REALISTIC PERFORMANCE EVALUATION OF EDGE COMPUTING SCENARIOS OVER A HYBRID TESTBED USING SOFTWARE-DEFINED NETWORKING

by

Raşit Mete Eşrefoğlu

B.S., Computer Engineering, Boğaziçi University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2018

REALISTIC PERFORMANCE EVALUATION OF EDGE COMPUTING SCENARIOS OVER A HYBRID TESTBED USING SOFTWARE-DEFINED NETWORKING

APPROVED BY:

Prof. Cem Ersoy (Thesis Supervisor)	.....
Assist. Prof. Bahri Atay Özgövde (Thesis Co-supervisor)	.....
Prof. Mutlu Koca	.....
Assist. Prof. Ahmet Teoman Naskalı	.....
Prof. Tuna Tuğcu	.....

DATE OF APPROVAL: 02.01.2018

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my thesis supervisor and my role model Prof. Cem Ersoy. He never stopped believing in me and he always supported me with his encouraging comments on top of his always on-point guidance. I can't thank him enough for that and this thesis would not be completed without him.

I am also obliged to my co-advisor, Assist. Prof. Bahri Atay Özgövde, for his invaluable assistance and efforts to keep me focused and dedicated while forcing me to make every piece of this thesis a cut above with his wisdom and notions.

I would also like to express my thankfulness to Ahmet Cihat Baktır for his much-needed perspective in so many parts of this thesis at much needed times. I would like to thank all my friends and colleagues at NETLAB, especially Orhan Ermiş for sharing his vast experience, for his support and encouraging comments.

I am also appreciative to my company, Softtech, for letting me take the time I needed to complete the milestones of this thesis whenever it was necessary. I would like to express my gratitude to my team manager Selime Işık Kantar, my manager Hamza Metin Yarcı, my director Cem Kayan and to all of my other colleagues at Softtech for their inestimable supportive approaches and understandings.

Lastly, I would like to thank my parents, my brothers and my sister for making this happen with their unequivocal love, support and trust throughout my life.

This research has been supported by Boğaziçi University Research Fund (BAP) under the grant number 12663.

ABSTRACT

REALISTIC PERFORMANCE EVALUATION OF EDGE COMPUTING SCENARIOS OVER A HYBRID TESTBED USING SOFTWARE-DEFINED NETWORKING

As mobile computing becomes more and more widespread along with more complex and distributed applications, the importance of providing these applications to users at high speed and with high efficiency is increasing. One of the ways to better serve mobile users is cloud computing. However, sending data to be processed to data centers and receiving processed data take time because of the geographical separation of resources. Since users are less inclined to wait while they use their mobile applications, edge computing offers a better way to handle the QoS requirements for mobile users in the envisaged fifth generation (5G) mobile networks. Network maintenance and network monitoring have also a significant place if we aim to provide faster and more efficient services to users with edge computing. Therefore, an architecture which is built using OpenFlow based Software Defined Networks offers even more benefits in the context of 5G. To this end, the testbeds for Software Defined Networks are established in order to test the real-life scenarios. These testbeds are usually created either using Mininet or on real hardware. Even though they have great advantages, Mininet testbeds do not produce precise results and real testbeds are not easy to expand. In this study, we aim to bring these two approaches together and provide a hybrid testbed that utilizes the features of both approaches for those who wish to perform experimentation in this field. In the experiments we have done, we presented the comparative evaluation results obtained using Mininet and our hybrid testbed for the desired edge computing scenarios. We observed different results under the same experimental variables and examined these differences with their reasons.

ÖZET

YAZILIM TANIMLI AĞLAR KULLANARAK KENAR BİLİŞİM SENARYOLARININ HİBRİT TEST ORTAMI ÜZERİNDE GERÇEKÇİ BAŞARIM DEĞERLENDİRMESİ

Mobil bilişimin giderek daha da yaygınlaştığı, daha karmaşık ve dağıtık uygulamaların ortaya çıktığı günümüzde, bu uygulamaları kullanıcılara yüksek hızlarda ve daha etkin sunmanın önemi de artmıştır. Hareketli kullanıcılara daha iyi hizmet vermenin yollarından biri bulut bilişimdir. Ancak veri merkezlerinde işlenecek verileri gönderip işlenmiş verileri almak coğrafik mesafelerden ötürü zaman almaktadır. Hareketli kullanıcılar, uygulamaları kullanırken beklemeye çok daha az meyilli olduklarından, kenar bilişim, öngörülen beşinci nesil (5G) mobil ağlarda hareketli kullanıcılar için servis kalitesi gereksinimlerini karşılamanın daha iyi bir yolunu sunuyor. Kullanıcılara kenar bilişim kullanılarak daha hızlı, daha etkin hizmet vermek amaçlanırken, ağ bakımı ve ağ gözetimi de önemli bir yer tutmaktadır. Bu yüzden, 5G bağlamında OpenFlow tabanlı Yazılım Tanımlı Ağlar kullanılarak kurulmuş bir mimari daha fazla avantaj sağlamaktadır. Gerçek hayattaki senaryoları test etmek amacı ile Yazılım Tanımlı Ağlar için test ortamları kurulmaktadır. Bu test ortamları ya tamamen Mininet kullanılarak ya da tamamen gerçek donanımlar kullanılarak oluşturuluyor. İki yöntemin de birçok avantajları olsa da, Mininet ile tam doğru ölçumler alamayabiliyoruz ve gerçek test yataklarında da ölçeklenebilirlik sorunu ile karşılaşıyoruz. Bu çalışmada her iki yaklaşımı bir araya getirerek melez bir test ortamı sunup, deneyi tasarlayacakların ihtiyaçlarına göre her iki tarafın özelliklerinden faydalanacak düzenekler yaratmasına olanak sağladık. Yaptığımız deneylerle de hedeflenen kenar bilişim senaryolarını Mininette ve melez ortamda karşılaştırmalı olarak ortaya koyduk. Aynı deney değişkenleri altında farklı sonuçlar oluştuğunu gözlemledik ve bu farkları sebepleriyle inceledik.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xi
LIST OF SYMBOLS . . . xii
LIST OF ACRONYMS/ABBREVIATIONS . . . xiii
1. INTRODUCTION . . . 1
1.1. Contributions . . . 3
1.2. Thesis Outline . . . 4
2. BACKGROUND ON SOFTWARE-DEFINED NETWORKING AND EDGE COMPUTING . . . 6
2.1. Software-Defined Networking . . . 6
2.1.1. Programmable Networks . . . 7
2.1.2. Separation of Planes in SDN . . . 8
2.1.3. OpenFlow and Open vSwitch . . . 10
2.1.4. Controllers and Northbound Applications . . . 12
2.1.5. Mininet . . . 14
2.2. Edge Computing . . . 17
2.2.1. Cloud Computing . . . 17
2.2.1.1. Infrastructure . . . 18
2.2.1.2. The Actors Involved in Cloud Computing . . . 19
2.2.1.3. Business Models . . . 20
2.2.2. Edge Computing and Its Derivatives . . . 21
3. ROAD TO HYBRID TESTBED ENVIRONMENT CREATION USING SDN 25
3.1. Experimental Setup . . . 25
3.2. Environments . . . 28
3.2.1. Pure Mininet Environments . . . 29
3.2.2. Hybrid-1 Environments . . . 31

3.2.3. Hybrid-2 Environments 32
3.2.4. Hybrid-3 Environments 34
3.3. RYU SDN Controller 37
3.4. Northbound Applications 38
3.4.1. Topology Discoverer 40
3.4.2. Traffic Monitor 41
3.5. Automated Topology Creation for the Environments 42
4. APPLICATIONS FOR REALISTIC EDGE COMPUTING SCENARIOS . 44
4.1. Server-Client Communication Design 44
4.1.1. Request Schema with Poisson Point Process 44
4.1.2. Background Traffic Generation 45
4.2. Synthetic Workload Generation 46
4.3. Face Detection with OpenCV 49
5. PERFORMANCE EVALUATION WITH HYBRID TESTBEDS 55
5.1. Logging Mechanisms and Performance Charts 55
5.1.1. Cloudlet Link Load 55
5.1.2. Cloudlet CPU Usage 57
5.1.3. Service Delays for the Hosts 58
5.1.4. Event Logs for the Hosts 58
5.2. Comparative Evaluation of the Test Environments 60
6. CONCLUSION 72
REFERENCES 74

LIST OF FIGURES

Figure 2.1. Conventional Networks vs SDN 9

Figure 2.2. A Flow table entry in OpenFlow 11

Figure 2.3. A conventional network architecture for data centers 19

Figure 3.1. Final Form of the Hardware Setup (Hybrid-3) 26

Figure 3.2. Summary of the Environment Representations 29

Figure 3.3. Pure Mininet with one Cloudlet Network Topology (Env-1) 30

Figure 3.4. Mininet with one Real Cloudlet Network Topology (Env-2) 31

Figure 3.5. Mininet with one Real Cloudlet and one Real Switch Network Topology (Env-3) 34

Figure 3.6. Mininet with Real Cloudlet, one Real Switch, three Raspberry Pi's Network Topology (Env-4) 36

Figure 3.7. Real-life correspondence of Figure 3.6 36

Figure 3.8. The Northbound in our SDN Environment 39

Figure 4.1. Service Durations for the requests under heavy CPU utilization, from top to bottom mean interarrival time between requests are 5-10-15-20-25-30-35-40 seconds 48

Figure 4.2. Server-Client Interaction in Face Detection Application 50

Figure 4.3. Pseudo-code for the image send and receive with TCP socket programming. 51

Figure 4.4. Pseudo-code for the server code runs on the cloudlets. 52

Figure 4.5. Pseudo-code for the client code runs on the hosts. 53

Figure 4.6. One Example of the Images Used (taken from Google Images) . . 54

Figure 5.1. Cloudlet Link Loads under different background traffic levels and different requests per second 56

Figure 5.2. CPU utilization of Real Cloudlet PC under various loads 57

Figure 5.3. Env-4, Event Log for 0,05 Req/Sec with 40Mbps Background Traffic 59

Figure 5.4. Average Service Delays of Env-1-2-3-4 with 40Mbps Background Traffic 61

Figure 5.5. Average Service Delays of Env-5-6-7-8 with 40Mbps Background Traffic 63

Figure 5.6. Average of Average Service Delays for the Real and Virtual Cloudlets in Env-5-6-7-8 with 40Mbps Background Traffic 63

Figure 5.7. Average Service Delays for Env-1 and Env-4 with Different Background Traffic Modes 66

Figure 5.8. Average Service Delays for Env-1 and Env-3 when Different Number of Hosts Used 68

Figure 5.9. Average Service Delays for Env-1 and Env-3 under Light Request Load and Heavy Request Load 69

Figure 5.10. Mininet Hosts vs Raspberry Hosts Average Service Delay comparison for Env-4 and Env-8 71

LIST OF TABLES

Table 3.1. Ethernet Cards Used 28

Table 3.2. RYU Controller Dispatchers 41

LIST OF SYMBOLS

λ	Request rate (Requests/second)

LIST OF ACRONYMS/ABBREVIATIONS

API	Application Programming Interface
ARP	Address Resolution Protocol
CLI	Command-line Interface
CRM	Customer Relationship Management
IaaS	Infrastructure as a Service
IoT	Internet of Things
LAN	Local Area Network
MAC	Media Access Control
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
NFV	Network Functions Virtualization
OF	OpenFlow
ONF	Open Networking Foundation
OVS	Open vSwitch
PaaS	Platform as a Service
QoS	Quality of Service
RAN	Radio Access Network
SaaS	Software as a Service
SDN	Software-Defined Networking
TCP	Transmission Control Protocol
ToR	Top of the Rack
UDP	User Datagram Protocol
WAN	Wide Area Network
WLAN	Wireless Local Area Network

1. INTRODUCTION

The dramatic explosion in the number of devices connected to the Internet over the last decade directed researchers' attention to the inadequacy of the current network infrastructures that we have today. Although there is a comprehensive adoption of the conventional IP networks, they are pretty complex as well as they are hard to maintain and manage [1]. Moreover, current networks are vertically integrated, or in other words, the control plane that decides how the network traffic will be managed and the data plane that actually forwards the data to the desired destinations are bundled all together. This togetherness leads to network products reducing the pace of innovations and evolution and lowering the flexibility of the networking infrastructure [2, 3]. Because of all these reasons, the search for a better solution for the networking to expedite the innovative progress began.

Software-Defined Networking (SDN) is an emerging networking paradigm that is considered to resolve the limitations of the current network infrastructures [4]. The key idea behind the SDN is decoupling the control plane and the data plane. In SDN, the control plane has the global view of the topology and is responsible for calculating the routing algorithms. It also monitors the network state as needed and can take the necessary actions. For example, in case of a link failure between switches or in case of a need for load balancing, the control plane decides on the new routes. Whereas, the data plane (forwarding plane) consists of relatively dumb switches connected to the centralized network logic and this plane is responsible for forwarding the packets according to the rules on the switches where these rules acquired from the control plane. This centralized control mechanism can easily simplify the management of the network while increasing the capabilities. On the other hand, SDN smooths out asperities of the network virtualization in every aspect [5]

On the other hand, cloud computing is another relatively new area of technology that aims to provide on-demand computing resources such as processing power, storage and networking. It promises several attractive benefits for businesses and end users.

For example, companies that get cloud service provisions can scale up the necessary resources they need as demand increases and then scale down again as demand decreases. Pay per use model is applied so that computing resources are measured at a granular level, allowing users to pay only for the resources and workloads they use without deploying huge infrastructures for their own. From the end users' perspective, cloud computing is the key technology as their number of devices that are connected to the Internet but are incapable of self-providing computing resources they need for their applications increase. The inadequacy of computing resources can be resolved with cloud computing. To make this possible, cloud computing providers establish very large data centers to overcome numerous customers' demand. Although resource management for the cloud providers is more convenient because of the singularity of their establishment with one or a few very large data centers, the users of the cloud undergo network latency since there is usually a considerable geographic distance between them.

To tackle the challenges that we face for mobile computing and to circumvent the problems that remoteness of cloud computing brings forth, Edge Computing has another approach. It is based on the idea of setting up an environment which consists of comparatively less capable servers yet with more distributed structure and closer proximity to the end users. Thus, it can be said that Edge Computing devotes to Mobile Computing and efficient data streaming via caching and/or compressing the relevant data to be localized at the edge of the mobile network [6]. In this regard, Edge Computing can be considered as a method of accelerating and improving the performance of cloud computing for mobile users. Since the end users are less disposed to wait for the latency and this kind of an architecture partially prevents the network latency which is a natural outcome of Wide Area Networks (WAN), edge computing seems a promising solution.

In this study, we try to combine Software-Defined Networking and Edge Computing. The control mechanisms achieved by SDN can lower the complexity of the Edge Computing architectures which need to be taken care of in an efficient manner by utilizing the available resources. In a system that confederates less capable edge servers

and the traditional powerful cloud servers reside data centers, the traffic originated at the edge can be dynamically routed to the server that may provide better quality service to the user by using the available SDN mechanisms. SDN consolidates the network intelligence at the central controller and simpler edge devices are relieved from executing the complex networking activities. This type of architectural design mitigates the barriers and complications that edge devices encounter and users of these edge devices get service with less average waiting time (i.e., less average service delay). Although we do not examine and quantify the performance metrics beyond edge servers in this study, benefits of such combination (edge computing-cloud computing) controlled by SDN is apparent. This is why, in this study, we compose a system on a hybrid testbed consists of virtual machines and real hardware in order to analyze the performance of edge computing scenarios in which networking operations are managed by Software-Defined Networking.

1.1. Contributions

In this thesis, we concentrated on building a hybrid testbed for realistic edge computing scenarios and provided different kinds of environment models along with performance evaluation methods for the testbeds while exploiting SDN's features. Although building a completely virtual SDN emulation environment is possible via Mininet, adding real computers/hosts into the scene creates additional opportunities to make analyses more dependable. We can summarize the contributions of this thesis as follows:

- The main contribution of this thesis is the creation of a fully functional and configurable hybrid testbed that enables to run experiments and evaluate/monitor the performance of the Edge Computing scenarios for this study and the future studies.
- We implemented a "Topology Creator" application which sets up the Mininet topology with various adjustable parameters such as the number of hosts, the number of switches, the connections between the hosts and the switches. This

application also makes possible to add real components into the topology in order to create a hybrid-testbed. Besides, we used this application like a dynamic hub for the distribution of the Edge Computing scenario applications to the related components such as real/virtual cloudlets or Mininet Hosts/Raspberry Pis in accordance with the different levels of hybridization of the testbed.

- After building a complete Software-Defined Networking Environment using RYU controller and Mininet, we implemented two northbound applications, namely, "Topology Discoverer" and "Traffic Monitor". Topology Discoverer application registers the data paths between switches. In addition to that, it registers the MAC addresses that connected to each Open vSwitches' occupied ports to the Controller. Whereas Traffic Monitor application monitors the link loads, specifically on the cloudlet link which is between the cloudlet and the switch it is connected to.
- To test the edge computing scenarios, we present use-case approaches such as face detection on cloudlet, adjustable background traffic on the topology and synthetic CPU load producing for constant service rate.
- We provided methods to evaluate the performance of the edge computing scenarios in which the service delays for each host that gets provision from the cloudlet can be measured according to different parameters. Besides, cloudlet link load and CPU usage can be monitored with changing parameters.
- We show that there are notable differences in terms of performance and the similarity to real-life situations between the fully virtual environments and the hybrid environments. Obviously, the closer it gets from the virtual to the physical environment the more realistic the results are.

1.2. Thesis Outline

Chapter 2 presents a review of the necessary literature background for Software-Defined Networking and Edge Computing.

Chapter 3 describes, step-by-step, how we achieved to establish our SDN platform with northbound applications running on it and hybrid testbed infrastructure in four

phases. The first phase is “Pure Mininet”, the second phase is “Mininet with a Real Cloudlet”, the third phase is “Mininet with a Real Cloudlet and a Real Switch” and the last phase is “Mininet with a Real Cloudlet, a Real Switch and Raspberry Pi Hosts”

In Chapter 4, the implementation details of the use-case applications that are used in the edge computing scenarios are explained. Besides, we describe various ways to make use of our testbed with these use-case applications.

Chapter 5 presents the tools and charts which is used to evaluate performance for the experimentations and future studies that will take place on our hybrid testbed for edge computing scenarios.

In chapter 6, we make the conclusion for this thesis and present future work in the related research field.

2. BACKGROUND ON SOFTWARE-DEFINED NETWORKING AND EDGE COMPUTING

2.1. Software-Defined Networking

Traditional computer networks are typically built from a great number of network devices, such as routers and switches. Besides, there are a large number of middleboxes (e.g., firewalls, WAN optimizers, load balancers), devices that manage traffic for purposes other than packet forwarding exist in the traditional networks with many complex protocols implemented on each of them. Network operators are obliged to configure policies to respond to a wide range of network applications and activities. While readjusting to dynamic network conditions, they have to manually alter these high-level policies into low-level configuration commands. As a consequence, network management and performance tuning are very difficult and thus cause errors. The fact that network devices are often vertically integrated black boxes also increases the challenges for network operators and administrators. Another difficulty faced by practitioners and researchers is called "Internet ossification". Because of the fact that the Internet has come together as a part of the broad and critical infrastructure of our society (just like transportation system and power grids), the Internet has become extremely difficult to transform both in terms of its physical infrastructure as well as its protocols and performance [7]. Nonetheless, as current and emerging Internet applications and services become increasingly complex and challenging, the Internet must be able to evolve to solve these new challenges.

"Programmable networks" is the idea that has been proposed to expedite the evolution of computer networks. In this context, Software Defined Networking is a new networking paradigm in which the forwarding plane that forwards the actual data is decoupled from the control plane that decides how these forwarding actions will be performed. The main idea is to allow software developers to exploit network resources as they do computing resources today. In SDN, software-based controllers (the control

plane) are the intelligence, whereas network devices are just simple packet forwarding devices (the data plane) that can be programmed via an open interface like ForCES [8], OpenFlow [9], protocol-oblivious forwarding POF [10].

2.1.1. Programmable Networks

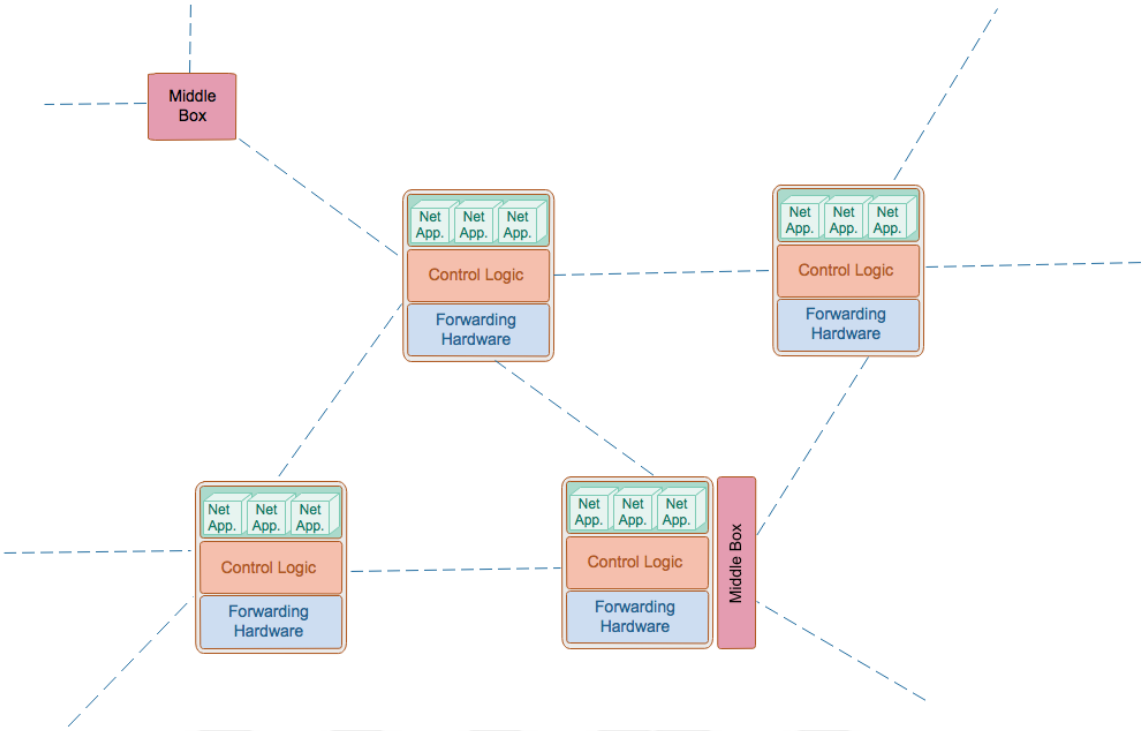
Despite the fact that SDN proposed benefits that range from centralized control over the network, simplifying the network management, lowering the prices of network hardware, eliminating middleboxes and enabling the rapid innovation pace for the third-party applications, it is not the first programmable network idea in the history of computer networks. In [11], Open Signaling (OPENSIG) research, the goal was to make networks more extensible, open and programmable and they suggested that the control software and communication hardware should be separated. The essence of their proposal was to break the vertical integration of the network hardware and access it via the programmable interface in order to make the rapid deployment of the new network services possible.

In [12], [13], Active Networking which drew massive attention at that time, proposed two approaches. One of them was user programmable switches and the other one was capsules, program fragments carried by user messages, that can be interpreted by routers. However, Active Networking could not make it to the industry because of the security and performance concerns. The research 4D Project [14], proposed separation of the planes where the decision plane that has the global view of the network. Later on, this research influenced the work NOX [15], which proposed network operating system of the OpenFlow-enabled network. The Ethane project [16], [17] which led to the creation of OpenFlow, proposed a flow-level solution that is managed by logically centralized controller that has high-level security policies for enterprise networks. It also has Ethane switches in which flow tables are kept for the data forwarding plane.

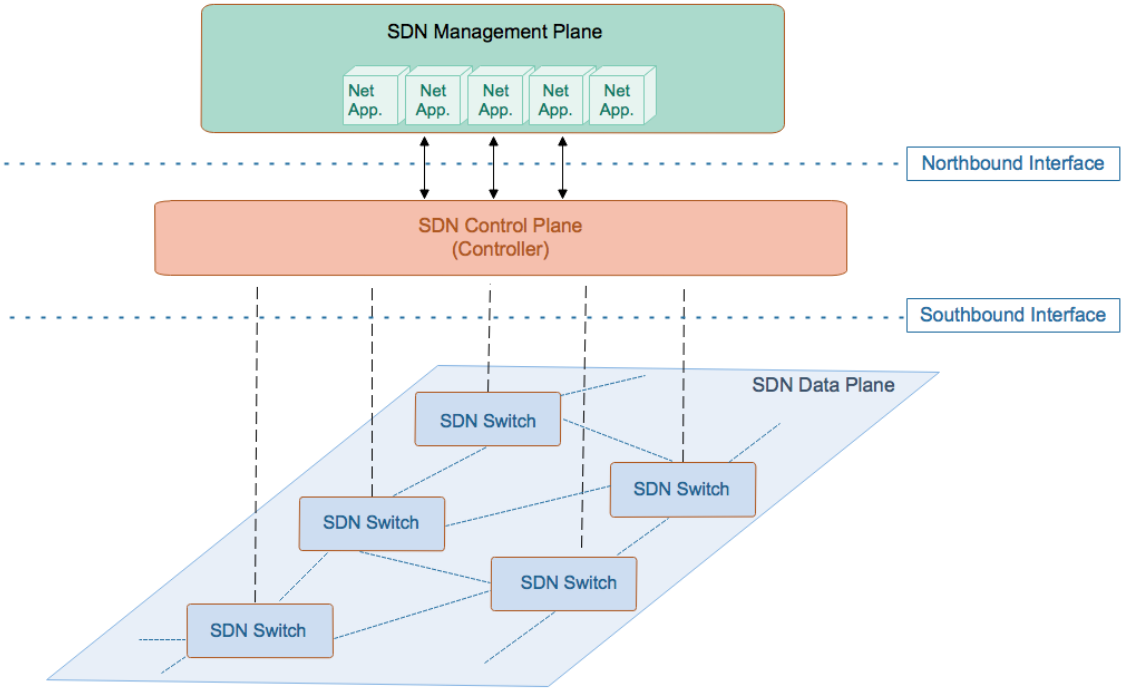
2.1.2. Separation of Planes in SDN

Years ago, the computer systems in Information and Communications Technology (ICT) were vertically integrated. The technology was built upon specialized hardware, specialized operating systems and specialized applications. In this way, the systems were closed and proprietary which led to slow innovation and small industry. Later on, the ICT has evolved in a way that the hardware was independent from the operating systems such as Windows, Linux, Mac OS where the interface between the operating systems and the microprocessors are open. On top of this, the applications that are used were not specialized anymore. In other words, the developers could start contributing by implementing the applications they needed using the open interfaces that operating systems provided that led to rapid innovation and a huge industry.

The analogy is obvious, the conventional networking system rests on specialized hardware, specialized control logic and specialized features which are predetermined by the network device vendors that again leads to slow innovation pace and small industry driven by monopoly vendors. On the other hand, SDN is based on the idea of decoupling the controlling functionality and the networking functionality. The control functionality is achieved with management plane that are applications implemented by developers and the control plane that possess the whole view of the network [2]. Although the control plane managed by the SDN controllers are logically centralized rather than physically centralized [18], to assure an adequate level of performance, scalability, and reliability. In fact, real-life SDN network designs comprise physically distributed control planes [18], [19]. The networking functionality is achieved with forwarding hardware, the building blocks of the data plane, which is comparatively cheaper and has less computing power that takes instructions from the control plane to forward the packets. The communication between the planes is carried out with open application programming interfaces (API). The northbound interface is the interface between the management plane and the control plane, the southbound interface is between the control plane and the data plane. In Figure 2.1(b), separation of the planes and the interfaces between them in SDN paradigm mentioned above can be seen as well as the structural difference between conventional networking and SDN.



(a) Conventional Networking Structure



(b) Seperation of Planes in Software-Defined Networking

Figure 2.1. Conventional Networks vs SDN

2.1.3. OpenFlow and Open vSwitch

McKeown *et al.* [9], started OpenFlow project aiming to create a standardization for the communication between the SDN controllers and the forwarding devices at Stanford University. OpenFlow allows controllers to manage the packet forwarding mechanism for the switches from different vendors. Besides, the purpose was to enable this protocol for not only virtual switches but also physical switches. To this extent, the OpenFlow is considered as an enabler for the Software-Defined Networking. Major leading tech companies collaborate and work with The Open Networking Foundation (ONF), an organization which promotes the adoption of SDN and Network Function Virtualization (NFV). Currently, ONF manages and improves OpenFlow protocol as the first SDN standard for the southbound interface.

Most of the modern Ethernet switches and routers already had a flow table to implement network functionalities such as Firewall, NAT or QoS. Although the switches from different vendors have different flow tables, OpenFlow exploited the common set of functions in these switches to come up with the instruction set for the protocol [9]. From this aspect, OpenFlow is very similar to an x86 instruction set, well-defined instruction set for the networking. OpenFlow is a flow-based protocol for the OF-enabled switches in which the flow entries kept inside a table called Flow Table. The flows that are being forwarded match with a rule and action in that flow entry takes place by the OF-enabled switch. To elaborate a flow entry, it consists of a packet header that defines the flow, an action that defines how the packets should be processed and statistics for the flows matched with this entry in terms of the number of packets and bytes for each flow [9]. Figure 2.2 shows the presented structure of a flow entry in the context of OpenFlow. If an incoming flow does not match with any rule in the flow tables, the packets of the flow are forwarded to SDN controller. The controller decides what to perform on these packets of the flow. It either drops the packet or installs a new rule on the switches for the incoming packets belong to this flow.

Open vSwitch (OVS) [20] is a relatively new virtual switch technology to overcome the massive networking virtualization need in cloud computing and in fact heavily

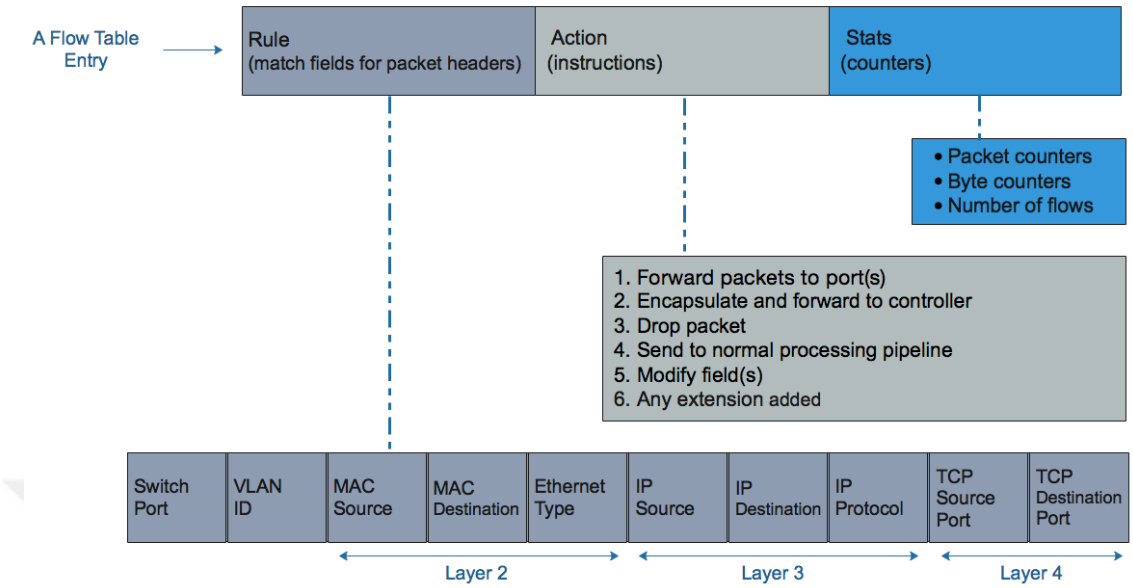


Figure 2.2. A Flow table entry in OpenFlow

used in cloud computing frameworks such as OpenStack and Open Nebula. Nowadays, it is not unusual that a single server contains hundreds of Virtual Machines (VM) [21] in a data center, thus leading to the emergence of virtual networking. The network connection of these great number of VMs has to be achieved via not physical network interfaces but virtual network interfaces. Open vSwitch has introduced a way to optimize virtualization for the software switching architectures, to enable adding the virtual network interfaces to this switching mechanism [22]. Albeit the previous virtual switch technologies like Linux bridges are fast and reliable, OVS is proposed to respond network dynamics for the virtual environments that are often characterized by high rates of change. In [23], Emmerich, Raumer, Wohlfart, and Carle conducted the performance evaluation and comparison of the Open vSwitch and other virtual switching technologies such as IP forwarding, Linux bridge and DPDK vSwitch under various scenarios. Their research concludes that OVS performs fairly better than other virtual switching techniques.

Network namespaces, allows different processes to have different views of the network were started to be supported since Linux 2.2.26 distribution on kernel-level.

Since Linux 3.3 distribution, OVS is also included in Linux Kernel as a module and it works with most hypervisors such as Xen, KVM or QEMU and container systems such as Docker on Linux. That being said, network emulation has become a very useful application of Open vSwitch. When emulating the networks, different network namespaces for different applications can have its own virtual Ethernet cards. These Ethernet cards can be connected to the virtual NICs of the Open vSwitch.

Unlike other virtual switch technologies, Open vSwitch adopted the idea of reprogrammability through OpenFlow instead of being tied to tightly vertically integrated network control stack [22]. Because of these reasons, OVS is used not only in cloud computing frameworks but also in broadly used network emulator -Mininet: fastest way to test SDN scenarios- as we will explain later in this section.

2.1.4. Controllers and Northbound Applications

In the context of SDN, controllers which constitute the control plane are basically software, more specifically operating systems that are responsible for the networking logic. The logical centralization of the control in networking logic, in particular, brings some additional advantages. First, when compared to lower-level device-specific configurations, the tendency to change network policies through high-level languages and software components is simpler and less error-prone. Moreover, a control program can automatically react to counterfeit changes in the state of network components, thus keeping the top level policies untouched. Besides, the centralization of control with general knowledge of network state simplifies the development of more sophisticated and complex network functions, services, and applications.

As stated earlier, the control plane keeps the centralized view of the network topology does not mean it needs to be handled by only one physical controller. Physically distributed controller designs also exist for the deployment of SDN while the design force the controllers to be synchronized with each other and share the information of the link statuses or new added/removed switches/hosts in order to be logically centralized control plane. For example, Beacon [24] Floodlight [25] Maestro [26] and

RYU [27] are some of the most known centralized controllers. Their architecture is based on centralized but multi-threaded design. They are devised to be highly concurrent and process over 10 million flows per second like Beacon [24]. However, centralized controllers still suffer from posing a single point of failure and having scaling limitations. Distributed controllers, however, can scale up to meet the performance criteria and have the superior solution for resiliency in exchange for extra effort to keep controllers synchronized. Yeganeh, Tootoonchian and Ganjali [28], present the concerns on scalability for SDN controllers and the trade-off between these centralized and distributed controllers. In [29], Elasticon Distributed SDN controller presents the distributed controllers schema in which controllers share distributed data storage. The research also proposes the technique for switch migration from one controller to another, in case of a controller failure or a need for load balancing which results in more reliable and scalable architectural design for SDN.

Lastly, we will mention the northbound applications (i.e., management applications) to cover the background for imagined SDN schema in full. Since the controllers are the thought to be the operating systems, the user applications are needed to perform networking operations. To put in a different way, high-level "network brain" applications that actually implement the control logic give the instructions to the network operating system in which these instructions are translated into forwarding rules and installed on the underlying low-level hardware infrastructure. A broad range of real-life networking operations (e.g., traffic engineering, mobility and wireless, measurement and monitoring, security and dependability and data center networking) is in the scope of northbound applications. As in today's computer technologies, there can be countless networking applications that run on top of the network operating systems leading to expedite the innovation in networking.

Although SDN is relatively new in the networking domain, there are already a vast number of applications which have been developed over the past couple of years. We can give some examples for the applications that stand out the most as; Aster*x [30] and PolicyCop [31] for traffic engineering, LIME [32] for data center networking, Active Security [33] and FRESCO [34] in the security domain. These applications and many

others, conducted by the researchers and/or the companies, are definite indications that SDN can indeed help us to solve the inevitable challenges we face now and will face in the future in computer networking.

2.1.5. Mininet

When SDN paradigm first proposed by McKeown *et al.*, the need for prototyping to test, to debug, to validate this new networking method was a necessity. Actually, due to the ossification of the Internet, new networking propositions always need to be assessed with simulations, emulations or on real hardware before their deployment in real life. To prove the benefits of SDN and validate the benefits it promises for networking operations, Mininet [35] a network emulator created. Mininet is a software that uses lightweight virtualization mechanisms, such as processes and Linux network namespaces to create virtual hosts, links and switches (Open vSwitches specifically) to create, customize, interact with the network topology and share the constructed prototype quickly [36].

Mininet is created to reflect some characteristics that are needed to test SDN scenarios. Those characteristics can be listed as follows; flexibility, applicability, interactivity, scalability, realistic and reproducibility. New topologies and features can easily be configured using software languages and operating systems for example. This property makes Mininet flexible. The created scenarios in Mininet can be usable in real networks and hardware testbed without any changes in source codes and this makes Mininet applicable. Emulated networks in Mininet runs and can be managed in real time as if it happens in a real network to achieve interactivity. It also allows creating large networks that consist of hundreds or thousands of switches on one single computer [37]. It is also proven with many researches that it behaves quite realistically for many real-life networking scenarios with high-degree of confidence and without the need for changing the real-life protocol stacks. Mininet is also a great tool for the sake of reproducible and shareable experiments. The reproducible research results gain significance as indicated in [38] and widely adopted network emulation tool Mininet provides this important feature as well as granting the possibility for the experiments

to be shared, modified and improved by other researchers. However, Mininet also has some limitations. One of the limitation is that Mininet-based networks cannot exceed the CPU or bandwidth available on a single server restricting the scalability. Non-Linux-compatible OpenFlow switches or applications cannot be used in Mininet and this is another shortcoming of the Mininet.

As to the technicality of Mininet, it is fully written in Python except for a short C utility and also provides straightforward and extensible Python API for network creation and experimentation. Topology-aware CLI that runs network-wide is included to ease debugging. Besides, Mininet's CLI is OF-aware along with supporting the OVS which makes it preferable for SDN/OpenFlow experimentations. With the help of the Python API it provides, main components of the topologies (i.e., hosts, switches and links) are created as follows;

(i) addHost('name of the host', 'IP address of the host', 'MAC address of the host', 'CPU to be allocated for the host') method adds a new virtual host with the given parameters.
(ii) addSwitch('name of the switch', 'port number for listening to controller', 'OF protocol to be used') method adds a new OVS which is able to connect to an SDN controller to the topology
(iii) addLink('node 1', 'node 2', 'link options') method adds a new link between two nodes. Nodes can either be one host and one switch or two switches.

To customize the topology created, the API also subsumes plenty of other methods after accessing a component by its given name. These methods can be used to allocate number of cores allocated to hosts or alter the features of them, to enable/disable the protocols inside Open vSwitches such as Spanning-Tree protocol, to set bandwidth or delay for the links. There are also methods to change the generic ways of network behavior such as setting a remote controller. Besides, monitoring the status or connections of the components and testing the connectivities between them inside a created topology is available.

Even though Mininet offers all the convenient features as we stated above, it is a network emulator tool in the end. The network emulation is different than the network simulation. Network simulators are basically software that models the virtual networks with various aspects based on completely theoretical methods. As an example, in a network simulation, there is no actual network packet which is sent/received or is exposed to any delay. Instead, packet sending and receiving are modeled in a way, as if they actually take place and the packets experience a delay with regards to the certain parameters. Since everything happens within the software, the simulations don't need to simulate the networks in real-time [39]. The simulations can take a very short amount of time when simulating a network scenario which would normally take a much longer time or vice versa. Yet in the results of the simulations that are reported, all the metrics correspond to real-time values thanks to the theoretical models used for the simulations. However, this is not the case with network emulators. In network emulators such as Mininet, actual network packets are used on the link layer between virtual components and the emulation of the network occurs in real-time [40]. In other words, network performance metrics can be observed, tested and validated under various conditions so that the prototype behavior should represent real-time behavior with a high degree of confidence.

In this sense, it can be said that network emulators are more reliable than network simulators but the emulation also comes with its limitations especially when designing complex distributed network systems like in edge computing scenarios. The reason is, performed emulations are usually tightly coupled with the hardware they run on since the virtual components along with every packet that is sent need to be processed and this processing depends on the CPU power of the hardware [41]. Since the emulations are performed in real-time, the deviation in the processing time in such complex scenarios affects the reliability of the performed experiment [36]. As indicated in the work conducted by Handigol *et al.* [38], network emulators may fail to provide adequate performance isolation for the experiments performed on them especially in terms of the timing behavior of the system. Just like other emulators, Mininet also suffers from the same limitations. Thus, emulating server-client systems for edge computing scenarios on a single machine using Mininet would not give the most desired results. Therefore,

in this thesis, our motivation is; to make these shortcomings of Mininet, coming with the emulation of many components for a network, less severe with the help of real hardware dedicated to carrying out certain tasks in our hybrid environment design.

2.2. Edge Computing

In order to present substantive background and literature review on Edge Computing, first, we need to introduce and provide elaboration on Cloud Computing. It is because of the fact that Edge Computing is defined as a subbranch of Cloud Computing. In a way, it inherits the same logic of service provisioning and the key conceptual elements of Cloud Computing. Thus, in the following sections, first, we present a thorough background on Cloud Computing and second we elaborate the Edge Computing (and its derivatives such as Fog Computing, Mobile Edge Computing, etc.) in detail alongside answering the questions of why there is a need for convergence towards mentioned technologies.

2.2.1. Cloud Computing

Cloud Computing is a form of utility computing, a term that has started to be used nearly 50 years ago. It involves providing a broad array of computing-related services as public utilities much like water, gas, electricity or telecommunications. Back then, it was thought that it will profoundly transform the nature of companies' IT services, IT strategies and technological infrastructure [42]. Years later, after enough advancement in the speed of the Internet and other telecommunication technologies, Cloud Computing emerged as a reality that offers a number of benefits for the companies. Flexibility, cost reduction in capital-expenditure, disaster recovery and work-from-everywhere model are some of the examples of the benefits that Cloud Computing offers.

Although there are abundantly many definitions of Clouds stated by the researchers, these definitions blend into the following comprehensive definition in the work of Vaquero *et al.* in [43]; "Clouds are a large pool of easily usable and accessi-

ble virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs." As the definition above indicates, there are a number of key elements of Cloud Computing that need to be discussed. First, we will start with the infrastructures take place in the context of Cloud Computing, namely data centers - a large cluster of networked computer servers. After that, we will mention the actors of the cloud computing, benefits that it promises to deliver and why it is one of the current trends in ICT domain. Lastly, we will present the most widely accepted business models that are being used to enable cloud computing practices.

2.2.1.1. Infrastructure. As we mentioned earlier, the core infrastructure in the context of cloud computing is called data center. Data centers are a large cluster of highly capable inter-connected servers managed by the cloud providers who are the owner of these data centers. Although a typical data center consists of hundreds of thousands of computers [44], tenants (clients) use the same infrastructure to get provisioning. Therefore, providing computational resources (i.e., processing power, storage, networking) is achieved with virtualization of the infrastructure. In other words, the computing resources are allocated to the tenants in the form of VMs, instead of dedicating specific servers to each one of them. The virtualization of resources is carried out at a granular level in order to maximize the utilization of this infrastructure [45]. Since maximizing the utilization of the infrastructure in this multi-tenant system is critical to the providers, they try to perform server consolidation as much as possible through not only virtualizing the hardware underneath but also migrating the VMs to the physical hardware that is not consolidated enough. This operation also plays an important role in energy saving, since the concentration of the active VMs to close proximity of each other creates the opportunity to save energy on the idle hardware by putting them into sleep/hibernation mode [46]. Considering the fact that a data center can consume as much energy as 25,000 households [47], saving on energy via VM migration/server consolidation is crucial.

From the network point of view, data center networking infrastructure has some special characteristics too as described in [48]. The network is a hierarchy in which layer of servers in racks (typically 20 to 40 servers per rack) at the bottom are connected to a Top of Rack (ToR) switch. ToRs connect to aggregation switches and these switches aggregate further connecting to access routers that are placed on top of this hierarchy as can be seen in Figure 2.3. This kind of hierarchy is referred as the fat-tree topology. From the perspective of utilization and management, especially for VM migration inside data centers, there is plenty of research that suggest using SDN for the networking would offer many benefits [32, 49, 50].

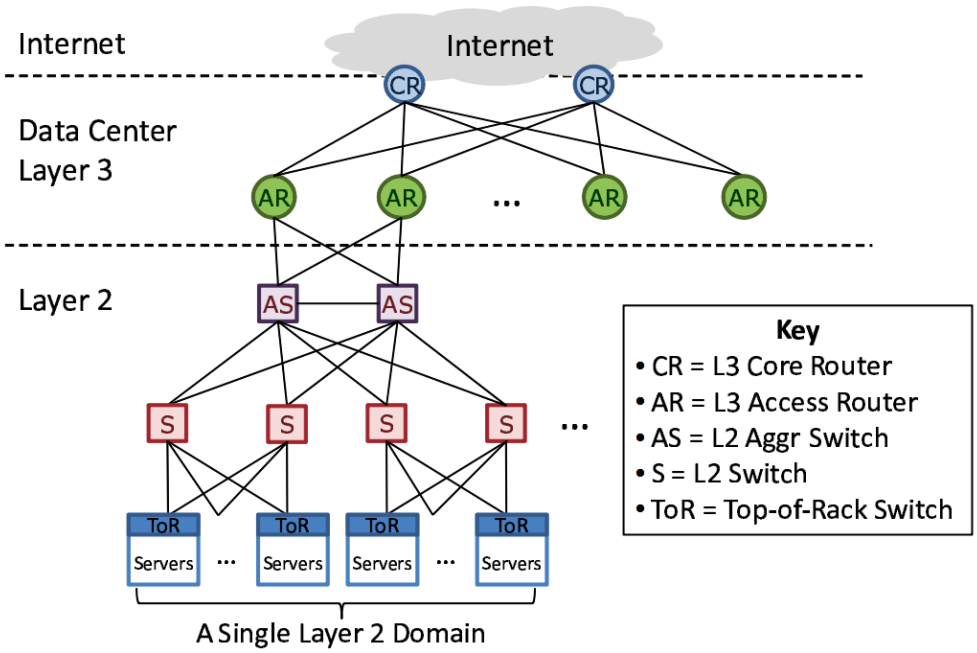


Figure 2.3. A conventional network architecture for data centers (adapted from figure by Cisco [48]).

2.2.1.2. The Actors Involved in Cloud Computing. As to the actors involved in cloud computing, there are clients who get the computing services and there are cloud providers who deploy date centers and supply the provisioning that those clients demand. The client profiles can vary in a broad range from a single person to large companies. As we will mention later, since the business model is pay-per-use, the clients can benefit from the computational resources (i.e., processing power, storage,

networking) as much as they need within a certain period of time instead of deploying their own expensive infrastructure (which also brings its own maintenance burden) and then they can scale up/down these resources they benefit from whenever they want. From the clients' perspective, this flexibility is invaluable when considering large batch-oriented tasks can get results as quickly as they scale since using 1000 servers for one hour costs no more than using one server for 1000 hours [51]. On the other hand, the utilization of the resources with high-level of granularity thanks to the advancement in virtualization technologies over the past years offers highly profitable business to the cloud providers.

2.2.1.3. Business Models. Business models for Cloud Computing are whole another topic which we mention briefly in this thesis. Cloud Computing offers many solutions to remotely access different kinds of computing resources ranging from bare metal services to well developed ready-to-use applications. These different business model approaches enable granularity in provided capability by offering different ways of utilization for the remote computing resources, as virtualization enables granularity for the same remote hardware resources. These business models offer solutions throughout all architectural layers of cloud computing from the top of the stack to the bottom.

The cloud providers can rent the infrastructure itself to the clients, which is known as Infrastructure as a Service (IaaS). This model involves delivering the hardware (server, network and storage) as well as operating systems virtualization technology as a service. IaaS providers do very little management other than keeping the hardware systems (data centers) operational. Keeping systems operational also includes scaling of bandwidth, memory and storage. These details lead providers to compete on the performance and pricing offered on their dynamic services [52]. This type of model is an evolution of traditional hosting which does not compel any long-term commitment and allows clients to provision resources they need on demand. The clients must deploy and manage their software services themselves as they would do in their own data centers. Amazon EC2 is an example of this kind of offering where clients can hire the infrastructure itself at large scales.

They can rent a platform to clients so that they can build their own systems meanwhile utilizing the pre-installed/pre-configured tools inside the platform, a.k.a. Platform as a Service (PaaS). This model includes IaaS by default plus a certain amount of software so that developers can utilize over the web. It provides a framework in which for example, there can be a set of integration tools or programming functionalities for numerous database systems and the applications that will be developed on top of these databases. PaaS, as indicated in [52], facilitates development and deployment of applications since it lifts the complexity and cost for the management of the underlying infrastructure, providing the necessary tools for the complete life cycle of building and delivering many kinds of software services. The provider is responsible for the administration of such development tools, middleware and the operating system. Sometimes, they even provide specific programming languages and API for the usage of the platform. As an example of PaaS, Google App Engine is a cloud solution that provides a platform which contains many programming languages including Java, Python, PHP and Node.js for rapid application development.

Last business model we will mention is Software as a Service (SaaS) and in this approach, the software is already deployed on remote servers and made ready for clients' usage. This kind of provisioning often referred as "on-demand software". The clients use the software that runs on a platform or infrastructure without any development or programming. Instead, they pay for the utilization of the software by having the ability to customize and configure the software according to their needs. The provider is responsible for the maintenance and the further development of the software as well as making it available to multiple tenants over the web by running the application on its own data center (or another cloud provider's PaaS or IaaS service offerings). For example, Saleforce.com is one of the well-known SaaS that provides customer relationship management (CRM) solutions to its clients, who in this case companies.

2.2.2. Edge Computing and Its Derivatives

Edge Computing is a general term that is used to describe the concept of pushing the computational resources to the edge of the network, near the source of the data. It

is usually referred as an extension or optimization methodology for Cloud Computing systems. With the increase in data generated by edge devices (i.e., mobile devices, IoT devices), the necessity to process these data with the speed of their generation is also increased. Sending the data to centralized data centers with these high generation frequency over WAN would not be the optimal solution since these edge devices usually operate with the concern of energy efficiency and require small service delay QoS. The big data that are generated by these edge devices has to be processed in their physical vicinity for making use of the services provided in a more optimal way. Although Cloud Computing brought countless benefits to our lives, we need an extension to it, as it does not solve the challenges in processing zeta-bytes of data that are produced by billions of end-devices. The Edge Computing is the name of the paradigm that has emerged to solve these challenges.

Recently, there have been many operational and architectural design proposals for the implementation of Edge Computing systems. The terms Mobile Cloud Computing (MCC), Mobile-Edge Computing (MEC), Cloudlet, Fog Computing are examples of these implementations in order to put Edge Computing into practice. Even though they all are enablers for Edge Computing systems and they focus on delay sensitiveness, scalability, energy efficiency and regulating core network traffic, they target different uses cases for Edge Computing.

Mobile-Cloud Computing is described as a new paradigm for mobile applications that the data processing and storage take place outside of the mobile device (this operation is usually referred as offloading). It provides these services in remote and more powerful servers and this type of provision mitigates the requirement of powerful configurations (e.g., CPU speed and memory capacity) for mobile devices. Today's mobile applications such as augmented reality applications or health-monitoring applications necessitate real-time interactive request/response delays even though the demand for computational resources are very high for these kind applications [53]. Early researches on the infrastructure of MCC focus on utilizing Cloud Computing for the blatant computing resource need, however, the "cloud" in this context requires to be closer to the mobile-device users and to have more distributed structure. Recently, the term

"cloudlet" has been coined by Satyanarayanan *et al.* in 2009 [54]. The cloudlets are described as powerful computer/cluster of computers that are well-connected to the Internet, physically close to the mobile users and available for the use of nearby mobile devices. The proximity of the cloudlets often considered as a couple hops away from mobile devices in order to satisfy the intended service delay QoS [55]. Fast network connection over Local Area Network (LAN) makes cloudlets as an extension of classic Cloud Computing. Briefly, cloudlet based MCC and Cloud Computing, both offer multi-tenancy, easy and on-demand access, scalability and processing power. On top of these, MCC also offers real-time service response, energy conservation and mobility management for the edge devices.

Fog Computing term, on the other hand, is mostly characterized as being related to the Internet of Things (IoT). Unlike MCC - which can be thought and designed as another level in the cloud hierarchy, Fog Computing is the paradigm that provides the needed computational services for a number of critical IoT services and applications such as smart grid, smart vehicles, smart homes, crowdsourcing [56]. Such services and applications exhibit some common characteristics; (i) delay-sensitiveness, (ii) generating very large network traffic, (iii) energy-efficiency. In the envisaged deployment of IoT services, solutions to these characteristics often have to be addressed together in order to satisfy the requirements for the management and analysis of generated big data. Fog servers, similar to cloudlets, are the key infrastructural components of this paradigm to enable such provisioning for the increasing number of IoT devices. Fog servers are considered as more intent-driven than general purpose cloudlets. However, they both have the common set of objectives: minimizing network latency and maximizing real-time interaction.

The idea behind Mobile-Edge Computing is to have cloud computing capabilities at the edge of a cellular network. Since there are already deployed base station infrastructures which belong telecommunication companies, the motive is to add cloudlet-like capable servers into the topology of Radio Access Network (RAN) in the vicinity of mobile subscribers. Like other Edge Computing approaches, MEC also tries to achieve accelerated service speeds, increased responsiveness at the edge of the network.

However, the main motivation behind this approach is to solve the challenges for the increased mobile data traffic due to the relatively recent rise of the smartphones. In the desired MEC architecture, MEC servers are located within or near the base stations by mobile operators in order to provide increased service quality and effective mobility management to the mobile network users over the RAN [57]. As in all other types of Edge Computing approaches, in case of insufficiency of computing resources or storage, these MEC servers also have the capability to offload their tasks to data centers [58].

Lastly, in the envisaged fifth generation (5G) mobile networks, these Edge Computing scenarios are often tightly coupled with Software-Defined Networking since the next generation of mobile networks needs to be designed to meet the strict requirements such as higher bandwidths and link utilizations, lower latencies and service delays and increased mobility for mobile/end devices. From this aspect, the management complexity of billions of end devices is hard to deal with for our current legacy networking structure. SDN offers many benefits to overcome this management complexity and thus, we used both of these technologies together to carry out this research.

3. ROAD TO HYBRID TESTBED ENVIRONMENT CREATION USING SDN

In this chapter, as the first part of our work, we elaborate on the hardware setup/ configuration, the SDN controller we used and the northbound applications that runs on top of it as well as the topologies created using Mininet and other components that we used in the process of hybridization. Furthermore, we give the detailed explanation of our environments, the step stones that led us to gradually hybridize the whole system. In this research, we used a comparatively small topology, since our principal motivation is to establish a hybrid testbed installation in which the percentage of hybridization should be above a certain level and to show the advantages of using this real hardware in the domain of SDN and edge computing studies.

3.1. Experimental Setup

Because of the necessity of high computer resources and real hardware to deploy the desired hybrid environment for our research, giving the specifications of these hardware serves to the purpose. Hence, in this section, we describe the experimental setup for the hybrid testbed we created using three different computers and three Raspberry Pi's. The first computer is the main desktop computer on which RYU controller and Mininet components are running. The hardware specifications of this machine are as follows; Intel(R) Core(R) 8 core CPU920@2.67GHZ, 24 GB RAM, 280 GB HD. This desktop computer equipped with two Network Interface Cards (NIC): one of them is Realtek RTL8111 PCI Express Gigabit Ethernet card integrated on the motherboard and the other NIC is quad-port National Semiconductor Corp. DP83065[Saturn] Gigabit Ethernet card. The second computer which is used as a cloudlet has Intel(R) Core(R) i7 4 core CPUQ8300@2.50GHZ, 4 GB RAM, 125 GB HD and this computer is equipped with only one NIC: built-in Intel Corp. 82567V-2 Gigabit Ethernet card. Lastly, we used one laptop on which Open vSwitch is built and real Ethernet cards are integrated into this Open vSwitch. As to the specifications of this laptop, it has

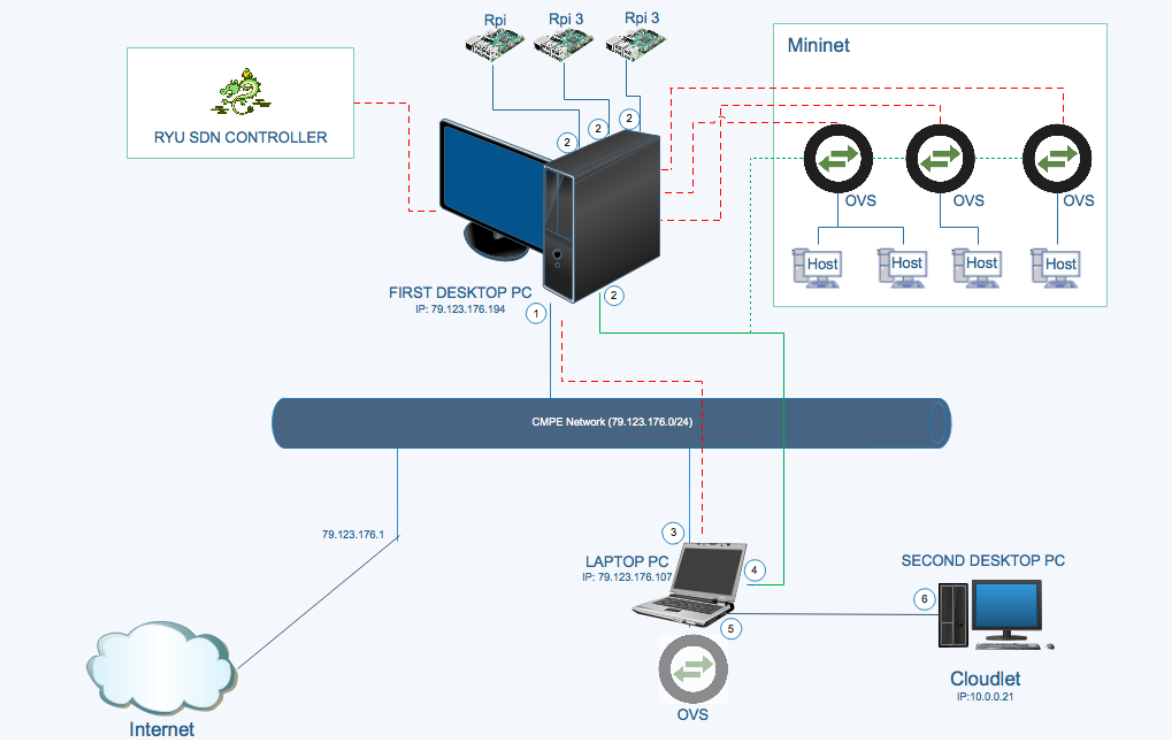


Figure 3.1. Final Form of the Hardware Setup (Hybrid-3)

Intel(R) Core(R) i7 4 core CPUM620@2.67GHZ, 4 GB RAM, 125 GB HD. This laptop has only one integrated Intel Corp. 82577LM Gigabit Ethernet card, however, we connected two additional USB-to-Ethernet devices so that we can use it as a switch in our environments. One of the USB-to-Ethernet NIC is Realtek USB 10/100 LAN and the other one is Kontron DM9601 Fast Ethernet Adapter. As we will elaborate in detail later, both of these USB-to-Ethernet devices can carry 100Mbps. The first USB-to-Ethernet device is used for the connection between the Open vSwitch inside the laptop and other OVS's inside Mininet on the main PC. The second one is used for the control link between Open vSwitch on the laptop and the RYU controller on the main PC. The integrated NIC is used to connect the Open vSwitch on the laptop and the cloudlet, which is the second desktop PC.

Finally, we used one Raspberry Pi and two Raspberry Pi 3 as real hosts that get provisioning from the cloudlets in our environments. In Figure 3.1, detailed visualization for the final hardware setup (Hybrid-3 in Figure 3.2) and the Ethernet cards with

their connections can be seen. Apart from these, for all computers and all Raspberry Pi's, we used Ubuntu 14.04 LTS Desktop and Raspbian Jessie as the operating systems respectively.

On the main desktop computer, we installed the RYU SDN Controller on the port 6634 which is the default port for the RYU controller. We also used the same computer for the deployment of the Mininet components (i.e., Open vSwitches, virtual hosts and virtual links). The controller is able to communicate with the Open vSwitches either on this computer created by Mininet or the other Open vSwitch that is installed on the laptop after the IP address of the controller and the port 6634 specified on the OVS's. Although the SDN controller is on the same computer with Mininet components, theoretically it is considered as a remote controller since Mininet creates different network namespaces for the Mininet hosts [37]. However, it is possible to deploy the SDN controller to a complete remote computer and connect the OF enabled switches to it.

In Figure 3.1, the red dashed-lines represent the control links between the RYU controller and the OVS's. These links are used to send PACKET_OUT from controller to Open vSwitches and PACKET_IN messages from Open vSwitches to the controller whenever they are needed. The green dashed-lines represent forwarding links between the switches when the hosts that are connected to the switches send packets to each other, these links are used. The green solid line represents a real cable connection between the network interface on first desktop PC which is added to one of the OVS created by Mininet and the network interface on the laptop that is added to the OVS running in the laptop. The blue solid line between the laptop and the second desktop PC represent another real Ethernet cable between the OVS in the laptop and the cloudlet inside the second desktop PC. Likewise, the blue solid lines between the first desktop PC and Raspberry Pi's are real Ethernet cables between the Raspberry Pi hosts and the OVS's in the first desktop PC. In Table 3.1, the matchings between the numbered interfaces in Figure 3.1 and all the physical network interfaces we mentioned earlier can be found.

Table 3.1. Ethernet Cards Used

Port Number in Figure 3.1	**Ethernet Card**
1	Realtek RTL8111 PCI Express
2	National Semiconductor Corp. DP83065[Saturn]
3	Kontron DM9601 USB-to-Ethernet
4	Realtek USB 10/100 LAN USB-to-Ethernet
5	Integrated Intel Corp. 82577LM
6	Integrated Intel Corp. 82567V-2

A little notice here is, we used red dashed lines to represent the control link between Interface 1 and Interface 3. Although it is not a direct link, the OVS in the laptop uses the CMPE LAN network to find routing and to connect to the RYU controller with its IP address.

3.2. Environments

In our research, we run our simulations for eight different environments, the summary of the environment representations can be seen in Figure 3.2. Four environments, for which we will use Env-1, Env-2, Env-3 and Env-4 from now on, have one cloudlet. The other four environments, namely Env-5, Env-6, Env-7 and Env-8 have two cloudlets. The environments that include two cloudlets have the same topology with the environments that include one cloudlet except for an additional virtual cloudlet inside Mininet.

The real hardware components we described in Section 3.1 are used in these environments with regards to the hybridization levels of them. For Pure Mininet environments every component is virtually created inside Mininet. For Hybrid-1 environments we used the second desktop PC as a real cloudlet instead of the virtual one we created inside Mininet. We took one OVS out of Mininet and deployed on a standalone laptop PC in Hybrid-2 environments. Lastly, we replaced three virtual Mininet hosts with

Raspberry Pi's in Hybrid-3 environments. In this section, we will give the detailed explanation for these environments with different hybridization levels.

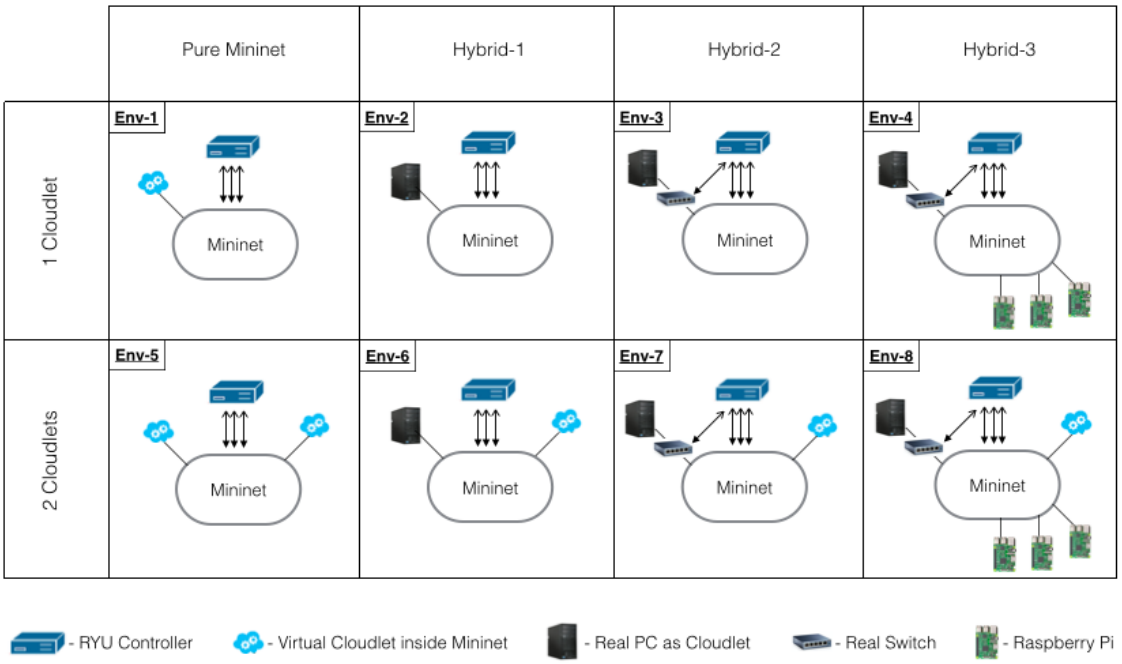


Figure 3.2. Summary of the Environment Representations

3.2.1. Pure Mininet Environments

In Pure Mininet Environments (Env-1 and Env-5), every component (cloudlets, hosts, switches and links) is virtual. The cloudlets are also Mininet hosts that provide provisioning for the other Mininet hosts, the hosts that are imagined as end-users in our edge computing scenarios. The difference between the cloudlets and the other hosts is that the cloudlets run server code whereas other hosts run client code to get the provisioning. This scenario represents the experimentations that can be done using only Mininet emulation tool for edge computing scenarios which we will use to compare the results between this environment and the hybridized environments.

Our topology creator script is implemented in a way that all the hosts are on the subnet with CIDR 10.0.0.0/24 where the IP of Host-n is 10.0.0.n. The cloudlet's IP is set to 10.0.0.21 (in Env-5, the second cloudlet has IP 10.0.0.22) which we later use these

IPs for the provisioning to other hosts. The MAC addresses for the cloudlets are also assigned so that we can gather the cloudlet link load stats inside the "Traffic Monitor" northbound application. When we setup the topology, we also trigger the server code on the cloudlets created thanks to the Python API of Mininet. The server code is being ready and starts responding the client requests as soon as the environments deployed and the switches connected to RYU controller. In the same script, we also trigger the client code on all of the Mininet hosts so that they can start sending requests to the cloudlet or cloudlets depending on the environment used.

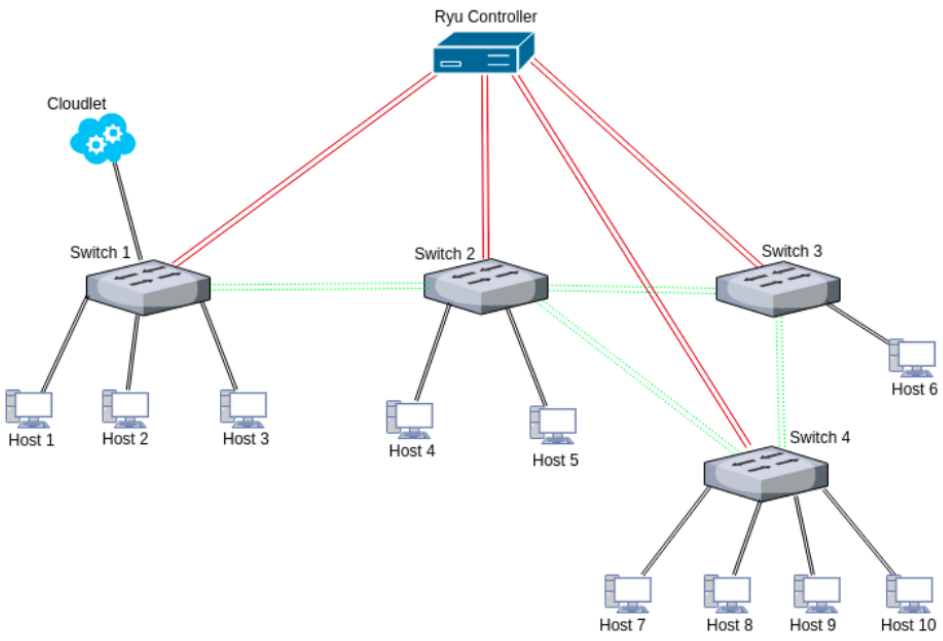


Figure 3.3. Pure Mininet with one Cloudlet Network Topology (Env-1)

We set the link capacities to 100Mbps for the links between the switches inside Mininet since for the other hybridized environments we used 100Mbps Cat5 real Ethernet cables in order to create environments with the same settings/configurations and to highlight the difference between the real components versus virtual components only. The network topology for the Pure Mininet Environments can be seen in Figure 3.3. For Env-5, the additional virtual cloudlet is attached to Switch 3 while preserving the topology of the network.

3.2.2. Hybrid-1 Environments

Hybrid-1 environments have one real cloudlet as a different computer which is the second desktop PC in Figure 3.1 instead of one of the virtual cloudlets. This PC is connected with a real 100Mbps Cat5 Ethernet cable to the main desktop PC through one of its NICs. We set "Auto" option for its network interface because the real components cannot be added to the Mininet topology if the IP address of the host is assigned statically or by DHCP. Later on, we explicitly set its IP to 10.0.0.21, the same IP we used for the virtual cloudlet, this time however, the virtual cloudlet is not in the topology and this real cloudlet replaces it. For the same interface on this real cloudlet PC, we set gateway, netmask and broadcast as 0.0.0.0, 255.255.255.0 and 10.0.0.255 respectively so that the host can become discoverable by other hosts of the switch it is connected to and by the controller. The topology creator script again loads the client application on all Mininet hosts, server application however, is triggered manually on the real cloudlet PC.

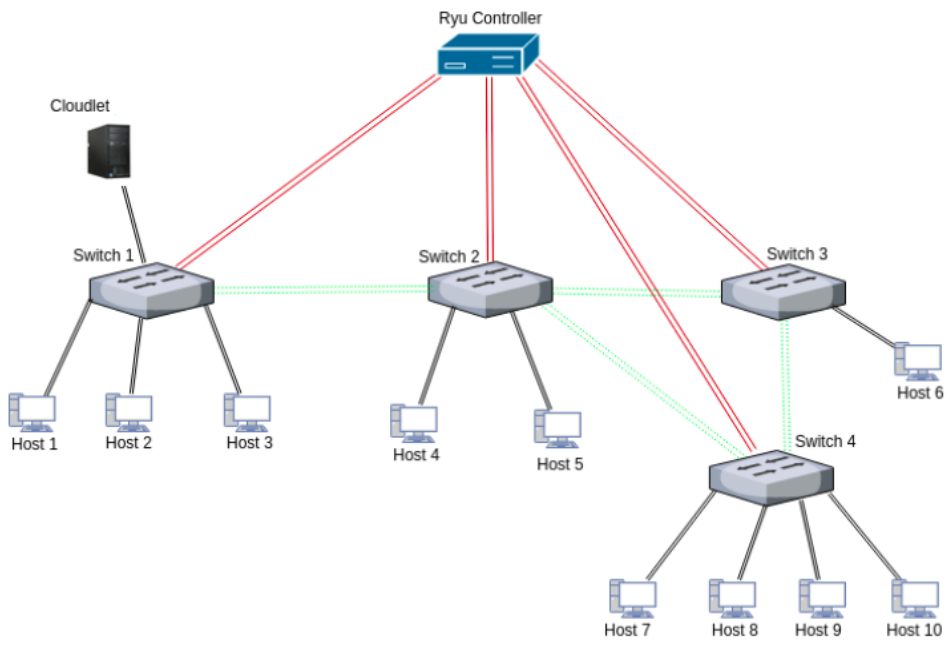


Figure 3.4. Mininet with one Real Cloudlet Network Topology (Env-2)

For the connection of the real cloudlet to the OVS created inside Mininet, inside the "Topology Creator" script which runs on the main desktop, all available network interfaces on the main desktop are parsed and the interface connected to the second desktop PC (one of the quad-port NIC numbered 2 in Table 3.1) is included to the Switch 1 programmatically. This methodology allows the real cloudlet to be included homogeneously into the virtual environment created by Mininet. The Env-2 topology can be seen in Figure 3.4 and again for the Env-6 other virtual cloudlet is created and connected to Switch 3 inside Mininet just like in Pure Mininet environments.

3.2.3. Hybrid-2 Environments

In these environments, we took out the Open vSwitches that connect to the real cloudlet PC from inside of Mininet and replaced it with OVS inside the laptop PC in Figure 3.1 as one more step to hybridization. The configurations of the real cloudlet PC stayed the same in these environments as in Hybrid-1 environments except it is connected to the laptop that comprises an OVS as a standalone switch, instead of the main desktop computer.

Here we present, the details of how to configure the Open vSwitch installed on the laptop in order to include this switch and the real cloudlet into the Mininet topology. The configurations are done via the command line of Linux with "ovs-vsctl" command. The "ovs-vsctl" command has become a native command inside Linux to configure the Open vSwitch after the OVS is included inside the Linux kernel. The respective order of the commands we run to have the desired OVS as follows;

(i) sudo ovs-vsctl add-br s1
(ii) sudo ovs-vsctl add-port s1 eth0
(iii) sudo ovs-vsctl add-port s1 eth1
(iv) sudo ovs-vsctl add bridge s1 other_config datapath-id:'0000000000000001'
(v) sudo ovs-vsctl set bridge s1 fail_mode=secure
(vi) sudo ovs-vsctl add bridge s1 other_config disable-in-band='true'
(vii) sudo ovs-vsctl set-controller s1 tcp:79.123.176.194:6653 ptcp:6634

The command (i) creates a new bridge named s1, (ii) and (iii) commands add the interfaces (eth1 and eth2), which come to life after connecting USB-to-Ethernet devices to the laptop, as ports into the created s1 bridge. The configurations of these interfaces are left as default. The command (iv) updates the datapath ID of the bridge created (a random datapath ID is assigned when first created) in order for us to distinguish the PACKET_IN messages from this switch inside the "Topology Discoverer" and "Traffic Monitor" applications. The command (v) executed so that the switch does not install new flow rules by itself in case of a controller failure because the OVS is by default behaves as an ordinary MAC-learning switch. The in-band control in OVS domain means using the same ports and network to forward data and to communicate with the controller. Although this method does not require a dedicated port or network, out-band control method has more benefits. Out-band control simplifies the traffic management, it is more reliable in case of excessive switch traffic that might interfere with the control traffic and it eliminates the possibility of impersonation by other machines that are not in the control network. The command (vi) disables this in-band control for this bridge and this is the reason we used USB-to-Ethernet device on the laptop to connect the RYU controller through CMPE network as our control network for this standalone switch. Lastly, (vii) command sets up the connection between the RYU controller which is at 79.123.176.194:6653 for active listening and ptcp:6634 for passive listening.

After the execution of the commands above for configurations, we also enable the IP forwarding for the interfaces inside the laptop. Since the interfaces are now included to the created OVS bridge, Linux does not forward the packets between these interfaces and IP forwarding needs to be configured manually. Otherwise, the packets that come from one OVS switch inside Mininet to the switch inside the laptop will not be sent to the real cloudlet PC as we intend.

The physical cable connections for Hybrid-2 environments are as follows. First, the cloudlet PC is connected to integrated NIC of the laptop with 100Mbps Cat5 Ethernet cable. Second, a Cat6 Gigabit Ethernet cable is used between the laptop and the main desktop PC to connect the OVS's inside them. One end of the cable

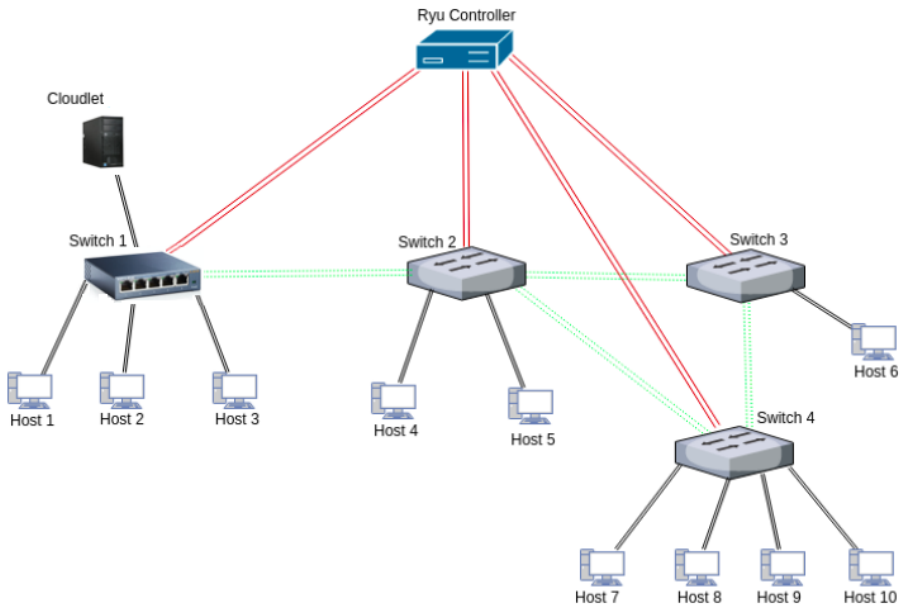


Figure 3.5. Mininet with one Real Cloudlet and one Real Switch Network Topology (Env-3)

is attached to the NIC with number 4 in Table 3.1 and the other end of the cable is attached to one port of the quad-port NIC with number 2 in Table 3.1 on the main desktop. Despite the fact that we used a Gigabit Ethernet Cable here, USB-to-Ethernet device has the capacity of 100Mbps which is the same as the switch link capacities we set inside Mininet. As mentioned above, for the control link between the OVS inside the laptop and RYU SDN controller, we attached another USB-to-Ethernet device on the laptop (the NIC with number 3 in Table 3.1) to CMPE network with another real Ethernet cable. Figure 3.5 shows the network topology for these environments. The second virtual cloudlet for Env-7 is again created and connected to Switch 3 inside Mininet just like in Pure Mininet and Hybrid-1 environments

3.2.4. Hybrid-3 Environments

The last hybridization step we took in these Hybrid-3 environments is that we replaced three of the Mininet hosts in Hybrid-2 environments with Raspberry Pi's. In

order to include these Raspberry Pi's into the Mininet topology, we configured their network interfaces. We set "Auto" option for the network interfaces on all of them. We assigned the IP's as 10.0.0.5, 10.0.0.8 and 10.0.0.9 for our three Raspberry Pi's explicitly and for all of them, we set gateway, netmask and broadcast as 0.0.0.0, 255.255.255.0 and 10.0.0.255 respectively so that these Raspberry Pi's can become discoverable by other hosts of the switch it is connected to and by the RYU controller.

The automation of the simulations take place on these environments is managed by running the client code on the Raspberry Pi's with ssh command inside "Topology Creator" script. The reason is the client code needs to be triggered for every iteration of the simulation. A machine is required inside of the namespace of Mininet network (with CDIR 10.0.0.0/24) to be able to communicate with other Mininet hosts. To pass the client codes on the Raspberry Pi's when environments deployed, we used Host-1 Mininet host that is in the same network with other Mininet hosts along with the Raspberry Pi's. This host is able to run ssh command remotely to trigger the client application on Raspberry Pi's when an iteration of a simulation starts. For this operation we put the client code on Raspberry Pi's and enabled the ssh service on them. Host-1 Mininet host is able to trigger the client applications by using the IP addresses of the Raspberry Pi's.

The connections of the Raspberry Pi's to the Mininet is achieved with three 100Mbps Cat5 Ethernet Cables. One end of the cables are attached to the Raspberry Pi's Ethernet cards, whereas the other end of the cables are attached to the three ports of the quad-port NIC with number 2 on the main desktop PC in Figure 3.1 (the last remaining port of this quad-port NIC is already occupied as we mentioned in Hybrid-2 Environments section). Then, the corresponding interfaces of this quad-port NIC are included into Mininet topology. The interface that is linked to the Raspberry Pi 1 with the IP 10.0.0.5 is included to Switch-2, other two interfaces linked to Raspberry Pi 3's with the IP's 10.0.0.8 and 10.0.0.9 are included to Switch-4. Note that we exclude the corresponding virtual hosts from corresponding switches in Mininet. Thus we can say that Host-5, Host-8 and Host-9 turned into real hosts (as Rpi's). The topology for theses environments are presented in Figure 3.6. The second virtual cloudlet is

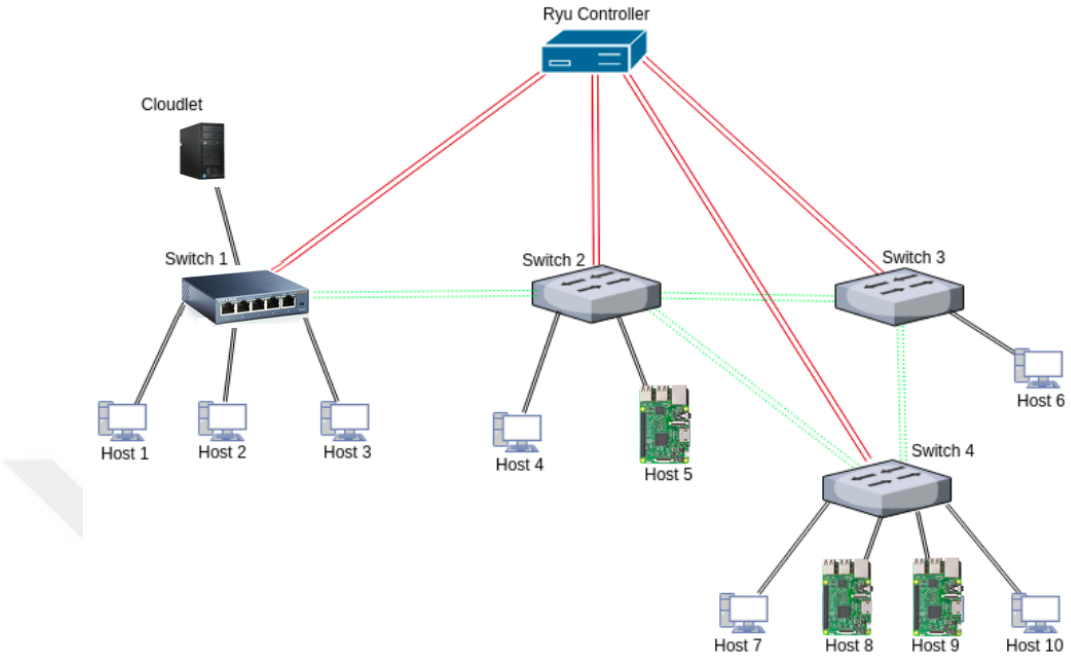


Figure 3.6. Mininet with Real Cloudlet, one Real Switch, three Raspberry Pi's Network Topology (Env-4)

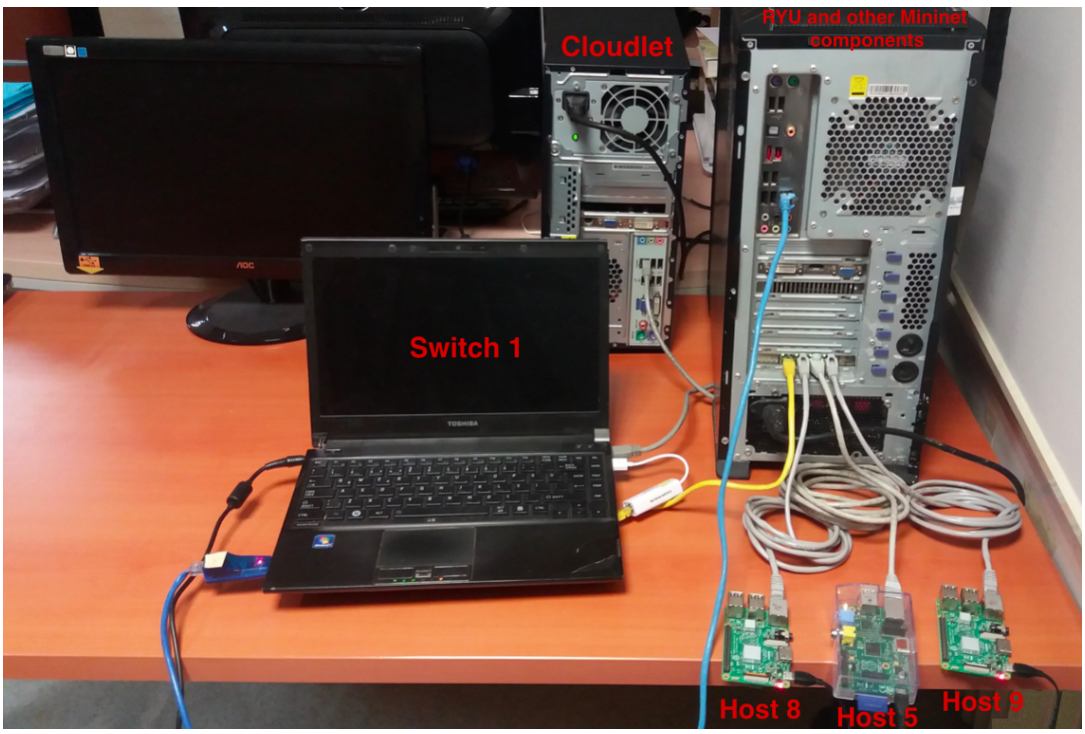


Figure 3.7. Real-life correspondence of Figure 3.6

attached to Switch-3 for the environment Env-8. Since Hybrid-3 is the most hybridized version of our environments, we wanted to present the real-life correspondence of this environment. The final form of our experimental setup and hybrid testbed can be seen in Figure 3.7.

3.3. RYU SDN Controller

As indicated earlier, in this work we used the SDN paradigm for networking because of the benefits that it brings to the table. For the SDN controller, we used RYU SDN Controller version 4.10, which is fully written in Python and it supports numerous protocols for managing network devices, such as OpenFlow, Netconf, OF-config. RYU controller supports OpenFlow 1.0, 1.2, 1.3, 1.4, 1.5 versions and for the southbound interface between the Open vSwitches and RYU Controller, we used OpenFlow 1.3 in this thesis. The versions of OF needs to be the same on both sides, the OF enabled switches and the controller since the features used in the protocol differentiates from one version to another. The OF version that is used for RYU controller can be handled within the northbound applications. To use the desired OF version for the switches, the forwarding devices have to support it. As to Open vSwitches we used, they also support OpenFlow 1.0, 1.2, 1.3, 1.4, 1.5 versions as they are virtual switches merged into the Linux kernel. Nevertheless, the OF version that will be used has to be set explicitly for the OVS's.

There is a wide range of SDN controllers on the market such as Java based Floodlight and the OpenDaylight are the most used controllers. Python based RYU, POX and Onix and C++ based NOX are some other most known and widely used SDN controllers. All of these SDN controllers have different features with some advantages and disadvantages over each other. The reason why we used RYU controller is because it is open source, highly modular and easy to deploy on Linux operating systems as we have used Linux on almost all of the environment components. The RYU source code can easily be modified in order to accomplish certain tasks as it is open source. Modularity is especially important since after some modifications or adjustments made on some of the features in the source code of the controller or after implementing/integrating the

northbound applications for RYU, it is fairly straightforward to take everything from one machine and deploy on another machine, then run the the same controller with the same northbound applications.

RYU controller highly utilizes the Python decorators. A python decorator is a specific change to the Python syntax that allows us to conveniently alter the functionality of the methods dynamically. With the help of the Python decorators, RYU controller runs multi-thread and processes the incoming OpenFlow event packets in a parametrized way. According to the used parameters, the controller catches the OpenFlow packets, which can be configuration messages sent to switches, the status or stat request/reply messages for the port of the switches or regular PACKET_IN messages from the OF enabled switches.

The controller also has the mechanism to collect and register the MAC addresses of the hosts which are connected to the ports of the OF enabled switches. The controller then sends down the related flow rules to all of the switches. This operation is called flow insertion and after the insertion of a flow rule, it prevents sending the same flow rule if it is already sent to a switch by matching the "in_port" (packet's incoming port) and "dst_port" (packet's destination port) of a PACKET_IN message.

3.4. Northbound Applications

Software-Defined Networking without the applications running on top of the controller would not be considered as a whole networking as the functionalities needed are implemented via these applications in SDN. The SDN controller handles the flow rules to be added to the switches according to some rules. However, those rules have to be dictated by some applications that run the algorithm itself that is needed. Since the controllers thought to be the operating systems in the SDN paradigm, it needs applications namely, northbound applications or management applications to bring functionality required for the networking. These applications that take care of the management of the network still exist on the traditional networks, yet they are either run on the vertically integrated routers or they exist as middle-boxes which are dis-

tributed all over the network. Routing protocols, shortest path algorithms, firewalls, load balancers are some of the examples of these applications.

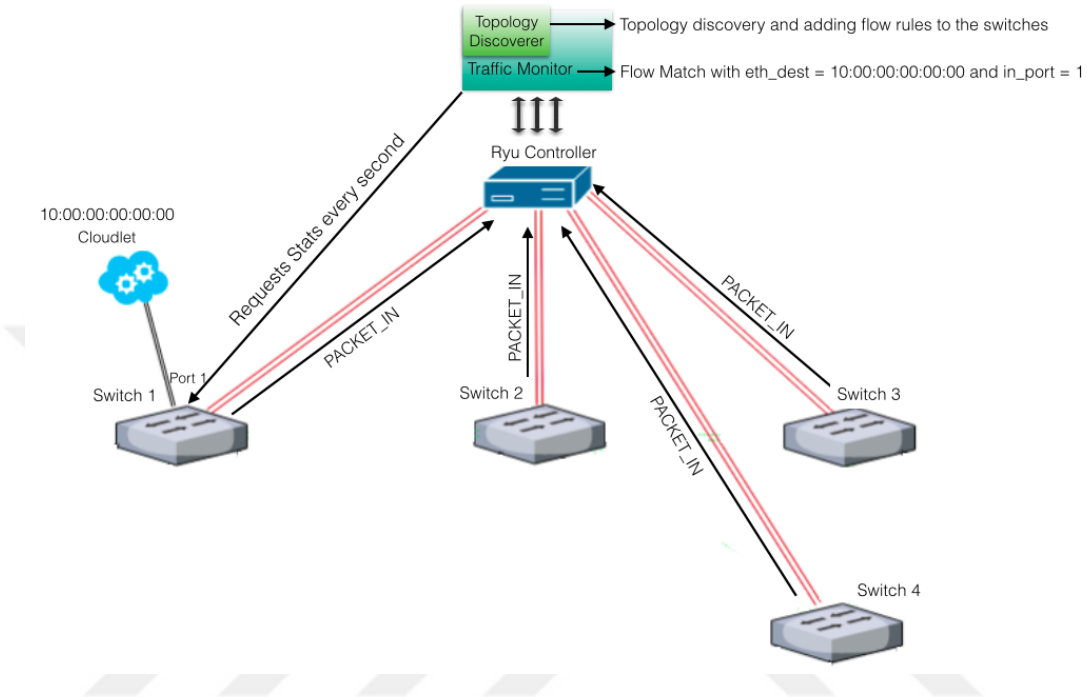


Figure 3.8. The Northbound in our SDN Environment

In this work, we deployed our data plane with the OVS's and we connected these data plane elements to the installed RYU controller. Afterward, we implemented two northbound applications which use the API (northbound interface) that RYU offers. We named our first northbound application "Topology Discoverer". This application collects the port locations of the MAC addresses for the hosts with ARP messages, then with the help of RYU API, it dispatches the necessary flow rules to the switches with OpenFlow. The second application "Traffic Monitor", runs in tandem with "Topology Discoverer" to extract the statistics from the ports of the OVS's. This application sends a request to all ports that are registered by "Topology Discoverer", then gathers the number of sent/received bytes for every port every second. The general look of the northbound in this work can be seen in Figure 3.8. We will discuss the technical details for both of these applications in the following sections.

3.4.1. Topology Discoverer

When our Topology Discoverer application starts running on top of the controller, it aims to catch two kinds of events with the python decorators which are basically, in our case, OpenFlow event listeners that RYU API provides. The first one is to catch any change in the switch features and the second one is to catch the PACKET_IN events from the switches using the OpenFlow event identifier and the dispatchers (dispatcher types in RYU can be seen in Table 3.2). The identifier and the dispatcher are given as parameters to the python decorators in order to determine the specific type of the messages when they arrive at the controller so that it takes an action according to the message types.

For the first event listener, parameters used in the decorators are "EventOFPSwitchFeatures" and CONFIG_DISPATCHER as the event identifier and the dispatcher type respectively. The version negotiation that takes place between the controller and the switches when the switches are connected to the controller for the first time is also also being performed with this event listener. Besides, whenever a new switch is added to the topology it sends a message to the controller to inform whatever the configurations it has such as the protocols enabled on the switch or the datapath-id (basically, the id of the switch) of it, if it is specified (otherwise, controller assigns a unique datapath-id to each switch to remember them).

The second OpenFlow event listener used in this application catches the actual PACKET_IN messages from the switches that contain the flows necessary to forward the data using "EventOFPPacketIn" event identifier and MAIN_DISPATCHER. After the network setup that is performed via the "Topology Creator" application, The ARP messages are broadcasted throughout the network in order to identify the hosts (specifically, MAC addresses of the hosts). This application creates a mapping between the port numbers of the switches and the MAC addresses they are connected in a way that the pair [datapath-id, port-number] maps to a specific [MAC Address]. With this method, the controller learns MAC-to-port mappings and avoids flooding the next time. Learning this mapping is also important for the installation of the flow rules to the

switches. We used the same flow adding mechanism as indicated in one of the example northbound functionalities provided by RYU itself [59]. Whenever a PACKET_IN message arrives, incoming port, the source MAC address and the destination MAC address are being parsed, then the MAC-to-port mapping is checked to determine which one of the switches has the corresponding destination MAC address attached to any of its ports. When the matching is found, the flow rule is being sent to the switch with a PACKET_OUT messages in which outgoing port is the matched-port for this switch. With this mechanism, if another packet arrives at the switch from the same incoming port, with the same source and destination MAC addresses, the switch knows which port it should use to forward the packet without asking the controller.

Table 3.2. RYU Controller Dispatchers [59]

Dispatcher Name	**Description**
HANDSHAKE_DISPATCHER	Sending/waiting hello messages
CONFIG_DISPATCHER	Version negotiation and features-requests sending
MAIN_DISPATCHER	Packet-in/out and Stats message receiving/sending
DEAD_DISPATCHER	Disconnecting from the peer

3.4.2. Traffic Monitor

This application requires the Topology Discoverer application to have the centralized view of the topology. Using the registered MAC-to-port mappings by the discoverer application, it aims to monitor the link loads of the links. The same python decorators are used to listen to the OpenFlow events. In this application the parameters for the decorators are stated as "EventOFPFlowStatsReply" for the event identifier and MAIN_DISPATCHER for the dispatcher. These two parameters are used to send stat requests and catch the stat replies from the Open vSwitches with respect to the defined monitoring time interval which we set as one second. Thus, every second the controller sends stat request messages to the switches and they send the stats of every single port through those ports back to the controller. In the controller, we match the

incoming port number of the requests with the port number that cloudlet is connected to its switch along with the MAC address of the cloudlet to identify the request that we are interested to log the byte count passes through that port. As can be seen in Figure 3.8, the Cloudlet has the MAC address 10:00:00:00:00:00 and it is connected to its switch through Port-1. In this application, we filtered the stat requests that are matched to these values in order to monitor the link loads for the cloudlets.

3.5. Automated Topology Creation for the Environments

The automation of the topology creation for all experiments that will be run is achieved with "Topology Creator" python script. The features of the hosts, the OVS's and the links between them inside Mininet can be dynamically customized within this Topology Creator. The customizations we did as follows, (i) for the hosts: we set IP addresses and MAC addresses for each one of them, (ii) for the switches: we set the port number to listen the controller and the OF protocol to be used and (iii) for the links: we set the bandwidth. More importantly, we managed to add real interfaces on the main desktop PC into the Mininet topologies in this script in a parametrized way so that it configures the physical interfaces that will be used for different hybrid environments created. This method allows us to include physical NIC's on the PC into the Mininet topology so that a physical NIC can behave as a port of an Open vSwitch inside Mininet.

This script also constitutes a foundation for our automated simulations that enables to run our scenarios with different parameters. The automation is managed by passing the desired parameters iteratively to this script. Later on, this script passes the incoming parameters to the Mininet hosts created inside this script. As we later elaborate our scenarios in which the clients send requests to the servers, the code distribution (i.e., server code, client codes) to the desired locations (i.e., cloudlets, hosts) is managed inside this script by using the Mininet Python API. This API enables to run desired commands, in our case triggering the server-client applications, on the hosts created inside Mininet.

As we said earlier, the maintenance and the deployment challenges of hybrid testbeds are harder to tackle compared to Mininet testbed. For the sake of the precision of experimentations, all real hardware needs to be in the correct state before every instance of simulations. Ensuring the availabilities and the correct states of every real hardware in the system right before the automated simulations are performed on different hybrid environments also take place in this script.

4. APPLICATIONS FOR REALISTIC EDGE COMPUTING SCENARIOS

There are numerous scenarios in which an edge device gets provisioning from a cloudlet server in Edge Computing. These scenarios or the solutions exhibit either heavy I/O, heavy processing, or both. Due to this reason, we implemented two different applications. One of them is to compare the results for the scenarios that require low I/O but heavy processing on the environments created. The other one is to simulate a scenario that displays heavy I/O but low processing.

The scenarios that we implemented consist of a server code and a client code. The server code runs on top of the cloudlets whereas the client code runs on top of every Mininet hosts or Raspberry Pi hosts depends on the environment used. For example, in Env-1 there is only one virtual cloudlet and ten hosts all inside the Mininet. In this case, when the Topology Creator application is started, it triggers the server code on the virtual cloudlet and triggers the client code on every virtual host where the server and the client code is picked according to the application used. In the following sections, we will give the detailed explanation for these applications.

4.1. Server-Client Communication Design

In this section, we will present structural details of the applications we used in our edge computing scenarios and how client-server system was designed realistically between the hosts-cloudlets.

4.1.1. Request Schema with Poisson Point Process

In order to simulate real-life queuing systems properly (including computer networks), Poisson point process is a widely-used method to model the arrival schema of these systems (arrival schema of packets, request or clients) [60]. We can give a techni-

cal summary about this point process as follows; Poisson Point Process is a stochastic process where the points on a line are distributed in a way that the interval lengths between consecutive points are the samples of an exponential distribution with a certain parameter (i.e, mean or λ). This line, in the context of queuing theory, can be considered as the timeline and the points on this line are arrival time points of real-life phenomenon such as phone calls, requests, network packets or even arrival times of clients.

In our communication design that takes place between hosts and cloudlets, the request sending schema is designed as Poisson Point Process. With a given mean interarrival time, the wait-time between two consecutive requests in every host is sampled from the exponential distribution with a mean value equal to this given mean interarrival time. This ensures that every request arrival time is independent of other previous requests like in real life that shows a certain behavior related to the given mean-interarrival time.

For our request schema, to test our environments, we decided on eight different mean interarrival times which are respectively 5, 10, 15, 20, 25, 30, 35, 40 seconds between requests being made by the hosts. Of course, the request schema can be a lot different than this for other edge computing scenarios but, let's say, for face detection applications performed by end devices (e.g., smartphones, IoT devices) these numbers make sense. The request arrival rates, which we used to carry out performance evaluations of our systems, for these mean interarrival times correspond to 0.20, 0.10, 0.066, 0.05, 0.04, 0.033, 0.028, 0.025 requests/second respectively. When the simulations run in our environments, these different request arrival rates emphasize different loads for all components of the system. Our idea is to see how our different environments with different hybridization levels behave under different load levels.

4.1.2. Background Traffic Generation

In real networks, network latency is dependent to a large extent on the link loads. In other words, the network latency between the source and destination varies according

to the links' utilization. That is why simulating real networks requires background network traffic on links because evaluating network delays with highly underutilized links might be misleading. In our experiments, background traffic generation has been achieved with Iperf [61] which is a widely used traffic generation tool. Setting up the Iperf servers on the cloudlets and the Iperf clients on the hosts, we managed to generate background traffic for our network with different characteristics and loads. In Chapter 5, we will also show how these different characteristics and loads affect the average service delays in our edge computing scenarios.

As we stated earlier, our link bandwidths between the switches and the cloudlets are equal to 100Mbps. We wanted to examine the performance of different setups under different link utilization levels by sending data traffic from Iperf clients to the Iperf servers. The traffic generation is modeled as follows; when the server code starts running on the cloudlets, it also sets up Iperf server to accept the background traffic from clients. The client codes, on the other hand, set up Iperf clients on the hosts with the destination IP address of the polled cloudlet. Once the connection has been established, every host starts sending constant data traffic to the cloudlets. With knowing the number of hosts in the environments we could adjust the background traffic, specifically the traffic on the cloudlet links since the generated traffic from hosts to cloudlets has to pass through these cloudlet links. We tested our systems with different background traffic levels that led to different link utilization. We generated 4Mbps, 6Mbps and 8Mbps traffic from every host to the cloudlets that accumulated a total of 40Mbps, 60Mbps and 80Mbps (there are 10 hosts) link loads respectively. Apart from the performance evaluation of the system under different background traffic loads, we used 40Mbps background traffic generation version as the default configuration when evaluating other performance metrics (Figure 4.2).

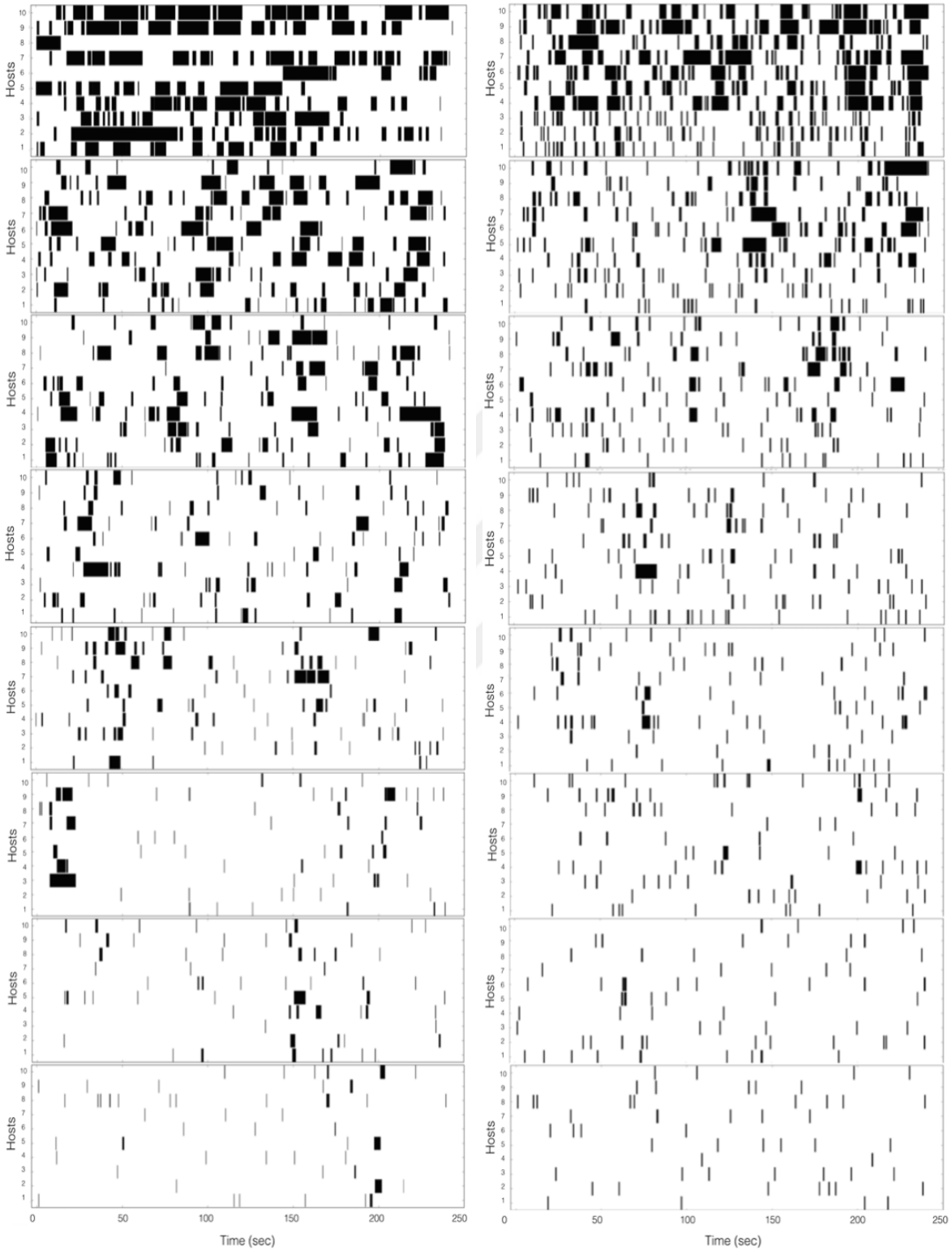
4.2. Synthetic Workload Generation

In this application, every host communicates with the cloudlet over a TCP socket. First, the hosts are sending one integer number as a symbolic request with a constant light network traffic where we set the network traffic on black links shown in Figure 3.3

so that it occupies 40% of the total bandwidth of the cloudlet link. After receiving the symbolic integer from a host as a request, the service on the cloudlet starts running a square root function for the numbers from one to one million. We choose the square root function to create synthetic load on the CPU of the cloudlet since it is a labor-intensive function for the CPU. Afterwards, the cloudlet returns to the host with a message indicating that the processing is finished and the request has been answered. The host that gets the message waits for a time interval which is determined by Poisson Point Process for a given mean interarrival time and sends another request in the same way.

For this method, the service time for any request is constant while the cloudlet is idle since the number of CPU cycles is the same to execute the function. In other words, if there is no other request that is being processed on the cloudlet, the service time for any request from any host takes the same time to be replied. However, when the cloudlet is not idle, the provisioning takes longer since the cloudlet processing more than one square root function at the same time in a multi-threaded fashion. Thus, when the service time window widens, the hit chance of another request increases. It results in processing other requests and the window widens more and so on. Therefore, this method allows us to examine the service delays for the hosts and to have a better understanding of how to assess the effect of the cloudlets' processing power on the edge computing scenarios.

Figure 4.1(a) and Figure 4.1(b) shows the service time for every request that every host experienced throughout one repetition of the simulation, for the virtual cloudlet inside Mininet in Env-1 and for a real PC as a cloudlet in Env-2 respectively under this scenario. As can be seen with longitudinal sections on a figure for a specific mean interarrival time, the service time windows widen exponentially when more than one request is being processed at the same time. If the requests per second increases, it can be seen that the virtual cloudlet inside Mininet cannot reply to the requests coming from the hosts at an acceptable level. On the other hand, real cloudlet performs relatively better although our main desktop PC on which virtual cloudlet exists within Mininet is more powerful than the PC that we used for the cloudlet. Due to the fact



(a) Cloudlet inside Mininet

(b) Real PC as a Cloudlet

Figure 4.1. Service Durations for the requests under heavy CPU utilization, from top to bottom mean interarrival time between requests are 5-10-15-20-25-30-35-40 seconds

that all Mininet components are running along with other applications on the main PC and as a result the virtual cloudlet displays poor performance.

4.3. Face Detection with OpenCV

In order to test the performance of our environments created when a real-life application is used, we implemented a Face Detection application as our second Edge Computing scenario. For the IoT devices, for example, image processing in real-time is a challenging task. Their processing power usually is not enough for such CPU intensive applications and more importantly they have energy efficiency concerns. Therefore, Edge Computing offers great benefits to evolving IoT systems and we wanted to use this application in order to test the performance of our hybrid systems.

For this particular job, we used OpenCV, widely used leading open source library for computer vision, image processing and machine learning applications. The frontal face detection is achieved with the help of a Haar Cascade. This classifier, developed by Viola and Jones [62] and later improved by Lienhard and Maydt [63], largely speeds up the facial detection process and the faces can be fairly reliably detected in real-time. With the help of their work and OpenCV library that utilizes the cascade, we implemented our own Edge Computing scenario where we detect faces in an image on a cloudlet and send them to the requesting hosts.

The general flow of the face detection procedure is as follows (also presented in Figure 4.2). The clients upload one image to the server using the TCP connection it has between itself and the cloudlet. Then, all the faces in the received image are being detected on the server (i.e., cloudlet) using OpenCV Face Detection application. The processed image (i.e., the image after the green rectangles around the faces are drawn) is sent back to the respective host. As soon as the processed image is downloaded/received completely, the waiting time for the next request starts where the same procedure is repeated for the next request. We implemented our own mechanism to send/receive an image with TCP-socket programming. The details of image sending and image receiving procedures that utilize TCP-socket programming is presented in Figure 4.3.

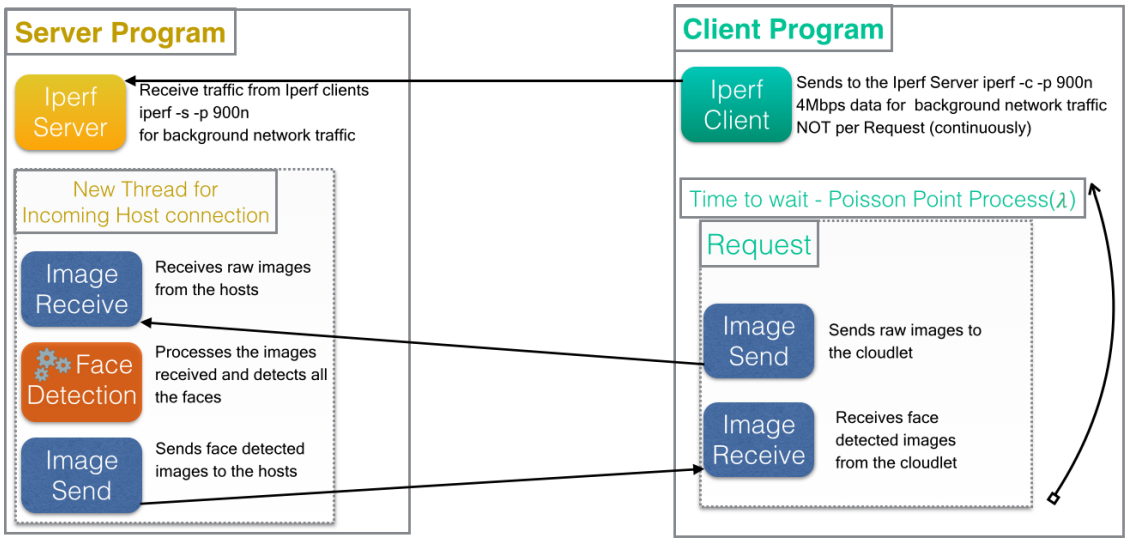


Figure 4.2. Server-Client Interaction in Face Detection Application

Using these procedures, complete server-client system has been built between the hosts and the cloudlets where the pseudo-codes run on them are present.ed in Figure 4.4 and Figure 4.5 respectively. When a new host choose (like in Env-5-6-7-8, the hosts randomly choose one of two cloudlets) its cloudlet to get provisioning and start sending images, a new thread is being forked in the server code specific to the connected host. In this thread, the current image is being processed and sent back.

In Figure 4.6(a) and Figure 4.6(b), an example image uploaded to the cloudlet(s) and the same image downloaded from the cloudlet after it is processed can be seen respectively. We have a set of images like these where the size of the images varies from 200KB to 3MB for the face detection application. Besides, the images in this set can include only one face or multiple faces. Both of these characteristics imply that the images in the set show an alteration in the time needed for processing them since the Haar Cascade depends on both the size of the image and the number of faces it includes.

The set consists of 60 images that are randomly chosen by the hosts at the beginning of a new request. After the selection, they sent the chosen images as we

```
1: function ImageSend(imagePath)
2:     image ← open(imagePath)
3:     imageByteLength ← image.size
4:     Complete imageByteLength to 8 bytes by putting 0's in front
5:     socket.sendall(imageByteLength)        ▷ Send the length of the Image first
6:     imageData ← image.read(8192)   ▷ Read the first 8192 Bytes of the Image
7:     while imageData is not empty do
8:         socket.sendall(imageData)
9:         imageData ← image.read(8192)
10:    end while
11: end function
12: function ImageReceiveAndWrite(imagePath)
13:    ▷ WAITALL parameter ensures exactly indicated amount of bytes received
14:    imageByteLength ← socket.recv(8, WAITALL)    ▷ Receive Image Length
15:    Convert imageByteLength from byte to integer as imageLength
16:    remainingImageLength ← imageLength
17:    receivedBytes ← socket.recv(4096, WAITALL)
18:    while receivedBytes is not empty do
19:        write receivedBytes to the image on imagePath
20:        remainingImageLength ← remainingImageLength − 4096
21:        if remainingImageLength > 4096 then
22:            receivedBytes ← socket.recv(4096, WAITALL)
23:        else
24:            receivedBytes ← socket.recv(remainingImageLength, WAITALL)
25:            write receivedBytes to the image on imagePath
26:            receivedBytes ← empty
27:        end if
28:    end while
29: end function
```

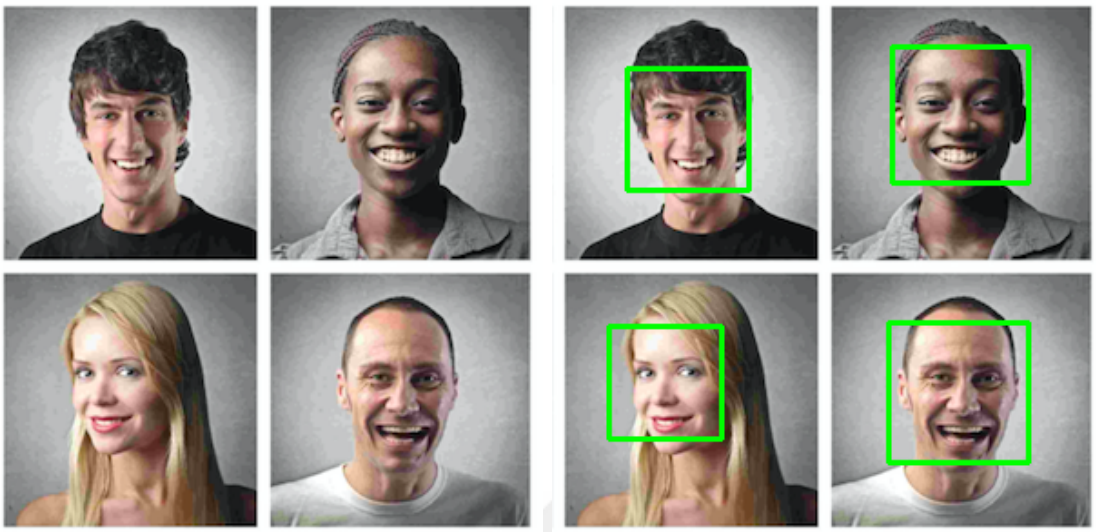
Figure 4.3. Pseudo-code for the image send and receive with TCP socket programming.

```
1: procedure SERVER
2:     Create TCP-socket socket as a global variable to connect server
3:     socket.bind(anyHost, portToConnect)
4:     socket.listen(30)              ▷ Socket accepts connection requests up to 30
5:     repeat
6:         connection ← socket.accept()        ▷ Whenever a client tries to connect
7:         StartNewThread(CLIENTTHREAD(connection))
8:     until the end of the simulation
9: end procedure
10: function CLIENTTHREAD(connection)
11:     while true do          ▷ Whenever the client of this thread sends an image
12:         currentImagePath ← build a new path with connection.address
13:         IMAGERECEIVEANDWRITE(currentImagePath, connection)
14:         pathOfProcessedImage ← FACEDETECTION(currentImagePath)
15:         IMAGESEND(pathOfProcessedImage, connection)
16:     end while
17:     connection.close()
18: end function
19: function FACEDETECTION(imagePath)
20:     cascadePath ← Path of Haar Cascade Frontal Face Feature Classifier
21:     faceCascade ← CascadeClassifier(cascadePath)
22:     image ← OpenCV.imageRead(imagePath)
23:     Convert image to grayscale image using OpenCV
24:     faceList ← faceCascade.detectMultiScale using grayScaleImage
25:     Draw green rectangles on image for every face found in faceList
26:     Write the new image to the path pathOfProcessedImage
27:     return pathOfProcessedImage
28: end function
```

Figure 4.4. Pseudo-code for the server code runs on the cloudlets.

```
1: procedure CLIENT(meanInteraarivalTime)
2:     Create TCP-socket socket as a global variable to connect server
3:     socket.connect(server'sIP, portToConnect)
4:     lamda ← 1/meanInteraarivalTime
5:     repeat                                  ▷ Every iteration constitutes one request
6:         timeIntervalToBeWaited ← PoissonProcess(lamda)
7:         sleep(timeIntervalToBeWaited)
8:         set next image from the set as currentImage
9:         sendingTime ← CurrentTimeStamp
10:        IMAGESEND(currentImage)
11:        IMAGERECEIVEANDWRITE(pathForImageToBeReceived)   ▷ waiting
   for the server's response for this request takes place in this function
12:        receivingTime ← CurrentTimeStamp
13:        serviceDelay ← receivingTime − sendingTime
14:        write serviceDelay to the log
15:    until the end of the simulation
16: end procedure
```

Figure 4.5. Pseudo-code for the client code runs on the hosts.



(a) Image sent to the cloudlet (b) Image received from the cloudlet

Figure 4.6. One Example of the Images Used (taken from Google Images)

mentioned above. Since in real life the set of images that would be sent to the cloudlets do not necessarily show a certain type of behavior, we decided to use this kind of set in our experiments. For the face detection application, we did not perform event log analysis since the service duration of the requests are not entirely dependent on the cloudlet CPU utilization. The service duration of the requests varies from one image to another and the analysis would not be indicative for any obtained result .

5. PERFORMANCE EVALUATION WITH HYBRID TESTBEDS

In this section, we present the tools to examine the performance of our testbed environments by explaining the logging mechanisms throughout our system. For illustrative purposes, later in this section, we present a number of charts of the important metrics that can be obtained from our environments. The framework we created, which allows us to make simulations for edge computing scenarios over our testbeds, allows us to extract various performance charts. In order to better understand the logging mechanism and the comparison of the environments, these charts have a significant value.

5.1. Logging Mechanisms and Performance Charts

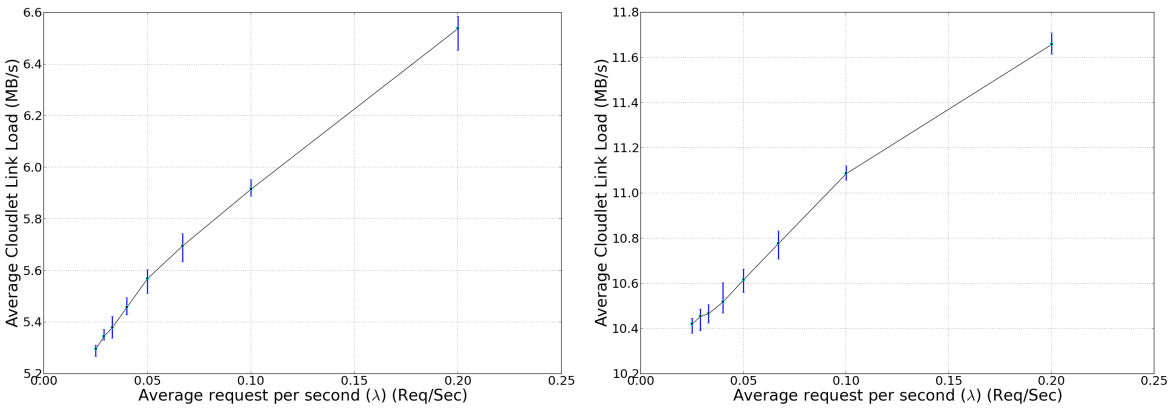
5.1.1. Cloudlet Link Load

On the Traffic Monitor northbound application, the information of the MAC addresses that cloudlets have and the ports that they are connected to their switches kept. By using this information we could achieve to measure the cloudlet link loads as follows. Let say one host starts a request and wants to get service from one cloudlet. Since the request is based on the TCP/IP, it only knows the IP address of the cloudlet that it will get the provisioning. After successfully connecting to the cloudlet it starts sending the image inside the request. From this point on, the northbound application works on Layer 2. When the OVS's forward the packets it eventually comes to the OVS that the cloudlet is connected. In this OVS, the packet is forwarded to the cloudlet since it matches with the flow rule inside the switch.

To monitor the traffic, RYU controller requests the amount of transferred bytes on every port of every switch, every second. When the stat request messages come to the switches from the controller, we match the destination MAC address and the incoming

port together. This matching creates uniqueness and the stat reply message from this port includes the byte count on this port. To give an example, let's say the cloudlet that we want to monitor the link load has MAC address 10:00:00:00:00:00 and it is connected to its switch on Port-1. If the port number is not included into the matching mechanism, the messages from the other ports on that switch that has destination MAC address with 10:00:00:00:00:00 would be matched too. If the destination MAC address is not included to the matching mechanism, it would match Port-1 of all switches regardless of the destination MAC addresses. However, using both MAC address and the port together leads to exactly one unique destination. Thus, when the stat reply message with this correct matching is achieved, the current transferred byte count on the related port is written to a file with the timestamp.

In Figure 5.1(a) and Figure 5.1(b), the cloudlet link loads in Env-1 that are extracted using these logs can be seen. For example, in Figure 5.1(a) the cloudlet link load is around 5MB/s. The reason is that every host creates 4Mbps background traffic via Iperf that has the destination as the cloudlet. Since we have 10 hosts in this environment setup, it results in 40Mbps in total which is equal to 5MB/s as an additional cloudlet link load (on top of the cloudlet link load due to image sending/receiving). In Figure 5.1(b), the hosts are sending 8Mbps background traffic and result in 80Mbps that equals to additional 10MB/s cloudlet link load.



(a) Background Traffic is equal to 40Mbps (b) Background Traffic is equal to 80Mbps

Figure 5.1. Cloudlet Link Loads under different background traffic levels and different requests per second

5.1.2. Cloudlet CPU Usage

Cloudlet CPU usage log is only valid for the environments that have real cloudlets. Since the cloudlet inside Mininet uses the same CPU cores with other processes that run on the computer, extracting the CPU usage of the cloudlets -or any other Mininet hosts' CPU usage- is not meaningful. To extract the CPU usage on the real cloudlet, we utilized the "mpstat" command that comes within the "sysstat" package for Ubuntu. Command "mpstat" reports the percentage of time that CPU or CPU's are idle with the timestamp.

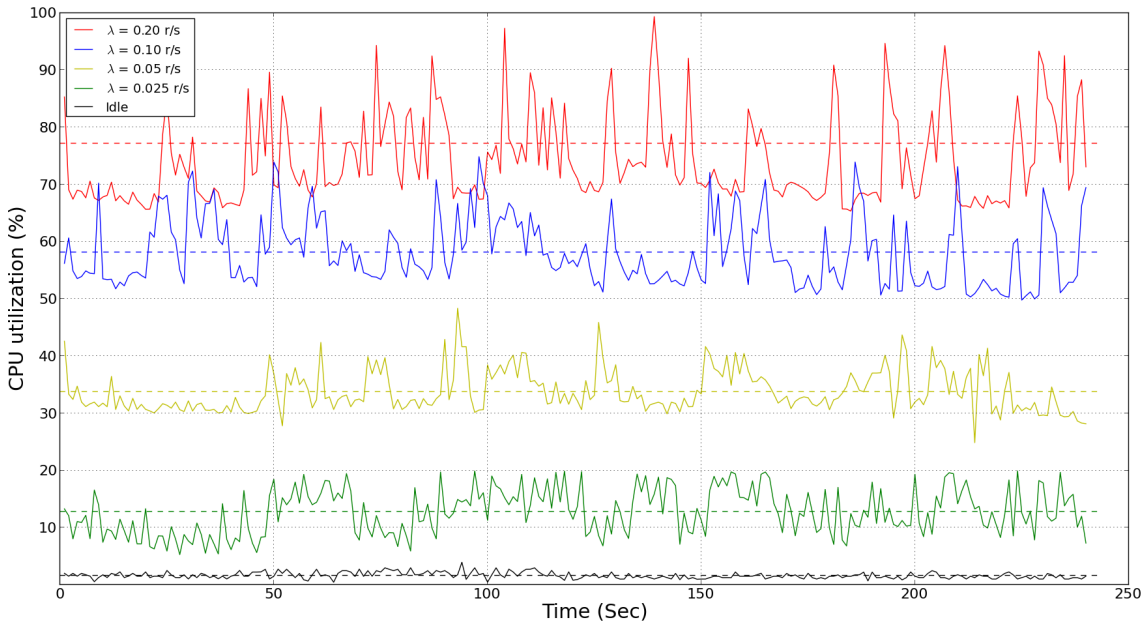


Figure 5.2. CPU utilization of Real Cloudlet PC under various loads

In our scenarios, the cloudlets do not have any other purpose other than serving to the hosts, in other words, there is no other programs running on the cloudlets except default Ubuntu processes and services. Therefore, we can imagine our cloudlets as Ubuntu Virtual Machines that are dedicated to giving provisioning to the end users, in our case to the hosts. Because of this reason, we extracted the raw CPU usage with "mpstat" on the cloudlets where we measured that the average idle process utilization is below 1% (Figure 5.2) when the cloudlets are not serving to any host. The idle

percentage we take into consideration is the average idle percentage of all cores since we do not assign any specific core to the server and the data requested by hosts can be processed on any core in our system.

5.1.3. Service Delays for the Hosts

The reason why the current state of the technology needs edge computing is actually because of the fact that we intend to lower the service delays that end users experience from the services they want to get. Thus, in this thesis, service delays for the hosts had become our main criteria for the performance evaluation.

To extract the service delays, we designed the client application running on the hosts in a way that each host, either a Mininet host or a Raspberry Pi host, keeps the timestamp right before sending a request and the timestamp right after the end of a reply from the cloudlet for that request. The difference between these timestamps gives us the service delay for that request. Besides, our client application allows each host to write its service delays to a different log file respective to its IP address after parsing the IP address of the host in order for us to analyze and compare the service delays host-by-host later.

5.1.4. Event Logs for the Hosts

In Figure 5.3, the event log for every host for the simulation that lasts 240 seconds in Env-4 with 10 hosts and 0.05 requests per second by each host to the cloudlet can be seen. The timestamps are the real simulation timestamps and the thickness of the lines shows the duration of corresponding requests which can be seen in the bar charts for Host-1 and Host-10. As we explained earlier, Host-1 is a Mininet host whereas Host-10 is a Raspberry Pi in Env-4. Even though the mean interarrival time between the requests is 20 seconds ($\lambda = 0.05$ r/s) for these hosts, this figure shows only one of the five repetitions of the simulations with this configuration. The service delay logs for the hosts are used to extract these charts, however it is important to lay emphasis on the time synchronization between the main PC and the Raspberry Pi's because the

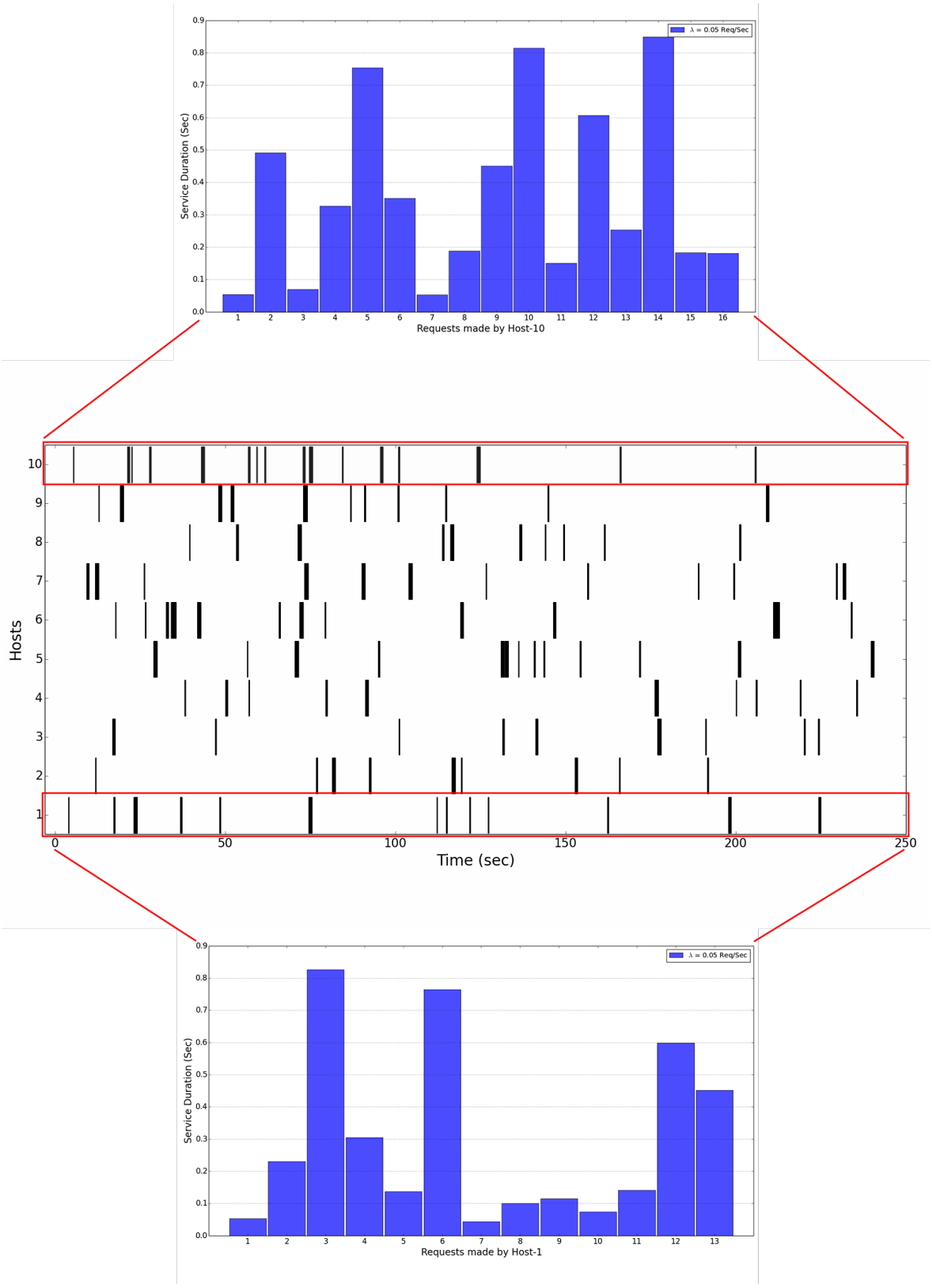


Figure 5.3. Env-4, Event Log for 0,05 Req/Sec with 40Mbps Background Traffic

timestamps need to be aligned for every host. For this purpose, first we adjusted time on every Raspberry Pi hosts that will be used before running the simulations.

5.2. Comparative Evaluation of the Test Environments

In this section, we will present the performance evaluation comparisons between the environments we created using different metrics. Since our main purpose, in this research, was to see how different the edge computing scenarios, such as face detection in our study, behave when the hybridization level of the testbeds that are used for the experimentation varies. In order to see the difference in the hybridization levels, when performing an analysis, we used the exact identical load for all environments for that analysis. Thus, the different results for the different testbeds stem from the differences in their hybrid structures. We mainly focused on the average service delay differentiations for the environments since the most primal motivation behind Edge Computing paradigm is to solve challenges related to the delay sensitiveness.

The average service delays for Face Detection application in Env-1, Env-2, Env-3 and Env-4 under different request loads are illustrated in Figure 5.4. All four environments have the same configurations for the simulation that lasts 240 seconds. In other words, the number of the hosts (10 hosts) and cloudlets (1 cloudlet), link capacities between every component and the background traffic generated (40Mbps) are the same in all environments. The only difference, as we stated earlier, is the usage of the real components instead of their virtual counterparts. We repeated the simulation with the same configurations five times for every λ value in every environment. Then, we compared the average service delays for each of them.

In Figure 5.4, it appears that when the request load is below 0.04 requests/second, average service delays for Pure Mininet environment (which is Env-1) were lower yet they heightened dramatically when the request rate is increased to 0.20 requests/second. The reason is that Mininet uses virtual components to emulate the network and carries the traffic through these virtual components which results in less latency between the endpoints of the links even though the capacities of these links are set as the same

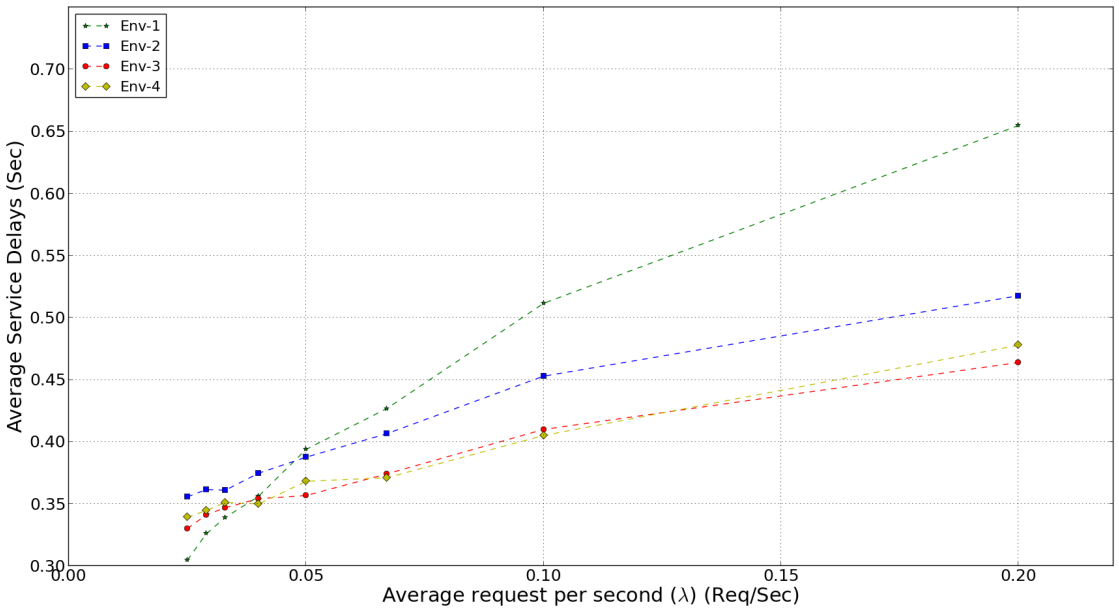


Figure 5.4. Average Service Delays of Env-1-2-3-4 with 40Mbps Background Traffic

capacities of the real links in other environments. However, since the cloudlet in this environment also runs inside the Mininet along with other components, the service delays are mostly determined by the processing time of the images. When the load increases to 0.20 requests/second, the images are sent to this cloudlet by 10 hosts create heavy CPU load on the main desktop machine and this affects the service delays greatly because of the fact that the desktop machine also emulates Mininet and keeps running many other services/processes.

In Env-2, where only the cloudlet and the attached link are real while other components are still emulated within Mininet, the average service delays are slightly more than the delays in Env-3 for every λ value. The result shows the effects of the real switch usage in the testbeds because all the switches used in Env-2 are emulated in Mininet, while Env-3 has one standalone OVS running on the laptop. This clear average service delay difference between Env-2 and Env-3 stems from the time consumption for processing the packets in virtual switches. Despite the fact that Mininet uses OVS to emulate the switches, the system calls performed by the hypervisor to the kernel in

order to process the packets take more time in the desktop machine. The OVS runs on the laptop processes the packets faster, by using the same system calls, since no other application runs on the laptop. Env-3 and Env-4 differ from each other only in the set of hosts used in them. For Env-4, we used three Raspberry Pi hosts and seven Mininet hosts whereas in Env-3 only Mininet hosts are used. As we will later present the differences between the Raspberry Pi's and Mininet hosts, the closeness of the curves (Env-3 and Env-4) indicates that the usage of the real hosts in the testbeds has relatively less effect than the usage of real cloudlets and real switches for our edge computing scenarios.

We again used the Face Detection application for the performance evaluation comparison among Env-5, Env-6, Env-7 and Env-8 where there are two cloudlets in the system. The simulation configurations we used are the same as before; the number of hosts is 10, the link capacities are the same and the link load on both cloudlets links are 40Mbps. We achieved the same background traffic on both links by sending additional 40Mbps traffic from Host-6 to Cloudlet-2 and keep sending other background traffic from other hosts to Cloudlet-1. In this way, every host sends again 4Mbps background traffic generated by Iperf to the destination 10.0.0.21 (Which is the IP of Cloudlet-1) which makes every switch link load equal to the ones in the previous simulation we performed for Env-1-2-3-4, whereas the only link got affected by Host-6's additional traffic is the link between Cloudlet-2 and Switch-3. The reason for doing so is to use the same cloudlet link load for both of them as it affects the delays of the services they provide. It is important to note that, in this setup, the hosts poll one of two cloudlets randomly at the beginning and they send their images to the polled cloudlets independently from the background traffic they are sending. Again, we repeated the simulation with the same configurations five times for every λ value in every environment where each of the simulations lasts 240 seconds.

In Figure 5.5, the result of this simulation is presented. The average service delay curve for the hosts in Env-5, which is the only environment that includes two virtual cloudlets, shows the same characteristic as in Env-1 but slightly increased. Even though there are two cloudlets in the system and their load of provisioning is the half

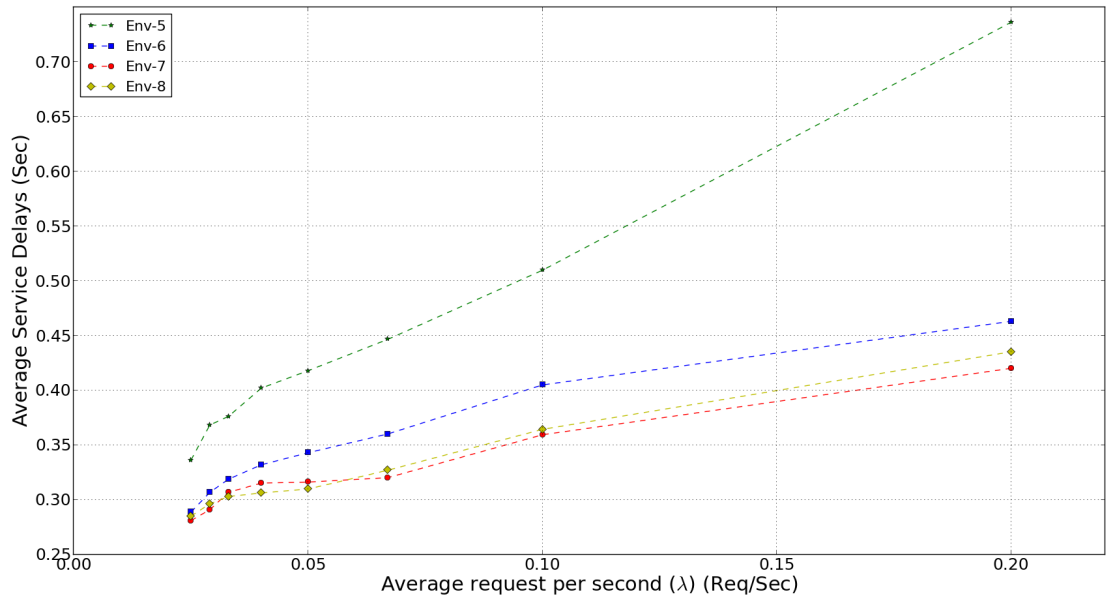


Figure 5.5. Average Service Delays of Env-5-6-7-8 with 40Mbps Background Traffic

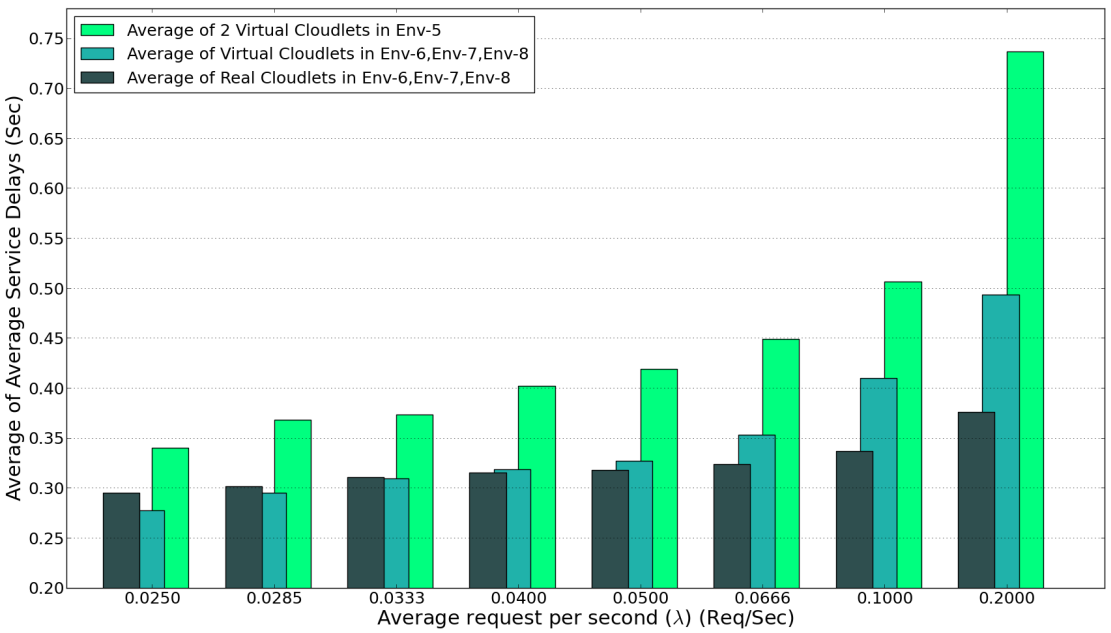


Figure 5.6. Average of Average Service Delays for the Real and Virtual Cloudlets in Env-5-6-7-8 with 40Mbps Background Traffic

on average (for each cloudlet, 5 hosts get provisioning on average), the average service delays show the same characteristic because the virtual cloudlets run within Mininet inside the main desktop computer and they produce the same CPU load as in Env-1 with the preservation of the same request rates. The increase, however, is due to the fact that there is one more Mininet component that is virtualized and has to be taken care of by the same desktop machine.

Starting from Env-6, we have one real cloudlet and one virtual cloudlet in the system where we can see the average service delay improvements for the hosts. The cloudlets in this environments have half of the provisioning load on average compared to Env-2 and this leads to better average service delays as expected. The difference between the Env-6 and Env-7 is still notable for this simulation but for the λ values below 0.05 requests/second, the average service delays are closer. This is because the service delays for the hosts that get provisioning from the real cloudlets are more than the ones that get provisioning from the virtual cloudlets for the small λ values as we also stated before for Figure 5.4. This type of behavior for the systems that have both virtual and real cloudlets, cancel out the service delay decrease thanks to the usage of a real switch for the small λ values, and thus the curves for Env-6 and Env-7 appear closer to each other for these small λ values.

In Figure 5.6, the explained behavior above can be seen as we compare the average of average service delays for the two virtual cloudlets in Env-5, the virtual cloudlets in Env-6-7-8 and the real cloudlets in Env-6-7-8. This figure is derived from the average service delays of the hosts after we group them according to the cloudlets they got provisioning in the simulations. In other words, the service delays for the hosts that are connected to the virtual cloudlet in Env-6, the service delays for the hosts that are connected to the virtual cloudlet in Env-7 and the service delays for the hosts that are connected to the virtual cloudlet in Env-8 are grouped together. The same is performed for the hosts that are connected to the real cloudlets in those environments. The reason why we did not include the two virtual cloudlets in Env-5 to the average of virtual cloudlets in Env-6-7-8 is the fact that Env-5 is not structurally identical to other three environments as can also be seen in the figure. Figure 5.6 shows the clear

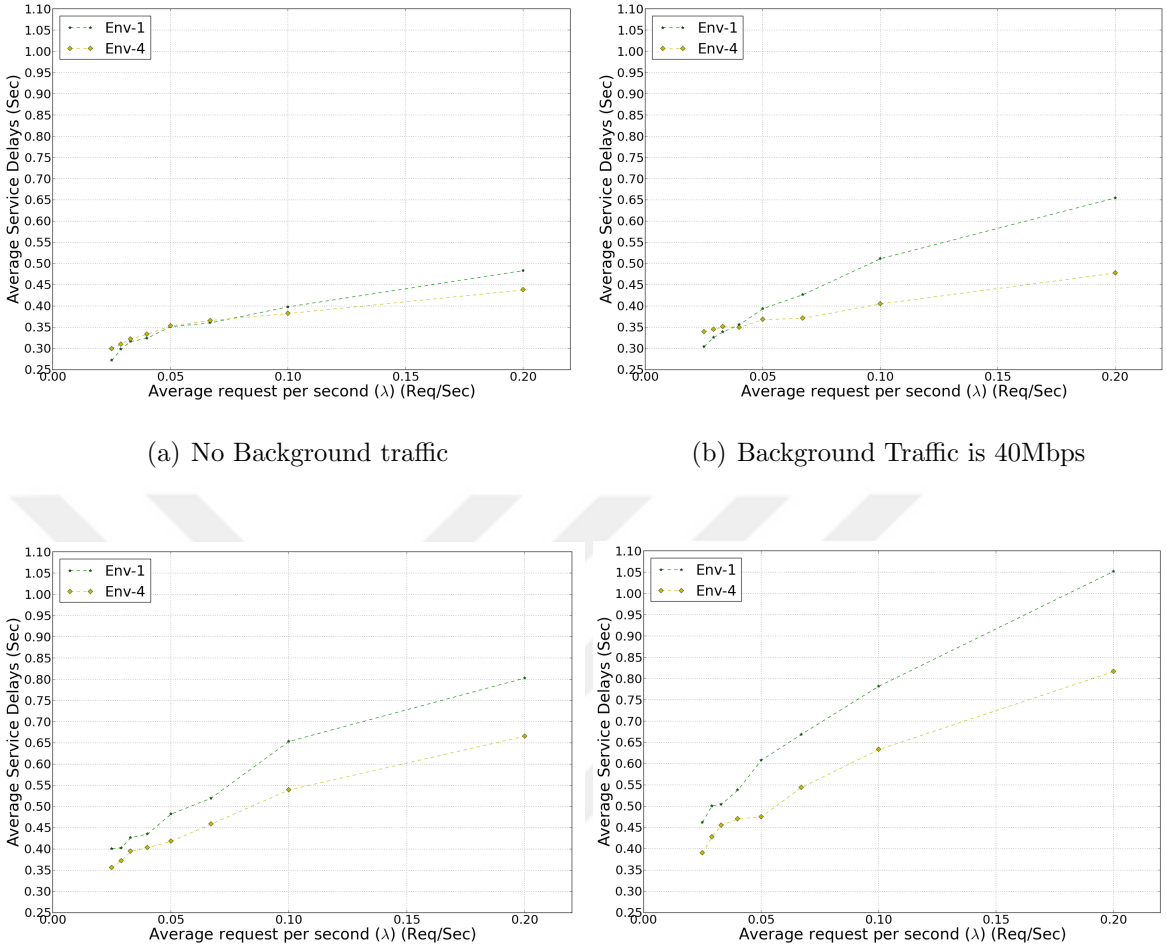
difference between the usage of real cloudlet and the usage of virtual cloudlet in our experiments that take place on the hybrid environments.

The system in Env-7 again shows distinguishably lower average service delays than Env-6 for the λ values greater than 0.05 requests/second because of the real switch usage. On the other hand, we again observe that the average service delay difference between Env-7 and Env-8 are close to each other and usage of Raspberry Pi's as real hosts makes little difference.

We also wanted to show how different background traffic modes affect different hybridization levels in our edge computing scenario. We wanted to stress this difference by comparing Env-1 which includes nothing but virtual Mininet components and Env-4 which is the most hybridized environment. Since we will compare the traffic modes, we performed it for the environments that have only one cloudlet to avoid changing other parameters that might also have an effect.

As we explained in Chapter 4, we increased the traffic sent by hosts to the cloudlet. Figures 5.7(a), 5.7(b), 5.7(c) and 5.7(d) show the average service delays for Env-1 and Env-4, under no background traffic, 40Mpbs, 60Mbps and 80Mbps background traffic respectively. The other configurations are the same in both environments and the simulations again lasted 240 seconds and repeated five times before calculating the averages for the environments using Face Detection application.

The figures indicate that as the background traffic increases, the average service delays in Pure Mininet environment gets strikingly higher. The divergence between the slopes of two curves shows that Env-1 suffers from the background traffic increase more than Env-4 does even though everything but the hybridization level is the same. In Figure 5.7(b), Env-4 has $\approx$0.50 second average service delay when λ is 0.20 with 40Mbps background traffic request/second and $\approx$0.80 second average service delay with the same λ under 80Mbps background traffic. Thus, the effect of the background traffic is 0.30 second. For Env-1, as can be seen in 5.7(d), the average service delays for the same λ values are $\approx$0.65 and $\approx$1.05 where the difference is 0.40 second.



(a) No Background traffic

(b) Background Traffic is 40Mbps

(c) Background Traffic is 60Mbps

(d) Background Traffic is 80Mbps

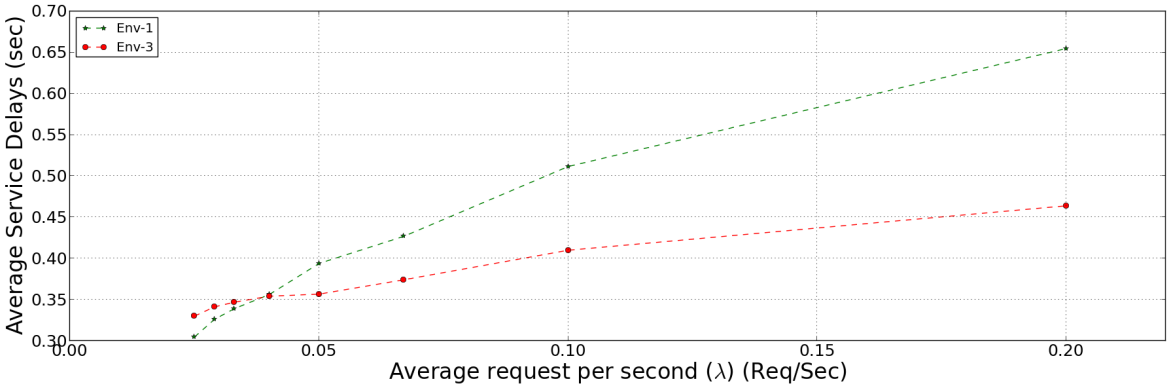
Figure 5.7. Average Service Delays for Env-1 and Env-4 with Different Background Traffic Modes

The more interesting part here is when there is no background traffic, Pure Mininet environment shows great similarity to our most hybrid environment even for the high request rates compared to its behaviors in other background traffic modes. This might be because of the decrease in the packets that have to be processed by the computer that runs Mininet. We can even make the same observation in Figure 5.7(c) and Figure 5.7(d) when the number of packets to be processed increased, lower service delay behavior is no longer reinforced in Pure Mininet environment for the light request loads.

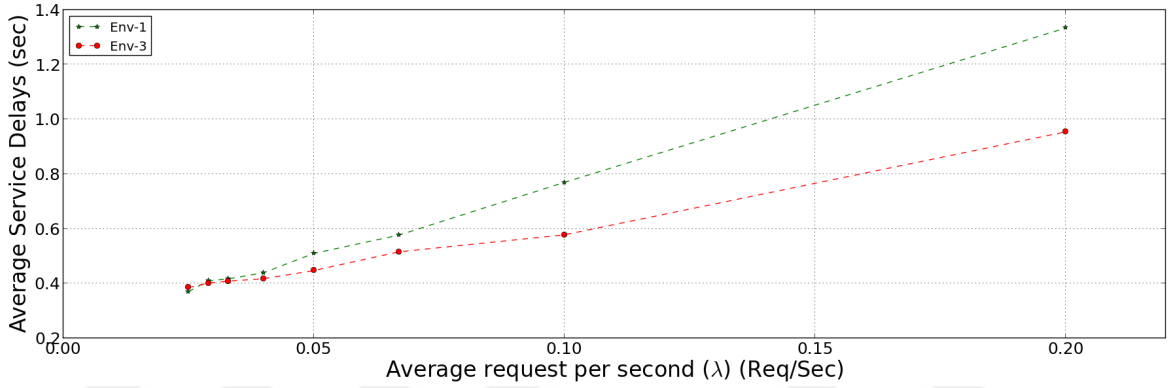
In terms of scalability, we wanted to contrast the characteristics of the environments we created. For this objective, we compared the environments with the different number of hosts that are used in them. We updated our Topology Creator application so that we have 15 and 20 Mininet hosts in the system other than the default versions with 10 hosts. Since we wanted to keep the hybridization level the same for the environments, we used Env-1 and Env-3 although Env-4 is the most hybrid environment we have. The reason is, Env-4 includes real and virtual hosts together and increasing the number of hosts in this environment is equal to decreasing the level of hybridization for the environment. However, in Env-3, since all the hosts are virtual, this is not the case.

For this experiment, the topologies used in the environments are different than before as we added more hosts. We kept the number of switches the same but we added one extra host to Switch 1, two extra hosts to Switch 2 and Switch 3 for the 15-host experiment. For the 20-host experiment, we added five extra hosts to the 15-host topology so that one host is added to Switch 1, one host is added to Switch 2, two hosts added to Switch 3 and lastly, one host is added to Switch 4. For the background traffic, we kept sending 4Mbps background traffic from the hosts for each environment because our main concern for this experiment was evaluating the behavioral differences between Env-1 and Env-3 according to their scalability, and every added host might generate its own background traffic in real life. From this aspect, the link load on the cloudlet link is equal to 40Mbps, 60Mbps and 80Mbps for 10 hosts, 15 hosts and 20 hosts respectively.

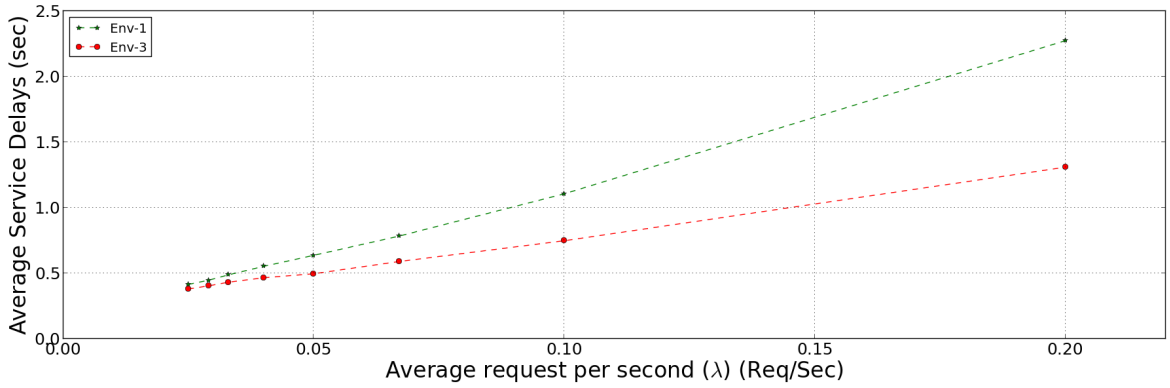
Using Face Detection application, the simulations again run for these new updated-environments so that one of them lasts 240 seconds and they are repeated five times for every λ. After collecting the logs that keep service delays for the hosts, we calculated the average service delays for 15 hosts and 20 hosts (for 10 hosts we used the same data that we presented in Figure 5.4). The results are illustrated in Figures 5.8(a), 5.8(b) and 5.8(c). The figures indicate that average service delay curves for Env-3, which includes a real cloudlet and a real switch, are less inclined to get worse when the system scales up. The hosts in Pure Mininet environments undergo serious service



(a) The Number of Hosts is 10



(b) The Number of Hosts is 15



(c) The Number of Hosts is 20

Figure 5.8. Average Service Delays for Env-1 and Env-3 when Different Number of Hosts Used

delays when the number of hosts are increased. As can be seen in Figure 5.8(c), the average service delays of the hosts are equal to ≈2.25 seconds when the request rate for the host is 0.20 requests/second.

Our another observation with the scalability is that under light request load (when λ is equal to 0.025, 0.0285, 0.033 and 0.04), Env-1 and Env-3 perform similarly. However, when the request load is high (when λ is equal to 0.05, 0.066, 0.10 and 0.20), the behavioral difference between Pure Mininet environment and the hybrid environment becomes more prominent as can be seen in Figure 5.9. This figure is obtained as follows; we categorized the service delays for the hosts in the environments into two groups. The first group is the average service delays for the lambda below 0.05 requests/second (which we call Light Request Load in the figure) and the second group is the average service delays for the lambda greater or equal to 0.05 requests/second (which we call High Request Load in the figure) for each environment that has 10-host, 15-host and 20-host.

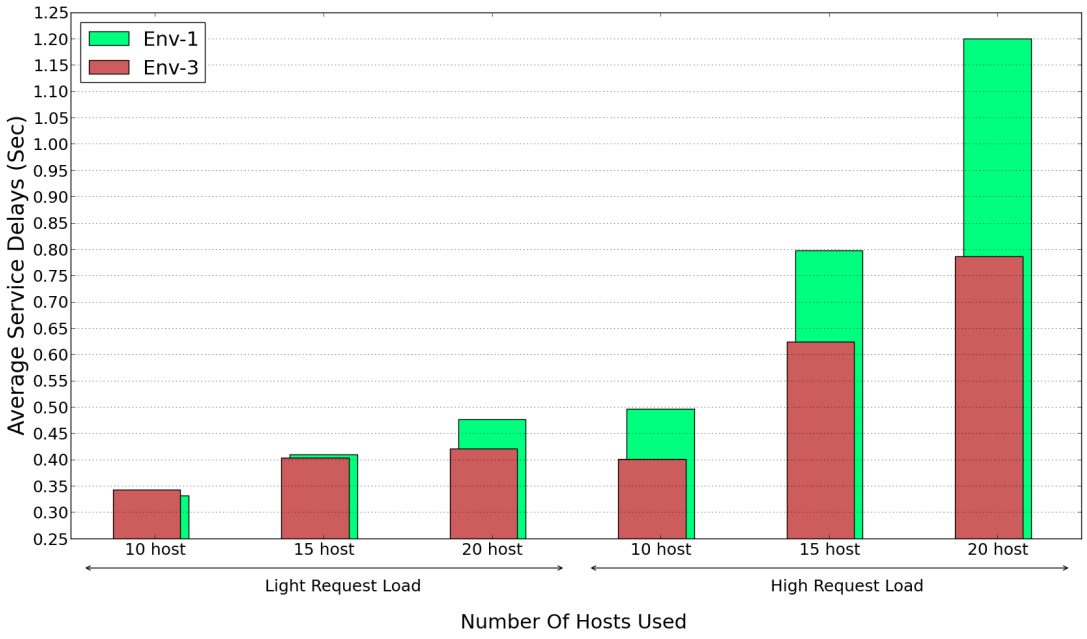


Figure 5.9. Average Service Delays for Env-1 and Env-3 under Light Request Load and Heavy Request Load

As we stated earlier the average service delay difference between Env-1 and Env-3 is open to interpretation when the request load is light even though the system scales up to twice as many hosts, however, under high request load average service delays appear noticeably higher in Pure Mininet environment (Env-1). In Figure 5.9, for example, the average service delays for the hosts are ≈0.80 seconds in Env-3 with 20 hosts, where as the same average service delays are raised to 1.20 seconds in Env-1 with 20 hosts. The degradation is almost 50% when everything but the hybridization level is the same for these environments. We can state our conclusion for this experiment as follows; cloudlet based edge computing studies which benefit from SDN should not be performed on mere Mininet environments if the scalability is a concern for the study.

Lastly, we compared the average service delays for the requests made by three Raspberry Pi's and seven Mininet hosts in Env-4 and Env-8. The result of this comparison can be found in Figure 5.10. We could extract these result since we know the host numbers that are turned into Raspberry Pi's. The service delays for the requests are kept host-by-host and we used Host-5, Host-8 and Host-9 (we turned these hosts into Raspberry Pi hosts as we mentioned in Chapter 3) when calculating the average service delays for the requests made by Raspberry Pi's in both Env-4 and Env-8. Other seven hosts are used to extract the average service delays for the Mininet hosts in the figure in both Env-4 and Env-8.

As can be seen in the figure, Mininet hosts and Raspberry Pi's do not seem to outperform each other in terms of the service delays for the requests they make. However, there might be a reason for this. The client code that runs on top of the host does not require much processing power. Therefore, Mininet hosts inside the desktop computer do not create much load on the CPU of the desktop machine and the usage of Mininet hosts does not influence the service delays. With a different scenario that consists of a more CPU-intensive client application, this result might change as Raspberry Pi's can outperform the virtual Mininet hosts.

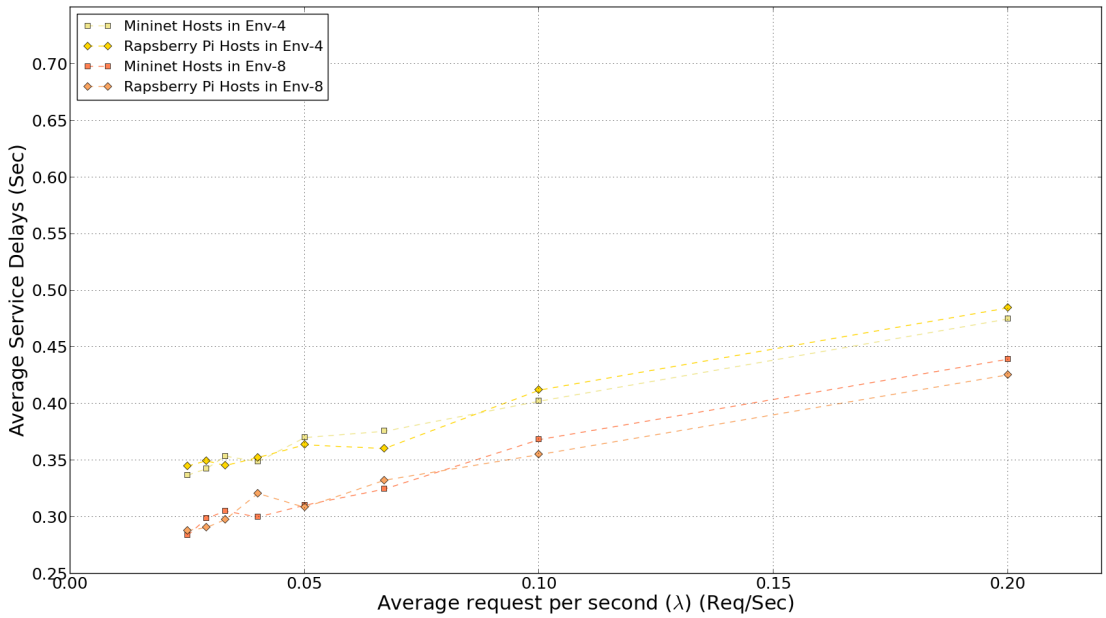


Figure 5.10. Mininet Hosts vs Raspberry Hosts Average Service Delay comparison for Env-4 and Env-8

6. CONCLUSION

In this thesis, we presented a detailed background for the new emerging technologies, such as Software-Defined Networking and Edge Computing, that promise to solve the current challenges for the computer networks in the era of mobile and IoT devices. The need for providing the complex and distributed applications to the mobile/end devices with high speed and efficiency necessitates this evolution for the computer networks. The testbeds, which are created to perform simulations for these evolved technologies, either utilize Mininet or are built using real hardware. However, both approaches fall short in some ways, and therefore in this study, we combined the aforementioned technologies on a hybrid testbed by bringing the two approaches together to carry out comparative performance evaluations for the realistic edge computing scenarios.

We started from Mininet, which is widely used SDN emulator, and expanded it with real hardware. We hybridized our system step-by-step and assessed the influences of using environments with different hybridization levels by means of comparing their performance results. Besides, we made the testbed available for the future studies with different edge computing scenarios. We provided monitoring mechanisms such as cloudlet link load, cloudlet CPU load, event logs for the hosts in the system, for these future studies. We presented every detail for the environment creation phases as well as the methods to hybridize the system. While creating the testbed, OpenFlow based Software-Defined Networking with RYU SDN controller utilized and cloudlet based Edge Computing model is adopted for the experiments on the testbed. For the experiments, we presented two different Edge Computing scenarios. In the first scenario, the synthetic CPU load generation is performed to comprehend the behavioral differences between the virtual cloudlet inside Mininet and a standalone real cloudlet. In the second scenario, we implemented our own server-client structure with Face Detection application for the end devices to carry out the performance evaluation of different hybridization levels for our environments.

We analyzed the average service delays, which is the most desired metric to be improved for end devices in real edge computing use-cases, for the environments we created to emphasize the observable distinctions between Mininet testbeds and hybrid testbeds in terms of performance. We tested our different environments with different configurations by changing the number of cloudlets, background traffic modes and the number of hosts. We concluded that pure Mininet environments suffer from scalability as far as the real-life cloudlet based edge computing scenarios are concerned. However, it has to be noted that there are deployment and maintenance trade-offs between the usage of pure Mininet environments and the usage of hybrid testbeds. The experiments on pure Mininet environments are easy to carry out and have more portability advantage over the experiments on our hybrid testbeds. Since there are physical components in the system for hybrid testbeds, the manual work to connect the components and making sure of their desired states before the simulation executions are some of the challenges that have to be tackled for the precisions of the experiments.

In our obtained results, we observed that the usage of dedicated Open vSwitch has a significant influence on the evaluation results for the environments created. The usage of a real cloudlet has even a greater effect in the performed comparative evaluations between pure Mininet environments and the hybrid environments. On the other hand, we concluded that using Raspberry Pis as real hosts does not make much difference with regards to the system's behavior.

As a future work, we want to increase the hybridization levels for the testbed and add wireless components into the topologies since in real life the mobile/end devices are mostly included into the systems via a WLAN. In addition to this, we want to try more edge computing scenarios to classify the types of them for which it is worth to undergo the challenges in creation of hybrid testbeds as in some cases Mininet performs fairly acceptable compared to hybrid environments.

REFERENCES

1. Benson, T., A. Akella and D. A. Maltz, "Unraveling the Complexity of Network Management.", *NSDI*, pp. 335–348, 2009.

2. Kreutz, D., F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-defined networking: A comprehensive survey", *Proceedings of the IEEE*, Vol. 103, No. 1, pp. 14–76, 2015.

3. Xia, W., Y. Wen, C. H. Foh, D. Niyato and H. Xie, "A survey on software-defined networking", *IEEE Communications Surveys & Tutorials*, Vol. 17, No. 1, pp. 27–51, 2015.

4. McKeown, N., "Software-defined networking", *INFOCOM keynote talk*, Vol. 17, No. 2, pp. 30–32, 2009.

5. Chiosi, M., D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen *et al.*, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action", *SDN and OpenFlow World Congress*, pp. 22–24, 2012.

6. Baktir, A. C., A. Ozgovde and C. Ersoy, "How Can Edge Computing Benefit from Software-Defined Networking: A Survey, Use Cases & Future Directions", *IEEE Communications Surveys & Tutorials*, 2017.

7. Nunes, B. A. A., M. Mendonca, X.-N. Nguyen, K. Obraczka and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks", *IEEE Communications Surveys & Tutorials*, Vol. 16, No. 3, pp. 1617–1634, 2014.

8. Doria, A., J. H. Salim, R. Haas, H. Khosravi, W. Wang, L. Dong *et al.*, *Forwarding and control element separation (ForCES) protocol specification*, Tech. rep., Internet Engineering Task Force, 2010.

9. McKeown, N., T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford *et al.*, "OpenFlow: enabling innovation in campus networks", *ACM SIGCOMM Computer Communication Review*, Vol. 38, No. 2, pp. 69–74, 2008.

10. Song, H., "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane", *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 127–132, ACM, 2013.

11. Campbell, A. T., I. Katzela, K. Miki and J. Vicente, "Open signaling for ATM, internet and mobile networks (OPENSIG'98)", *ACM SIGCOMM Computer Communication Review*, Vol. 29, No. 1, pp. 97–108, 1999.

12. Tennenhouse, D. L., J. M. Smith, W. D. Sincoskie, D. J. Wetherall and G. J. Minden, "A survey of active network research", *IEEE communications Magazine*, Vol. 35, No. 1, pp. 80–86, 1997.

13. Tennenhouse, D. L. and D. J. Wetherall, "Towards an active network architecture", *DARPA Active Networks Conference and Exposition, 2002. Proceedings*, pp. 2–15, IEEE, 2002.

14. Greenberg, A., G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie *et al.*, "A clean slate 4D approach to network control and management", *ACM SIGCOMM Computer Communication Review*, Vol. 35, No. 5, pp. 41–54, 2005.

15. Gude, N., T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown *et al.*, "NOX: towards an operating system for networks", *ACM SIGCOMM Computer Communication Review*, Vol. 38, No. 3, pp. 105–110, 2008.

16. Casado, M., M. J. Freedman, J. Pettit, J. Luo, N. McKeown and S. Shenker, "Ethane: Taking control of the enterprise", *ACM SIGCOMM Computer Communication Review*, Vol. 37, pp. 1–12, ACM, 2007.

17. Casado, M., M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown *et al.*,

"Rethinking enterprise network control", *IEEE/ACM Transactions on Networking (ToN)*, Vol. 17, No. 4, pp. 1270–1283, 2009.

18. Koponen, T., M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu *et al.*, "Onix: A distributed control platform for large-scale production networks.", *OSDI*, Vol. 10, pp. 1–6, 2010.

19. Jain, S., A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh *et al.*, "B4: Experience with a globally-deployed software defined WAN", *ACM SIGCOMM Computer Communication Review*, Vol. 43, No. 4, pp. 3–14, 2013.

20. *Open vSwitch*, http://www.openvswitch.org, accessed at December 2017.

21. Pfaff, B., J. Pettit, K. Amidon, M. Casado, T. Koponen and S. Shenker, "Extending Networking into the Virtualization Layer.", *Hotnets*, 2009.

22. Pfaff, B., J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme *et al.*, "The Design and Implementation of Open vSwitch.", *NSDI*, pp. 117–130, 2015.

23. Emmerich, P., D. Raumer, F. Wohlfart and G. Carle, "Performance characteristics of virtual switching", *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pp. 120–125, IEEE, 2014.

24. Erickson, D., "The beacon openflow controller", *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 13–18, ACM, 2013.

25. *Project Floodlight*, http://www.projectfloodlight.org/floodlight/, accessed at December 2017.

26. Ng, E., Z. Cai and A. Cox, "Maestro: A system for scalable openflow control", *Rice University, Houston, TX, USA, TSEN Maestro-Techn. Rep, TR10-08*, 2010.

27. *RYU network operating system, Nippon Telegraph and Telephone Corporation*,

http://osrg.github.io/ryu, accessed at December 2017.

28. Yeganeh, S. H., A. Tootoonchian and Y. Ganjali, "On scalability of software-defined networking", *IEEE Communications Magazine*, Vol. 51, No. 2, pp. 136–141, 2013.

29. Dixit, A., F. Hao, S. Mukherjee, T. Lakshman and R. R. Kompella, "ElastiCon; an elastic distributed SDN controller", *Architectures for Networking and Communications Systems (ANCS), 2014 ACM/IEEE Symposium on*, pp. 17–27, IEEE, 2014.

30. Handigol, N., M. Flajslik, S. Seetharaman, N. McKeown and R. Johari, "Aster* x: Load-balancing as a network primitive", *9th GENI Engineering Conference (Plenary)*, pp. 1–2, 2010.

31. Bari, M. F., S. R. Chowdhury, R. Ahmed and R. Boutaba, "PolicyCop: An autonomic QoS policy enforcement framework for software defined networks", *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*, pp. 1–7, IEEE, 2013.

32. Keller, E., S. Ghorbani, M. Caesar and J. Rexford, "Live migration of an entire network (and its hosts)", *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pp. 109–114, ACM, 2012.

33. Hand, R., M. Ton and E. Keller, "Active security", *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 17, ACM, 2013.

34. Shin, S., P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu and M. Tyson, "FRESCO: Modular Composable Security Services for Software-Defined Networks.", *NDSS*, 2013.

35. *Mininet, An Instant Virtual Network on your Laptop (or other PC)*, http://mininet.org/, accessed at December 2017.

36. De Oliveira, R. L. S., A. A. Shinoda, C. M. Schweitzer and L. R. Prete, "Using

mininet for emulation and prototyping software-defined networks", *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pp. 1–6, IEEE, 2014.

37. Lantz, B., B. Heller and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks", *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19, ACM, 2010.

38. Handigol, N., B. Heller, V. Jeyakumar, B. Lantz and N. McKeown, "Reproducible network experiments using container-based emulation", *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pp. 253–264, ACM, 2012.

39. Pan, J. and R. Jain, *A Survey of Network Simulation Tools: Current Status and Future Developments*, Tech. rep., Washington University in St Louis, 2008.

40. Ahrenholz, J., C. Danilov, T. R. Henderson and J. H. Kim, "CORE: A real-time network emulator", *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pp. 1–7, IEEE, 2008.

41. k Jha, R., P. Kharga, I. Z. Bholebawa, S. Satyarthi, S. Kumari *et al.*, "OpenFlow technology: A journey of simulation tools", *International Journal of Computer Network and Information Security*, Vol. 6, No. 11, p. 49, 2014.

42. Parkhill, D., *The Challenge of the Computer Utility*, Addison-Wesley Educational Publishers Inc., US, 1966.

43. Vaquero, L. M., L. Rodero-Merino, J. Caceres and M. Lindner, "A break in the clouds: towards a cloud definition", *ACM SIGCOMM Computer Communication Review*, Vol. 39, No. 1, pp. 50–55, 2008.

44. Azodolmolky, S., P. Wieder and R. Yahyapour, "Cloud computing networking: challenges and opportunities for innovations", *IEEE Communications Magazine*,

Vol. 51, No. 7, pp. 54–62, 2013.

45. Wang, P., W. Huang and C. A. Varela, "Impact of virtual machine granularity on cloud computing workloads performance", *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pp. 393–400, IEEE, 2010.

46. Beloglazov, A., J. Abawajy and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing", *Future generation computer systems*, Vol. 28, No. 5, pp. 755–768, 2012.

47. Kaplan, J. M., W. Forrest and N. Kindler, *Revolutionizing data center energy efficiency*, Tech. rep., McKinsey & Company, 2008.

48. *Data center: Load balancing data center services*, Tech. rep., Cisco Systems, 2004.

49. Wang, H., Y. Li, Y. Zhang and D. Jin, "Virtual machine migration planning in software-defined networks", *IEEE Transactions on Cloud Computing*, 2017.

50. Nasim, R. and A. J. Kassler, "Network-centric Performance Improvement for Live VM Migration", *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pp. 106–113, IEEE, 2015.

51. Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski *et al.*, *Above the clouds: A berkeley view of cloud computing*, Tech. rep., UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

52. Bhardwaj, S., L. Jain and S. Jain, "Cloud computing: A study of infrastructure as a service (IAAS)", *International Journal of engineering and information Technology*, Vol. 2, No. 1, pp. 60–63, 2010.

53. Dinh, H. T., C. Lee, D. Niyato and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches", *Wireless communications and mobile computing*, Vol. 13, No. 18, pp. 1587–1611, 2013.

54. Satyanarayanan, M., P. Bahl, R. Caceres and N. Davies, "The case for vm-based cloudlets in mobile computing", *IEEE pervasive Computing*, Vol. 8, No. 4, 2009.

55. Pang, Z., L. Sun, Z. Wang, E. Tian and S. Yang, "A survey of cloudlet based mobile computing", *Cloud Computing and Big Data (CCBD), 2015 International Conference on*, pp. 268–275, IEEE, 2015.

56. Bonomi, F., R. Milito, J. Zhu and S. Addepalli, "Fog computing and its role in the internet of things", *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

57. Sapienza, M., E. Guardo, M. Cavallo, G. La Torre, G. Leombruno and O. Tomarchio, "Solving critical events through mobile edge computing: An approach for smart cities", *Smart Computing (SMARTCOMP), 2016 IEEE International Conference on*, pp. 1–5, IEEE, 2016.

58. Liu, H., F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy and Y. Zhang, "Mobile Edge Cloud System: Architectures, Challenges, and Approaches", *IEEE Systems Journal*, 2017.

59. *RYU Source Code, Nippon Telegraph and Telephone Corporation*, https://github.com/osrg/ryu, accessed at December 2017.

60. Kingman, J. F. C., *Poisson Processes*, Wiley Online Library, 1993.

61. *Iperf: The TCP/UDP bandwidth measurement tool*, https://iperf.fr/, accessed at December 2017.

62. Viola, P. and M. Jones, "Rapid object detection using a boosted cascade of simple features", *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, Vol. 1, pp. I–I, IEEE, 2001.

63. Lienhart, R. and J. Maydt, "An extended set of haar-like features for rapid object detection", *Image Processing. 2002. Proceedings. 2002 International Conference on*, Vol. 1, pp. I–I, IEEE, 2002.