# A TESTBED FOR SIMULATION-BASED ANALYSIS OF FORWARDING PLANE

**FARAS MOHAN DEWAL**

A Thesis
in
The Department
Of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science (Electrical and Computer Engineering)
at

Concordia University
Montréal, Québec, Canada

July 2016

**CONCORDIA UNIVERSITY**
**SCHOOL OF GRADUATE STUDIES**

This is to certify that the thesis prepared

By: Faras Mohan Dewal

Entitled: "A Testbed for Simulation-based Analysis of Forwarding Plane"

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science**

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

| | |
|---|---|
| ______________________________ | Chair |
| Dr. R. Raut | |
| ______________________________ | Examiner, External |
| Dr. A. Hanna (CSE) | To the Program |
| ______________________________ | Examiner |
| Dr. M. R. Soleymani | |
| ______________________________ | Supervisor |
| Dr. S. Abdi | |

Approved by: ______________________________
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

July 2016

______________________________
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

# ABSTRACT

A Testbed for Simulation-based Analysis of Forwarding Plane

by

Faras Mohan Dewal

Master of Applied Science (Electrical and Computer Engineering)

Concordia University, Montreal QC, 2016

This thesis presents a testbed capable of generating scalable realistic network traffic on a standalone machine. The functionality of the proposed testbed is to model a scalable network of client and server instances and generate network traffic to perform simulation based-analysis of forwarding plane designs. The testbed enables the designer to successfully conduct experiments on the design under test using realistic traffic profiles and assess the performance for multiple use cases.

The proposed testbed defines simulation models for client and server nodes. The testbed modeling has been abstracted to three different levels. First, a base node design allows us to instantiate and manage multiple instances within the node. Second, a transmission protocol is implemented to enable data transfer between client and server instances. The final stage is the Internet application modeling stage. Our experiments show that we are able to reliably generate network traffic for up to 400 client and server instances on a standalone machine.

## ACKNOWLEDGEMENTS

I take this opportunity to express my heartfelt gratitude towards Dr. Samar Abdi for guiding and motivating me throughout my degree program. He helped me through various stages of my project with his constant encouragement, regular assessment and constructive criticism.

I am thankful to all my professors at Concordia for helping me get back on track after a long education gap. I am grateful to Concordia University for providing me with an awesome workplace and the opportunity to pursue my higher education. I would like to thank my examining committee members for providing valuable comments and suggestions to shape this thesis.

I am thankful to my colleagues in the PFPSim team: Shafigh, Umair, Gordon, Eric, and Kamil for their suggestions, discussions and support. I am thankful to have the opportunity to work with Aryan, Sara, Deepak, Samneet and Michael. It has always been a pleasure working and learning with all of you. I would like to thank my friends - Pallavi, Sadhana, Shakul and Hitesh for being the support I can depend on.

I am truly thankful to my mother, father, brothers – Pranshu, Sushant, Aditya and sister-in-law Urvashi (Pinky) for motivating and loving me.

# CONTENTS

## List of Tables

## List of Listings

## List of Figures

## List of Acronyms

| | |
|---|---|
| DNS | Domain Name Server |
| FAD | Forwarding Architecture Description |
| GENI | Global Environment for Network Innovations |
| IPv4/ IP | Internet Protocol version 4 |
| LPM | Longest Prefix Match |
| MAC | Media Access Control |
| NAT | Network Address Translation |
| NPU | Network Processor Unit |
| OS | Operating System |
| P4 | Programming Protocol-Independent Packet Processors |
| PE | Processing Elements |
| PFPSim | Programmable Forwarding Plane Simulator |
| RMT | Reconfigurable Match Tables |
| SDN | Software Defined Networking |
| TCP | Transmission Control Protocol |
| TOS | Type-of-Service |
| TTL | Time-to-live |
| UDP | User Datagram Protocol |
| URL | Uniform Resource Locator |
| VoIP | Voice over Internet Protocol |

# CHAPTER 1

## Introduction

Testbeds are defined as a testing platform for analyzing new designs, algorithms and protocols during the research and development phase of a project. Testbeds are used to conduct experiments that represent real-world scenarios in a controlled environment. A testbed usually consists of definitive hardware components and software applications to create a test environment for conducting isolated, rigorous and replicable test scenarios. The most apparent benefit of using a testbed is the prediction of behavior of a system under extreme conditions without any real world consequences. A well-defined testbed capable of performing reliable analysis is vital for optimizing the performance of a system and in reducing the time-to-market for new technologies. Testbeds can be categorized on the basis of the system being analyzed as hardware testbeds [1] or software testbeds [2] or, on the basis of their implementation as physical testbeds [3] or virtual testbeds [4]. Specific testbeds are used in different disciplines to analyze the consequences of new technologies, theories and projects before their implementation in the real world. For example, the Industrial Internet Consortium [5] currently lists 17 testbeds in active development, for example a factory environment simulation testbed (FOVI), a water management testbed, a High-Speed Network Testbed and more.

In the field of network research, network testbeds are extremely popular and are defined as a combination of hardware and software components of a network. Network testbeds are used extensively by researchers to conduct repeatable, controlled, realistic, configurable, isolated and scalable tests for analyzing protocols and network components. Physical network testbeds

[3, 6] are highly desirable and provide a real-life network with configurable hardware nodes, Internet applications executing on OSs and real OS interaction. The use of physical network testbeds allows the experiments to be realistic, reliable and scalable. However, many factors affect the real-time operation of a network which often leads to unrepeatability of experiments conducted on a physical testbed. It is also important to note that access to these network testbeds is often restricted and sometimes impossible to attain due to various legalities. These restrictions on the physical network testbeds is justified because with an ever-changing Internet model, these testbeds are quite expensive and challenging to establish, manage and upgrade. Mininet [4] provides a quick, inexpensive and easy to use alternative to physical network testbeds. We use Mininet to perform realistic emulation of a virtual network on a single system. However, Mininet is limited in terms of network size and bandwidth by the processing capability of the underlying system.

## 1.1 Motivation

The objective of this thesis is to design a test environment for analyzing simulation models of network processors (NPU) to perform reliable design-space exploration. The input stimulus to an NPU is network traffic. To perform the analysis, we could generate network traffic with characteristics which have been determined using pseudo-random, deterministic or probability distribution functions. However, such traffic profiles will amount to unrealistic analysis of the NPU. To perform realistic analysis of the NPU model, we propose to design a test environment which is capable of generating scalable realistic network traffic on a standalone system. Figure 1 illustrated the proposed testbed. To setup the test environment, we use emulation tools to create a virtual network and generate realistic network traffic. However, by design the emulation tools cannot exceed the CPU capacity or available bandwidth on the system. We perform a study of the emulation tools to formulate our problem statement and establish the design rules for implementing simulation models for a network testbed.

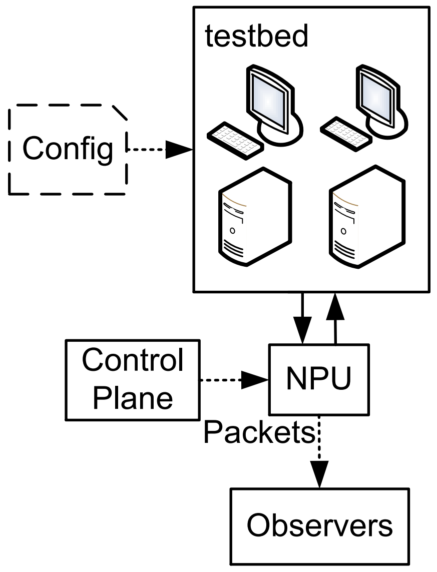


Figure 1: Proposed testbed design

## 1.2 Problem Statement

The scalability of a virtual network using Mininet [4] in terms of non-functional metrics such as required emulation time and memory usage has been discussed in [7, 8]. [9] performs a comparative analysis between Mininet and EstiNET [10]. The authors report an aberrant behavior in the functionality of Mininet but do not provide any conclusive remarks. Our analysis of the virtualized network using Mininet to generate realistic network traffic provides us an insight into the limitations of emulation tools. We summarize our findings below:

1. The performance bottle neck for emulating a large virtual network on a single system is the parallel processing capability of the system. We observe that as the network is scaled, the number of threads increase proportionally which leads to increased thread contention which could amount to aberrant behaviour in the performance of the

network. Table 1 illustrates the proportional increase in number of threads with growing network size.

Table 1: Virtual network - Scalability bottleneck

| Hosts | Threads spawned | Threads increased |
|---|---|---|
| 2 | 47 | - |
| 4 | 59 | 12 |
| 8 | 82 | 23 |
| 16 | 130 | 48 |

2. The link bandwidth limit for the virtual network is determined by the processing capability of the system. To demonstrate the bandwidth limitation on the generated traffic, we performed the iPerf [11] bandwidth test on a single-switch topology virtual network. The switch functionality is defined by the simple_router [12] application provided with the P4 soft-switch compiler. For a two-host network, iPerf reports an available bandwidth of 843 Mbps between the client and server hosts. Next, we scale the virtual network to two clients and two servers. The available bandwidth between the client-server pairs drop down to 373 Mbps and 436 Mbps. We also observe that the maximum available bandwidth varies from experiment to experiment and is affected by the OS processes and user tasks.

3. We note that the network traffic generator models the inter-connection delays and inter packet delays using the sleep functionality. The sleep functionality does not guarantee complete control over the timing and leads to variations in the generated traffic between experiments. Figure 2 illustrates the significant variation between the expected and observed network traffic over a virtual network when the number of hosts are increased.

4. The network traffic generator application depends on the virtual hosts to handle the packet level dynamics for TCP file transfers. The dependence of the traffic generator on the underlying system to handle protocol would lead to variation in generated network traffic between experiments and also between different OSs.

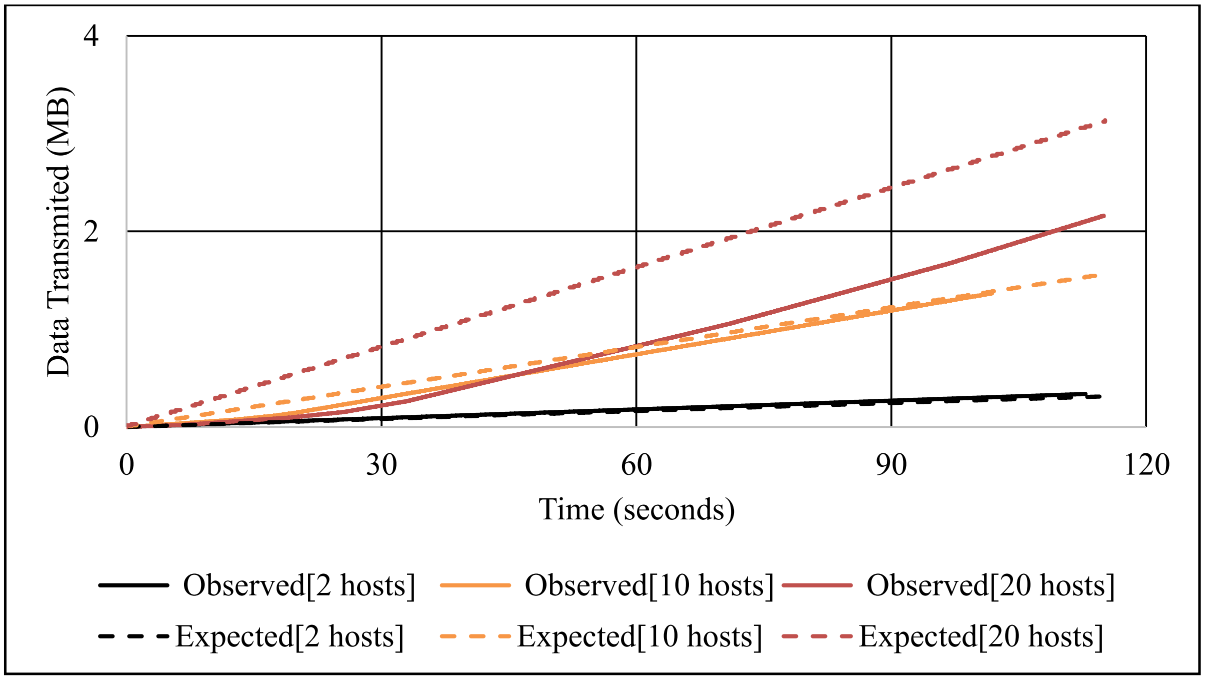


Figure 2: Virtualized network throughput – Expected vs. Observed

## 1.3 Writing Conventions

Throughout this thesis we use the words node, instance, host and system extensively. We wish to associate these words with specific design components in order to facilitate ease of expression. The term node refers to a group of client or server instances. By design, every node in the proposed testbed is configured to simulate multiple identical instances. The client and server instances communicate over the testbed. The term host is used to express the client and server elements of the virtualized network as in, Mininet hosts. Finally, we use the term system to refer the machine which we use to execute the proposed testbed and for testing our designs.

## 1.4 Thesis Contributions

In this thesis, we propose the implementation of a network testbed to perform simulation-based analysis of network processors using realistic network traffic. The significant contributions of this thesis are:

1. An evaluation of network emulation tools to generate network traffic on a standalone machine. We also present a methodology to generate realistic network traffic using well-defined traffic characteristics of Internet applications such as online video streaming and VoIP.

2. Design and implementation of SystemC [13] based abstract simulation models of clients and servers to simulate nodes of a network testbed. We define a connection modeling framework for minimal implementation of network protocols to connect client and server nodes.

3. Implementation of a SDN based load-balancing application in the control plane of a network processor simulator. The proposed testbed is used to perform a comparative analysis between three load balancing algorithms – round robin, static and shortest queue.

4. A comparative analysis of network emulation vs. simulation to validate and assess the proposed simulation models. The simulation models are used to analyze forwarding plane designs using realistic network traffic.

## 1.5 Outline

The rest of the thesis is organized in four chapters. Chapter 2 describes our study to evaluate emulation tools for generating network traffic in a virtualized network. The chapter describes integration of existing tools to emulate a virtual testbed. We perform network virtualization using Mininet, switch emulation using the P4 language and network traffic generation using Harpoon. Next, we discuss the modeling of two Internet applications – online video streaming and VoIP. We conclude our study of the emulation tools with a performance analysis and a scalability analysis of the virtual testbed.

Chapter 3 describes the simulation models for the proposed testbed. We discuss in detail the architecture of the client and server nodes. The architecture of the nodes describes the hardware abstraction of the testbed. Next, we discuss in detail the modeling of client and server nodes, the connection modeling between client-server pairs and implementation of the communication protocol between clients and servers.

In the first half of chapter 4 we discuss the proposed testbed for SDN applications, a forwarding plane design and a control plane design. We implement a load balancer and discuss the architecture and modeling of the control plane. In the second half, we present a comparative analysis of the emulation tools and the proposed testbed. We also perform a scalability analysis of the proposed testbed. Finally, we present the simulation-based analysis of a network processor using realistic network traffic.

Chapter 5 summarizes the thesis with the inferred conclusions and future work for the proposed testbed.

# CHAPTER 2

# Evaluation of Emulation Tools

An emulator is an entity which can be used to imitate a single device or multiple devices. A software emulator is an application which provides additional functionality to a device outside of the expected functionalities. Hardware emulation allows one hardware device to imitate the behavior of another device. Virtualization tools [4, 14] can be used to emulate networks on a single laptop and provide the combined functionality of software and hardware emulators. To setup a virtual network, Oracle's VirtualBox [14] allows execution of multiple operating systems as applications. The operating systems run as applications on a virtualized hardware and are generally known as Virtual Machines (VM). A major drawback of using VMs to emulate a virtual network is the high system requirements in terms of memory and disk storage. Mininet [4] provides a light-weight process-based alternative to emulate a virtual network. To mimic the behavior of Internet applications, a network traffic generator, Harpoon [15] is used. To emulate Internet applications, we study the traffic characteristics for online video streaming and VoIP. We use the derived traffic profiles to configure Harpoon and generate realistic network traffic in the virtual network. Finally, we provide an in-depth analysis of the integrated emulation tools and evaluate their performance.

## 2.1 Network Virtualization

In this study, we use Mininet to create a virtualized network on a standalone system. Mininet uses process-based virtualization to emulate multiple hosts, links and switches. To understand how Mininet works we need to understand network namespaces. Upon installation, the Linux OS has a single set of routing table entries and network interfaces which are shared among all the processes executing on the system. Using network namespaces, the OS allows the user to create multiple routing tables and multiple virtual instances of the network interfaces. These routing tables and virtual network interfaces operate in isolation from each other within their namespaces. The processes executing within a network namespace have unique access to the virtualized interface and routing tables. The virtualized interfaces on a system can be connected to each other using software switches [16].

To help understand this concept, Listing 1 shows the simplicity of creating a virtual network using network namespaces on a single OS kernel. Line 1 creates two network namespaces called abc and xyz. These two namespaces will constitute the virtual hosts in the emulator. Line 2 creates a virtual switch with the name sw1. Next, in lines 3 and 4 we create two virtual Ethernet (veth) pipes and assign the ends of the pipes to four ports – a-eth0, s1-etha and x-eth0 and s1-ethx. In line 5, the port a-eth0 is assigned to the network namespace abc. Line 6 allocates the port x-eth0 to the xyz network namespace. In lines 7 and 8, we add the ports s1-etha and s1-ethx to the virtual switch – sw1. Once the veth pipes have been assigned we activate the ports at the end of the pipes. Using lines 9 and 10 we activate the ports assigned to the switch sw1. Using lines 11 and 13 we execute the command to activate the ports within the abc and xyz network namespaces. To make the network namespaces fully functional we also need to setup and activate local loop-back interfaces in both the namespaces. This is performed using the commands in line 12 and 14. Once the veth pairs have been assigned and the ports have been activated we assign network IP addresses to the Ethernet ports of the created network namespaces abc and xyz. Finally, we can check the connectivity between the namespaces by executing a ping command from one namespace to the IP address defined in the second namespace. Figure 3 shows the virtual network created using the commands discussed in Listing 1.

Listing 1: Process-based virtualization

```
~# ip netns add abc && ip netns add xyz
~# ovs-vsctl add-br sw1
~# ip link add a-eth0 type veth peer name s1-etha
~# ip link add x-eth0 type veth peer name s1-ethx
~# ip link set a-eth0 netns abc
~# ip link set x-eth0 netns xyz
~# ovs-vsctl add-port sw1 s1-etha
~# ovs-vsctl add-port sw1 s1-ethx
~# ip link set s1-etha up
~# ip link set s1-ethx up
~# ip netns exec abc ip link set dev lo up
~# ip netns exec abc ip link set dev a-eth0 up
~# ip netns exec xyz ip link set dev lo up
~# ip netns exec xyz ip link set dev x-eth0 up
~# ip netns exec abc ip address add 10.0.0.1/30 dev a-eth0
~# ip netns exec xyz ip address add 10.0.0.2/30 dev x-eth0
~# ip netns exec abc ping 10.0.0.2 –c 10
```

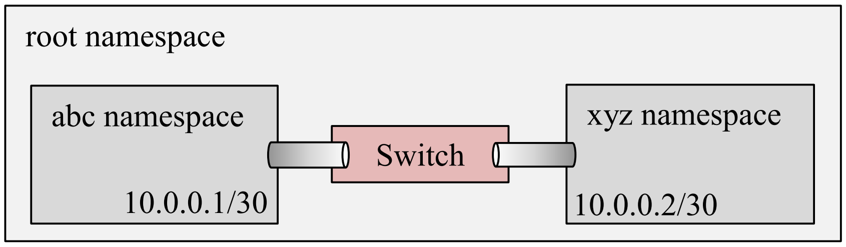


Figure 3: A network example using process-based virtualization

It is obvious that creating a large virtual network with hundreds of nodes on a single system using the above example will be difficult to manage and update. Mininet simplifies the process extensively. The above example can be performed in Mininet using the code provided in Listing 2.

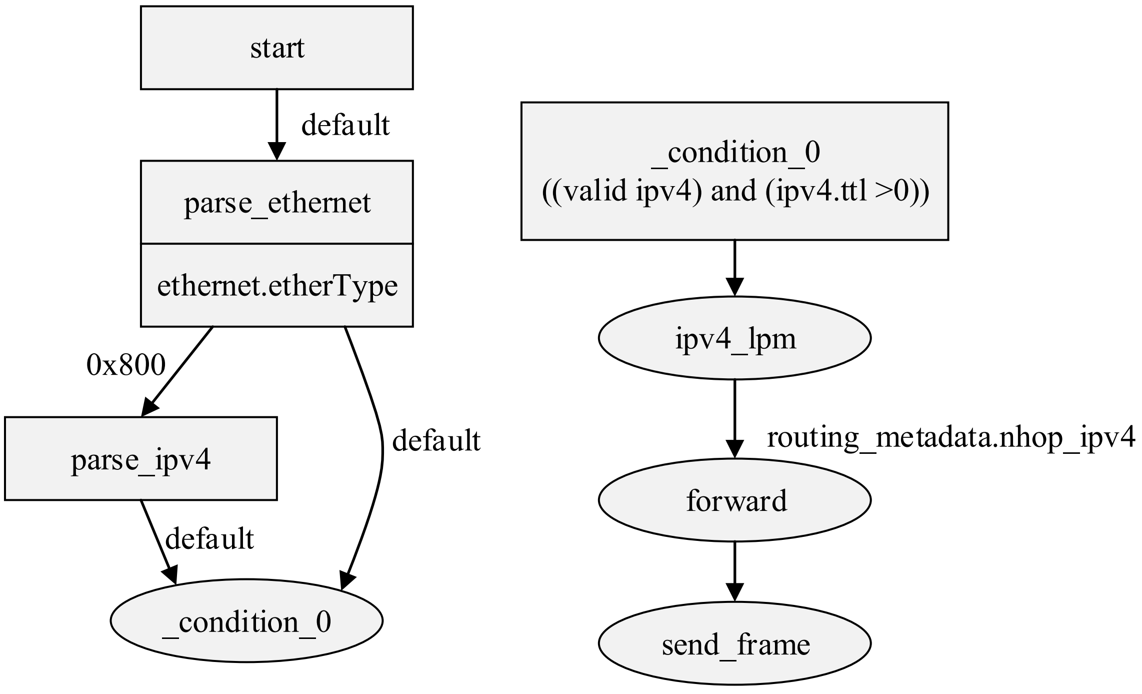
Listing 2: Network virtualization using Mininet

```
1.    ~# mn --switch ovsk --topo single,2
2.    mininet> pingall
3.    mininet> h1 ifconfig
4.    mininet> h2 ifconfig
5.    mininet> xterm h1 h2
```

Line 1 creates a virtual single-switch topology based network of two hosts and an Open vSwitch. Once the network is created a “pingall” command within the Mininet prompt will test connectivity between all the hosts. To obtain the interface configurations for the virtual hosts we can execute the “ifconfig” command on the hosts as illustrated in Lines 3 and 4. The terminal emulators for the virtual hosts within the network can be obtained using the “xterm” command as shown in Line 5. The terminal emulators can be used to simultaneously execute commands on both the hosts, e.g. iPerf client and server. Mininet also provides an extensible python API to create virtual networks and execute commands on the virtual hosts. We use a Mininet python API script [17] in our study of the emulation tools to setup the virtual network.

## 2.2 Switch Emulation

By default, Mininet implements the Open vSwitch [16] for connecting the virtual hosts across the emulated virtual network. We replace the Open vSwitch default implementation by the simple_router application [12] implemented using the P4 language [18]. The advantage of using a P4 defined router is an improved insight into the functionality of the router. P4 allows the programmer to specify packet processing logic by defining actions based on the packet header fields. The P4 router implements match-action tables to store the matching packet header field values and corresponding actions. The simple_router functionality has been illustrated in Figure 4. Every incoming packet goes through the parser and the router parses the Ethernet and IPv4 headers. Upon parsing the IPv4 header the router extracts the time-to-live count and the required IP header fields, in this case the destination IP address, for matching the table entries. If the IPv4 header is valid and the time-to-live count is greater than zero, the router will perform a longest prefix match on the destination IP address of the packet and assign an egress port number. Next, the router will perform the forward and send_frame table operations on the packet and update the source and destination MAC addresses.



(a) Parser (b) Forwarding Plane Tables

Figure 4: Simple router functionality

The simple_router is configured using P4 specified semantics. Listing 3 shows a set of sample commands used to configure the simple_router implementation of the P4 router within a network. The ipv4_lpm table performs a longest prefix match on the destination IPv4 address of an incoming packet. The router will assign egress port 1 to a packet whose destination address is 10.1.0.0. The router will assign egress port 2 to the packet whose destination address is either 11.1.0.0 or 11.1.0.1. The MAC addresses of the packets will also be updated based on the forward and send_frame table configurations.

Listing 3: P4 configuration

```
#ipv4_lpm
10.1.0.0/32     set_nhop 11.1.0.0     1
11.1.0.0/31     set_nhop 10.1.0.0     2
#forward
10.1.0.0 set_dmac FF:FF:FF:FF:FF:01
11.1.0.0 set_dmac BB:BB:BB:BB:BB:01
#send_frame
1'2 rewrite_mac AA:AA:AA:AA:AA:01
2'2 rewrite_mac AA:AA:AA:AA:AA:02
```

## 2.3 Network traffic generator

To emulate client and server behavior within the Mininet hosts, we use a network traffic generator, Harpoon [15], to generate packets in our virtual environment. Harpoon is a flow-level traffic generator. To identify packets within a flow the packet header fields are compared at an IP header level to determine uniqueness of the packets sent between a source and destination. Harpoon can be configured to generate TCP and UDP packets over IPv4 in a client-server network. Listing 4 and Listing 5 illustrate the server and client configuration files for the Harpoon traffic generator.

Listing 4: Harpoon server configuration

```
<plugin name="se" objfile="tcp_plugin.so" maxthreads="1", personality="server">
    <active_sessions>
        7
    </active_sessions>
    <file_sizes>
        200 4000 3080 100 500
    </file_sizes>
    <address_pool name="server_address">
        <address ipv4='0.0.0.0' port='56000'/>
    </address_pool>
</plugin>
```

Listing 5: Harpoon client configuration

```
<plugin name="cl" objfile="tcp_plugin.so" maxthreads="1", personality="client">
    <active_sessions>
        5 6 1 7
    </active_sessions>
    <interconnection_times>
        1 8.5 0.5 0.7 1.9
    </interconnection_times>
    <address_pool name="client_source_addresses">
        <address ipv4='10.1.0.1/32' port='0' />
    </address_pool>
    <address_pool name="client_destination_addresses">
        <address ipv4='11.1.9.0/24' port='56000'/>
    </address_pool>
</plugin>
```

The architecture of the Harpoon traffic generator consists of a connection level model and a session level model. At the connection level, the model has two parameters – the size of the file being transferred and the inter-connection delay between the file transfers. The file size and inter-connection delay parameters are used to describe the network traffic characteristics and mimic Internet applications. At the session level, the traffic generator defines the number of active sessions and the IP spatial distribution. However, the spatial diversity in the generated traffic is primarily governed by the IP configuration of the hosts in the virtual network. The active sessions parameter specifies the number of client-server pairs active during an interval of the emulation. The file sizes in the server configuration file are specified in bytes and the interconnection times for the clients are specified in seconds.

## 2.4 Internet Applications Modeling

Once we have established a virtual network, we need to define the behavior of the virtual hosts to generate realistic network traffic. Realistic network traffic can be defined simply as the network traffic generated by Internet applications such as online video streaming, remote login, online gaming, VoIP, HTTP and other applications. In this section we define two applications whose traffic characteristics have been modeled to generate Internet traffic. The first application is YouTube's online video streaming which is implemented using TCP client and server nodes. The second application defined is VoIP and is implemented using UDP client and server nodes.

### 2.4.1 Online Video Streaming

The Visual Networking Index at Cisco [19] has forecasted that by year 2020 video traffic will form more than 80 percent of all the consumer Internet traffic. To model the traffic characteristics of online video streaming, Rao et. al [20] provide the network traffic analysis of YouTube and Netflix. Due to a more in-depth analysis, we use the YouTube traffic characteristics for modeling online video streaming traffic. The YouTube video streaming traffic has been characterized to have two stages – buffering and steady state. The traffic characteristic for the buffering stage is described as an initial burst of data to accumulate content on the client. This allows the client to maintain a buffer against possible future bandwidth loss and maintain uninterrupted streaming. Once sufficient data has been buffered at the client side, the servers may throttle down the rate of data transfer. This region of video streaming is called the steady state phase. The steady state phase analysis of YouTube traffic can be classified based on block/ file sizes and inter-connection delays between blocks/ files. The steady state phase analysis of YouTube traffic is classified into four categories – short on-off cycles, long on-off cycles, combination of short and long on-off cycles and, no on-off cycles. Figure 5 [20] illustrates the buffering and steady state phases of YouTube's video streaming traffic.

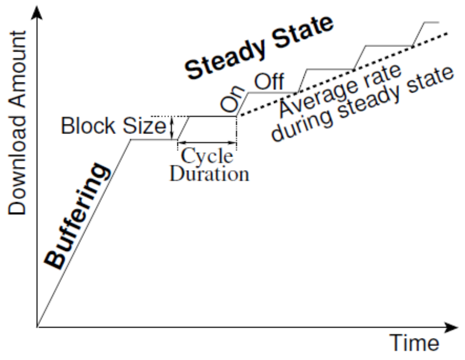


Figure 5: YouTube traffic characteristics [20]

The data transferred during the buffering and steady state phases is determined using an accumulation ratio which is defined in Listing 6.

Listing 6: Calculate accumulation ratio

$$\text{Accumulation ratio} = \frac{\text{Average download rate}}{\text{Video encoding rate}}$$

An accumulation ratio of at least one is desirable for a video streaming application. A value of accumulation ratio less than one would result in empty buffers and cause interruptions in video rendering. We use the presented traffic characteristics to model YouTube clients and servers in our virtual network to generate realistic network traffic. The streaming strategy for YouTube videos is attributed to the type of client and video application connecting with the servers. The traffic characteristics presented by the authors [20] have been interpreted and summarized in Table 2.

Table 2: YouTube traffic characteristics

| Strategy | YouTube Client | Minimum block delay (s) | Maximum block delay (s) | Minimum block size (kB) | Maximum block size(kB) |
|---|---|---|---|---|---|
| Short on-off (buffering) | Flash | 0.32 | 2.41 | 1,024 | 8,192 |
| | HTML5 | 0.77 | 9.66 | 10,240 | 15,360 |
| Short on-off (steady-state) | Flash | 0.32 | 2.41 | 64 | 64 |
| | HTML5 | 0.77 | 9.66 | 256 | 256 |
| Long on-off (buffering) | Chrome | 60 | 80 | 10,240 | 15 360 |
| | Android | | | 4,096 | 8,192 |
| Long on-off (steady-state) | Chrome | | | NA | 2,560 |
| | Android | | | | 2,560 |

The short on-off strategy is characterized by a buffering stage of 1 MB to 15 MB of data depending upon the client application. The analysis of steady state phase during the data transfer is defined chiefly by blocks of size either 64 kilobytes or 256 kilobytes. The accumulation ratio for the short on-off strategy was modeled at 1.04. We estimate an inter-block delay of 0.32 seconds to 9.66 seconds for maintaining the required accumulation ratio at the client. The long on-off strategy is characterized by a buffering stage of 4 MB to 15 MB of data. The steady state phase is characterized with block sizes greater than 2.5 MB and inter-block delay between 60 seconds and 80 seconds.

For modelling YouTube application, we configure Harpoon servers with the buffering and steady state block sizes. The Harpoon clients are configured with the inter-block delays. Thus, by adjusting the configurations of the block sizes and inter-block delays we can emulate YouTube's video streaming for short On-Off and long On-Off strategies.

### 2.4.2 Voice over Internet Protocol

Increased flexibility and decrease in cost has led to a rise in the number of VoIP solutions available to customers. With the smartphone industry dominating the technology market, mobile applications such as WeChat, Whatsapp and Viber have introduced VoIP for the public. Facebook and skype also use VoIP in their messaging applications. In the analysis of Internet traffic, [21] have concluded that fractal properties of traffic characteristics for application such as VoIP can pass over to other traffic flows due to the adaptive nature of TCP traffic. To model the traffic characteristics of a UDP based Internet application, we choose VoIP as the second Internet application for modeling the behavior of client and servers. Dang et. al [22] perform analysis of VoIP traffic at the call and packet levels and present the traffic characteristics. The VoIP traffic can be grouped under two categories depending on the codecs – a constant bit rate traffic stream (G.711) and an on-off traffic flow generated by silence compression codecs such as G.723, G.729 and GSMFR. The authors have provided in-depth analysis of VoIP traffic characteristics by silence compression codecs due to their higher prominence in real-world applications than constant bit rate codecs. In our modelling, we utilize the packet level analysis of VoIP traffic. The VoIP traffic volume is determined by the number of client-server pairs in the virtual network. The packet level analysis of VoIP traffic characterizes the On and Off lengths of packet transmissions using the generalized Pareto distribution (GPD) with shape and scale parameters (-0.28, 1.7) for On lengths and (-0.35, 1.02) for Off lengths. The parameter values have been reported similar for the silence compression codecs. However, different codecs have different data rates and payload length specifications for transmitting VoIP traffic. To generate generalized Pareto random numbers, we can use inversion of the cumulative distribution function for GPD. The formula is shown in Listing 7. We use the formula to calculate the on and off durations for configuration of the traffic generator. The off durations are used as inter-file delays to configure the clients. To configure the file sizes at the server, we use the on duration and the packet characteristics of a VoIP codec, for example GSMFR.

The GSMFR codec generates packets with 20ms VoIP payloads (33 bytes) at a bit rate of 13.2 kbps during the On lengths. The G.729 codec has two payload options – 20ms (20 bytes) and 30ms (30 bytes). The bit rate of transmission for both options is 8 kbps. Thus, to model a VoIP server we can calculate the maximum transmission unit as illustrated in Listing 8.

Also, the file sizes for configuring the server can be determined as illustrated in Listing 9.

Listing 7: Generate generalized Pareto random numbers

For uniformly distributed values of U ε (0, 1]

$$\mathbf{X = \mu + \frac{\sigma (U^{-\zeta} - 1)}{\zeta}} \sim \text{GPD } (\mu, \sigma, \zeta \mathrel{!}= 0)$$

X = generated random number

μ = location parameter (= 0, in our calculations)

σ = shape parameter

ζ = negative of scale parameter (-k)

Listing 8: Maximum transmission unit (GSMFR)

MTU = IPv4 (20 B) + UDP (8 B) + RTP (12 B) + Payload (33 B) = 68 B

Listing 9: VoIP server file size (GSMFR)

File size (B) = On duration (**X** seconds) * Codec bit rate (13,516.8 Bps)

The off duration is modeled by the clients using a sleep functionality and the on duration is the time consumed by the server to transfer the determined file size at the codec's bit rate of transfer.

## 2.5 Network Emulation Setup

The integrated emulation tools setup to emulate a virtual network is shown in Figure 6. The network emulation uses the following tools and languages to obtain the desired functionality:

1. Mininet – Network virtualization [4]
2. Harpoon – Generating application specific network traffic [15]
3. P4 – Programming the software switch used in the Mininet network [18]

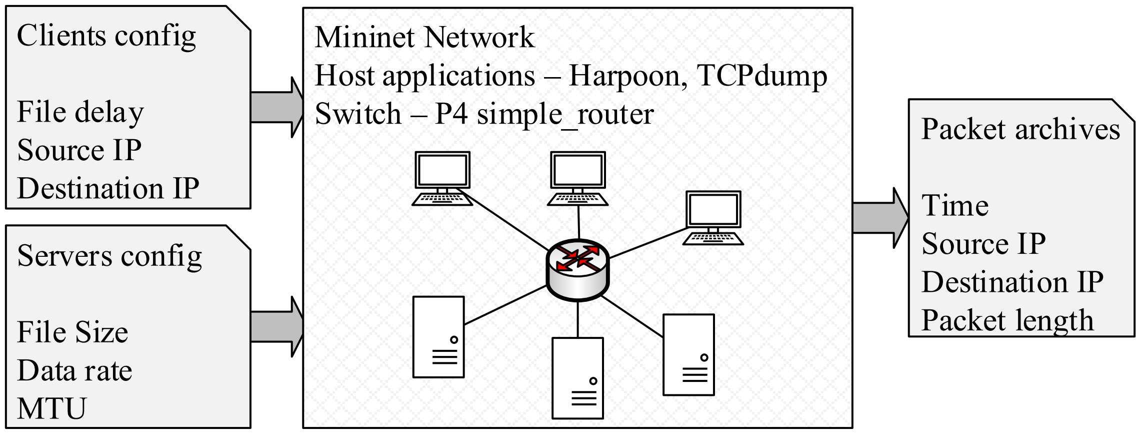


Figure 6: Network emulation architecture

To generate network traffic in the virtual network, the first step is to generate the Harpoon client and server files. Next, we setup the virtual network using Mininet's python API [17] and execute Harpoon on the virtual hosts to generate network traffic. A packet archive tool, TCPDump [23] is used to archive the network traffic generated by the hosts in the virtual network.

Once the traffic archive is available, we can replay the captured packets and translate physical time to SystemC logical time for analysis of the NPU simulation model. Listing 10 illustrates this concept. A pcap traffic archive file assigns a pcap header for every packet captured. The pcap header contains the following metadata regarding a captured packet:

1. The capture time of the packet
2. The actual length of the packet (wire length)
3. The captured packet length

While simulating the network traffic for the NPU model, the physical time of the captured packet is translated to logical time using the SystemC wait functionality. Once the appropriate logical time has lapsed, the next packet is sent to the ingress port of the switch model.

Listing 10: Replay PCAP files in SystemC

```
handle = pcap_open_offline (archive.pcap)
Loop: packet, pcap_header = pcap_next (handle)
    packet_time = pcap_header.time_stamp
    SystemC_wait (packet_time)
    model_ingress.send(packet)
```

We note that this methodology for simulating the NPU model cannot be unified. The simulation needs to be performed in two phases:

1. Generation of network traffic to obtain an archive file.
2. Simulation of NPU model by translating physical time to logical time.

The packet archive files are generally huge and difficult to manage and process. Also, due to the dependence of the virtualization tools on the underlying system, the experiments are not repeatable. We provide a complete list of the issues faced by us in using the emulation tools in the Summary (Section 2.7).

## 2.6 Experimental Results

In our study of emulation tools, we discussed the integration of Mininet and Harpoon to generate realistic network traffic. The described network emulation setup is easy to implement, execute and manage. In this section we perform a scalability analysis of the network emulation setup in terms of network size and performance. We also analyze the emulation tools to determine the performance bottleneck of a virtual network. To isolate the network emulation setup from OS tasks we execute the proposed network emulation setup on isolated core of an Intel i5 2.30GHz processor with 4 GB RAM running Ubuntu 14.04.

### 2.6.1 Scalability Analysis

We perform the scalability analysis of the network emulation setup using TCP traffic. The expected values for the network traffic generated are summarized in Table 3. Figure 7 illustrates the client requests issued and generated traffic volume when we increase the network size.

Table 3: Configuration and expected values

| Property | Configured/ Expected |
|---|---|
| Simulation time | 60 seconds |
| Inter-connection time | 0.5 seconds |
| File size | 1024 bytes |
| File metadata | 5 bytes |
| File requests per client | 120 |
| Data transfer per client-server pair | 123,480 bytes |

During our analysis, we increase the number of client-server pairs (hosts) in the virtual network till we observe a significant variation between the observed and expected behaviors. In the ideal case, we expect to see a proportional increase in the number of files requests and volume of traffic generated with increase in the number of client-server pairs. However, we observe a significant variation between the observed and expected outputs for a network with more than six hosts. The total data transferred is proportional to the number of file requests.

This is expected because we do not model packet loss in our network environment or within the P4 router. To understand this aberrant behavior of the network emulation setup we observe that the number of threads spawned for the network emulation setup increase proportionally as we increase the network size. Figure 8 shows the number of threads spawned by the network emulation setup on the isolated system core.

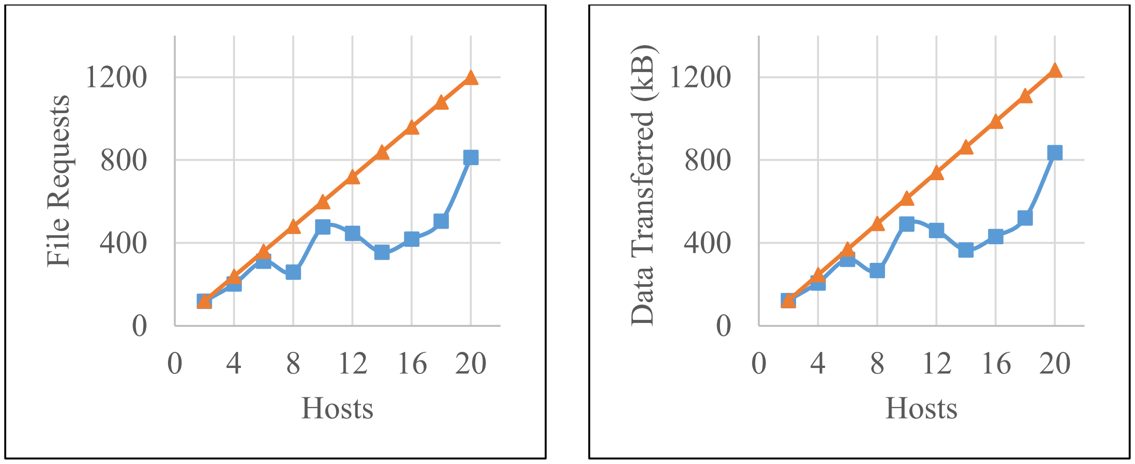


Figure 7: Expected △ vs. Observed □

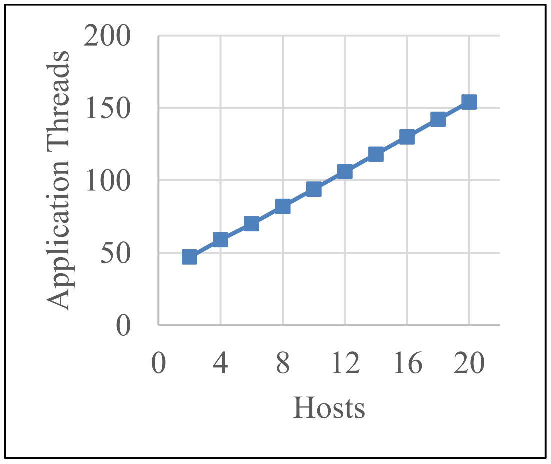


Figure 8: Emulation scalability – Application threads

We note that the emulation tools work in real time and consequently we observe that the number of application threads spawned for network emulation setup increase proportionally to the number of hosts. This proportional increase in the number of threads to be executed results in higher contention amongst threads during emulation. The thread contention results in lower than expected number of files requests and traffic volume. We conclude that the proposed emulation setup is scalable reliably only up to six hosts for the emulation setup described using Harpoon and TCPDump and for the given traffic profile.

### 2.6.2 Emulation Bottleneck – Bandwidth

In this section we determine the limitations of a virtualized network in terms of maximum bandwidth available at the ingress port of a switch in a single-switch topology. To obtain the link bandwidth(s) for the virtual network we execute iPerf [11] on the Mininet hosts. Table 4 lists the average bandwidth between hosts in the virtual network. For a two host network, we observe that the maximum bandwidth is limited to 843 Mbps. Upon increasing the network size, the individual link bandwidth in the network keeps on decreasing proportionally to the number of hosts in the network. This behavior shows that the total available network bandwidth is limited by the processing capability of the underlying host.

Table 4: Bandwidth analysis of a virtual network

| Hosts | Data Transferred (MB) | Average link bandwidths (Mbps) |
|---|---|---|
| 2 | 1006 | 843.0 |
| 4 | 962 | 399.5 |
| 6 | 918 | 254.3 |
| 8 | 899 | 188.5 |
| 10 | 886 | 148.4 |
| 12 | 888 | 123.8 |
| 14 | 873 | 104.4 |
| 16 | 874 | 91.2 |

## 2.7 Summary

In our study of network emulation tools, we were able to emulate an entire network on a standalone system. We also devised a methodology to generate realistic network traffic using a traffic generator in the virtual network. However, the analysis of emulation tools did not yield promising results. Our learnings from the evaluation of emulation tools are summarized below:

1. The Mininet virtual network is limited by the underlying system and cannot exceed the CPU or bandwidth available on the system. The link bandwidth of a virtualized network is limited by the system processing capability while the network size depends on the parallel processing capability of the system.
2. In our analysis of the network traffic generator, we note that the application uses sleep functionality to model inter-connection and inter-packet delays. Thus, the minimum configurable delay is limited by the system clock. Also, the sleep functionality does not guarantee the wakeup time of a thread. Hence, the traffic generator application does not provide a clean implementation of the configured inter-connection delay and data rate.
3. The execution time is not scalable. The network emulation tools work at real-time. As a consequence, to emulate one day of traffic we need to execute the network emulator for the entire duration.
4. The emulation results are not repeatable because the performance of emulation tools is affected by OS tasks running on the host system and user processes.

# CHAPTER 3

# Simulation Models for a Network Testbed

Simulation can be defined as the execution of the logical model of a physical system. While the emulator mimics the outside behavior of a device, a simulator models the underlying state of the device being analyzed. In this chapter we describe SystemC based simulation models for the nodes of a network testbed. The network nodes have been categorized as client nodes and server nodes. In the subsequent sections we provide a detailed description of the architecture and modeling of the nodes, connection modeling and implementation of network protocols within the proposed network testbed.

## 3.1 Architecture

The proposed testbed defines abstract simulation model for client and server nodes. The testbed has been designed to simulate a typical client-server architecture. Figure 9 illustrates the testbed architecture.

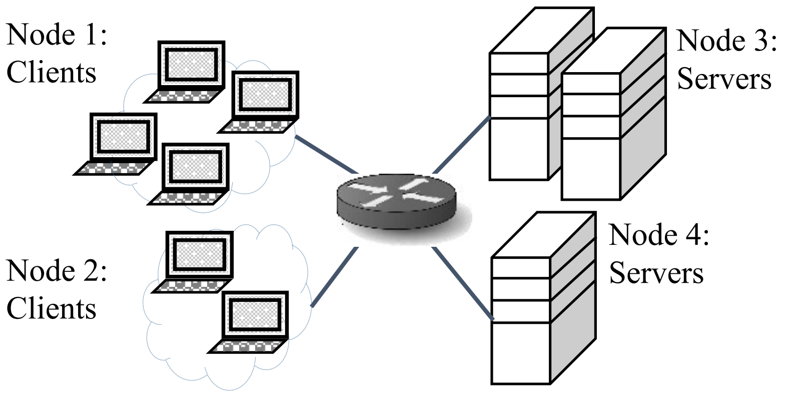


Figure 9: Testbed architecture

The client and server nodes are synchronized using a request-response model. Every client and server node is capable of spawning multiple identical instances. Client instances issue request for a server instance allocation, connection setup, file transfer and connection teardown. The client and server nodes communicating during the simulation should have well defined and synchronized communication protocols to enable meaningful traffic generation. The architecture allows for multiple client and server nodes to connect with each other dynamically at runtime. However, the designer must ensure that IP addresses for client and server nodes should always be unique within the network testbed.

### 3.1.1 Client Node

The client node is responsible for managing all instances defined for the node. Figure 10 illustrates the client node architecture.

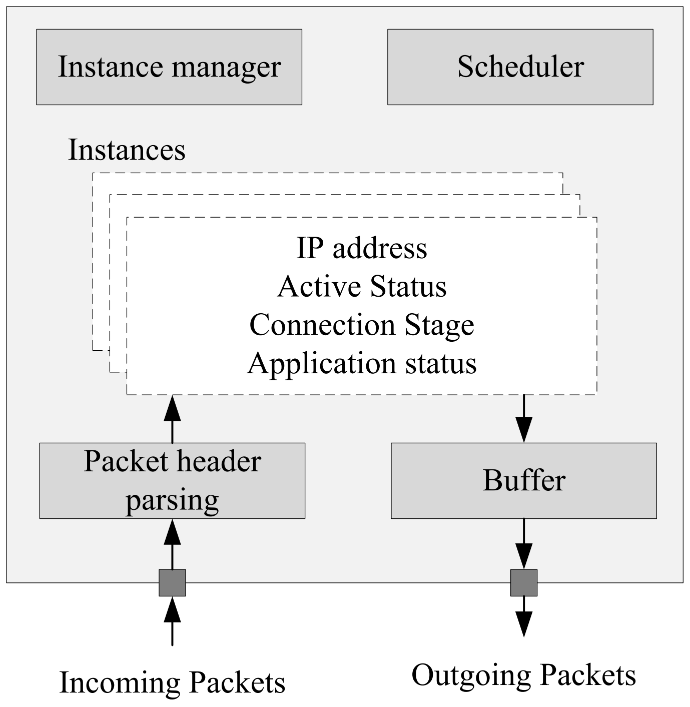
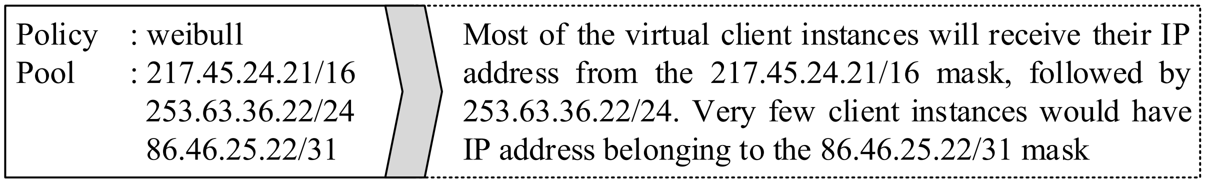


Figure 10: Client node architecture

The clients are instantiated at the start of the simulation and allocated static IP addresses. The instance manager and the scheduler are together responsible for activating inactive client instances at the appropriate time during the simulation. The connection stage is used to determine the behavior of the instance based on the defined protocol. The application status defines the file size to be transferred and the inter-connection delay to be processed for an ongoing connection. Once we have implemented a client node for a desired protocol, the node can be configured to generate network traffic. The configuration parameters of a client node have been grouped into three categories – node, connection delays and packet fields. Listing 11 illustrates a typical client node configuration. The configuration file specifies six node parameters, four connection parameters and five packet header fields. The number of clients instantiated by the instance manager is defined by the "instances" field. The "simulation time" specifies the total logical time for which the client node will be participating in the simulation. We can also obtain a pcap log of all packets sent out from a node by setting the "archive" parameter. The "IP address value" field specifies the prefixes used to allocate IPv4 addresses to the client instances. The corresponding "policy" field is used to specify the access pattern for the specified prefixes. Figure 11 illustrates the use of values and policy fields for the client and server node configuration files. The use of a Weibull policy will ensure that most of the clients receive their IP addresses from the first prefix (217.45.24.21/16) and prefix (86.46.25.22/31) is used minimally. The scale and shape parameters of the Weibull distribution are also configurable. For Weibull distribution, the first specified parameter is shape and the second is scale of the distribution curve. Distribution parameters for Log-normal, Normal and Uniform distributions are calculated by the tool based on the available values. Table 5 provides a list of all available policies for configuration of both client and server nodes.

Listing 11: Client node configuration

| | | | |
|---|---|---|---|
| NODE | | | |
| Type | client | | |
| Instances | 20 | | |
| Archive | false | | |
| Simulation Time | 20 SEC | | |
| IP Address | | | |
| Pool | 217.45.24.21/16 | | |
| | 253.63.36.22/24 | | |
| | 86.46.25.22/31 | | |
| Policy | weibull 1 1 | | |
| Server URL | www.youtube.com/short | | |
| CONNECTION DELAYS | | | |
| Timeout | 10 MS | | |
| Values | 1.41 | 1.08 | 1.34 |
| | 0.63 | 2.18 | 0.76 |
| Unit | SEC | | |
| Policy | uniform | | |
| PACKET FIELDS | | | |
| TOS | 0 | | |
| TTL | 64 | | |
| Source port | 51324 | | |
| Destination port | 443 | | |
| Headers | ethernet_t | | |
| | ipv4_t | | |
| | tcp_t | | |


Policy : weibull
Pool : 217.45.24.21/16
253.63.36.22/24
86.46.25.22/31

Most of the virtual client instances will receive their IP address from the 217.45.24.21/16 mask, followed by 253.63.36.22/24. Very few client instances would have IP address belonging to the 86.46.25.22/31 mask

Figure 11: Distribution policy for accessing values

Table 5: Available distributions

| Distribution | Parameters | Default |
|---|---|---|
| Binomial | Success probability | 0.5 |
| Exponential | Rate parameter (λ) | 1 |
| Geometric | Success probability | 0.5 |
| Log-normal | Mean(μ), Standard deviation(σ) | - |
| Normal | Mean(μ), Standard deviation(σ) | - |
| Poisson | Mean(μ) | 4 |
| Random | - | - |
| Round Robin | - | - |
| Uniform | Maximum, Minimum | - |
| Weibull | Shape, Scale | 1, 1 |

The connection parameters also specify a “timeout” value which configures the maximum time to establish a connection with the server. In case of a timeout, the scheduler will retry to establish the client connection again at a later logical time. The “connection delay values” specifies the inter-connection delay. During the modeling of the connection delay a client instance is kept in an inactive state. The configuration file also lists IP and TCP header fields which have a significant impact on the routing of packets and flows, namely, type of service, time-to-live and, the source and destination ports. The type of service and time-to-live specify the priority of packets generated by the node and the lifetime of the packet in the network respectively. The source and destination ports can be used to represent Internet application being modeled. For example, all DNS requests sent by the client to the controller should have the destination port number 53 which is by standard reserved for DNS. Currently the test bed supports two packet header patterns: “ethernet_t ipv4_t tcp_t” and “ethernet_t ipv4_t udp_t”.

The client connects to the server node which publishes itself to the control plane as the owner of the specified “server URL”. To obtain the IP address of the server node, the client instance sends a DNS query to the load balancer module configured in the testbed environment. The DNS queries are created by the client instances during the simulation as required. As of now, all DNS requests are transmitted over UDP. This works well for our test cases as the main focus of using a DNS service is to implement a DNS-based load balancer on the control plane. For the client node configuration file, the following time units have been defined – SEC (second), MS (millisecond), US (microsecond), NS (nanosecond), PS (picosecond) or FS (femtosecond). These units can be used for the simulation time, timeout and connection delay values.

### 3.1.2 Server Node

The server nodes are responsive by nature and respond to client stimulus. Figure 12 illustrates the server node architecture. The sessions manager creates new server instances dynamically at run time if required during the simulation. The data rate manager manages the inter-packet delay for all the server sessions. The application status corresponding to an ongoing connection consists of file size remaining to be transferred, the idle time to be lapsed before the next packet is sent out and the header information of the packet last received for the server instance. After defining the server side protocol implementation, the server node is configured to define the testbed architecture and the traffic profile. Listing 12 depicts the configuration file for a typical server node. The server node configuration has some parameters identical in their interpretation to the client node configuration – type, archive, IP address, policy and headers.

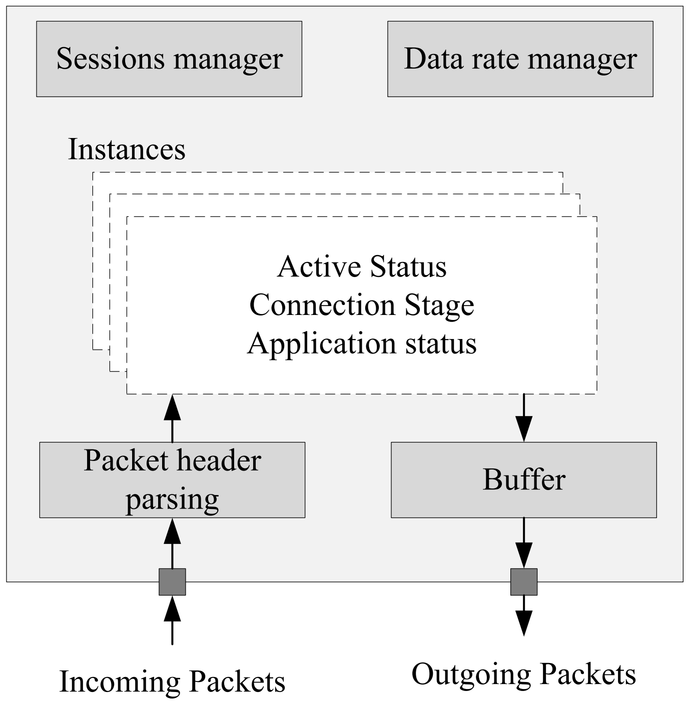


Figure 12: Server node architecture

Within the server node, the server instances are created at runtime by the sessions manager. The "sessions" parameter defines the maximum number of sessions that a server instance can service. The "threshold" defines the optimum load percentage for an instance. If the existing server instances reach the optimum work load, a new server instance will be created and allocated an IP address by the sessions manager. Any new connection to an instance beyond the maximum sessions will be dropped.

Listing 12: Server node configuration

| | | |
|---|---|---|
| NODE | | |
| Type | | server |
| Archive | | false |
| IP Address | | |
| | Pool | 32.64.22.64/14<br>83.35.22.53/20<br>91.10.46.48/20<br>188.76.43.23/26 |
| | Policy | random |
| Server URL | | www.youtube.com/short |
| INSTANCE | | |
| Sessions | | 7 |
| Threshold | | 0.75 |
| MTU | | 1500 B |
| Data rate | | 500 kbps |
| SESSION DELAYS | | |
| Values | | 0.5 |
| Unit | | MS |
| Policy | | round_robin |
| FILE SIZES | | |
| Values | | 64 64 64<br>64 7550 1088 |
| Unit | | kB |
| Policy | | exponential |
| PACKET FIELDS | | |
| Headers | | ethernet_t<br>ipv4_t<br>tcp_t |

The “MTU” parameter defines the maximum IP packet size to be generated by the server instance. All the files are segmented by the server according to the “MTU” parameter before transmission. The node is configured for the maximum rate of transmission in a session using the “data rate” parameter. The data rate is also affected by the number of concurrent active sessions on the server instance. The delay due to multiple connections on a server instance is modeled by the “session delays” parameter. If a server instance has more than one connection, for every new active connection the inter packet delay is increased by value derived from the session delays parameter. The “file sizes” parameter defines the size of files to be transmitted for a single client request. For session delays, the time units are identical to connection delay unit for a client configuration. For the ‘sizeUnit’ and ‘mtu’ parameters, the following units have been defined – B (byte), kB (kilobyte), MB (megabyte) or GB (gigabyte). For the data rate parameter, the designer can use kbps (kilobits per second), Mbps (megabits per second) or Gbps (gigabits per second). The multiplication factor used to convert the sizes is ‘1024’.

## 3.2 Modeling

The testbed modeling is done in five stages – connection, protocol, application, client and server. The connection modeling defines the connection stages between client and server instances. The protocol modeling defines the packet level detail of client-server transactions. Once the protocol has been defined, the client and server modeling involves implementation of the network protocol between the client-server pairs. The application modeling for the testbed is identical to application modeling applied in the evaluation of emulation tools, section 2.4.

### 3.2.1 Connection Modeling

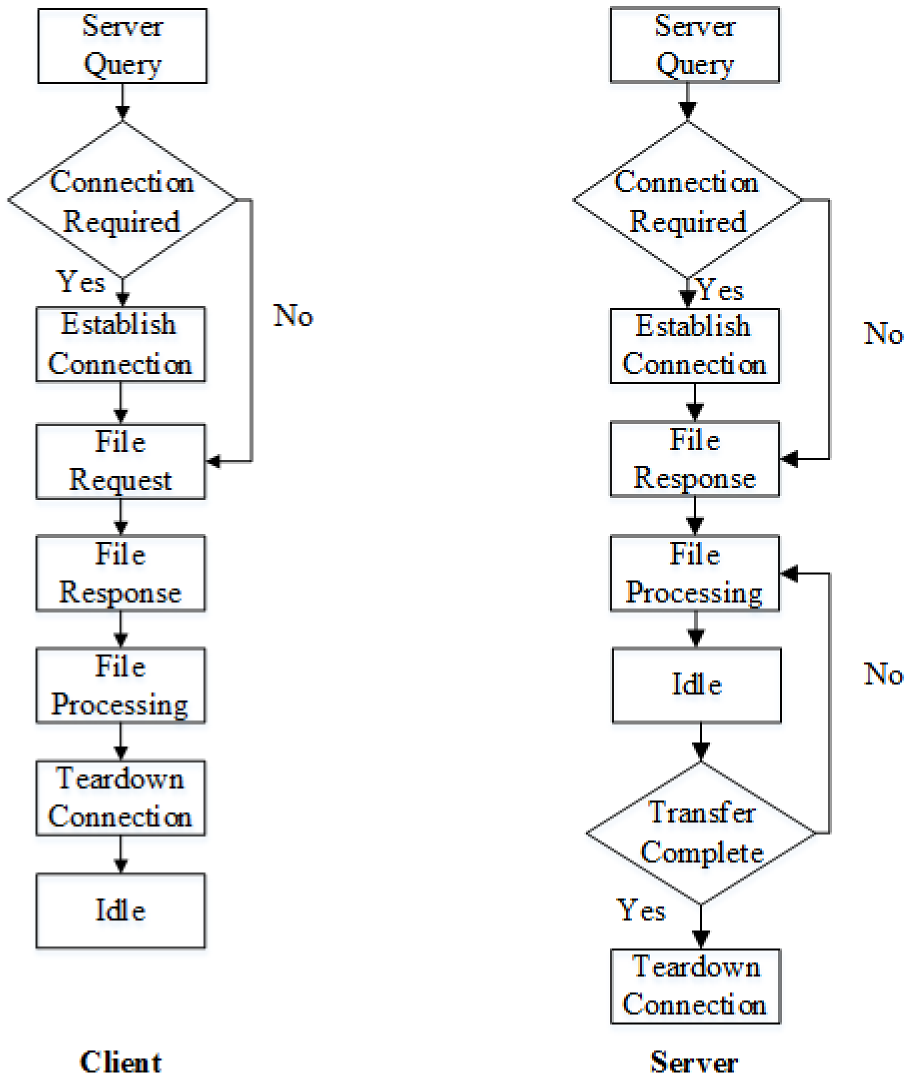


Figure 13: Connection modeling

The connection modeling defines the stages of interaction between the client and server instances. The connection modeling workflows for the client and server nodes are illustrated in Figure 13. The protocol definition by the clients and servers has been divided under seven stages of connection modeling. The stages are – server query, establish connection, file request, file response, file processing, teardown connection and idle. The server query stage is implemented by the clients to acquire the IP address for the configured server URL using DNS protocol. For implementing the UDP protocol we skip establish connection and teardown connection stages. The idle stage defines the inactive state of clients and servers. For server instances the idle stage is used to indicate the inactive state of a server for modeling the inter-packet delay. The inter-packet delay defines the rate of data transmission during a file processing. For the client instances, the idle stage defines an inactive client instance for modeling the inter-connection delay.

### 3.2.2 Protocol modeling – TCP

The nodes implement the protocol for transmission of data between servers and clients. The client and server nodes define minimal implementation of TCP and UDP protocols over IPv4 and Ethernet. Figure 14 illustrates the TCP protocol modeling. The description of the protocol modeling is given below:

1. During the initialization, the server node creates a virtual server instance and updates the control plane.
2. The client node instance manager activates a client instance which sends out a DNS request to the control plane.
3. The controller resolves the DNS query and provides the server node IP.
4. Packet 4-6 are used to establish TCP connection: The three-way handshake.
7. The client sends a file request.
8. The server responds with the metadata of the file to be transferred.
9. The client acknowledges the metadata.

10. Packets 10-11 define packets containing the actual payload sent from the server and the corresponding acknowledge packets issued by the client.
11. Upon receiving the entire file contents, the client issues a reset packet to close the connection.

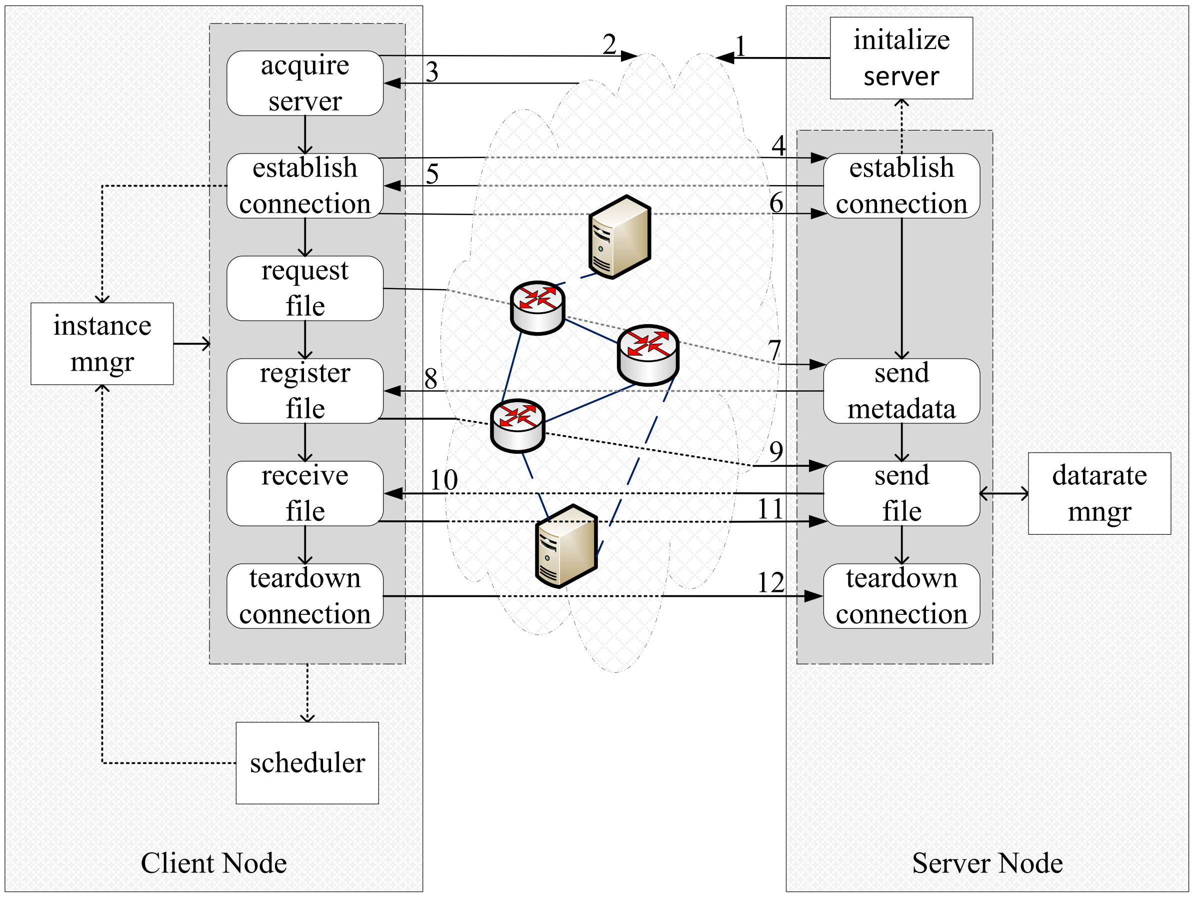
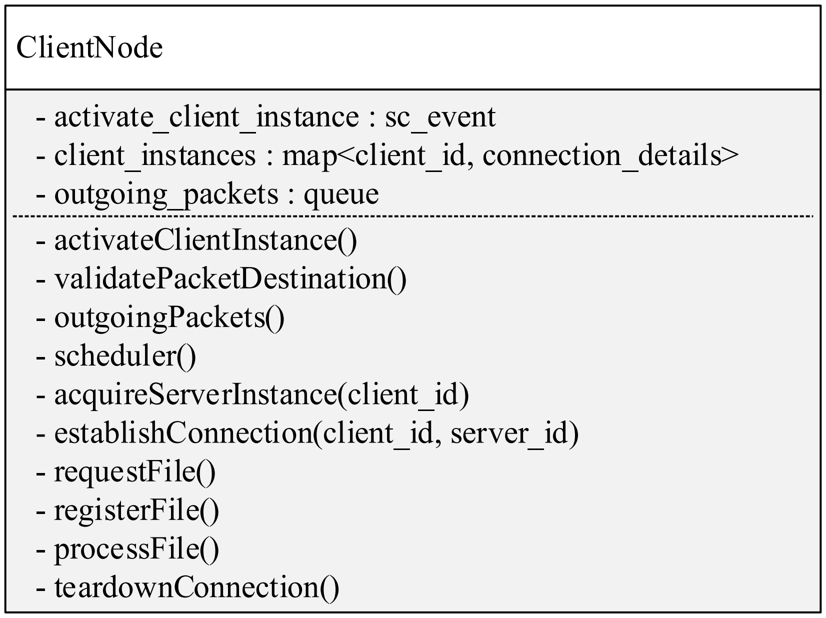


Figure 14: TCP protocol modeling

The choice of sending a reset packet after every successful file transfer is made to mimic the behavior of the Harpoon traffic generator. This allows a comparative analysis between the defined network emulation setup and proposed testbed.

### 3.2.3 Client Modeling

The node modeling focuses on scalability and ease of protocol implementation. The client node is modeled under two groups – administrative threads and behavioral methods. Figure 15 illustrates the client node class.



```
ClientNode
- activate_client_instance : sc_event
- client_instances : map<client_id, connection_details>
- outgoing_packets : queue
- activateClientInstance()
- validatePacketDestination()
- outgoingPackets()
- scheduler()
- acquireServerInstance(client_id)
- establishConnection(client_id, server_id)
- requestFile()
- registerFile()
- processFile()
- teardownConnection()
```

Figure 15: Client node modeling

Administrative threads are responsible for managing the client instances. These threads model the node architecture. The administrative threads maintain a record of all the client instances. There are four administrative client node threads – activate client instance, validate packet destination, outgoing packets and scheduler. The activate client instance thread instantiates and activates the client instances as per the configured load during the simulation. The validate packet destination thread monitors all the packets received by the client node. The thread drops any packet which is not intended for any of the node's instance. If the packet received belongs to an instance, the thread acquires the current connection stage of the instance and invokes the appropriate behavioral method. The outgoing packets thread maintains and processes the outgoing_packets buffer for all the packets to be sent out by the node. All the client instances in the node submit their packets to be sent at the current logical time to this buffer. Finally, the scheduler thread governs the idle time of the client instances once they have finished a transaction. Figure 16 shows the modeling of the client scheduler. A client instance in the idle stage is maintained by the scheduler. If all the client instances of a node are idle, the scheduler suspends the client node for the minimum required duration until at least one of the client instance can be activated. The scheduler invokes the activate client instance thread for all instances ready to establish their next connection. Our design allows us to manage multiple client instances within a client node using four threads only.

In addition to the administrative threads, the client node also implements six behavioral methods – acquire server instance, connection setup, request file, register file, process file and connection teardown. Behavioral methods define the functionality of the instances at every stage of the connection. The behavioral methods are responsible for the node protocol implementation. Figure 17 shows the lifecycle of a client instance. Once a server is acquired, the client waits for a configured timeout period for the server to establish connection. For a TCP client, the connection setup method implements the three-way handshake. For a UDP client, the connection setup method is used to simply invoke the file request method once the server is assigned. If the connection gets timed out, the client will request a new server instance.

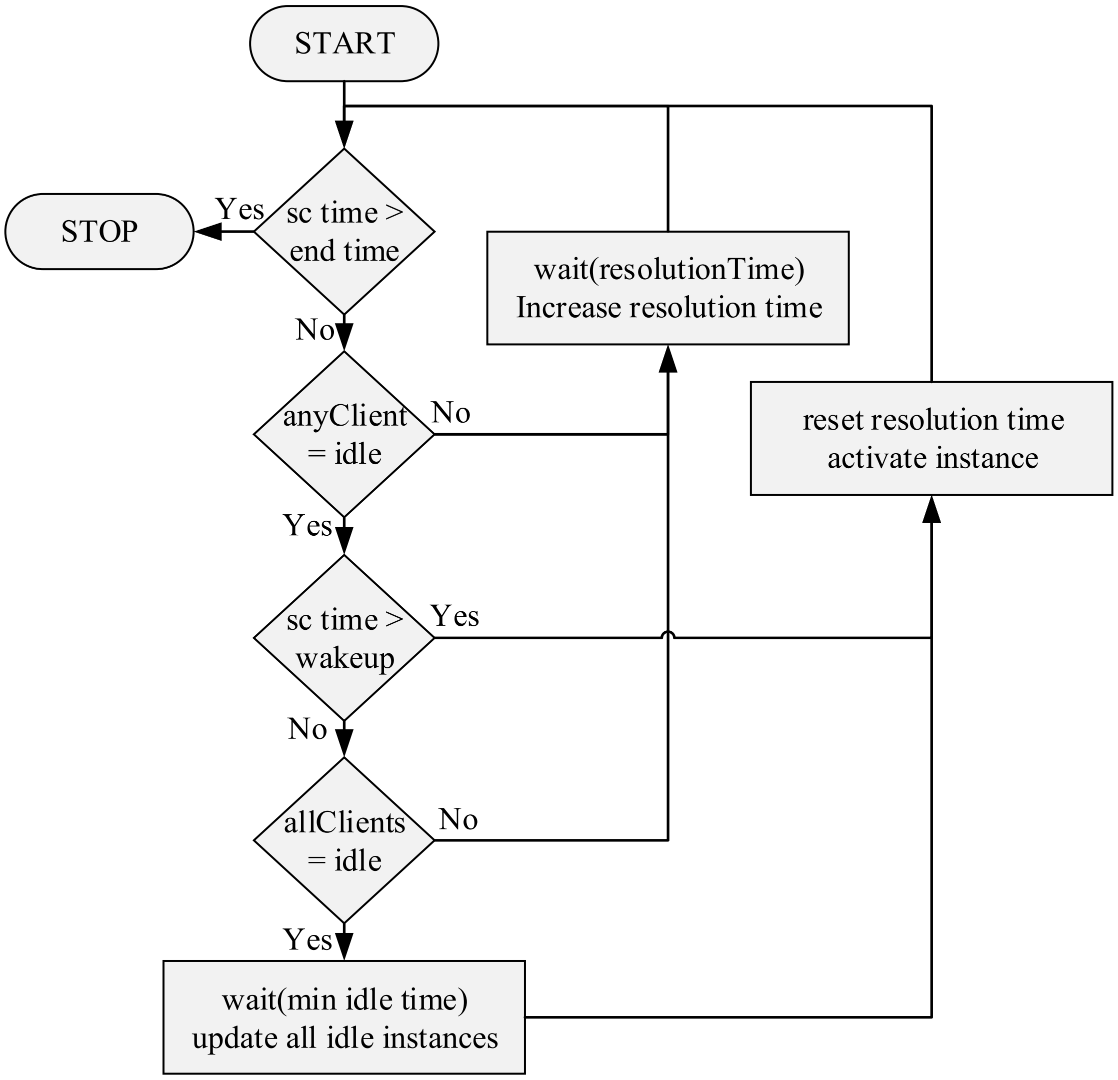


Figure 16: Client scheduler

The data members of the client node class facilitate communication between administrative threads and record the client instance details. The activate client instance event allows the scheduler to activate a client instance at the end of its idle stage. The client_instances data member maps the IP address of a client instance to the connection details. The connection details are used to determine the next behavioral action required for a client instance. Connection details consist of the active status, connection state, header for the last received packet, bytes pending for transfer during file transfer, pending idle time, the wakeup time and the index for the delay value. Every virtual client instance is associated with one active connection at a given time during the simulation.

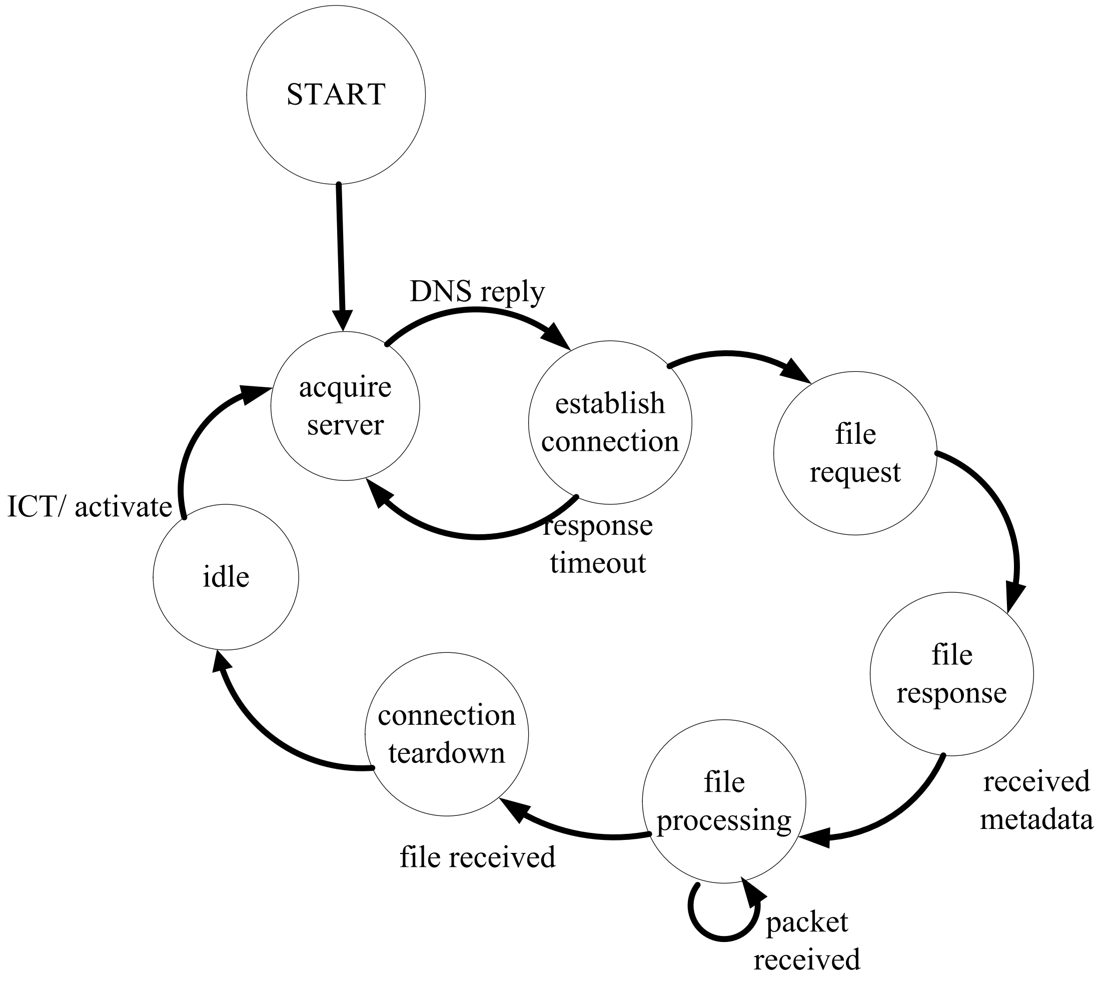


Figure 17: Client instance lifecycle

### 3.2.4 Server Modeling

The server node modeling is similar to client node modeling in some aspects. Figure 18 shows the server node modeling details. The server node consists of four administrative threads – server sessions manager, data rate manager, validate packet source and outgoing packets. The server sessions manager is responsible for creating new server instances if required during the simulation. The data rate manager manages the inter-packet delay for all the server sessions.

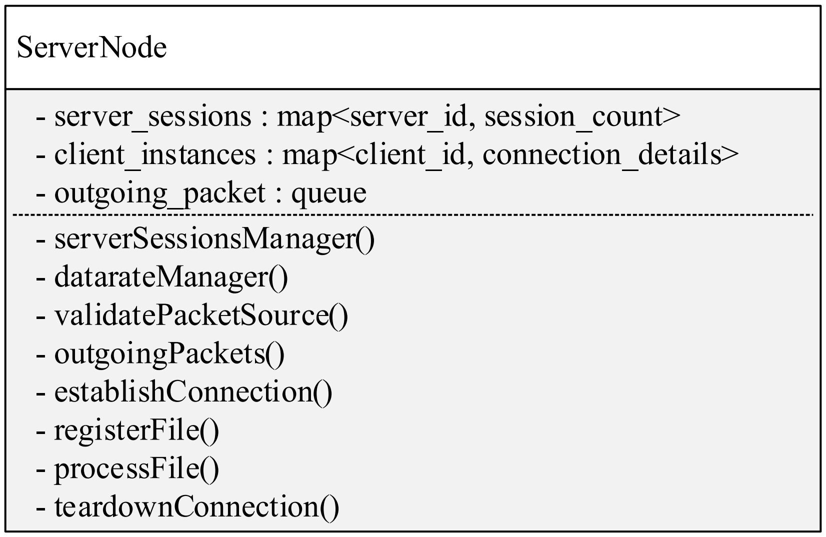

ServerNode
- server_sessions : map<server_id, session_count>
- client_instances : map<client_id, connection_details>
- outgoing_packet : queue
- serverSessionsManager()
- datarateManager()
- validatePacketSource()
- outgoingPackets()
- establishConnection()
- registerFile()
- processFile()
- teardownConnection()

Figure 18: Server node modeling

The inter-packet delays for each transmitted packet are calculated by the server instances during the file transmission stage. The delays are derived from the configured data rate and size of packet transmitted. The validate packet source thread observes the incoming packets. The packet source represents the client's IP address which is used to identify the connection state and invoke the required behavioral method. The outgoing packets thread functionality is identical to the outgoing packets thread in the client node. Figure 19 illustrates the working of a virtual server instance. The server instance waits for a client request for connection (TCP) or file (UDP). If the receiving server instance is overloaded, the request is dropped and the server updates the controller. If the server is available for processing new requests, it checks if the threshold limit is still valid for all the server instances collectively. If the arriving request crosses the configured threshold value, a new server instance will be created and communicated to the controller. Finally, the server will process the transaction using the implemented protocol and the specified data rate.

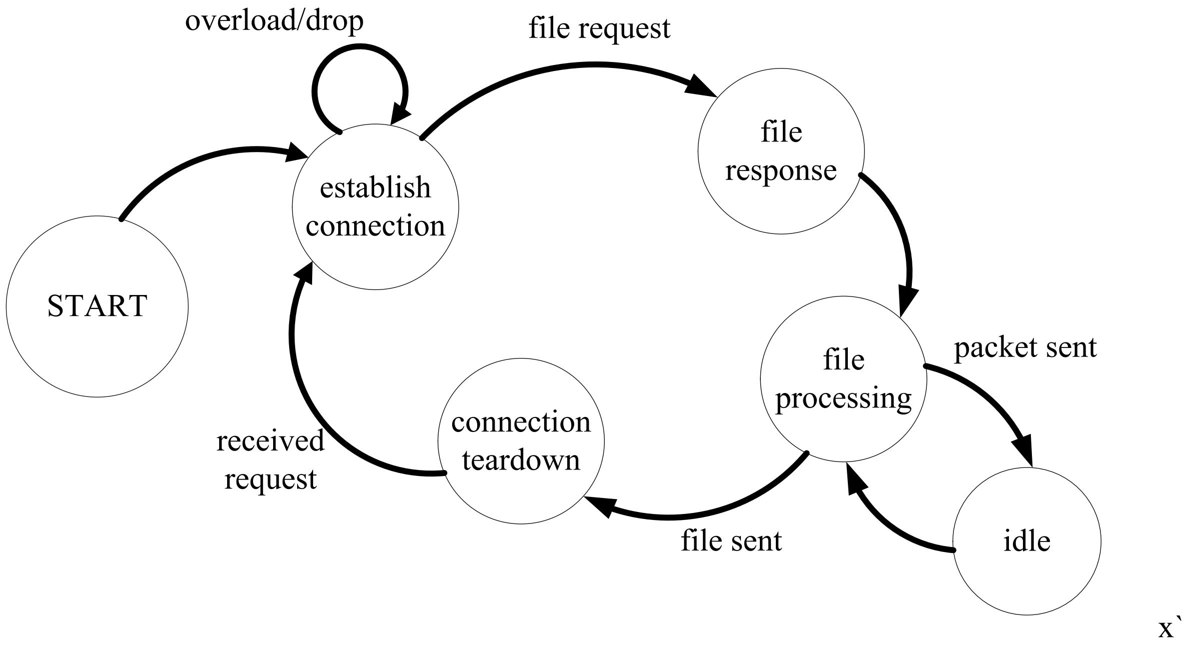


Figure 19: Server instance modeling

The server node protocol is implemented using four behavioral methods – establish connection, register file, process file and teardown connection. The establish connection behavioral method defines the server side protocol for establishing a connection when requested by a client instance. The connection setup and teardown methods are undefined for a UDP server. The register file method is used to process a file request from the client instance and updates the client with the expected size of the file to be transferred. The file transfer method is responsible for fragmenting a file into packets with maximum size as configured by the MTU parameter in the server node configuration. Once the entire file is transmitted and the connection has been closed, the server node erases the connection details. The flow updates of the control plane from the server is performed to compensate for a flow monitoring mechanism at the control plane.

## 3.3 Summary

The SystemC based simulation models implemented for the testbed define both, the hardware and software components of the testbed. SystemC defines an event-driven simulation interface. For every event, the simulation models make controlled advances in logical time. This allows the simulation kernel to process events in multiple nodes at the same logical time. Thus, the simulation models are not limited by the processing capabilities of the underlying system. We list below the significant benefits of using our simulation models:

1. The simulation models can be configured with round_robin policy to create repeatable test scenarios. The designer can also choose to introduce variations in the traffic generated by the instances by using a probability distribution in the policy fields.
2. The simulation models provide the designer a certain degree of control over the spatial distribution within the generated traffic. For example, if a designer provides a single value for the file sizes and delays, the spatial distribution in the generated traffic would strictly follow the probability distribution used to configure the policy of IP addresses in the client and server nodes.
3. The server model allows creation of dynamic server instances at runtime depending upon the sessions and threshold configurations.
4. The server models can be configured for dynamic data rate of transmission using the session delay parameter which varies depending on the load on a server instance.

# CHAPTER 4

## Testbed for Forwarding Plane Analysis

The Open Networking Foundation [24] defines Software-Defined Networking as the separation of the packet processing infrastructure from the network control logic to form a layered network architecture. Figure 20 illustrates the layered SDN architecture.

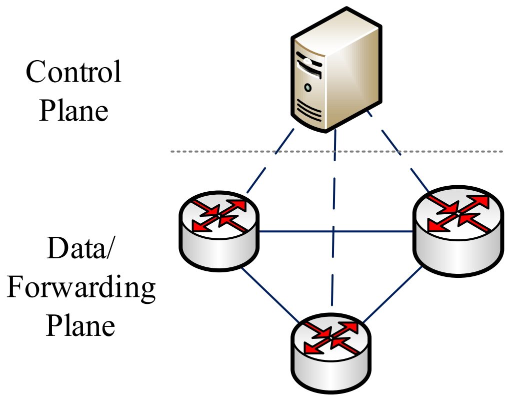


Figure 20: SDN Architecture

Under SDN, a single controller executing the control logic can supervise multiple forwarding plane elements. The remote controller performs the task of interacting with neighboring network elements, maintaining the routing tables and configuring the forwarding plane elements when required. To promote programmability of SDN based forwarding plane elements, packet processing languages [18, 25] allow an application developer to program forwarding hardware devices such as network processor unit (NPU) and reconfigurable match table (RMT) [26]. To perform design-space exploration of SDN based forwarding plane elements, PFPSim [27] provides hardware designers with a library to develop, design and assess forwarding plane simulators to test their architectures. PFPSim provides application developers with a simulation framework to debug and optimize forwarding plane logic on a hardware simulator.

In this chapter we describe the integration of the proposed simulation models for a network testbed with the PFPSim library to analyze network processors. Next, we will also describe a PFPSim based forwarding plane model. The motivation for the proposed testbed has been to analyze forwarding plane elements. However, we also extend the functionality of the testbed to perform a comparative analysis of load balancing algorithms. To perform the analysis, we implement a load balancer in the control plane of the pre-existing hardware model of the NPU. Finally, we use the proposed testbed to perform simulation-based analysis of the forwarding plane and control plane models.

## 4.1 Network Testbed Modeling

To integrate the simulation models discussed in Chapter 3 as a network testbed and with PFPSim models, a Forwarding Architecture Description (FAD) file has been generated for the testbed. Appendix A shows the FAD file for the proposed testbed. All the nodes of the testbed have been implemented as processing elements (PEs). We define four PEs for implementing TCP and UDP based clients and servers. The TCP client PE is instantiated twice to model clients for online video streaming application – YouTube long on-off and YouTube short on-off. The UDP client PE is instantiated once to model a VoIP client. We also instantiate the corresponding server PEs to establish client-server node pairs. Hence, we have a total of six instantiations (three client nodes and three server nodes). In addition to the Internet application nodes, we also instantiate test client-server pairs to generate TCP and UDP test traffic. These nodes can be configured as desired to generate TCP or UDP packets.

To interface testbed nodes with PFPSim models, we define a multiplexer PE and a de-multiplexer PE. All the client and server outgoing packets are aggregated by the multiplexer. The output port of the multiplexer PE is bound with the ingress port of PFPSim's simple_router model [28]. The packets sent out by the simple_npu model are associated with a logical egress port which is assigned by the P4 application executing on the simple_npu model. The logical egress port is used by the de-multiplexer component to route the packet to the correct node in the testbed environment. The testbed model is illustrated in Figure 21. The testbed integrates seamlessly with PFPSim models and allows the designers to perform replicable and reliable analysis using realistic network traffic.

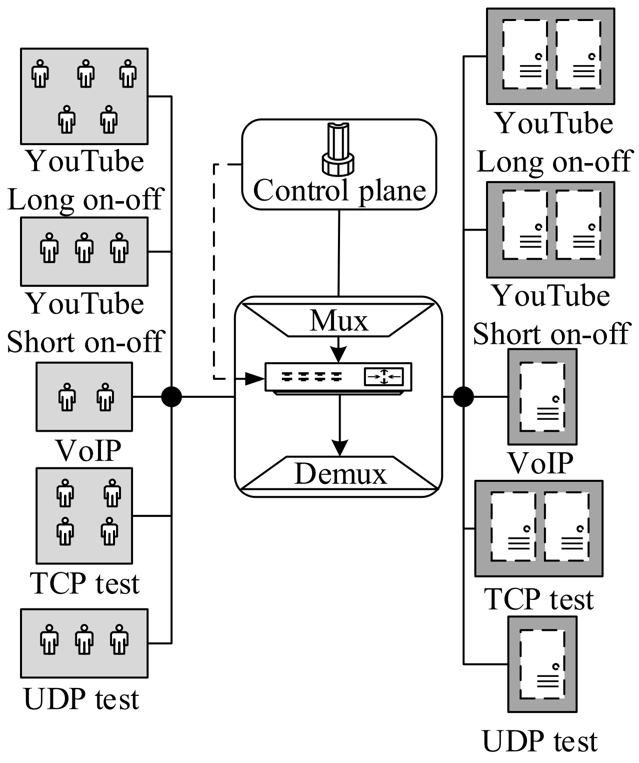


Figure 21: Testbed for SDN applications

## 4.2 Forwarding plane design: simple_npu

The PFPSim simple_npu [28] is a simulation model of a typical network processor. Figure 22 illustrates the simple_npu architecture.

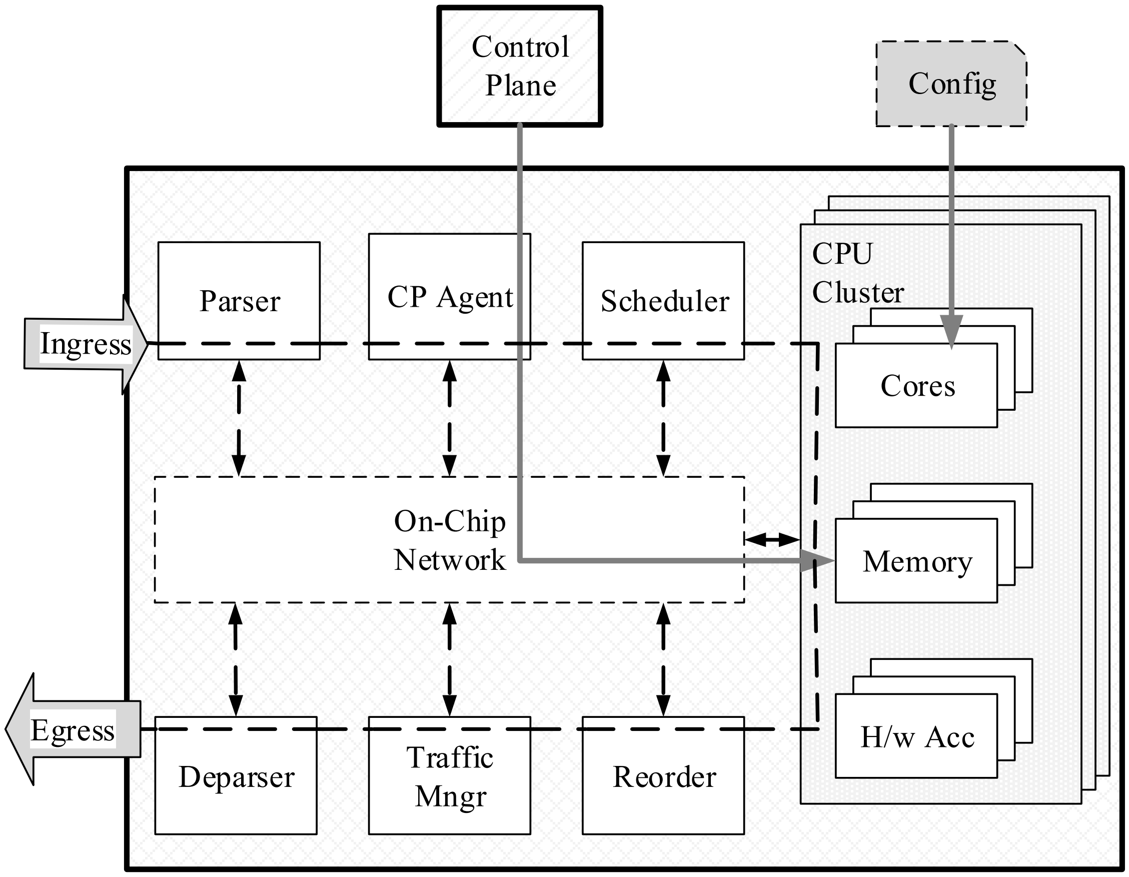


Figure 22: Typical network processor architecture

The simple_npu is modeled using configurable PEs. Under PFPSim, the PEs are configured using JSON files. Every configurable processing element can be provided with two level of configurability. A PE, for example memory, may be instantiated multiple times within the design. For the PE, the designer can have a parent configuration file which defines the default parameters for all the instances of the PE. At a lower level, every instance can also specify its own configuration file which distinguishes it from other instances. The architecture defines one ingress and one egress port. However, the P4 application provides the logical egress port for every packet that has been processed.

While integrating the simple_npu model with the testbed and control plane, the top and control plane PEs were removed from the simple_npu FAD to obtain a clean forwarding plane FAD. The top PE has been moved into the testbed FAD and the control plane PE has been moved into a new control plane FAD as a “controller” PE for better organization.

## 4.3 SDN based Application: Load Balancer

The simple_npu model defines a simple control plane which initializes the forwarding plane and populates routing tables in the memory. To showcase analysis of a SDN application, a load balancer module has been implemented in the control plane.

### 4.3.1 Architecture

The complete testbed architecture with the modified control plane is shown in Figure 23. The simple_npu’s control plane functionality has been transferred into a controller PE. The controller PE implements a load balancer service, which allows the load balancer PE to insert, modify and delete table entries in the forwarding plane. The load balancer manages two network address translation (NAT) tables in the forwarding plane – the forward NAT table and the reverse NAT table. The forward NAT table updates the destination IP address for all uplink packets. The reverse NAT table updates the source IP address of all downlink packets. Listing 13 and Listing 14 show the commands available to update the forward and reverse NAT tables from the load balancer.

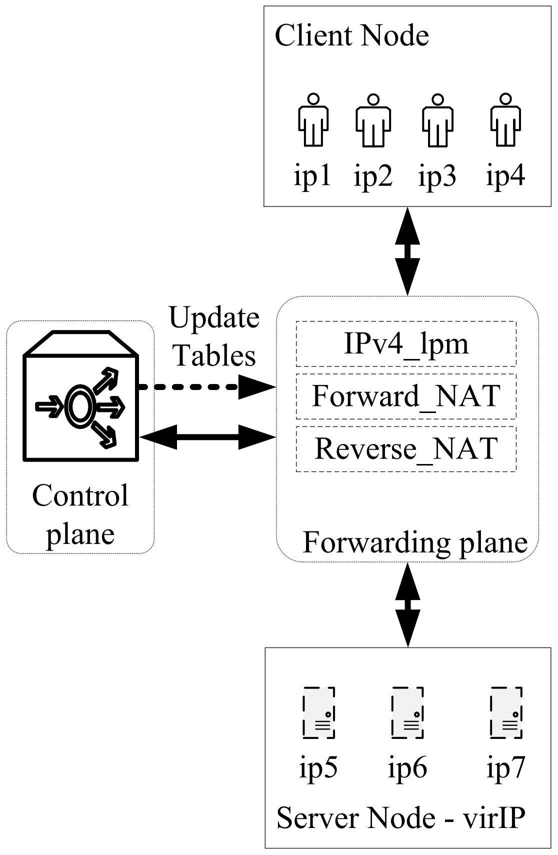


Figure 23: Testbed architecture with load balancer

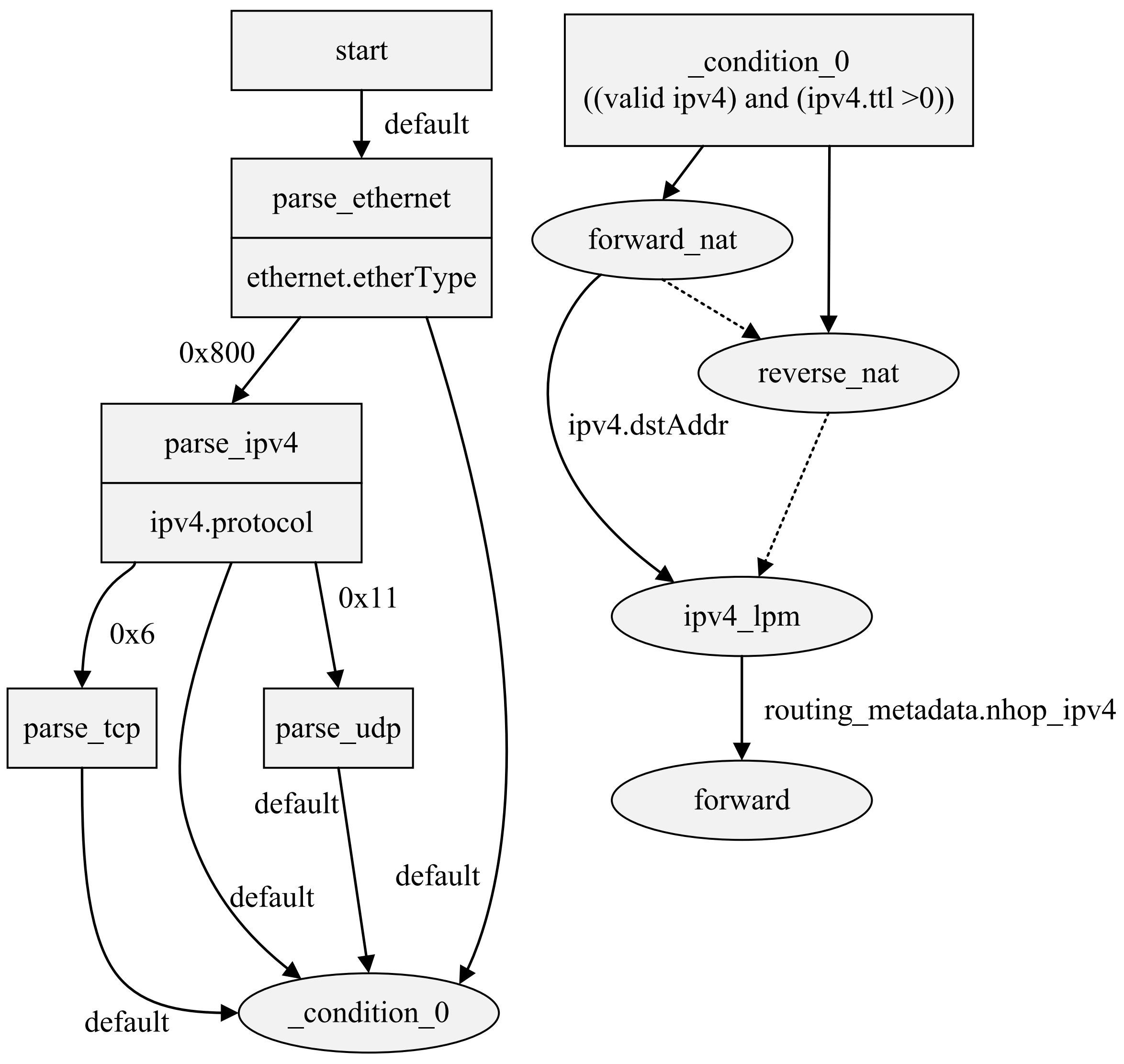
Listing 13: Forward NAT table commands

1. insert_entry forward_nat [*vClient_ip*] [*node_ip*] perform_forward_nat [*vServer_ip*]
2. modify_entry forward_nat [*entry_handle*] [*new_vServer_ip*]
3. delete_entry forward_nat [*entry_handle*]

Listing 14: Reverse NAT table commands

1. insert_entry reverse_nat [*vServer_ip*] perform_reverse_nat [*node_ip*]
2. modify_entry reverse_nat [*entry_hande*] [*new_node_ip*]
3. delete_entry reverse_nat [*entry_handle*]

The modified forwarding plane router functionality is illustrated in Figure 24.



(a) Parser (b) Forwarding Plane Tables

Figure 24: Load balancing forwarding plane functionality

The forwarding plane functionality for performing load balancing has been updated in terms of parsing of the packet headers and the number of match-action tables in the forwarding plane. The parser functionality has been updated to parse the TCP and UDP headers. The router retains relevant header fields as metadata for every packet. Once a packet has been parsed, we perform either forward or reverse NAT. Once the IP address has been updated, the normal workflow of the simple_router is followed to route the packet for the appropriate destination.

### 4.3.2 Modeling

The forward and reverse NAT tables perform address translation in the forwarding plane once a flow has been determined by the load balancer. The load balancer PE is configurable to any one of the following three load balancing algorithms for allocating server instances to the clients – static, round robin and shortest queue. Figure 25 shows the interaction of server and client nodes with the control plane. The blue lines and the adjacent numbers on the figure show the workflow.

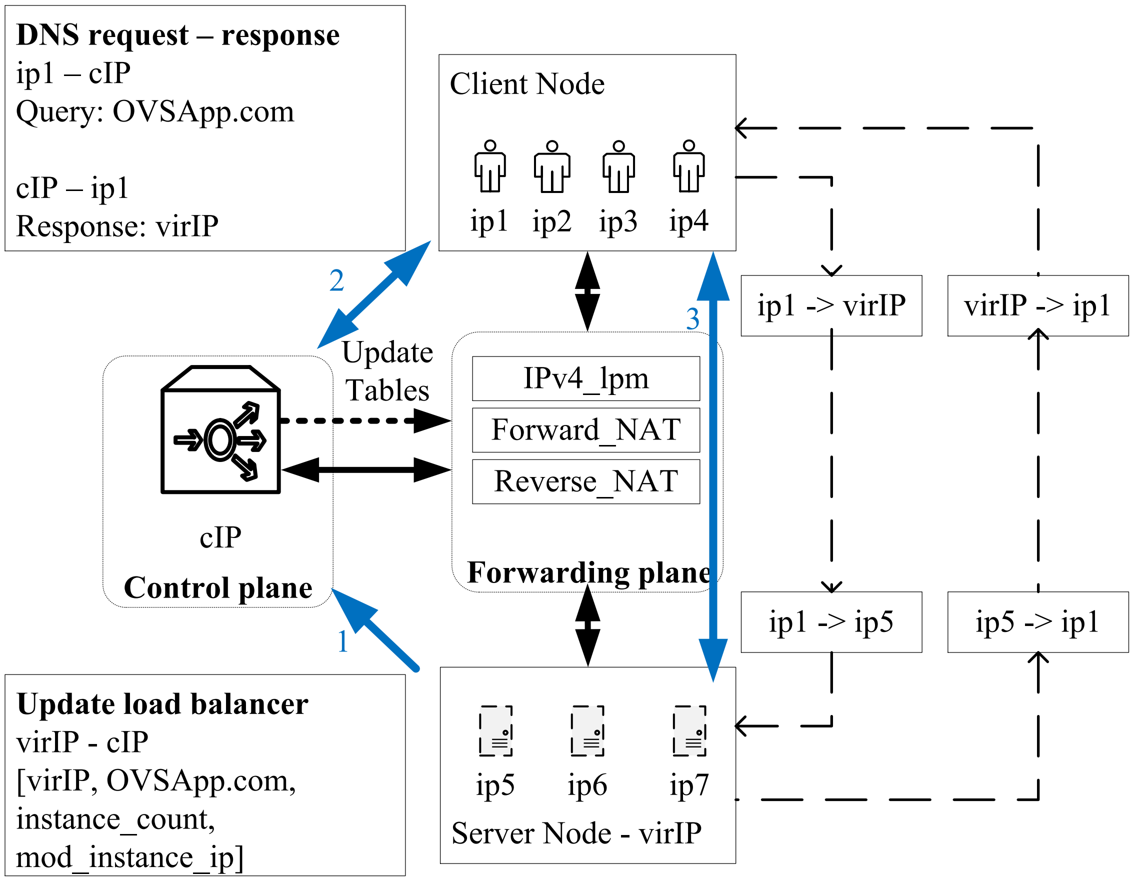


Figure 25. Workflow of the load balancer

Upon initialization, the server nodes update the control plane with the server URL, the server node IP address and a list of existing virtual server instances. Figure 26 shows the packet format used by the server sessions manager to communicate with the control plane. The virtual server instance IP address and the node IP of the server node are used to create the reverse NAT table. Next, the clients issue DNS requests to the load balancer. The DNS response to the clients contains the public IP for the servers. At the same time, the forward NAT table is also updated. The forward NAT table has three entries, the virtual client instance IP, the public/server node IP and the virtual server instance IP. For all future packets from the client instance with an exact match on the public IP as the destination, the forwarding plane will update the destination address of the packet using the forward NAT table to that of the assigned virtual server instance IP. The reverse NAT table is used to reassign the node IP as the source address before dispatching a packet to the client instance. Hence, the forwarding plane performs double NAT using the forward and reverse NAT tables on all interactions between the clients and servers.

The shortest queue algorithm ensures that the server with the least load at the time of the request is allocated to the client. This algorithm needs to maintain a state of all server instances. In case of multiple servers being available with the minimum load, we choose a server instance randomly to distribute the load over various server instances during the simulation. The forward NAT table is updated only if a client is unknown for the load balancer or if the client is assigned a different server instance. The static algorithm allocates a server to a client based on the shortest queue algorithm initially. For every subsequent request by the client, the same server instance is allocated to the client. Thus the number of updates of the forward NAT table are minimum for the static load balancing algorithm. The round robin algorithm allocates the server instances in a round robin manner to the requesting clients. The static and round-robin algorithms depend on client behavior and similarity within traffic characteristics to balance the load on servers.

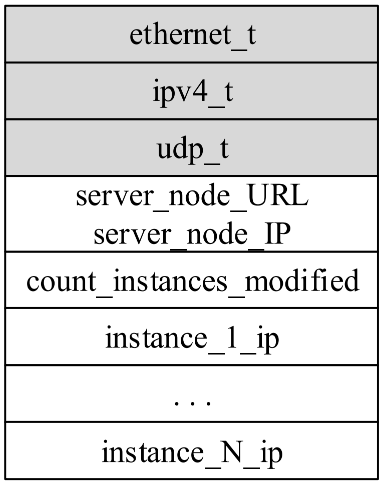


Figure 26: Load balancer update packet format

## 4.4 Experimental Results

In this section we present a comparative analysis of the emulation setup and the proposed testbed. We also present an analysis of the scalability the proposed testbed. Finally, we present an analysis of the simple-npu model and the load balancing application using the testbed simulator. All the experiments are performed on an isolated core of an Intel i5 2.30GHz processor with 4 GB RAM running Ubuntu 14.04

### 4.4.1 Comparative Analysis – Emulation vs. Simulation

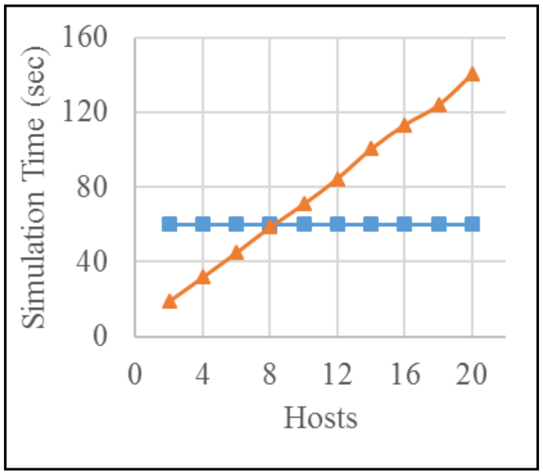
We perform the comparative analysis using only the TCP test nodes in the network emulation setup and the proposed testbed. All other nodes of the proposed testbed have been shut down by configuring the nodes with zero virtual client instances. The configured and expected values for the traffic generated in network emulation setup and the testbed are summarized in

Table 6. After every experiment we increase the number of client and server instances for both the network emulation setup and the proposed testbed till we observe significant variation in execution behavior from expected. In the ideal case, we expect to see a proportional increase in the number of files requests and volume of traffic generated with increase in the number of client-server pairs. Figure 27 illustrates the performance comparison in terms of simulation speed, memory usage, application threads, client requests issued and generated traffic volume.

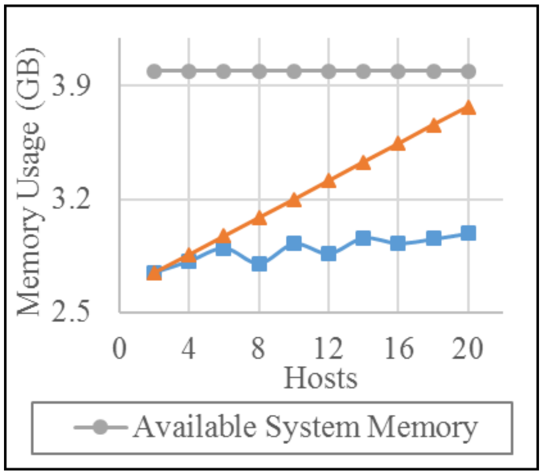
Table 6: Configuration and expected values

| Property | Configured/ Expected |
|---|---|
| Simulation time | 60 seconds |
| Inter-connection time | 0.5 seconds |
| File size | 1024 bytes |
| File metadata | 5 bytes |
| File requests per client | 120 |
| Data transfer per client-server pair | 123,480 bytes |

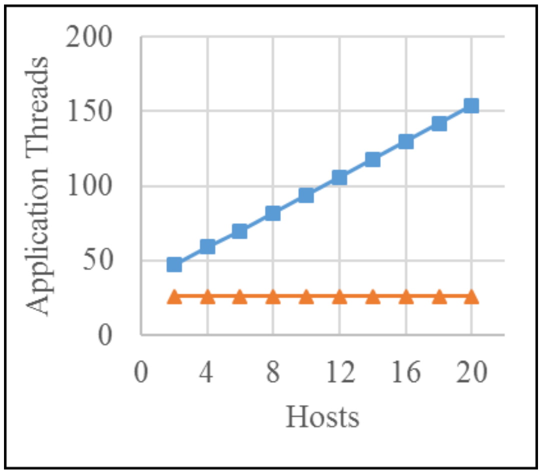
From the results we observe that the simulation speed for the proposed testbed depends on the number of nodes being modeled. We observe increase in the required simulation time and memory usage for the testbed with an increase in the number of instances. The high memory usage of the testbed could be attributed to memory leaks. We observe that the emulation setup is scalable reliably only up to six hosts. Beyond six hosts the thread contention makes the thread execution in real-time unmanageable. The simulator output remains consistent with the expected values throughout the experiment.



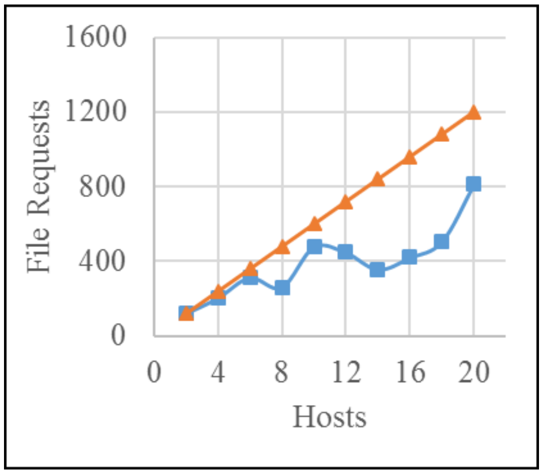
(a) Execution Time



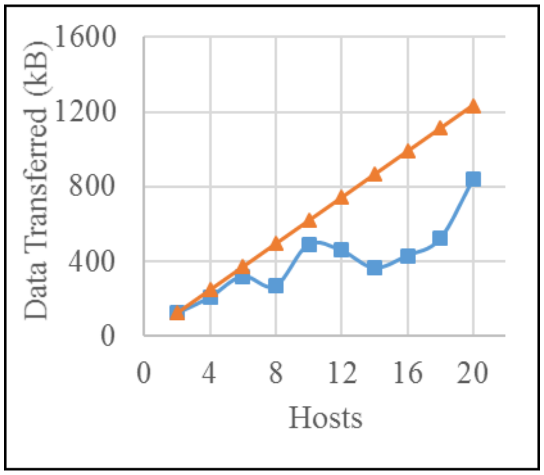
(b) Memory Usage



(c) Application Threads



(d) File Requests Issued

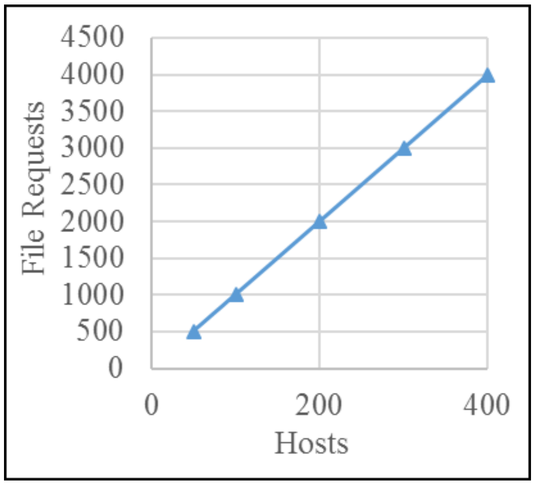


(e) Data Transferred

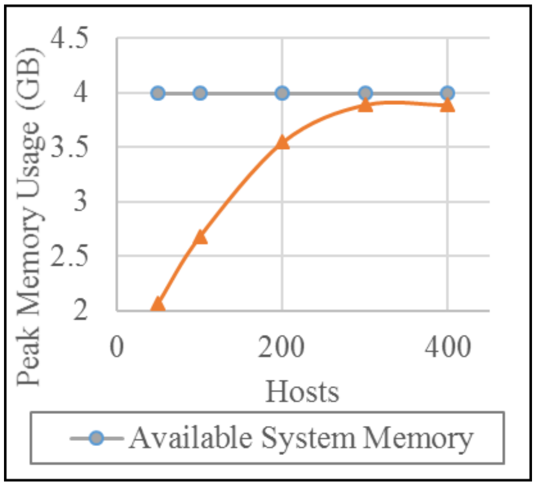
Figure 27: Emulation □ vs. Simulation △

### 4.4.2 Testbed Scalability

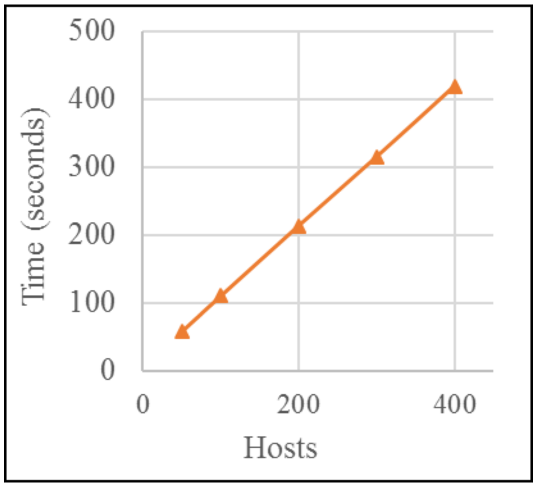
In this section we explore the scalability of the proposed testbed. The testbed configuration is kept similar to section 4.4.1. We execute the simulation for a logical time of ten seconds. Figure 28 shows the results of this experiment. The experiments are done for testbeds with 50, 100, 200, 300 and 400 instances.



(a) File Requests Issued



(b) Memory Usage



(c)

Figure 28: Testbed scalability analysis

We observe a linear increase of total number of requests with increasing number of hosts. The memory usage for the simulator gets saturated at the 4GB mark. This is due to the system limitations of maximum available memory. The execution time for a simulation also increases linearly with increase in the number of hosts in the testbed. By comparing the memory usage by the simulator in section 4.4.1, we observe that the memory usage of the testbed depends on both, the number of hosts being simulated and the execution time of the simulation. During the scalability analysis we were able to successfully simulate the testbed with 400 unique hosts generating the TCP traffic as described in Table 6 for ten seconds on a single laptop.

### 4.4.3 Load Balancer

In this section we present a comparison of three different load balancing algorithms. The load balancing algorithms have been implemented in the control plane. To perform the analysis, we use real-life traffic representing YouTube's video streaming using the short on-off strategy and VoIP traffic. Table 7 illustrates the node configuration of the traffic.

Table 7: Node configurations - DUT

| Node | Parameter | Values |
|---|---|---|
| Client | Instances | 35 |
| | Time out | 10ms |
| Server | Sessions | 6 |
| | Session delay | 100us |

The load balancing algorithms implemented in the design are – shortest queue, static and round robin. The analysis allows the designer to predict the behavior of the algorithms for two different traffic profiles. The YouTube video streaming profile consists of comparatively long

IP flows than the VoIP traffic. Hence, for every packet transmitted between the client and server node, the number of server requests to the load balancer is quite low for YouTube traffic. Table 8 illustrates a summary of the differences due to the traffic profiles. We observe that on an average for YouTube traffic there are 500 packets in every flow. On the other hand, for VoIP traffic there are approximately 100 packets in every flow.

Table 8: Traffic profiles - Load balancer analysis

| Traffic profile | Simulation Time | Packets transmitted | Server Requests |
|---|---|---|---|
| YouTube | 5 seconds | ~ 60 thousand | ~120 |
| VoIP | 50 seconds | ~ 20 thousand | ~200 |

Next, we analyze the efficiency of the load balancing algorithms for the given traffic profiles. Table 9 provides an analysis of the load balancing algorithms. The metrics used are:

1. Load balancing: The difference of load between the least utilized and maximum loaded server instance for the entire simulation.
2. Response time: The time lapse between the client request for a server allocation to the time it takes for the client to receive a valid file response for a requested file.
3. Control plane updates: For all incoming new flows, percent of flows which needed the forwarding plane's NAT tables to be updated.
4. Connections dropped: Percent connections dropped by the server instances

Table 9: Comparison - Load balancing algorithms

| Traffic profile | YouTube | | | VoIP | | |
|---|---|---|---|---|---|---|
| Algorithms | Round robin | Static | Shortest queue | Round robin | Static | Shortest queue |
| Load Balancing | 12101 ms | 12343 ms | 11146 ms | 87491 ms | 101389 ms | 27280 ms |
| Average response time | 4594 us | 7 us | 8 us | 14748 us | 9 us | 13 us |
| Worst case response time | 80119 us | 10 us | 10 us | 140127 us | 100 us | 100 us |
| Control plane updates issued | 90.32% | 30.17% | 90.76% | 85.26% | 17.50% | 86.80% |
| Connections dropped | 7.26% | 0 | 0 | 6.25% | 0 | 0 |

For both YouTube and VoIP, we observe the shortest queue algorithm provides the best balance for load amongst the available servers. However, we observe that the average response time for static algorithm is slightly better. This is expected because every time the controller updates the forwarding plane, the response gets delayed. We observe that static algorithm has approximately 60% less NAT table updates compared to round robin and shortest queue. Finally, we observe that for both, static and shortest queue, the number of connections actually dropped by the server due to overloading is 0. However, for round robin we observe 6 – 7% connection drops which cause connection timeout and also a poor response time. Overall, we observe that shortest queue performs best in terms of load balancing. The implementation of a static load balancing algorithm could prove beneficial if updating of the forwarding plane is associated with high penalties. Even though it is easiest to implement, round robin can cause overloading of servers which could be highly undesirable.

### 4.4.4 Forwarding Plane Analysis – simple_npu

In this section, we analyze the PFPSim simple-npu model using the proposed testbed. We analyze the simple_npu model based on size of the routing table under two different scenarios. For both the scenarios, we use the YouTube and VoIP traffic profiles. For the first scenario we configure the server and client nodes to use only a single prefix. Hence, the IPv4_LPM table consists of only three entries – the client node prefix, the server node prefix and the controller prefix. For the second scenario, the server and client nodes use five prefixes each. The IPv4_LPM table consists of 11 entries – total 10 for the client-server nodes and one for the controller. By modifying the IP pool size and number of entries in the IPv4_LPM table, we can analyze the effects of spatial diversity on the latency of packets. The IPv4_LPM table is used by the P4 application running on the simple_npu model to dictate the egress port based on the destination IP address of a packet. For the given scenarios we observe that for the two traffic profiles with two different spatial distributions, we obtain different latencies. Table 10 summarizes our observations.

Table 10: PFPSim simple-npu analysis

| Traffic profiles | IPv4_LPM size | Latency (nanoseconds) |
|---|---|---|
| YouTube | 3 | 724 |
|  | 11 | 875 |
| VoIP | 3 | 592 |
|  | 11 | 698 |

The consistent higher latency for both traffic profiles for the second scenario can be attributed to larger routing tables and longer prefix lengths required for assigning the egress ports.

## 4.5 Summary

In this chapter we described a network testbed created using simulation models of clients and servers. The testbed integrates seamlessly with PFPSim models. Using the testbed, we were able to perform analysis to determine the effect of spatial diversity in network traffic on the performance of the network processor. This analysis is not possible using the network emulation setup. The testbed created using the simulation models is not limited by the capability of the underlying system and has absolute control over the generated network traffic in terms of generated bandwidth, spatial diversity, spatial distribution and throughput.

We also showcased the analysis of a SDN based load balancing application using the proposed testbed and performed a comparative analysis of different load balancing algorithms.

A comparative analysis of the testbed with the network emulation setup showcased a high level of accuracy and reliability of the proposed testbed in generating network traffic for large networks on a standalone system.

# CHAPTER 5

## Conclusion and Future work

In this thesis, we presented our work on generating scalable realistic network traffic on a standalone system. We presented a methodology to generate realistic network traffic using well-defined traffic characteristics of Internet applications such as, online video streaming and VoIP. We implemented a testbed using SystemC based simulation models of clients and servers. The clients and servers were synchronized using a connection modeling framework. The connection modeling framework was used to define minimal implementation of TCP and UDP protocols over IPv4 and Ethernet to transfer data between the client and server models. We use the proposed testbed to perform simulation-based analysis of forwarding plane designs. We analyzed the performance of the forwarding plane design for varying spatial diversity in the network traffic. We also analyzed SDN based load balancing algorithms and performed a comparative analysis of load balancing algorithms. A comparative analysis of the network emulation setup and the proposed testbed showcased that the proposed testbed is highly accurate and reliable.

### 5.1 Benefits

The proposed testbed provides significant advantages over emulation tools for generating network traffic on a standalone system as listed below:

- Scalability: The testbed is capable of simulating a network of up to 400 hosts and generate realistic network traffic on a single laptop.

- Repeatable experiments: The testbed simulator can be used to generate repeatable network traffic. The testbed integrates seamlessly with PFPSim models. This enables the designer to conduct repeatable network experiments without the need to archive packet traces.

- Realistic design analysis: Using the testbed we can perform analysis of forwarding plane designs using realistic traffic profiles.

## 5.2 Future work

In the future, we will extend the testbed to incorporate the following to:

1. Modeling of Internet applications such as HTTP, FTP and remote login to generate diverse use case scenarios and more realistic network traffic.
2. Modeling of IPv6, ICMP protocol on the client and server simulation models.
3. More detailed modeling of transmission protocols to incorporate packet level dynamics such as TCP congestion control and ACK clocking.
4. Modeling of network parameters such as jitter, link delays, link packet loss, etc.
5. Developing a GUI to configure the testbed and the traffic profiles.

# Appendix

## A. Testbed FAD

```
import control_plane;
interface TestbedRdI, TestbedWrI;
CE TestbedQueue("TestbedQueue.cfg") implements TestbedRdI, TestbedWrI;
CE ExtQueue("ExtQueue.cfg") implements TestbedRdI, TestbedWrI, QueueRdI, QueueWrI;
PE TCPServer("TCPServer.cfg"){
  TestbedRdI in;
  TestbedWrI out;
};
PE TCPClient("TCPClient.cfg"){
  TestbedRdI in;
  TestbedWrI out;
};
PE UDPServer("UDPServer.cfg"){
  TestbedRdI in;
  TestbedWrI out;
};
PE UDPClient("UDPClient.cfg"){
  TestbedRdI in;
  TestbedWrI out;
};
PE TestbedMux {
  TestbedRdI in[];
  TestbedWrI out;
  TestbedWrI bypass;
};

PE TestbedDemux {
  TestbedRdI in;
  TestbedWrI out[];
  TestbedRdI bypass;
};
PE Testbed("Testbed.cfg") {
  TestbedRdI in_npu;
  TestbedWrI out_npu;
  TestbedRdI in_cp;
  TestbedWrI out_cp;
  TestbedQueue mux_in[11];
  TestbedQueue demux_out[11];

  TCPServer server_ytl("server_ytl.cfg");
  TCPClient client_ytl("client_ytl.cfg");
```

```
bind client_ytl.out {mux_in[0]};
bind client_ytl.in {demux_out[0]};
bind server_ytl.out {mux_in[1]};
bind server_ytl.in {demux_out[1]};

TCPServer server_yts("server_yts.cfg");
TCPClient client_yts("client_yts.cfg");
bind client_yts.out {mux_in[2]};
bind client_yts.in {demux_out[2]};
bind server_yts.out {mux_in[3]};
bind server_yts.in {demux_out[3]};

UDPServer server_voip("server_voip.cfg");
UDPClient client_voip("client_voip.cfg");
bind client_voip.out {mux_in[4]};
bind client_voip.in {demux_out[4]};
bind server_voip.out {mux_in[5]};
bind server_voip.in {demux_out[5]};

TCPServer server_tcp_test("server_tcp_test.cfg");
TCPClient client_tcp_test("client_tcp_test.cfg");
bind client_tcp_test.out {mux_in[6]};
bind client_tcp_test.in {demux_out[6]};
bind server_tcp_test.out {mux_in[7]};
bind server_tcp_test.in {demux_out[7]};

UDPServer server_udp_test("server_udp_test.cfg");
UDPClient client_udp_test("client_udp_test.cfg");
bind client_udp_test.out {mux_in[8]};
bind client_udp_test.in {demux_out[8]};
bind server_udp_test.out {mux_in[9]};
bind server_udp_test.in {demux_out[9]};

TestbedQueue bypass;
TestbedMux mux;
TestbedDemux demux;

bind mux.bypass {bypass};
bind demux.bypass {bypass};
bind mux.in {mux_in[0], mux_in[1], mux_in[2], mux_in[3], mux_in[4], mux_in[5],
        mux_in[6], mux_in[7], mux_in[8], mux_in[9], in_cp};
bind demux.out {demux_out[0], demux_out[1], demux_out[2], demux_out[3],
        demux_out[4], demux_out[5], demux_out[6], demux_out[7], demux_out[8],
        demux_out[9], out_cp};

bind mux.out {out_npu};
```

```
  bind demux.in {in_npu};
};
PE top("TopConfig.cfg") {
  ExtQueue testbed_npu, npu_testbed;
  ExtQueue testbed_cp, cp_testbed;
  Testbed;
  NPU npu("NPU.cfg");
  ControlPlane control_plane;
  bind control_plane.cpa {npu};
  bind testbed.out_cp    {testbed_cp};
  bind control_plane.in  {testbed_cp};
  bind control_plane.out {cp_testbed};
  bind testbed.in_cp     {cp_testbed};
  bind testbed.out_npu {testbed_npu};
  bind npu.ingress {testbed_npu};
  bind npu.egress {npu_testbed};
  bind testbed.in_npu  {npu_testbed};
};
```

## B. Control Plane FAD

```
import forwarding_plane;
service LoadBalancerS;
PE LoadBalancer("LoadBalancer.cfg") {
  QueueRdI in;
  QueueWrI out;
  LoadBalancerS lbs;
};
PE MainController implements LoadBalancerS {
  ControlPlaneAgentS cpa;
};
PE ControlPlane {
  QueueRdI in;
  QueueWrI out;
  ControlPlaneAgentS cpa;
  MainController main_controller;
  bind main_controller.cpa {cpa};
  LoadBalancer load_balancer;
  bind load_balancer.in  {in};
  bind load_balancer.out {out};
  bind load_balancer.lbs {main_controller};
};
```

## C. Double NAT P4 switch

```
header_type ethernet_t {
   fields {
      dstAddr : 48;
      srcAddr : 48;
      etherType : 16;
   }
}
header_type ipv4_t {
   fields {
      version : 4;
      ihl : 4;
      diffserv : 8;
      totalLen : 16;
      identification : 16;
      flags : 3;
      fragOffset : 13;
      ttl : 8;
      protocol : 8;
      hdrChecksum : 16;
      srcAddr : 32;
      dstAddr: 32;
   }
}
parser start {
   return parse_ethernet;
}
#define ETHERTYPE_IPV4 0x0800
header ethernet_t ethernet;
parser parse_ethernet {
   extract(ethernet);
   return select(latest.etherType) {
      ETHERTYPE_IPV4 : parse_ipv4;
      default: ingress;
   }
}
header ipv4_t ipv4;
field_list ipv4_checksum_list {
      ipv4.version;
      ipv4.ihl;
      ipv4.diffserv;
      ipv4.totalLen;
      ipv4.identification;
      ipv4.flags;
      ipv4.fragOffset;
      ipv4.ttl;
      ipv4.protocol;
      ipv4.srcAddr;
      ipv4.dstAddr;
}
field_list_calculation ipv4_checksum {
   input {
      ipv4_checksum_list;
   }
   algorithm : csum16;
   output_width : 16;
}
calculated_field ipv4.hdrChecksum  {
   verify ipv4_checksum;
   update ipv4_checksum;
}
#define IP_PROT_TCP 0x06
#define IP_PROT_UDP 0x11
header_type meta_nat_t {
   fields {
      ipv4_sa : 32;
      ipv4_da : 32;
      srcp : 16;
      dstp : 16;
      tcpLength : 16;
   }
}
metadata meta_nat_t meta_nat;
parser parse_ipv4 {
   extract(ipv4);
   set_metadata(meta_nat.ipv4_sa,
ipv4.srcAddr);
   set_metadata(meta_nat.ipv4_da,
ipv4.dstAddr);
   set_metadata(meta_nat.tcpLength,
ipv4.totalLen - 20);
   return select(ipv4.protocol) {
      IP_PROT_TCP : parse_tcp;
      IP_PROT_UDP : parse_udp;
      default : ingress;
   }
}
header_type tcp_t {
   fields {
        srcPort : 16;
        dstPort : 16;
        seqNo : 32;
        ackNo : 32;
        dataOffset : 4;
        res : 4;
        flags : 8;
        window : 16;
        checksum : 16;
        urgentPtr : 16;
    }
}
header tcp_t tcp;
parser parse_tcp {
    extract(tcp);
    set_metadata(meta_nat.srcp,
tcp.srcPort);
    set_metadata(meta_nat.dstp,
tcp.dstPort);
    return ingress;
}
field_list tcp_checksum_list {
        ipv4.srcAddr;
        ipv4.dstAddr;
        8'0;
        ipv4.protocol;
        meta_nat.tcpLength;
        tcp.srcPort;
        tcp.dstPort;
        tcp.seqNo;
        tcp.ackNo;
        tcp.dataOffset;
        tcp.res;
        tcp.flags;
        tcp.window;
        tcp.urgentPtr;
        payload;
}
field_list_calculation tcp_checksum {
    input {
        tcp_checksum_list;
    }
    algorithm : csum16;
    output_width : 16;
}
calculated_field tcp.checksum {
    verify tcp_checksum if(valid(tcp));
    update tcp_checksum if(valid(tcp));
}
header_type udp_t {
    fields {
        srcPort : 16;
        dstPort : 16;
        len : 16;
        checksum : 16;
    }
}
header udp_t udp;
parser parse_udp {
    extract(udp);
    set_metadata(meta_nat.srcp,
udp.srcPort);
    set_metadata(meta_nat.dstp,
udp.dstPort);
    return ingress;
}
field_list udp_checksum_list {
        ipv4.srcAddr;
        ipv4.dstAddr;
        8'0;
        ipv4.protocol;
        udp.len;
        udp.srcPort;
        udp.dstPort;
        payload;
}
field_list_calculation udp_checksum {
    input {
        udp_checksum_list;
    }
    algorithm : csum16;
    output_width : 16;
}
calculated_field udp.checksum {
    verify udp_checksum if(valid(udp));
    update udp_checksum if(valid(udp));
}
action _drop() {
    drop();
}
header_type routing_metadata_t {
    fields {
        nhop_ipv4 : 32;
    }
}
action perform_forward_nat(server_ipv4)
{
    modify_field(ipv4.dstAddr,
server_ipv4);
}
table forward_nat {
    reads {
        meta_nat.ipv4_sa : lpm;    // this is
the client ID
        meta_nat.ipv4_da : exact;
    }
    actions {
        perform_forward_nat;
    }
    size: 1024;
}
action perform_reverse_nat(node_ipv4) {
  modify_field(ipv4.srcAddr, node_ipv4);
}
table reverse_nat {
  reads {
    meta_nat.ipv4_sa : lpm;   // this is the
server instance ip
  }
  actions {
    perform_reverse_nat;
  }
}
metadata routing_metadata_t
routing_metadata;

action set_nhop(nhop_ipv4, port) {

modify_field(routing_metadata.nhop_ipv4
, nhop_ipv4);

modify_field(standard_metadata.egress_p
ort, port);
    add_to_field(ipv4.ttl, -1);
}
table ipv4_lpm {
    reads {
        ipv4.dstAddr : lpm;
    }
    actions {
        set_nhop;
        _drop;
    }
    size: 1024;
}
action set_dmac(dmac) {
    modify_field(ethernet.dstAddr, dmac);
}
table forward {
    reads {
        routing_metadata.nhop_ipv4 : exact;
    }
    actions {
        set_dmac;
        _drop;
    }
    size: 512;
}
action rewrite_mac(smac) {
    modify_field(ethernet.srcAddr, smac);
}
table send_frame {
    reads {
        standard_metadata.egress_port:
exact;
    }
    actions {
        rewrite_mac;
        _drop;
    }
    size: 256;
}
control ingress {
    if(valid(ipv4) and ipv4.ttl > 0) {
        apply(forward_nat);
        apply(reverse_nat);
        apply(ipv4_lpm);
        apply(forward);
    }
}
control egress {
    apply(send_frame);
```

## D.	Testbed routing table entries

```
# IPv4 LPM
insert_entry ipv4_lpm 117.45.24.21/16      set_nhop 10.1.0.0   1
insert_entry ipv4_lpm 17.244.34.233/16     set_nhop 11.1.0.0   2
insert_entry ipv4_lpm 147.45.24.21/16      set_nhop 10.2.0.0   3
insert_entry ipv4_lpm 47.244.34.233/16     set_nhop 11.2.0.0   4
insert_entry ipv4_lpm 187.45.24.21/16      set_nhop 10.3.0.0   5
insert_entry ipv4_lpm 87.244.34.233/16     set_nhop 11.3.0.0   6
insert_entry ipv4_lpm 217.45.24.21/16      set_nhop 10.4.0.0  7
insert_entry ipv4_lpm 117.244.34.233/16    set_nhop 11.4.0.0  8
insert_entry ipv4_lpm 247.45.24.21/16      set_nhop 10.5.0.0  9
insert_entry ipv4_lpm 147.244.34.233/16    set_nhop 11.5.0.0  10
#entry for the control plane controller - dns_load_balancer
insert_entry ipv4_lpm 50.54.33.36/32       set_nhop 12.1.0.0  11
# forward
insert_entry forward 10.1.0.0  set_dmac bb:bb:bb:bb:bb:00
insert_entry forward 11.1.0.0  set_dmac bb:bb:bb:bb:bb:01
insert_entry forward 10.2.0.0  set_dmac bb:bb:bb:bb:bb:02
insert_entry forward 11.2.0.0  set_dmac bb:bb:bb:bb:bb:03
insert_entry forward 10.3.0.0  set_dmac bb:bb:bb:bb:bb:04
insert_entry forward 11.3.0.0  set_dmac bb:bb:bb:bb:bb:05
insert_entry forward 10.4.0.0  set_dmac bb:bb:bb:bb:bb:06
insert_entry forward 11.4.0.0  set_dmac bb:bb:bb:bb:bb:07
insert_entry forward 10.5.0.0  set_dmac bb:bb:bb:bb:bb:08
insert_entry forward 11.5.0.0  set_dmac bb:bb:bb:bb:bb:09
insert_entry forward 12.1.0.0  set_dmac bb:bb:bb:bb:bb:20
# send_frame
insert_entry send_frame 1'2  rewrite_mac aa:aa:aa:aa:aa:00
insert_entry send_frame 2'2  rewrite_mac aa:aa:aa:aa:aa:01
insert_entry send_frame 3'2  rewrite_mac aa:aa:aa:aa:aa:02
insert_entry send_frame 4'2  rewrite_mac aa:aa:aa:aa:aa:03
insert_entry send_frame 5'2  rewrite_mac aa:aa:aa:aa:aa:04
insert_entry send_frame 6'2  rewrite_mac aa:aa:aa:aa:aa:05
insert_entry send_frame 7'2  rewrite_mac aa:aa:aa:aa:aa:06
insert_entry send_frame 8'2  rewrite_mac aa:aa:aa:aa:aa:07
insert_entry send_frame 9'2  rewrite_mac aa:aa:aa:aa:aa:08
insert_entry send_frame 10'2 rewrite_mac aa:aa:aa:aa:aa:09
insert_entry send_frame 11'2 rewrite_mac aa:aa:aa:aa:aa:09
```

# References

[1] M. Valenti, B. Bethke, D. Dale and A. Frank, "The MIT Indoor Multi-Vehicle Flight Testbed," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, Roma, 2007.

[2] W3C, "Amaya Home Page," [Online]. Available: https://www.w3.org/Amaya/Overview.html.

[3] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci and I. Seskar, "GENI: A federated testbed for innovative network experiments," *Comput. Netw.,* vol. 61, no. March, 2014, pp. 5 - 23, 2014.

[4] B. Lantz, B. Heller and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Monterey, California, ACM, 2010, pp. 19:1-19:6.

[5] Internet Industrial Consortium, "Testbeds | Internet Industrial Consortium," [Online]. Available: http://www.iiconsortium.org/test-beds.htm.

[6] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," *SIGOPS Oper. Syst. Rev.,* vol. 36, no. SI, pp. 255--270, 2002.

[7] F. Keti and S. Askar, "Emulation of Software Defined Networks Using Mininet in Different Simulation Environments," in *2015 6th International Conference on Intelligent Systems, Modelling and Simulation*, Kuala Lumpur, 2015.

[8] R. L. S. d. Oliveira, C. M. Schweitzer, A. A. Shinoda and L. R. Prete, "Using Mininet for emulation and prototyping Software-Defined Networks," in *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on* , Bogota , 2014.

[9] S.-Y. Wang, "Comparison of SDN OpenFlow network simulator and emulators: EstiNet vs. Mininet," in *2014 IEEE Symposium on Computers and Communications (ISCC)*, Funchal, 2014.

[10] S.-Y. Wang, C.-L. Chou and C.-M. Yang, "EstiNet openflow network simulator and emulator," *IEEE Communications Magazine,* vol. 51, no. 9, pp. 110-117, 2013.

[11] ESnet; Lawrence Berkeley National Laboratory, "iPerf - The network bandwidth measurement tool," ESnet; Lawrence Berkeley National Laboratory, [Online]. Available: https://iperf.fr/. [Accessed January 2016].

[12] P4, "p4lang/behavioral-model," [Online]. Available: https://github.com/p4lang/behavioral-model/tree/master/targets/simple_router.

[13] T. Grotker, System Design with SystemC, Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[14] Oracle, "VirtualBox - Oracle VM VIrtualBox," [Online]. Available: https://www.virtualbox.org/wiki/VirtualBox.

[15] J. Sommers and P. Barford, "Self-configuring Network Traffic Generation," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, Taormina, Sicily, Italy, ACM, 2004, pp. 68-81.

[16] Open vSwitch, "Open vSwitch," [Online]. Available: http://openvswitch.org/.

[17] Barefoot Networks, Inc., " behavioral-model/1sw_demo.py at master," Barefoot Networks, Inc., [Online]. Available: https://github.com/p4lang/behavioral-model/blob/master/mininet/1sw_demo.py.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.,* vol. 44, no. 3, pp. 87-95, 2014.

[19] Cisco, "Cisco Visual Networking Index: Forecast and Methodology, 2015–2020," 2016.

[20] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat and W. Dabbous, "Network Characteristics of Video Streaming Traffic," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, Tokyo, Japan, 2011.

[21] A. Veres, K. S. Molnár and G. Vattay, "On the Propagation of Long-range Dependence in the Internet," *SIGCOMM Comput. Commun. Rev.,* vol. 30, no. 4, pp. 243-254, 2000.

[22] T. D. Dang, B. Sonkoly and S. Molnar, "Fractal analysis and modeling of VoIP traffic," in *Telecommunications Network Strategy and Planning Symposium. NETWORKS 2004, 11th International*, 2004.

[23] TCPDump, "TCPDUMP & LibPCAP," [Online]. Available: http://www.tcpdump.org/index.html.

[24] Open Networking Foundation, "Software-Defined Networking (SDN) Definition - Open Networking Foundation," [Online]. Available: https://www.opennetworking.org/sdn-resources/sdn-definition.

[25] H. Song, "Protocol Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, Hong Kong, China, ACM, 2013, pp. 127--132.

[26] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, Hong Kong, China, ACM, 2013, pp. 99-110.

[27] S. Abdi, U. Aftab, G. Bailey, B. Boughzala, F. Dewal, S. Parsazad and E. Tremblay, "PFPSim: A Programmable Forwarding Plane Simulator," in *Proceedings of the 2016*

*Symposium on Architectures for Networking and Communications Systems*, Santa Clara, California, USA, ACM, 2016, pp. 55--60.

[28] S. Abdi, U. Aftab, G. Bailey, B. Boughzala, F. Dewal, S. Parsazad and E. Tremblay, "pfpsim/simple-npu," [Online]. Available: https://github.com/pfpsim/simple-npu.