerinin isterleri sıradan IT uygulamalarından farklıdır. Telekomünikasyon uygulamaları veri düzleminde paket işleme işlevlerinin yanı sıra çok düşük gecikme, neredeyse sıfır kesinti süresi, yüksek trafik kapasitesi ve sıradan IT uygulamalarından 1000 kat fazla veri akış hızı gibi oldukça zorlayıcı isterlere sahiptir. Sonuç olarak, telekomünikasyon uygulamalarında kullanılan CNF'ler trafik yüküne göre esnek, düşük gecikmeli, çok yüksek performanslı ve yüksek trafik yükü altında dirençli olmalıdır.

Önceki araştırmalar, üst ağların kapsülleme, dekapülasyon veya yönlendirme nedeniyle ağ performansında kötüleşmeler meydana getirdiğini ortaya koymuştur. Bununla birlikte, konteyner ağ çözümleri farklı uygulama yöntemleri kullanarak farklı performanslar verebilmektedir. Bu sebeple, uygulama türüne göre en iyi performansa sahip konteyner ağ oluşturma çözümünü seçmek belirli VNF işlemleri için kritik öneme sahiptir. Çünkü, VNF uygulamaları için trafik talebi dalgalanabilir ve çeşitli durumlara bağlı olarak artabilir. Sonuç olarak, VNF uygulamaları için konteyner küme ağ çözümlerinin performansı; ağ oluşturma yöntemlerine, uygulama türlerine ve trafik hacmine bağlı olarak değişebilir.

Bu çalışma, iş yükünü, trafik türünü ve trafik hacmini kapsamlı bir şekilde dikkate alarak yüksek performanslı konteyner ağı çözümünü keşfetmeyi amaçlamaktadır. Çok sayıda

konteyner küme ağ çözümünün davranışı - Flannel, Weave, Libnetwork, Open vSwitch için açık sanal ağ ve Calico - en yaygın kullanılan konteyner ağı işlevleri açısından MongoDB ve Web erişimi biçiminde araştırılmaktadır. Bu nedenle, bu çalışma, daha önce gerçekleştirilmemiş olan konteyner küme ağ çözümleri üzerinde veritabanı (MongoDB) ve web uygulaması erişimi gibi en çok kullanılan NFV iş yükleri için yüksek trafik yükü altında performans ve güvenilirlik analizi boşluğunu doldurmaktadır.

Bu tez, konteyner ağ oluşturma çözümlerini iki açıdan değerlendirerek katkı sağlar; tekli iş parçacığı davranışları ve çoklu iş parçacığı kullanılarak trafik yüküne göre davranışlar. Başlangıçta, kapasite, gecikme, bant genişliği ve güvenilirlik göz önünde bulundurularak üst ağ oluşturma çözümlerinin sıradan davranışlarını ölçmek için temel testler tek bir iş parçacığı ile yapılır. Daha sonra, ağır yük durumları birkaç paralel iş parçacığı ile taklit edilmiştir. Performans ve güvenilirlik; ağ debisi, gecikme, bant genişliği, yanıt süreleri, Jitter, yeniden iletilen TCP segmentleri ve kayıp datagramlar açısından farklı trafik yükleri altında web erişimi ve MongoDB işlemleri gibi farklı iş yükleri kullanılarak ölçülmüştür. Pratik olarak, test edilmiş iş yükleri, VNF'lerdeki ortak kullanımlarından dolayı seçilir ve konteyner ağ oluşturma çözümlerinin iş yüklerine göre uygunluğu açıkça ortaya konulur.
Tek iş parçacıklı değerlendirmelerde, bare-metal sunucuların performansı, verim ve gecikme açısından konteyner ağ oluşturma çözümleriyle de karşılaştırılır. Konteyner ağ çözümleri paket yönlendirme veya kapsülleme ek yüklerine sahip olduğundan bare-metal ana bilgisayarları tüm konteyner ağ çözümlerinden daha iyi performans gösterir. Şaşırtıcı bir şekilde, 4096 bayt mesaj boyutları için Calico'nun iş hacminin TCP_CRR testlerinde bare-metal sunucu iş hacmini geçmesi Docker'ın donanım boşaltma özellikleri etkisi ile açıklanabilmektedir. Bu çalışma ayrıca konteyner ağ çözümlerindeki donanım boşaltma iyileştirmesini de desteklemektedir. Tek iş parçacıklı değerlendirmelerde, OVN, UDP trafiğinde düşük bant genişliği ile birlikte TCP trafiği için verim, bant genişliği ve gecikme için tatmin edici sonuçlar vermez. Bu durum OVN'nin Geneve donanım boşaltma özelliğini kullanamaması, daha büyük üst ağ başlık boyutu, OVS uzun veri yolu ve ilk akış oluşturmada yaşanan kayıplar ile açıklanabilir. Weave, tüm pencere boyutlarında düşük güvenilirliğe sahip küçük ve büyük pencere boyutlarında sırasıyla ortalama ve düşük bant genişliğine sahip OVN'den sonra en kötü performans gösteren ikinci çözümdür. Bu vesile ile, Weave tek iş yükü ile gerçekleştirilen testlerde diğer çözümlerle karşılaştırıldığında yeterli performansı gösterememiştir. Libnetwork diğer üst ağ çözümlerine benzer bazı dezavantajlara sahip olduğu için, Libnetwork'ün tek iş parçacığı testlerinde performansı, Calico ile karşılaştırıldığında tatmin edici değildir. Flannel, düşük bant genişliği ve yüksek gecikme çıktıları verdiği için tek iş parçacığı performans değerlendirmelerinde verimli bulunmamıştır. Calico, tüm tek iş parçacığı operasyonlarını göz önünde bulundurulduğunda, en iyi performansı gösteren çözümdür. Calico bir üst ağ çözümü olmadığından bu sonuç makuldür. BGP ve L3 yönlendirme kullanan Calico, herhangi bir kapsülleme ek yükünden etkilenmez.

Çok iş parçacıklı değerlendirmelerde, OVN tüm iş yükü işlemleri için nispeten düşük çoklu iş parçacığı kazancı sağlar. Weave, tüm veritabanı işlemleri için yüksek çoklu iş parçacığı kazancı sunar, ancak web erişimi işlemleri için oldukça düşük kazanç sağlar. Libnetwork, veritabanı güncelleme işlemleri hariç tüm iş yükleri için en yüksek çoklu iş parçacığı kazancını sağlar. Bununla birlikte, Libnetwork'ün performansı, yoğun trafik yükü altında tüm konteyner ağı çözümleri arasında en çok kötüleşendir. Flannel, tüm

veritabanı işlemleri için düşük çoklu iş parçacığı kazancı sağlarken, web erişimi işlemleri için ortalama çoklu iş parçacığı kazancı sunar. Flanel, Jitter, kayıp UDP datagramları ve yeniden iletilen TCP segmentleri açısından en az güvenilir çözümdür. Calico, tüm iş yükü işlemleri için düşük çoklu iş parçacığı kazancı sağlar. Calico, çoklu iş parçağı kullanımından Libnetwork ve Weave çözümleri kadar faydalanamamıştır. Ancak, Calico'nun performansı trafik yükünden de pek etkilenmemiştir.

Çok iş parçacıklı değerlendirmeler, çözümlerin hiçbirinin optimum veya ağır yük durumlarında tüm iş yükü türleri için yüksek bir verim sağlamadığını göstermektedir. Bu nedenle, bu araştırma, geleneksel konteyner ağı uygulama yöntemlerinin, konteyner ağ işlevlerinin ağ performansı gereksinimlerini karşılayamayabileceği görüşünü sunmaktadır. Bunun nedeni, konteyner ağ performansının trafik yüküne ve uygulama türüne bağlı olarak dinamik bir şekilde değişmesidir. Bu sorunun üstesinden gelebilmek için, Smart Container Network Interface Manager'ı konteyner izleme araçlarıyla birlikte kullanan yeni bir akıllı konteyner ağı mimarisi önerilmektedir. Önerilen mimari, konteyner'ların istenen performans gereksinimlerine bağlı olarak birçok ağ çözümünü dinamik olarak kullanmasına olanak sağlamaktadır.

Anahtar Kelimeler: Konteyner Küme Ağ Çözümü, Performans, Güvenilirlik, Çoklu İş Yükü, Yoğun Trafik Yükü.

CONTENTS

TABLES

FIGURES

ABBREVIATIONS

5G	:	Five Generation
ADSL2	:	Asymmetric Digital Subscriber Line
API	:	Application Programming Interface
ARP	:	Address Resolution Protocol
AUFS	:	Advanced Multi-layered Unification Filesystem
AWS	:	Amazon Web services
BGP	:	Border Gateway Protocol
BIRD	:	Bird Internet Routing Daemon
BSD	:	Berkeley Software Distribution
CLI	:	Command Line Interface
CNF	:	Container Network Function
CNCF	:	Cloud Native Computing Foundation
CNI	:	Container Network Interface
CNM	:	Container Network Model
CPU	:	Central Processing Unit
CoAP	:	Constrained Application Protocol
CRDT	:	Conflict-free Replicated Data Type
CRI	:	Container Runtime Interface
CRR	:	Connect Request Response
DNS	:	Domain Name System
DPDK	:	Data plane Development Kit
ECS	:	Amazon EC2 Container Service
EKS	:	Elastic Kubernetes Service
ENB	:	Enodeb
EPC	:	Enhanced Packet Core

FTP	:	File transfer protocol
GENEVE	:	Generic Network Virtualization Encapsulation
GKE	:	Google Container Engine
GRE	:	Generic Routing Encapsulation
GRO	:	Generic Receive Offload
GSO	:	Generic Segmentation Offload
HPC	:	High Performance Computing
HSS	:	Home Subscriber Service
HTTP	:	Hypertext Transfer Protocol
I/O	:	Input Output
IPC	:	Inter-process Communication
IT	:	Information Technology
JDBS	:	Java Database Connectivity
JSON	:	Javascript Object Notation
KVM	:	Kernel Based Virtual Machine
KVS	:	Key Value Store
LDAP	:	Lightweight Directory Access Protocol
LXC	:	Linux containers
MAC	:	Media Access Control Address
MTU	:	Maximum Transmission Unit
NAS	:	Network Attached Storage
NAT	:	Network Address Translation
NFV	:	Network function virtualization
NIC	:	Network Interface Card
OCI	:	Open Container Initiative
OVN	:	Open Virtual Network

OVS : Open Virtual Switch

OVSDB : Open vSwitch Database

QEMU : Quick Emulator

QoS : Quality of Service

RDMA : Remote Direct Memory Access

REST : API RESTful Application Programming Interface

ROCE : RDMA Over Converged Ethernet

RR : Request Response

RTT : Round Trip Time

SCTP : Stream Control Transmission Protocol

SMTP : Simple Mail Transfer Protocol

STT : Stateless Transport Tunneling

TCP : Tranmission Control Protocol

TCP_CRR : TCP Connect Request Response

TCP_RR : TCP Request Response

TPM : Trusted Platform Modules

TSO : TCP Segmentation Offload

UDP : User Datagram Protocol

UDP_RR : UDP Request Response

UI : User Interface

VDSL2 : Very High Speed Digital Subscriber Line

VLAN : Virtual LAN

VM : Virtual Machine

VNF : Virtual Network Function

VTEP : VXLAN Tunnel Endpoint

VXLAN : Virtual Extensible Lan

XML : Extensible Markup Language

SYMBOLS

Mean value	:	μ
Standard deviation	:	σ

1. INTRODUCTION

NFV has the great importance in development of 5G networks. NFV is going to be mature by overcoming the automated deployment, VNF onboarding, scaling, configuration and updating [48]. In traditional method, VNFs are the network functions which are hosted in the virtual machines which contain necessary software to run the applications. Traditional VNFs (virtual machines) have some deficiencies such as consuming hardware resources excessively in high traffic situations and difficulty in operating dynamically pursuant to traffic load by causing inadequate user experience or even service interruption [50].

Building cloud-native VNFs (containers) which are called CNFs overcome above discussed limitations of traditional VNFs [52]. CNFs have API's which facilitate automated scaling according to dynamic traffic requirements, self-healing or fault tolerance together with automated capacity, fault and performance management by the help of container orhestrator platforms such as Docker Swarm, Kubernetes, Amazon EKS etc [48]. Simplified management with reduction of unnecessary allocated resources, reusability together with sharing of processes by reducing power consumption and hardware resources are very beneficial especially for heavy loaded traffic situations [50]. Those are all crucial benefits of Cloud native deployments in order to operate in throttled situations.

Although providing many benefits, there are certain apprehensions of using container technology in NFV especially for network domain. Requirements of VNFs for Telecom applications are different than any cloud native IT application. Telecom VNF applications are created for data plane packet processing functions together with control, signalling and media processing with critical processing requirements such as less than 1 millisecond end-to-end over-the-air latency, availability and coverage rises reaching 100 percent like nearly zero downtime and 99.99 percent severally, high traffic handle within 1000 times larger throughput than ordinary IT applications [51]. The reason why any error or insufficiency will impact number of subscribers using the network, containers networking performance become the biggest challenge to fullfill the 5G high

performance targeted applications within very low latency, agility, enabling real time use cases like IoT, Augmented Reality or Machine to Machine Communication.

Considering 5G NFV implementation requirements for remarkable agility in every level of deployment processes and operational requirements especially for high traffic, CNFs seems to be the only solution. Dynamically managed, quickly deployed and scaled according to traffic load are significant containerization benefits [49]. Consequently, CNFs in telecommunications should be resilient, low latency, ultra-high performance and robust under high traffic situations [49].

This study is aiming to propose the high performing container networking solution according to traffic load by exploring the behaviour of several multi-host container networking solutions under excessive traffic situations over mostly deployed CNF's application types. Containers multi-host networking solutions such as Flannel [53], Weave [27], Libnetwork [21], OVN [32] and Calico [54] are deployed as seperate scenarios on top of baremetal hosts; basic demeanor and high traffic situations' attitudes are evaluated from the aspect of performance and reliability. Container networking solutions behaviours under high traffic situations are simulated by multithreading in order to imitate real world cases.

1.1 PROBLEM STATEMENT

Telecommunication NFV applications need to serve publicly to plenty of subscribers simultaneously. Demands of traffic volume to Telecom VNF applications can fluctuate depend on several situations such as social events, disaster situations, certain hours of day, certain days of year. However, low-latency and ultra-high performance networks are required in every traffic volume. Investigating performance and reliability of multi-host container networking solutions according to traffic load for different applications and proposing the high-performing networking solution is the aim of this study.

Previous researches put out that overlay networks give an overhead on network performance becase of encapsulation, decapsulation or routing. However, all container overlay networks caused unexpectedly performance loss. For example, Docker native

overlay inflicted 82.8 percent throughput drop and 55 percent latency increase compared to the baseline [1]. Even, container multi host networking performance has impacted with several overlay methods, container networking solutions can give different performances by using different implementation methods. Additionally, descrying the best performant container networking solution according to application type is critically important for Telecommunications. Furthermore, traffic demand for VNF applications can fluctuate and boost depend on several situations.

Consequently, multi host container networking performance for VNF applications may vary depend on implementation methods, application types and traffic volume. This study is aiming to compare the network performance and relibility of different container multi-host networking solutions (Libnetwork, Flannel, Weave, OVN, Calico) according to workload, traffic type and traffic volume comprehensively, then propose the high-performant networking solution within smart container networking architecture. This research is filling the gap of multi-host container networking performance and reliability analysis under high traffic load within database (MongoDB) and web application which is not investigated before. Peformance under load has measured in terms of throughput, bandwidth, latency and response times; while reliability has evaluated over jitter, retrasmitted TCP segments and percentage of lost datagrams. Additionally, this study is such compherensive that it's the first time of comparing several popular container networking solutions as well as OVN under high traffic situations.

1.2 RESEARCH OBJECTIVES

The main objectives of the thesis are investigating the docker multi-host container networking performance for different traffic loads, workloads and data transfer types by selecting docker as a CRI because of it's popularity and common usage. Docker multi-host networking methods are implemented without using any container orhestrator solutions and containers are directly kept in bare-metal hosts to prevent virtualization overheads which are caused by hypervisors.

Multithreading or multi-streaming executions are performed in order to increase the traffic load to simulate different traffic situations. Networking performance for bulk,

request-response, connect-request-response data transfer types together with database and web application traffics are evaluated to imitate the real World cases.
Under low and high workloads, HTTP requests throughput and response times to an external web application and MongoDB database operations are appraised to observe the workload effects.

1.3 RESEARCH QUESTIONS

This research tries to brighten the performance and reliability effects of following items for container multi-host networking solutions:

a. Different implementation methods used in networking solutions
b. Amount of traffic
c. Different network protocols, data transfer types, message sizes, TCP window sizes
d. Different workloads which are commonly used as VNF application such as MongoDB and Web application

1.4 CONTRIBUTIONS

This thesis focuses on container networking performance for different traffic loads, workloads and data transfer types. It aims to discover high-performant container networking solution for mostly used VNF applications under high traffic loads. By evaluating all the experiments, that none of the solutions provided relatively high throughput for all types of workload under optimum or heavy load conditions. As it may not possible to apply the suggested networking solutions adaptively with traditional container networking implementation methods, a new smart networking implementation design is proposed. Proposed solution solves the traditional container networking implementation methods' inadequacy about meeting 5G NFV implementation various operational requirements. The main contributions of the thesis are listed as follows:

a. Contribution 1: comparing hosts native performance with container networking solutions

Within single thread evaluations, baremetal hosts native performance is compared with container networking solutions in terms of throughput and latency. Throughput and latency comparison of baremetal hosts, Libnetwork, Weave, Flannel, OVN and Calico are performed according to message sizes with TCP_STREAM, UDP_STREAM and TCP_CRR data transfer methods. It's observed that normally message size increment improved throghput in all data transfer methods. Baremetal hosts could give sufficient throughput which is 974 Mbits/sec that is quite close to baseline value (1 Gbit/sec) at bulk TCP transfers. It's observed that bulk TCP transfer throughputs are always higher than bulk UDP transfer throughputs which are at the same message sizes. This is because of TCP whole bandwidth utilization irreleavant from the message sizes by using Nagle's algorithm. Baremetal hosts outperformed than all container networking solutions as all container networking solutions have packet routing or encapsulation overheads.

It is discovered that all offloading features are become enabled on the docker interface docker0 by default and they can be used if they are supported by the hardware. Except the GRO, all other offloading features are disabled on hosts during these experiments. Although hardware offloading is not enabled in pysical interfaces, it improved the container networking performance. Suprisingly, Calico's throughput within 4096 bytes message sizes passed the baremetal host throughput in TCP_CRR tests. Similar situations are also observed in previous studies [3], [5] and explained with the hardware offloading features effect of Docker. This study is also supporting the offloading impact in container networking solutions. Besides, hardware offloading features improved the TCP performance more than UDP performance. This work appears in the following paper.

i. Atici, G. and Boluk, P. (2020) "A Performance Analysis of Container Cluster Networking Alternatives," *The 2nd International Conference On Industrial Control Network And System Engineering Research, ICNSER2020*, pp. 1–8. (Accepted)

b. Contribution 2: comparing container networking solutions ordinary behaviour

Within single thread evaluations, OVN has the lowest bandwidth in TCP and UDP traffic, it also has the highest latency in individual TCP tests. Hereby, OVN does not provide satisfied results for throughput, bandwidth and latency for TCP traffic, furthermore it has low bandwidth in UDP traffic. Nevertheless, OVN uses kernel datapath which accelerates the packets which suit to the flows, so performance improvement is usually expected for large message sizes but this expection could not be observed clearly.

There may be several reasons that procreate gathared results. OVN uses Geneve encapsulation which has bigger encapsulation header (58 bytes) than VXLAN encapsulation header (50 bytes). Physical network interfaces of hosts does not support Geneve offloading although they support VXLAN offloading features called tx-udp_tnl-segmentation and tx-udp_tnl-csum-segmentation. If tx checksum offload is enabled and the adapter has the capability; VXLAN offloading is also enabled. However, OVN is not benefitted from those features while Flannel, Weave and Libnetwork could utilize them. In addition, OVN has long data path wandering user space and kernel space for initial packets of transfers which causes first hit misses till the new flow is created for unspecified traffic flows.

In single thread experiments, Weave is the second worst performing solution after OVN, with fair and low bandwidth respectively in small and large WS (Window Size) together with having low reliability on all window sizes. Weave gives low bandwidth both in TCP and UDP, although it gives relatively low latency in individual TCP transfers. Herewith, Weave could not operate adequately compared with other tested solutions in single thread operations. Weave has disadvantage of using smaller MTU sizes (1376 bytes) compared with other solutions which causes more segmentation for large message sizes. Weave uses VXLAN as overlay solution which brings the overhead of VXLAN encapsulation. Howbeit, the encapsulation overhead is waned with VXLAN offloading.

Calico is the only solution which provides high throughput for TCP/UDP bulk data transfers alongside highest throghput for HTTP like connect-request-response traffic

within small message sizes. Calico yields superior performance which gives low and lowest latency results for both small and large message sizes respectively together with highest bandwidth both in small and large window sizes. Calico accomplished the tests with highest bandwidth and lowest latency both in TCP and UDP. Consequently, Calico is best performing solution by considering all single thread test operations. This result is reasonable as Calico is not an overlay solution. Calico which uses BGP and pure L3 routing, is not impacted with any encapsulation overheads. Nowadays, BGP suppport is applicable in several cloud platforms such as GCP, AWS, Azure by supporting VPG which is new virtual gateways. Those platforms allows customers to configure Private Autonomous System Number (ASN), then set the ASN on the cloud side of the BGP session for VPNs [56]. Although Calico's has limited support of Layer 4 protocols, it enhance it's support day by day. Calico supports TCP, UDP, ICMP, UDPlite for a while; then it starts to support SCTP protocol within version 3.3 [55]. Nevertheless, it has one drawback that having big and complex routing tables which complicate it's running on large environments.

In single thread experiments, Libnetwork has maximum throughputs both in small and large message sizes for bulk UDP data transfers. However, Libnetwork gives low throughput for TCP_CRR and fair thoughput for bulk TCP transfers regardless from message sizes. From the latency point of view, Libnetwork gives comparatively low latency in TCP and fair latency in UDP in individual tests. Hereby, Libnetwork single thread performance is unsatisfactory compared with Calico as it has some drawbacks similar to other overlay solutions. It is supposed that Libnetwork's performance is impacted by using VXLAN overlay. Additionally, it uses Linux bridges instead of OVS bridges and does not use fast datapath like OVN and Weave.

Within single thread performance evaluations, Flannel bulk TCP data throughput enhances with large message sizes; meantime it provides fair throughput for bulk UDP transfers and low throughput for TCP_CRR independent from message sizes. It also gives comparatively fair bandwidth in TCP and low bandwidth in UDP by giving low latency in individual TCP transfers and high latency in individual UDP transfers. Consequently, Flannel is not opportune in single thread performance

evaluations as giving low bandwidth and high latency outputs. It is supposed that performance has declined with VXLAN overlay drawbacks similar to Libnetwork and Weave. This contribution appears in the following papers:

i. Atici, G. and Boluk, P. (2020) "A Performance Analysis of Container Cluster Networking Alternatives," *The 2nd International Conference On Industrial Control Network And System Engineering Research, ICNSER2020*, pp. 1–8. (Accepted)

ii. Atici, G. and Boluk, P. (2020b) "TCP Pencere Boyutuna Göre Konteyner Küme Ağ Çözümlerinin Performans Analizi Container Cluster Networking Performance Analysis According to TCP Window Size," *IEEE Conference on Signal Processing and Communications Applications 2020*, pp. 1–4. (Accepted)

c. Contribution 3: Evaluations based on traffic load with different workloads

In multithreaded evaluations, OVN performance improvement is normally expected with multithreading because of OVS kernel datapath. However, OVN has low throughput at all database operations meantime it has fair throughput at web access operations. OVN gives low multithreading gain for all type of workload operations. Under heavy traffic load, OVN yields fair throughput for web access operations, meanwhile it assures low throughput for insert, query and update operations. It's ranking under heavy load remained same compared with performance under optimum number of threads. Under heavy traffic load, OVN assures high TCP bandwidth, however it yields high response times. Besides, OVN is unreliable compared with other tested solutions under heavy traffic load. OVN has several disadvantages which cause poor performance within multithreading such as lack of VXLAN offloading, bigger header size, first hit misses because of OVS long datapath. Furthermore, OVN includes several processes such as ovsdb-server, ovs-vswitched together with ovn-northd and ovn-controller processes which may cause higher cpu utilization under high traffic loads.

Weave provides highest throughput for database update operations and high throughput for database query operations, although it provides low throughput at database insert and web access operations with multithreading. Weave is the only solution which

provides highest thoughput with multithreading at update operations with significant difference than other tested solutions. Weave offers the high multithreading gain for all database operations however it render quite low gain for web access operations. Weave's relative performance under high traffic load is improved by taking better rank for MongoDB query operations, while it's rank is not changed for remaining operations. Weave gives the lowest web access response times under heavy traffic. Additionally, it is the most reliable amongst all tested solutions under heavy traffic load. The results are put forth that multithreading has improved the Weave performance. Actually, this situation is supposed to be the influence of Weave special datapath. Weave utilizes OVS bridges which activate fast datapath in kernel level and route the packets similar to OVN. But Weave does not have disadvantage of first hit misses like OVN has, as route is already defined for the packets.

Calico offers high throughput for insert operations, although it yields low throughput at query, web access and update operations within multithreading. Calico gives low multithreading gain for all type of workload operations. Under high traffic load, Calico provides relatively low throughput for query and update operations, while giving high throughput for insert and web access operations. It's heavy load behaviour is not affected except web access operations. Hereby, it's relative performance amongst all container networking solutions ameliorated by taking better rank in throughput of web operations under heavy traffic load. Calico could not benefitted from multithreading as much as other solutions such as Libnetwork and Weave. However, it is ranking is not much impacted under heavy traffic load.

In traffic-load-based tests with multithreading, Libnetwork gives the highest throughput for MongodB insert and query operations, while giving the low throughput for update and web accces operations. Libnetwork gives the highest multithreading gain for all workloads except update operations. Libnetwork caters high throughput for insert and query operations, however gives relatively low throughput for update and web request operations. It is observed that Libnetwork relative performance has deteriorated under high traffic load as it yields the lower performances for MongoDB query and update operations. Libnetwork assures lowest TCP bandwidth, together with highest response

times and fair reliability under heavy traffic load. Consequently, Libnetwork is the most beneffited solution from multithreading within optimum number of threads having the highest multithreading gain compared with single thread executions. However, Libnetwork's performance has most deteriorated under heavy traffic load amongst all container networking solutions.

Flannel delivers the lowest thoughput for all database operations while it is providing the highest throughput at web access operations under optimum number of threads. Flannel gives the low multithreading gain for all database operations while offering average multithreading gain for web access operations. Under heavy traffic load, Flannel gives high throughput only for web request operations and provides relatively low throughput for all database operations. Flannel's rank is escalated for MongoDB update operations while it displays the similar ranks for other operations under heavy traffic load. Flannel assures fair TCP bandwidth, while giving the lowest web access response times under heavy traffic. Flannel is the most unreliable solution in terms of jitter, lost UDP datagrams and retransmitted TCP segments. Consequently, Flannel is generally good at HTTP like traffic such as web operations, request/response transfers distinctively with multithreading even under heavy traffic load. However, it is the most inefficacious for all database operations, together with being most unreliable solution.

It is discovered from the experiments that solutions are prone to specific type of workload operations. However, there is not enough entry to explain solutions' different behaviours within specific type of operations. This situation probably related with the container solutions adequacy on specific type of resource utilization. It is also observed that container networking solutions behaviour and success ranking in order to a specific type of operations generally unchanged within different peformance indicators. Flannel best performs at web operations while Weave best perfoms on database operations under heavy traffic load. Increament in the load after the optimum load point just worsened the performance for all container networking solutions. It is observed that Libnetwork performance has deteriorated while Calico, Flannel and Weave's relative performance has improved for some specific type of

operations under heavy traffic load. Additionally, OVN heavy load performance ranking remained same compared with optimum load point.

Consequently, It is observed that none of the solutions provided high throughput for all type of workloads at their maximum throughput or at heavy load situations. As it is impossible to apply the suggested networking solutions adaptively with traditional container networking implementation methods, a new smart networking implementation design is proposed by means of SIM. SIM decides the best solution by allowing containers to use several networking solutions dynamically, depending on requirements in cooperation with container monitoring tools. This study appears in the following paper:

i. Atici, G. and Boluk, P. (2020a) "1 2 A New Smart Networking Architecture for Container Network Functions," *Turk J Elec Eng & Comp Sci*, pp. 1–15. (Submitted)

1.5 OUTLINE OF THESIS

This thesis is organized orderly into 6 chapters as follows:

Chapter 1 gives the importance of containers usage in NFV environment because of their several benefits such ease of scalability, fault-tolerance, agility, cost efficiency. Container network performance evalution for different traffic situations and workloads in order to simulate the real world scenarios is the main intention. Problem statement, research objectives, research questions, contributions and outline of thesis are displayed in this chapter.

Chapter 2 presents the previous research studies regarding with the container network performance analysis by emphasizing of this research which fills the gap of reliability and performance analysis under load within different applications and data transfer types.

Chapter 3 explains the terms of Virtualization, Container, Docker and Docker networking. Docker multi-host networking solutions such as Libnetwork with Etcd, Weave, Flannel, OVN with Open vswitch and Calico are presented.

Chapter 4 describes the research methodology (Quantitative), test environment and test scenarios, deployment considerations together with measurement tools, performance metrics, experimentations and analysis techniques.

Chapter 5 put forwards the results of experimentations according to different traffic situations by considering the metrics such as throughput, bandwidth, latency, retransmitted TCP segments, lost datagrams, jitter and response times. The figures of results and discussions are inluded in this chapter. This sections also discusses about the reasons, summarizes the results and put out the proposed new smart networking architecture.

Chapter 6 gave the final outcomes of the research consequently. Future research suggestions are also proposed in this chapter.

2. RELATED WORK

Many researchers investigated on container cluster networking performance in different environments, applications and performance indicators because of growing demand to containerized platforms. Normally, containers multi host networking studies which are focused on fundamental performance evaluations can be splitted into two parts according to components of investigation environments; utilizing container orhestrators, without container orhestrators. Regarding with the performance investigation under high traffic load, available studies were generally concentrated on HPC platforms by judging the conformity of containers in HPC area.

2.1 FORMER STUDIES WITHOUT CONTAINER ORHESTRATOR

The cluster networking solutions without container orhestrator platforms commonly use Docker as container runtime interface. Zeng et al. has measured and evaluated the performance of three mainstream network solutions (Flannel, Swarm Overlay, Calico) [2]. Their experiments showed that Calico has the highest performance and its TCP throughput is almost same with host, but the configuration is the most complicated; Flannel configuration is simple, but it is designed for Kubernetes and the performance is not adequate; Swarm overlay configuration is simple with low performance [2].

Suo et al. compared the performance of different types of applications directly running in VM and running Docker containers on top of VM with Swarm, Flannel, Calico and Weave overlays [1]. They measured the CPU, memory, end-to-end TCP, UDP throughput and showed that containerization does not degrade the performance of compute-intensive and memory-intensive workloads in virtualized environments [1]. However, he presented that network throughput drops dramatically when applications communicate through the container network compared to communicating directly via the VM network [1]. Similar to [2], [1] presented that although Calico (BGP mode) has the best performance among to other overlays, it exposed considerable performance degradation [1]. [1] put out significant performace loss for Docker native overlay which thrilled 82.8 percent throughput drop and 55 percent latency increase compared to the baseline. In addition, performance degradation varies depending on the type of network protocol

and packet size such as container networks has larger overhead to TCP than to UDP workloads [1].

Bankston and Guo also explored the network performance of Calico, Flannel, Weave and Libnetwork [3]. Unlike [1] and [2] , they implemented real world deployment scenarios on top of virtual machines over popular public clouds (AWS, Microsoft Azure, Google Cloud Platform) [3]. They reported that adjusting MTU provides the best throughput and encryption had a large negative impact both on throughput and CPU [3]. They encountered the inequality between UDP and TCP traffic and could not find an explanation for this situation as both protocols are encapsulated with VXLAN headers [3].

Hermans et al. analyzed performance of various Docker overlay solutions such as native Docker overlay driver, third party solutions Flannel and Weave. They deployed these solutions in a high latency and geographically distributed environments by executing the point-to-point and streaming media application benchmarks [4]. They observed small effects of latency and jitter however UDP throughput has significantly dropped for all overlay solutions [4]. They indicated that not similar to UDP, overlays TCP throughput has out-performed than VM which they are running on, needs further investigation [4]. [4] also output that native overlay driver performance is nearly equal to third party solutions which is conflicting with results of [1].

Zismer et al. examined the performance of Docker overlay networks Libnetwork, Flannel and Weave such as former studies [5]. They focused on the problems which are overlay's very low UDP throughput whereas outperforming TCP throughput than VM's reported by Hermans et al. [5]. By using the iperf3 and netperf, they explored on correlation between CPU usage and throughput, impact of fragmentation on throughput, impact of UDP and TCP offloading together with Weave's multi-hop routing [5]. They also suprisingly discovered that hardware offloading is enabled on Docker containers by default [5]. They found that segmentation offload increased UDP thoughput of overlay networks however their throughput couldn't exceed the VM native throughput. Furthermore, TCP throughput is boosted and overperformed than VM throughput which explains the TCP over-performing results in [4].

Unlike previous studies, Abbasi et al. compared the performance of Docker overlays Flannel, Weave, OVS DPDK, Soft-ROCE and Hard ROCE located in two physical hosts [6]. They showed that highest CPU utilization is observed for OVS-DPDK unlike RDMA based solution which has the minumum CPU consumption [6]. Results showed that flannel gave the highest latency and the minimum throughput together with Weave [6].

2.2 FORMER STUDIES UTILIZING CONTAINER ORHESTRATOR

Implementing cluster networking solutions by the help of container orhestrator platform utilize the several benefits of container orchestrator features such as continuity, scalability, failover, recovery, healthcheck, replication etc., although implementation is more difficult according non-orhestrated solutions.

Kwon et al. designed container-enabled HPDA cluster within Kubernetes as container orchestration engine and Docker as a container runtime interface within different file systems for two different Kubernetes CNI plugins: Calico and Weave [7]. They benchmarked CNI's by increasing the number of executors combined with different filesystems, taking job execution time as performance indicator and got the minimum execution time with Calico and BeeGFS combination [7].

Mao et al. studied the problem of minimizing latencies in a real-time cloud using Docker and Kubernetes with three different CNI plugins which are Flannel, DPDK enabled Open vSwitch and plain Open vSwitch inluding several radio access network equipments such as EPC, HSS, ENB etc [8]. They reported that Flannel and DPDK gave the same throughput while OVS has slighly less than others in cluster solution [8]. In the same scenario, Flannel had the highest latency and DPDK had the minumum latency [8]. They put out that DPDK was the optimum solution with fifty percentage latency of plain Open vSwitch, although it had equal throughput with others [8].

Park et al. designed network structure based on CNI utilizing OpenStack cloud platform, Kubernetes container orhestrator and compared throughput of Flannel-VXLAN, OVS-VXLAN and OVS-GRE according to message sizes. [9] They put out that

OVS-GRE and OVS-VXLAN gave the nearly same throughput which are higher than Flannel overlay [9].

Krishnakumar et al. tried to discover container networking fast packet processing solution in Kubernetes to meet the latency requirements in NFV and they took the network latency as a performance indicator in the benchmark tests [11]. He compared both non-orhestrated and orhestrated container networking scenarios using DPDK, SR-IOV, Open vSwitch (OVS), Open Virtual Network (OVN) [11]. Besides, he utilized Multus as integrator of DPDK for Kubernetes and Flannel as management interface in Kubernetes [11].

They reported that latency of application container, utilizing DPDK in Kubernetes was less than 10 percent of application, utilizing only OVN in Kubernetes and this latency is equal to DPDK container running standalone [11]. This study showed that it is possible to achieve the same performance results of standalone containers even containers are managed by an orhestrator.

As explained above, most of the studies focused on well-known solutions such as Docker native overlay, Flannel and Weave. Calico could not be tested extensively as BGP protol testing in cloud implementations requires additional support and Cloud providers assured to give these features in recent years. Open vSwitch included in small number of studies generally which are interested in performance improvement methods. That's why it is implemented with and without DPDK and compared with other performance improvement methods such as RDMA in some studies. Therefore, there is not an exhaustive study which compares OVS with other container networking solutions such as Calico, Weave, Flannel and Libnetwork together.

2.3 FORMER STUDIES INVESTIGATED ON LOAD BASED PERFORMANCE

Most of the studies examined the performance of popular container cluster networking solutions within different performance indicators such as throughput, latency, response times, bandwidth within different applications on local environments or public cloud infrastructure. Merely, few studies have focused on performance under load. Those researches are generally executed on High Performance Computing (HPC) platforms

with the applications which has different resource intensities such as data and computing. Nevertheless, a few studies actualized on public cloud environments within the workloads which has similarity to common VNF applications.

Buzachis et al. investigated the performance of four network overlays which are OVN, Calico, Weave, and Flannel by using Kubernetes in a clustered environment designed for Cloud, Edge and Fog [10]. They deployed distributed microservices based on FTP and CoAP then executed the benchmarks by considering the transfer times of each microservices [10]. They presented that OVN based solution is better than others as it gave lower execution time and offers several functionalities in order to manage geographically distributed microservices [10].

They compared the overhead of overlays according to container running in the host within five different configurations and increasing payloads [10]. They found that best solution is the OVN with better time performances in all configurations both for CoAP and FTP scenario [10].

Ruan et al. researched for the suitable way to use containers from different aspects and performed series of experiments to measure performance differences between application containers and system containers by observing the overhead of extra virtual machine layer between the bare metal and containers [43]. Then, they inspected the service quality of ECS (Amazon EC2 Container Service) and GKE (Google Container Engine) are different. Besides, they found out that system containers are more suitable to sustain I/O bound workload than application containers as application containers are suffering from high I/O latency because of layered filesystem [43].

Martin et al. investigated the interoperability of Rkt container in high performance applications by running the HPL and Graph500 applications, then simulated the performance with the commonly used container technologies such as Docker and LXC containers. They analyzed the performance results of data intensive and computing intensive practical applications in several scenarios [44].

[45] explored on measurement of Docker performance from the perspective of the host operating system and the virtualization environment by outputting a characterization of

the CPU and disk I/O overhead generated by containers. [45] studied with two use cases; CPU intensive workload and disk I/O intensive workload doing sequential or random reads, writes. However, they could not find the correlation between quotas and overhead which requires further investigations [45].

[46] studied on resource management problems in containerized HPC environments by aiming achieve better application performance and system utilization from three perpectives which are CPU allocation for CPU-intensive applications, disk I/O contention with disk I/O load unbalance and network bandwidth throttling, prioritization. They indicated that a communication-intensive containerized application may experience unacceptable performance when hosted on nodes with other containerized applications because of parameters such as bandwidth, reliability and packets per second can not be guaranteed for a specific HPC application. They proposed a mechanism that enables bandwidth limits and privileged delivery service for critical applications in containers. They claimed that this solution could control latency and delay while providing a way to reduce data losses. Besides, proposed network management mechanism enabled application specific bandwidth priorities and bandwidth limits [46].

[47] studied the impact of using containers in the HPC research environment by evaluating the performance of Linux-based container solutions which are based on cgroups and namespaces by executing NAS parallel benchmarks in several configurations. [47] realized the several benchmarks with different profiles of execution and significant number of machines to measure the uses of containers in HPC area in terms of portability of complex software stacks and oversubscription.

They effectuated the several benchmarks such as LU, EP, CG, MG, FT, IS from NPB benchmark suite by modelling different communication patterns within each benchmark. [47] showed that the container technology is becoming mature and performance problems are being construed within the new releases of Linux kernel [47].

Several studies evaluated the multi-host container networking solutions according to specific application requirements. They have presented that it is difficult to choose the best performing overlay as it depends on the requirement criteria, environment, expected features which it offers. Generally, each study focuces on only few popular

solutions and there is not a study widely covering most of applicable container networking solutions by considering behaviour under load and reliability. Additionally, there is not any study which investigate on traffic load effects for common VNF workloads and environments.

This thesis is imlementing the traffic load based performance and reliability analysis within mostly used VNF workloads which are MongoDB database and web application. Besides, it is such compherensive that it covers several multi-host container networking solutions including OVN which is rarely compared with other popular solutions. Herewith, the research is giving the contribution by widely comparing most of container networking solutions from the aspect of behaviour under load with several performance indicators.

3. CONTAINER MULTI-HOST NETWORKING

3.1 VIRTULIZATION AND CONTAINERIZATION

Virtualization implies that running a virtual instance in a computer system by isolating it from the hardware. Generally, it purports to running multiple operating systems in a single hardware at the same time. From the applications inside of virtual machines, they seem running their own dedicated environments within libraries, operating systems and other programs as each virtual instance running independently with its own operating system without connection to other instances or host.

Virtualization facilitate the NFV and IT implementations by faster deployment, reduced infrastructure and operating expenses and complexity, increased application availability, flexibility and performance. Virtualization is very essential for efficient usage of hardware resources such as processors, memory, storage and I/O by using a software solution that allows to share those resources between several instances called virtual machines. Besides, the software which is used to create the separation layer for resources called hypervisor.

Hypervisor allows one pyhsical machine to host several virtual machines by maximizing the effective utilization of CPU, memory, network resources in addition to isolating the resources of virtual machines from each other. The first hypervisors were introduced in the 1960s to allow for different operating systems on a single mainframe computer [12]. Hypervisor became more popular after Linux and Unix operating systems began to utilize virtualization functions such as cost controlling, improved reliability and security, hardware capabilities dilation around 2005 [12].

There are mainly two types of hypervisors called type 1 baremetal or native and type 2 hosted. Native hypervisors administer the guest operating system and each virtual machines seems as a process on it. Vmware ESXi/ESX, Citrix, XenServer are mostly known native hypervisors [12]. VirtualBox, QEMU are extensively used hosted hypervisors by running a virtual machine as an application and isolating it from the operating system.

Besides, there is another type of hypervisors called hybrid which runs as a program while strictly integrating with hardware and kernel of host such as KVM.

Virtualization and containerization are two ways of insulated and multiple application deployment in a single environment. Although, they are alternative to each other, they have certain differences from the aspect of isolation, resource consumption and efficiency. Containerization is quite lightweight and efficient compared to virtualization as it only encapsulates the application in the same operating system [13]. This makes containers faster and smaller. However, server virtualization uses independent operating systems for each instances together with several unnecessary application and code duplicates.

Containerization of applications introduce several benefits such as efficient utilization of resources, portability, easy life cycle management, easy scaling, faster start-up compared to VM's. Container technologies in Linux generally take the advantage of mature and performant Linux kermel feautures. This improves the containers robustness, and performance.

3.2 CONTAINERS

Containers are lightweight software constituents which are dependent to an image for the configuration by running on traditional operating systems in isolated user environments [14]. Containers can be deployed much faster with shorter start-up times and movable between different environments and operating systems and platforms.

Containers called as control groups (cgroups) in 2007, after Google annunciated the launch of container processes which are designed for resource usage restriction and isolation in 2005 [15]. LXC has born by combining Linux kernel 2.6.24 with cgroups in 2008 and it provided operating system level virtualization within sharing Linux Kernel to several isolated environments which are called containers having their own network spaces and processes [15].

Docker which was firstly reposed of LXC and LXC's replacement Libcontainer released in 2013 as an open source Project. Docker equipped with Linux namespaces,

AppArmor security profiles, Libcontainer control groups, network interfaces and firewall rules in the beginning because of relying on Libcontainer [14]. Contributions to Docker proceeded with local and global container registries, CLI client, restful API and cluster management system named Docker Swarm.

There are plenty of container types. Docker is the mostly used which is 83 percentage of all containers in 2018, that number is down from 99 percentage in 2017 [15]. Other container runtime environments including CoreOS rkt, Mesos, LXC and others are constantly enlarging as the demands continues diversify and expand.

Rkt which has some unique features such as supporting Trusted Platform Modules (TPM) uphold two types of images which are Docker and appc. However, compared with Docker, it has fewer third-party integrations and deficities such as OCI compliance which is never planned to develop. Besides, Rkt is not developing appc anymore. Furthermore, Rkt which is upholded by Red Hat announced it's CRI compliance with Rklet just 2 years ago. By providing compatibility to public standart, Rkt spread out it's usage by community.

Mesos which is developed by Apache offers quality performance and it supports both appc image and Docker image types. Mesos started to developing for OCI support, however Mesos containers requires Mesos framework to make them run which is a potential drawback currently.

LXC came to stage before Docker, however it lost its popularity against to Docker. It includes three main components which are the runtime lxc, a daemon which manages containers and images named lxd and lxfuse which conducts the file system. Currently, they are preparing an expanded LXD which offers a new UI and CLI. LCX's main disadvantage is lack of OCI compliance which obstructs its integration with Kubernetes.

OpenVZ which is started in 2005 as an extension of the Linux kernel provides container-based virtualization by running several virtual servers inside one Linux operating system [15]. OpenVZ brings benefits of low memory consumption compared to other container runtimes because of hosts share a single kernel.

Containerd developed as a Project by Cloud Native Computing Foundation (CNCF) in order to accentuate simplicity, stability and portability. Besides, containerd can be used as a daemon working together with gRPC by supporting OCI images both in Linux or Windows operating systems together with many lifecycle management features.

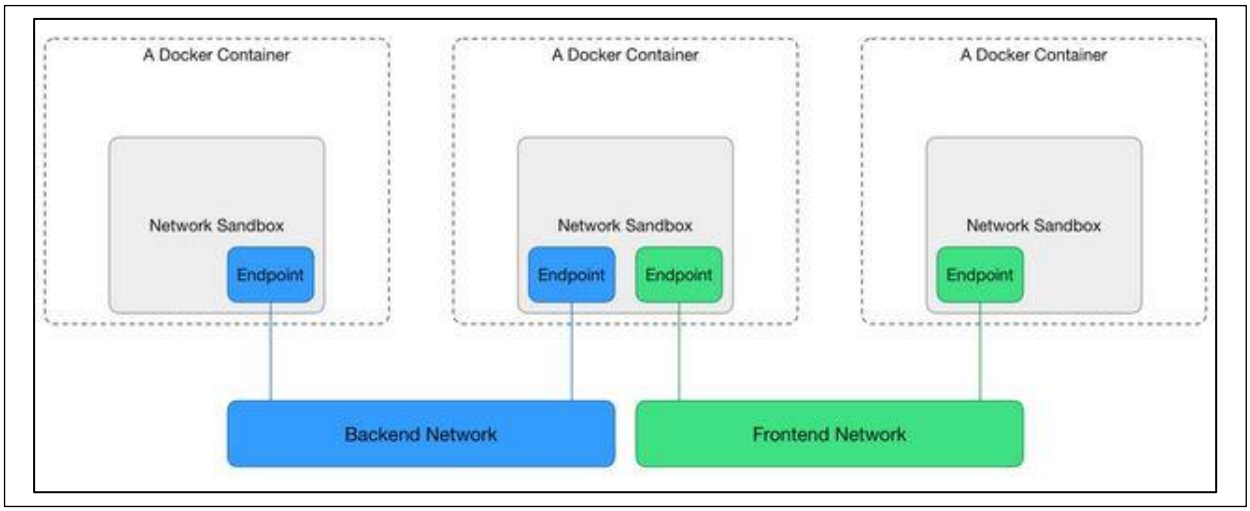
Besides above, there are also several container types such as Windows Server containers, Linux VServer, Hyper-V containers, Unikernels, Java containers which have very small amount of distribution.

3.3 CONTAINER NETWORKING

3.3.1. Container Network Model (CNM)

CNM is a standard, proposed by Docker's Libnetwork Project. Container Network Model (CNM) stamp out the steps required to provide networking for containers which puts the isolation to support multiple network drivers. The CNM is developed on three main components named Sandbox, Endpoint and Network which are clearly displayed at Figure 3.1.

Figure 1.1: Container network model architecture



Source: [16]

A sandbox which is isolated network environment holds the configuration of a container's network information including container's interfaces, DNS management

settings and routing can preserve multiple endpoints from several networks. Sandbox can be implemented within Linux Network Namespace or other similar environments.

An endpoint is attached to a Sandbox to forge the Network. It represents the connection point of container which access the network such as veth pair or a physical network card. An endpoint can belong to only one network and only one Sandbox. If it is required a sandbox to be joined to multiple networks, multiple endpoints need to be added to sandbox.

A network is a set of endpoints which are connecting to each other directly. Networks which are containing many endpoints can be builded by a Linux bridge or a VLAN, etc. Because of using separate network space for each network on the host, containers on the same network can join to the network namespace [2].

3.3.2. Container Network Interface (CNI)

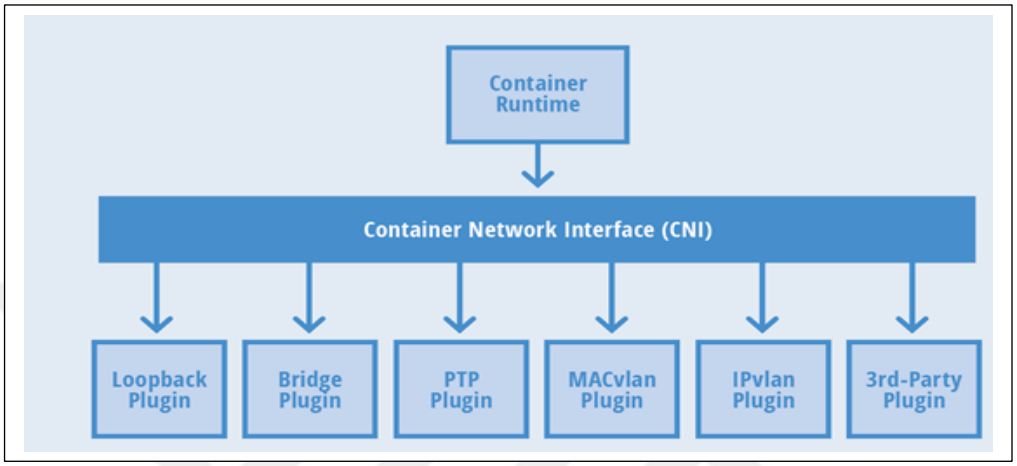
CNI proposed by CoreOS and interiorized by several projects such as Kubernetes, Rkt, Apache Mesos, Cloud Foundry, represents the container networking specifications as a simple contract between container runtime and network plugins.

CNI is an interface between a network plugin and network namespace which is a container runtime such as Docker, Rocket. CNI formalize the network plugin implementation by getting a container runtime and configuring it to network by attaching and detaching. Several plugins are created to be implemented for CNI by the projects Calico, Contiv, Weave etc. A JSON file is used to construct the CNI architecture by defining input and output from network plugins. Containers can be joined to multiple networks driven by different plugins. CNI plugins generally supports two operations which are adding and removing network interfaces to and from networks. CNI architecture is represented at Figure 3.2.

CNI and CNM are not opposite standarts however there are some differences between them. CNI is more generic and can be used with any container runtime while CNM only supports Docker runtime. It is easier to create a CNI plugin than a CNM plugin as CNM requires a distributed key-value-store like Consul, Etcd. CNI uses JSON

format which is becoming a de facto standart for configuration day by day. Besides, there are several projects constituted on CNM such as Kubernetes, Kuryr, Open Virtual Network (OVN), Project Calico.

Figure 1.2: Container network interface architecture



Source: [17]

3.4 DOCKER

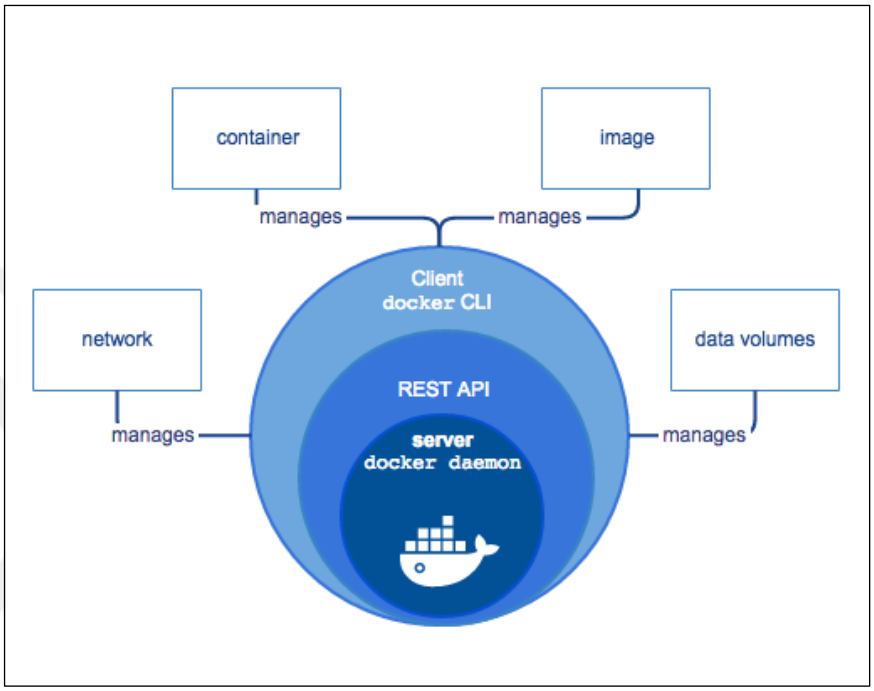
Docker which is an open source platform written in Go aims to develop, move and run the applications easily. Docker can deliver the applications rapidly by the help of isolation between the applications and infrastructure in addition to utilitization of Linux kernel's several features.

Docker uses the isolated environment called a container to wrap and run the application. Many containers can run at the same time in one host by courtesy of isolation and security. Docker provides lifecycle management of containers to deploy the application as a standalone container or an orchestrated service. Docker has a client-server architecture which is demonstrated at Figure 3.3 which consists of docker daemon (server), docker CLI (client) and REST API.

The Docker daemon (dockerd) performs the Docker API requests, manages Docker objects such as networks, volumes, images and containers, in addition to talking with other daemons in order to steer Docker services. Besides, Docker daemon forge, execute and disperse the docker containers.

The Docker client communicate with the Docker daemon locally or remotely through a REST API over a network interface or UNIX sockets. The Docker client (docker) provides the interaction between Docker users and Docker can talk with more than one daemon.

Figure 1.3: Docker platform architecture



Source: [18]

Docker images are stored in docker registry. There are public and private registries. While private registries can be defined by user, default registry, Docker Hub can be used to configure or search for the images publicly.

Docker images are read-only templates called Dockerfile inludes several directives to create a container. Private or public images can be created and hold in the public or private of registries.

Docker also provides in-built cluster management services named Swarm which is available for Docker 1.12 and higher versions [18]. Services scale containers across many Docker daemons, by working together with multiple Swarm managers and workers.

All swarm members are docker daemons and they use Docker API to talk to each other to ensure the desired states such as number of replicas and load balancing.

Docker constitutes set of namespaces for each running container to provide the layer of isolation. Docker Engine utilizes the Linux components such as uts namespace for kernel isolation, ipc namespace for IPC access management, pid namespace for separation of processes, mnt namespace for filesystem management, net namespace for network interfaces management and cgroups for limiting the application from usage of particular resources.

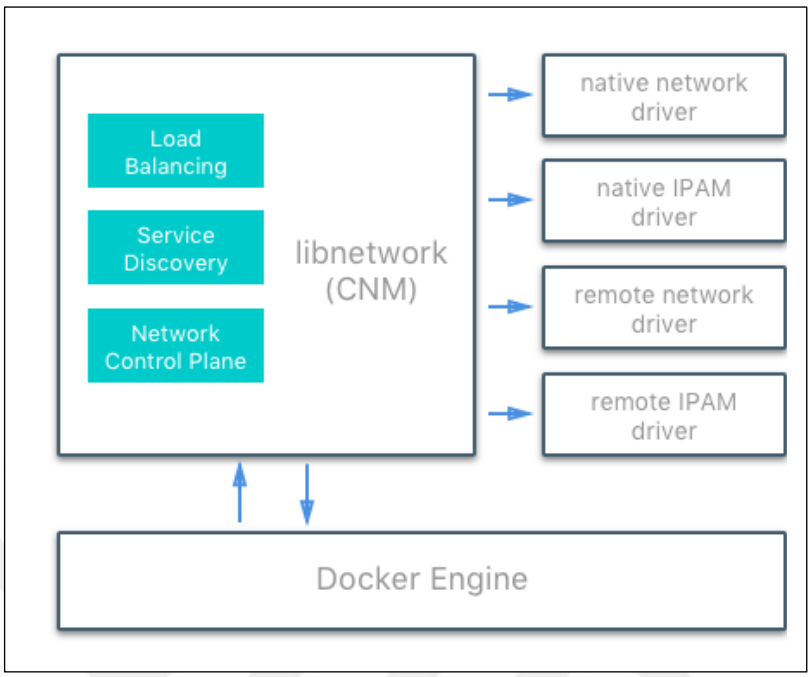
Generally, cgorups assures the sharing and restricting the hardware resources between containers. Docker Engine uses UnionFS to build the containers because of it's lightweight and fast operations. Docker Engine employ several sort of UnionFS such as DeviceMapper, AUFS, btrfs and vfs [19]. By combining of namespaces, UnionFS and control groups, container format called Libcontainer is created by Docker Engine.

3.5 DOCKER NETWORKING

Docker networking lies down on the Linux networking by using several mature Linux kernel features such as TCP/IP stack implementation, packet filtering and VXLAN. Use of existing Linux kernel features brings several benefits such as high performance, sturdiness, changeover between different distributions and portability.

Libnetwork was born from the merge of libcontainer and Docker Engine by introducing the Container Network Model (CNM) and Libnetwork arhitecture is illustrated at Figure 3.4. Through the agency of CNM, all containers in the same network can talk to each other without any additional adjustment. In case of requirement to separate traffic between containers, containers need to attach to the multiple networks by using multiple endpoints. Network namespaces, iptables, veth pairs, Linux bridges are the main generation blocks of CNM network drivers. These tools also provide the management for network seperation, dynamic network policies and forwarding rules.

Figure 1.4: Libnetwork Architecture



Source: [20]

Linux bridge which is virtual form of physical switch commonly used in Docker network drivers and functions as Layer 2 device by forwarding traffic based on MAC addresses. Docker bridge network drivers are very similar to Linux bridges as they are implemented on top of Linux bridges. Docker network also uses the Linux networking interface (veth) in order to provide particular connections between namespaces. Containers connect to Docker networks by veth when one veth is resided inside the container while other has joined to Docker network. Docker network also employ the iptables which acts like native L3/L4 firewall in Linux kernel in order to traffic separation, load balancing and port mapping processes.

Docker has five default networking modes named bridge, macvlan, host, overlay, and none. Besides, Docker also provides pluggable networking subsystem which can be integrated with several custom network drivers.

Host mode adds the container directly to the host's network by extracting all the isolation between container and host network stack. None network means the deficiency of network interface in the container. Bridge is the default driver as if

none of the driver type is given, container will be created within bridge network which is represented by docker0. Multiple containers on the bridge network can communicate with each other similtaneously and also external access can be provided by bridge driver.

Containers can access to each other through their ip addresses as automatic DNS resolution is not supported by default bridge driver. If automatic name resolution is required, user defined bridge networks can be created.

MACVLAN driver assigns a MAC address to the container which is seemed as a physical interface in the network by forwarding traffic to containers using this MAC addresses. Using MACVLAN driver usually provide better performance than default bridge because of container is directly connecting to pyhsical network instead of routed through host network stack.

Bridge network is also used to create overlay networks. Overlay network create a high level network over pyhsical network infrastructure to provide container to container communication between different hosts. Overlay network helps to implement a subnet overlaid on the physical networks of different hosts by providing communication between containers that uses their own default gateway, subnet and ip addresses.

Docker default overlay network is implemented by swarm services with docker_gwbridge which provide the traffic flow between swam managers and workers [21]. Additionally, docker_gwbridge is used for external connectivity between any container to external networks or swarm nodes.

Customized overlay networks also can be created for multi host container networking instead of Docker default overlay. In that case, docker_gwbridge can be used with custom configuration. Overlay networks communicate with multiple Docker daemons in different hosts within VXLAN by the help of Linux bridges, Libnetwork and Linkv. There are also other type of encapsulation methods such as GRE, GENEVE,

STT can be used to build custom overlay networks which will be mentioned in the next chapter.

Docker Engine in swarm mode uses VXLAN tunnels as Layer 2 overlay over Layer 3 infrastructure by carrying all the packets as IP/UDP format. VXLAN tunnels are lasted with VXLAN Tunnel Endpoint (VTEP) which realize the encapsulation/de-encapsulation process. New network namespaces are created in each host to attach the containers to this single VXLAN overlay.

Custom overlay networks can be created within Docker Engine which is not in swarm mode, requires a valid key-value store to keep the networking information that can be synchronized and reachable by all overlay members. Docker engine supported key-value stores are Etcd, ZooKeeper (Distributed store) and Consul.

There are also other custom methods to provide multi hosts container communication. Project Calico brings Layer 3 approach virtual networking for multi host container communication by routing traffic instead of tunnelling.

In this thesis, various types of container networking solutions are implemented and analyzed. Overlay methods including Libnetwork with Etcd key-value store, Weave, Flannel, OVN; Layer 3 approach Project Calico are investigated. While Libnetwork, Weave and Flannel using VXLAN tunneling, OVN based overlay has implemented by Geneve Encapsulation.

3.6 MULTI HOST CONTAINER NETWORKING SOLUTIONS

3.6.1 OVN with Open vSwitch

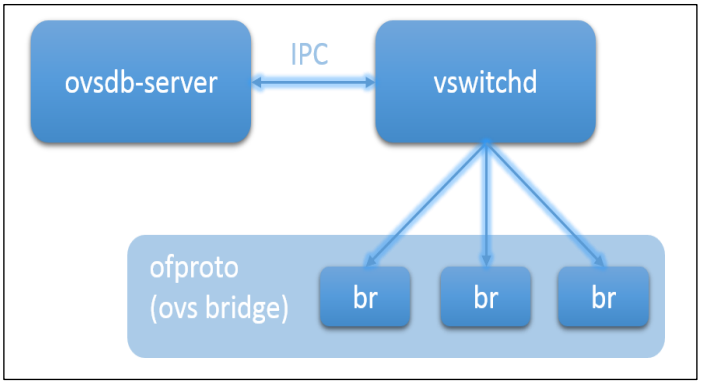
Open vSwitch (OVS) has born from an academic Lab named Ethan SIGCOMM while testing the flow based switches by the help of a central controller which forged the Open Flow protocol and it is released at 2009 as OVS [22]. OVS is supported by many virtualization platforms such as Docker, VirtualBox KVM, Openstack, oVirt even it can be used in hardwares.

OVN, Open source virtual networking for Open vSwitch (OVS) is a system to provide virtual network partition supplements the capabilities of OVS. It has developed by the same community as Open vSwitch. OVN which is implemented using L2 and L3 overlays assures virtual network segregation for OVS, together with managing connectivity to physical networks.

OVS is a production quality, multi layer software switch which is licensed under open source Apache 2.0 license with flow based management and Open Flow capability. It is written in C. Open vSwitch utilize the virtual bridges and forward the packets through the flow rules between hosts by connecting the tap interfaces and namespaces. It is kind of Linux bridge storing the MAC addresses of attached devices within enhanced capabilities such as supporting several overlay networking methods (GRE, VXLAN, Geneve) and hardware integrations.

OVS consists from three main components named vswitchd, ovsdb-server and kernel module (datapath) which is exhibited at Figure 3.5. OVS main process vswitchd is a userspace program. It passes the configuration from ovsdb-server to ovs-bridges and carries the status and statistical information back to ovsdb-servers through the IPC channels.

Figure 1.5: Open vSwitch Structure



Source: [23]

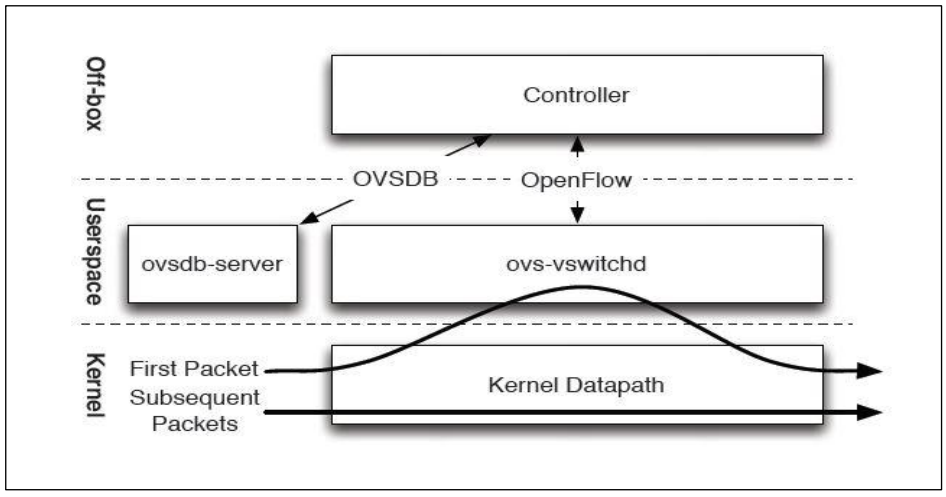
OVSDB stores the persistent configurations accross any reboots while temporary configurations are kept by vswitchd and datapaths. OVSDB works in combination with ovsdb-server which provides the interface for OVSDB. This JSON-RPC type interface

passes the client connections through the active/passive Unix domain sockets or TCP/IP. Ovsdb-server can wok in two modes; active or passive, however only the active server transactions can change OVSDB.

Comparing with Linux bridge, OVS functions differently as Linux bridges use the VLAN to tag the traffic. However OVS uses the flows to orient the inbound and outbound traffic behaviour.

Datapath, which is different from ovsdb-server and vswitchd, is builded in kernel space in order to give high performance. Datapath makes the packet forwarding by caching the flows and executing the actions on the received packets according to matched flows which is called fast datapath which is dipslayed at Figure 3.6. If the received packet is not matching with any flows, this packet sent to ovs-vswitchd in userspace. In that case, usually new flow is created by the ovs-vswitchd and following packets are forwarded in the kernel space without userspace interaction according to this new flow. Every packet which goes to userspace decrease the performance of OVS and this is called slow datapath. Real performance improvement comes from the kernel (fast) datapath. In case, fast datapath is mostly used, OVS proccessing speed is faster than Linux bridge. Fast datapath also improves the multithreading performance [22].

Figure 1.6: Open vSwitch Traffic Flow



Source: [23]

Actually, datapath generation according the different types of traffic is OpenFlow controller's responsibility. OpenFlow controller functions as a manager of all OVS components. Because of flow management through the OpenFlow controller, OVS is quite important for NFV functionality. Main requirements such as traffic differentiation and QoS management can be satisfied by Open vSwitch together with an OpenFlow controller such as ONOS or OpenDaylight. The controller yields the northbound interface between network application and southbound interfaces to communicate with all OVS components.

The actions which will be implemented on received packets, desribes the flow. Flows formed by the actions such as forwarding, dropping, modifying the received packets and mostly used flows are cached in the datapath. Besides, the flows which are less used are stored in the ovs-vswitchd. Ovs-vswitchd, communicates with OpenFlow controller which is in the off-box through the OpenFlow message format; ovsdb-server with OVSDB-protocol format (RFC 7047) and datalink via Netlink [23] depicted at Figure 3.6.

3.6.2 Docker Libnetwork Overlay with Etcd Key-Value Store

Libnetwork overlay is Docker default overlay which is used to provide communication between multiple nodes. Build-in overlay is implemented by swarm services, however swarm is not used in this study. Docker engine is used with external key-value store named Etcd. Etcd cluster forges the management plane in multi-host networking.
Libnetwork overlay uses native VXLAN features to build the overlay network together with iptables, Linux bridges and veth. VXLAN is an old solution which is part of Linux kernel till version 3.7 [24]. Overlay datapath exists in the kernel space, it brings the benefits of less CPU usage, low latency, fewer context switches and direct traffic between NIC and the application. Layer 2 ethernet frames are encapsulated in Layer 3 UDP packets, by identifying each layer 2 subnet with a VXLAN header, container traffic is traversed between the hosts. This process requires VXLAN tunnel endpoints (VTEP) to encapsulate and decapsulate the packets that brings the significant performance overhead [2].

Each container joined to the overlay network has an ip address which is known globally as cluster state information, stored in the Etcd and accessable by all hosts. As a part of Docker built-in function, Proxy address resolution (ARP) is used to retain the each container's host IP location through the Etcd sync communication. For each host, a Linux bridge device is generated per subnet and each container joins to this subnet for indicated overlay. By this way, traffic between the containers on the overlay is encrypted by IPsec tunnels.

3.6.3 Flannel

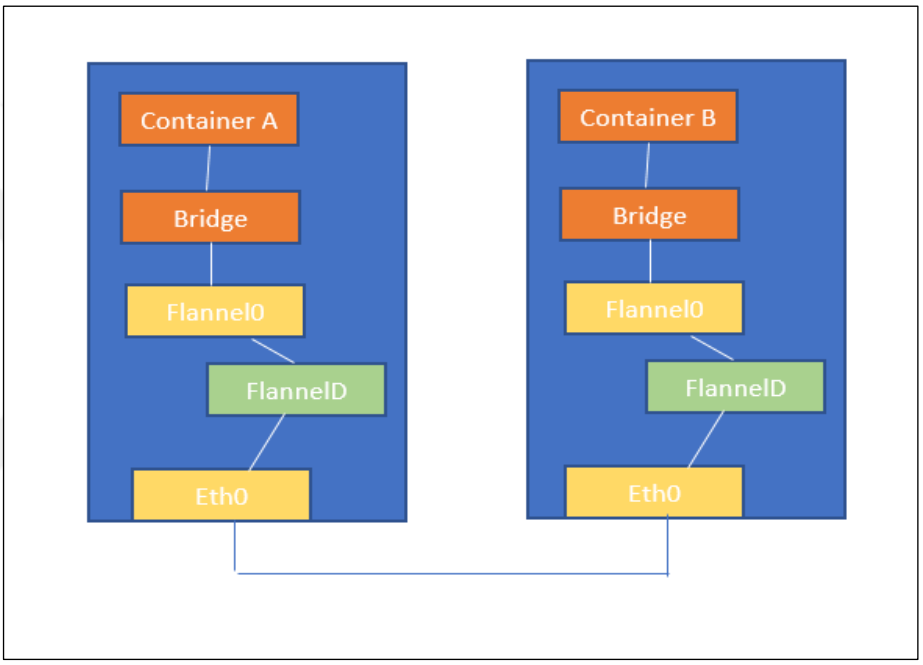
Flannel is built by CoreOS which is creator of the rkt container and designed to be simple, lightweight and transcendent in performance. Flannel assigns each Docker daemon an ip segment by solving the IP conflict in Docker default configuration. However, it has drawback that it is not possible to assign fixed ip addresses to container and brings deficiency of multi subnet isolation [2]. It stores the assigned subnets and hosts routing table, hosts information in a distributed key-value store such as Etcd. Flannel offers a layer 3 IPv4 network between overlay host by controlling the traffic put accross between hosts. Flannel cluster networking implementation on two hosts are illustrated at Figure 3.7.

Flannel has several backend modes like UDP, VXLAN, AWS VPC and GCE [2]. Although Flannel's default backend is UDP, VXLAN mode is commonly prefered in production as VXLAN forwarding increase the performance of the overlay solution according to UDP. UDP backend offers an IP-over-IP solutions by using a TUN device to encapsulate the IP fragments in UDP packets.

VXLAN backend is similar to Docker Libnetwork overlay solution. In VXLAN backend, Linux kernel VXLAN tunnel devices are constituted and user space process flanneld has executed. Flanneld uses the preassigned subnet information which is stored in key-value store and fills the bridge forwarding database and ARP table on each host. How to perform the traffic forwarding between physical hosts is already known by Flanneld by using iptables.

Weave and Calico build discrete containers for network management while Flannel only runs a process named flanneld on each cluster host. Besides, Flannel does not have an integration with the Docker libnetwork plugin, while other solutions already offered the integration [25]. Nevertheless, Flannel provides a CNI plugin for Kubernetes and it is the default networking solution of Kubernetes. Flannel is the most mature container networking solution.

Figure 1.7: Flannel Networking Structure



Source: [26]

3.6.4 Weave

Weave is widely deployed overlay networking solution builded by the company Weaveworks. It has two working modes named fastdp added in Weave version 1.2 which is default mode and sleeve (pcap). Weave starts with fastdp mode except using untrusted networks and encryption. Fastdp mode is quite similar to libnetwork which uses encapsulation with VXLAN and encryption via IPSec. But, Weave brings a prominent difference by using Open vSwitch datapath module in Linux kernel to accelerate the traffic flows by reducing context switches [27]. Running this module in

kernel space throughout the known flows boosts the performance. Besides, Weave also deploys Linux bridge for traffic broadcasting then learn and update flows.
Weave sleeve mode uses UDP encapsulation which is similar to VXLAN in terms of encryption required. Sleeve mode route the traffic along user space and perform encryption via NaCL [28].

Weave assigns the ip addresses to containers in overlay hosts from the same subnet but these ip addresses are not sequential unlike Libnetwork. Weave comes with two different plugins; weavemesh and weave. Weavemesh which is deployed by default works without key-value store and weave operates with a global cluster store. Weavemesh uses the conflict-free replicated data type (CRDT) to store all the clusterwise information such as subnets, traffic routing information [29].

Weave networking occured from Weave routers such as software routers and bridge interfaces in every cluster hosts. All the containers together with Weave routers connect to bridge interface. As Weave operates throughout Docker's existing bridge networking, containers in the single host can communicate without using Weave. Weave router is not utilized for local switching. Because of Weave Routers carry the topology information to other peers, all the peers know entire topology. Weave routers set up the TCP and UDP connections between all overlay hosts. While UDP is used for data encapsulation, TCP is applied by protocol related exchange and discovery. Weave has added the Libnetwork plugin in version 1.2 and it has a build-in DNS server. By the help of this, service discovery is performed easily. This also brings the benefits such as load balancing and simplified high availability for applications [30].

3.6.5 Calico

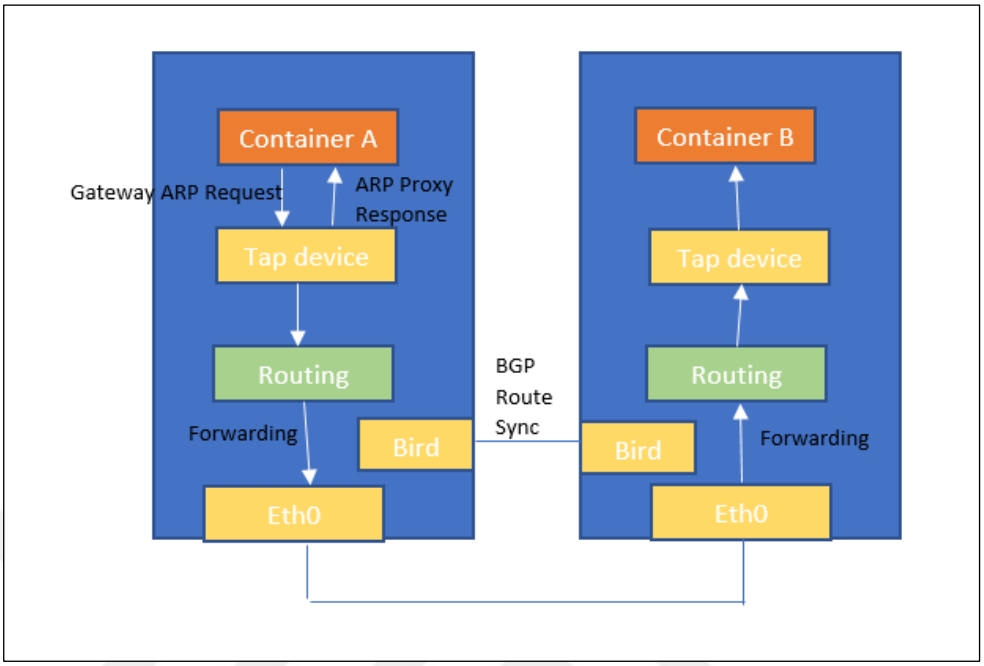
Calico is the only tested solution which does not apply any encapsulation methods, so it is not an overlay solution in fact. It is pure layer 3 network by establishing the multi hosts container networking through the IP forwarding functionality in Linux kernel. Calico uses the Border Gateway Protocol (BGP) client named Bird Internet

routing daemon (BIRD) together with vRouter to manage the end to end layer three networking. Calico cluster networking architecture between two hosts is shown in Figure 3.8. Calico handles the data forwarding through the vRouter in each host and propagate the workload routing data to all Calico network by the help of BGP protocol [2]. In this way, it is guaranteed that all data traffic is transported by IP packets.

Node interconnection is easy in Calico network as containers have their own network protocol stack. Besides, Calico's transfer efficiency is elevated because of wielding the elements of data center network directly. In a physical datacenter, Calico hosts can match with datacenter routers and ensure the route information of containers such as physical network devices. Routes of containers are announced through the physical network in Calico's control plane which is consists from several docker containers such as BGP route reflector. Calico requires a BGP route reflector for large networks. Host ip tables are used to ensure the segmentation between containers. Routers on every hosts are adjusted in order to relate the subnets of containers and the host ip. Calico need to keep and run several routing tables as those tables are instantly and dynamically generated and managed. Calio utillizes the distributed key-value store to keep information of clusterwide hosts, subnet and ip addresses. Calico has a deficiency that it has very large routing tables as every container has an ip address [2]. Even if Calico has complex routing task, it is not affected with any overlay overhead such as NAT and tunnelling. This makes Calico very appropriate for the environments which requites high performance and isolation.

However, BGP is not supported in some virtual environments. In that case, Calico offers IPIP tunnelling solution by encapsulating IP in IP and implementing a layer 3 network on top of layer 3. IPIP solution adds 20 bytes additional outer header for every transported packet which includes the host ip address of destination container [3]. Even using IPIP, it has less overhead than VXLAN because of small encapsulation header. Although Calico does not offer a built-in encryption method, it assures the forcefull and stable network policy implementation. Calico yields the plug-in for Libnetwork [4].

Figure 1.8: Calico Networking Structure



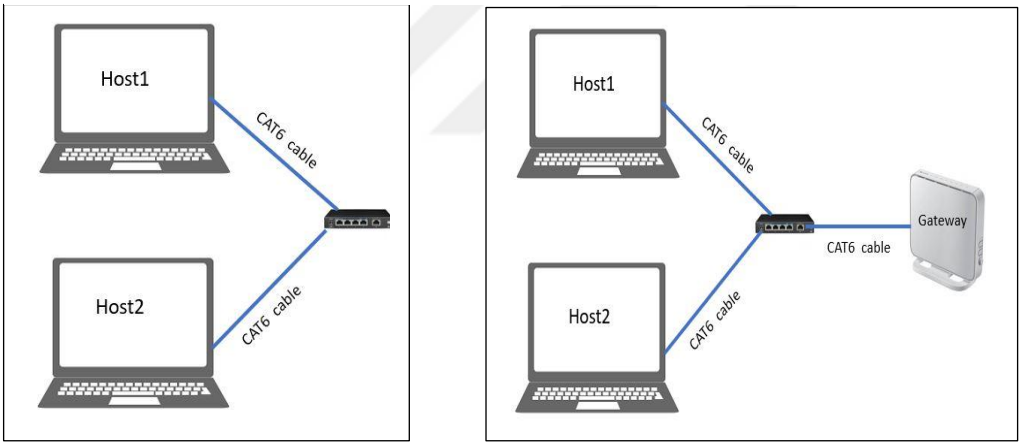
Source: [26]

4. TEST ENVIRONMENT

4.1 EXPERIMENTAL SET UP

Multi-host container networking environment consists from two bare-metal hosts, one Gigabit Ethernet switch and one gateway. Computer Dell Latitude E7440 with 4 CPUs Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz, 16 GB memory, 1Gbit/s Intel Corporation I218-LM network adaptor is used as host 1. Host 2 is Dell Inspiron 15 7000 Gaming computer with 8 CPUs Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 16 GB memory, 1Gbit/s Realtek 8169 network adaptor. Each hosts use the Ubuntu 16.04.5 LTS operating system with kernel version 4.15.0-66-generic x86_64. Host 1 and host 2 detailed properties are recorded at Table 4.1 and Table 4.2 respectively.

Figure 1.9: (a) Test environment illustration (b) Test environment illustration only for external web access benchmark tests



a) b)

Two hosts are connected through the CNet CGS-800 8 port Gigabit Ethernet switch by CAT6 cables which is illustrated at Figure 4.1 a and device specifications are written down at Table 4.3. Except Apache jmeter test execution, gateway device is not utilized. During the external web access benchmark tests executed with Apache Jmeter in order to improvise it to a real world scenario, 4 ports x 100 Mbit/s ADSL2 VDSL2

Huawei HG658 V2 gateway device is added to the environment and this case is presented at Figure 4.1 b. Table 4.4 contains the gateway device properties.

Table 1.1: Test-host1 properties

Host 1 Model	Dell Latitude E7440
CPU(s) number	4 (2 cores per socket, 2 threads per core)
CPU model	Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
NUMA node	1
Virtualization	VT-x
Memory	16 GB
Network adaptor	Intel Corporation I218-LM e1000e Driver Version 3.2.6-k Firmware 0.7-3
Network capacity	1Gbit/s
OS Relese	Ubuntu 16.04.5 LTS
Kernel version	4.15.0-66-generic x86_64

Table 1.2: Test-host2 properties

Host 2 Model	Dell Inspiron 15 7000 Gaming
CPU(s) number	8 (4 cores per socket, 2 threads per core)
CPU model	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
NUMA node	1
Virtualization	VT-x
Memory	16 GB
Network adaptor	RTL8111/8168/8411 Driver r8169 Driver Version 2.3LK-NAPI
Network capacity	1Gbit/s
OS Relese	Ubuntu 16.04 LTS
Kernel version	4.15.0-66-generic x86_64

Table 1.3: Gigabit ethernet switch specifications

Switch Model	CNet CGS-800
Type	Gigabit Ethernet
Capacity	8 ports x 1Gbit/s

Table 1.4: Gateway device specifications

Gateway Model	Huawei HG658 V2
Capacity	4 ports x 100 Mbit/s ADSL2 VDSL2

4.2 IMPLEMENTATIONS

Container multi-host networking has investigated within five different scenarios which are Docker Default Overlay Driver with Etcd, Flannel, Weave, OVN and Calico which are recorded at Table 4.5.

Table 1.5: Container multi-host networking scenarios

Scenario1	Docker Default Overlay Driver with Etcd
Scenario2	Flannel
Scenario3	Weave
Scenario4	OVN
Scenario5	Calico

Each overlay scenario has been deployed in the same environment and tested separately. The configuration of container networking solutions are kept default as much as possible. Interfaces in the implementations are used with their default MTU sizes. Regarding with the key value stores, Etcd and Consul are prefered according to scenario. As key value stores are only used for service registration and storing the network state, it is assumed that KVS's does not affect the performance. Etcd is used with Libnetwork, Flannel and Calico. Open vSwitch is deployed with Consul as implementation is more clear because of documentation. Weave does not require any KVS as it utilizes an eventually consistent distributed configuration model (CRDT).

Table 1.6: Implemented software versions

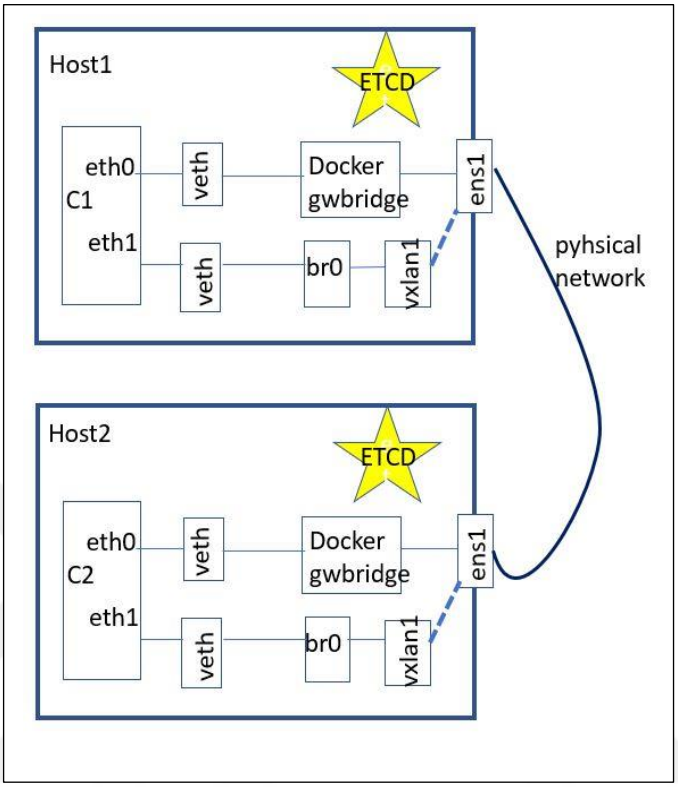
Docker	18.09.7, build 2d0083d
Calico	3.2.8
Etcd	3.3.9
Flannel	0.8.0
Weave	2.5.2
Open vSwitch	2.12.0
Consul	1.0.6

The versions of applications which are used in the implementations are presented at Table 4.6. Same docker version 18.09.7, build 2d0083d, same Etcd version 3.3.9 are used during the implementation of scenarios. Flannel version 0.8.0, Weave version 2.5.2, Calico version 3.2.8 and Open vSwitch version 2.12.0 are deployed in the setups. Consul global cluster store with version 1.0.6 is deployed with Open vSwitch unlike Flannel, Calico and Libnetwork implementations.

4.2.1 Case Study One

Docker build-in overlay networking driver is used by default if Docker runs in Swarm mode. However, this scenario is not running Docker in Swarm mode, instead Docker engine is used with Etcd global cluster store to enable the multi-host overlay networking. So, Etcd cluster can be considered as management plane of multi-host networking. Docker default overlay with Etcd setup is presented at Figure 4.2. In data plane, container traffic is encapsulated in VXLAN headers which are allowed to transmit on physical Layer 2 or Layer 3 network. Docker Engine forges the required network substructure in each hosts during the creation of overlay network. Overlay network is created in one host and it is synchronized to other hosts through the Etcd. A linux bridge is generated for per overlay with its consolidated VXLAN interfaces on each hosts. Docker engine wisely launches the overlay networks on the hosts when a container is devised to attach to this network by avoiding to distribute the overlay networks even if there is no existing container joined to it.

Figure 1.10: Docker default overlay with Etcd test scenario projection



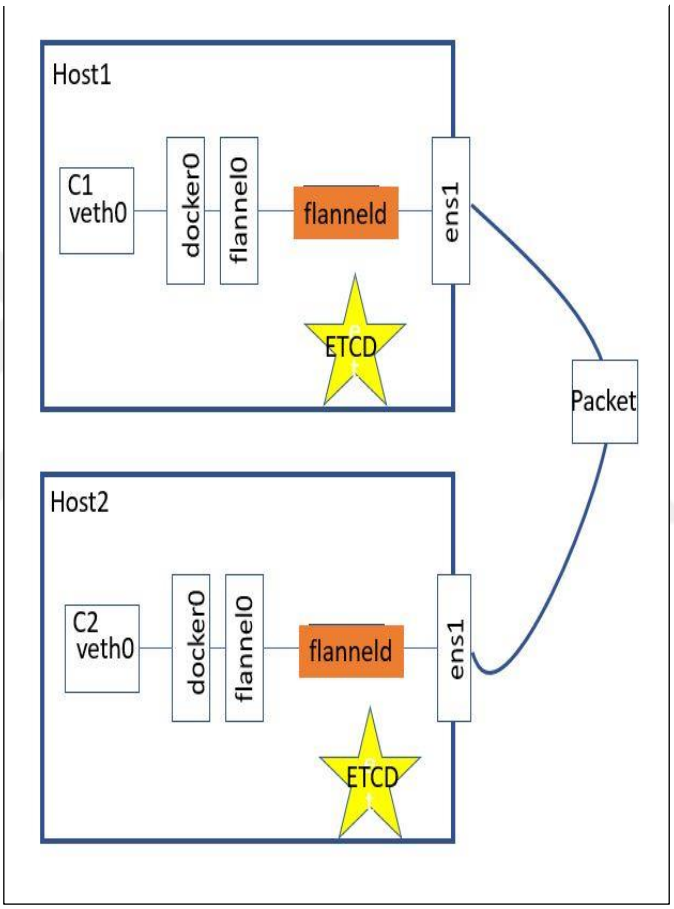
Each container keeps two interfaces on itself, one is used to connect docker_gwbridge and other is utilized for overlay network. First of all, Etcd cluster is set up on the hosts, then Docker engine is started with Etcd cluster configuration to store the clusterwise information into it. New overlay network is created by the help of docker overlay driver in one of the hosts. Overlay interface came up with default MTU size 1450 bytes. This network is automatically added to the other host through the global cluster store.

4.2.2 Case Study Two

Flannel operates as an agent which is liable from allocating subnets from appointed address space. All clusterwise information such as allocated subnets, host ip addresses, network configuration is stored in Etcd. Packet forwarding is performed by backends. VXLAN backend which is similar to docker overlay network is used in this scenario. Container networking implementation over two hosts with Flannel is limned at Figure 4.3. Initially, Etcd cluster is set up on each hosts. JSON format configuration file

which includes backend type and network information together with subnet interval is inserted to Etcd cluster before executing Flanneld agents. Then, Flanneld agents are started on each hosts. Afterwards, Docker daemon has started with Flannel network configuration and finally iptables forwarding policy changed from DROP to ACCEPT as after Docker 1.13 default forwarding policy is drop. Flannel overlay interface occured with 1450 bytes MTU sizes.

Figure 1.11: Flannel test scenario projection



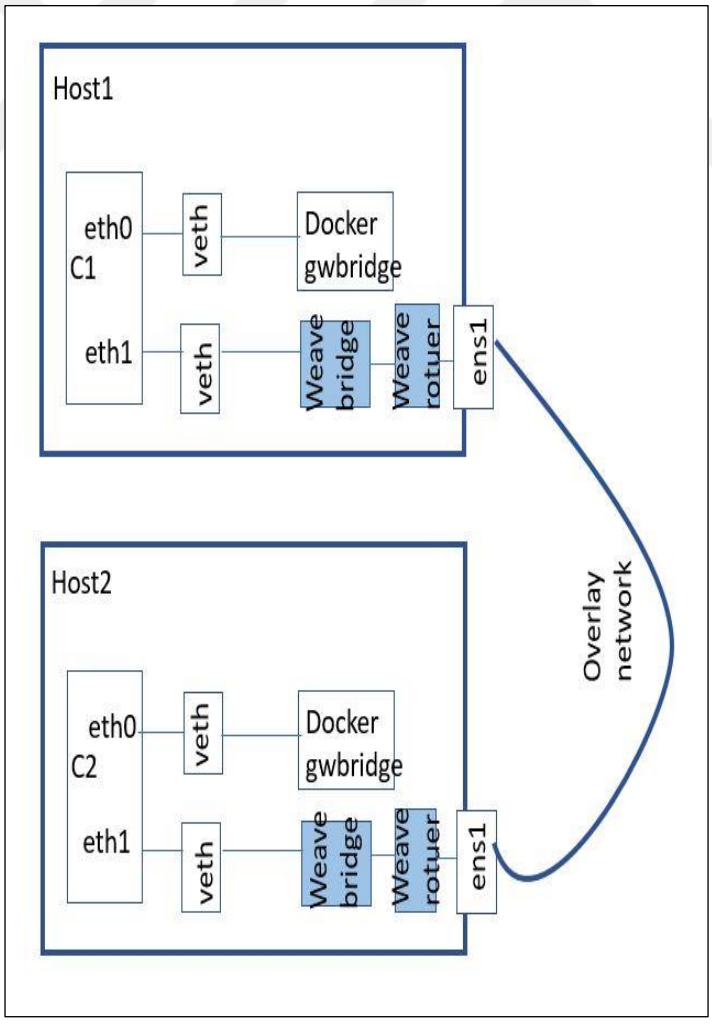
4.2.3 Case Study Three

Weave creates a Weave bridge as well as a Weave router in the host machine. Weave router establishes both TCP and UDP connection across hosts to other Weave routers. TCP connection is used for discovery and protocol related exchange. UDP is used for data encapsulation. Encryption can be done if needed. The Weave bridge is configured to sniff the packets that needs to be sent across hosts and redirect to the Weave router. For local switching, weave router is not used.

Weave is the only scenario which does not require any cluster store as it maintain conflict-free replicated data type (CRDT) to store all cluster information within Weavemesh driver. Weavemesh driver and fastdp mode are default modes during Weave launch. This scenario is operated with default modes utilizing fastdp and Weavemesh driver. Weave is used with it's default MTU size which is 1376 bytes.

Weave cluster networking setup used in this research is depicted at Figure 4.4. After installing Weave, it's lauched on host1. It runs the Weave containers on the host. Later, peer connection is created between hosts while launching the Weave on host 2 by telling host 2 is peer with host 1. Weave set up is so easy that after launching Weave on both hosts within peer information, weave containers run on the hosts and a network named weave is displayed in the Docker networks.

Figure 1.12: Weave Test Scenario Projection

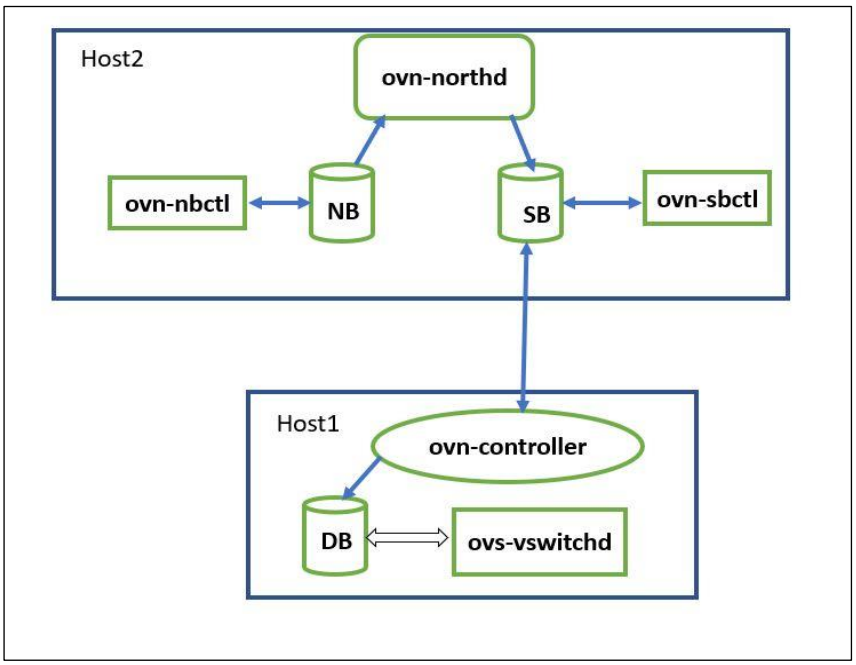


4.2.4 Case Study Four

Docker overlay with Openvswitch which is called Open Virtual Network (OVN) provides communication between containers in different hosts. There are generally to methods to connect containers with Open vSwitch. First way is connecting OVS bridge with default docker0 and second way is attaching containers with ovs bridge through veth pair directly. In this implementation, first method is chosen because of it's simplicity. In the second method, if the default docker0 bridge is not applied, more steps are required to attach containers to OVS such as network namespace and veth pairs creation, linking veth to containers etc. OVS has two working modes which are standalone and secure. Default mode is standalone and it operates as learning switch. Secure mode depends on a Openflow controller to insert flows' rules. OVS is operated in default mode in this study.

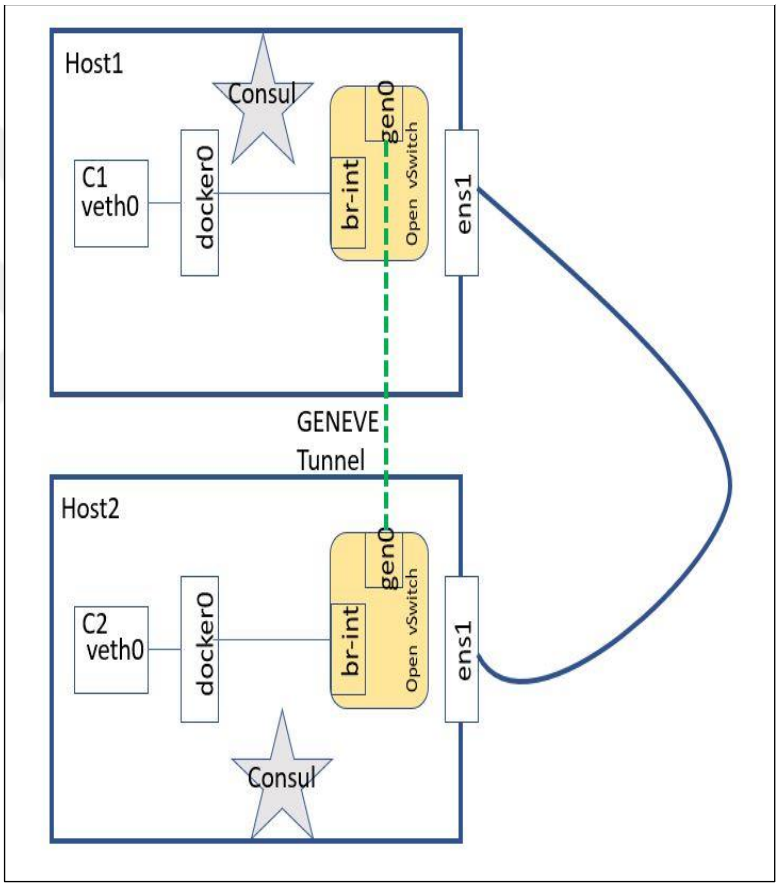
Docker cluster networking utilizing Open vSwitch which is implemented in this scenario has limned at Figure 4.6. Initially, OVS kernel module has builded, installed and loaded in each host machines. Open vSwitch architecture consists from several daemons which are simple processes. It includes shell script called ovs-ctl which is very helpful to automate the starting and stopping tasks of ovs-vswitchd and ovsdb-server. After OVS is installed, a database is configured for ovsdb-server, then ovsdb-server has started by ovs-ctl utility. After the database initialized, ovs-vswitchd is started on each hosts.

Figure 1.13: OVN building blocks



The next step is preparing docker for OVN. Docker has to be started with a global cluster store for multi-host networking. Consul is used as key-value store in this implementation. Consul has started on each host machines, then Docker daemon started with Consul cluster store. For the time being, OVN's integration with Docker has two modes named underlay and overlay. Overlay mode is used to create the logical network between containers running on multiple hosts while underlay is prefered for different purposes such as OpenStack implementations.

Figure 1.14: OVN test scenario projection



In host 2, ovn-northd daemon which commute networking intent kept in the OVN_Northbound database to logical flows in OVN_Southbound database has initiated [32]. OVN components ovn-northd, ovn-controller, together with SB and NB databases which are used in it's working mechanism and relationships between them are demonstrated at Figure 4.5. On each host, ovn-remote, ovn-nb, ovn-encap-ip and ovn-encap-type have configured by the help of ovs-vsctl utility. ovn-encap-type is defined as Geneve. System-id's are set in each hosts as each Open vSwitch instance in OVN

implementation required a persistent and original descriptive. Afterwards, ovn-controller has started on the hosts before starting OVS network driver which utilize Python Flask module. Finally, logical switch with OVS driver has forged within a given specific subnet in one of the hosts. Thence, a network which utilize OVS driver has displayed in docker networks in both hosts. OVS uses totally 1500 bytes MTU sizes for the overlay interfaces in default by allocating 58 bytes for encapsulation header.

4.2.5 Case Study Five

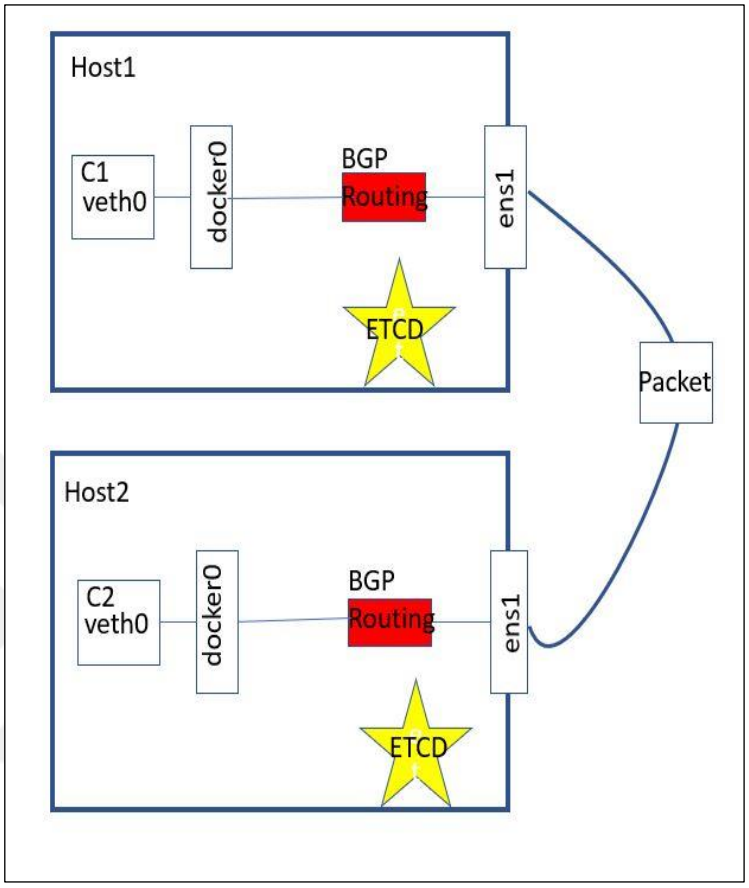
Calico assures pure Layer 3 implementation to come through a simpler, better performance, higher scaling and more efficient multi-host networking. This approach abstains the packet encapsulation consolidated with the Layer 2 solution by simplifying the observation and abates the transport overhead. Calico is implemented with BGP protocol for routing combined with a pure IP network.

Containers cluster networking with Calico setup is limned at Figure 4.7. Initially, Etcd key-value store has configured on docker hosts and Docker daemon has started in order to wield Etcd. A configuration file is created in order to configure calicoctl to utilize the Etcd datastore on host 1 before starting Calico. When Calico is started, Calico node container appears on each hosts. Afterwards, node resource configuration file which includes asNumber and ip address are prepared and applied on cluster hosts. A configuration file for ip tool settings is prepared and applied.

Parameter --nat-outgoing is inserted in ip pool configuration in order to allow containers to reach external network communication. New docker network is created with Calico driver by configuring with subnet. A configuration file for profile setting is prepared and applied to allow the actions for all ingress and egress ports in the Calico networks. Otherwise, containers may have communication problems. Finally, BGP Peer configuration file is applied and BGP Peer establishment status became active. Calico uses totally 1500 bytes MTU on the interfaces which communicates clusterwide. Normally, BGP peer status is ensured by Calico router (BIRD). If the topology is complicated with large quantities of hosts, it is suggested to deploy Calico-

route reflector. As this cluster inludes only two hosts, calico-route reflector is not utilized.

Figure 1.15: Calico test scenario projection



4.3 RESEARCH METHODOLOGY

This chapter explains the applied methology of research in this thesis. The thesis' goal is finding out the behaviour under high traffic load for several multi hosts container networking solutions within different applications, protocols and performance indicators. Proper experimental research and analysis technique is implemented in order to reach this goal. Selected tehnique should give concrete outcomes to compare the several scenarios and differentiation between basic and under load performance. Genereally, there are two applied methods in modern research studies which are qualitative and quantitative analysis.

a. Quantitative analysis:

It is a research technique which covers statictical data analysis and even-handed evaluations. Quantitative analysis requires to collect data from several resources which are taken in the same conditions. This data is analyzed and generalized to propound a specific notion and statictical or mathematical models [33].

b. Qualitative Analysis:

This research technique applies to the subjective determination as the data is unmeasurable such as cultural and social connections, reflections from natural occurrences, industrial cycles etc. This kind of scientific research can not be measured objectively, it can only be observed and classified [33].

This study applies the Quantitative analysis as concrete results are educed from numerical outputs through the experimentations. Experimentation metrics are selected as throughput, latency, response times, jitter, average response times, lost datagrams and retrasmitted TCP segments to find out the best performing networking solution for high traffic scenarios. All experiments are performed on actual systems and each scenario has been builded separately in the same conditions in order to model the behaviour of container networking solution under different traffic loads.

There are two models which are mathematical and physical are available for research's [34].

Physical Modelling is a realistic modelling which is duplication of real system. It is assumed that results executed on physical model and real environments will be similar and real environment's behaviour can be educed from the physical modelling results. Therefore, physical systems keep all the specifications of real implementations.

Mathematical Modelling is the statistical and mathematical projection of real systems. Although, it is used by several disciplines, it is generally used to explain the attitude of systems by considering the impacts of several components.

This research is based on physical model as all the experiments are performed on physical environment which consists two computers and one switch in real time. No virtual simulation is practiced.

After performing the experimentation, several outcomes are gathered. Various measurement indicators' results are displayed in different numerical ranges. However, performance and reliability evaluation need to be performed by courtesy of different tests which has similar intentions. So, It is required to convert the results of all tests in to the same value range in order to evaluate the group of features together. Normalization method is selected for this purpose as it makes every datapoint have the same scale and each tests become equally important in overall. There are 3 common normalization methods which are used for data equal scaling: decimal scaling, min-max normalization, z-score normalization [58].

Z-score normalization is selected for data analysis in this thesis as it is the best method which prevents outlier problems. The formula for Z-score normalization is given at (4.1) [58].

$$\mathrm{Z-score\ normalization} = \frac{\mathrm{value}-\mu}{\sigma} \tag{4.1}$$

In this formula, σ is the standart deviation and μ is the mean value of the property. Accordig to this equation, if the value is mean of all values, it will be normalized to 0; if it is higher than mean, it will be positive or lower that mean, it will be negative. Normally, size of negative and positive numbers are decided by the standart deviation. In this study, maximum results of test is placed to 5, minium value is set to -5 and mean value is located at 0, while other values are placed according to their standart deviation by rolling the decimal numbers to integers. Finally, results' tables are constituted for all the tests which includes values between -5 to 5. For positive features, 5 express the best performing feature and -5 denotes worst performing feature. For negative features, sign of the number has changed to its negative to properly evaluate its contribution. During the evaluation of features which consists from several test results, normalized results are summed up by giving equal importance to each sub features. Normalized results after measurements are put into the Annex A.

4.4 MEASUREMENT TOOLS

Table 1.7: Test tools software versions

Iperf	3.0.7
Netperf	2.7.0
Qperf	0.4.9
Apache Jmeter	3.3 r1808647
Mongo-perf	r20190405
Mongo server	3.2.0

The benchmarking tools such as Netperf, Iperf3, Qperf, Apache JMeter are used to measure the networking performance. Particularly, Mongo-perf which is a micro bechmarking tool for MongoDB is also used. Software versions of tools which are used in the measurements are took part in the Table 4.7.

Measurement tools are containerized, and measurements are executed in containers as it's mandatory to compare the container overlay performance. Dockerfiles are prepared for each measurement tools and they are included in Appendix B. Test scripts which are inserted in Appendix C are prepared for each test scenario as several tests are executed within different tuning parameters. Test cases in [31] are taken as example during the preparation of Jmeter test scenarios, while it's benefitted from mongo-perf test cases under [41] in order to prepare mongo-perf test scenarios. Each test has executed 20 times to get the more accurate results. Same test containers and same test scripts are used to provide the identical test conditions.

Measurement tools are tuned to find out the behavior of implemented solutions under different loads. Parallel streams and threads are executed to increase the load. Tuning parameters are particularly selected for each measurement tool in order to change the specific parameters such as message sizes, TCP window sizes.

In order to effectively evaluate each scenario with several performance indicators, benchmarking method is implemented as benchmarking is a standard approach for correct evaluation. Benchmarking is set of operations by executing a program intending to

discover the relative performance of tested systems. Benchmarking entails measuring and appraising computational performance relatively by considering measurement parameters together with devices, networking protocols and networks between tested systems. So, benchmarking tools has great importance in this research.

NetPerf is used to measure network performance for varied networking scenarios which is primarily used for TCP and UDP bulk data transfer measurements [35]. Netperf simply acts in server-client model by netserver (server) which is a standing process running with a specific port (default 12865) and Netperf (client) which is a CLI. Netperf client is used for control connection setup between client and server hosts. There is not a dedicated client port as it is chosen from any available port automatically [35]. Although two connections are set up between client and server, they do not interfere each other. Control connection is established by using BSD sockets as a TCP connection. Data connection is used to carry test configuration data and results between client and server by protocols based on executed tests and API's. Data connection is builded up after control connection has set up. Netperf supports tuning of several parameters regarding with the measurement parameters.

Iperf3 is also measurement tool for network performance which uses TCP and UDP data streams. It is mainly used for bandwidth calculations for IP networks. Besides, loss and jitter can be calculated with Iperf3 while Netperf can't measure them. [36] Iperf3 also operates in client server model. Unlike Netperf, Iperf3 server runs as a CLI program, need to run and stop manually for the executions. Iperf3 client set up the connection between remote server and local client and measurements are calculated according to data sent from client. If sent data type is UDP, packet loss and jitter are count up [36].

Apache Jmeter is an open source tool which is written in 100 percent Java and designed for load testing of web applications. Jmeter can be used for functional and performance tests within dynamic or static resources together with different applications, server, protocol types such as SOAP/REST Webservices, HTTP, HTTPS web applications (Java, NodeJS, PHP, ASP.NET), Database via JDBC, LDAP, SMTP(S), POP3(S) and IMAP(S) mail services, native commands or shell scripts, TCP [37].

Jmeter mainly used for load testing on server or network by examining the performance within diversified load types. It presents a multi-threading test platform in order to execute isochronous tests with many threads and disparate thread groups. Besides, several functional tests can also be performed on webservices, databases, websites. Jmeter also allows distributed load testing to simulate the real users' activities and load balancing functionality. Like browsers, Jmeter observe the responses as HTML, however it does not include the timings to the samples, at a time only one sample in a one thread is analyzed [38]. Apache Jmeter provides very exhaustive IDE which provides test plan composing, recording and debugging. Furthermore, it has correlation capability for the data extracted from several response formats such as XML, HTML, JSON [37].

Qperf ordinarily used for latency, bandwidth and CPU utilization tests between two nodes over TCP/IP or RDMA transfers [39]. Running Qperf without any arguments starts it in server mode; running it with ip address of the first server and test options starts it in client mode. Variety of tests with several parameters can be used to obtain measurements outcome. Bandwidth tests calculates the rate of transfer within different protocols such as TCP, UDP, STCP as well as RDMA at packet level or byte level. Latency tests plumbs the average transfer time of the packet for per hop [40].

Mongo-perf is a micro performance measurement tool for MongoDB. It is a different tool from Mongoperf. It is composed with Mongo shell benchrun command and calculates the throughput of commands such as insert, update, query as operations per second. Mongo-perf includes several pre-builded tests and allows execution with many threads simultaneously. It maintains a Python script named benchrun.py in order to execute the test cases [41].

MongoDB is open source, flexible document-based database which allows query and indexing the JSON-like documents. By mapping objects in the application to document model simplify the working with data. Additionally, it provides online aggregation, indexing and ad hoc queries which allows the access and analyze the data quickly. MongoDB can work in distributed architecture by providing high availability and horizontal scalability [42].

4.5 PERFORMANCE METRICS

4.5.1 Throughput

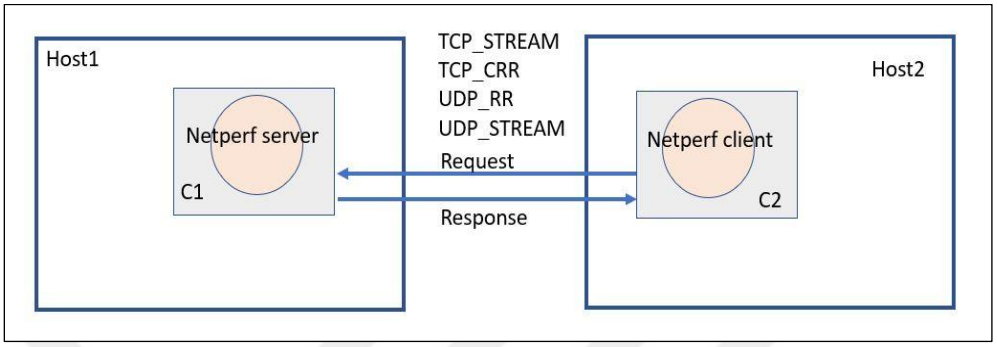
Throughput usually refers to amount of data which is transfered one place to another place in a certain time period. Because of analysing the network throughput aspect of different applications, several throughput tests are performed varying according to application and data transfer types. Throughput of basic data transfer types such as Stream (bulk data), RR (request- response), CRR (connect-request-response) are tested for TCP and UDP network protocols through the Netperf. Test parameters which are used in throughput evaluations by Netperf are given at Table 4.8. During Nertperf executions, message sizes are increased from 32 to 4096 for bulk data transfers and alongside sizespec has increased from 32 to 4096 for request response tests. Each tests executed in 10 seconds period with 20 repetition to provide 95 percent confidence interval and throughput is measured as Megabits/sec within average of results. TCP nodelay is used which disables Nagle algorithm to force packets to be transfered as soon as possible in TCP_RR and TCP_CRR tests. 10 parallel transactions are executed during TCP_RR and UDP_RR tests to observe the behaviour in higher traffic load.

Table 1.8: Netperf test parameters for throughput measurements

Test Type	TCP_STREAM	UDP_STREAM	TCP_RR	TCP_CRR	UDP_RR
Duration	10s				
Test repeated(times)	20				
Message size(bytes)	32, 64, 128, 1024, 4096				
Sizespec(bytes)			32, 64, 128, 1024, 4096		
Tcp_nodelay			yes	yes	
Receive Socket size (bytes)	87380	212992	87380	87380	212992
Send Socket Size(bytes)	16384	212992	16384	16384	212992
Burst size (number of transactions)			10		10

Netperf test environment is illustrated in the Figure 4.8 within Netperf server and Netperf client are running in different hosts and communicate over implemented scenarios.

Figure 1.16: Netperf test environment



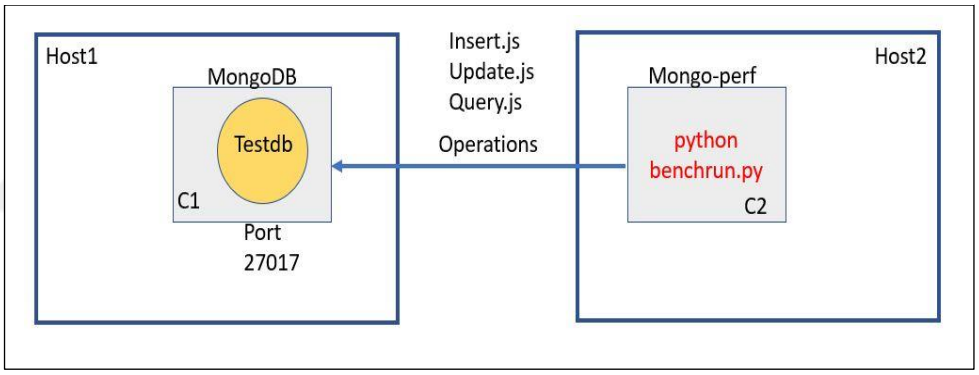
Database throughput is evaluated by utilizing MongoDB through Mongo-Perf. Mongodb benchmark tests are executed according to number of threads (1 5 50 100 200) within Mongo-Perf which uses a Python script named benchrun.py. Mongo-Perf test paramters are yielded at Table 4.9. Commonly used database operations such as insert, update and query are tested with diversified transactions for 5 seconds period. Three different insert operations are actualized. Test named Insert_Empty puts empty documents into database, while Insert_IntVector is inserting a vector of documents which has an integer field. Insert_LargeDocVector inserts a vector of documents which has a long string. Four different query operations are performed named Query_Empty, Query_NoMatch, Query_IntIDRange and Query_FindProjection. While empty query returning all documents, NoMatch scans all documents and returns no documents as query does not exits. Query_IntIDRange searches all the documents with integer _id in range of (50, 100) and finally all the threads gave the same documents. Query_FindProjection query all the documents and uses projection to give back the field y by accessing all the documents. One update test is applied named Update_FieldAtOffset which makes two multi updates on all the documents. Throughputs are measured as operations per second and tests are repeated 20 times to increase the accuracy.Results is calculated by taking the average of 20 results for each type of executions. Mongo-Perf test environment is displayed in Figure 4.9. MongoDB runs in a container in Host 1 and serves with the port 27017. Mongo-

perf tool runs in another container in Host 2 and effectuate the operations on MongoDB over the implemented networking scenarios.

Table 1.9: Mongo-Perf test parameters for throughput measurements

Test Type	Test Name	Duration(seconds)	Test repeated(times)	Threads
Insert	Empty	5	20	1, 5, 50, 100, 200
Insert	IntVector	5	20	1, 5, 50, 100, 200
Insert	LargeDocVector	5	20	1, 5, 50, 100, 200
Query	empty	5	20	1, 5, 50, 100, 200
Query	NoMatch	5	20	1, 5, 50, 100, 200
Query	IntIDRange	5	20	1, 5, 50, 100, 200
Query	FindProjection	5	20	1, 5, 50, 100, 200
Update	FieldAtOffset	5	20	1, 5, 50, 100, 200

Figure 1.17: Mongo-Perf test environment



Another throughput test is performed with Apache Jmeter. External web access benchmark test is executed with Apache Jmeter in order to improvise it to a real World scenario in a distributed way which inludes one Jmeter master/client and one slave/server. Distributed scenario is selected to gather the total effect of both container to container and container to the external web application access.

Requests which increased from 5 to 200 to display the behaviour under load are sent from master to server and server has sent the HTTP request to a certain web site which is “www.google.com”. In all other experiments communications are done on cluster networking. However this scenario, data is sent from cluster to internet. Therefore, a gateway is added to the test environment only for Jmeter experiments.

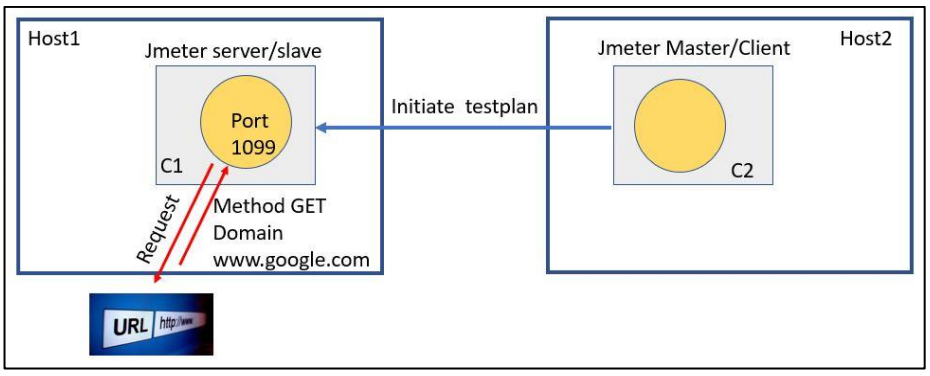
In Jmeter experiments, it is assumed that speed of internet is not changed. Jmeter test parameter used for throughput and repsonse time measurements are given at Table 4.10. Jmeter test plan tuned with 5 seconds ramp up time which the duration to ramp all the threads, and 30 seconds group duration period. Jmeter executes the thread group for this duration period. Schedular is enabled with constant timer delay which adds 300 ms delay between each request which a thread makes. Throughput is evaluated by requests/unit of time and time is computed from the start of each thread to end of final thread. Throughput is gathered within request per second.

Table 1.10: Jmeter test parameters for throughput and average response time measurements

Test repeated(times)	20
ThreadGroup.num_threads	5, 50, 100, 200
ThreadGroup.ramp_time(seconds)	5
ThreadGroup.duration(seconds)	30
HTTPSampler.domain	www.google.com
HTTPSampler.method	GET
HTTPSampler.use_keepalive	true
ConstantTimer.delay(miliseconds)	300
ThreadGroup.scheduler	true
ThreadGroup.delayedStart	true

Jmeter test environment is depicted in Figure 4.10. Jmeter server executes the test plan which is initiated from Jmeter client. Jmeter server listens for RMI connection on default port 1099. After the test plan is initiated by client, Jmeter server sent the requests to configured HTTP domain.

Figure 1.18: Jmeter test environment



4.5.2 Latency

Latency is measured with two different tools for different data transfer types and protocols. Basic latency comparison is performed between different networking solutions and multi threads executions are not applied. Qperf test parameters for latency measurements are represented at Table 4.11. Basic latency tests are performed with Qperf tool for TCP, UDP and SCTP protocols for 2 seconds period within 1 byte message size and 20 repetitions.

Table 1.11: Qperf test parameters for latency measurements

Test Type	TCP_Latency	UDP_Latency	SCTP_Latency
Test repeated(times)	20		
Duration(seconds)	2		
Message Size(bytes)	1		

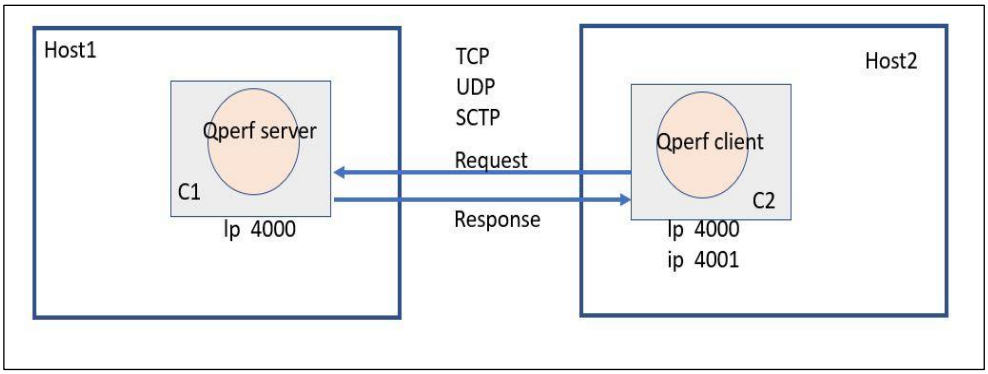
Latency with different data transfer types are realized with Netperf and results are received as mean latency. Bulk data transfer tests named TCP_STREAM and UDP_STREAM are executed with message sizes (32, 64, 128, 1024, 4096) bytes. Request response tests are actualized with 1 bytes mesage size for 10 seconds period and 20 repetitions which are given at Table 4.12. Results are gathered as microseconds for latency both for Netperf and Qperf.

Table 1.12: Netperf test parameters for latency measurements

Test Type	TCP_STREAM	UDP_STREAM	TCP_RR	UDP_RR
Duration(seconds)	10			
Test repeated(times)	20			
Message size(bytes)	32, 64, 128, 1024, 4096		1	
Latency parameter	mean_latency			

Qperf test environment is given in the Figure 4.11. Qperf server runs in a container in Host 1 whereas Qperf client runs in another container in Host 2. Both client and server uses the listen-port 4000 as it should be same, used for synchronization, and ip port which runs the socket tests set as 4001 on the client.

Figure 1.19: Qperf test environment



4.5.3 Bandwidth

Iperf3 and Qperf are used to obtain bandwidth tests. Qperf is utilized to unveil fundamental bandwidth with respect to TCP, UDP, STCP protocols without tuning the parameters. Qperf bandwidth tests are practiced with default values 64 KB message sizes for TCP, 32 KB message sizes for UDP and SCTP within 2 seconds time period and 20 repetitions which are clearly indicated at Table 4.13.

Table 1.13: Qperf test parameters for bandwidth

Test Type	TCP_Bw	UDP_Bw	SCTP_Bw
Test repeated(times)	20		
Duration(seconds)	2		
Message Size(bytes)	65536	32768	32768

Iperf3 is also operated for bandwidth tests with two aspects. Initially, it is executed to observe the load effects on TCP and UDP bandwidth. Therefore, parallel streams (5,10, 30 ,100) are executed simultaneously for 10 seconds time period. Iperf3 test parameters for parallel executions are recorded at Table 4.15.

On the other hand, it is used to measure the TCP window size effects on bandwidth with window sizes (8, 16 ,32, 64, 128) which are noted down at Table 4.14. Message size are not tuned for the tests; 128 KB message sizes for TCP and 8KB message size for UDP are wielded. Bandwidth results are gathered in Megabits/sec both for Qperf and Iperf3.

Table 1.14: Iperf3 test parameters for bandwidth and retransmitted TCP segments according to TCP window size

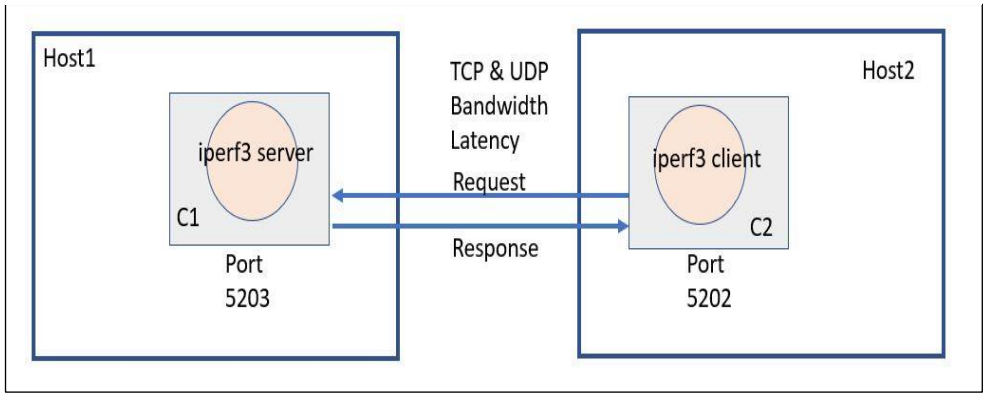
Test Type	TCP
Duration(seconds)	10
Test repeated(times)	20
Message Size(bytes)	131072
TCP MSS(default)	1448
TCP window size	8, 16, 32, 64, 128

Table 1.15: Iperf3 test parameters for bandwidth, retransmitted TCP segments, jitter and lost datagram with multithreading

Test Type	TCP	UDP
Duration(seconds)	10	
Test repeated(times)	20	
Message Size(bytes)	131072	8192
TCP MSS(default)	1448	
Paralel Streams	5, 10, 30, 100	1, 5, 10, 30, 100

Iperf3 test environment consists from Iperf3 server, serves on port 5203 in Host 1 and Iperf3 client in Host 2 which is limned at Figure 4.12.

Figure 1.20: Iperf3 test environment



4.5.4 Response Times

TCP sets up connections via three-way handshake. The client sends a request for connection, the server replies and the client acknowledges the response. Then, data has sent after getting the acknowledgement. As client need to wait for server before sending the data, minimum time of two RTT's are already passed before data transmission. Response time for computer system and web sites has different significations. For computer system, it's time which system react to a request mean while it's the respond time for a users's request for websites [57]. In this thesis, response times are measured for website. Response time meaurements are gathered from Apache Jmeter executions for throughput.

While executing the test plan with several threads, response times are also calculated in the results in miliseconds together with throughput. So, same tuning parameters which are given at Table 4.10 are valid for response times and throuhput executions of Jmeter.

4.5.5 Jitter

Jitter is the alteration in periodicity of a signal or event from it's particular frequency. Packets transfered on the network may have delays because of reasons routed separately or waited in the queue of network devices. Those cases are impacted by packet network congestion and packet loss. So delay discrepancy of individual packets are implied as jitter which is an reliability indicator.

UDP Jitter values of different netwoking implementations under load is fetched from Iperf3 paralel stream executions for bandwidth. Same tuning parameters of Iperf3 multi thread bandwidth executions which are indicated at Table 4.15 are effective for UDP jitter. Jitter results are calculated in miliseconds.

4.5.6 Retrasmitted TCP Segments

Retransmissions are required for lost packets. This process take more time and make the application slower. Retransmitted TCP segments affect the application performance considerably depending on the how frequent it is happening and how fast they are recovered. That's why it is the reliability indicator. Number of retrasmitted TCP segments are also come up with Iperf3 bandwidth executions within the parametes given at Table 4.14 and Table 4.15 and no additional experiment is required for it. In this study, retrasmitted TCP segments are find out from two different perspective which are TCP window size and load.

4.5.7 Lost Datagram

For UDP, if the packet loss are detected in receiver side, lost packets are sent again. However, this situation detracts the peformance because of resending takes more time. So, UDP lost datagram is a significant reliability indicator that it is also fetched from Iperf3 experiments within the test parameters given at Table 4.15. Results are taken as percentage of lost datagrams from all transfers.

5. PERFORMANCE RESULTS

5.1 SINGLE THREAD

The aim of all single thread performance evaluations are obtaining the normal performance and reliability attitudes in ordinary situations without load. Besides, container and native baremetal host throughput and latency performance comparisons are done in this section.

5.1.1 Throughput

5.1.1.1 Throughput according to packet sizes

In TCP stream throughput results which is demonstrated at Figure 5.1 a, baremetal host gave the highest throughput for all packet sizes as expected because of all container networking solutions have packet routing or encapsulation overheads. OVN gave the highest throughput within 32 bytes packet size and 64 bytes packet. When the packet size has increased to 128 bytes and more, Calico gave the highest throughput among all solutions.

According to TCP CRR throughput results limned at Figure 5.1 c, baremetal host gave the maximum throughput for all packet sizes except 4096 bytes. Within 4096 bytes packet size, an unusual behaviour has observed that container networking solution Calico exceeded the throughput of baremetal host. This situation is unexpected as Calico should have throughput loss because of routing. However this situation has observed in previous studies [3], [5] and explained with the hardware offloading features effect of Docker. Except the GRO, all other offloading features were disabled in baremetal host in this study and it is checked that Docker enabled all hardware offloading features by default. It is assumed that docker hardware offloading utilization boosted the Calico's TCP throughput within 4096 bytes message sizes. This result also supporting the offloading impact in container networking solutions.

However, the solutions can not utilize from offloading at the same level the reason why NIC of hosts can not perform offloading for Geneve, while they have capable of VXLAN offloading. Generally enabling TSO, GSO (tx-udp_tnl-segmentation for

VxLAN encapsulation) on transmitter and GRO on receiver increases TCP throughput. So, the OVN solution which uses Geneve could not utilize the encapsulation offloading benefits. It is supposed that this is the one of reason for OVN to give minimum throughput in all packet sizes. Additionally, solutions MTU sizes are different although they all squeeze into Ethernet 1500 bytes MTU. While Calico does not have any loss as it does not require any encapsulation, Libnetwork and Flannel lose 50 bytes because of VXLAN header, OVN lose 58 bytes because of Geneve encapsulation. Weave has the solution which has smallest MTU sizes with 1376 bytes. Having different MTU capabilities also can impact on the throughput.

UDP stream throughput is displayed at Figure 5.1 b. From the Docker's hardware offload usage perspective, enabling Tx checksumming on transmitter and GRO on receiver increase UDP throughput. However, from the host side, udp-fragmentation-offload is not supported by NIC of hosts that used in this study.

Additionally, in the previous studies [3] and [5], it is also observed that hardware offloading does not effect UDP traffic as much as TCP. Supporting this finding that baremetal host UDP stream throughput is not surpassed by none of the container networking solution for all packet sizes. OVN and Libnetwork provided the maximum throughput for small messages sizes, while Libnetwork also gave highest throughput for large message sizes for UDP bulk data transfers. Besides, Weave yielded the minimum throughput for small packet sizes which are less than 1024 bytes, Calico provided least throughput for packet sizes which are higher than 1024 bytes among tested container overlay solutions. All the throughput measurement results gathered by TCP stream, UDP stream and TCP CRR are served at Table 5.1.

Figure 1.21: Throughput comparison of multi-host container networking solutions according to packet size (bytes). (a) TCP_STREAM, (b) UDP_STREAM, (c) TCP_CRR.

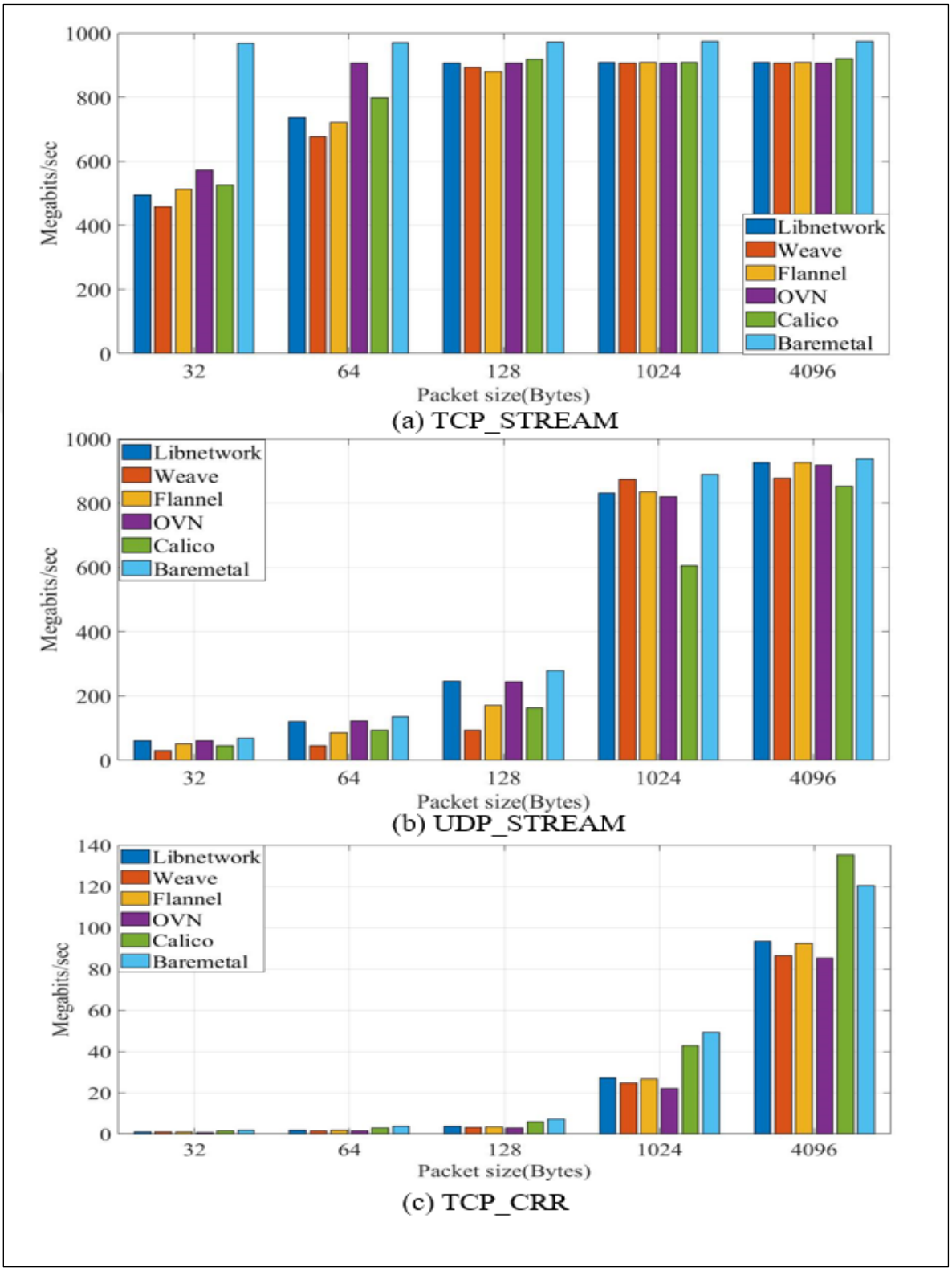


Table 1.16: Throughput (Mbits/sec) comparison according to packet size (bytes)

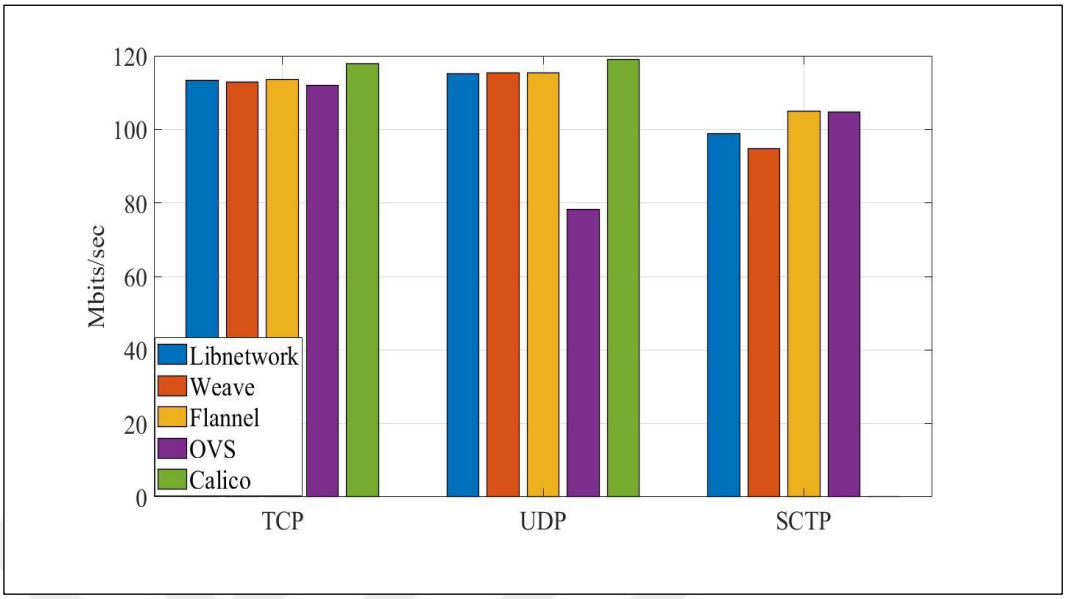
Test Type	Packet size	Baremetal	Libnetwork	Weave	Flannel	OVN	Calico
TCP_STREAM	32	967,735	494,937	458,339	511,763	571,509	526,217
TCP_STREAM	64	969,706	736,351	675,808	720,261	905,736	798,19
TCP_STREAM	128	972,152	905,995	892,225	879,802	906,201	918,484
TCP_STREAM	1024	974,444	908,64	905,898	908,482	905,516	908,947
TCP_STREAM	4096	974,704	908,592	906,895	908,121	905,522	919,758
UDP_STREAM	32	69,4556	60,877	29,1725	50,3495	60,8795	45,0635
UDP_STREAM	64	135,566	121,506	45,942	85,6825	121,797	94,472
UDP_STREAM	128	279,091	247,006	94,113	170,519	243,994	163,181
UDP_STREAM	1024	889,139	831,656	872,942	835,818	819,595	604,831
UDP_STREAM	4096	936,4509	924,968	878,172	925,505	917,599	852,876
TCP_CRR	32	1,822	0,8965	0,8125	0,871	0,7065	1,525
TCP_CRR	64	3,5985	1,7785	1,585	1,718	1,41	2,948
TCP_CRR	128	7,1565	3,5655	3,215	3,474	2,8125	5,8555
TCP_CRR	1024	49,2565	27,2235	24,7325	26,7525	22,0465	42,8505
TCP_CRR	4096	120,337	93,4565	86,3065	92,3195	85,293	135,2

5.1.2 Bandwidth

5.1.2.1 Bandwidth according to protocol types

In bandwidth comparison of container networking solutions which is showed at Figure 5.2, channel capacity has been calculated with Qperf for different protocols within default message sizes which are 64 KB for TCP; 32 KB for UDP and SCTP. Calico provided the highest bandwidth amongst all container networking solutions both for TCP and UDP. It is observed during the executions that even OVN UDP sent bandwidth is as big as other container networking solutions, it's receive bandwidth is quite small. As Figure 5.2 diplays the receive bandwidth of UDP, it's the lowest among all solutions.

Figure 1.22: Protocols bandwidth comparison for TCP, UDP and SCTP



From the host side, udp-fragmentation-offload and tx-checksum-sctp are disabled and could not be used by Docker. Physical network interfaces of test environments also do not recognize Geneve. So, OVN is the least benefitted solution from hardware offloading. It may cause UDP bandwidth loss. Another reason could be OVS working mechanism which operates as a learning switch which may have first hit misses together with having long data path for unkown flows. Flannel and OVN gave the highest bandwidth for SCTP while Weave was giving the minimum bandwidth.

Calico does not provide any result as SCTP because of Calico's SCTP support has arrived with version 3.3 and this test utilized version 3.2.8 which does not support SCTP. Protocols bandwidth comparison results are given at Table 5.2.

Table 1.17: TCP/UDP/SCTP protocols bandwidth(Mbits/sec) comparison

Protocols	Libnetwork	Weave	Flannel	OVN	Calico
TCP	113,4	112,85	113,5	111,917	117,9
UDP	115,158	115,35	115,4	115,25	119,05
SCTP	98,92	94,715	104,895	104,79	

5.1.2.2 Bandwidth according to TCP window size

TCP bandwidth according to window size is depicted at Figure 5.2, while all the results are diplayed in the Table 5.3. Calico gave the maximum bandwidth according the various window sizes changing from 8 to 128. OVN bandwidth was improved by window size increment ultimately. Although, direct proportional relation displayed between TCP window size and bandwidth for the OVN and Calico; Libnetwork, Weave and Flannel bandwidth's reached their maximum values at 32KB window size then their bandwidth's has deteriorated in 64KB and enhanced again.

Figure 1.23: TCP bandwidth according to window size within message size 128 KB.

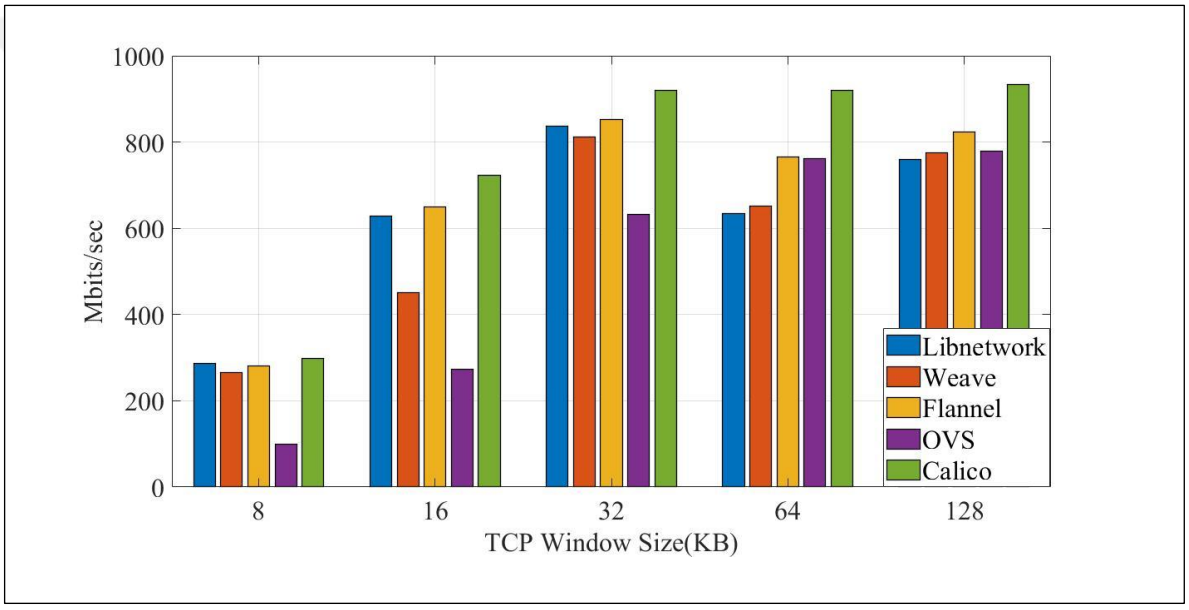


Table 1.18: TCP bandwidth outputs (Mbits/sec) acccording to window size(KB)

TCP Window Size	Libnetwork	Weave	Flannel	OVN	Calico
8	285,85	265,7	279,95	99,85	297,4
16	628,55	450,6	650,222	272,8	723,3
32	836,4	811,15	852,65	631,9	919,75
64	634,85	651,2	765,65	761,5	920,3
128	759,7	775,7	823,15	778,65	932,6

5.1.3 Latency

5.1.3.1 Latency according to packet size

TCP latency has relation with TCP thoughput. Baremetal host gives the maximum TCP bulk data thoughput, it is expected to have the minumum latency. According to the TCP stream latency results which are shown at Figure 5.4 a, it is verified that native host has the minumum latency as it is expected. All container networking solutions give quite similar results. Calico had the minumum latency for packet sizes larger than 64 KB, while OVN was giving the minimum latency for packet sizes smaller than 64 KB.

According UDP stream latency test results which are displayed at Figure 5.4 b, OVN had the minimum latency among container networking solutions. Flannel had the worst results with highest latency which is similar to [6] together with the Weave which had second rank in high latency. Baremetal host had the minimum latency as expected as it does not have any additional delay because of encapsulation, no additional hops or routing. All the latency results belong to single thread bulk data transfers are written down at the Table 5.4.

Figure 1.24: Latency comparison according to packet size (bytes).(a) TCP_STREAM, (b) UDP_STREAM.

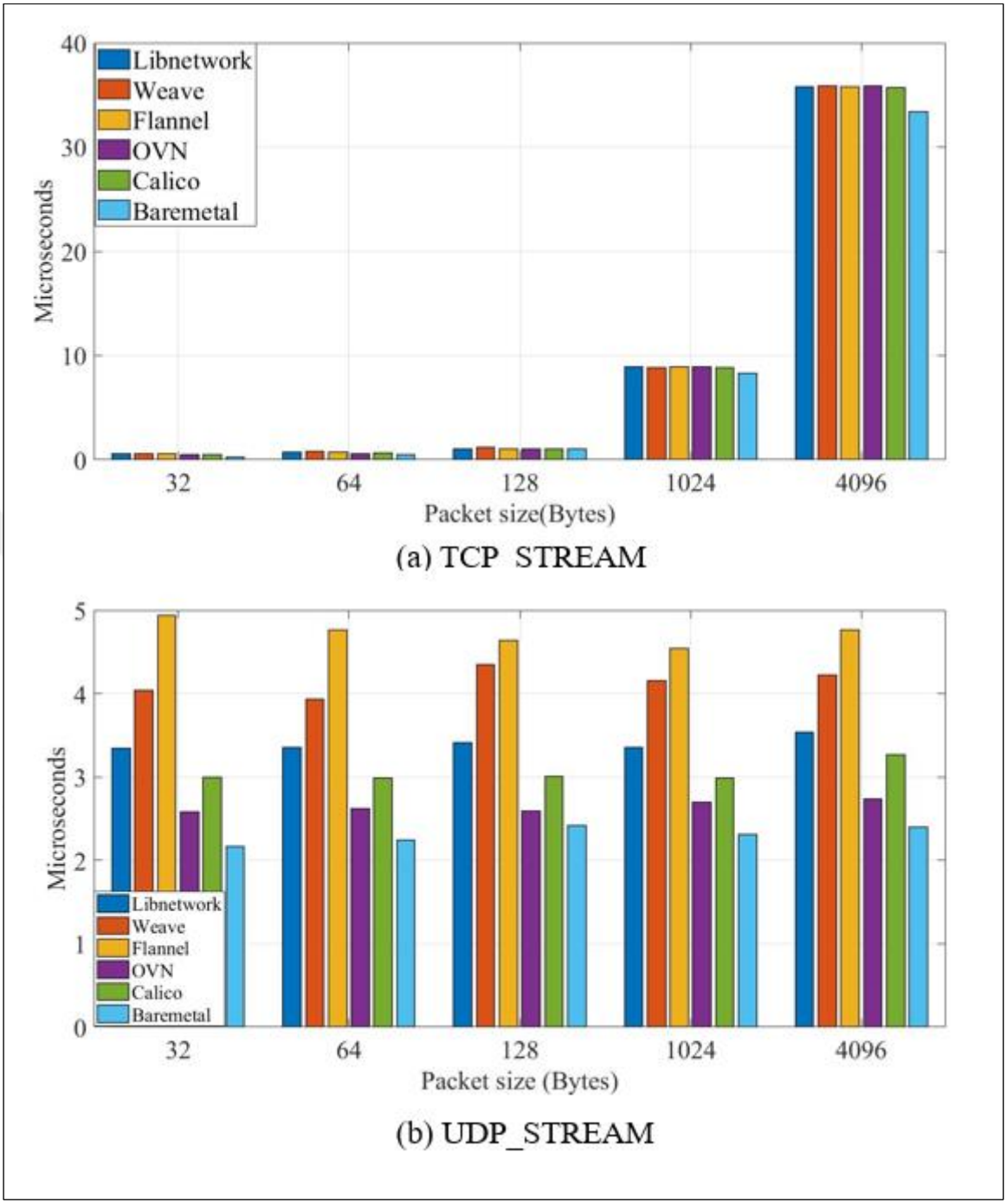


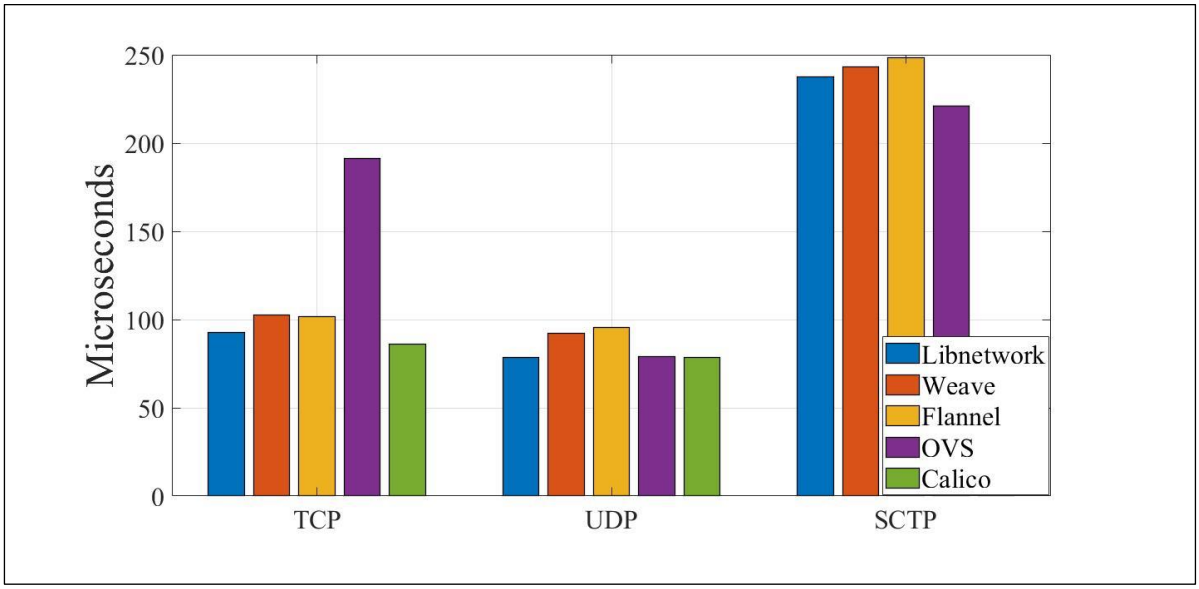
Table 1.19: Latency outputs according to packet size (microseconds) for bulk data transfers

Test Type	Packet size	Baremetal	Libnetwork	Weave	Flannel	OVN	Calico
TCP_STREAM	32	0,2485	0,5505	0,6115	0,5905	0,4825	0,529
TCP_STREAM	64	0,49	0,7205	0,787	0,7285	0,58	0,6725
TCP_STREAM	128	1,0075	1,0255	1,2195	1,045	1,0515	1,0235
TCP_STREAM	1024	8,2655	8,8815	8,8515	8,891	8,9145	8,8135
TCP_STREAM	4096	33,434	35,821	35,8615	35,821	35,905	35,7035
UDP_STREAM	32	2,163	3,3465	4,0395	4,9345	2,5775	2,993
UDP_STREAM	64	2,247	3,354	3,9315	4,76	2,617	2,9915
UDP_STREAM	128	2,42	3,4095	4,3525	4,6405	2,59	3,0025
UDP_STREAM	1024	2,3076	3,3585	4,155	4,54	2,695	2,9875
UDP_STREAM	4096	2,4021	3,5405	4,223	4,766	2,736	3,268

5.1.3.2 Latency according to protocol types

Protocols latency comparisons which is displayed at Figure 5.5 has showed that UDP has smaller latency than TCP and SCTP as predicted for all container networking solutions, the reason why UDP does not have a control mechanism of sending packets. OVN gave the maximum latency in TCP. However in bulk data transfers, OVN did not have such salient latencies.

Figure 1.25: Protocols latency comparison for TCP, UDP, SCTP



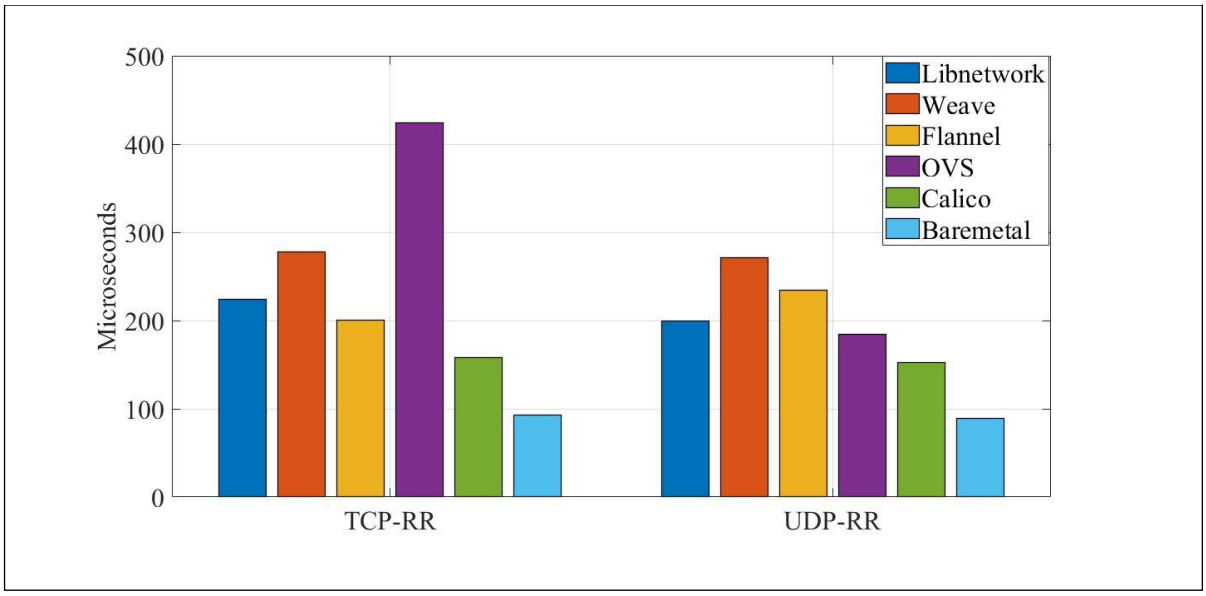
It is supposed that this is because of the OVS working mechanism is different than other overlay solutions which has two different data path passthrough kernel space and userspace together with not benefitting from hardware offloading for Geneve. However, OVN gave the minumum latency in SCTP which is compatible with SCTP bandwidth results. At the same time, tx-checksum-sctp is not supported by pyhsical layer. That puts the question mark that this situation is whether the effect of disabled offloading in SCTP or not. Calico had the minumum latency both for TCP and UDP traffic as it implements layer 3 network without any encapsulation. Flannel had the maximum SCTP and UDP latency. All the numerical results belong to protocols latency are delivered at Table 5.5.

Table 1.20: TCP/UDP/SCTP protocols latency (microseconds) outputs

Protocols	Libnetwork	Weave	Flannel	OVN	Calico
TCP	92,54	102,365	101,795	191,417	85,94
UDP	78,5789	92,095	95,435	78,8867	78,33
SCTP	237,5	243,1	248,45	221,2	

5.1.3.3 Request/response traffic latency

Figure 1.26: TCP/UDP request response latency comparison



TCP and UDP request response latency which measured with 1 bytes message sizes are demonstrated at Figure 5.6. OVN gave the worst latency for TCP-RR while it had the lowest latency after Calico for UDP request -response traffic. Baremetal host had the minumum latency as expected and Calico is the best performing solution with minimum latency for TCP and UDP request response traffic among all tested container networking solutions. Weave had the maximum latency for UDP-RR traffic. OVN surplus TCP latency is also observed in TCP-RR which is congruent with TCP latency in Figure 5.5 which rise the thought of effects about OVS learning switch mechanism and lack of Geneve offloading. In addition, latency measurements are both performed with 1 bytes message sizes which are put forthed at Figure 5.5 and 5.6. So, this results also solidify the OVN TCP latency with small message sizes. All numerical results gathered by request response traffic latency are shown at Table 5.6.

Table 1.21: Request/response trafic latency (microseconds) results

Test Type	Baremetal	Libnetwork	Weave	Flannel	OVN	Calico
TCP_RR	92,5314	224,0424	278,2536	200,303	424,5138	157,5636
UDP_RR	89,285	199,6716	271,4252	234,751	184,5398	152,684

5.1.4 Reliability

5.1.4.1 Retrasmitted TCP segments according to TCP window size

Number of retransmitted TCP segments which are accepted as an reliability indicator are calculated according to TCP window size and results are limned at Figure 5.7. Libnetwork retrasmitted maximum number of TCP segments in overall except 128KB Window size. While increasing the TCP window size, TCP packet trasnmission has increased surplusly for all tested networking solutions except Calico. Calico's TCP retransmission did not increased too much comparing to other solutions, even it's TCP retrasnmission rate has decreased within 128 KB window size while all other solutions making maximum number of retrasnmission. Libnetwork and Weave retranmission rates has tripled in transition from 32KB to 64KB window sizes, besides OVS retranmission rate has tripled in transition from 64KB to 128KB window size. Test outputs are also written down at Table 5.7.

Figure 1.27: TCP retransmitted segments comparison according to window size

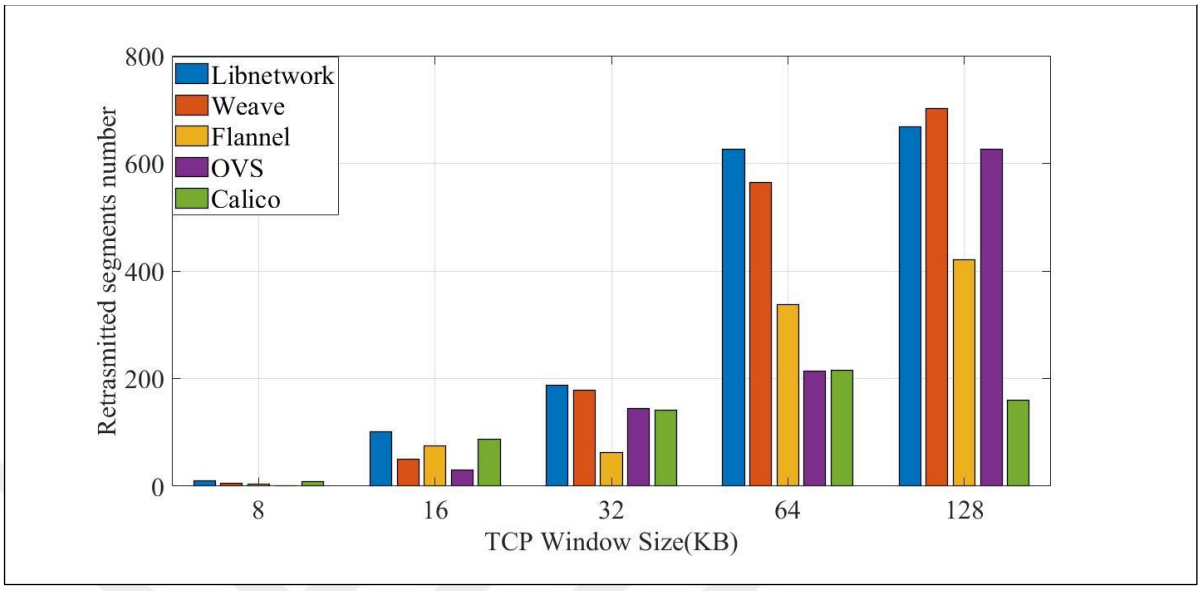


Table 1.22: Retransmitted TCP segments according to window size

TCP Window Size	Libnetwork	Weave	Flannel	OVN	Calico
8	9,3	5,8	3,8	0	8,6
16	100,7	49,85	74,8889	29,55	87,95
32	187,8	177,8	63,15	144,1	141,75
64	625,6	563,9	337,6	214,25	215,15
128	667,25	702,6	420,55	625,55	160,05

5.2 MULTI THREAD

The results which are presented in this section are gathered from multithreading executions which increase the traffic loads.

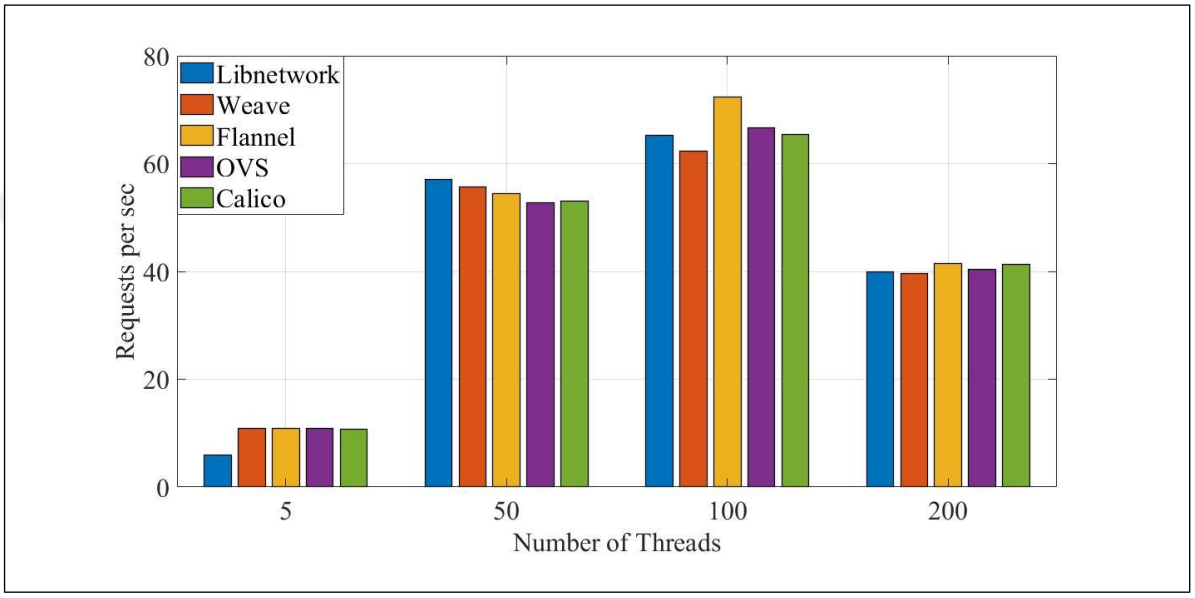
5.2.1 Throughput

5.2.1.1 Web access operations throughput

HTTP traffic average throughput with multithreading is exhibited at Figure 5.8 and all the outputs obtained through the tests are denoted at Table 5.8. While increasing the number of threads, after 100 threads throughput has decreased for all container

overlay solutions, that is supposed to be the impact of resource congestion in the host. Although networking solutions results are quite close to each other, Flannel gave the highest throughput in overall. All the solutions gave their maximum throughput with 100 threads. Flannel throuhput is 8.7, 10, 11, 16 percent more than OVN, Calico, Libnetwork and Weave respectively.

Figure 1.29: HTTP traffic average throughput with multithreading



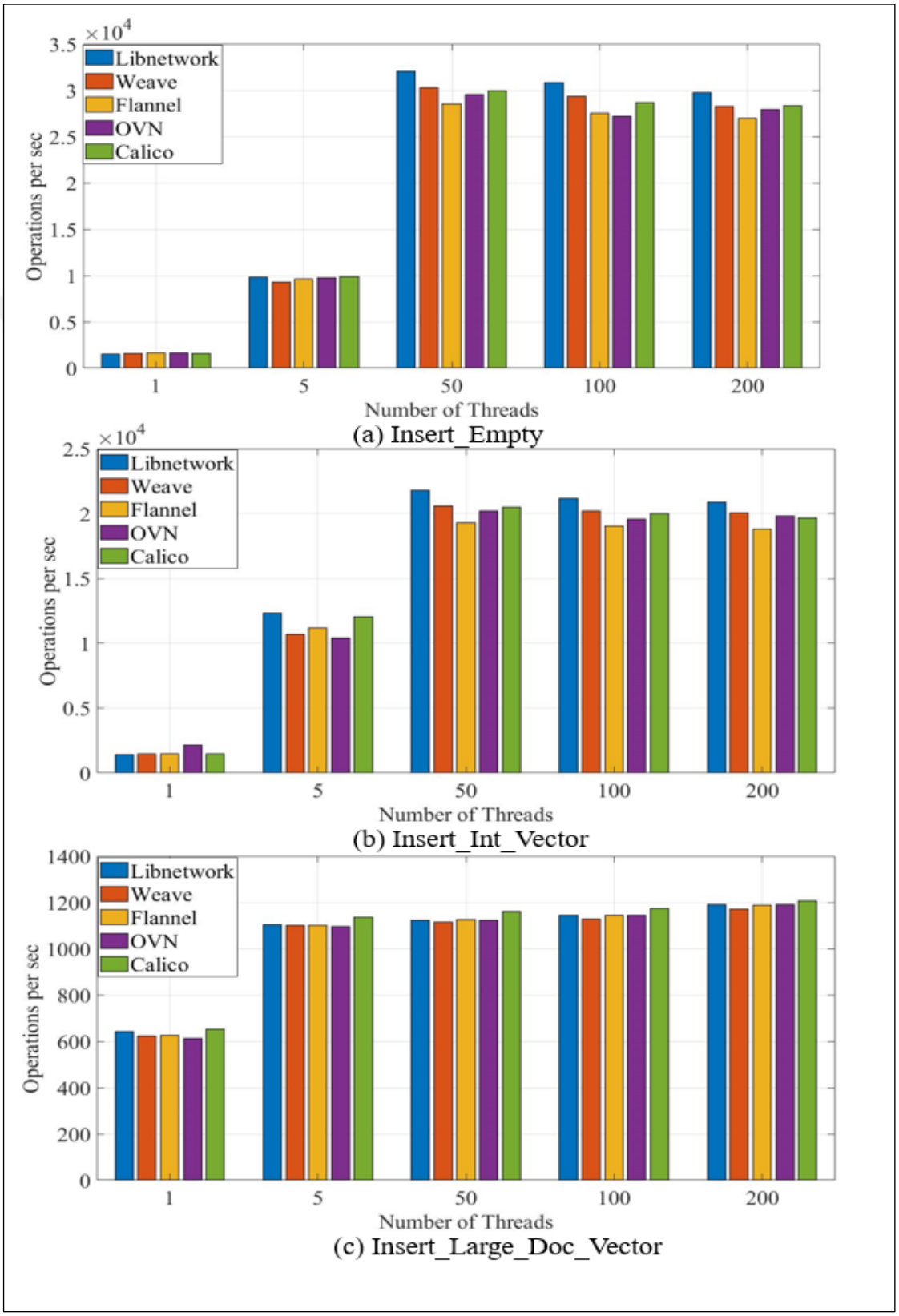
Calico and Flannel gave the highest performance results among tested container networking solutions at the maximum number threads which is 200. In 200 threads, all networking solutions performance has significantly dropped which is around 40 percent according to 100 threads. Unusually, in small number of threads Libnetwork had the worst performance which is nearly 50 percent less than all others.

Table 1.23: HTTP traffic average throughput (request_per_second) results

# of Threads	Libnetwork	Weave	Flannel	OVN	Calico
5	5,88	10,91	10,8182	10,9	10,7727
50	57,05	55,6727	54,41	52,7545	53,12
100	65,1818	62,31	72,4182	66,6909	65,4
200	39,99	39,6	41,5091	40,41	41,3455

5.2.1.2 Database operations throughput

Figure 1.30: MongoDB insert operations throughput comparison. (a) Insert_Empty, (b) Insert_Int_Vector, (c) Insert_Large_Doc_Vector



Three different insert operations which are insert_empty, insert_int_vector, insert_large_doc_vector are performed to evalute the throughput in high traffic situations. The outcomes are given at Table 5.9 are also illustrated at Figure 5.9. In insert_empty operations, all solutions made their highest number of operations within 50 threads, then even thread numbers are increased throughput started to decrease slightly not like HTTP traffic throughput which is presented at Figure 5.9 a. For insert_empty operations, libnetwork gave the highest thoughput than Weave and Calico which has quite close performance to each other in overall. Flannel had the worst performing solution with making 10 percent less operations than Libnetwork. Analyzed container networking solutions' throughputs were very similar in single thread operations, however the multi threading affects the Libnetwork in a positive manner more than others. Libnetwork performance has been boosteed 5.3 times within 5 threads and 19,7 times within 50 threads according the single thread operations. The performance increase in other solutions were not as high as Libnetwork which is maximum 5,1 times in Calico within 5 threads and 18,3 times in Weave within 50 threads. Networking solutions performance ranking was not changed at 50, 100 and 200 threads.

For insert_int_vector operations which are emceed at Figure 5.9 b, Libnetwork performance was the highest in overall which is 5,3 percent more than Calico, around 7,1 percent higher than Weave, OVS which are quite similar results and 11,3 percent more than Flannel. Flannel was the worst performing solution in int_vector_insert operations which has the same situation which is observed in insert_empty operations. OVN made a difference with superior performance which is 45 percent higher than Calico that was second ranked solutions in single thread operations. However, multithreading has least improvement on OVN, number of operations has increased 3,8 times within 5 threads, 8,38 times increased within 50 threads respectively. Nevertheless, multi threading boosted the performance of Libnetwork 7,6 times with 5 threads and 14,3 times with 50 threads which has similar behaviour in insert_empty operations. Besides, multi threading has effected the performance of all container networking solutions in a positive manner. They have reached their highest throughput around 50 threads then performance started to decrease slightly even

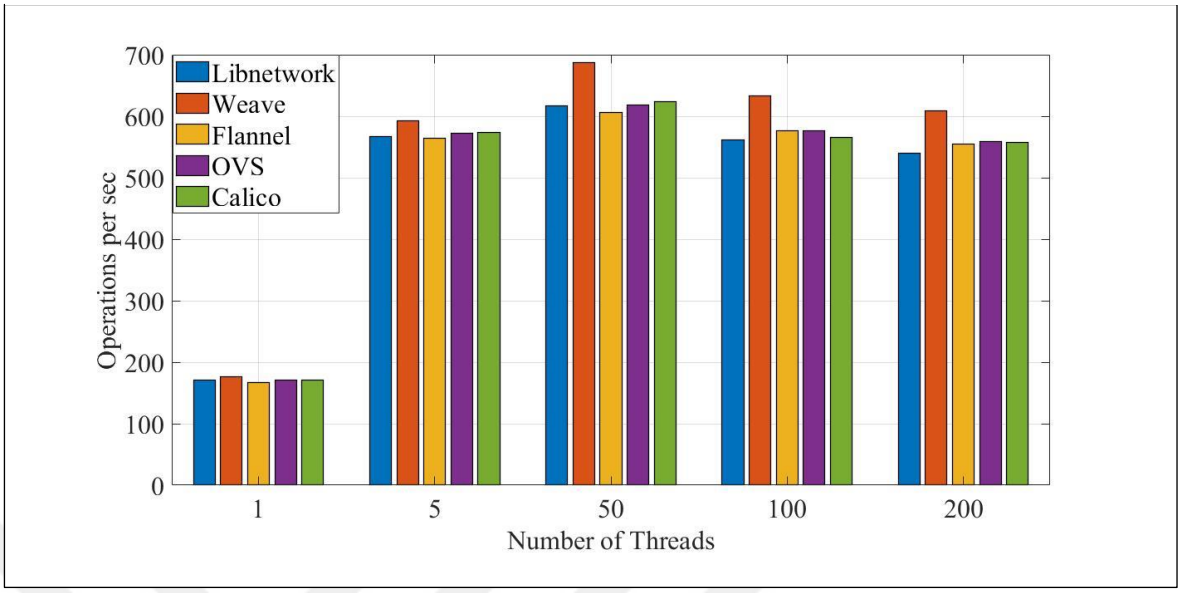
number of operations keep to raise. Generally, container networking solutions performance ranking did not change at 50, 100 and 200 threads.

Insert_large_doc_vector operations outcome of measurements are set out at Figure 5.9. c. Although, solutions performance results are pretty close to each other, Calico made the highest number of operations in all number of threads and it's overall performance is 2,4 percent higher than Libnetwork; 2,8 percent higher than Flannel; 3,2 percent higher than OVS; 3,6 percent higher than Weave at insert_large_doc_vector operations. Unlike insert_empty and insert_int_vector operations, multi threading improved the number of operations insufficiently as number of operations doubled within 5 threads for insert_large_doc_vector operations. While raising the number of threads, quantity of operations has improved only around 2 percent for additional 50 threads. However, performance keep to improve slightly even reaching to 200 threads not like other insert operations throughput that start to decrease after 50 threads.

Table 1.24: MongoDB insert operations throughput (operations per second) results

Operation Type	# of Threads	Libnetwork	Weave	Flannel	OVN	Calico
Insert.Empty	1	1546,16	1562,58	1637,42	1659,6	1593,32
Insert.Empty	5	9855,46	9320,84	9625,72	9758,6	9877,06
Insert.Empty	50	32049,98	30306,88	28551,78	29582,6	29987,64
Insert.Empty	100	30843,42	29374,46	27520,76	27188,8	28698,7
Insert.Empty	200	29748,56	28303,58	26988,46	27907,2	28317,9
Insert.IntVector	1	1423,2	1458,44	1449,54	2152,4	1480,2
Insert.IntVector	5	12339,68	10675,06	11159,68	10389,4	12024,84
Insert.IntVector	50	21776,14	20604,12	19259,28	20206,4	20480,5
Insert.IntVector	100	21181,2	20183,7	19031,02	19561,6	19992,06
Insert.IntVector	200	20869,6	20070,28	18803,82	19835,6	19663,32
Insert.LargeDocVector	1	641,86	623,42	624,7	611	652,38
Insert.LargeDocVector	5	1102,88	1101,86	1101,84	1096	1137,54
Insert.LargeDocVector	50	1124,06	1115,38	1124,52	1122,6	1160,14
Insert.LargeDocVector	100	1144,9	1129,5	1144,9	1145,2	1174,1
Insert.LargeDocVector	200	1189,3	1171,96	1187,44	1190	1206,56

Figure 1.31: MongoDB Update_Field_At_Offset operations throughput comparison

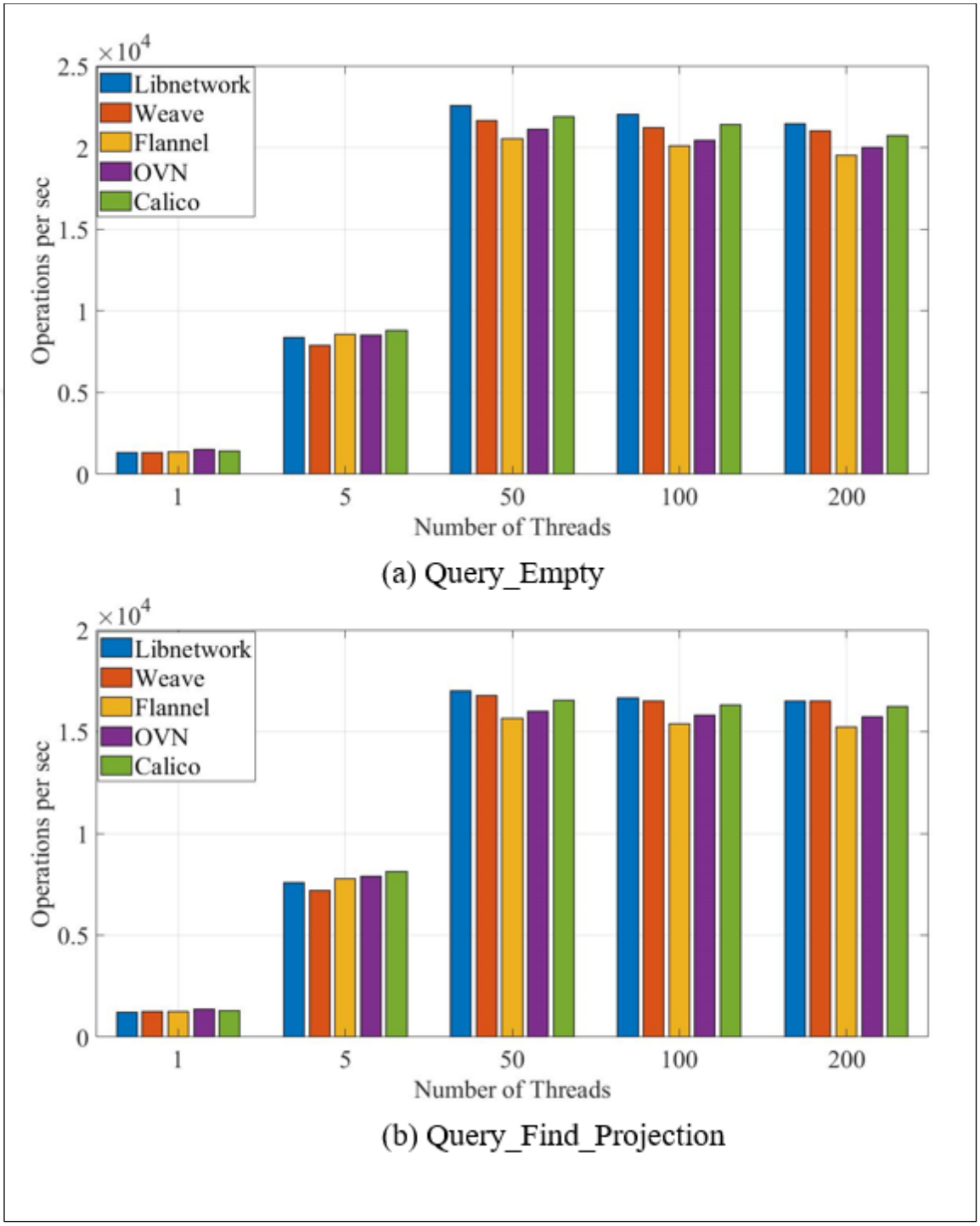


One type of MongoDB update operation that is update_field_at_offset is evaluated. The results are emceed at Figure 5.10 and recorded at Table 5.10. Weave had the maximum throughput in all number of threads. It's throughput is 9.3 percent more than Libnetwork and Flannel by considering the whole test results. Networking solutions gave their maximum throughput at 50 threads, then throughput started to dwindle slightly by adding more threads. Flannel gave the worst performance at 50 threads and Libnetwork made the minimum operations at maximum number of threads.

Table 1.25: MongoDB update operations throughput (operations per second) results

Operation Type	# of Threads	Libnetwork	Weave	Flannel	OVN	Calico
Update.FieldAtOffset	1	171,32	176,26	167,08	170,42	171,24
Update.FieldAtOffset	5	566,78	592,6	563,98	572,44	573,58
Update.FieldAtOffset	50	616,88	687,2	606,26	617,92	623,36
Update.FieldAtOffset	100	561,22	633,1	576,38	576,5	565,78
Update.FieldAtOffset	200	539,32	609,02	555,04	558,2	557,12

Figure 1.32: MongoDB query operations throughput comparison (a) Query_Empty, (b) Query_Find_Projection.

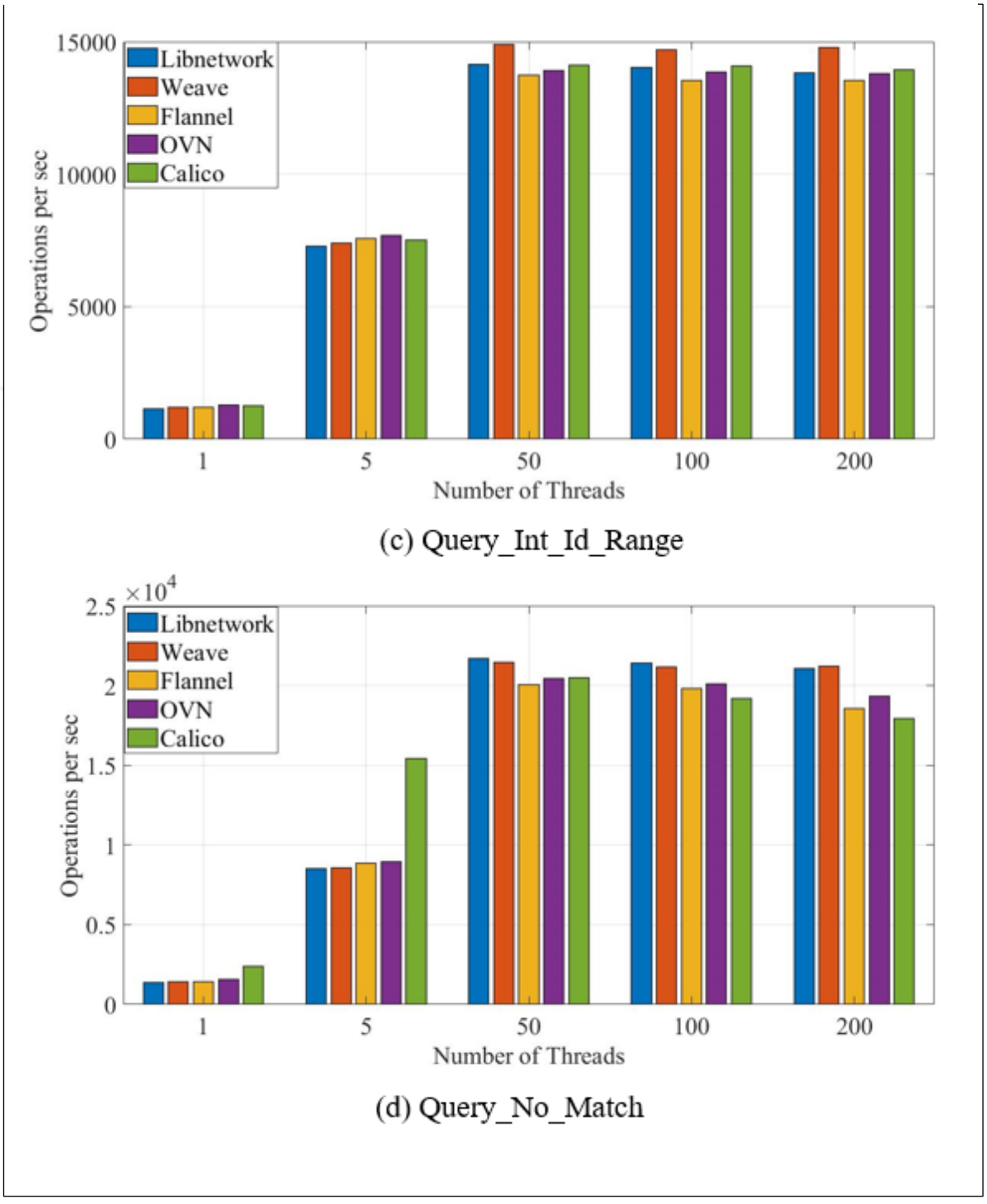


MongoDB query operations behaviour under high traffic load is investigated with 4 different operations which are query_emty, query_find_projection, query_int_id_range, query_nomatch. Outcomes of investigation regarding with query_emty and

query_find_projection has depicted at Figure 5.11 together with numerical outcomes that are written down at Table 5.11. For query_empty operations, OVN which made the maximum number of operations which are 6 percent higher than Calico, aroud 13 percent higher than Weave and Flannel within single thread, gave 5,5 percent less throughput than Libnetwork within multi threads in overall. Although, Libnetwork gave the worst perfomance which is 14.7 percent less than OVN within single thread, its performance has boosted with multi threading. Then it became the best performing solution in overall. Calico, which was the second best performing solution in single thread, gave 3.6 percent less throughput than Libnetwork in overall. Flannel was worst performing solution with 8 percent less thoughput than the Libnetwork in overall. All solutions gave their maximum throughput with 50 threads and throughput has deteriorated slightly while appending more threads which is shown at Figure 5.11 a.

For query_find_projection operations, all tested solutions reached their maximum performance within 50 threads distinctively and their performance started to diminish slightly while keeping to increase the number of threads. OVN is the best performing solution which has higher throughput than Calico, Flannel, Weave and Libnetwork; 5.3, 8.7, 10.2 and 12.4 percent sequantially within single thread. However, Libnetwork performance has boosted with multi threading and became the first ranked solution in overall. Calico which made just 0.7 percent less operations, had quite similar results with Libnetwork. Weave also gave good performance results with 1.4 percent less operations than Libnetwork. OVN, which was highest performing solution in single thread, could not maintain its leadership with multi thread operations and made 3.9 percent less operations in overall. Flannel made the 6.6 percent less operations than Libnetwork and it was the worst performing solution in all container networking solutions for query_find_projection workload which was similar to query_empty, insert_empty and insert_int_vector operations.

Figure 1.33: MongoDB query operations throughput comparison. (c) Query_Int_Id_Range, (d) Query_No_Match



For query_int_id_range database operations which are exhibited at Figure 5.12 c, OVN made the maximum number of operations which is 3 percent higher than Calico, 7.2 , 8.3 percent higher than Flannel and Weave sequantially within single thread.

Libnetwork was the worst performing solution within single thread operations which is 11.6 percent less than OVN. This alignment has changed with the effect of multi thread operations. Weave became the best performing solution which had higher performance than Calico, OVN and Libnetwork by the percent of 4, 4.8, 4.9 by order in whole tests including single and multi thread operations. Flannel maintained its worst performing rank with 6.8 percent less operations than Weave according to all tested scenarios of query_int_id_range operations. While keeping to increase the number of threads, achieving the maximum performance within 50 threads and vaguely decrease, were displayed in query_int_id_range operations.

Table 1.26: MongoDB query operations throughput (operations per second) results

Operation Type	# of Threads	Libnetwork	Weave	Flannel	OVN	Calico
Queries.Empty	1	1331,7	1343,92	1387,58	1528,72	1442,02
Queries.Empty	5	8383,8	7868,38	8569,7	8539,04	8819,88
Queries.Empty	50	22586,64	21666,08	20536,34	21110,48	21882,72
Queries.Empty	100	22017,96	21215,66	20124,5	20429,24	21410,92
Queries.Empty	200	21434,92	21025,08	19522,38	20023,66	20712,3
Queries.NoMatch	1	1379,52	1409,1	1441,16	1546,44	2404,34
Queries.NoMatch	5	8540,32	8556,54	8834,94	8957,9	15418,68
Queries.NoMatch	50	21713,32	21443,1	20040,74	20424,44	20503,16
Queries.NoMatch	100	21402,62	21150,66	19788,1	20091,74	19201,16
Queries.NoMatch	200	21088,72	21197,22	18550,2	19327,38	17951,88
Queries.IntIDRange	1	1155,78	1191,68	1202,92	1290,66	1249,5
Queries.IntIDRange	5	7284,74	7392,22	7569,26	7677,56	7501,8
Queries.IntIDRange	50	14142,7	14910,78	13731,4	13910,68	14121,48
Queries.IntIDRange	100	14037	14703,94	13527,78	13868,8	14090,9
Queries.IntIDRange	200	13835,82	14776,4	13548,72	13784,06	13934,42
Queries.FindProjection	1	1215,54	1238,94	1256,94	1366,38	1297,26
Queries.FindProjection	5	7576,82	7190,1	7786,1	7892,16	8133,34
Queries.FindProjection	50	16998,16	16757,7	15639,7	15994,16	16562,62
Queries.FindProjection	100	16674,28	16487,34	15393,4	15794,58	16320,2
Queries.FindProjection	200	16522,96	16493,6	15221,62	15722,24	16225,62

All tested container networking solutions achieved the maximum number of operations around 50 threads and number of operations started to decrease while keeping to increase thread numbers in query_no_match operations which are given at Figure 5.12.d. Distinguishing from other operations, Calico captured the best

performing solution title from OVN and it gave a excellent performance both in single and 5 multi threaded operations. Calico obtained 55 percent more operations within single thread and 72 percent more operations within 5 multi threads than OVN which was second best performing solution. Libnetwork gave the worst performance in single thread operations. However, Libnetwork performance has rised extremely while increasing thread numbers to 50,100 and 200. Libnetwork captured the first rank by providing 11.3 percent more operations than Calico and 0.6 percent operations than Weave.

5.2.1.3 Throughput according to message size

TCP request-response tests are executed by changing the packet size with 10 multiple streams together and results are limned at Figure 5.13. a. Throughput has exceeded the host's network interface capacity which is 1 Gbps because of multi threading and using big message sizes. Baremetal host has exceeded the container networking solutions throughput till 1024 bytes packet sizes, however Libnetwork and Flannel throughput has surpassed the baremetal host at that size of packages. This behaviour also continued for bigger packet sizes and throughput of all tested container networking solutions outgone the baremetal throughput. It is presumed that this is the effect of offloading which is encountered in TCP_CRR results with single stream. Additionally, except the 32 bytes message sizes, Libnetwork gave the highest performance by the influence of multi threading.

Similar to TCP_RR tests, UDP_RR tests are also performed by increasing the packet size from 32 to 4096 within 10 multiple streams. Outcomes are depicted at Figure 5.13 b and all the request-response results are recorded at Table 5.12. Unlike TCP_RR, baremetal gave the highest throughput for all packet sizes. Similar to TCP_RR multi stream performance results, throughput has exceeded the host's network interface capacity which is 1 Gbps because of multi threading and utilizing big message sizes. While Calico was giving the highest throughput for small packet sizes, it provided the bad performance within big packet sizes with multiple streams. Moreover, Weave provided the worst performance for all packet sizes for whole test scenario. OVN UDP_RR performance has improved within bigger packet sizes such as 4096 bytes.

Figure 1.34: Throughput comparison within ten multiple streams according to packet size. (a) TCP_RR, (b) UDP_RR

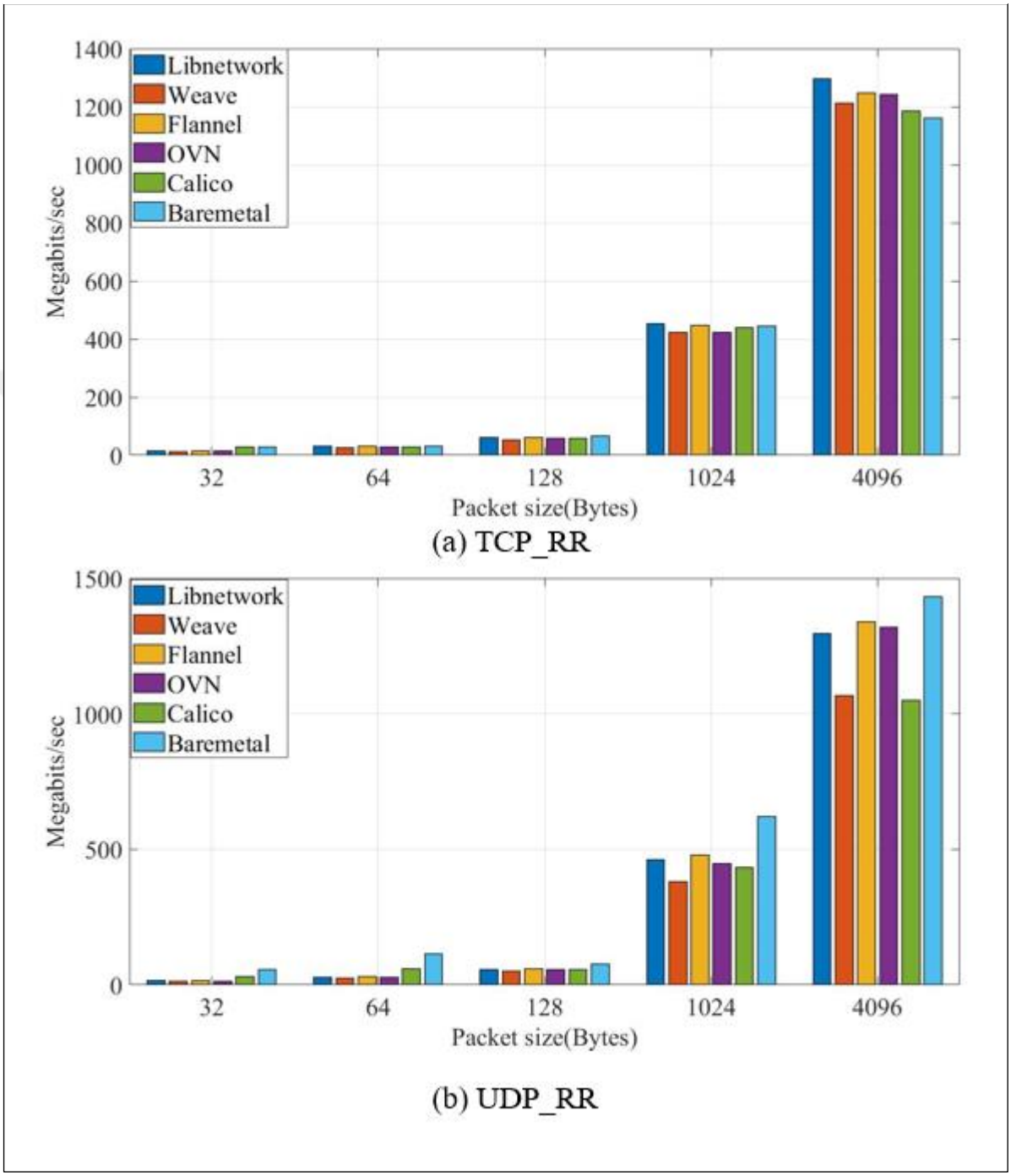


Table 1.27: Throughput(Mbits/sec) according to packet size with paralel streams

Test Type	Packet Size	Baremetal	Libnetwork	Weave	Flannel	OVN	Calico
TCP_RR	32	28,246	14,625	12,49	15,8815	13,5995	27,726
TCP_RR	64	32,0565	29,9985	25,894	31,7955	28,296	29,6025
TCP_RR	128	67,452	60,967	52,06	61,9025	57,602	58,618
TCP_RR	1024	443,624	451,914	421,819	446,772	421,707	439,526
TCP_RR	4096	1159,92	1295,73	1211,69	1246,82	1242,96	1184,05
UDP_RR	32	57,4735	14,9255	12,84	15,757	14,0275	29,885
UDP_RR	64	113,183	28,825	25,0995	30,4395	27,93	60,562
UDP_RR	128	75,63	57,348	49,4165	59,787	55,62	57,1275
UDP_RR	1024	620,036	463,109	381,267	479,272	447,19	431,531
UDP_RR	4096	1432,33	1296,49	1067,83	1339,36	1318,54	1049,7

5.2.2 Bandwidth

TCP bandwidth is measured within multiple streams with 128KB message sizes and outcomes are written down at Table 5.13, together with picturizing the results in Figure 5.14 a. Calico had the maximum bandwidth for all number of streams varying from 5 to 100. Although, all container networking solutions bandwidth's were quite close to each others, OVN gave the minimum bandwidth with 5 and 10 streams. Maximum bandwidth for all solutions has observed within 10 multiple streams. Increasing the streams from 10 to 100, insignificant bandwidth loss was observed for all container networking solutions and Libnetwork gave the minimum bandwidth with 100 streams.

UDP bandwidth is measured withing 8KB message size by different number of streams varying from 1 to 100. Bandwidth increased proportional to the number of streams and all container networking solutions supplied the same bandwidth as expected; the reason why UDP bandwidth has calculated according to data which is sent over the channel unlike TCP which utilizes the whole channel capacity. The outcomes regarding with the executions are recorded at Table 5.14 while depicting at Figure 5.14 b.

Figure 1.35: Bandwidth comparison according to load. (a) TCP bandwidth, (b) UDP bandwidth

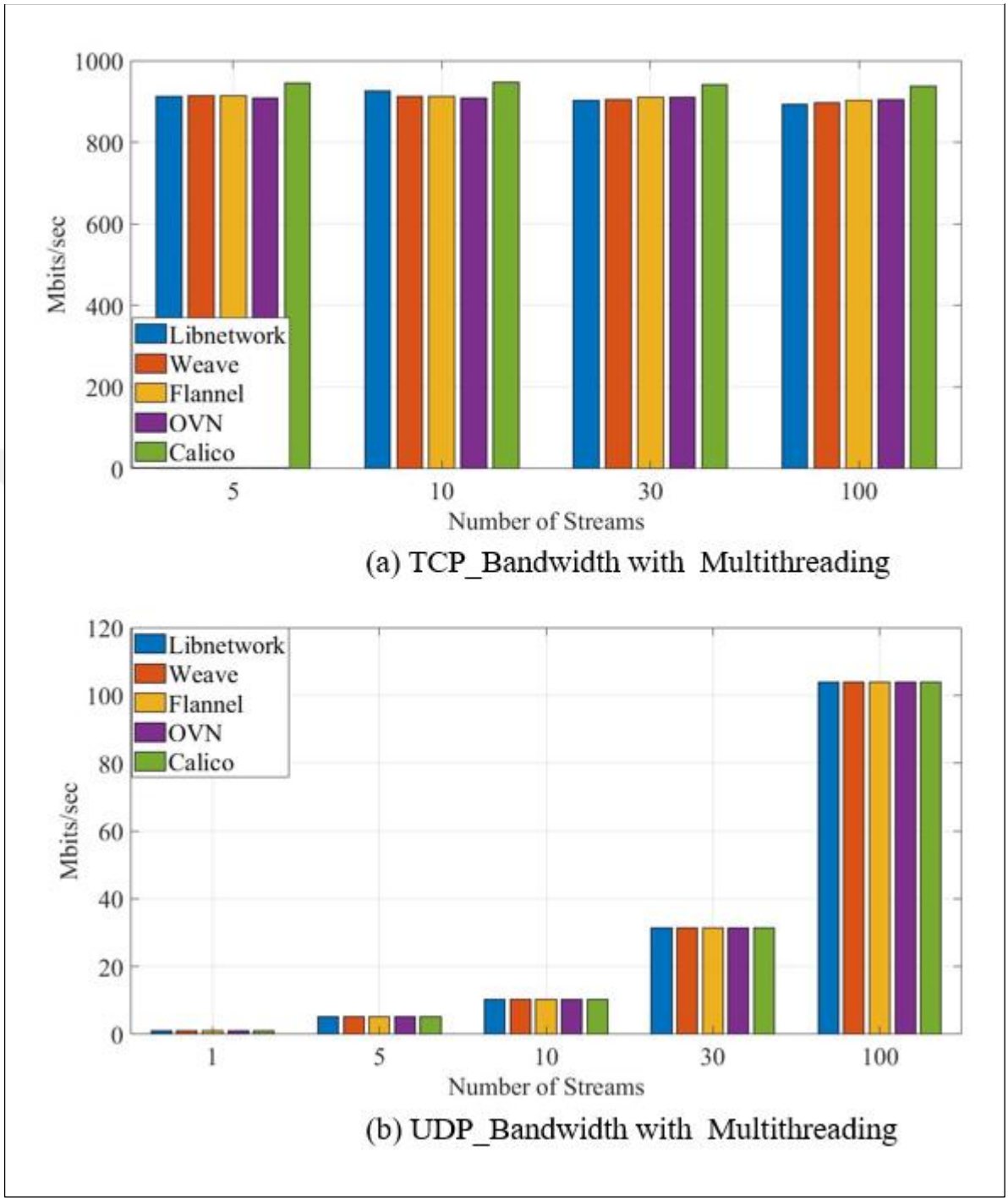


Table 1.28: TCP bandwidth (Mbits/sec) according to load

# of Streams	Libnetwork	Weave	Flannel	OVN	Calico
5	911,85	913,3	914,7	908,632	945,3
10	926,05	913,1	913,1	908,5	946,1
30	902,85	904,15	910,353	909,789	941,647
100	891,933	896,588	902,182	903,909	937,273

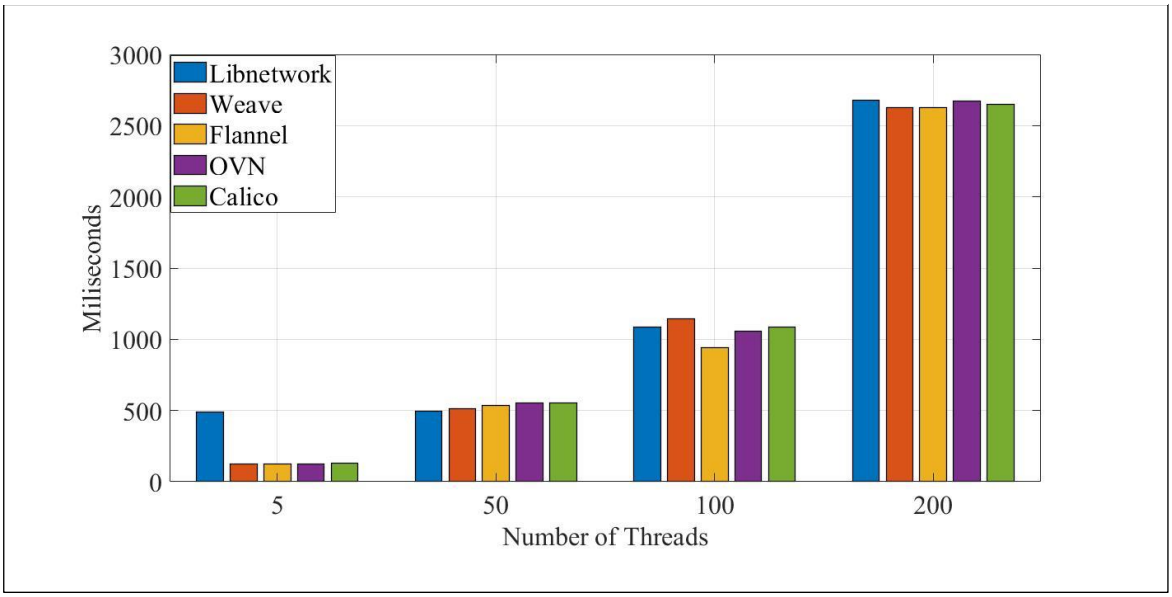
Table 1.29: UDP bandwidth (Mbits/sec) according to load

# of Streams	Libnetwork	Weave	Flannel	OVN	Calico
1	1,04	1,04	1,04	1,04	1,04
5	5,21	5,21	5,21	5,21	5,21
10	10,4	10,4	10,4	10,4	10,4
30	31,3	31,3	31,3	31,3	31,3
100	104	104	104	104	104

5.2.3 Response Times

5.2.3.1 Web access operations response times

Figure 1.36: Web application average response times comparisons



Web application response times within multiple threads are evaluated and depicted at Figure 5.15. Libnetwork gave the worst response times around 3 times higher than other solutions within 5 threads. While increasing the number of threads, response times are increased significantly. Furthermore response times are deteriorated more than twice when threads are increased from 100 to 200. While other solutions were giving very similar response times, Libnetwork gave the highest response times that are 12 percent higher than Flannel, which had the minimum response times in overall. The outcomes of this experiment is recorded at Table 5.15.

Table 1.30: HTTP traffic average response time (miliseconds)results

# of Threads	Libnetwork	Weave	Flannel	OVN	Calico
5	489,4	126,5	126,818	127,4	129,364
50	495,9	511,818	534,7	554,909	552,8
100	1087,91	1146,4	941	1054,36	1086,4
200	2679,2	2627,42	2625,27	2671,6	2646,82

5.2.4 Reliability

Number of retransmitted TCP segments, UDP jitter and number of UDP lost datagrams are selected as reliability indicators and multi-host container networking reliability under load are evaluated over those parameters.

5.2.4.1 Retrasmitted TCP segments

During multi threaded bandwidth evaluations on TCP, number of TCP retransmittted segments are also gathered and they are picturized at Figure 5.16 a. While increasing the number of streams, Libnetwork TCP retransmissions has mounted up within 100 streams. Weave made minimum retransmissions during the whole test except 5 number of streams as Flannel had the minimum retransmission at that execution. Libnetwork made the maximum number of retransmissions on 5 and 100 streams which are minimum and maximum stream numbers. All the numerical outputs are displayed at Table 5.16.

Figure 1.37: Reliability comparisons under Load. (a) Retrasnmitted TCP segments, (b) UDP Jitter, (c) UDP Lost Datagram

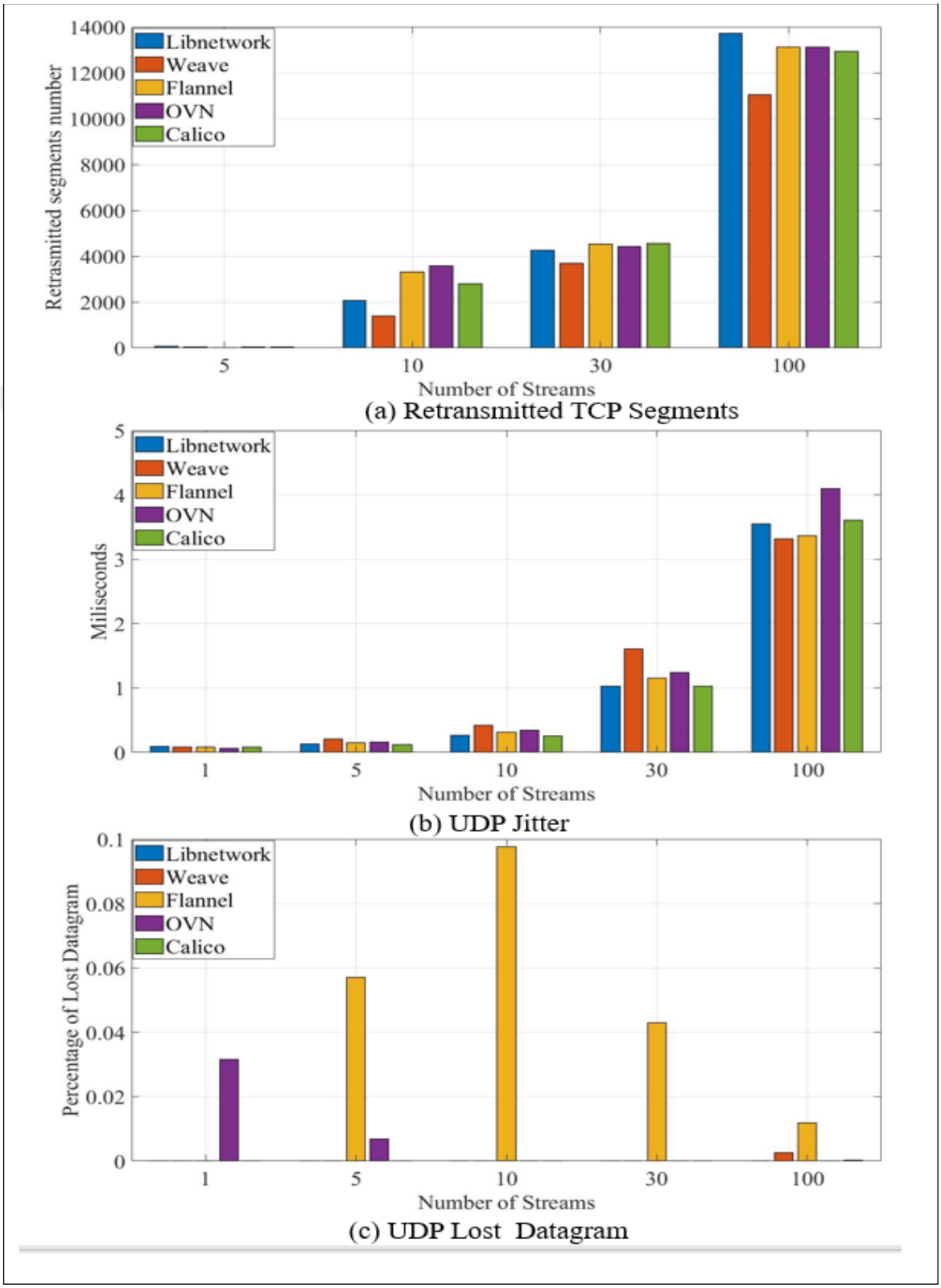


Table 1.31: Number of retransmitted TCP segments according to load

# of Streams	Libnetwork	Weave	Flannel	OVN	Calico
5	54,35	41,45	11,95	31,9474	32,65
10	2076,45	1403,65	3317,9	3582,15	2787,7
30	4244,1	3692,05	4518	4431,11	4564
100	13709,4	11049,2	13110,7	13117,1	12930,4

5.2.4.2 UDP jitter

During the multithreaded bandwidth evaluations on UDP, jitter is also observed and limned at Figure 5.16 b. Libnetwork gave the maximum jitter within single streams. While increasing the number of streams, Weave gave the worst jitter within 5, 10 and 30 streams. Furthermore, OVN jitter increased freakingly in transition from 30 to 100 streams similar to Calico and Libnetwork. Calico had the minimum jitter for 5,10 and 30 streams. While having the highest jitter within 5, 10, 30 streams, Weave had the minimum jitter which is slightly less than Flannel's jitter value within 100 streams. Although OVN gave the minimum jitter with single streams, it gave the maximum jitter within 100 streams. Obtained jitter outcomes are recorded at Table 5.17.

Table 1.32: UDP Jitter (miliseconds)according to load

# of Streams	Libnetwork	Weave	Flannel	OVN	Calico
1	0,09555	0,08625	0,08145	0,0663	0,07965
5	0,1259	0,210263	0,14665	0,159421	0,1239
10	0,2691	0,4246	0,31245	0,3394	0,259
30	1,02595	1,6067	1,15695	1,2386	1,0236
100	3,54265	3,3169	3,36265	4,10065	3,60674

5.2.4.3 Lost datagrams

Multithreaded bandwidth evaluations on UDP also provided the UDP lost datagram results that are displayed at Figure 5.16 c. None of the container networking solutions lost any UDP package within single stream except OVN. It is supposed that this is the effect of first hit misses and solidifies bad impacts of long data path of OVN as

after path flow has created, none of the packets were lost which are shown at Table 5.18. Starting from 5 multiple streams, Flannel started to lose UDP datagrams and it lost around 0.1 percent of datagrams within 10 multiple streams; although other solutions didn't lose any datagram till 100 operations. However, Flannel lost datagrams in all multi thread operations, Weave and Calico only lost insignificant amount of datagram at maximum number of threads.

Table 1.33: UDP lost datagram as percentage of all datagrams

# of Streams	Libnetwork	Weave	Flannel	OVN	Calico
1	0	0	0	0,0315	0
5	0	0	0,057	0,00684211	0
10	0	0	0,0975	0	0
30	0	0	0,043	0	0
100	0	0,002565	0,0118	0	0,000331579

5.3 ANALYSIS AND DISCUSSIONS

Multi-host container networking solutions performance results are gathered using several performance metrics for single thread and multi thread executions. In order to evaluate several features together, Z-score normalization has applied to the results and all the values are reduced in to same scale. These results which are in the same scale has placed into Annex A. Normalized relative results are classified according to their sign and values. Normalized outcomes which are close to zero denominated as fair, while highest positive outcomes are entitled as very good and lowest negative results are called as very bad. The positive normalized results between mean and highest are called good meanwhile negative results between mean and lowest values are called bad in order the compare cotainer multi-host networking solutions with each others. For negative features such as latency, response times, signs of the normalized results are reversed. During the assessments, 32, 64, 128 bytes message sizes are assumed as small; 1024, 4096 bytes message sizes are called as large. Therewithal, 8, 16, 32 KB window sizes are called as small and 64, 128 KB window sizes are called large.

Table 1.34: Message size effects evaluation on throuhput and latency within single thread

Name	Small MS Thput	Large MS Thput	Small MS Latency	Large MS Latency
Libnetwork	Fair at TCP_STREAM Very Good at UDP_STREAM Bad at TCP_CRR	Fair at TCP_STREAM Very Good at UDP_STREAM Bad at TCP_CRR	Fair at TCP_STREAM Good at UDP_STREAM	Bad at TCP_STREAM Good at UDP_STREAM
Weave	Very Bad at TCP_STREAM Very Bad at UDP_STREAM Bad at TCP_CRR	Bad at TCP_STREAM Fair at UDP_STREAM Bad at TCP_CRR	Very Bad at TCP_STREAM Bad at UDP_STREAM	Bad at TCP_STREAM Bad at UDP_STREAM
Flannel	Bad at TCP_STREAM Fair at_UDP_STREAM Bad at TCP_CRR	Fair at TCP_STREAM Fair at UDP_STREAM Bad at TCP_CRR	Bad at TCP_STREAM Very Bad at UDP_STREAM	Bad at TCP_STREAM Very Bad at UDP_STREAM
OVN	Very Good at TCP_STREAM Very Good at UDP_STREAM Very Bad at TCP_CRR	Very Bad at TCP_STREAM Good at UDP_STREAM Very Bad at TCP_CRR	Very Good at TCP_STREAM Very Good at UDP_STREAM	Very Bad at TCP_STREAM Very Good at UDP_STREAM
Calico	Good at TCP_STREAM Good at UDP_STREAM Very Good at TCP_CRR	Very Good at TCP_STREAM Very Bad at UDP_STREAM Very Good at TCP_CRR	Good at TCP_STREAM Good at UDP_STREAM	Very Good at TCP_STREAM Good at UDP_STREAM

Container networking solutions single thread throughput and latency evaluation in terms of message sizes are shown in Table 6.1. Calico is the only solution which provides high throughput for TCP/UDP bulk data transfers alongside highest throghput for HTTP like connect-request-response traffic within small message sizes. For large messages sizes, Calico still served the highest throughput values among all tested solutions except bulk UDP data transfers which has maximum throughputs on Libnetwork both in small and large message sizes. However, Libnetwork gave low throughput for TCP_CRR and fair thoughput for bulk TCP transfers regardless from message sizes.

Weave yielded the low throughput results for all data transfer types with small messages, while giving low TCP thoughput results within large message sizes.

Flannel bulk TCP data throughput enhanced with large message sizes, meantime providing fair throughput for bulk UDP transfers and low throughput for TCP_CRR regardless from message sizes. While increasing the message sizes, OVN relative performance remained insufficient according to other solutions for TCP stream and CRR traffic. At the same time it gave high UDP bulk throughput.

Calico is the only solution which gave low and lowest latency results for both small and large message sizes respectively. OVN which gave the lowest latency results within small message sizes for bulk TCP and UDP transfers, TCP latency got worsened within large message sizes, while UDP bulk data latency was not affected.

Flannel and Weave gave the high latency results for TCP and UDP bulk transfers both in small and large message sizes. Libnetwork gave the low latency for bulk UDP transfers irrelevant from message sizes, however it gave the fair and high bulk TCP transfer latency respectively in small and large messages sizes.

Table 1.35: Window size effects evaluation on reliability (based on retransmitted TCP segments) and TCP bandwidth within single thread

Name	Small WS Band	Large WS Band	Small WS Rel	Large WS Rel
Libnetwork	Good	Very Bad	Very Bad	Very Bad
Weave	Fair	Bad	Bad	Very Bad
Flannel	Good	Fair	Good	Good
OVN	Very Bad	Bad	Very Good	Good
Calico	Very Good	Very Good	Bad	Very Good

TCP bandwidth and reliability by considering retransmitted TCP streams evaluation according to window size is displayed at Table 6.2. Calico gave the highest bandwidth both in small and large window sizes. OVN gave the lowest bandwidth in small window size (WS) and increment in WS has positive effect on it's bandwidth more than Libnetwork, Weave and Flannel. Weave was the second worst performing solution after OVN with fair and low bandwidth respectively in small and large WS. Libnetwork and Flannel which had high bandwidth at small WS, gave in order of lowest and fair bandwidth within large WS.

Reliability according to WS has calculated through the number of retransmitted TCP segments. Calico had the low reliability in small WS, it became the most reliable

solution within large WS. OVN and Flannel had high reliability in all WS, while Libnetwork and Weave had low reliability on all window sizes.

Table 1.36: Single thread bandwidth and latency evaluation

Name	Bandwidth	Latency
Libnetwork	Bad at TCP Very Bad at UDP Fair at SCTP	Good at TCP Fair at UDP Fair at SCP
Weave	Bad at TCP Very Bad at UDP Very Bad at SCTP	Good at TCP Very Bad at UDP Bad at SCTP
Flannel	Fair at TCP Bad at UDP Very Good at SCTP	Good at TCP Bad at UDP Very Bad at SCTP
OVN	Very Bad at TCP Very Bad at UDP Very Good at SCTP	Very Bad at TCP Good at UDP Very Good at SCTP
Calico	Very Good at TCP Very Good at UDP	Very Good at TCP Very Good at UDP

Container networking solutions single thread bandwidth and latency evaluation are given at Table 6.3. Calico was the best berforming solution with highest bandwidth and lowest latency both in TCP and UDP. While OVN had the lowest bandwidth, Libnetwork and Weave gave the low bandwidth both in TCP and UDP. Unlike them, Flannel gave fair bandwidth in TCP and low bandwidth in UDP. From the latency point of view, Libnetwork, Weave and Flannel gave low latency when OVN gave highest latency in TCP. However OVN gave low latency, while Weave, Flannel were providing high latency and Libnetwork has fair latency in UDP.

Calico was not evaluated for SCTP as tested version does not support SCTP yet. OVN was the most successful solution with highest bandwidth and minimum latency in SCTP. Libnetwork had fair and Weave had bad results both in SCTP bandwidth and latency. Flannel gave highest SCTP bandwidth although having highest SCTP latency.

Table 1.37: MongoDB and web application workloads throughput and gain evaluation within multithreading

Name	Max Thput	Multithreading Gain	Heavy Traffic Load Thput
Libnetwork	Very Good at MongoDB_Ins Very Good at MongoDB_Que Bad at MongoDB_Upd Bad at MongoDB_HW Bad at HTTP	Very Good at MongoDB_Ins Very Good at MongoDB_Que Very Bad at MongoDB_Up Very Good at HTTP	Very Good at MongoDB_Ins Good at MongoDB_Que Very Bad at MongoDB_Up Bad at HTTP
Weave	Bad at MongoDB_Ins Good MongoDB_Que Very Good MongoDB_Up Very Bad at HTTP	Good at MongoDB_Ins Good at MongoDB_Que Very Good at MongoDB_Up Very Bad at HTTP	Bad at MongoDB_Ins Very Good at MongoDB_Que Very Good at MongoDB_Up Very Bad at HTTP
Flannel	Very Bad at MongoDB_Ins Very Bad at MongoDB_Que Very Bad at MongoDB_Up Very Good at HTTP	Bad at MongoDB_Ins Bad at MongoDB_Que Bad at MongoDB_Up Fair at HTTP	Very Bad at MongoDB_Ins Very Bad at MongoDB_Que Bad at MongoDB_Up Very Good at HTTP
OVS	Bat at MongoDB_Ins Bad at MongoDB_Query Bad at MongoDB_Up Fair at HTTP	Bad at MongoDB_Ins Very Bad at MongoDB_Que Bad at MongoDB_Up Bad at HTTP	Bad at MongoDB_Ins Bad at MongoDB_Que Bad at MongoDB_Up Fair at HTTP
Calico	Good at MongoDB_Ins Bad at MongoDB_Que Bad at MongoDB_Up Bad at HTTP	Bad at MongoDB_Ins Bad at MongoDB_Que Bad at MongoDB_Up Bad at HTTP	Good at MongoDB_Ins Bad at MongoDB_Que Bad at MongoDB_Up Good at HTTP

MongoDB and web application access workloads throughput and gain evaluation is depicted at Table 6.4. MongoDB insert, update and query operations may utilize I/O, CPU and memory resources with different portions in tems of operation type and it's difficulty. Although resource utilizations on the test containers are not observed in this thesis, HTTP request operations are estimated as CPU intensive operations. While increasing the threads, close to a certain number of threads maximum throughput has taken with multithreading. This is the optimum value that resources are utilized most effectively. Usually maximum thoughput has taken within 50 threads except insert_large_doc_vector test with MongoDB operations. Insert_large_doc_vector test gives the maximum thoughput within 200 threads which is the maximum thread number. Multithreading gain has calculated by the ratio of maximum throughput to single thread throughput. Maximum number of threads to be executed has determined by testing the container while increasing the thread. After 200 threads, the container which is under test can not respond in a healty way in web acess and MongoDB operations, hence 200 threads are assigned as heavy load state. For bandwidth tests, heavy load point identified as 100 threads. It is assumed that container resources are congested and utilized at the highest level at heavy load.

According to outputs that are shown at Table 6.4, Libnetwork gave the highest throughput for MongodB insert and query operations, while giving the low throughput for update and web accces operations. Weave provided highest throughput for update operations and high throughput for query operations, although it was bad at insert and web access operations. Flannel delivered the lowest thoughput for all database operations while it was providing the highest throughput at web access operations. Similar to Flannel, OVN had low throughput at all database operations meantime it had fair throughput at web access operations. Calico offered high throughput for insert operations, although it yielded low thoughput at query, web access and update operations. From these results, Weave was the only solution which provided highest thoughput at update operations with significant difference than others and none of the solutions gave relatively high throughput for all type of workloads.

From the aspect of multithreading gain, Libnetwork gave the highest multithreading gain for all workloads except update operations. Weave offered the high multithreading gain for all database operations however it rendered quite low gain for web access operations. Contrariwise, Flannel gave the low gain for all database operations while offering average gain for web access operations. OVN and Calico gave low multithreading gain for all type of workload operations.

According to the results which are gathered under heavy traffic load, Libnetwork catered high throughput for insert and query operations, however gave relatively low throughput for update and web request operations. It is observed that Libnetwork relative performance has deteriorated with multithreading as it yielded the lower performances under high traffic load for MongoDB query and update operations compared to optimum number of threads point. Weave provided high throughput for query and update operations, while giving low throughput for insert and web request operations under heavy load. Weave relative performance under traffic load was improved by taking better rank for MongoDB query operations, while the it's rank was not changed for remaining operations.

Flannel gave high throughput only for web request operations and provided relatively low throughput for all database operations under heavy load. Flannel's rank for MongoDB update operations was also escalated under heavy traffic load while it displayed the similar ranks for other operations. OVN yielded fair throughput for web access operations, meanwhile assured low throughput for insert, query and update operations under heavy traffic load. It's behaviour under heavy load remained same.

Calico provided relatively low throughput for query and update operations, while giving high throughput for insert and web access operations under high traffic load. It's rank for web operations was meliorated under heavy load while heavy load behaviour for other operations stayed same.

Table 1.38: Message size effects with multithreading and heavy traffic load effects on bandwidth, response times and reliability

Name	Message Size &Throughput	Heavy Load TCP Bandwidth	Heavy Load HTTP Response Time	Heavy Load Reliability
Libnetwork	Fair at Small TCP&UDP Very Good at Large TCP Good at Large UDP	Very Bad	Very Bad	Fair
Weave	Very Bad at Small TCP&UDP Very Bad at Large TCP&UDP	Bad	Very Good	Very Good
Flannel	Good at Small TCP&UDP Good at Large TCP Very Good at Large UDP	Fair	Very Good	Very Bad
OVN	Bad at Small TCP&UDP Good at Large TCP Fair at Large UDP	Good	Bad	Bad
Calico	Very Good at Small TCP& UDP Bad at Large TCP&UDP	Very Good	Good	Good

Message size effects with multithreading together with bandwidth, response times and reliability evaluation under heavy traffic load is commentated at Table 6.5. Message size effects are observed in TCP_RR and UDP_RR transfers within multithreading covering 10 parallel streams. As TCP_RR is HTTP like traffic, Flannel which was quite succesful on HTTP like transfers exhibited relatively high throughput both in small and large message sizes in TCP and UDP among other container networking solutions with multithreading. Similar results also observed at Table 5.8 for Flannel.

Libnetwork and OVN which give fair and low request/response throughput respectively at small messages sizes, improved their throughput within large message sizes both in TCP and UDP with paralled executions. Weave provided the lowest request/response throughput both in TCP and UDP regardless from the message sizes.

Contrariwise to Libnetwork and OVN, Calico provided the highest request/response throughput at small message sizes, however it obtained the relatively low throughput within large message sizes both in TCP and UDP.

Calico provided the highest TCP bandwidth, while OVN, Flannel, Weave and Libnetwork were assuring high, fair, low and lowest bandwidth under heavy load. Response time results were obtained from web operations and Flannel, Weave gave the lowest web access response times under heavy traffic, meantime Calico catered low response times. Besides, OVN and Libnetwork was giving in order of high and highest response times.

Reliability under heavy load has obtained by considering the retransmitted TCP segments, UDP lost datagram and UDP jiiter outputs under heavy traffic load. Taking into account of these three features, Weave was the most reliable and Calico was the second ranked solution among all of them. While Libnetwork had fair reliability, OVN and Flannel were in order of unreliable and most unreliable solutions.

5.4 PROPOSED ARHITECTURE

Container cluster networking solutions provided mutable performances depending on traffic load for different VNF applicable operations. It is deduced from the experiments that container networking solutions' performance changes depending on the traffic load and application type. Container networking solutions usually yield a maximum throughput within 50 threads which is therefore designated as the optimum load. The tested maximum thread number is 200. This is assigned as a heavy load state for Web access and MongoDB operations. In order to evaluate several features together, Z-score normalization has been applied to the results, and all the values are reduced to the same scale. Under these circumstances, by utilizing the normalized results, high performing container networking solutions affiliated to application type, traffic load and robustness demand are pieced together in Table 5.24.

Table 1.39: High performing networking solutions according to test results by considering application requirements, traffic loads and robustness

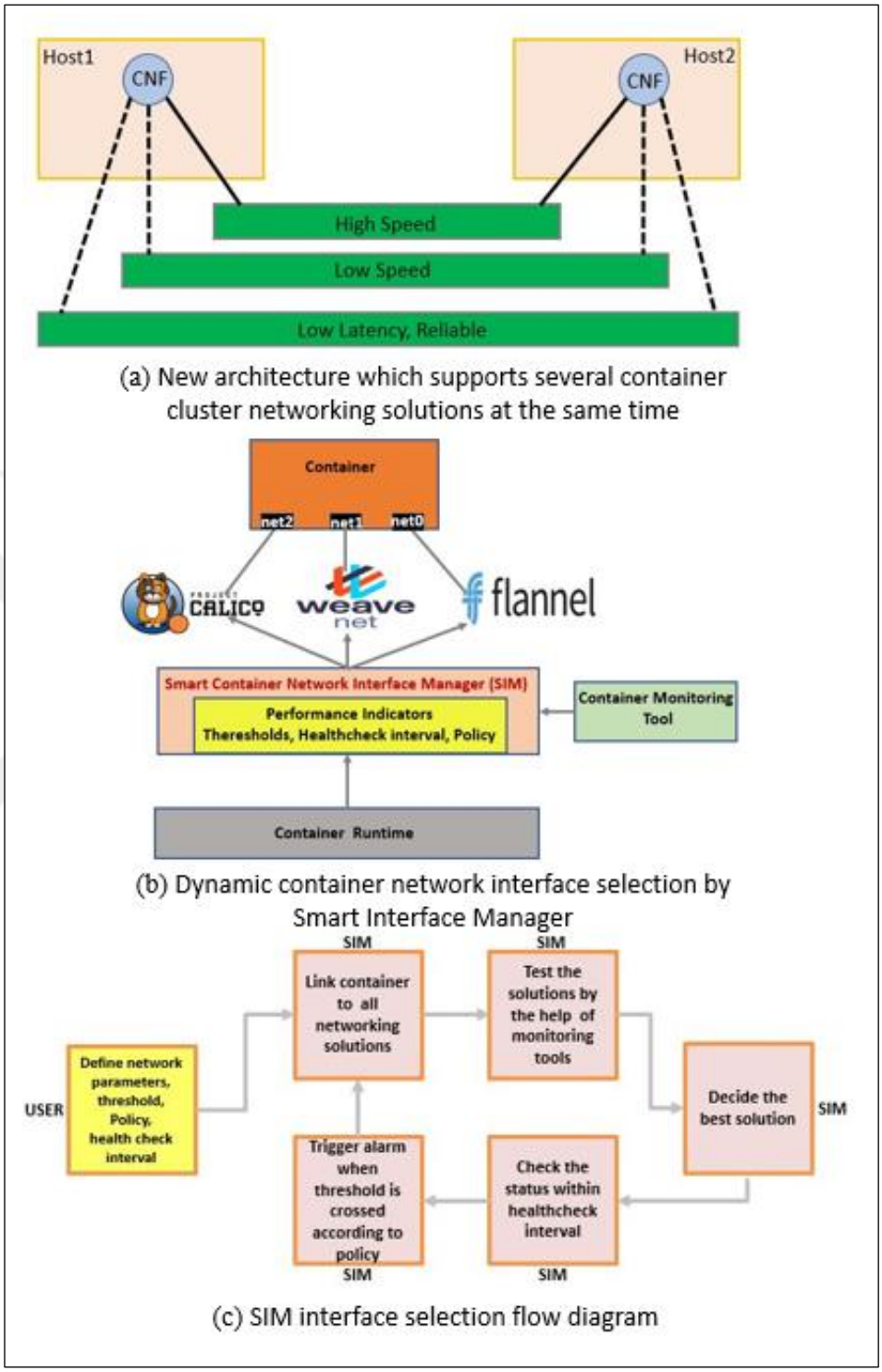
Application	Heavy Load, Robust	Heavy Load, Unrobust	Optimum Load, Robust	Optimum Load, Unrobust
Web acess	Calico	Flannel	OVN	Flannel
MongoDB insert	Calico	Libnetwork	Calico	Libnework
MongoDB update	Weave	Weave	Weave	Weave
MongoDB query	Weave	Weave	Weave	Libnetwork

The high performing container networking solutions which have been discovered are as indicated in Table 4. The advice is to use Calico for Web access and MongoDB for insert operations, while using Weave for MongoDB updates and for query operations if robustness is required under a heavy load traffic. Peradventure robustness is not taken into account, Flannel provides the best performance for Web access operations, and Libnetwork provides a higher throughput for MongoDB insert operations under heavy load conditions. The tested networking solutions highest throughput values are obtained at the optimum traffic load points amongst the total range of traffic loads. If robustness is necessary, it is advised that OVN for Web access operations, Calico for MongoDB insert operations, and Weave for MongoDB update and query operations in optimum traffic load is used. If peradventure robustness is not necessary, the use of Flannel for Web access operations, Libnetwork for MongoDB insert and query operations in optimum traffic load is proposed. It is concluded from the evaluation presented in Table 5.24 that solution behaviours differ dynamically depending on traffic load and application type. Therefore, it is impossible to use the suggested solutions adaptively with traditional container networking implementation methods. To overcome this problem, a new smart container networking implementation design is proposed as shown in Figure 5.17.

In smart container networking architecture, containers can connect to different container networking solutions dynamically in terms of selection, depending on user defined demands which are shown in Figure 5.17 (a). A new component, SIM (Smart Container Network Interface Manager), is inserted between the container runtime and the container networking solutions as demonstrated in Figure 5.17 (b).

SIM performs the dynamic container network interface selection by utilizing the performance results gathered from container monitoring tools, to reach the required thresholds. Consequently, SIM cooperates with container monitoring tools. In addition, SIM provides an interface for users to define policies with specific thresholds regarding networking parameters such as latency, throughput, packet loss etc. for each container or group of containers. Policies are used to trigger actions by associating and prioritizing the thresholds. SIM links the containers with various networking solutions in order to evaluate their performance during the selection process. SIM then comes to a decision and connects to the proper networking solution according to defined thresholds and policies which are represented in Figure 5.17(c). As traffic load changes dynamically, a user-defined health check interval is used by SIM to compare the thresholds defined in the policies with the monitoring results. If the thresholds are crossed, a new networking selection is performed. The transition between container networking solutions should be smooth without any interruptions. If none of the solutions are able to fulfill the requirements, an alarm is created and the solution proximate to the desired demands is selected. Consequently, a dynamic networking selection is performed within this smart container networking architecture.

Figure 1.38: Smart networking architecture for CNFs



6. CONCLUSION AND FUTURE WORK

This thesis focuses on networking solutions performance from two perpectives; ordinary behaviour and behaviour according to traffic load. Initially, fundamental tests are performed within single thread to measure container networking solutions' regular behaviour by considering throughput, latency, bandwidth and reliability. Afterwards, heavy load situations has imitated with several parallel threads; performance and reliability has measured by using different workloads that are MongoDB and web access operations under different traffic loads in terms of throughput, latency, bandwidth, response times, jitter, retransmitted TCP segments and lost datagram. Practically, tested workloads are selected because of their common usage in VNF's and tested container networking solutions conformity according to workloads are clearly represented.

By evaluating all the experiments, the tested networking solutions behaviours depending on application type under specific traffic loads for CNF workloads are revealed. It is observed that none of the solutions provided relatively high throughput for all types of workload under optimum or heavy load conditions. It is discovered from the experiments that solutions are prone to specific type of workload operations. However, there is not enough entry to explain solutions' different behaviours within specific type of operations. This situation probably related with the container solutions adequacy on specific type of resource utilization. As it may not impossible to apply the suggested networking solutions adaptively with traditional container networking implementation methods, a new networking implementation design is proposed by means of SIM. SIM decides the best solution by allowing containers to use several networking solutions dynamically, depending on requirements in cooperation with container monitoring tools. Hence, this new smart container networking architecture solves the traditional container networking implementation methods' insufficiency about meeting 5G NFV implementation various operational requirements in every type of application and traffic level. The contributions of the thesis are summarized in the following section.

i. **Contribution 1: comparing hosts native performance with container networking solutions**

Baremetal hosts native performance is compared with performance of container networking solutions in terms of throughput and latency. It's observed that normally message size increment improved throghput in all data transfer methods. It's also put out that bulk TCP transfer throughputs are always higher than bulk UDP transfer throughputs which are at the same message sizes. Baremetal host outperformed than all container networking solutions as all container networking solutions have packet routing or encapsulation overheads. It is discovered that all offloading features are enabled on the docker interface docker0 by default and they can be used if they are supported by the hardware. Although hardware offloading is not enabled in pysical interfaces, it improved the container networking performance. Suprisingly, Calico's throughput within 4096 bytes message sizes passed the baremetal host throughput in TCP_CRR tests can be explained with the hardware offloading features effect of Docker. This result supports the offloading impact in container networking solutions. Besides, hardware offloading features improved the TCP performance more than UDP performance.

ii. **Contribution 2: comparing container networking solutions ordinary behaviour**

In single thread evalutions of container networking, OVN does not provide satisfied results for throughput, bandwidth and latency for TCP traffic. There may be several reasons that procreate gathared results such as using Geneve encapsulation which has bigger encapsulation header than VXLAN encapsulation. Besides, Geneve offloading is not supported in the physical network interface of hosts. Weave is the second worst performing solution after OVN with fair and low bandwidth respectively in small and large WS together with having low reliability on all window sizes. Herewith, Weave could not operate adequately compared with other tested solutions in single thread operations. Calico is the only solution which provides high throughput for TCP/UDP bulk data transfers alongside highest throghput for HTTP like connect-request-response traffic within small message sizes. Calico yields superior performance which gives low and lowest latency results for both small and large message sizes respectively

together with highest bandwidth both in small and large window sizes. Consequently, Calico is best performing solution by considering all single thread test operations. Libnetwork has maximum throughputs both in small and large message sizes for bulk UDP data transfers. However, Libnetwork gives low throughput for TCP_CRR and fair thoughput for bulk TCP transfers regardless from message sizes. Hereby, Libnetwork single thread performance is unsatisfactory compared with Calico as it has some drawbacks similar to other overlay solutions. Under single thread performance evaluation tests, Flannel bulk TCP data throughput enhances with large message sizes; meantime it provides fair throughput for bulk UDP transfers and low throughput for TCP_CRR independent from message sizes. Consequently, Flannel is not opportune in single thread performance evaluations as giving low bandwidth and high latency outputs.

iii. Contribution 3: Evaluations based on traffic load with different workloads

In multithreaded tests, OVN gives low multithreading gain for all type of workload operations. It's ranking under heavy load remained same compared with performance under optimum number of threads. Besides, OVN is unreliable compared with other tested solutions under heavy traffic load.

Weave is the only solution which provides highest thoughput with multithreading at update operations with significant difference than other tested solutions. Weave offers the high multithreading gain for all database operations however it render quite low gain for web access operations. Additionally, it is the most reliable amongst all tested solutions under heavy traffic load. The results are put forth that multithreading has improved the Weave performance.

Calico offers high throughput for insert operations, although it yields low thoughput at query, web access and update operations within multithreading. Calico give low multithreading gain for all type of workload operations. Calico could not benefitted from multithreading as much as other solutions such as Libnetwork and Weave. However, it is ranking is not much impacted under heavy traffic load.

Libnetwork gives the highest multithreading gain for all workloads except update operations. It is observed that Libnetwork relative performance has deteriorated under high traffic load as it yields the lower performances for MongoDB query and update

operations. Consequently, Libnetwork is the most beneffited solution from multithreading within optimum number of threads having the highest multithreading gain compared with single thread executions. However, it is performance most deteriorated under heavy traffic load amongst all container networking solutions. Flannel gives the low multithreading gain for all database operations while offering average multithreading gain for web access operations. Flannel is most unreliable solution in terms of jitter, lost UDP datagrams and retransmitted TCP segments. Consequently, Flannel is generally good at HTTP like traffic such as web operations, request/response transfers distinctively with multithreading even under heavy traffic load. However, it is the most inefficacious for all database operations, together with being most unrelible solution.

Consequently, It is observed that none of the solutions provided high throughput for all type of workloads at their maximum throughput or at heavy load situation. For this reason an adaptive networking architecture is required which indicates best performing solutions pursuant to application type under specific traffic load and robustness demands. As it is difficult to apply the suggested networking solutions adaptively with traditional container networking deployment methods, a new networking implementation design is proposed by means of SIM which decides the best solution by allowing containers to use several networking solutions dynamically, depending on requirements in cooperation with container monitoring tools.

6.1 AS A CONCLUSION

As a final mark of thesis, networking performance and realibility has investigated with single and multi thread operations for container networking solutions which are Libnetwork, Flannel, Calico, OVN and Weave. Performance of MongoDB and web access operations which are mostly applied VNF processes, are evaluated with solutions reliability under various traffic loads. Container networking solutions displayed quite different behaviours compared to single thread analysis outcomes. It is also discovered that solutions relative performance ranks changes excessively with respect to traffic load and application types. Hence, a new adaptive networking design has required depending on application type, traffic load and robustness requirement in order to

obtain highest performance in mostly used VNF workloads. As it is not possible to apply the suggested networking solutions adaptively with traditional container networking implementation methods, a new networking implementation design is proposed called as SIM which decides the best solution by allowing containers to use several networking solutions dynamically, depending on requirements in cooperation with container monitoring tools by meeting the operational requirements of 5G NFV implementations by considering application type and traffic level.

6.2 FUTURE RESEARCH DIRECTIVES

a. Implementing smart container networking architecture

As a future work, it is proposed that smart container networking architecture can be implemented by developing and integrating SIM with all its necessary interfaces. The SIM networking selection mechanism can also be improved in a learning mode by using artificial intelligence. In addition, container monitoring functionality can be integrated into SIM by monitoring the traffic flows concurrently, according to defined network parameters. This facilitates and accelerates the different networking solutions performance testing during the network interface selection process.

b. Enabling hardware offloading features on hosts physical network interfaces

It is suggested to enable offloading features on hosts physical NIC's to perform the tests in equal conditions during the basic performance evaluation between baremetal hosts and container multi-host networking solutions in terms of offloading as a future study. The outcomes of those tests will totally clarify the overperformed results of container networking solutions are because of hardware offloading effect or not.

c. Using Geneve aware NICs, predefined flows or controller for OVN scenario

In OVN solution, OVS has utilized as learning switch mode without any controller or predefined flows which impacts on it's performance. It has long data path at the beginning of tests, even it caused packet losses which is clearly seen at Figure 5.16 c. As a future work, OVS can be managed with an openflow controller or pre-defined flows can be inserted to the ovs-vswitchd to improve the OVN performance. Besides,

this study can be repeated with Geneve supported NIC which brings the offloading gain to OVN scenario.

d. Applying the use cases on environments builded with container orhestrator

This research is performed on physical machines without using any container orhestrator platforms. It is recommended to repeat the study in an environment created with container orhestrator such as Kubernetes, EKS (Amazon Elastic Kubernetes Service), GKE (Google Kubernetes Engine). Thence, more realistic results will be gathered for the production environements builded with container orhestrators.

REFERENCES

[1] Suo, K. et al. (2018) "An Analysis and Empirical Study of Container Networks," IEEE INFOCOM, 2018-April, pp. 189–197. doi: 10.1109/INFOCOM.2018.8485865.

[2] Zeng, H. et al. (2018) "Measurement and Evaluation for Docker Container Networking," 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2017, 2018-January, pp. 105–108. doi: 10.1109/CyberC.2017.78.

[3] Bankston, R. and Guo, J. (2018) "Performance of Container Network Technologies in Cloud Environments," IEEE International Conference on Electro Information Technology, 2018-May, pp. 277–283. doi: 10.1109/EIT.2018.8500285.

[4] Hermans, S. and Niet, P. de (2016) Docker Overlay Networks Performance Analysis in High-latency Environments.

[5] Zismer, A. (2016) Performance of Docker Overlay Networks.

[6] Abbasi, U. et al. (2019) "A Performance Comparison of Container Networking Alternatives," IEEE Network, 33(4), pp. 178–185. doi: 10.1109/mnet.2019.1800141.

[7] Kwon, J. et al. (2019) "Design and Prototyping of Container-Enabled Cluster for High Performance Data Analytics," International Conference on Information Networking, 2019-January, pp. 436–438. doi: 10.1109/ICOIN.2019.8718135

[8] Mao, C.-N. et al. (2016) "Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks," EEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015, pp. 611–616. doi: 10.1109/CloudCom.2015.67.

[9] Park, Y., Yang, H. and Kim, Y. (2018) “Performance Analysis of CNI (Container Networking Interface) Based Container Network,” 9th International Conference on Information and Communication Technology Convergence: ICT Convergence Powered by Smart Intelligence, ICTC 2018, pp. 248–250. doi: 10.1109/ICTC.2018.8539382.

[10] Buzachis, A. et al. (2018) “Towards Osmotic Computing: Analyzing Overlay Network Solutions to Optimize the Deployment of Container-Based Microservices in Fog, Edge and IoT Environments,” 2018 IEEE 2nd International Conference on Fog and Edge Computing, ICFEC 2018 - In conjunction with 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE/ACM CCGrid 2018, pp. 1–10. doi: 10.1109/CFEC.2018.8358729.

[11] Krishnakumar, R. (2019) Accelerated DPDK in Containers for Networking Nodes.

[12] Virtualization 101: What is a Hypervisor? (2013) Pluralsight.com. Available at: https://www.pluralsight.com/blog/it-ops/what-is-hypervisor (Accessed: December 8, 2019).

[13] What Is Containerization? Webopedia Definition (2019) Webopedia.com. Available at: https://www.webopedia.com/TERM/C/containerization.html (Accessed: December 8, 2019).

[14] The History of Container Technology – Linux Academy (2018) Linux Academy. Available at: https://linuxacademy.com/blog/containers/history-of-container-technology/ (Accessed: December 8, 2019).

[15] Container Journal (2019) 5 Container Alternatives to Docker - Container Journal, Container Journal. Available at: https://containerjournal.com/topics/container-ecosystems/5-container-alternatives-to-docker/ (Accessed: December 8, 2019).

[16] Julien Barbier (2015) cnm-model - Docker Blog, Docker Blog. Available at: https://www.docker.com/blog/docker-networking-takes-a-step-in-the-right-direction-2/cnm-model/ (Accessed: December 8, 2019).

[17] https://www.facebook.com/leecalcote (2016) The Container Networking Landscape: CNI from CoreOS and CNM from Docker - The New Stack, The New Stack. Available at: https://thenewstack.io/container-networking-landscape-cni-coreos-cnm-docker/ (Accessed: December 8, 2019).

[18] Docker Overview (2019) Docker Documentation. Available at: https://docs.docker.com/engine/docker-overview/ (Accessed: December 8, 2019).

[19] FreeCodeCamp.org (2016) A Beginner-Friendly Introduction to Containers, VMs and Docker, freeCodeCamp.org. freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/ (Accessed: December 8, 2019).

[20] Docker - Docker Swarm Reference Architecture: Exploring Scalable, Portable Docker Container Networks (2016) Docker.com. Available at: https://success.docker.com/article/networking (Accessed: December 8, 2019).

[21] Networking with overlay networks (2019) Docker Documentation. Available at: https://docs.docker.com/network/network-tutorial-overlay/ (Accessed: December 8, 2019).

[22] Open vSwitch (OVS) Basics (2015) Technology Focused Hub. Available at: https://network-insight.net/2015/11/open-vswitch-ovs-basics/ (Accessed: December 8, 2019).

[23] OVS Deep Dive 0: Overview (2016) Github.io. Available at: https://arthurchiao.github.io/blog/ovs-deep-dive-0-overview/ (Accessed: December 8, 2019).

[24] Multi-Host Overlay Networking with Etcd — Docker Kubernetes Lab 0.1 documentation (2016) Readthedocs.io. Available at: https://docker-k8s-lab.readthedocs.io/en/latest/docker/docker-etcd.html (Accessed: December 8, 2019).

[25] Google Groups (2019) Google.com. Available at: https://groups.google.com/forum/#!topic/coreos-user/Kl7ejtcRxbc (Accessed: December 8, 2019).

[26] Shashank Jain (2018) Flannel vs Calico : A Battle of L2 vs L3 Based Networking, Medium. Medium. Available at: https://medium.com/@jain.sm/flannel-vs-calico-a-battle-of-l2-vs-l3-based-networking-5a30cd0a3ebd (Accessed: December 8, 2019).

[27] Weaveworks (2015) Weave.works. Available at: https://www.weave.works/blog/weave-docker-networking-performance-fast-data-path/ (Accessed: December 8, 2019).

[28] Bernstein, D. J., Lange, T. and Schwabe, P. (2002) "The Security Impact of a New Cryptographic Library," International Conference on Cryptology and Information Security in Latin America (LATINCRYPT), 2(1), pp. 1–17.

[29] Letia, M., Preguiça, N. and Shapiro, M. (2015) "Consistency Without Concurrency Control in Large , Dynamic Systems," LADIS 2009 - 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, pp. 29–34. doi: 10.1145/1773912.1773921.

[30] Ordina Belgium (2018) Docker multihost networking with weave - Bas Moorkens, Github.io. Available at: https://ordina-jworks.github.io/docker/2018/09/24/docker-networking-with-weave.html (Accessed: December 8, 2019).

[31] vIns (2016) JMeter distributed load testing using Docker, Testautomationguru.com. Available at:

http://www.testautomationguru.com/jmeter-distributed-load-testing-using-docker/ (Accessed: December 28, 2019).

[32] Open Virtual Networking With Docker — Open vSwitch 2.10.90 documentation (2016) Readthedocs.io. Available at: https://ovs-dpdk-1808-merge.readthedocs.io/en/latest/howto/docker.html (Accessed: December 8, 2019).

[33] Mcleod, S. (2008) Qualitative vs. Quantitative Research, Simplypsychology.org. Simply Psychology. Available at: https://www.simplypsychology.org/qualitative-quantitative.html.

[34] WSZiB : Lectures of Prof. dr Peter Sloot (2019) Wszib.edu.pl. Available at: http://artemis.wszib.edu.pl/~sloot/1_2.html (Accessed: December 8, 2019).

[35] Netperf(1): network performance benchmark - Linux man page (2019) Die.net. Available at: http://linux.die.net/man/1/netperf (Accessed: December 8, 2019).

[36] IPERF - The Easy Tutorial (2010) Openmaniak.com. Available at: https://openmaniak.com/iperf.php (Accessed: December 8, 2019).

[37] Apache (2019) Apache/jmeter, GitHub. Available at: https://github.com/apache/jmeter (Accessed: December 8, 2019).

[38] JMeter - Open Source Functional and Load Testing (2009) Methodsandtools.com. Available at: https://www.methodsandtools.com/tools/tools.php?jmeter (Accessed: December 8, 2019).

[39] Qperf(1): Measure RDMA/IP performance - Linux man page (2019) Die.net. Available at: https://linux.die.net/man/1/qperf (Accessed: December 8, 2019).

[40] RapidLoop (2018) Measuring Network Performance in Linux with qperf - OpsDash, Opsdash.com. Available at: https://www.opsdash.com/blog/network-performance-linux.html (Accessed: December 8, 2019).

[41] Mongodb (2019) Mongodb/mongo-perf, GitHub. Available at: https://github.com/mongodb/mongo-perf (Accessed: December 8, 2019).

[42] What Is MongoDB? (2019) MongoDB. Available at: https://www.mongodb.com/what-is-mongodb (Accessed: December 8, 2019)

[43] Ruan, B. *et al.* (2016) "A Performance Study of Containers in Cloud Environment," *Advances in Services Computing - 10th Asia-Pacific Services Computing Conference, {APSCC}*, 10065, pp. 343–356. doi: 10.1007/978-3-319-49178-3.

[44] Martin, J. P., Kandasamy, A. and Chandrasekaran, K. (2018) "Exploring the support for high performance applications in the container runtime environment," *Human-centric Computing and Information Sciences*, 8(1), pp. 1–15. doi: 10.1186/s13673-017-0124-3.

[45] Casalicchio, E. and Perciballi, V. (2017) "Measuring Docker Performance: What a mess!!!∗ Emiliano," *ICPE 2017 - Companion of the 2017 ACM/SPEC International Conference on Performance Engineering*, pp. 11–16. doi: 10.1145/3053600.3053605.

[46] Herbein, S. *et al.* (2016) "Resource Management for Running HPC Applications in Container Clouds," *ecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9697, pp. v–vi. doi: 10.1007/978-3-319-41321-1.

[47] Ruiz, C., Jeanvoine, E. and Nussbaum, L. (2015) "Performance evaluation of containers for HPC," *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops Vienna, Austria*, 9523. doi: 10.1007/978-3-319-27308-2

[48] Sagar.nangare (2018) *Benefits of Containers & Microservices for Cloud Native NFV Deployment, Sagar Nangare*. Available at: http://sagarnangare.com/evaluating-containers-based-vnf-deployemnt-for-cloud-native-nfv/ (Accessed: December 17, 2019).

[49] Anderson, J. *et al.* (2016) "Performance Considerations of Network Functions Virtualization using Containers," *2016 International Conference on Computing, Networking and Communications, ICNC 2016*. doi: 10.1109/ICCNC.2016.7440668.

[50] Rotter, C. *et al.* (2016) "Using Linux Containers in Telecom Applications," *Innovations in Clouds, Internet and Networks, ICIN (2016)*, pp. 234–241.

[51] Bolivar, L. T. *et al.* (2018) "On the deployment of an open-source, 5G-aware evaluation testbed," *6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2018*, 2018-January, pp. 51–58. doi: 10.1109/MobileCloud.2018.00016.

[52] Struye, J. *et al.* (2017) "Assessing the Value of Containers for NFVs: A Detailed Network Performance Study," *2017 13th International Conference on Network and Service Management, CNSM 2017*, 2018-January, pp. 1–7. doi: 10.23919/CNSM.2017.8256024.

[53] CoreOS (2019) *CoreOS, Coreos.com*. Available at: https://coreos.com/flannel/docs/latest/ (Accessed: December 17, 2019).

[54] Era (2019) *Project Calico - Secure Networking for the Cloud Native Era, Project Calico*. Available at: https://www.projectcalico.org/ (Accessed: December 17, 2019).

[55] What's new in Calico v3.3 | Project Calico (2018) Projectcalico.org. Available at: https://www.projectcalico.org/whats-new-in-calico-v3-3/ (Accessed: December 28, 2019).

[56] AWS VPN FAQs – Amazon Web Services (2018) Amazon Web Services, Inc. Available at: https://aws.amazon.com/tr/vpn/faqs/ (Accessed: December 28, 2019).

[57] What is response time? definition and meaning (2019) BusinessDictionary.com. Available at: http://www.businessdictionary.com/definition/response-time.html (Accessed: December 28, 2019).

[58] Codecademy, N. (2019) Normalization | Codecademy, Codecademy. Available at: https://www.codecademy.com/articles/normalization (Accessed: December 28, 2019).

APPENDIXES

Appendix A.1.1 Normalized results of all single thread experiments

Performance Indicator	Libnetwork	Weave	Flannel	OVN	Calico
TCP_STREAM_Thput_S_MS	2	-12	-8	12	6
TCP_STREAM_Thput_L_MS	1	-8	1	-10	10
UDP_STREAM_Thput_S_MS	15	-15	-5	15	1
UDP_STREAM_Thput_L_MS	8	4	2	7	-10
TCP_CRR_Thput_Thput_S_MS	-9	-12	-9	-15	15
TCP_CRR_Thput_Thput_L_MS	-6	-9	-7	-10	10
TCP Bandwidth_S_WS	9	3	10	-15	15
TCP Bandwidth_L_WS	-10	-8	-1	-5	10
TCP_Bandwidth	-3	-3	-2	-5	5
UDP_Bandwidth	-5	-5	-4	-5	5
SCTP_Bandwidth	-1	-5	5	5	NA
TCP_Stream _Latency_S_MS	3	-15	-1	14	7
TCP_Stream _Latency_L_MS	-3	-2	-4	-10	10
UDP_Stream_Latency_S_MS	5	-6	-15	15	9
UDP_Stream_Latency_L_MS	2	-5	-10	10	5
TCP_Latency	5	3	3	-5	5
UDP_Latency	1	-3	-5	5	5
SCTP_Latency	0	-3	-5	5	NA
TCP_RR_Latency	3	0	3	-5	5
UDP_RR_Latency	1	-5	-2	2	5
TCP_Retrans_Seg_S_WS	-15	-3	5	9	-8
TCP_Retrans_Seg_L_WS	-9	-9	3	2	10

Appendix A.1.2 Web access operations multithreading gain

Nme	# of Threads (at Max Throughput)	Libnetwork	Weave	Flannel	OVN	Calico
Gain	100	10,08534014	4,71127406	5,694108077	5,118431193	5,070901445

Appendix A.1.3 MongoDB operations multithreading gain

Name	# of Threads (at Max Throughput)	Libnetwork	Weave	Flannel	OVN	Calico
Insert.Empty	50	19,72876028	18,39541016	16,43705341	16,82513859	17,82085206
Insert.IntVector	50	14,30082912	13,1275061	12,28647709	8,387846125	12,8363059
Insert.LargeDocVector	200	0,85289627	0,879888358	0,900816392	0,947626841	0,849474233
Update.FieldAtOffset	50	2,60074714	2,898785884	2,628561168	2,625865509	2,640270965
Queries.Empty	50	15,96075693	15,12155485	13,80011243	12,80925218	14,17504612
Queries.NoMatch	50	14,73976456	14,21758569	12,90597852	12,20739246	7,527562658
Queries.IntIDRange	50	11,2364983	11,51240266	10,4150567	9,777958564	10,30170468
Queries.FindProjection	50	12,98404002	12,5258366	11,44267825	10,7054992	11,76738665

Appendix A.1.4 Normalized results of all multithreaded experiments

Performance Indicator	Libnetwork	Weave	Flannel	OVS	Calico
MogoDB_Insert_Multi_T_Gain	5	2	-3	-4	-3
MogoDB_Query _Multi_T_Gain	18	14	-3	-14	-8
MogoDB_Update _Multi_T_Gain	-5	5	-4	-4	-4
HTTP _Multi_T_Gain	5	-5	-3	-4	-4
MongoDB_Insert_Thput_Max	10	-5	-11	-3	4
MongoDB_Insert_Thput_Heavy_Load	10	-4	-11	-2	4
MogoDB_Query _Thput_Max	13	12	-20	-10	-1
MogoDB_Query _Heavy_Load	12	18	-18	-7	-3
MogoDB_Update _Thput_Max	-4	5	-5	-4	-3
MogoDB_Update _Heavy_Load	-5	5	-3	-2	-2
HTTP_Thput_Max	-2	-5	5	-1	-2
HTTP_Thput_Heavy_Load	-3	-5	5	-1	4
TCP_RR _Thput_10T_S_MS	2	-15	7	-4	8
TCP_RR _Thput_10T_L_MS	10	-8	4	-5	-5
UDP_RR _Thput_10T_S_MS	-5	-15	-1	-7	12
UDP_RR _Thput_10T_L_MS	7	-9	10	6	-5
TCP Band_Heavy_Load	-5	-4	-3	-2	5
HTTP_Resp_Time_Heavy_Load	-5	5	5	-4	1
UDP_Jitter_Heavy_Load	2	5	4	-5	1
TCP_Retrans_Seg_Heavy_Load	-5	5	-3	-3	-2
UDP_Lost_Datag_Heavy_Load	5	3	-5	5	5

Appendix A.2 Dockerfiles which are used to build up test tools

Dockerfile for Iperf3

```
# iperf3 in a container run as server:
# docker run  -it --rm --name=iperf3-srv -p 5201:5201 networkstatic/iperf3 -s
# Run as Client (first get server IP address):
# docker inspect --format "{{ .NetworkSettings.IPAddress }}" iperf3-srv
# docker run  -it --rm networkstatic/iperf3 -c <SERVER_IP>
#
FROM ubuntu:xenial
MAINTAINER gatici
# install binary and remove cache
RUN apt-get update \
    && apt-get install -y iperf3 \
    && rm -rf /var/lib/apt/lists/*
# Expose the default iperf3 server port
EXPOSE 5201
# entrypoint allows  to pass the arguments to the container at runtime similar to a binary
which is run
# docker run -it <IMAGE> --help' is like running 'iperf3 --help'
ENTRYPOINT ["iperf3"]
```

Dockerfile for Jmeter master

```
# Use Ubuntu
FROM ubuntu:xenial
MAINTAINER gatici
# Install wget & JRE
RUN apt-get clean && \
        apt-get update && \
        apt-get -qy install \
                                wget \
                                default-jre-headless \
```

```
                                telnet \
                                iputils-ping \
                                unzip
# Install jmeter
RUN   mkdir /jmeter \
    && cd /jmeter/ \
    && wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-2.13.tgz \
    && tar -xzf apache-jmeter-2.13.tgz \
    && rm apache-jmeter-2.13.tgz \
    && mkdir /jmeter-plugins \
    && cd /jmeter-plugins/ \
    && wget https://jmeter-plugins.org/downloads/file/JMeterPlugins-ExtrasLibs-
                                        1.4.0.zip \
    && unzip -o JMeterPlugins-ExtrasLibs-1.4.0.zip -d /jmeter/apache-jmeter-2.13/
# Set Jmeter Home
ENV JMETER_HOME /jmeter/apache-jmeter-2.13/
# Add Jmeter to the Path
ENV PATH $JMETER_HOME/bin:$PATH
# Ports to be exposed from the container for JMeter Master
EXPOSE 60000
```

Dockerfile for Jmeter server/slave

```
# Use Ubuntu
FROM ubuntu:xenial
MAINTAINER gatici
# Install wget & JRE
RUN apt-get clean && \
        apt-get update && \
        apt-get -qy install \
                                wget \
                                default-jre-headless \
                                telnet \
```

```
                                iputils-ping \
                                unzip
# Install jmeter
RUN   mkdir /jmeter \
    && cd /jmeter/ \
    && wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-2.13.tgz \
    && tar -xzf apache-jmeter-2.13.tgz \
    && rm apache-jmeter-2.13.tgz \
    && mkdir /jmeter-plugins \
    && cd /jmeter-plugins/ \
    &&  wget  https://jmeter-plugins.org/downloads/file/JMeterPlugins-ExtrasLibs-
1.4.0.zip \
    && unzip -o JMeterPlugins-ExtrasLibs-1.4.0.zip -d /jmeter/apache-jmeter-2.13/
# Set Jmeter Home
ENV JMETER_HOME /jmeter/apache-jmeter-2.13/
# Add Jmeter to the Path
ENV PATH $JMETER_HOME/bin:$PATH
# Ports to be exposed from the container for JMeter Slaves/Server
EXPOSE 1099 50000
# Application to run on starting the container
ENTRYPOINT $JMETER_HOME/bin/jmeter-server \
                -Dserver.rmi.localport=50000 \
                -Dserver_port=1099
```

Dockerfile for Mongo-Perf

```
FROM debian:latest
ENV \
  BUILD_PKG='sudo make automake libtool pkg-config' \
  PERF_PKG='python-pip python-setuptools python-dev' \
  KEEP_PKG='curl git libaio-dev vim-common libmysqlclient-dev libpq-dev unzip' \
  BASE_PKG='ca-certificates apt-transport-https software-properties-common
mongodb-clients' \
```

```
RUN DEBIAN_FRONTEND=noninteractive \
  && apt-get -qq update && apt-get -qqy dist-upgrade \
  && apt-get -qqy --no-install-recommends install \
    $BUILD_PKG \
    $KEEP_PKG \
    $BASE_PKG \
  && echo '%sudo ALL=(ALL) NOPASSWD:ALL'>> /etc/sudoers \
  && cd /usr/local \
  && git clone --depth 1 https://github.com/mongodb/mongo-perf.git \
  && cd /usr/local/mongo-perf \
  && pip install -r requirements.txt \
WORKDIR /usr/local/mongo-perf/
COPY assets /assets
ENTRYPOINT [ "/assets/start" ]
```

Dockerfile for Netperf

```
FROM ubuntu:xenial
MAINTAINER gatici
#install dependencies
RUN apt-get update && \
    apt-get install -y gcc make curl && \
    apt-get clean && \
# Download Netperf
RUN curl -LO ftp://ftp.netperf.org/netperf/netperf-2.7.0.tar.gz && tar -xzf netperf-2.7.0.tar.gz
RUN cd netperf-2.7.0 && ./configure && make && make install
CMD ["/usr/local/bin/netserver", "-D"]
```

Dockerfile for Qperf

```
FROM centos:centos7
RUN yum -y install net-tools qperf
ENTRYPOINT ["qperf"]
```

Appendix A.3 Test Scripts that are used to perform test scenarios

Test scripts to initiate Jmeter tests

```
for b in {1..20}
do
    jmeter -n -t sample-test/sample-test5.jmx -R172.16.100.136   >>  test5
          jmeter -n -t sample-test/sample-test50.jmx -R172.16.100.136   >> test50
          jmeter -n -t sample-test/sample-test100.jmx -R172.16.100.136    >> test100
          jmeter -n -t sample-test/sample-test200.jmx -R172.16.100.136  >> test200
done
```

sample-test5: Jmeter test plan within 5 threads

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="3.2" jmeter="3.3.20171030">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Test Plan"
enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">false</boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
      <elementProp name="TestPlan.user_defined_variables"
elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments"
testname="User Defined Variables" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></stringProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup"
testname="Thread Group" enabled="true">
        <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
```

```
      <elementProp name="ThreadGroup.main_controller"
elementType="LoopController" guiclass="LoopControlPanel"
testclass="LoopController" testname="Loop Controller" enabled="true">
        <boolProp name="LoopController.continue_forever">false</boolProp>
        <intProp name="LoopController.loops">-1</intProp>
      </elementProp>
      <stringProp name="ThreadGroup.num_threads">5</stringProp>
      <stringProp name="ThreadGroup.ramp_time">5</stringProp>
      <longProp name="ThreadGroup.start_time">1488034056000</longProp>
      <longProp name="ThreadGroup.end_time">1488034056000</longProp>
      <boolProp name="ThreadGroup.scheduler">true</boolProp>
      <stringProp name="ThreadGroup.duration">30</stringProp>
      <stringProp name="ThreadGroup.delay"></stringProp>
      <boolProp name="ThreadGroup.delayedStart">true</boolProp>
    </ThreadGroup>
    <hashTree>
      <HTTPSamplerProxy guiclass="HttpTestSampleGui"
testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments"
guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined
Variables" enabled="true">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain">www.google.com</stringProp>
        <stringProp name="HTTPSampler.port"></stringProp>
        <stringProp name="HTTPSampler.protocol"></stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path"></stringProp>
        <stringProp name="HTTPSampler.method">GET</stringProp>
        <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
        <boolProp name="HTTPSampler.auto_redirects">false</boolProp>
        <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
```

```
        <boolProp
name="HTTPSampler.DO_MULTIPART_POST">false</boolProp>
        <stringProp name="HTTPSampler.embedded_url_re"></stringProp>
        <stringProp name="HTTPSampler.connect_timeout"></stringProp>
        <stringProp name="HTTPSampler.response_timeout"></stringProp>
       </HTTPSamplerProxy>
       <hashTree>
        <ConstantTimer guiclass="ConstantTimerGui" testclass="ConstantTimer"
testname="Constant Timer" enabled="true">
          <stringProp name="ConstantTimer.delay">300</stringProp>
        </ConstantTimer>
        <hashTree/>
       </hashTree>
     </hashTree>
   </hashTree>
   <WorkBench guiclass="WorkBenchGui" testclass="WorkBench"
testname="WorkBench" enabled="true">
     <boolProp name="WorkBench.save">true</boolProp>
   </WorkBench>
   <hashTree/>
  </hashTree>
</jmeterTestPlan>
```

sample-test50: Jmeter test plan within 50 threads

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="3.2" jmeter="3.3.20171030">
  <hashTree>
   <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Test Plan"
enabled="true">
     <stringProp name="TestPlan.comments"></stringProp>
     <boolProp name="TestPlan.functional_mode">false</boolProp>
```

```
    <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
    <elementProp name="TestPlan.user_defined_variables" elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
      <collectionProp name="Arguments.arguments"/>
    </elementProp>
    <stringProp name="TestPlan.user_define_classpath"></stringProp>
   </TestPlan>
   <hashTree>
    <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Thread Group" enabled="true">
      <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
      <elementProp name="ThreadGroup.main_controller" elementType="LoopController" guiclass="LoopControlPanel" testclass="LoopController" testname="Loop Controller" enabled="true">
        <boolProp name="LoopController.continue_forever">false</boolProp>
        <intProp name="LoopController.loops">-1</intProp>
      </elementProp>
      <stringProp name="ThreadGroup.num_threads">50</stringProp>
      <stringProp name="ThreadGroup.ramp_time">5</stringProp>
      <longProp name="ThreadGroup.start_time">1488034056000</longProp>
      <longProp name="ThreadGroup.end_time">1488034056000</longProp>
      <boolProp name="ThreadGroup.scheduler">true</boolProp>
      <stringProp name="ThreadGroup.duration">30</stringProp>
      <stringProp name="ThreadGroup.delay"></stringProp>
      <boolProp name="ThreadGroup.delayedStart">true</boolProp>
    </ThreadGroup>
    <hashTree>
      <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">
```

```
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments" guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain">www.google.com</stringProp>
        <stringProp name="HTTPSampler.port"></stringProp>
        <stringProp name="HTTPSampler.protocol"></stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path"></stringProp>
        <stringProp name="HTTPSampler.method">GET</stringProp>
        <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
        <boolProp name="HTTPSampler.auto_redirects">false</boolProp>
        <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
        <boolProp
name="HTTPSampler.DO_MULTIPART_POST">false</boolProp>
        <stringProp name="HTTPSampler.embedded_url_re"></stringProp>
        <stringProp name="HTTPSampler.connect_timeout"></stringProp>
        <stringProp name="HTTPSampler.response_timeout"></stringProp>
      </HTTPSamplerProxy>
      <hashTree>
        <ConstantTimer guiclass="ConstantTimerGui" testclass="ConstantTimer" testname="Constant Timer" enabled="true">
          <stringProp name="ConstantTimer.delay">300</stringProp>
        </ConstantTimer>
        <hashTree/>
      </hashTree>
    </hashTree>
  </hashTree>
  <WorkBench guiclass="WorkBenchGui" testclass="WorkBench" testname="WorkBench" enabled="true">
    <boolProp name="WorkBench.save">true</boolProp>
```

```
  </WorkBench>
  <hashTree/>
 </hashTree>
</jmeterTestPlan>
```

sample-test100: Jmeter test plan with 100 threads

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="3.2" jmeter="3.3.20171030">
 <hashTree>
  <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Test Plan" enabled="true">
   <stringProp name="TestPlan.comments"></stringProp>
   <boolProp name="TestPlan.functional_mode">false</boolProp>
   <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
   <elementProp name="TestPlan.user_defined_variables" elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
    <collectionProp name="Arguments.arguments"/>
   </elementProp>
   <stringProp name="TestPlan.user_define_classpath"></stringProp>
  </TestPlan>
  <hashTree>
   <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Thread Group" enabled="true">
    <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
    <elementProp name="ThreadGroup.main_controller" elementType="LoopController" guiclass="LoopControlPanel" testclass="LoopController" testname="Loop Controller" enabled="true">
     <boolProp name="LoopController.continue_forever">false</boolProp>
     <intProp name="LoopController.loops">-1</intProp>
    </elementProp>
    <stringProp name="ThreadGroup.num_threads">100</stringProp>
```

```
      <stringProp name="ThreadGroup.ramp_time">5</stringProp>
      <longProp name="ThreadGroup.start_time">1488034056000</longProp>
      <longProp name="ThreadGroup.end_time">1488034056000</longProp>
      <boolProp name="ThreadGroup.scheduler">true</boolProp>
      <stringProp name="ThreadGroup.duration">30</stringProp>
      <stringProp name="ThreadGroup.delay"></stringProp>
      <boolProp name="ThreadGroup.delayedStart">true</boolProp>
    </ThreadGroup>
    <hashTree>
      <HTTPSamplerProxy guiclass="HttpTestSampleGui"
testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments"
guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined
Variables" enabled="true">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain">www.google.com</stringProp>
        <stringProp name="HTTPSampler.port"></stringProp>
        <stringProp name="HTTPSampler.protocol"></stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path"></stringProp>
        <stringProp name="HTTPSampler.method">GET</stringProp>
        <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
        <boolProp name="HTTPSampler.auto_redirects">false</boolProp>
        <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
        <boolProp
name="HTTPSampler.DO_MULTIPART_POST">false</boolProp>
        <stringProp name="HTTPSampler.embedded_url_re"></stringProp>
        <stringProp name="HTTPSampler.connect_timeout"></stringProp>
        <stringProp name="HTTPSampler.response_timeout"></stringProp>
      </HTTPSamplerProxy>
      <hashTree>
```

```
        <ConstantTimer guiclass="ConstantTimerGui" testclass="ConstantTimer" testname="Constant Timer" enabled="true">
          <stringProp name="ConstantTimer.delay">300</stringProp>
        </ConstantTimer>
        <hashTree/>
       </hashTree>
     </hashTree>
   </hashTree>
   <WorkBench guiclass="WorkBenchGui" testclass="WorkBench" testname="WorkBench" enabled="true">
     <boolProp name="WorkBench.save">true</boolProp>
   </WorkBench>
   <hashTree/>
  </hashTree>
</jmeterTestPlan>
```

sample-test200: Jmeter test plan within 200 threads

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="3.2" jmeter="3.3.20171030">
  <hashTree>
   <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="Test Plan" enabled="true">
     <stringProp name="TestPlan.comments"></stringProp>
     <boolProp name="TestPlan.functional_mode">false</boolProp>
     <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
     <elementProp name="TestPlan.user_defined_variables" elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
       <collectionProp name="Arguments.arguments"/>
     </elementProp>
     <stringProp name="TestPlan.user_define_classpath"></stringProp>
   </TestPlan>
```

```
   <hashTree>
    <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup"
testname="Thread Group" enabled="true">
      <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
      <elementProp name="ThreadGroup.main_controller"
elementType="LoopController" guiclass="LoopControlPanel"
testclass="LoopController" testname="Loop Controller" enabled="true">
        <boolProp name="LoopController.continue_forever">false</boolProp>
        <intProp name="LoopController.loops">-1</intProp>
      </elementProp>
      <stringProp name="ThreadGroup.num_threads">200</stringProp>
      <stringProp name="ThreadGroup.ramp_time">5</stringProp>
      <longProp name="ThreadGroup.start_time">1488034056000</longProp>
      <longProp name="ThreadGroup.end_time">1488034056000</longProp>
      <boolProp name="ThreadGroup.scheduler">true</boolProp>
      <stringProp name="ThreadGroup.duration">30</stringProp>
      <stringProp name="ThreadGroup.delay"></stringProp>
      <boolProp name="ThreadGroup.delayedStart">true</boolProp>
    </ThreadGroup>
    <hashTree>
      <HTTPSamplerProxy guiclass="HttpTestSampleGui"
testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments"
guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined
Variables" enabled="true">
          <collectionProp name="Arguments.arguments"/>
        </elementProp>
        <stringProp name="HTTPSampler.domain">www.google.com</stringProp>
        <stringProp name="HTTPSampler.port"></stringProp>
        <stringProp name="HTTPSampler.protocol"></stringProp>
        <stringProp name="HTTPSampler.contentEncoding"></stringProp>
        <stringProp name="HTTPSampler.path"></stringProp>
```

```
        <stringProp name="HTTPSampler.method">GET</stringProp>
        <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
        <boolProp name="HTTPSampler.auto_redirects">false</boolProp>
        <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
        <boolProp
name="HTTPSampler.DO_MULTIPART_POST">false</boolProp>
        <stringProp name="HTTPSampler.embedded_url_re"></stringProp>
        <stringProp name="HTTPSampler.connect_timeout"></stringProp>
        <stringProp name="HTTPSampler.response_timeout"></stringProp>
       </HTTPSamplerProxy>
       <hashTree>
        <ConstantTimer guiclass="ConstantTimerGui" testclass="ConstantTimer"
testname="Constant Timer" enabled="true">
          <stringProp name="ConstantTimer.delay">300</stringProp>
        </ConstantTimer>
        <hashTree/>
       </hashTree>
     </hashTree>
   </hashTree>
   <WorkBench guiclass="WorkBenchGui" testclass="WorkBench"
testname="WorkBench" enabled="true">
     <boolProp name="WorkBench.save">true</boolProp>
   </WorkBench>
   <hashTree/>
  </hashTree>
</jmeterTestPlan>
```

Test scripts to initiate Mongo-Perf Tests

```
for b in {1..20}
do
```

```
python benchrun.py --host 172.16.100.131 --port 27017 -f testcases/simple_insert_test.js -t 1 5 50 100 200 --out insert_results
        python benchrun.py --host 172.16.100.131 --port 27017 -f testcases/simple_update_test.js -t 1 5 50 100 200 --out update_results
        python benchrun.py --host 172.16.100.131 --port 27017 -f testcases/simple_query_test.js -t 1 5 50 100 200 --out query_results
done
```

simple_insert_test.js: insert tests of Mongo-Perf

```
if ( typeof(tests) != "object" ) {
    tests = [];
}
/*
* Test: Insert empty documents into database
*/
tests.push( { name: "Insert.Empty",
            tags: ['insert','regression'],
            pre: function( collection ) { collection.drop(); },
            ops: [
                { op:  "insert",
                  doc: {} }
            ] } );
/*
 * Setup:
 * Test: Insert a vector of documents. Each document has an integer field
 * Notes: Generates the _id field on the client
 */
tests.push( { name: "Insert.IntVector",
            tags: ['insert','regression'],
            pre: function( collection ) { collection.drop(); },
            ops: [
```

```
            { op:  "insert",
              doc: docs }
        ] } );
/*
* Test: Insert a vector of large documents. Each document contains a long string
 * Notes: Generates the _id field on the client
*/
tests.push( { name: "Insert.LargeDocVector",
        tags: ['insert','regression'],
        pre: function( collection ) { collection.drop(); },
        ops: [
            { op:  "insert",
              doc: docs }
        ] } );
```

simple_query_test.js: query tests of Mongo-Perf

```
 /**
 * Creates test cases and adds them to the global testing array.
 */
function addTestCase(options) {
    var isView = true;
    var indexes = options.indexes || [];
    var tags = options.tags || [];
    tests.push({
        tags: ["query"].concat(tags),
        name: "Queries." + options.name,
        pre: collectionPopulator(
            !isView, options.nDocs, indexes, options.docs, options.collectionOptions),
        post: function(collection) {
            collection.drop();
        },
```

```
        ops: [options.op]
    });
    if (options.createViewsPassthrough !== false) {
        tests.push({
            tags: ["views", "query_identityview"].concat(tags),
            name: "Queries.IdentityView." + options.name,
            pre: collectionPopulator(
                isView, options.nDocs, indexes, options.docs, options.collectionOptions),
            post: function(view) {
                view.drop();
                var collName = view.getName() + "_BackingCollection";
                view.getDB().getCollection(collName).drop();
            },
            ops: [options.op]
        });
    }
    // Generate a test which is the aggregation equivalent of this find operation.
    tests.push({
        tags: ["agg_query_comparison"].concat(tags),
        name: "Aggregation." + options.name,
        pre: collectionPopulator(
            !isView, options.nDocs, indexes, options.docs, options.collectionOptions),
        post: function(collection) {
            collection.drop();
        },
        ops: [rewriteQueryOpAsAgg(options.op)]
    });
}

/**
 * Setup: Create a collection of documents containing only an ObjectId _id field.
  * Test: Empty query that returns all documents.
```

```
 */
addTestCase({
    name: "Empty",
    tags: ["regression"],
    // This generates documents to be inserted into the collection, resulting in 100 documents
    // with only an _id field.
    nDocs: 100,
    docs: function(i) {
        return {};
    },
    op: {op: "find", query: {}}
});
/**
* Setup: Create a collection of documents with only an ObjectID _id field.
* Test: Query for a document that doesn't exist. Scans all documents using a collection scan and  returns no documents.
 */
addTestCase({
    name: "NoMatch",
    tags: ["regression"],
    nDocs: 100,
    docs: function(i) {
        return {};
    },
    op: {op: "find", query: {nonexistent: 5}}
});
/**
 * Setup: Create a collection of documents with only an integer _id field.
* Test: Query for all documents with integer _id in the range (50,100). All threads are returning   the same documents.
 */
```

```
addTestCase({
    name: "IntIDRange",
    tags: ["regression"],
    nDocs: 4800,
    docs: function(i) {
        return {_id: i};
    },
    op: {op: "find", query: {_id: {$gt: 50, $lt: 100}}}
});
var largeArray = [];
for (var i = 0; i < 1000; i++) {
    largeArray.push(i * 2);
}
/**
 * Setup: Create a collection of documents with indexed integer field x.
* Test: Query for all the documents (empty query), and use projection to return the field
x. Each   thread accesses all the documents.
 */
addTestCase({
    name: "FindProjection",
    tags: ["regression", "indexed"],
    nDocs: 100,
    docs: function(i) {
        return {x: i};
    },
    indexes: [{x: 1}],
    op: {op: "find", query: {}, filter: {x: 1}}
});
```

simple_update_test.js: update tests of Mongo-Perf

```
/* Setup: Populate the collection with 100 documents that have 512 fields with
 *       a single character "a"
 * Test:  Each thread does two multi updates on all documents
 *       First change a_256 to "a", then to "aa"
 *       High contention on the documents as a result from the multi-updates
 */
tests.push( { name: "Update.FieldAtOffset",
            tags: ['update','regression'],
            pre: function( collection ) {
               collection.drop();
               var kFieldCount = 512;
               // Build the document and insert several copies.
               var toInsert = {};
               for (var i = 0; i < kFieldCount; i++) {
                  toInsert["a_" + i.toString()] = "a";
               }
               var docs = [];
               for (var i = 0; i < 100; i++) {
                  docs.push(toInsert);
               }
               collection.insert(docs);
               collection.getDB().getLastError();
            },
            ops: [
               { op:  "update",
                 multi: true,
                 query: {},
                 update: { $set: { "a_256": "a" } }
               },
               { op:  "update",
                 multi: true,
```

```
            query: {},
            update: { $set: { "a_256": "aa" } }
          }
        ] } );
```

Test scripts to initiate Netperf Tests

```
#!/bin/bash
#tcp_stream throughput
for a in 32 64 128 1024 4096
do
   for b in {1..20}
   do
      sudo docker exec ouy sh -c "netperf -t TCP_STREAM -f m -H 172.16.100.129
-l 10 -- -m $a " >> tcp_stream_t_$a
   done
done
#ts latency
for a in 32 64 128 1024 4096
do
   for b in {1..20}
   do
      sudo docker exec ouy sh -c "netperf -l 10 -H 172.16.100.129 -t TCP_STREAM
-- -O mean_latency -m $a " >> tcp_stream_la_$a
   done
done
#udp_stream throughput
for a in 32 64 128 1024 4096
do
   for b in {1..20}
   do
```

```
        sudo docker  exec   ouy  sh  -c "netperf -t UDP_STREAM -f m -H 172.16.100.129
-l 10 -- -m $a "  >>   udp_stream_t_$a
    done
done
#udp stream latency
for a in 32 64 128 1024 4096
do
    for b in {1..20}
    do
        sudo docker  exec  ouy  sh  -c "netperf -l 10 -H 172.16.100.129  -t UDP_STREAM
-- -O mean_latency  -m $a "  >>   udp_stream_la_$a
    done
done
#tcp_rr throughput with  10 parallel streams
for a in 32 64 128 1024 4096
do
    for c in {1..20}
    do
        sudo docker  exec   ouy  sh  -c "netperf -t TCP_RR  -f m -H 172.16.100.129   -- -
r $a,$a -b 10 -D "  >>   tcp_rr_t_$a
    done
done
#tcp rr  latency
for a in 32 64 128 1024 4096
do
    for b in {1..20}
    do
        sudo docker  exec   ouy  sh  -c "netperf -l 10 -H 172.16.100.129  -t TCP_RR  -- -
O mean_latency   "  >>   tcp_rr_la_$a
    done
done
#udp_rr throughput with  10 parallel streams
```

```
for a in 32 64 128 1024 4096
do
   for c in {1..20}
   do
      sudo docker  exec   ouy  sh  -c "netperf -t UDP_RR  -f m -H 172.16.100.129   -- -r $a,$a -b 10 "  >>   udp_rr_t_$a
   done
done

#udp rr latency
for a in 32 64 128 1024 4096
do
   for b in {1..20}
   do
      sudo docker  exec   ouy  sh  -c "netperf -l 10 -H 172.16.100.129  -t UDP_RR  -- -O mean_latency  "  >>   udp_rr_la_$a
   done
done
#tcp_crr throughput
for a in 32 64 128 1024 4096
do
   for c in {1..20}
   do
      sudo docker  exec   ouy  sh  -c "netperf -t TCP_CRR  -f m -H 172.16.100.129   -- -r $a,$a -D "  >>   tcp_crr_t_$a
   done
done
```

Test scripts to initiate Qperf Tests

```
#!/bin/bash
#tcp_bandwidth & latency
for b in {1..20}
do
     docker run -ti --net=calico-net  dockerqperf -vvs  172.16.100.134  -lp 4000 -ip 4001 tcp_bw tcp_lat  >>  tcp_bandwidth_latency
done
#udp_bandwidth_latency
for b in {1..20}
do
     docker run -ti  --net=calico-net dockerqperf -vvs  172.16.100.134  -lp 4000 -ip 4001 udp_bw udp_lat  >>  udp_bandwidth_latency
done
#sctp_bandwidth_latency
for b in {1..20}
do
     docker run -ti --net=calico-net  dockerqperf -vvs  172.16.100.134  -lp 4000 -ip 4001 sctp_bw sctp_lat  >>  sctp_bandwidth_latency
done
```

Test scripts to initiate Iperf3 Tests

```
#!/bin/bash
#5203  server  port
#tcp_bandwidth_according to window_size
for b in {1..20}
do
    iperf3 -c  172.16.100.132 -p 5203 -w  8KB  >>  tcp_bandwidth_window_size_8
         iperf3  -c  172.16.100.132  -p  5203  -w  16KB  >> tcp_bandwidth_window_size_16
         iperf3  -c  172.16.100.132  -p  5203  -w  32KB  >> tcp_bandwidth_window_size_32
```

```
        iperf3 -c 172.16.100.132 -p 5203 -w 64KB >> tcp_bandwidth_window_size_64
        iperf3 -c 172.16.100.132 -p 5203 -w 128KB >> tcp_bandwidth_window_size_128
done
#udp_bandwith
for b in {1..20}
do
    iperf3 --udp -c 172.16.100.132 -p 5203 >> udp_bandwith
done
#tcp_bandwidth with_multithreading
for a in 5 10 30 100
do
   for b in {1..20}
   do
       iperf3 -c 172.16.100.132 -p 5203 -P $a >> tcp_bandwidth_paralel_$a
   done
done
#udp_bandwidth_ with_multithreading
for a in 5 10 30 100
do
   for b in {1..20}
   do
       iperf3 --udp -c 172.16.100.132 -p 5203 -P $a >> udp_bandwidth_paralel_$a
   done
done
```

CURRICULUM VITAE

Name Surname:	Gülsüm ATICI
Address:	Zümrütevler Mah. Sinem Sok. Ravza Sitesi No: 20 A Blok Daire: 13 Maltepe / İSTANBUL
Date and Place of Birth:	28.02.1988 MUĞLA
Languages:	Turkish (native), English (fluent)
Elementary Education:	Ekinanbari Elementary School, 1998
Secondary Education:	Milas Menteşe Secondary School, 2001
High School:	Milas Anatolian High School, 2005
B. S.:	Istanbul Technical University, 2010
M.S.:	Bahçeşehir University, 2020
Institute:	The Graduate School of Natural and Applied Sciences
Program:	Computer Engineering
Work Experience:	Ulak Haberleşme, *Cloud DevOps Engineer*, (Istanbul, 2018 – continue)
	Merkezi Kayıt İstanbul, *Linux System Engineer*, (Istanbul, 2015 – 2018)
	Huawei Turkey, *Linux System Engineer,* (Istanbul, 2010 – 2015)
	Orbitel Telecommunication, *Telecommunication Engineer,* (İstanbul, 2009-2010)

Publications:

iii. **Atici, G. and Boluk, P.** (2020) "TCP Pencere Boyutuna Göre Konteyner Küme Ağ Çözümlerinin Performans Analizi Container Cluster Networking Performance Analysis According to TCP Window Size," *IEEE Conference on Signal Processing and Communications Applications 2020*, pp. 1–4.

iv. **Atici, G. and Boluk, P.** (2020) “A Performance Analysis of Container Cluster Networking Alternatives,” *The 2nd International Conference On Industrial Control Network And System Engineering Research, ICNSER2020*, pp. 1–8.