Mousa Ryad Al Naser

Experimental and Analytical Analysis of a Virtualized Network Function

Master's thesis in Communication Technology
Supervisor: Yuming Jiang, Besmir Tola
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

NTNU
Kunnskap for en bedre verden

Title: Experimental and Analytical Analysis of a Virtualized Network Function
Student: Mousa Ryad Al Naser

Problem description: Current legacy networks are over populated with a huge number of hardware appliances. Furthermore, deploying a new Network Function (NF) often requires installing yet another variety of proprietary hardware. This is compounded by the increasing costs of energy and capital investment. The concept of Network Function Virtualization (NFV) originated from the requirement of the service providers to enhance performance, reduce deployment and operating cost, foster competition, and improve scalability and manageability. NFV takes advantage of cloud computing and virtualization technology to accelerate the deployment of Virtualized Network Functions (VNFs). NFV promises to bring a multitude of benefits to the network field as it decouples software from hardware which means that the implementation of both hardware and software is no longer dependent on each other. In other words, NFV promotes the implementation of network functions in software instead of installing new dedicated hardware every time a new network function is added. The service provider can simply launch a new VNF which can run on a commodity hardware and can be managed without the need of modifying the physical infrastructure. Since many VNFs prototype are available to be studied and analyzed, it is equally important to identify their performance, and ability to meet the user expectations.

In this project, we study the performance of VNF in the context of a virtualized IP Multimedia Subsystem (IMS), called ClearWater, which is an open source NFV based IMS. Clearwater follows IMS architectural principles and supports all of the services, and standardized interfaces expected of an IMS. ClearWater is available to be deployed in two widely used virtualization technologies: an open source Linux-based container (Docker), and Virtual Machine (VM). Similar to typical IMS, ClearWater utilizes Session Initiation Protocol (SIP) for call session establishment and control. SIP is a signalling protocol used for managing multimedia sessions over an IP data network, and negotiating the parameters of the created session.

To this end, we consider a common deployment where all the VNF services are running on the same physical machine. The main goal is to study and analyze the initial user registration process of ClearWater for both deployment options, and to derive an analytical model based on queuing networks. The experimental results will form the basis to eventually refine and validate the analytic model. The project aims to discover the influence of different system features like resource allocation, service request arrival rates, number of processing instances on the system performance in terms of response time, successful and failure rate, and resource utilization.

Responsible professor: Yuming Jiang, IIK
Supervisor: Besmir Tola, IIK

Abstract

NFV is an emerging technology which aims to improve the performance, foster competition, and increase innovation. NFV decouples software from hardware which means that the implementation of both hardware and software is no longer dependent on each other. In case of NFV, The deployed VNFs share a common standard infrastructure, and they can be modified and managed without the need of modifying the physical infrastructure.

In this project, we study the performance of an open-source widely-referenced VNF prototype, namely ClearWater. The aim is to investigate how this new technology comply to the expectations. The system will be deployed on top of two widely used virtualization technologies: Linux-based container (Docker), and VMs. We mainly focus on the initial subscriber registration process. In addition, we monitor the Central Processing Unit (CPU) and Random Access Memory (RAM) utilization. The project aims to investigate the influence of different system features like resource allocation, arrival rates, and number of processing instances on the system performance in terms of response time, and resource utilization.

The project also includes deriving an analytical model based on the queuing network theory. We consider two well-known methods to derive the analytical model: M/M/1, M/G/1, and G/G/1. The experimental results will form the basis to eventually refine and validate the analytic model.

Preface

The basis for this project stemmed from the idea that I could use the topic of the NFV to work on a project that would not only encompass my academic proficiency but also provide benefits to the ongoing research within the area. Through this project, I was able to learn new skills from different resources whilst generating an outcome that provides a well rounded picture of the research surrounding NFV and generating a better idea of how good the performance of the new technology is. My goal was to get a better knowledge about the NFV technology, and how software implementation of the network functions operates.

In truth, I could not have achieved my current level of success without a strong support group. First, My family, who gave me the love and the unconditional support all through the way. And secondly, I would like to thank Professor Yuming Jiang and PhD student Besmir Tola for their academic support and contributions to my academic and even personal development. Thirdly, a great gratitude to all at the department of Information Security and Communication Technology and my friends. Your help was instrumental to improve the quality of out work and also contributed to my personal development.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

λ	Arrival process rate.
μ	Service process rate.
$1/\lambda$	Inter-arrival time.
$1/\mu, E[X]$	Expected Service time.
ρ	Utilization factor.
σ	Standard deviation.
σ_s	Standard deviation for service time.
C^2	Coefficient of variation.
C_s^2	Coefficient of variation for service time.
$E[S]$	Expected Sojourn time.
$E[W]$	Expected Waiting time.

List of Acronyms

AA Affinity Aggregates.

AIC Akaike Information Criterion.

AS Application Server.

AV Authentication Vector.

CAGR Compound Annual Growth Rate.

CPU Central Processing Unit.

CSCF Call Session Control Functions.

CSV Comma-Separated Values.

DNS Domain Name System.

EB Exabytes.

EMS Element Management System.

EPC Evolved Packet Core.

ETSI European Telecommunications Standards Institute.

FCFS First Come First Served.

GB Gigabytes.

GSL GNU Scientific Library.

GUI Graphical User Interface.

HSS Home Subscriber Server.

HTTP Hypertext Transfer Protocol.

I-CSCF Interrogating Call Session Control Functions.

IDS Intrusion Detection System.

iFC initial filter criteria.

IMS IP Multimedia Subsystem.

IoT Internet of Things.

IP Internet Protocol.

LTE Long Term Evolution.

LTS Long Term Support.

MANO NFV Management and Orchestration.

MME Mobility Management Entity.

MRF Media Resource Functions.

NAS Network-Attached Storage.

NAT Network Address Translation.

NF Network Function.

NFV Network Function Virtualization.

NFV ISG Network Functions Virtualisation Industry Specification Group.

NFVI NFV infrastructure.

NFVO NFV Orchestrator.

NGN Next Generation Networks.

OSS/BSS Operations and Business Support System.

P-CSCF Proxy Call Session Control Functions.

PSTN Public switched telephone network.

RAM Random Access Memory.

RQT Robust Queuing Theory.

S-CSCF Serving Call Session Control Functions.

SDP Session Description Protocol.

SFC Service Function Chaining.

SIP Session Initiation Protocol.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

VIM Virtualized Infrastructure Manager.

vIMS virtualized IMS.

VM Virtual Machine.

VNF Virtualized Network Function.

VNFM VNF Manager.

XDMS XML Document Management Server.

XML eXtensible Markup Language.

Chapter 1
Introduction

1.1 Introduction

Today's world is increasingly digitalized and people are becoming more and more connected. It is not only humans who are connected anymore, the Internet of Things (IoT) devices are demanding new connectivity requirements. The service providers, who are in charge of providing such connectivity and services, are facing huge challenges to keep providing highly performing and stable services while the traffic volume and the number of subscribers on their networks are experiencing an exponential rise. According to Ericsson Mobility Report of 2018, the generated traffic from mobile devices is anticipated to witness an increase of 500 % by the end of 2024 reaching a data volume of 136 Exabytes (EB) per month. Furthermore, it is expected to have 8.9 billion mobile subscriptions and 8.4 billion mobile broadband subscriptions by 2024 [Jej18].

The increasing number of mobile and broadband subscriptions makes it essential to adapt a new technology which aims to overcome the limitations of legacy networks with respect to scalability, cost, resources management, and complexity of the network infrastructure. In sake of improving the network performance, some of the dominant service providers among the world, who extensively use NFs like Network Address Translation (NAT) or Firewall, in cooperation with European Telecommunications Standards Institute (ETSI), introduced the concept of NFV for the first time in October 2012. Since that day, ETSI has established an open membership institution specialized in NFV called Network Functions Virtualisation Industry Specification Group (NFV ISG), which drives innovation and standardization of NFV technology [ETS13b].

NFV has changed the approach of deploying network functions to be virtualized instead of using specific property hardware and software. NFV promises to bring a multitude of benefits to the network field as it decouples software from hardware which means that the implementation of both hardware and software is no longer

dependent on each other. Separating software from hardware improves sharing of the resources and makes deploying of new services much faster. NFV is expected to significantly reduce costs related to network operation, and increase providers' profit. This is done by reducing the complexity of hardware infrastructure and sharing of resources among several VNFs [ETS13a]. Instead of installing new dedicated hardware every time a new network function is added, the service provider can simply set up a new virtual function to provide the desired service.

Despite the fact that NFV was conceived a few years back, a recently published report shows that it is expected for the NFV market to witness a grow at a Compound Annual Growth Rate (CAGR) of 32.88 % within the period 2016-2022 [Rep]. NFV has proven itself to be superior not only with regard to cost saving, simplicity of deploying, and manageability of services, but also a reliable technology with high level of performance.

The high-level architecture of NFV framework, as defined by ETSI, is shown in Figure 1.1. It consists of four main components: NFV infrastructure (NFVI), VNFs, Operations and Business Support System (OSS/BSS), and NFV Management and Orchestration (MANO).

First, NFVI is the layer that contains standard hardware and virtual resources which form the foundation of the NFV environment. NFVI itself consists of three parts:

1. Hardware resources: consist of computing hardware such as servers, storage hardware such as Network-Attached Storage (NAS), and network hardware such as switches.

2. Virtualization layer: abstracts hardware resources and detaches software from hardware which enables software to be run independently from hardware.

3. Virtualized resources: includes virtual computing, storage, and network resources which are exploited by the VNFs.

Those shared resources are used by VNF for its processing and connectivity needs. The virtualization layer is responsible for decoupling the hardware resources from the VNFs so that multiple software can be implemented across the hardware resources.

Second, the VNF part represents the software implementation of various NFs. It is possible to combine several VNFs together in order to achieve more specialized function, known as Service Function Chaining (SFC), aiming at increasing scalability and network agility. An Element Management System (EMS) performs management

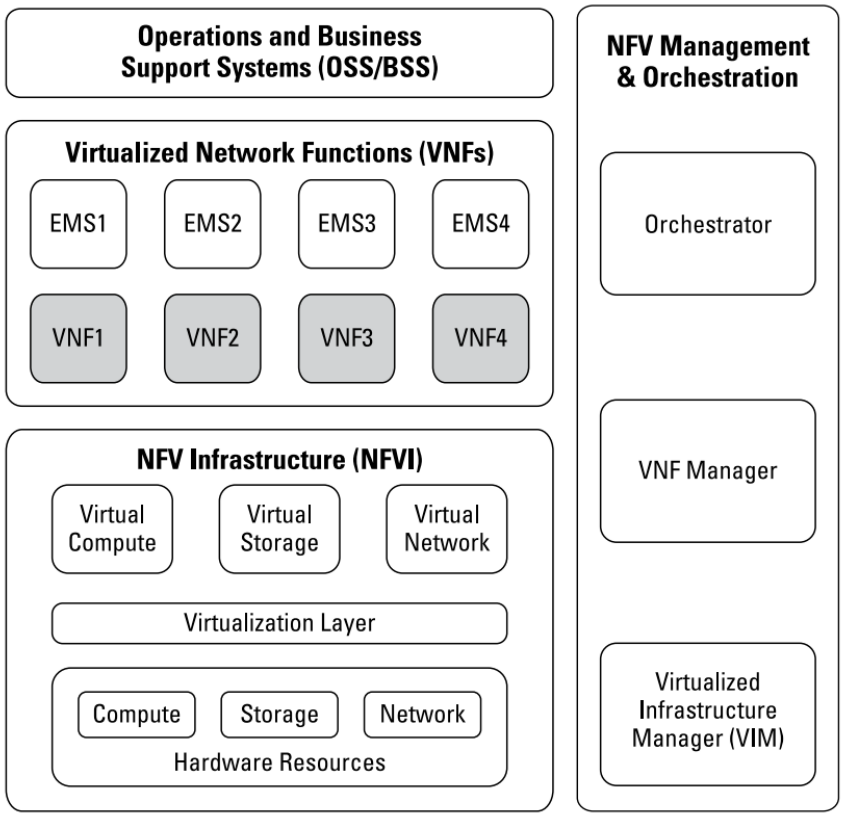


Figure 1.1: NFV high-level Architectural Framework, adapted from [ETS13a]

for network elements. It assists to take action on fault and performance information, and it handles the security aspects of the network element.

Third, the NFV MANO functional block consists of the following parts:

1. Virtualized Infrastructure Manager (VIM): is used to manage and monitor the connections of VNFs with the available resources.
2. VNF Manager (VNFM): is in charge of the VNF lifecycle. It initializes, updates, scales, and terminates VNF instances.
3. NFV Orchestrator (NFVO): is responsible for the network services lifecycle.

Finally, the OSS/BSS performs billing, fault, element, configuration, and operations management.

NFV is expected to be widely deployed and used in a cloud computing fashion aiming at providing and supporting telecommunication services. An IMS, being an overlay architecture of Long Term Evolution (LTE) networks, provides multimedia signaling packets through multimedia servers, and multimedia data packets through multimedia gateways. It represents a core functionality of Evolved Packet Core (EPC) architecture by connecting user equipment with targeted multimedia applications which provide services like audio/video over Internet Protocol (IP) networks. As identified by ETSI [NC13], the IMS represents an excellent use case of applying NFV concepts into LTE and 5G mobile technologies. However, 5G services are expected to provide very high data rates therefore demanding very strict performance requirements.

In this project, we study an NFV-enabled service by assessing the performance of an open source and widely referenced VNF which implements a virtualized IMS (vIMS), namely Clearwater [Neta] [Netb]. ClearWater has several typical deployment options using virtualization technologies like VM and Linux-based Docker containers (see chapter 3 for more details about ClearWater, VM, and Docker containers). The NFV specifications do not recommend a particular virtualization layer solution for the NFV infrastructure. Therefore, performing the test for both virtualization technologies will help to assess which one is more convenient to be used with Clearwater.

The aim of this project is to analyze the performance of such a VNF when it is deployed on the two different options aiming at investigating the following research questions:

- Which of the both approaches acts better in terms of average registration delay?
- How they are compared to each other and how resource allocation and scaling impact the delay?
- How the performance of ClearWater elements is influenced when varying the arrival rate?

This project is structured from two main parts. The first part is to set up the experimental environment, and conduct the experimental measurements. The second part consists of deriving an analytical model based on queuing network theory and validating it based on the experimental results. With regard to the problem description, we excluded monitoring the successful and failure rate of the subscriber registration due to the intensive workload. We would rather focus on the average registration delay, influence of the resource allocation, different arrival rates, and several deployment options.

Chapter 2
Background and Related Works

This chapter aims to present some background on the chosen VNF software architecture and provide the reader with a general overview of what related references have investigated so far.

2.1 Background

On the core of the IMS architecture there is the SIP signalling protocol. The SIP, developed by ETSI, is used for managing multimedia sessions over an IP data network, and negotiating the parameters of the created session. SIP can operate on top of Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), and it supports the following features when establishing and terminating multimedia sessions:

1. User location: regardless of their locations and devices, users can connect to the network using the same identifier.
2. User availability: determine the willingness of the recipient to engage in communications.
3. User capabilities: parameters to be utilized during a session.
4. Session setup and management.

A session description is used to characterize the capabilities of a session. An INVITE request is used to initiate a SIP connection, and BYE message ends the established session. Option message includes information about the capabilities of the participated users, and Status message updates other servers about the progress of signaling actions when it is requested. ACK message is used to inform a successful message exchange.

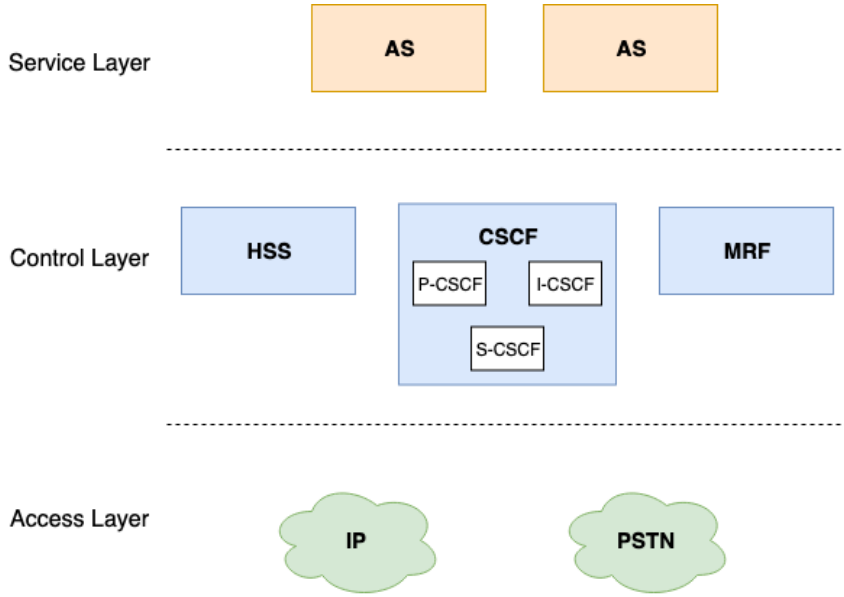


Figure 2.1: IMS High-level Architecture

SIP is a request-response protocol which means when a SIP endpoint received a request from a certain node, it sends back a response to the initiator. All the SIP responses have a response code and message. In total, there are six different types of response codes that start from 100 up to 600. Responses with a code from 1xx group indicate the statue of call progress. They are followed by other SIP messages to inform other nodes participated in the created session about the final outcome. Those message can be 2xx, 3xx, 4xx, 5xx, and 6xx for successful, redirection of the connection, client error , server error, and global failure respectively.

In order to understand how ClearWater operates and provides services, it is essential to realize the traditional IMS architecture shown in Figure 2.1. Generally, IMS consists of three main layers:

1. Access network layer: provides connectivity to external networks. It contains IP routers and Public switched telephone network (PSTN) switches.

2. Control layer: in charge of controlling calls establishment, management, and termination. It consists of four different parts: Home Subscriber Server (HSS), Call Session Control Functions (CSCF), and Media Resource Functions (MRF).

3. Service layer: consists of multiple Application Servers (ASs) which host and execute services.

CSCF is a significant part in IMS system which needs to be presented more in detail. CSCF of traditional IMS has three main types: Proxy Call Session Control Functions (P-CSCF), Interrogating Call Session Control Functions (I-CSCF), and Serving Call Session Control Functions (S-CSCF). First, P-CSCF acts as the anchor point between the client and the IMS network. The P-CSCFis a SIP proxy which provides integrity and confidentiality for SIP signalling and confirms identity of the client to other nodes of IMS. Second, the I-CSCF plays a key role in IMS roaming. Its main responsibilities is to retrieve user location information from HSS, assign a S-CSCF based on the received information, and route arriving requests to the correct S-CSCF. Lastly, the S-CSCF acts as a registrar server. It is the main control node in the signalling plane. The S-CSCF is the most loaded node of the IMS due to the initial filter criteria (iFC) processing which enables IMS service management. The S-CSCF provides interface to HSS in order to download user's service profile. It also enforces policies of network operators.

In this project, We use an open source vIMS, namely ClearWater. The architecture of traditional IMS and ClearWater is almost the same, but they have different terminologies. In case of ClearWater, Sprout element is equivalent to functionality of both I-CSCF and S-CSCF. P-CSCF function is implemented in Bono. See Chapter 3 for more details about ClearWater and its architecture.

2.2 Related Works

Several studies have been performed with the focus of exploring and assessing virtualization technologies for NFV-enabled architectures. In [BCC+15], the authors deployed chains of NFVs on a single server where the virtualization environment was container and VM. They also defined the bottleneck scenarios. Open vSwitches were used to control the traffic between different NFVs. They measured both latency and throughput for packets of various sizes and concluded that container provides performance close to VM, utilize less resources, and has better latency. In this project, we consider to both Docker container and VM since they provide different values of latency.

Another research [CW18] was performed to explore performance of an altered IMS architecture. The S-CSCF was split into two parts, one of them controlling call establishment and the other being in charge of user registration. Whereas, the I-CSCF functionality was combined with the part of S-CSCF that performs the user registration. They mainly measured user registration success rate and call flow success rate for the new IMS, original ClearWater with only one Sprout, and ClearWater with two Sprout elements. The registration success rate for ClearWater with a single Sprout node rapidly declines. The registration success rates of the

others deployment gradually decline at 2500 test run. They concluded that the new IMS has the best performance for user registration and call flow.

In [SSFS17], micro-service bundles, called Affinity Aggregatess (AAs), were suggested to be applied to ClearWater, and they were chosen to combine Sprout and Bono together. ClearWater was deployed on a Docker container on a single machine where all existed containers share the available resources. They used SIPp to generate SIP traffic aiming at monitoring the number of failures for each type of messages and investigating the effect of network latency on ClearWater.

In [NSV17], the authors designed a tool, called VNFPerf, to detect bottleneck and monitor the performance of VNFs. VNFPerf consists of local capturing unit, which is installed on all machines, and central analysis module which receives VNFs attributes from local capturing modules. Local capturing module sniffs all incoming and outcoming traffic and send regular reports to central analysis module. The authors found out that VNFPerf provides an accurate performance analysis and can efficiently discover bottlenecks in real-time networks.

With regard to analytical modelling of NFV, queuing network analyzer theory was applied to derive a mathematical model of VNF in order to estimate the system response time [PAR+17]. The authors made no assumptions regarding the arrival time or the service time. Each node was considered as G/G/m queue. They modeled the signaling procedure of Mobility Management Entity (MME) following a three-tiered architecture. It was assumed that each element serves the packets following a First Come First Served (FCFS) scheduling scheme. On the contrary to our model which will be validated based on the experimental results, the final model was validated by simulation.

Another related work is [SSJ19] where the authors compared tow popular approaches of deploying VNFs: monolithic and microservice. The comparison of performance included a mathematical analysis and experimental measurements. They also considered to study the impact of scaling up and decomposition on the performance. SNORT, which is an open source Intrusion Detection System (IDS), was used to conduct the experimental measurements. Monoliths and microservices of SNORT were deployed on Docker containers. With regard to the analytical model, they considered M/M/1 to derive the analytical model. It was concluded that scaling up monolithic VNFs has a better performance rather than decomposition using the microservice approach. This work differs from ours in the nature and complexity of the studied VNFs. In addition, we explore and compare two popular virtualization technologies: Docker containers and VMs. Finally, we consider several approaches to derive the analytical model like M/M/1 and G/G/1.

In [MG10], the authors analyzed the registration delay for tow different systems:

3G and WiMax. They also performed an analytical analysis by using M/M/1 and G/G/1. Multiple types of delays were considered like transmission delay, queuing delay, and processing delay. They concluded that WiMax networks provides better delay compared to 3G. Several papers [Mun08][NTN13] have justified that M/M/1 is a valid method to mathematically describe the IMSs processes. It is quite common with regard to modeling the IMS through queuing systems, the network is abstracted as a tandem of queues where each queue is modeled as an M/M/1 [WBBD05, FCP06].

The last related work is concerned in modelling of IMS. The author abstracted the queuing network as tandem of queues. They considered both M/M/1 and M/G/1. The validation of the analytical results was performed with simulation.

In this project, we study and analyze the user registration process of ClearWater in two different virtualization technologies (Docker and VM) and investigate the influence of different system features like resource allocation, service request arrival rates, and number of processing elements on the overall system performance. In addition, the project includes deriving an analytical model using classical queuing theory where both the arrival and service processes can be either Markovian or general processes.

Chapter 3
Methodology

This chapter introduces the methodology applied for evaluation and analysis of the Clearwater IMS network function for the different virtualization technologies and resource configurations. It also presents the approach of deriving an analytical model which is suitable for characterizing and describing performance-wise the system architecture.

The overall process consists of two main parts. The first part is to set up the experimental environment and conduct the necessary measurements when performing various tuning of the resource allocations, the incoming traffic load, and other related testbed specifications. The aim is to analysis the user registration process of ClearWater which will be deployed on top of Docker containers and VMs. The experimental work was run several times with different settings like generating varying number of service requests per second and changing the allocated resources for key Clearwater components. The second part is to derive an analytic model based on queuing network theory as a reference method. Through the experimental results we investigate the suitability of the queuing models by comparing the experimental measurements with the expected performance results given by each assumed queue model. For each model, we derive the relative error and identify the validity of it.

3.1 Used Tools

The purpose of this section is to introduce in details the tools utilized in setting up the experimental environment and justify their selection.

3.1.1 ClearWater

In an NFV architecture, a VNF is the software implementation of a specific network function and ETSI has identified the virtualization of an IMS as one of the potential use cases where the expected benefits of NFV can be fully exploited [ETS13c]. A widely adopted and referenced ETSI-compliant vIMS is developed by the Clearwater

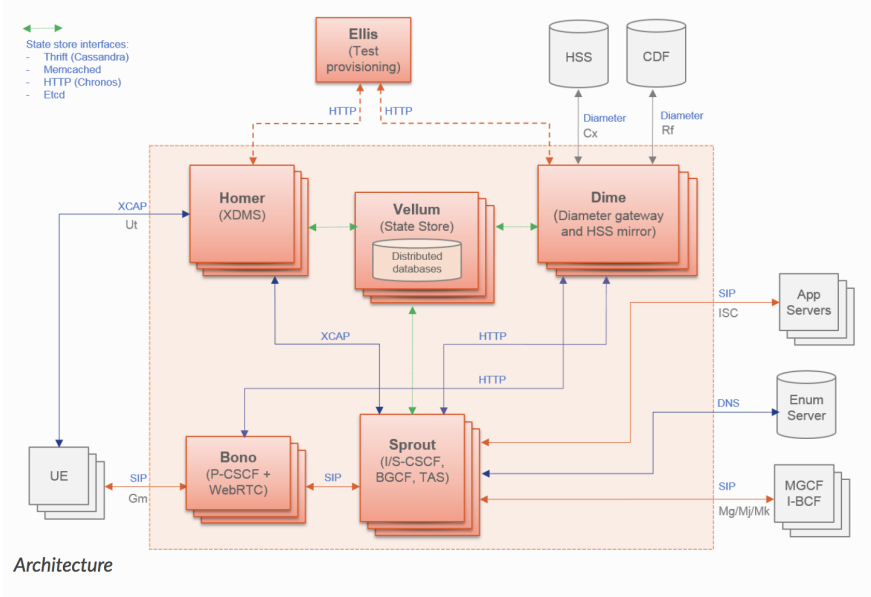


Figure 3.1: ClearWater Architecture, adapted from [Neta]

project [Neta]. ClearWater is an open-source NFV based IMS which enables IMS and its associated services to be deployed in a cloud environment. Similar to typical IMS, ClearWater utilizes SIP signaling for voice, video, and messaging services.

Figure 3.1 shows the architecture of ClearWater which consists of the following parts:

1. Sprout: acts as a SIP router and registrar, performs user authentication, and it supplies interface to another node called Vellum in order to save registration data. Since it is possible to perform load balancing across the Sprout cluster, a long time association between a subscriber and specific Sprout node is not used. Sprout provides interface to Homer and HSS to retrieve user information and service profile. Sprout performs the functionality of both I-CSCF and S-CSCF of typical IMS.

2. Bono: the edge point that users contact in order to get authenticated and reach other system's nodes. It acts as a SIP proxy and provides confidentiality and integrity for SIP. Bono is equivalent to the P-CSCF of IMS.

3. Dime: consists of the following components:

   a) Homestead: provides an interface to Sprout to exchange the authentication credentials and subscriber profile information.

   b) Homestead Prov: facilitate provisioning of subscribers in Cassandra on Vellum.
   c) Ralf: is responsible for the billing service.

4. Homer: is a XML Document Management Server (XDMS) used to store service settings documents for each subscriber in the system.
5. Vellum: is used to store all long lived state in ClearWater. It consists of Cassandra, Chronos, and Astaire:
   a) Cassandra: is used by Homestead to save user information like authentication credentials and profile information.
   b) Chronos: developed by ClearWater to provide timing service. Both Sprout and Ralf make use of Chronos to initiate timers to be run.

ClearWater has several features which facilitate the work progress. For example, it has two options to create subscribers' accounts either from Graphical User Interface (GUI) through Ellis, or from any Cassandra node by executing a bulk-provision script which can be used to create thousands of subscribers' accounts at once. Furthermore, ClearWater is well-documented, and it has a mailing list where you can drop your issue or can simply look how others solved any eventual deployment/operation problems you are facing. In addition, ClearWater is horizontally scalable which allows to create multiple instances of each node if needed. In this project, we consider scaling up of the Sprout component and analyze how this will influence the performance with regard to the registration phase and resource utilization.

The last released version of ClearWater, called Zamîn, was used in this project. All ClearWater's nodes were installed on a single physical machine running Linux Ubuntu operating system.

3.1.2 VM and Docker container

ClearWater has several deployment options. Typical deployments regard public or private infrastructures and they rely on two common virtualization technologies, hypervisor-based VMs and Linux containers. Such environments can be deployed on both private and public cloud infrastructures. We considered both Docker container and VM, and deployed the ClearWater software stack on our own hardware infrastructure.

A VM is a virtualized environment which enables the users to emulate several operating systems and applications on one physical machine. This is done by a Hypervisor, i.e., server virtualization software, running on top of a hardware infrastructure which allows multiple virtual machines to run on the same physical

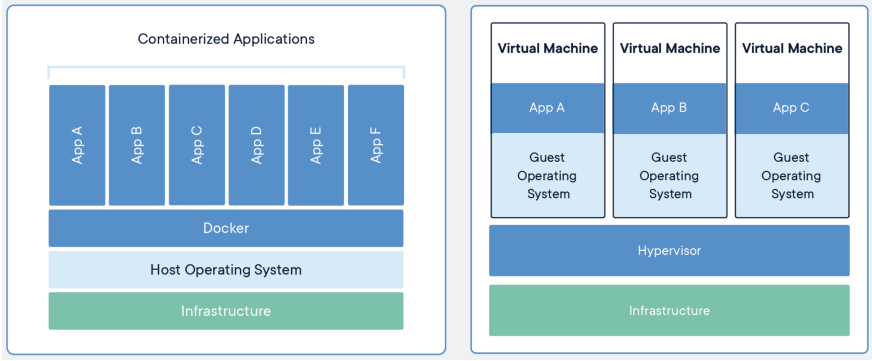


Figure 3.2: Container vs VM, adapted from [Doc18]

device. Each of the VMs, exploits virtualized resources which are mapped with physical ones through the Hypervisor.

Docker containers are an open source software which enables a code and all its dependencies to be executed quickly. It is a lightweight virtualized environment which has some promising features like low memory-foot print and booting times compared to VMs (few seconds to initialize).

Figure 3.2 shows the main differences between Linux-based containers and VMs. In terms of a VM-based virtualization, every virtual machine needs its own operating system which requires more dedicated resources and may greatly impact the booting process of the deployed VMs. A container-based virtualization is similar to the hypervisor-based with regard to having several containers running on the same host, but those containers share a unified operating system which reduces storage utilization, hence obtaining a much lower memory footprint, and an increased time efficiency by having faster boot up times compared to standard VMs. In few words, Linux-based containers are more efficient lightweight virtualized environments making them very suitable in dynamic architectures like cloud computing [Doc18].

With regard to the resource allocation, every VM has a dedicated amount of resources which means there is no flexibility in managing the available resources according to the workload on each VM. In contrast to VMs, Docker containers allow dynamic resources sharing where the resources allocated to the containers according to the workload on each of them. The resources can be limited to each container by executing the following command:

The memory flag can be set to the desired amount of RAM like 2048m which will assign 2 Gigabytes (GB) of RAM. The second flag is used to determine which cores will be dedicated to that container. It is possible to assign multiple cores to

Algorithm 3.1 Limiting a Container's Resources

```
1- sudo docker update --memory <memory_value> --cpuset-cpus <the_
core_number> --net <network_ID> <container_ID>
2- sudo docker network ls
3- sudo docker container ls Or sudo docker ps
```

a certain container. The network ID could be necessary if it is desired to change the assigned IP address for a certain container to be in the same network as other containers. A list of networks IDs can be shown by executing the second command in 3.1. The container ID can be fetched by running one of the third commands.

3.1.3 SIPp

SIPp [sip] is a free source SIP traffic generator with built-in default scenario files developed by Intel Corporation in accordance with the IMS/Next Generation Networks (NGN) Performance Benchmark specification. Some common built-in scenario files include the following:

1. Registration
2. De-registration
3. Re-registration
4. Successful call
5. Successful messaging

SIPp is used to generate the workload during the experimental tests. Each SIP session includes a sequence requests for registering the subscribers. The detailed messages' sequence are explained under section 3.2.

3.1.4 Wireshark

Wireshark is a traffic capture and network analyzer. It allows the client to perform deep packet inspection, filter the results to focus on the desired traffic or protocol, and export the traffic to eXtensible Markup Language (XML), Comma-Separated Values (CSV), or plain text files. Before sending Register requests, we started Wireshark on the physical machine listening to either VM or Docker container interface. The captured traffic was analyzed afterwards to estimate the actual service time of each

Table 3.1: CPUand RAM Monitoring Tools

Resource	**Docker container**	**VM**
CPU	docker stats, mpstat	mpstat
RAM	docker stats	sar

involved component during the registration process. Wireshark can be installed on Linux by executing the following command:

```
sudo apt install wireshark
```

3.1.5 CPU and RAM Monitoring Tools

The following parameters were observed during the experimental work:

1. Total average delay of succeeded requests.
2. Average CPU usage.
3. Average RAM usage.

Those information assist to characterize the main aspects of the system:

1. Capacity of the system.
2. Resources consumption.

There are plenty of available tools which can be utilized to monitor the CPU and RAM usage. The tools utilized within this project are presented in Table 3.1.

In case of mpstat and sar, we ran those tools inside the targeted Sprout node. Examples of how to run the monitors are provided below. With regard to sar, (-r) will show information related to the RAM status, and number one is used to show the RAM report every one second. Similarly to sar, mpstat will show CPU report every one second for a certain processing core number which is the specific core allocated to the component. Docker stats is a built-in command used with Docker containers. By default, it shows information about CPU and RAM status every one second.

```
1- sar -r 1
2- mpstat -P 0 1
3- docker stats
```

3.2 ClearWater's User Registration Flow

As previously introduced, this project is concerned in studying and analyzing user registration of an IMS architecture. Therefore, it is essential to have a closer look at messages being created and exchanged among the involved nodes of ClearWater during the user registration process. Figure 3.3 shows the exchanged SIP dialogue in order to successfully register new clients. In sake of facilitating the mapping of the SIP dialogue to a queuing network, the call flow was divided into four stages. The SIP dialogue consists of the following messages:

1. Bono received SIP register from SIP terminal. The register request will be forwarded to Sprout.

2. Sprout sends HTTP GET message to Homestead in order to retrieve authentication credentials.

3. Since we do not have any external HSS, Homestead will be in charge of calculating the Authentication Vector (AV) and generating the required authentication parameters.

4. HTTP 200 is forwarded back to Sprout accompanied with authentication parameters.

5. Sprouts sends SIP 401 unauthorized message to Bono asking the user to provide his security parameters and calculate the challenge response.

6. Since the SIP terminal has the secret key, it can then authenticate the network, calculate the required response, and send again a SIP register to the IMS system with the calculated values.

7. The messages will follow the same procedure as the first two stages except that Homestead will check if the user response is equal to the one that was calculated by Homestead in the first stage. If the two values are equal, then the user will be authenticated and 200 OK message will be forwarded back to the subscriber through Bono.

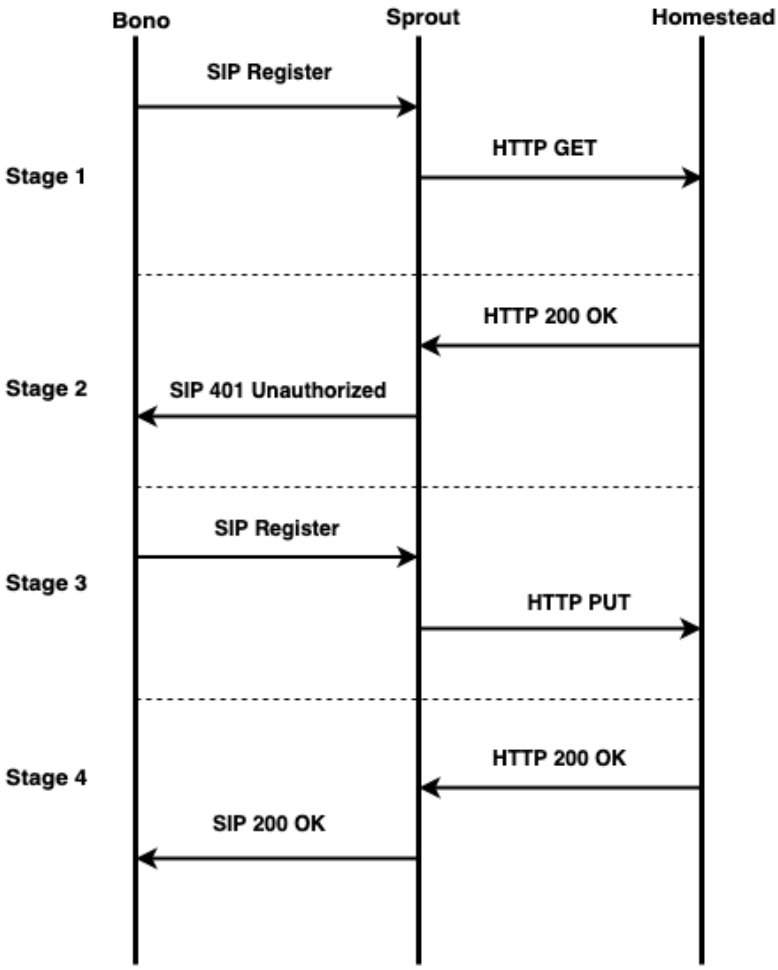


Figure 3.3: SIP: User Registration Dialogue, adapted from [Neta]

3.3 Testbed Specifications

In order to perform the experimental test, we use two physical machines with capabilities illustrated in Table 3.2. The first machine was dedicated to deploy the Docker container environment while the other one was used to host the VMs environment.

The two machines run Linux Ubuntu 16.04.6 Long Term Support (LTS) (Xenial Xerus) operating system. The first machine has higher capabilities with regard to the processing power and the memory capacity but when allocating physical resources to the service components, we have assigned the same amount to each VM and

Table 3.2: Specifications of the Physical Machines

Specification	**First machine**	**Second machine**
Chassis	Chassis with up to 8, 2.5" Hard Drives, Software RAID	Chassis with up to 8, 2.5" Hard Drives,Software RAID
Processor	2 x Intel® Xeon® E5-2650LV v4 1.7GHz	Intel® Xeon® E5-2650LV v4 1.7GHz
Memory	4 x 32 GB	4 x 16 GB
Storage	4 x 1TB Hard Drive	4 x 1TB Hard Drive
Network Card	Intel Ethernet X540 10 GB + I350 1 GB Network Daughter Card	Intel Ethernet X540 10 GB + I350 1 GB Network Daughter Card

container in order to ensure consistency of the computing capabilities among the two deployments.

3.4 Environment Setup

In this section, we introduce the steps to successfully install ClearWater on Docker containers and VMs. Afterwards, we present the necessary configurations to ensure ClearWater is prepared to conduct the experimental test.

3.4.1 ClearWater on Docker container

The entire process of deploying ClearWater on Docker containers can be found on [cle]. Algorithm 3.2 shows the commands needed in order to install ClearWater system. The first four commands used to install Docker and Docker compose on Ubuntu. The sixth command is to build the base image of ClearWater while the seventh is to build all other images and start ClearWater system. The last command used when scaling is desired. It will increase the number of Sprout instances to be two. After successfully installing ClearWater, a bunch of Docker containers will be running forming the ClearWater system. Table 3.3 shows the deployed containers and the resources assigned to each of them. All the containers, except Sprout which was assigned with different values of CPU cores and RAM, had one processing core and four GB of RAM.

A useful tool, called Weave Scope [Wea], can be used with Docker to provide a real-time interactive view including all deployed containers, connectivity between them, and resources usage.

Algorithm 3.2 Installing ClearWater on Docker container, , adapted from [cle]

```
1- wget -qO- https://get.docker.com/ | sh
2- git clone --recursive https://github.com/Metaswitch/clearwater-
docker.git
3- sudo apt-get install python-pip -y
4- sudo pip install -U docker-compose
5- cd clearwater-docker
6- sudo docker build -t clearwater/base base
7- sudo docker-compose -f minimal-distributed.yaml up -d
8- sudo docker-compose -f minimal-distributed.yaml scale sprout=2
```

Table 3.3: Deployed Docker containers

Containers	**CPU (core)**	**RAM (GB)**
Sprout1	1, 2	1, 2, 4
Sprout2	1	1, 2, 4
Chronos	1	4
Cassandra	1	4
Astaire	1	4
Homestead	1	4
Homestead-prov	1	4
Ralf	1	4
Ellis	1	4
Homer	1	4
Bono	1	4

Each Docker container has shared and local configuration files. The shared configuration file must be the same on all existed containers within ClearWater deployment. It contains the necessary parameters for ClearWater to operate properly. The shared configurations includes settings like the domain name, names of the deployment nodes, and the port numbers used to communicate among the different containers. A copy of the shared configuration is provided in 3.3. Once ClearWater is successfully installed on Docker, the shared and local configurations files will be created by default and there is no need to add or modify them to bring ClearWater working. If the physical machine, where ClearWater was installed, has an active Firewall, the port numbers existed in the shared configuration file must be permitted. Otherwise, the deployment nodes will fail to reach each other.

Algorithm 3.3 Docker's Shared Configuration,

```
home_domain=example.com
sprout_hostname=sprout
hs_hostname=homestead:8888
hs_provisioning_hostname=homestead-prov:8889
xdms_hostname=homer:7888
ralf_hostname=ralf:10888
chronos_hostname=chronos
cassandra_hostname=cassandra
sprout_registration_store=astaire
ralf_session_store=astaire
homestead_impu_store=astaire
# I-CSCF/S-CSCF configuration
upstream_hostname=sprout
# Keys
signup_key=secret
turn_workaround=secret
ellis_api_key=secret
ellis_cookie_key=secret
scscf_uri="sip:sprout:5054;transport=tcp"
icscf_uri="sip:sprout:5052;transport=tcp"
```

The local configuration file contains information regarding the private and public IP addresses, and the hostname. The public IP address can be set to the private IP address on all nodes, except Ellis and Bono in case they will be used by remote users to contact ClearWater. The format of the local configuration file is shown in 3.4.

Algorithm 3.4 Docker's Local Configuration, , adapted from [Netb]

```
local_ip=<IP_address>
public_ip=<IP_address>
public_hostname=<IP_address>
```

3.4.2 ClearWater on VM

To be able to implement ClearWater through VM, a virtualization player such as VMware workstation is required. Once the VMware player was installed, we started with creating seven different VMs running Ubuntu 14.04 - 64bit server edition which

can be downloaded from [ubu]. Other Linux versions might properly work with ClearWater, but it is recommended to use the mentioned above edition.

After creating the VMs, we had to grant them an access to Clearwater Debian repository. This done as shown in 3.5 :

Algorithm 3.5 Configuring Nodes to Access Clearwater Debian Repository, adapted from [Netb]

```
1- mkdir /etc/apt/sources.list.d/
2- vim clearwater.list
3- deb http://repo.cw-ngv.com/stable binary/
4- curl -L http://repo.cw-ngv.com/repo_key | sudo apt-key add -
5- sudo apt-key finger
```

First, we created a directory which contains the clearwater.list file which was created by using a text editor called vim. The file contains the third line of the algorithm 3.5 which is the link of ClearWater Debian server. Once the file was created, we imported the signing key of Clearwater by executing the fourth command. The final command is to check the correctness of the key fingerprint.

Second, creating the per-node configuration as it was done in 3.4. The next step, is to determine the role of each VM: Sprout1, Sprout2, Vellum, Dime, Ellis, Homer, and Bono. It is not possible to implement the functionality of Dime or Vellum on separated VMs as the Docker containers. Table 3.4 shows the amount of resources assigned to each VM. Each of Dime and Vellum has three dedicated processing cores and twelve GB of RAM in order to make them equivalent to the settings of the Docker implementation. In Docker environment, Dime was divided into three separated containers: Homestead, Homestead Prov, and Ralf. The total resources assigned to Dime's node was three cores and twelve GB of RAM. Similar to Dime, Vellum had three separated container: Cassandra, Chronos, and Astaire.

The commands shown in Algorithm 3.6 are used to assign roles to the created VMs.

In contrast to Docker environment, we had to create the shared configuration file on each VM. The file must be in /etc/clearwater directory. All VMs were equipped with the same shared configuration file as Docker containers 3.3. After creating the shared and local configuration files on each VM, ClearWater is ready to perform the experimental measurement.

Algorithm 3.6 Determining VMs Roles, adapted from [Netb]

```
#Sprout1
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install sprout --yes
2- sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
#Sprout2
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install sprout --yes
2- sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
#Ellis
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install ellis --yes
2- sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
#Homer
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install homer --yes
2- sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
#Vellum
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install vellum --yes
2- sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
#Dime
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install dime
clearwater-prov-tools --yes
2- sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
#Bono
1- sudo DEBIAN_FRONTEND=noninteractive apt-get install bono
restund --yes
2-sudo DEBIAN_FRONTEND=noninteractive apt-get install
clearwater-management --yes
```

Table 3.4: Deployed VMs

VM	**CPU (core)**	**RAM (GB)**
Sprout1	1, 2	1, 2, 4
Sprout2	1	1, 2, 4
Vellum	3	12
Dime	3	12
Ellis	1	4
Homer	1	4
Bono	1	4

3.5 SIPp: Preparation and Installation

The experimental test can be run from a separated node within the ClearWater deployment, or from the Bono node. We chose to install SIPp on the Bono node to reduce the number of exchanged messages during the registration process. The SIPp tool was used to produce a large amount of SIP traffic against the ClearWater deployment in order to observe how the system will perform under certain circumstances, and to monitor relevant performance metrics.

Several pre-requisites libraries are needed in order to compile and install SIPp successfully:

1. C++ Compiler.
2. curses or ncurses library.
3. GNU Scientific Library (GSL) [Sys] to enable distributed pauses at the test time.

Algorithm 3.7 SIPp v3.4: Preparation and Installation

```
1- sudo apt-get install -y ncurses-dev build-essential
libncurses5-dev gcc
2- wget https://sourceforge.net/projects/sipp/files/sipp/3.4/
sipp-3.3.990.tar.gz
3- tar -xvzf sipp-3.3.990.tar.gz
4- cd sipp-3.3.990
5- ./configure --with-gsl
6- make
```

Algorithm 3.7 shows the required commands to prepare and install SIPp. The first command used to install the required libraries such as gcc and ncurses. Then, the SIPp source code of version 3.4 was downloaded from the official website and decompressed. The configure command is responsible for getting the software ready to be installed on the system. Once configure command has finished, we executed make command to build the software from its source code. After executing the all commands, SIPp was installed and ready to be used by initiating the following command:

$$sipp - i < local\_ip > -sf < xml\_scenario > -inf < csv\_file >< remote\_ip > -r < arrival\_rate > -m < max\_requests >$$

The local IP is the IP address of the SIPp machine. The remote IP refers to Bono IP address. Flag r used to define the arrival rate which is the sent requests per second. And the last flag used to determine the total number of subscribers to be registered. We had to write the XML scenario and pass it to the SIPp command. The scenario file must include all the SIP messages. The scenario is shown in Section 6.4. The XML code includes four different SIP messages: SIP Register, 401 Unauthorized, SIP Register, and 200 OK. Field 0 and 1 in the XML scenario are imported from the CSV file which contains all the subscribers identities, home domain, and passwords. The CSV can be generated by several ways. We have written a simple script to generate the file:

```
echo "SEQUENTIAL"
 for i in ‘seq 2010000000 2010002000‘; do
     echo "$i;example.com;[authentication username=
     $i@example.com password=<password>];"
 done
```

The script will create a total number of 2000 subscribers and insert them in the first column. The second column refers to the domain name which is example.com in our case. The last contains the authentication parameters like username and password.

3.5.1 Experimental Test Settings

In this project, we focused on how the allocated resources to Sprout will affect the initial user registration's performance. Furthermore, we considered scaling up the Sprout in order to investigate how that would impact the system metrics.

Table 3.5 shows the combinations of the experimental test. First, we started the

Table 3.5: Stress Test's Settings

Sprout Settings	**CPU (core)**	**RAM (GB)**
1 Sprout	1	1 2 4
1 Sprout	2	1 2 4
2 Sprouts	1	1 2

test with one Sprout node with one processing core, and assigned different values of RAM. Secondly, the processing cores were changed to be two while changing the RAM from 1 to 4 GB. Finally, we increased the Sprout instances to investigate the impact that a scaling up procedure might have on the average delay for user registration and the resources utilization. It is interesting to compare the output of second and third settings which will highlight if decomposing of Sprout has a positive effect on the performance or not. The detailed comparison is introduced in Chapter 4.

We conducted the test for several values of arrival rate while keeping the total number of subscribers to be registered fixed to two thousands subscribers. The considered values of arrival rates for each single row of Table 3.5 were:

1. 20 Registers/second
2. 50 Registers/second
3. 80 Registers/second
4. 110 Registers/second
5. 170 Registers/second
6. 200 Registers/second
7. 250 Registers/second
8. 300 Registers/second

We started the test from 20 Registers per second and then increased it by a fixed step (30) until 200 Registers per second. The last two values are 250 and 300

Registers per second. The different values aimed to produce an extensive analysis of the system behaviour under different circumstances. The test was run fifty times for each single value of the arrival rate in order to produce more precise results by taking the average of the all fifty times' outputs. During the test, we monitored the CPU and RAM consumption of the Sprout node which is responsible for processing the generated data by the SIPp tool.

3.6 Analytical Model and Analysis

Queuing network theory is an approach to model systems as a network of queues where each queue represents a service center. In order to map ClearWater's components to the queuing network, we need to identify the involved nodes in the user registration process. Based on Figure 3.3, only three components (Bono, Sprout, and Homestead) are involved in the signalling path. The registration process starts with a SIP register sent from Bono to Sprout, and ends with SIP 200 OK sent back to Bono.

Before introducing the methods we considered to derive the analytical model, it is essential to introduce briefly Kendall's notation which is used to describe the queuing system. The following notation A/B/C/D refers to:

1. A: is the arrival distribution (M for Poisson, and G for general).
2. B: is the service time distribution (M for negative exponential distribution, and G for general distribution).
3. C: refers to the number of servers
4. D: maximum number of customers in the queue.

When regarding IMS systems, it is commonly assumed that user call requests, where SIP registrations represent the signaling part, are modeled as Poisson process and the service process to be a Markovian process with independent exponentially distributed service times [MG10, Mun08, WBBD05, NTN13, FCP06]. In considering the network service as a whole, the end-to-end system is represented as a tandem of queues in series where each queue represents an individual component. However, in some of the related work, the assumption of M/M/1 queues for the individual components raises some doubts with other assumptions regarding the processing delay, i.e., registration service. In particular, the authors of [MG10, Mun08] do assume that each queue is a separate M/M/1 so that the queuing delay has a closed form expression but on the other hand, they consider a fixed, i.e., deterministic, processing delay which goes in contrast with the very first assumption of having exponentially distributed service times.

Similarly to the aforementioned related work, we assume that each different network service entity, involved in the IMS signaling, is modeled through an M/M/1 queue. Therefore, the resulting network is a queuing network of M/M/1 tandem queues, and in case the input process is Poisson, the departure process is also Poisson and independent of the input process [Zuk13]. As a result, the total average waiting time or delay in the queuing network, consisting of queues in tandem, is the sum of the expected waiting times at each queue. However, we assume a different approach on validating the suitability of the assumed model.

First, during the experimental measurement, we specified the arrival process distribution to follow a Poisson process with exponential inter-arrival times of the user's Registration requests. This was achieved by setting a 'pause' process between each Registration request in the XML SIPp scenario. Through the GNU Scientific Library (GSL) package, we were able to set a specific distribution for the inter-arrival times. For each measurement, the 'pause' follows an exponential distribution with a mean value equal to the specific arrival rates under consideration. Therefore, in modelling the IMS system through a network of queues, the external arrival process consists in a Markovian process.

Secondly, we assume that the service process of each queue may have either negative exponential distribution or a general distribution. Therefore we consider two methods to derive the analytical model: M/M/1 and M/G/1. For both approaches, it is required to identify the service time since the actual service time for each of Bono, Sprout, and Homestead is unknown. In order to estimate the service time of each node, one packet was sent to ensure that the node is empty and the packet will be processed instantly upon reaching the node. The entire process was captured and analyzed by a network analyzer tool.

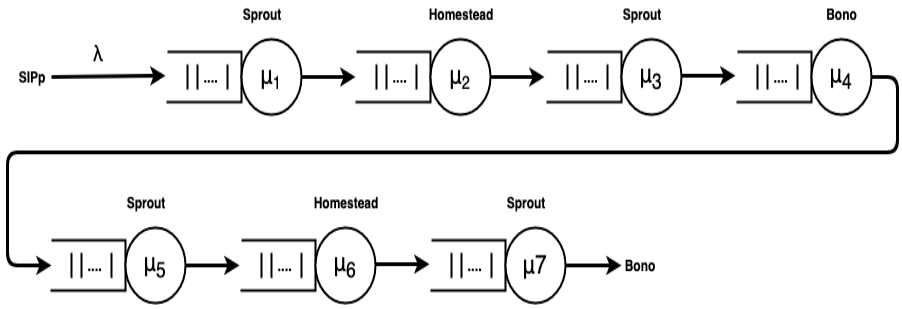


Figure 3.4: Feed-Forward Tandem of Queues

In total, we have seven different service times as shown in Figure 3.4. The service time is the time needed for a certain node to process and send the data out to the next node. As stated above, those service times values were captured using Wireshark

Table 3.6: Service Time Values for Docker containers

Average service time	**Average service rate**
$\frac{1}{\mu_1}$= 0.001588	μ_1= 629.72
$\frac{1}{\mu_2}$= 0.002314	μ_2= 432.15
$\frac{1}{\mu_3}$= 0.003057	μ_3= 327.11
$\frac{1}{\mu_4}$= 0.009585	μ_4= 104.32
$\frac{1}{\mu_5}$= 0.004378	μ_5= 228.41
$\frac{1}{\mu_6}$= 0.001225	μ_6= 816.32
$\frac{1}{\mu_7}$= 0.003024	μ_7= 330.68

Table 3.7: Service Time Values for VMs

Average service time	**Average service rate**
$\frac{1}{\mu_1}$= 0.002824	μ_1= 354.10
$\frac{1}{\mu_2}$= 0.004246	μ_2= 235.51
$\frac{1}{\mu_3}$= 0.006198	μ_3= 161.34
$\frac{1}{\mu_4}$= 0.010651	μ_4= 93.88
$\frac{1}{\mu_5}$= 0.007919	μ_5= 126.27
$\frac{1}{\mu_6}$= 0.002874	μ_6= 347.94
$\frac{1}{\mu_7}$= 0.005992	μ_7= 166.88

network analyzer.

Since we have quite many combinations of CPU and RAM during the measurement phase, it is cumbersome to derive the analytical model for all the available settings. Therefore, for sake of simplicity, we chose to analyse and validate the analytical model for the case when Sprout has one processing core and two GB of RAM for each of Docker container and VM.

Tables 3.6 and 3.7 show the numerical values for the average service times of Docker containers and VMs. Each value consists of the average service time of one hundred measurement when only one registration is generated.

3.6.1 M/M/1

Consider an IMS based network with K identical servers with no loss due to buffer overflows. By applying M/M/1, it was assumed that:

– Each node has an infinite capacity.

- Arrival process from outside is a discrete time process which follows a Poisson distribution with arrival intensity λ.
- The service time is a negative exponential distribution with parameter μ.

All messages related to the registration process have to traverse through the queues shown in Figure 3.4 sequentially. This can be viewed as a feed-forward tandem path. In particular, the Burke's theorem states that in an M/M/1 system where the arrival process is Poisson with arrival rate λ, at equilibrium, the departure process is still Poisson with the same parameter. Therefore, the arrival process of the second queue will still be a Poisson process with arrival rate equal to λ and so on for the rest of queues in the tandem network. The overall registration time is the sum of the response time for the all queues.

First, we calculated the traffic intensity for each queue by applying the following formula:

$$\rho = \frac{\lambda}{\mu} \qquad \rho < 1$$

As stated above, the queue is considered to be stable and the later formulas applicable if $\rho < 1$. In other words, the arrival rate must not be greater than the service rate. For an M/M/1 queue, the Sojourn time (response time) is given by the following formula:

$$\underbrace{E[S]}_{\text{Sojourn time}} = \underbrace{E[W]}_{\text{Waiting time}} + \underbrace{E[X]}_{\text{Service time}} = \frac{1}{\mu - \lambda}$$

The expected Sojourn time is the sum of the expected waiting and service time. Hence, the total expected end-to-end delay of the network equals the sum of the expected Sojourn times for each and every queue present in the queuing network.

$$Total\_delay = \underbrace{\sum_{n=1}^{7} E[S_i]}_{\text{Sojourn time}}$$

3.6.2 M/G/1 and G/G/1

Consider an IMS based network with K identical servers. It is assumed that the servers experience no loss due to buffer overflows. The arrival process from outside is assumed to follow a Poisson distribution. When a M/G/1 queuing system is assumed, the service time distribution can be any general distribution like Geometric, Weibull, Gamma, Lognormal, etc. It is assumed that the service time distribution is general with definite first and second moments.

In case of M/G/1, we can apply M/G/1 to the first queue where the arrival distribution was predefined within the XML scenario to follow a Poisson process. However, its output is basically no more Poisson. Even though the next queue might have the same arrival rate or inter-arrival time as the first queue, but the other characteristics (e.g. variance) of the input to the next queue is unknown. Therefore, M/G/1 is a valid method to be applied for the first queue analysis, but not the other queues in the tandem network. Thus, starting from the second queue in the tandem 3.4, the queues are considered to be G/G/1. In other words, only the first queue has a Poisson arrival process, while others in the tandem have a general arrival process.

The total end-to-end delay average delay is the sum of each queue in the tandem. Hence, the queuing network can be abstracted as a feed-forward tandem as it is shown in Figure 3.4. Similarly to the M/M/1 case, all the messages during the connection setup have to traverse through the tandem of queues sequentially.

As mentioned above, the expected Sojourn time is the sum of the expected service and waiting time. The expected service time for each queue is known as it was captured from the Wireshark tool. Regarding the waiting time, we make use of two different formulas to calculate the waiting time for all the queues within the tandem. For the first queue (M/G/1), we utilized a popular method to estimate the waiting time, called Pollaczek-Khinchin formula [Pol30]. It was developed in 1930 by a mathematician called Félix Pollaczek, and recast two years later by Aleksandr Khinchin. The formula gives an approximation of the waiting time for a queue with a single process. The formula is shown below:

$$E(W) = \frac{(1 + {C_s}^2)}{2} * \frac{\rho}{1 - \rho} * E[S]$$

where:

1. C_s^2: is the coefficient of variation of the service time. It is equal to the variance divided by the squared mean service time $\frac{\sigma_a^2}{(1/\mu)^2}$.
2. ρ: utilization of the server or the traffic intensity. It equals to $\frac{\lambda}{\mu}$.

In order to calculate the waiting time for other queues, we used a different method to estimate it, called Kingman formula. It was developed in 1961 by British mathematician John Kingman [Kin61]. The formula gives an approximation of the waiting time for a queue with a single process. The Kingman equation demands independently distributed inter-arrival and service times. The formula gives an

approximation of the mean waiting time in an G/G/1 queue an is know to be very accurate in particular for high utilization queues. The formula is shown below:

$$E(W) = \frac{({C_a}^2 + {C_s}^2)}{2} * \frac{\rho}{1-\rho} * E[S]$$

where C_a^2: is the coefficient of variation of the inter-arrival time. It is equal to the variance divided by the squared mean inter-arrival time $\frac{\sigma_a^2}{(1/\lambda)^2}$.

After calculating the waiting time for each and every queue within the signalling path, the total delay can be calculated and it equals:

$$Total\_delay = \underbrace{\sum_{n=1}^{7} E[W_i]}_{\text{Waiting time}} + \underbrace{\sum_{n=1}^{7} E[X_i]}_{\text{Service time}} = \underbrace{\sum_{n=1}^{7} E[S_i]}_{\text{Sojourn time}}$$

In order to investigate and validate the applicability of the assumed models, we calculated the relative error of the analytical models as follows:

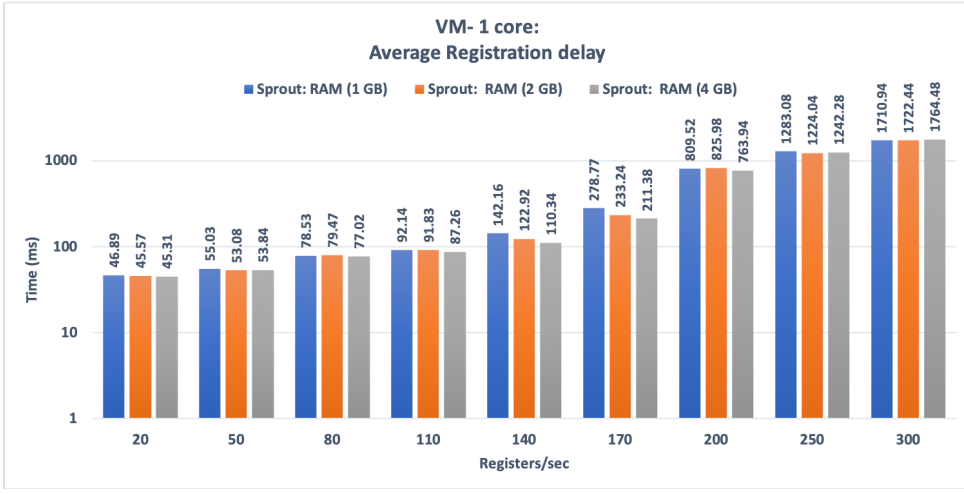
$$Relative\_error = \frac{|Experimental\_res - Analytical\_res|}{Experimental\_res} * 100$$

where the *Experimental_res* denotes the average end-to-end delay of the registration process retrieved from experimental measurements and the *Analytical_res* denotes the average end-to-end delay from the assumed queuing system.

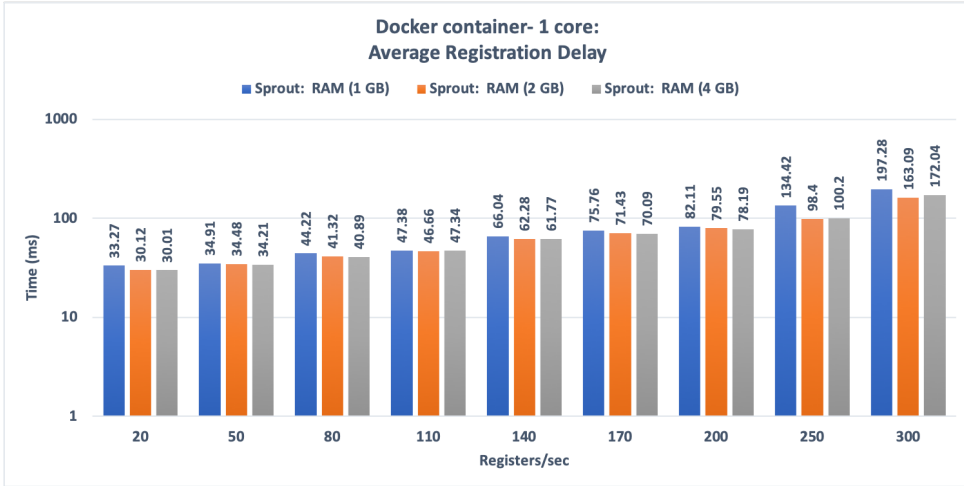
Chapter 4
Results and Discussion

4.1 Experimental Results

This chapter aims to introduce the results have been captured and calculated from the experimental measurements and analytical analysis respectively. All the introduced delay results represent the average delay for fifty different measurements. A single measurement consists in capturing the average system delay of performing the registration of 2000 subscribers for a specific registration rate, i.e., 20, 50 Reg/sec etc.



(a) VM: Average Registration Delay for Sprout with One Processing Core



(b) Docker container: Average Registration Delay for Sprout with One Processing Core

Figure 4.1: Average Registration Delay

4.1.1 Average Registration Delay

All the figures of average delay use a Logarithmic scale. The X axis represents several values of arrival rates for different values of RAM allocated to the Sprout component which represents the most important component in the Clearwater system since it implements both I-CSCF and S-CSCF. The Y axis is the average system time of 2000 registrations measured in ms.

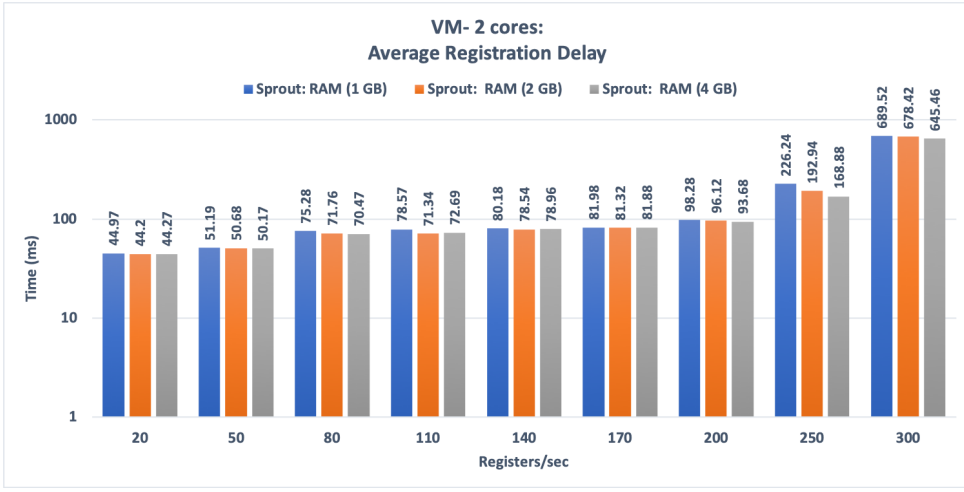
We observed that the trends in the average registration delay witness a significant increment when increasing the arrival rate across several environment settings with different constraints.

For VMs with on core dedicated to Sprout 4.1(a), the average registration delay starts from around 46 ms reaching more than 1700 ms at the end. This is due to the lack of the processing power. We observe here that amount of RAM has almost a negligible effect on the performance when having a fixed registration rate. Therefore, allocating a small amount of RAM does not produce a noticeable gain in terms of system delay. On the other side, the average delay for Docker container with on processing core assigned to Sprout 4.1(b) varies approximately from 33 up to around 200 ms. It is apparent that Docker containers can handle more requests per second with a better average delay compared to VMs.

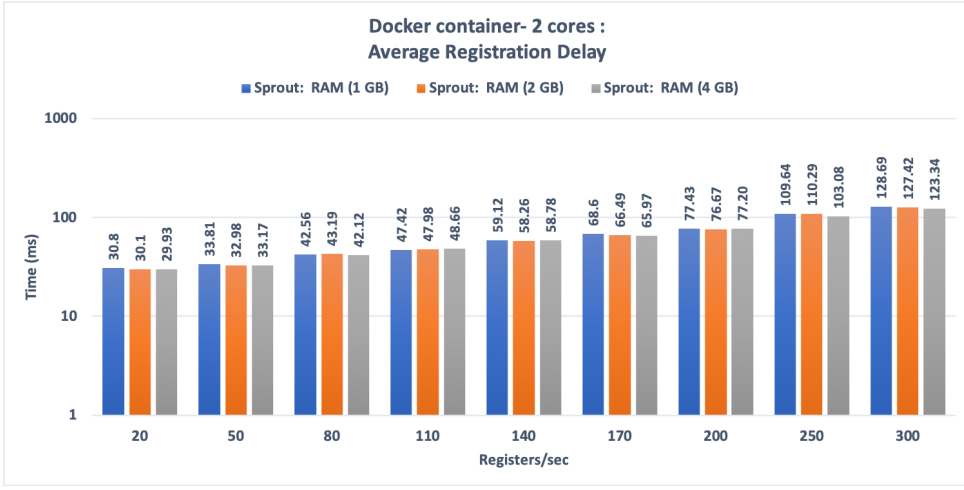
Dedicating two processing cores for VMs made the average delay declined to less than 700 ms as it is shown in Figure 4.2(a). The effect of having more processing power is obvious especially when the arrival rate is bigger than 110. Applying the same setting to Docker container 4.2(b) has a marginal enhancement over the average delay, but it still not that much compared to VMs. The average delay at the maximum value of arrival rate reaches about 125 ms. It is true that assigning more processing power will improve the performance of the system, but this is compounded by an increasing cost.

Having two separated instances of Sprout had almost the same impact on the average delay as having two dedicated cores assigned for Sprout. We measured the average delay when each of the Sprout instances has one GB of RAM, and then two GB of RAM. In case of VMs, the delay starts from around 45 ms and goes up to 530 ms at 300 registers per second. Whereas, Docker containers have a maximum delay of 142 ms when it is most loaded. To conclude, scaling up Sprout has a performance close to that provided by two processing cores, but it is advantageous when high availability and fault tolerance are desired.

To summarize, Docker containers provide better average delay through all the environment's settings compared to VMs. This is somewhat expected as the hypervisor virtualization comprises an additional layer between the guest operating system



(a) VM: Average Registration Delay for Sprout with Two Processing Cores



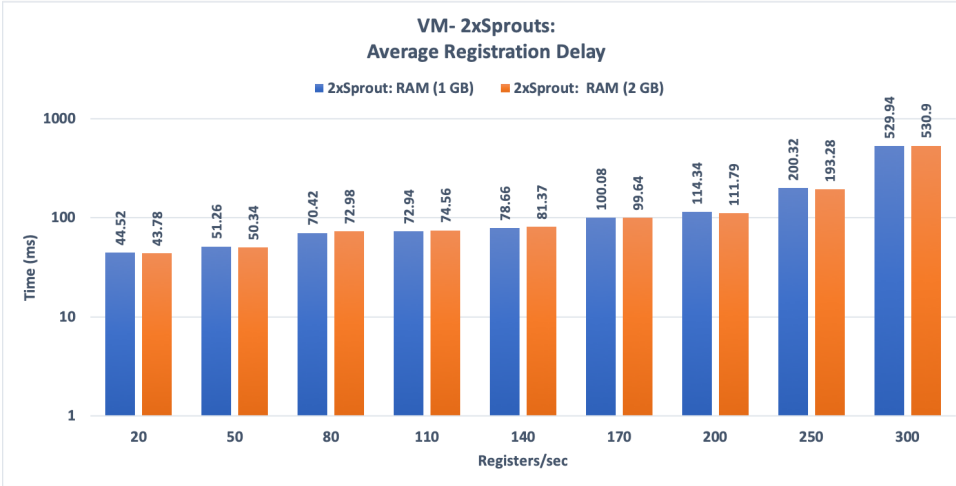
(b) Docker container: Average Registration Delay for Sprout with Two Processing Cores

Figure 4.2: Average Registration Delay

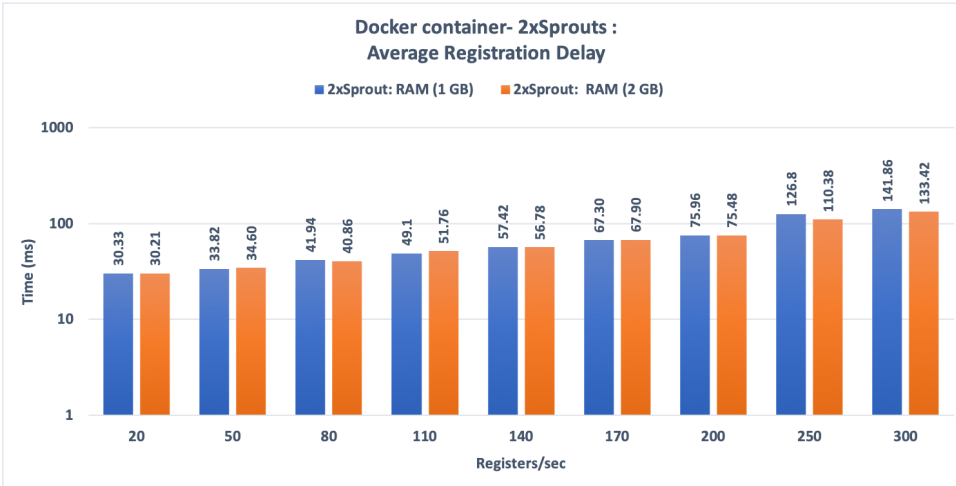
and the physical hardware resources, i.e., hypervisor itself. In this case, any . Since there is memory overhead, Docker containers has a lower service time which means that they can handle more traffic per second.

4.1.2 Average CPU Utilization

It is essential to monitor the available resources consumption under different amount of data. All the figures showing the CPU utilization of Sprout have a similar upward



(a) VM: Average Registration Delay for 2xSprouts

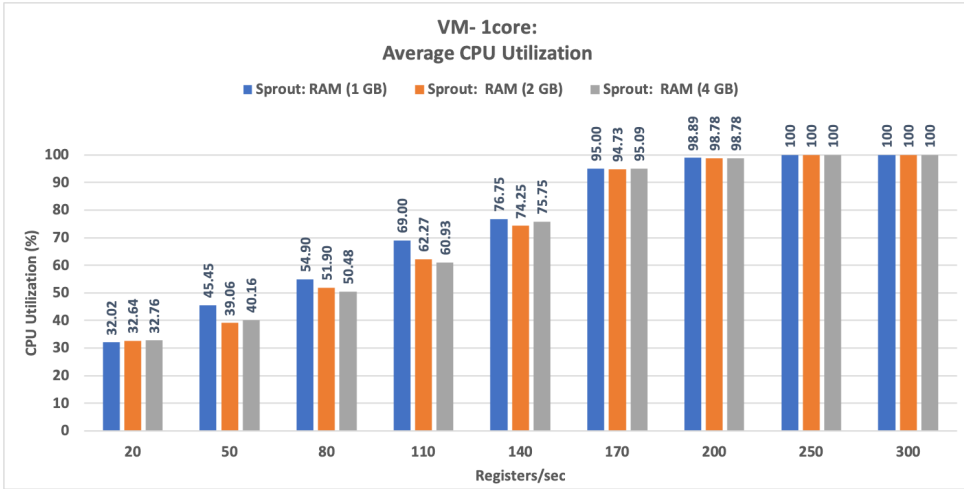


(b) Docker container: Average Registration Delay for 2xSprouts

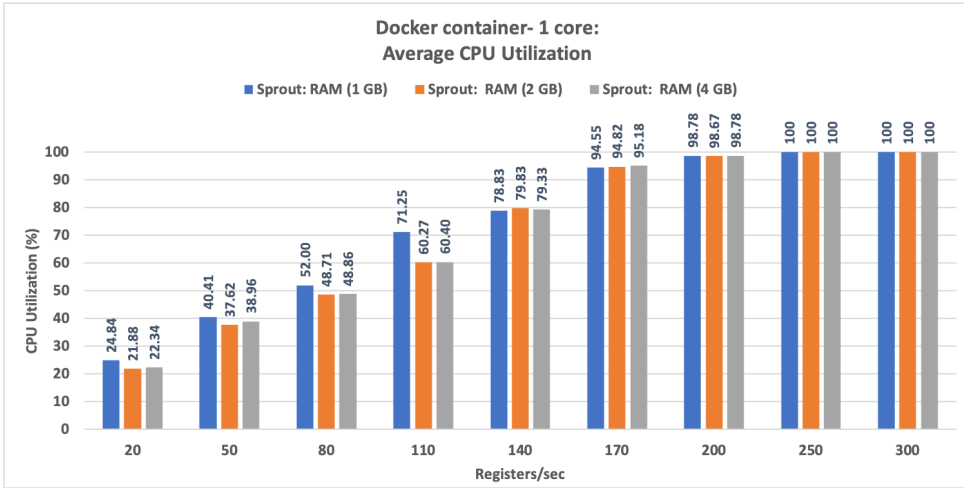
Figure 4.3: Average Registration Delay

trends. The CPU utilization increases as the number of sent requests rises. The Y axis indicates the percent CPU utilization. The X axis represents the arrival rate values.

From Figures 4.4, we observe that Docker containers and VMs provide almost similar CPU utilization when they are equipped with one processing core. Both Docker containers and VMs reach the saturation state when the arrival rate is around



(a) VM: Average CPU Utilization for Sprout with One Processing Core

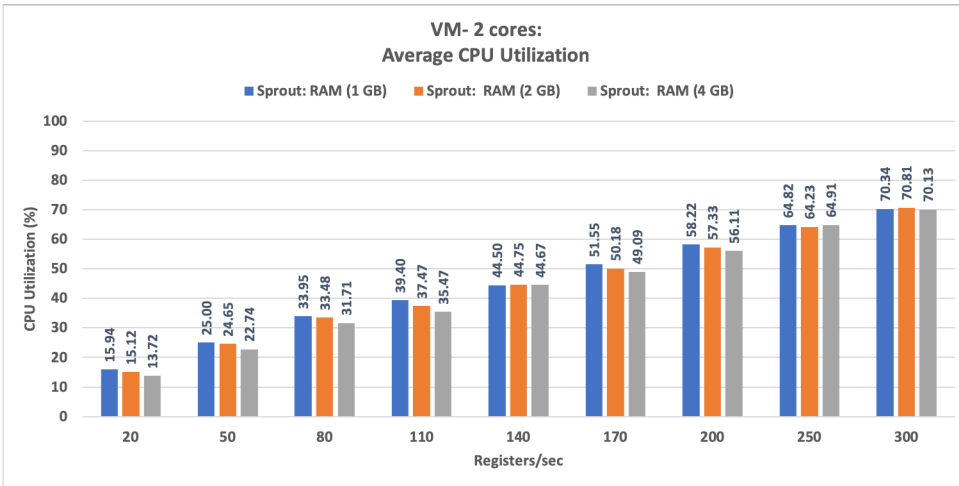


(b) Docker container: Average CPU Utilization for Sprout with One Processing Core

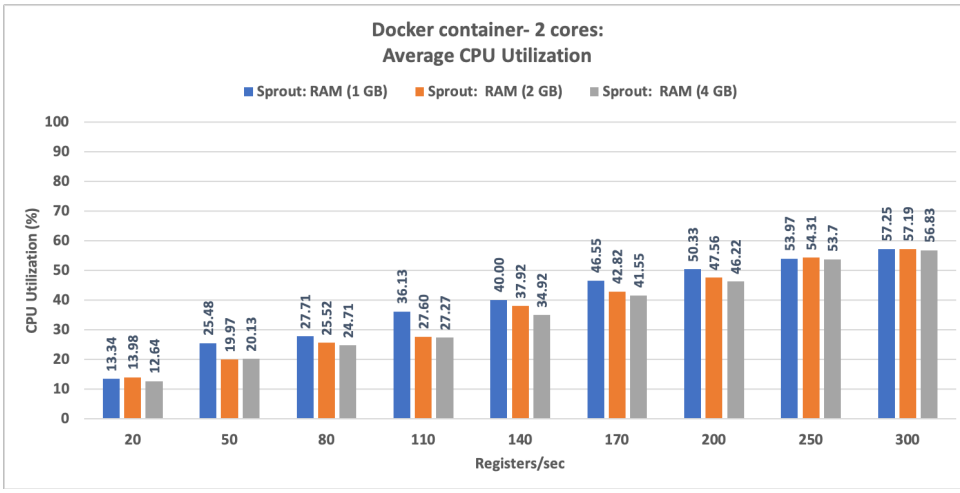
Figure 4.4: Average CPU Utilization

200 Registers per second.

Increasing the number of processing cores for Sprout significantly enhance the CPU utilization for both Docker containers and VMs. With regard to VMs, the CPU consumption decreases to reach around 71 % when it is mostly loaded as it is shown in Figure 4.5(a). In case of Docker containers, they witness a considerable drop in CPU utilization 4.5(b), and they produce a consumption of 57 % when they



(a) VM: Average CPU Utilization for Sprout with Two Processing cores

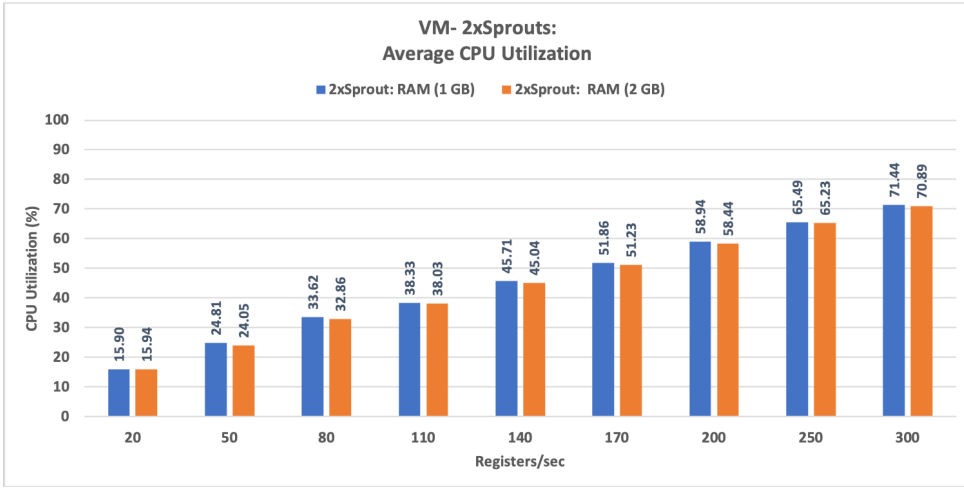


(b) Docker container: Average CPU Utilization for Sprout with Two Processing cores

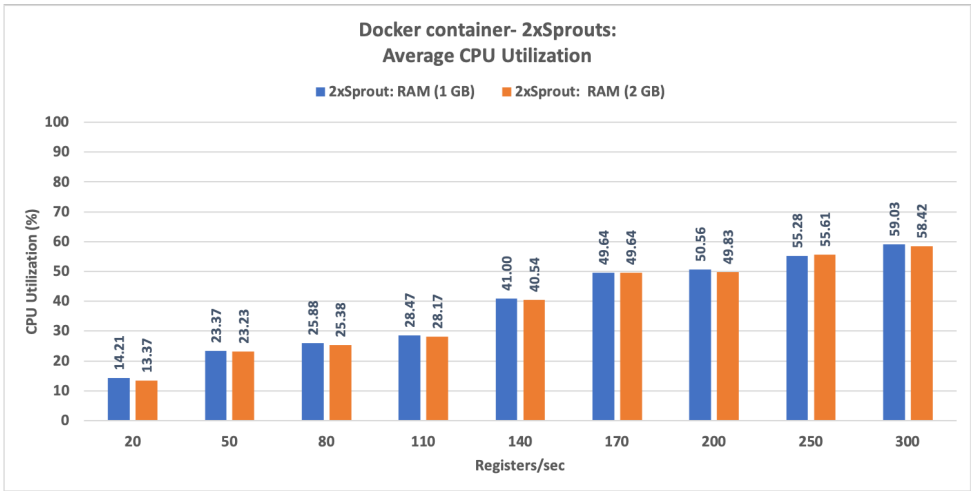
Figure 4.5: Average CPU Utilization

are actually under the most intensive generated traffic. Figure 4.5 is a good example to compare the CPU consumption of Docker containers and VMs. It is clear that Docker containers provide minimal CPU usage under different settings of RAM and arrival rates.

The last environment setting, where there are two Sprout instances, outputs marginally higher CPU consumption compared to that one provided when Sprout has two dedicated cores as it is shown in Figures 4.6. The CPU consumption of



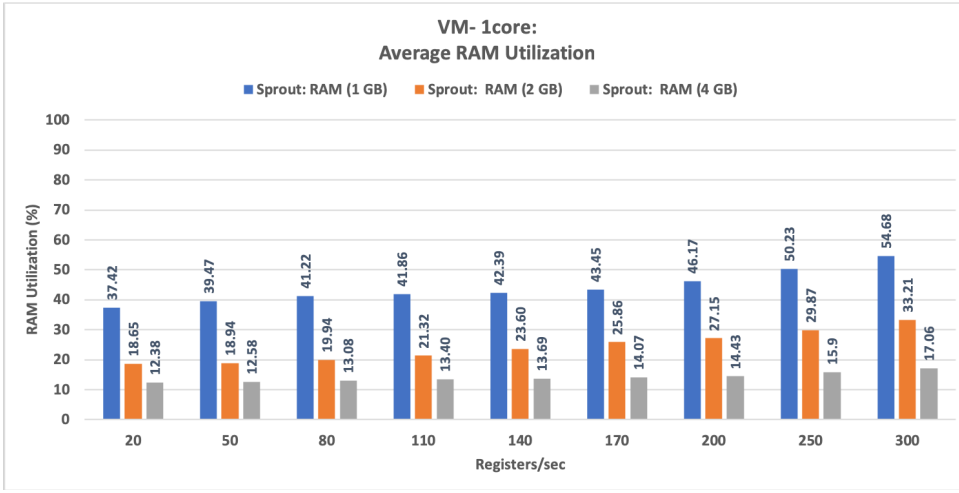
(a) VM: Average CPU Utilization for 2xSprouts



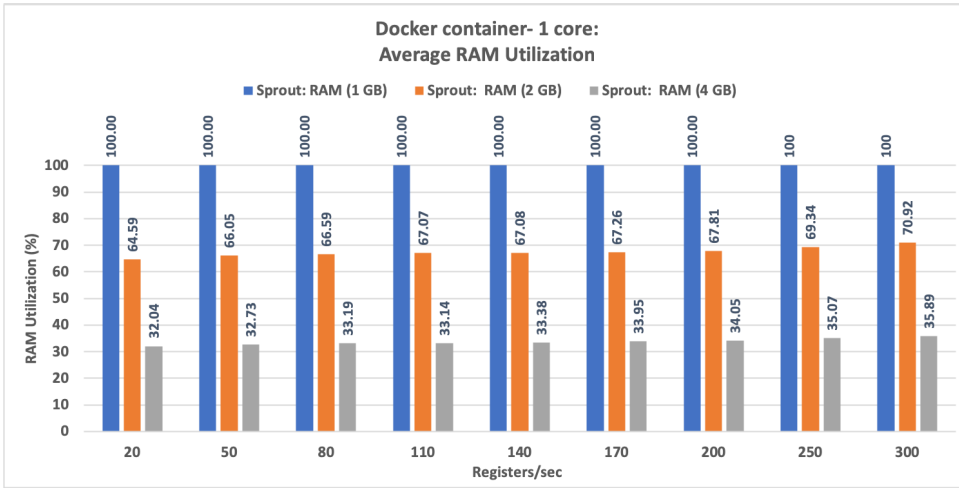
(b) Docker container: Average CPU Utilization for 2xSprouts

Figure 4.6: Average CPU Utilization

VMs starts from around 16 % and then increases considerably to 71 % at maximum value of arrival rate. While the CPU usage of Docker containers fluctuates between 14 % and 60 % over the different environment settings. Again, Figure 4.6 shows that Docker containers are superior to VMs when it comes to CPU consumption as the have beaten VMs in each and every single environment setting.



(a) VM: Average RAM Utilization for Sprout with One Processing Core



(b) Docker container: Average RAM Utilization for Sprout with One Processing Core

Figure 4.7: Average RAM Utilization

4.1.3 Average RAM Utilization

With regard to the RAM utilization, we considered one environment setting where Sprout has one processing core. The RAM usage for Docker containers and VMs is shown in Figure 4.7(b) and Figure 4.7(a) respectively. VMs provides less memory consumption as the RAM utilization did not exceed 50% of the total available RAM. In terms of Docker containers, Sprout consumes the entire available RAM when it is equipped with one GB of RAM. Reaching the saturation in case of Docker containers

is due to the minimum operating requirements.

CPU needs to access the RAM memory whenever it has to write or read to or from the RAM. Having no available RAM will reflect negatively on the speed of writing and reading operations which might result in a greater delay during the registration process. Varying the sent requests per second would slightly affect the RAM utilization. As more subscribers data needs to be saved on the RAM for the later registration process, the RAM consumption starts to witness a marginal increment.

4.2 Analytical Results

This section introduces the analytical results of M/M/1 and M/G/1 & G/G/1 for VMs and Docker containers. The used approaches and formulas are shown in Section 3.6. The relative error, which shows how much the experimental result differs from the analytical result, is confined to 25 %. The service and inter-arrival time distributions fitting are presented in the following two sub-sections respectively.

4.2.1 VMs

Figure 4.8 shows the relative error between the experimental and analytical results for M/M/1 and M/G/1. We have three values of the arrival rate fulfill the stability condition where $\rho < 1$. M/M/1 seems to provide appropriate results until 50 Reg/sec. After that value, the trend starts to increase considerably reaching a top of more than 65.12 %. We noticed that M/M/1 is not suitable to model user registration process when the queue utilization (ρ) > is greater than 0.6. On the contrary, M/G/1 & G/G/1 outputs a better relative error starting from 5.19 % when the arrival rate is 20 Registers per second, and ends at 8.39 % at the maximum value of arrival rate. The relative error of M/G/1 & G/G/1 over the all values of arrival rates did not exceed 10 %. Therefore, M/G/1 & G/G/1 is a valid method to model the subscriber registration process of ClearWater deployed on VMs, and it provides a minimal relative error. M/M/1 might be used if the queue utilization is around half.

4.2.2 Docker containers

Figure 4.9 shows the relative error between the experimental and analytical results for M/M/1 and M/G/1 & G/G/1. Similar to VMs, there are three values of arrival rates meet the stability condition of a queue. With regard to M/M/1, the relative error starts from around 5.53 % and ends at 49.78 %. M/M/1 is not valid when the queue utilization reaches around 0.6. Therefore, M/M/1 outputs adequate results until 50 Registers per second. On the other side, M/G/1 combined with G/G/1 has a relative delay less than 12 %. The relative error starts from 11.83 %, and it fluctuates

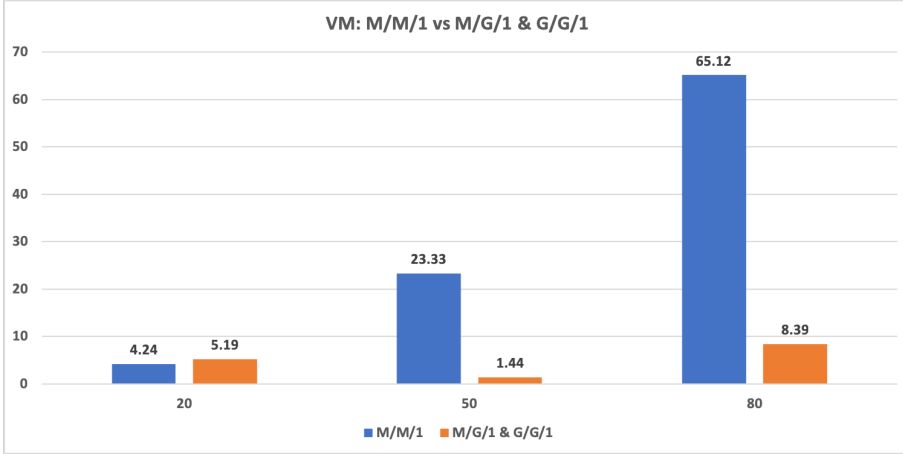


Figure 4.8: VMs relative error: M/M/1 vs M/G/1 & G/G/1

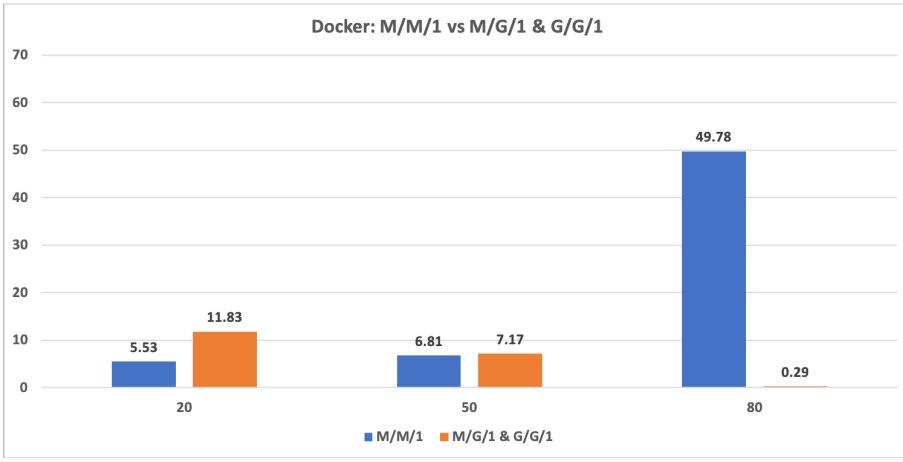
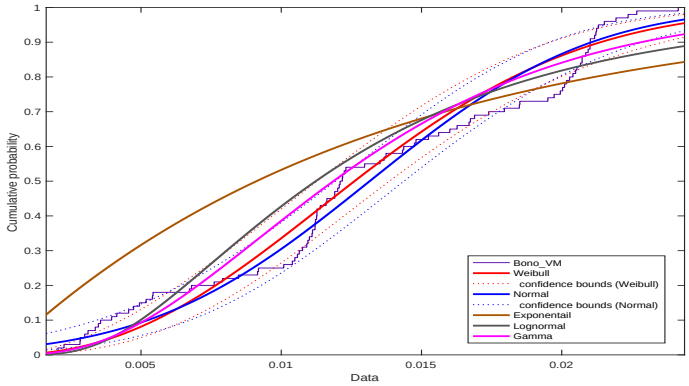


Figure 4.9: Docker containers relative error: M/M/1 vs M/G/1 & G/G/1

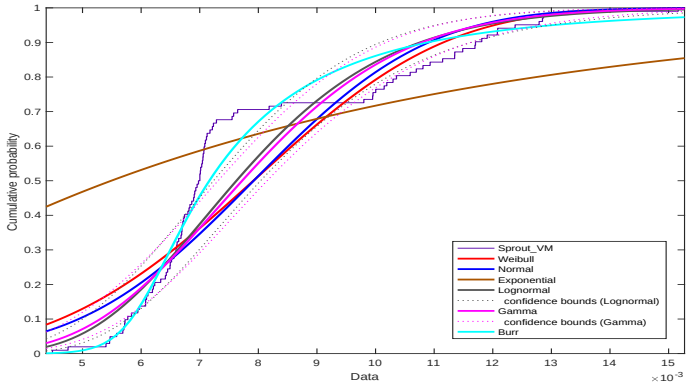
reaching a 0.29 % at the end. Hence, M/G/1 & G/G/1 is a sufficient method to model the studied process when ClearWater is deployed on top of Docker containers.

4.2.3 Service Time Distribution Fitting for M/G/1 & G/G/1

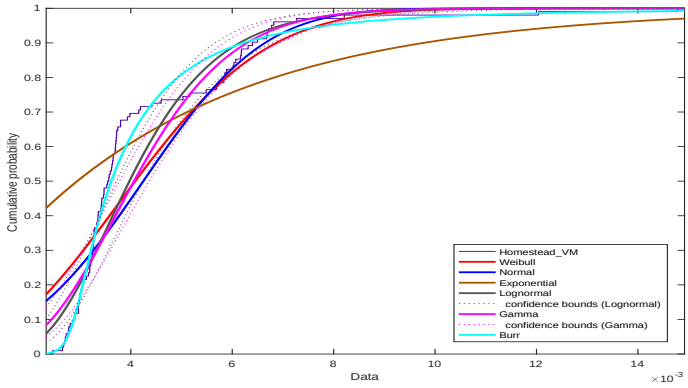
As stated previously, the service time measurement was done for a sample size of one hundred times. Then, the average for each service time was calculated. We used Matlab to define the service time distribution for each component in the signalling path: Bono, Sprout, and Homestead. Distribution fitting is a process that aims to select a statistical distribution that best fits a set of data. For illustration, we present



(a) Bono: Service Time Distribution



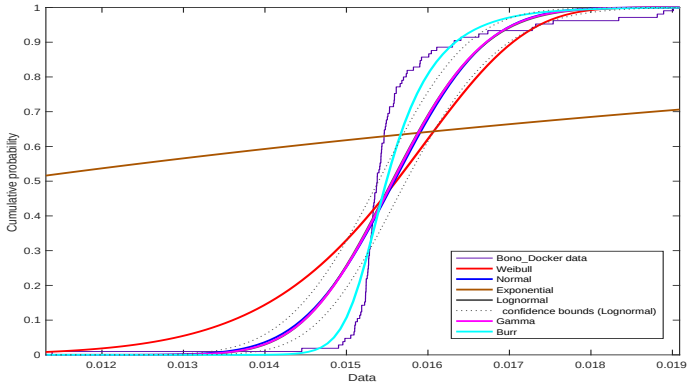
(b) Sprout: Service Time Distribution



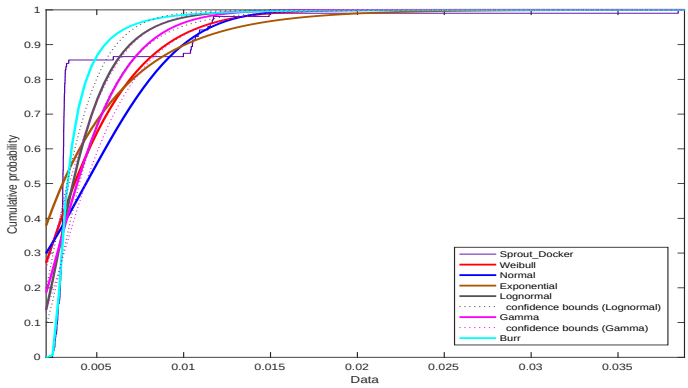
(c) Homestead: Service Time Distribution

Figure 4.10: VMs: Service Time Distribution

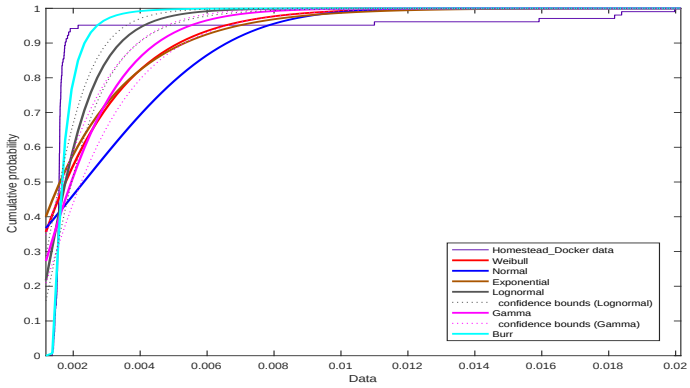
the service time distribution fitting for Bono, Sprout, and Homestead.



(a) Bono: Service Time Distribution



(b) Sprout: Service Time Distribution



(c) Homestead: Service Time Distribution

Figure 4.11: Docker Containers: Service Time Distribution

Figures 4.10 and 4.11 show the actual service time cumulative distribution of the experimental data and those of the various distributions with a confidence interval bounds of 95 %. For illustration purpose, we added the confidence interval bounds to the distributions that mostly fits the data. As shown in the Figures, we chose to fit the empirical distribution to the most popular statistical distributions: Weibull, Normal, Lognormal, Gamma, and Exponential. However, through a closer investigation, we noticed that for most of fittings, the Burr Type XII distribution outperforms the other distributions. The Burr type XII distribution is a three-parameter family of distributions on the positive real line. It is a very 'flexible' distribution that can fit a wide range of empirical data. Different values of its parameters cover a broad set of skewness and kurtosis. Hence, it is used in various fields such as finance, hydrology, and reliability to model a variety of data types.

In addition, the plotting of the cumulative distributions gives an estimation of the model correctness but in cases where more than one distribution may reasonably fit the data, a visual model selection may not be sufficient. To this end, we made use of the well-known Akaike Information Criterion (AIC) estimator. It is an estimator of the relative quality of statistical distribution for a set of data values. Given that we have several distribution, AIC is used to compare the relative quality to the other distributions. The distribution with the smallest AIC value is the preferred model. Due to the fact that our sample sets of the service distributions are limited to 100 values, we make use of the revised version of AIC, i.e., AIC-corrected. This is to avoid any over/under-fitting due to a small set of sample values. AICc can be calculated by applying the following formula:

$$AICc = AIC + \frac{2k^2 + 2k}{n - k - 1} = 2k - 2\ln(L) + \frac{2k^2 + 2k}{n - k - 1}$$

where:

1. k: number of parameters for a certain distribution.

2. L: log likelihood. It is generated for each distribution from Matlab.

3. n: the sample size. One hundred in our case.

Matlab has a bunch of commands to quickly calculate the AICc, or it can be done by writing a simple code in any programming language. After calculating the AICc, the selected distributions for each of VMs and Docker components are shown in Table 4.1. Most of the service time distributions follow the Burr Type XII distribution, except one which follows Weibull.

Table 4.1: Service Time Distribution Fitting Based on AICc

Component	**Distribution**
VM: Bono	Weibull
VM: Sprout	Burr
VM: Homestead	Burr
Docker: Bono	Burr
Docker: Sprout	Burr
Docker: Homestead	Burr

4.2.4 Inter-Arrival Time Distribution Fitting for M/G/1 & G/G/1

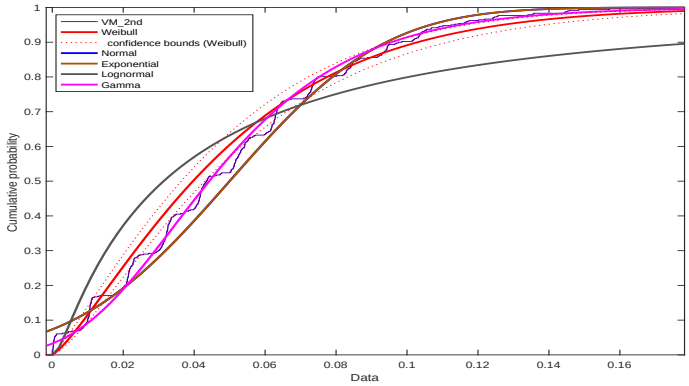
As stated previously, the input of the first queue is predefined to be Poisson with a mean corresponding with the arrival rate. However in theory, the input of the next queue is no more Poisson, and we chose to apply G/G/1 to analyze other queues in the tandem network. Each queue was considered as a single G/G/1 queue.

Similar to the service time distribution fitting above, we performed a distribution fitting for the inter-arrival times of the queues in the tandem. The inter-arrival time of each queue was captured and fetched from Wireshark during registering a total of 2000 subscribers with a rate of 20 Registers per second. Figures 4.12, 6.1, 4.13, and 6.2 show the distribution fitting of the experimental data to the most popular statistical distributions: Weibull, Normal, Exponential, Lognormal, and Gamma. A confidence interval bounds of 95 % was added distributions that most fits the data.

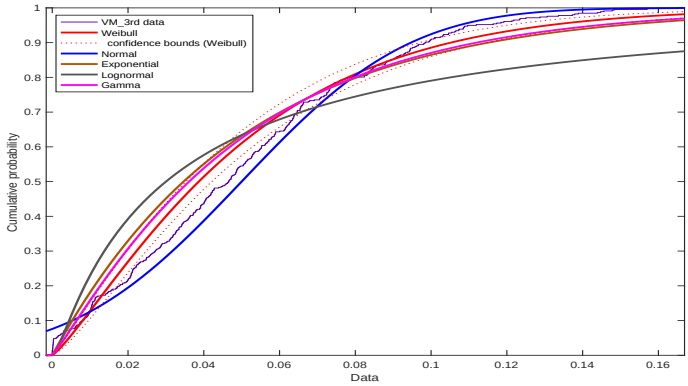
The best statistical distributions that fits the set of data was chosen based on AIC-c. The sample size here is two thousands, hence the AIC version can be used as the sample size is significant:

$$AIC = 2k - 2\ln(L)$$

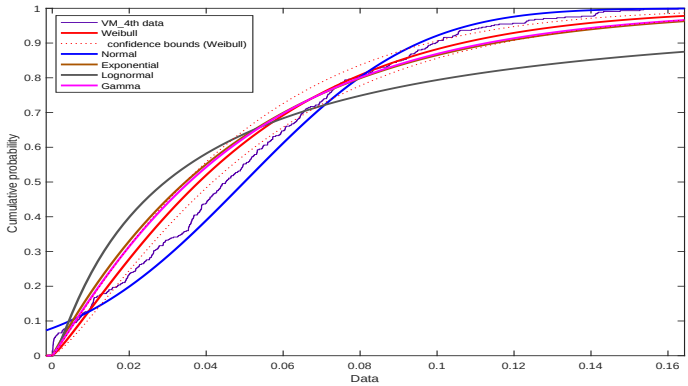
Tables 4.2 and 4.3 show the results of the fitting process. Based on the AIC calculations for most of fittings, Weibull is the most frequent type over the others. Weibull is a continuous probability distribution with two parameters. The first parameter is k $>$ 0 which is the shape parameter, and λ is the scale parameter. Exponential distribution is a special case of the Weibull distribution when the shape parameter equals one. The Weibull distribution is often used in failure analysis, wind speed distributions, electrical and industrial engineering.



(a) VM: Homestead Inter-Arrival Time Distribution

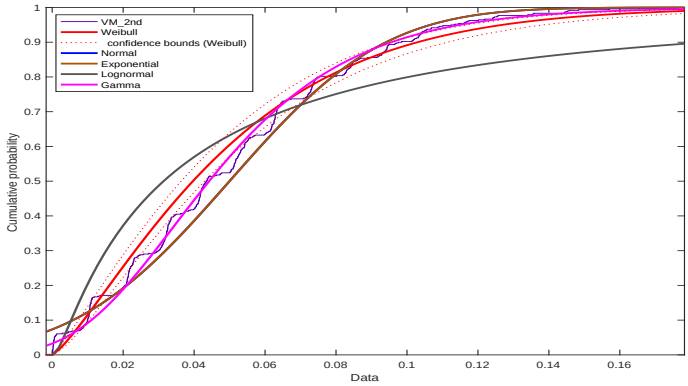


(b) VM: Sprout Inter-Arrival Time Distribution

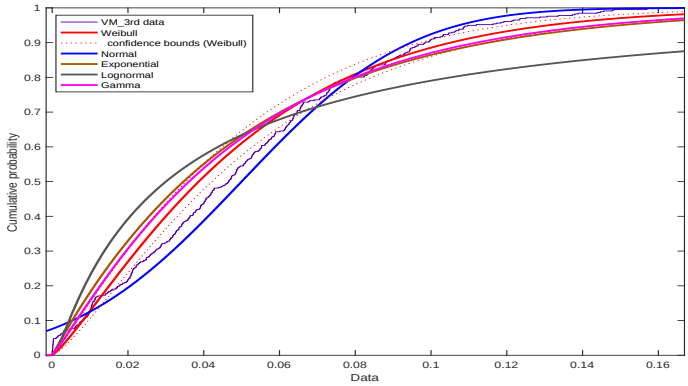


(c) VM: Bono Inter-Arrival Time Distribution

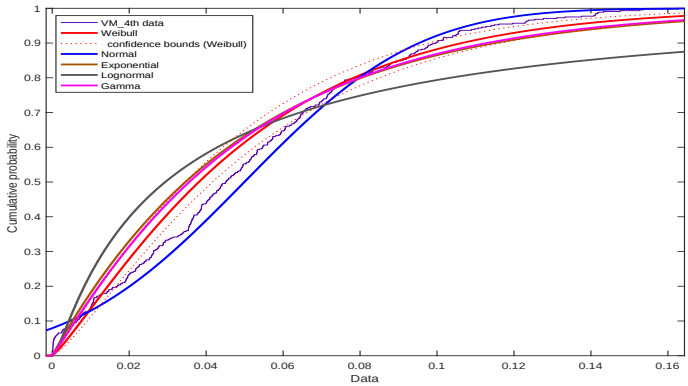
Figure 4.12: VMs Part 1: Inter-Arrival Time Distribution



(a) Docker: Homestead Inter-Arrival Time Distribution



(b) Docker: Sprout Inter-Arrival Time Distribution



(c) Docker: Bono Inter-Arrival Time Distribution

Figure 4.13: Docker Part 1: Inter-Arrival Time Distribution

Table 4.2: VM: Inter-Arrival Time Distribution Fitting Based on AICc

Inter-Arrival Time	**Distribution**
1st	Exponential (Poisson)
2nd	Gamma
3rd	Weibull
4th	Weibull
5th	Weibull
6th	Weibull
7th	Weibull

Table 4.3: Docker: Inter-Arrival Time Distribution Fitting Based on AIC

Inter-Arrival Time	**Distribution**
1st	Exponential (Poisson)
2nd	Weibull
3rd	Weibull
4th	Weibull
5th	Weibull
6th	Weibull
7th	Weibull

Chapter 5
Conclusion and Future Work

Performance analysis and analytical modelling are necessary to understand the integration of new technology services. Questions like how will the service operate under different circumstances, what are the performance constraints and deployment cost of the service are essential in planning. Service providers aim to know the characteristics of the service and what services they can reliably offer to the customers.

In this project, we experimentally and analytically studied the performance of the initial subscriber registration process for a virtualized IMS, called ClearWater. We considered two popular virtualization technologies to deploy ClearWater: VM and Docker containers. According to the experimental analysis, Docker containers produced lower average delay, and consumed less processing power. For this specific use case, VMs utilized less memory compared to Docker containers. A lack of adequate RAM can result in slowing down CPU or disabling operations from execution.

In addition, the project discussed the impact of varying the CPU power, RAM capacity, arrival rate, and scaling up on the system performance. Increasing the number of processing cores has a positive effect on the delay as it was enhanced significantly. However assigning more processing power results in extra operational cost. Scaling up Sprout also improved the average delay, and it might be essential if fault tolerance and high availability are desired. With regard to memory allocation, increasing RAM marginally enhanced the delay. It is recommended to assign more RAM to Docker containers since the RAM consumption is rather high on this specific VNF deployment.

Regarding the analytical analysis, two different queuing models were considered: M/M/1 and M/G/1 combined with G/G/1. The queuing network was abstracted as a feed-forward tandem path. M/M/1 is simple to use and it has a direct formula to find the approximation for the average delay in each queue. However, the M/M/1 queuing model did not provide adequate results when the queue utilization factor (ρ) is around 0.6. On the contrary, assuming general distributions for the service

process, i.e., M/G/1 & G/G/1, which requires definite first and second moments, outputs better results compared to M/M/1 as the relative error did not exceed 12 % for any single value of the arrival rates that meets the stability condition. Therefore, M/G/1 & G/G/1 is the recommended method to model the user registration process of ClearWater as the relative error is minimal.

The project is open for many future works where other attributes can be studied or another queuing model can be used. One suggestion here is to study the call flow parameters such as end-to-end delay, resource consumption, and successful and failure rates. The call scenario is much more complex than what we have studied where more messages being exchanged and more nodes are participated. It is also worth to try using Robust Queuing Theory (RQT) [BBY15] in order to investigate how much it is suitable to utilize a rather recent theory for finding close to exact bounds of queuing systems and networks for both heavy-tail and non heavy-tail distributions.

Chapter 6
Appendices

6.1 Useful Linux Commands

1. Create a file and then press ctrl+d to save: **cat > file1.txt**
2. Show the content of a file and the number of lines: **cat -n file1.txt**
3. Moves all files from the current directory to the directory called /home: **mv * /home**
4. Delete a directory: **rm -rf dire_name**
5. Copy a file from laptop to a remote server: **scp xxx/xxx/file_name username@domain:/xxx/xxx/file_name**
6. Extract tar.gz file: **tar -xzvf file_name**
7. Run tcpdump on specific interface and save it to pcap file: **sudo tcpdump -i interface_name -w file_name.pcap**

6.2 Useful Docker containers Commands

1. Lists the available docker machine: **docker-machine ls**
2. Show the IP addresses of the containers in your deployment: **utils/show_ips.sh**
3. Remove all the docker containers: **sudo docker rm $(sudo docker ps -aq)**
4. Show the docker information: **sudo docker info**
5. Remove all the docker images: **sudo docker rmi $(sudo docker images -aq)**

6. Start and stop docker containers: **sudo service docker start and sudo service docker stop**
7. Show networks status: **sudo docker network ls**
8. List of containers or images: **sudo docker container ls or sudo docker image ls**
9. Remove a certain container or image: **sudo docker container rm -f cont_id or sudo docker image rm -f imag_id**
10. Enter the CLI of a container: **sudo docker exec -it <container name> /bin/bash**
11. Updates the memory and CPU values for a certain docker container: **sudo docker update –memory ... –cpuset-cpus ... <container id>**

6.3 Useful ClearWater Commands

1. Query Chronos nodes over SNMP to get the number of active registrations: **utils/show_registration_count.sh**
2. Start ClearWater deployment: **sudo docker-compose -f minimal-distributed.yaml up -d**
3. Create 1000 subscriber from Ellis node: **cd /usr/share/clearwater/ellis/ sudo env/bin/python src/metaswitch/ellis/tools/create_numbers.py –count 1000**
4. Create 200000 subscriber from using bulk provision script on any Cassandra node: **/usr/share/clearwater/crest-prov/src/metaswitch/crest/tools/stress_provis 200000**
5. Restart ClearWater: **sudo service clearwater-infrastructure restart**
6. Check the cluster status: **sudo clearwater-etcdctl cluster-health**
7. Show the members of the ClearWater cluster: **sudo clearwater-etcdctl member list**
8. Run to regenerate any dependent configuration files: **sudo service clearwater-infrastructure restart**
9. Restart Clearwater node(s) using the following commands:
   a) Sprout: **sudo service sprout quiesce**

b) Bono: **sudo service bono quiesce**

c) Dime: **sudo service homestead stop && sudo service homestead-prov stop && sudo service ralf stop**

d) Homer: **sudo service homer stop**

e) Ellis: **sudo service ellis stop**

f) Vellum: **sudo service astaire stop && sudo service rogers stop**

6.4 XML Scenario

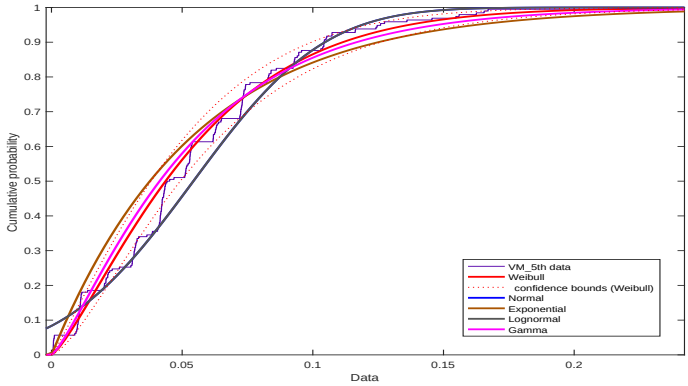
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">

<scenario name="registration">
<send retrans="500">
<![CDATA[
REGISTER sip:[field1] SIP/2.0
Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
Max-Forwards: 70
From: "sipp" <sip:[field0]@[field1]>;tag=[call_number]
To: "sipp" <sip:[field0]@[field1]>
Call-ID: reg///[call_id]
CSeq: 7 REGISTER
Contact: <sip:sipp@[local_ip]:[local_port]>
Expires: 3600
Content-Length: 0
User-Agent: SIPp
]]>
</send>

<recv response="401" auth="true" rtd="true">
</recv>

<send retrans="500">
<![CDATA[
REGISTER sip:[field1] SIP/2.0
Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
Max-Forwards: 70
From: "sipp" <sip:[field0]@[field1]>;tag=[call_number]
To: "sipp" <sip:[field0]@[field1]>
Call-ID: reg///[call_id]
```

```
CSeq: 8 REGISTER
Contact: <sip:sipp@[local_ip]:[local_port]>
Expires: 3600
Content-Length: 0
User-Agent: SIPp
[field2]
]]>
</send>
<recv response="200">
</recv>
</scenario>
```

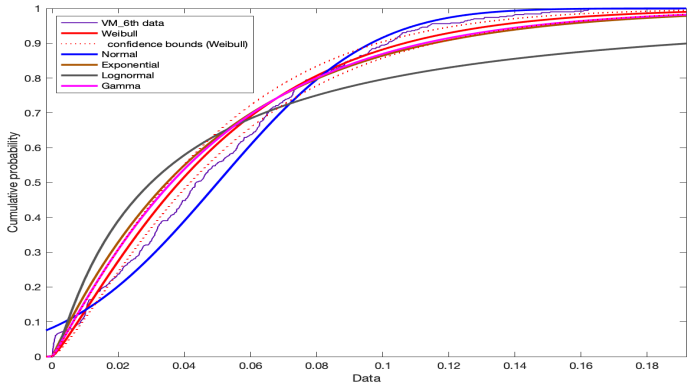
6.5 Inter-Arrival Time Distribution Fitting

6.5.1 VMs

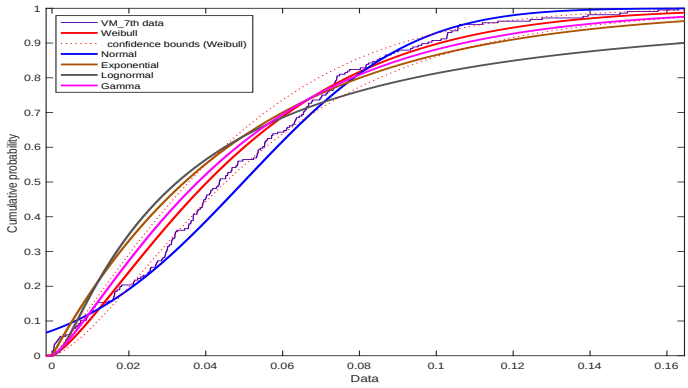
6.5.2 Docker containers



(a) VM: Sprout Inter-Arrival Time Distribution

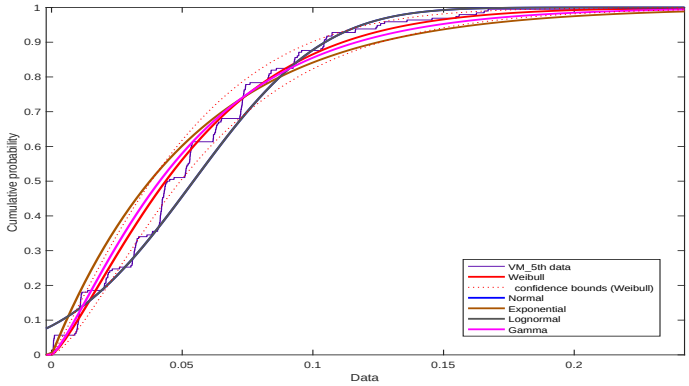


(b) VM: Homestead Inter-Arrival Time Distribution

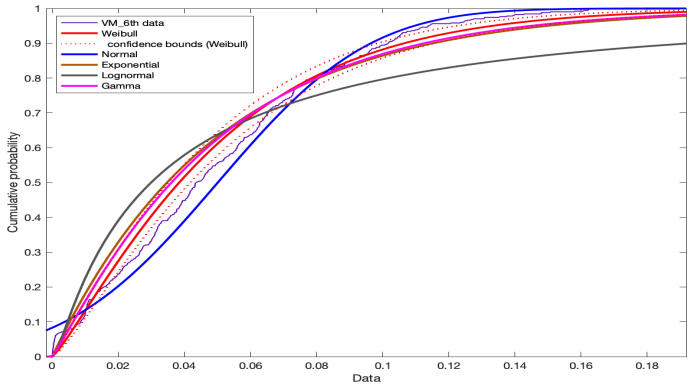


(c) VM: Sprout Inter-Arrival Time Distribution

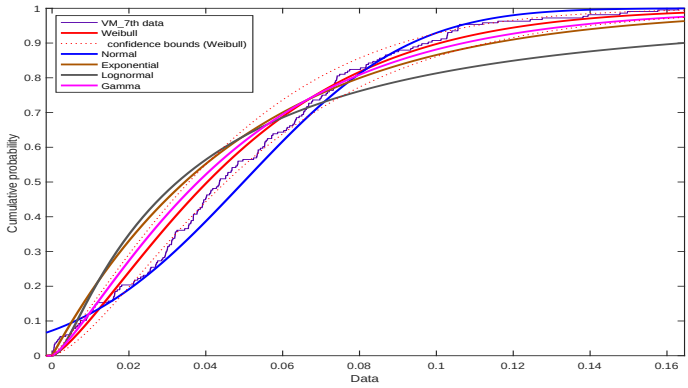
Figure 6.1: VMs Part 2: Inter-Arrival Time Distribution



(a) Docker: Sprout Inter-Arrival Time Distribution



(b) Docker: Homestead Inter-Arrival Time Distribution



(c) Docker: Sprout Inter-Arrival Time Distribution

Figure 6.2: Docker Part 2: Inter-Arrival Time Distribution

References

[BBY15] Chaithanya Bandi, Dimitris Bertsimas, and Nataly Youssef. Robust queueing theory. *Operations Research*, 63(3):676–700, 2015.

[BCC+15] Roberto Bonafiglia, Ivano Cerrato, Francesco Ciaccia, Mario Nemirovsky, and Fulvio Risso. Assessing the performance of virtualization technologies for nfv: A preliminary benchmarking. *2015 Fourth European Workshop on Software Defined Networks*, 2015.

[cle] Clearwater docker. https://github.com/Metaswitch/clearwater-docker.

[CW18] Wei-Kuo Chiang and Juin-Wei Wen. Design and experiment of nfv-based virtualized ip multimedia subsystem. 2018.

[Doc18] Docker. What is a container. https://www.docker.com/resources/what-container, 2018.

[ETS13a] ETSI. Network functions virtualisation (nfv); architectural framework. https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf, October 2013.

[ETS13b] ETSI. Nfv white paper. https://portal.etsi.org/nfv/nfv_white_paper2.pdf, October 2013.

[ETS13c] GSNFV ETSI. Network functions virtualisation (nfv); use cases. *V1*, 1:2013–10, 2013.

[FCP06] Hanane Fathi, Shyam S Chakraborty, and Ramjee Prasad. Optimization of sip session setup delay for voip in 3g wireless networks. *IEEE Transactions on Mobile Computing*, 5(9):1121–1132, 2006.

[Jej18] Fredrik Jejdling. Ericsson mobility report. https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-november-2018.pdf, November 2018.

[Kin61] J. F. C. Kingman. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society*, 57(4):902–904, 1961.

[MG10] A. Munir and A. Gordon-Ross. Sip-based ims signaling analysis for wimax-3g interworking architectures. *IEEE Transactions on Mobile Computing*, 9(5):733–750, May 2010.

[Mun08] A. Munir. Analysis of sip-based ims session establishment signaling for wimax-3g networks. In *Fourth International Conference on Networking and Services (icns 2008)*, pages 282–287, March 2008.

[NC13] Network Functions Virtualisation NFV and Use Cases. Etsi gs nfv 001 v1. 1.1 (2013-10). 2013.

[Neta] Metaswitch Networks. Clearwater. http://www.projectclearwater.org/.

[Netb] Metaswitch Networks. Clearwater. https://clearwater.readthedocs.io/en/stable/.

[NSV17] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. Nfvperf: Online performance monitoring and bottleneck detection for nfv. 2017.

[NTN13] L. Nagy, J. Tombal, and V. Novotny. Proposal of a queueing model for simulation of advanced telecommunication services over ims architecture. In *2013 36th International Conference on Telecommunications and Signal Processing (TSP)*, pages 326–330, July 2013.

[PAR+17] Jonathan Prados-Garzon, Pablo Ameigeiras, Juan J. Ramos-Muñoz, Pilar Andres-Maldonado, and Juan M. López-Soler. Analytical modeling for virtualized network functions. *CoRR*, 2017.

[Pol30] Felix Pollaczek. Über eine aufgabe der wahrscheinlichkeitstheorie. i. *Mathematische Zeitschrift*, 32(1):64–100, Dec 1930.

[Rep] Market Research Reports. Global network function virtualization market 2016-2020.

[sip] Sipp. http://sipp.sourceforge.net/.

[SSFS17] Amit Sheoran, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. Contain-ed: An nfv micro-service system for containing e2e latency. 2017.

[SSJ19] Besmir Tola Sachin Sharma, Navdeep Uniyal and Yuming Jiang. On monolithic and microservice deployment of network functions. *Accepted in IEEE Netsoft 2019*, 2019.

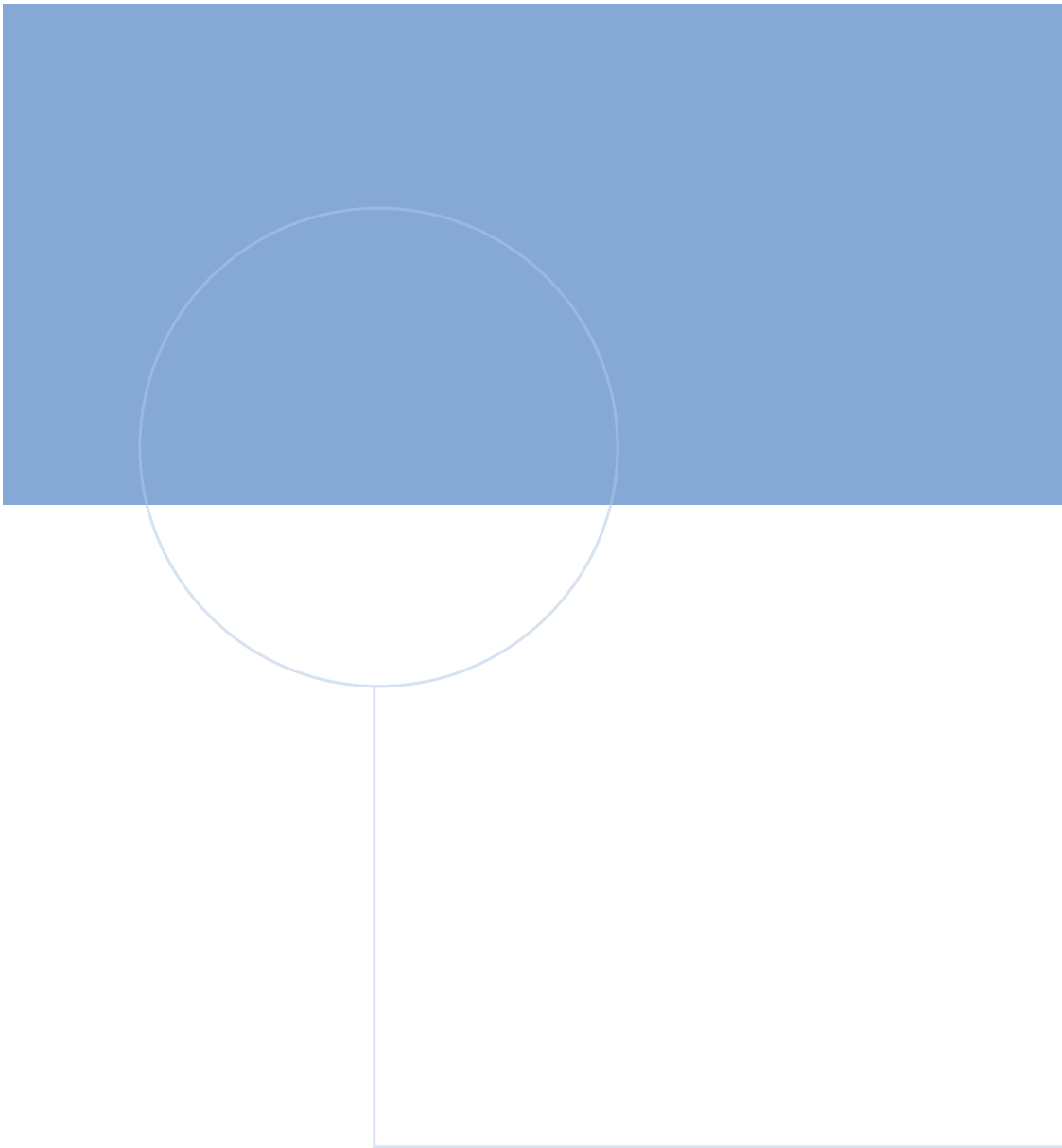
[Sys] GNU Operating System. Gsl - gnu scientific library. https://www.gnu.org/software/gsl/.

[ubu] Ubuntu releases. http://releases.ubuntu.com/trusty/.

[WBBD05] Wei Wu, Nilanjan Banerjee, Kalyan Basu, and Sajal K Das. Sip-based vertical handoff between wwans and wlans. *IEEE Wireless Communications*, 12(3):66–72, 2005.

[Wea] Weaveworks. Weave scope. https://www.weave.works/docs/scope/latest/installing/.

[Zuk13] Moshe Zukerman. Introduction to queueing theory and stochastic teletraffic models. *arXiv preprint arXiv:1307.2968*, 2013.



NTNU

Kunnskap for en bedre verden