December 2020

Network Virtualization and Emulation using Docker, OpenvSwitch and Mininet-based Link Emulation

Narendra Prabhu

NETWORK VIRTUALIZATION AND EMULATION USING DOCKER, OPENVSWITCH AND MININET-BASED LINK EMULATION

A Thesis Presented

by

NARENDRA PRABHU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2020

Electrical and Computer Engineering

NETWORK VIRTUALIZATION AND EMULATION USING DOCKER, OPENVSWITCH AND MININET-BASED LINK EMULATION

A Thesis Presented

by

NARENDRA PRABHU

Approved as to style and content by:

Russell Tessier, Chair

Aura Ganz, Member

Tongping Liu, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to thank Prof. Russell Tessier for his constant guidance and support on this thesis. His valuable insights were an integral part of the thesis work. I would also like to thank Xuzhi Zhang for his help and guidance for the duration of this thesis. I extend my gratitude to Prof. Aura Ganz and Prof. Tongping Liu for serving on my thesis committee and providing valuable suggestions. Lastly, my lab members, George Provelengios and Aiden Gula for providing nitty-gritty technical suggestions whenever I required.

ABSTRACT

NETWORK VIRTUALIZATION AND EMULATION USING DOCKER, OPENVSWITCH AND MININET-BASED LINK EMULATION

SEPTEMBER 2020

NARENDRA PRABHU

B.TECH., NATIONAL INSTITUTE OF TECHNOLOGY GOA

M.S.E.C.E, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

With the advent of virtualization and artificial intelligence, research on networked systems has progressed substantially. As the technology progresses, we expect a boom in not only the systems research but also in the network of systems domain. It is paramount that we understand and develop methodologies to connect and communicate among the plethora of devices and systems that exist today. One such area is mobile ad-hoc and space communication, which further complicates the task of networking due to myriad of environmental and physical conditions. Developing and testing such systems is an important step considering the large investment required to build such gigantic communication arrangements. We address two important aspects of network emulation in this work. We propose a network emulation framework, which emulates the functioning of a hierarchical software defined network. One such use-case is described using a mobile ad-hoc network (MANET) topology within a single system by leveraging contemporary network virtualization technologies. We

present various aspects of the network, such as the dynamic communication in the software domain and provide a novel approach to build upon existing emulation techniques. The second part of the thesis presents a dynamic network link emulator. This emulator enables suitable link property re-configurations such as bandwidth, delay and packet loss for networked systems using simulation software. We characterize the results of tests for the link emulation using a hardware and software testbed. Through this thesis, we aim to make a small yet crucial contribution to the niche area of software defined networks.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Network Emulation

The need for network emulation has risen due to the exorbitant costs of real-time hardware testing and potential system redesigns. Conducting live field experiments for wireless mobile systems incurs monetary and logistical issues to administer mobile platforms and support equipment, network automation and antennas. Hence, it is critical to gauge the performance and usability of an application in a software environment before making changes or additions to a system. Network emulation is an established technique to test the operation and performance of applications on a virtual network [1][62]. Mobile ad-hoc networks complicate testing as a result of time-varying changes in topology and communication channels. A mobile ad-hoc network (MANET) emulator must accomplish accurate portrayal and assess the exclusive features that a mobile ad-hoc network presents [75]. These features include smart peer-to-peer communication protocols and frequent link reconfiguration as a result of geo-positional changes. Unlike simulators, which perform tasks in an abstract manner to manifest the behavior of a network and corresponding components, an emulator can mimic the behaviour to functionally supplant it [78]. The specific contribution of this thesis in this area is the design and implementation of two distinct emulation frameworks. The first accounts for mobile node encapsulation and reconfigurability of a MANET using Docker [23], a container-based virtualization environment. The latter emulator addresses link property re-configuration in a network using Mininet [10], a network simulation environment.

1.2 Docker Based Emulation

The design space for network emulation is vast and existing techniques [14][69] have explored several simulation software implementations to represent complex networks. The requirements for a mobile ad-hoc network requires careful consideration to achieve an accurate implementation [42]. Our aim for this project is to build an emulator with the capability to run the same software that is embedded in the final implementation of the digital hardware. Thus, we require a operating system level of abstraction. In this work, we use Docker-based containers [50] to represent a mobile node in a MANET and the components it encompasses. The need for Docker is accentuated by the fact that it provides the bare-bones structure with the right amount of configurability, isolation and easy migratability [11]. Docker enables the rapid deployment of custom environments in containers by abstracting the operating system (OS). A contribution of this thesis will be to build a network emulator using Docker containers that leverages the isolation of nested Docker containers to represent the constituents of a network node. To better evaluate our approach against contemporary emulators, we juxtapose with respect to an existing emulation platform based on Docker containers, ContainerNet [59]. The design and architecture of a sample software-defined network (SDN) topology is illustrated using NestedNet and then a comparative study is charted out with respect to ContainerNet for the same. ContainerNet is not designed for nested/hierarchical containers for emulation purposes. This issue reduces its ability to implement node sub-components. We perform evaluations and illustrate the advantages and trade-offs of using nested containers in NestedNet with respect to single-layered ContainerNet.

1.3 Dynamic Link Emulation

Network emulators incorporate a varying amount of standard network attributes into their designs. The ability to vary link properties is an important feature for any

emulator based framework. It is imperative to have provisions to enable dynamic link configurations for a mobile ad-hoc network that encompasses changeable links. Herrscher et al. [37] presented an approach to support network configuration using XML description language-based scenario modelling. This work uses NISTNet [14] to perform traffic shaping for link reconfiguration.

Some of the most important network attributes for emulation are: round-trip time across the network (latency), the amount of available bandwidth, a given degree of packet loss, corruption and modification of packets, and/or the severity of network jitter. In this thesis, we present a Mininet based framework using OpenvSwitch [28] and Ryu-Router [16] to implement a dynamic link emulator. This link emulator is unrelated to the MANET emulator and is developed as a general purpose link emulator. It is primarily focused on a client-server topology framework. However, use-cases of such a link emulator may arise in future MANET applications.

In this work, we introduce tunable link characteristics such as bandwidth, delay and packet loss based on the source and destination IP address of nodes. Our implementation leverages the traffic control capabilities of Linux for a series of links.

1.4 Thesis Outline

The thesis work is divided into two major parts. The first part includes the development of an emulation structure using nested Docker containers for hierarchical software-defined networks. The second part involves the development of a system that allows for selective dynamic configuration of network link attributes. The document is organized as follows. Chapter 2 discusses previous related research and associated background. Chapter 3 describes the design and implementation of the Docker-based emulator for a mobile ad-hoc network use-case. It also explains the components of a mobile ad-hoc network, the components of a mobile node and implementation details of component software. Furthermore, the operating mechanisms and scalability met-

rics of the approach are elaborated. Evaluation against a contemporary Docker-based emulator, ContainerNet, is performed through a series of experiments. The results corroborate the advantages of the proposed emulator which uses a nested Docker container approach. Chapter 4 presents a separate framework for dynamic and selective configurations of link attributes. A detailed explanation of the parameters that are considered for the implementation is followed by the description of roles of Mininet, Open vSwitch and Ryu-router in the implementation. The testbed is used to perform a system-level evaluation. A software evaluation with more complex data streams is provided to conclude the chapter. Chapter 5 summarizes findings from both emulation systems and Chapter 6 describes potential directions for future work.

CHAPTER 2

BACKGROUND

2.1 Network Emulation

Network emulation has been explored since early 1980's to aid research and the teaching of distributed operating systems [4]. However, the emergence of virtualization technologies allowed for the possibility of encasing an entire emulated computer network in a single machine by leveraging virtual machines (VM). Emulation, unlike simulation [78] which uses abstractions, represents a physical network consisting of physical devices, applications, and products and services in a realistic, manageable and mutable platform.

Early emulators, including DummyNet [62] and CORE [1], often used a dedicated testbed or connections from a system under test to specialized hardware devices. Initial research was conducted on the emulation of wired networks and, some time later, wireless networks. The presence of multiple factors such as physical conditions, environmental variation and link characteristics made wireless network virtualization and emulation challenging. Noble et al. [56] introduced *trace modulation* in the late 1990's to recreate the end-to-end characteristics of a real-time wireless network. Machine virtualization technologies, such as VMware [64] and QEMU [6] and operating system virtualization tools, such as Xen [5] and KVM [35] drove the domain of network emulation using virtualization. The arrival of Docker [25] [50] provided a lightweight alternative for VMs without the use of a hypervisor. Recent work has focused on using Docker containers for emulating wireless networks and distributed

ad-hoc communication. To et al. [74] created Dockemu to emulate general-purpose wired and wireless networks.

Software-defined networks benefited considerably from network emulation research. Mininet is a popular network emulator used for emulation and prototyping software-defined networks [17]. Fontes et al. [26] created "Mininet-Wifi", an emulator for wireless OpenFlow/SDN scenarios facilitating high-fidelity experiments to replicate real networking environments. Another recent effort [59] uses Docker to create an emulator for network function virtualization (NFV) by extending Mininet.

2.2 Mobile Ad Hoc Network Emulators

An ad-hoc network encompasses arbitrarily-connected devices that communicate with each other. A MANET is typically characterized by the presence of multiple mobile hosts connected via direct or indirect links such that each host is capable of transmitting to all hosts within its transmission range using wireless communication. Mobile hosts can act as intermediate multi-hop routers establishing indirect links between incommunicable hosts. This approach allows for the creation of a scenario-based connected network of mobile hosts in a given deployment area.

Liu et al. [47] outlines the latest developments in vehicular ad-hoc networks (VANETs) and the state-of-the-art routing protocols used in VANETs. Roh et al. [63] propose a MANET architecture for an unmanned autonomous maneuver network. Search and rescue and disaster relief operations [61] are other prominent areas of MANET deployment. Such areas require quick response and fast establishment of communications and troop deployments in unknown environments. Macker et al. [48] provide software support for network node motion modeling.

Given the critical nature of these applications, foolproof testing and evaluation are required to avoid failures and optimize technology decision-making. A multitude of unpredictable environmental and physical conditions further complicate the develop-

ment and testing of real world systems. Considering the large investment required to build and modify real-world systems, it is imperative to have a scalable environment that can emulate functionality accurately and provide performance measurements to predict the impact of change.

Recent work has produced several testbeds for MANET modelling and testing. Simulation tools have been used to evaluate implementation scenarios. A network simulation tool called OMNeT++ [71] was used to test communication software in a controlled environment. Tuteja et al. [75] utilized the NS2 simulation tool to analyze different routing protocols for a MANET.

Emulators present synthetic network environments that can be parametrized to reproduce an original or fictitious network. Sharma et al. [69] juxtapose different existing emulation techniques and assess their pros and cons. Nordstrom et al. [57] developed a large scale multi-hop wireless ad-hoc network testbed based on Linux. They use scripted scenarios implemented through a graphical user interface (GUI) to coordinate node movements. Suri et al. [73] use CORE and EMANE to emulate realistic military scenarios. The use of Docker containers has also been explored by Alvarez et al. [3]. However, this implementation uses the network simulator NS3 with Docker containers for a specific hybrid monitoring algorithm. In this thesis, we provide a Linux-based emulator framework that can effectively represent a MANET. We test the system using pre-designed algorithms and support the execution of application source code. It enables the testing of hierarchical SDN applications with environment isolation, including intra-node isolation.

2.3 Link Property Reconfiguration for Emulation

To detect and compare performance measurements of network protocols and applications, it is necessary to leverage a realistic network environment. Network emulation techniques offer an extensible and custom approach to replicate network properties.

Software defined networks use such environments to study link saturation and aggregation. Previous work in emulation tools focuses on the emulation of network properties at a single network interface [36]. Beshay et al. [7] emulated networks on a machine using Linux-based traffic shaping across links to allow for easy experimentation.

Mininet is widely used for link-based simulation. It presents an API for Quality of Service (QoS) configuration [81]. Lantz et al. [46] used Linux-based network techniques in Mininet to tune link properties. Ryu, a SDN controller, is usually used in conjunction with an Open vSwitch in Mininet for software defined network emulation arrangements. Al-Somaidai et al. [2] studied the effects of different Linux scheduling algorithms with a Ryu controller, Open vSwitch and Mininet. This thesis leverages Linux-based network packet scheduling algorithms for link property configuration. The traffic control (TC) configuration API of Linux OS has been studied for link manipulation for simulations [41]. The Hierarchical Token Based (HTB) Queuing Technique [21] and Hierarchical Fair Sequence Curve (HFSC) Queuing Technique [60] are the most popular scheduling algorithms used by a Linux kernel for QoS-based resource allocation.

In this thesis, we propose a tool facilitating realistic emulation of network links. We strive to incorporate a mix of both, real machines and software models on a full-fledged network using Mininet, Open vSwitch and Ryu. We show how certain network links can be combined to allow a single point reconfiguration of link properties using traffic control techniques. Multiple points of reconfiguration can also be made available, depending on the levels of reconfiguration that are required. Changing link properties such as those from mobile communications and physical limitations are modelled. This work can be directly extended to mobile ad-hoc networks which have dynamically changing properties between links based on geo-location.

CHAPTER 3

MOBILE AD-HOC NETWORK EMULATION APPROACH

3.1 Introduction

The interaction of MANET components leads to a transitional network. Moreover, the presence of heterogeneous nodes requires accurate isolation for each host. Such complexity requires careful design choices for the development of a realistic emulator. A simulator may not accurately represent the real-time functioning of such a network, while an emulator can act as a prototype for physical mobile nodes by leveraging a full OS. Moreover, each MANET node is a complex network node with different sub-components that form an intra-node environment. Accurately emulating these intra-node devices, which are real hardware devices is an important aspect to enable hardware interoperability. In this chapter, an emulation environment that can support an extensible and robust ad-hoc network is described. A nested architecture of Docker containers is leveraged to support isolation of the intra-node environment. Finally, an arbitrarily connected network of mobile hosts with decentralized data and control flows via direct or indirect inter-node links is presented.

3.2 Design

A MANET emulator is characterized by specific functionalities. All mobile hosts are configured as quasi-centralized nodes that can support an infusion of periodic information related to its traffic modes, geographical position and the environment. Each mobile host should be isolated and encompass parts that represent hardware

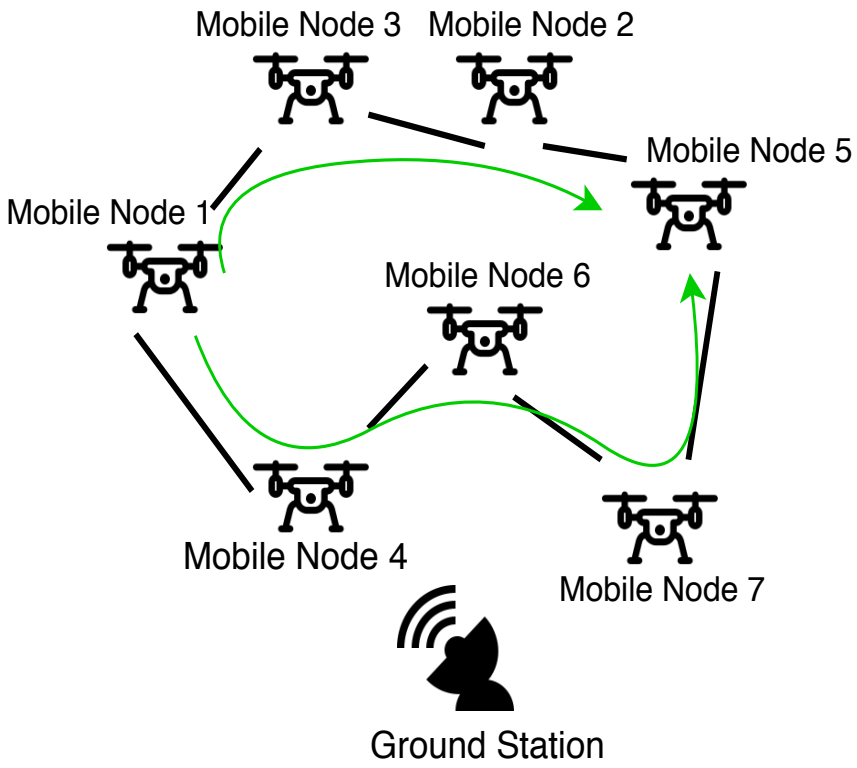

Figure 3.1. Decentralized collection of mobile nodes with multiple direct and indirect paths. Mobile Node 1 and Mobile Node 5 can communicate via two indirect paths indicated with green arrows. The ground station interacts with the nodes.

components. The interconnection and link establishment between hosts must be performed by a separate entity that is responsible for providing information about possible links and their characteristics. Software libraries control node mobile trajectories and distribute control data. Our emulator supports the use of user programs in mobile host nodes.

3.2.1 Components of a MANET

A MANET consists of mobile nodes that interact with each other. An illustration in Figure 3.1 shows a simple network of six mobile nodes. The network topology is variable depending upon the respective movement of each individual host. Each mobile node can communicate with peers via direct links or indirect routes. In Figure

3.1, we observe two paths from Mobile Node 1 to Mobile Node 5 (represented by green paths). A node may use decision algorithms to locate a faster and more efficient route. The identified path may not be valid over a period of time due to topology changes, and new route options may be needed. The network may have a ground station that relays link data. The network is formed when multiple hosts connect to peers subject to the number of communication devices they possess.

In this work, an initialization phase is used in which all mobile nodes are set up and made ready to transmit or receive data. A mobile node may then retrieve information about visible peer nodes and attempt to establish connections with them. This retrieval may happen via individual scanning or via an agent external to the network that has a global view of the nodes.

The hosts decide whether to connect to another node based upon resource availability. A loss of visibility can cause a disconnection between MANET links. This action causes mobile hosts to try and make new connections with available nodes via accessible links. Both direct and indirect links may be broken. We illustrate the framework in Figure 3.2. The Mobile Ad-hoc Emulator contains multiple mobile nodes with functional units. Each node includes a controller (brain) of the host and several interfaces to correspond with peer hosts. The *communication assistant* is an emulation assistant that has a global view of the network. The assistant contains visibility information for each host and manages link rearrangement. The transfer of information occurs between the controller unit of each host and the communication assistant via an emulator interface. A command line user interface (UI) generates the network framework and supports emulator changes.

In Section 3.3, our emulator design and its support for MANETs is described. The representation of nodes and links is provided. Software that performs the role of the agent with a global network view to dictate link visibility changes is also detailed. Finally, a test mechanism used to self-heal the network is discussed in Section 3.4.

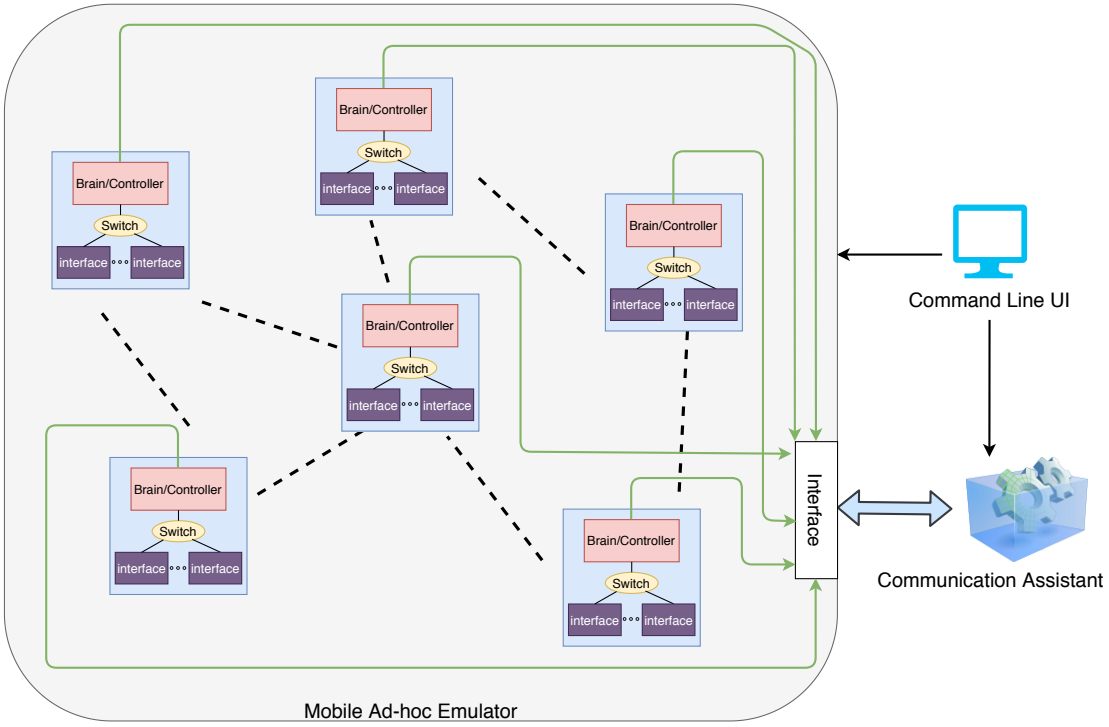

Figure 3.2. Framework design of an emulated MANET. The communication assistant communicates with individual controllers to distribute and retrieve link/visibility information. A command line UI allows a user to generate and edit emulator parameters.

3.2.2 Components of a Mobile Node

Components in mobile nodes can be categorized into two types. A *brain/controller* serves as a computational and control hub. *Transceivers* are endpoints for communication with other nodes (Figure 3.3). The brain makes decisions for the node, including storing and distributing routing information, keeping track of available resources, exchanging information with the communication assistant and evaluating the visibility of other nodes. We call this component the Global Network Access Brain (GNAB).

The transceivers are called *interface physical devices* (iPHYs) in this document. These devices communicate with transceivers on other hosts. The iPHYs take commands from the GNAB and forward data over established links. iPHY types may

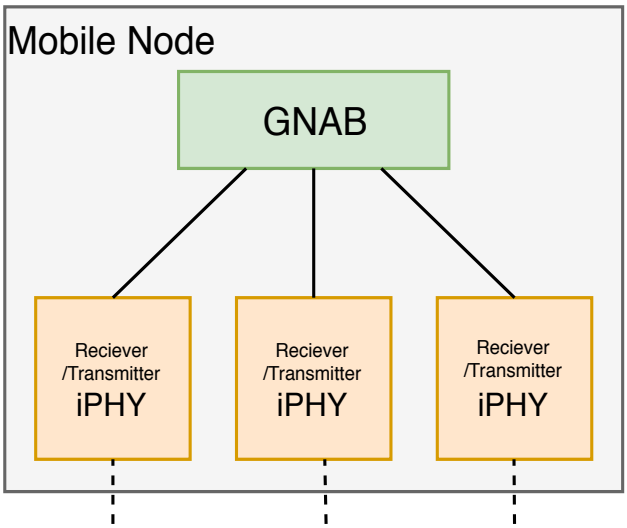

Figure 3.3. Components of a mobile node. A GNAB (Green) is the computational hub. iPHYs (orange) are transceivers that interface with send/receive units (iPHYs) of other mobile nodes.

differ depending on mobile host type and transmission modes. For example, radio frequency (RF) based iPHYs use radio waves to communicate control and data messages with ground stations. Laser-based iPHYs transfer data between mobile nodes.

3.3 Implementation

Virtualization technology is used to model mobile hosts. Docker containers are used due to their ability to support OS-level virtualization with fast startup and lower memory overhead compared to virtual machines. Docker uses containerization to confine and condense each mobile host into an isolated container. Docker supports a full Linux OS and has compatibility with a variety of software tools such Python [29], Open vSwitch and network-tools [19].

The Linux-based Ubuntu 16.04 operating system is used for network building and container-based tools. It supports in-built virtualization and networking tools, such as virtual Ethernet (Veth) links [18] and namespacing [20]. It is possible to incorporate dependencies such as virtual switches, kernel configurations and network utilities into

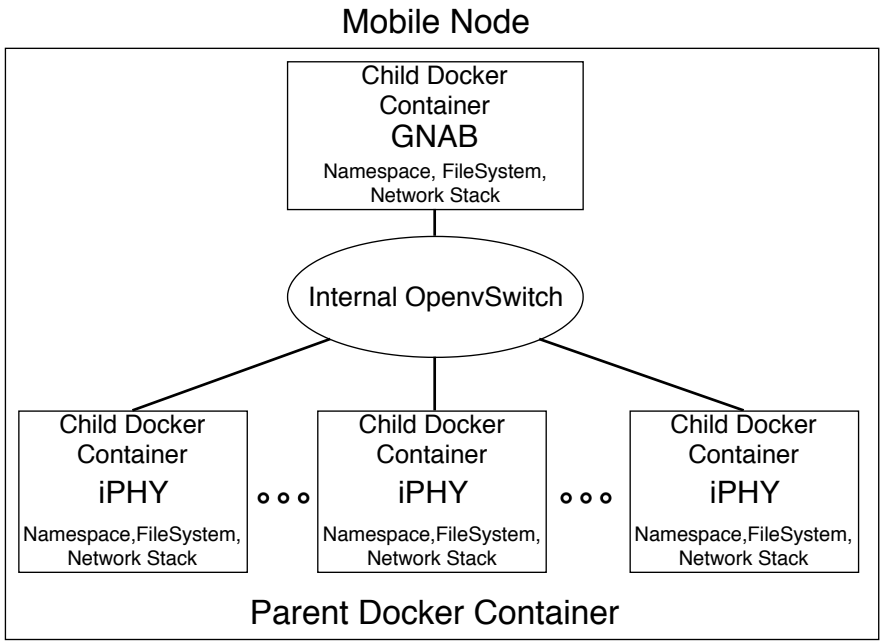

Figure 3.4. A nested Docker setup for an individual mobile node. Each container has a dedicated OS and network resources.

the Linux OS. The communication assistant collaborates with the GNABs to establish links. The CA is written in Python 2.7 to simplify communication with the mobile host containers. A command-line interface implemented with a bash script accepts user input and starts the emulation.

3.3.1 Virtualization with Docker

Docker helps a user package, deploy and run applications using containers [23]. Containers are widely used to package applications with libraries and dependencies. An important emulation property is support for compute server migration. Docker provides capabilities for container building using images that are portable across platforms. This feature allows for code portability across platforms and emulator migration to a variety of workstations.

A novel aspect of our work is the implementation of a nested Docker container setup to isolate each mobile host and its individual components. Docker supports

a *privileged* mode that allows a container to operate as a standalone machine. It allows for the creation of containers within a Docker container to generate a nested setup. Such a setup is called a *Docker-in-Docker (dind)* [22] implementation. This characteristic is most important for representing large systems with smaller operational units, as is the case for our mobile host. Figure 3.4 shows a nested Docker setup with one external parent container representing a mobile node encompassing several child containers. The child Docker containers are isolated from each other and serve different functions. One Docker container may serve the role of a GNAB while multiple containers represent iPHYs. Child containers are connected via an *internal* Open vSwitch.

To summarize, Docker ensures that an application and its resources (e.g. file system and network stack) are segregated in a container that is isolated from other containers. Thus, each child container that runs a unique application can be rebooted independently and have distinct root access, users, IP addresses, memory, processes, files, applications, system libraries and configuration files. This feature allows for clean application removal and modifications for unique individual components.

3.3.2 Network Namespaces and Virtual Ethernet Interfaces

Namespaces and interfaces represent machines and interconnect during virtual network construction. To support the execution of multiple applications or services inside mobile nodes, isolation is essential for security, stability and manageability. Namespacing tools in Docker allow for the control of workstation resources by each process.

A Linux system boots up with a process which has an associated ID (i.e. PID 1). This process is instantiated as the root of a process tree and all the other processes start below the root. The root process administers tasks by performing maintenance work and starting daemons and services. A namespace allows the user to spin off a

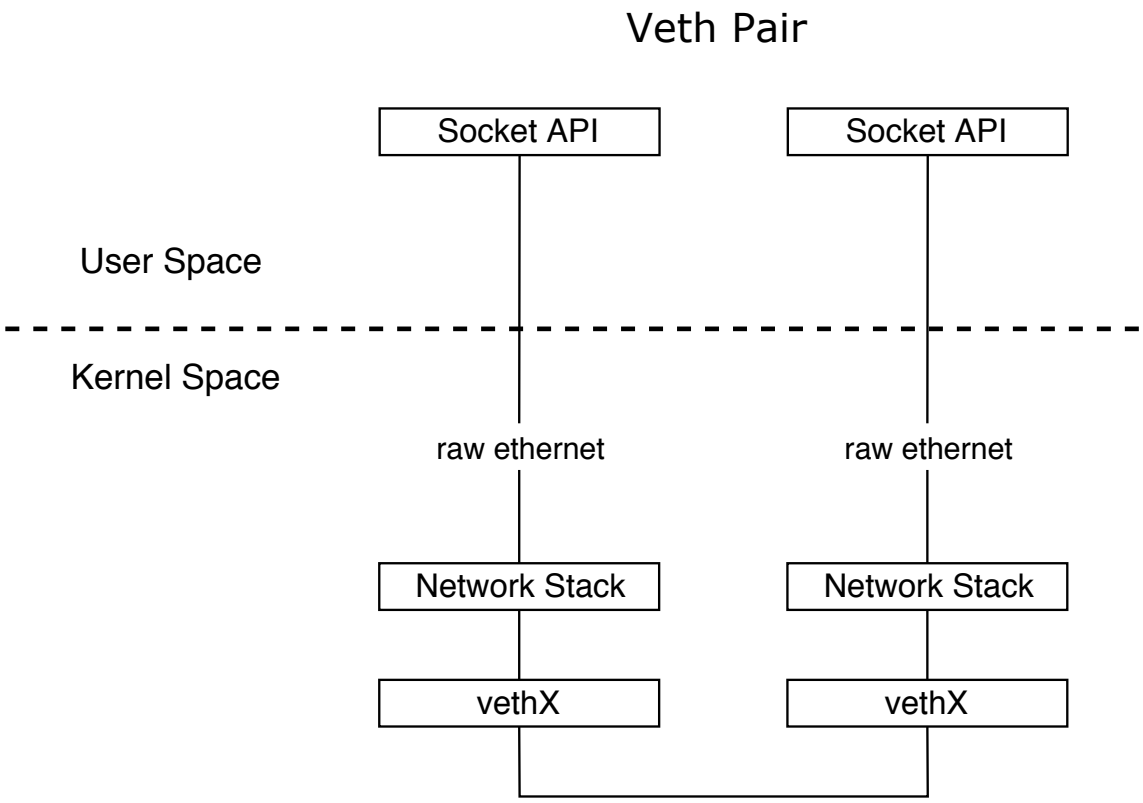

Figure 3.5. A network stack vantage point for virtual Ethernet devices. Veth devices behave as a pair of virtual interfaces by presenting an API to the user. The two ends are connected via the OS network stack in kernel space.

new tree with a specific PID 1 process such that there can be multiple child processes confined to the namespace. The process that creates this root for a namespace remains in the parent tree of the OS. With namespace isolation, processes in the child namespace are cognizant of the parent process's existence. The parent namespace has knowledge of the processes in the child's namespace.

A separation of applications and planes of communication between MANET components are needed for the emulator. This separation requires a unique network stack for each component. A network namespace allows each process in a namespace to interact with an entirely different set of networking interfaces including the loopback interface. This setup can be achieved with supplementary *virtual* network interfaces

that can interact across namespaces. Ethernet bridges can route packets between different namespaces to provide a networked ecosystem with isolated and virtualized machines. Docker provides an interface for these configurations. To establish the connections between namespaces, virtual Ethernet devices are used. The Veths can be identified as the virtual version of physical Ethernet cables used to interconnect physical devices.

Veth devices are constructed in pairs of connected virtual Ethernet interfaces. The pair acts like a virtual tunnel for network packets, as shown in Figure 3.5. Data are sent between devices without intervention by the network stack of the OS kernel. Each end of a Veth pair acts as a standalone network device. A Veth device can be interpreted as a virtual network interface through the socket API presented to the user. Veths are used to either interconnect virtual containers or make connections between containers and virtual bridges (e.g. Open vSwitch).

We use Linux-based Docker containers to construct a virtual network. Each network namespace is used to represent a GNAB, iPHY and other components of a mobile node.

3.3.3 Open vSwitch

Physical connections between GNABs and iPHYs inside nodes are fixed, preventing link loss. These connections can support SDN protocols that separate the data and control planes. When a GNAB wants to send a control message to connect the node to a remote iPHY in another node, it uses a control plane connection. When data must be sent to another node, a data plane is needed. A software-controller switch with packet switching capabilities is needed to provide this capability.

Open vSwitch (OVS) supports a number of features that allow a virtual network environment to respond and adapt to changing requirements. An OVS aids traffic forwarding between containers on the same host and on the same physical network.

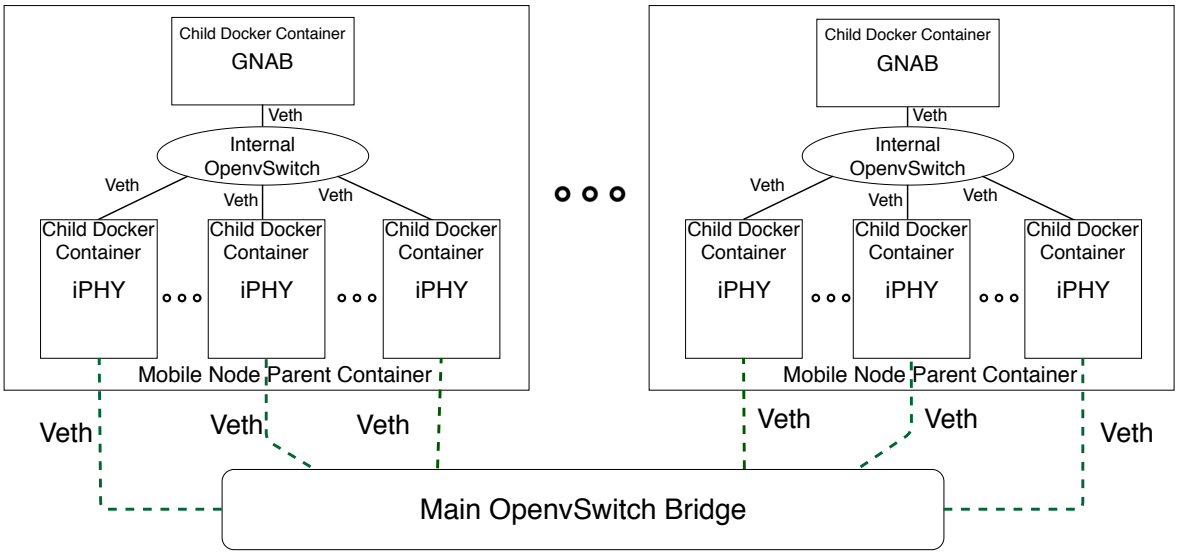

Figure 3.6. A two node MANET implemented in the emulator. The GNAB and iPHYs are nested child Docker containers connect via and internal OpenvSwitch. iPHYs are connected to the main OVS via Veth links.

Open vSwitch source code is written in platform-independent C and can be ported to many compute environments, including Linux-based virtualization environments. It is compatible with Docker and several popular SDN controllers such as POX [33], NOX [53] and Ryu [16].

An OVS can act as a Layer 2 switch for data forwarding. Multiple ports can be used to connect to the containers. Each iPHY of network nodes is connected to a *Main Open vSwitch bridge* via Veth links to individual ports (Figure 3.6). This OVS maintains a flow table with information about inter-port packet forwarding. This feature is used to alter the flow table whenever a link is added or deleted between two nodes. A flow table rule addition (deletion) depicts the gain (loss) of a link. The flow table is used to forward control or data packets.

As shown in Figure 3.6, an internal OVS is used to interconnect components in each node. The GNAB and iPHY containers are connected by individual Veth pairs to an OVS bridge. As shown in Figure 3.4, the GNAB and iPHYs are child containers

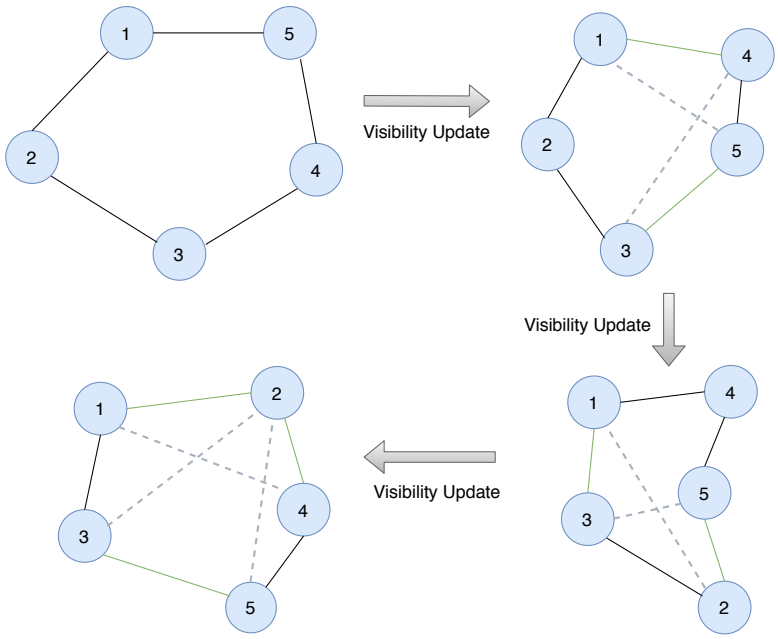

Figure 3.7. An example of visibility updates in a five node cluster. Each mobile node needs to reestablish its links.

embedded within a mobile node container. One end of a Veth pair is added as a port on the bridge while the other end is associated with the network namespace corresponding to a container.

3.3.4 Communication Assistant

The communication assistant (CA) controls node movement, link loss and communication processes for the emulated MANET. The CA assigns an ID to each node for identification. This ID is used for node identification and in determining node visibility. The communication assistant tracks the location and position of each host in the emulated system. For example, for a satellite system, the position could be the orbital location.

A mobile node is subject to dynamic positional changes and thus, peer node visibility may change over time. A change of position may cause a loss of connection with nearby mobile peers relative to their position. The CA intermittently updates the visibility information for each node. This information includes a list of IDs of

other nodes that could communicate with the node. The CA is a multi-threaded application implemented in Python 2.7 that uses multiple APIs (REST API [24], OpenFlow [49]). In our emulation environment, the visibility information is provided to the CA in a JavaScript Object Notation (JSON) [40] file. The CA parses this file and sends updates to nodes at pre-specified time steps.

Any link modification by the CA due to mobility changes is reported to the GNABs in the nodes. The CA maintains a global view of node connectivity and communication resource (iPHY) availability. If a mobile node desires to connect to another node, it must make a request to the CA to determine if a connection is possible. The CA stores visibility information in a global table and is able to check if connectivity is possible. The CA will respond affirmatively or negatively to the request by the GNAB. The CA configures the links using the main Open vSwitch Bridge, shown in Figure 3.6. It communicates directly with the OVS to set up or break port-to-port connections. A network modification triggers the CA to change the flow table in the OVS to include any connectivity changes.

3.3.5 Visibility Graph Generator

A visibility graph generator was developed in Python 3 language that can create custom JSON files defining the topology over a period of time. Graphs can be generated for any number of nodes with custom names, position values and time durations. The line of sight (LOS) between nodes is set as true or false by examining the *Euclidean* distance between each node. Other approaches for LOS could also be supported.

3.3.6 Command-Line Interface

Before nodes can be instantiated at the beginning of the emulation, information about the number of mobile nodes and the number of components within the nodes during the emulation run is needed. A command line interface (CLI) implemented

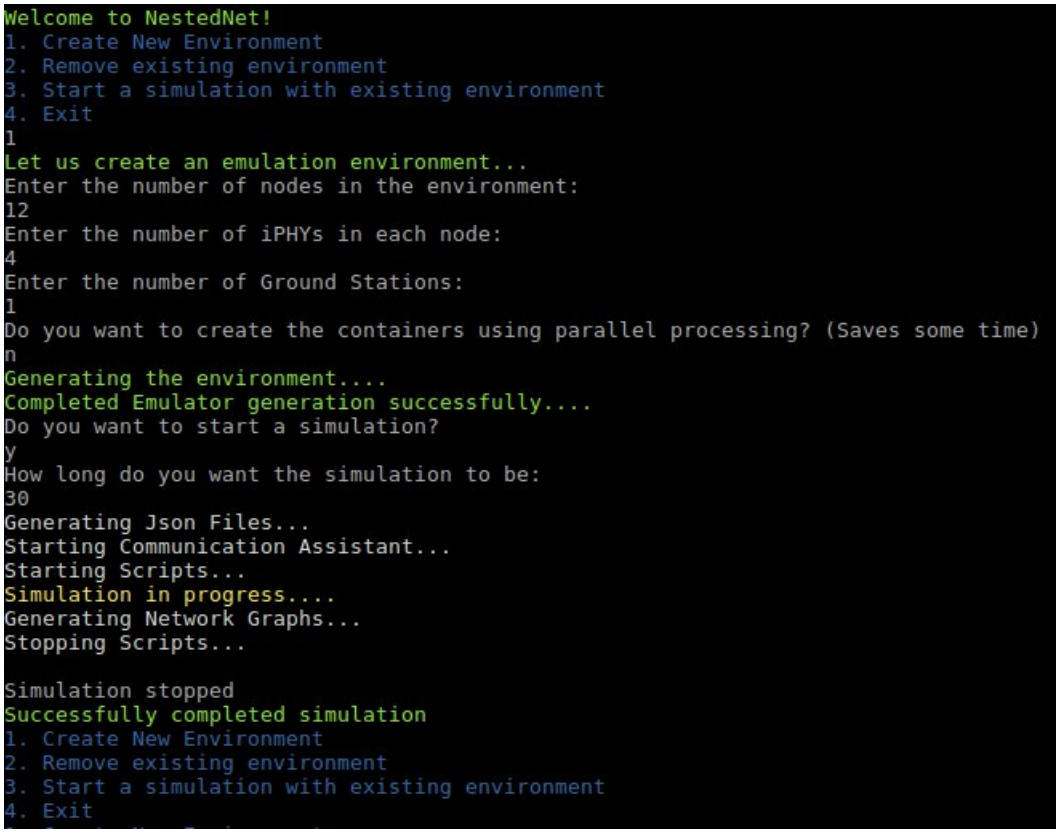

Figure 3.8. Command line interface for the MANET emulator (NestedNet). The CLI provides options to add/delete or run emulations. The number of nodes can be specified to generate a MANET topology.

with a bash shell script takes user defined input to determine the size of the network. The UI enables a user to build and run user-specific functions within GNABs and iPHYs. Scripts are also present to delete the emulator, swap new code into the components and fetch the latest information about connectivity.

A Dockerfile is used to build a custom image that has modules such has Python 3, Docker and Open vSwitch pre-installed. The *dind* image is used as a baseline. The Dockerfile is a text document which contains commands that users can call to assemble an image. This image is used to create the mobile hosts using Docker-based containerization. A shell-based scripting language based on Linux libraries was used to implement the UI. Thus, the emulator can be generated on any Linux-based

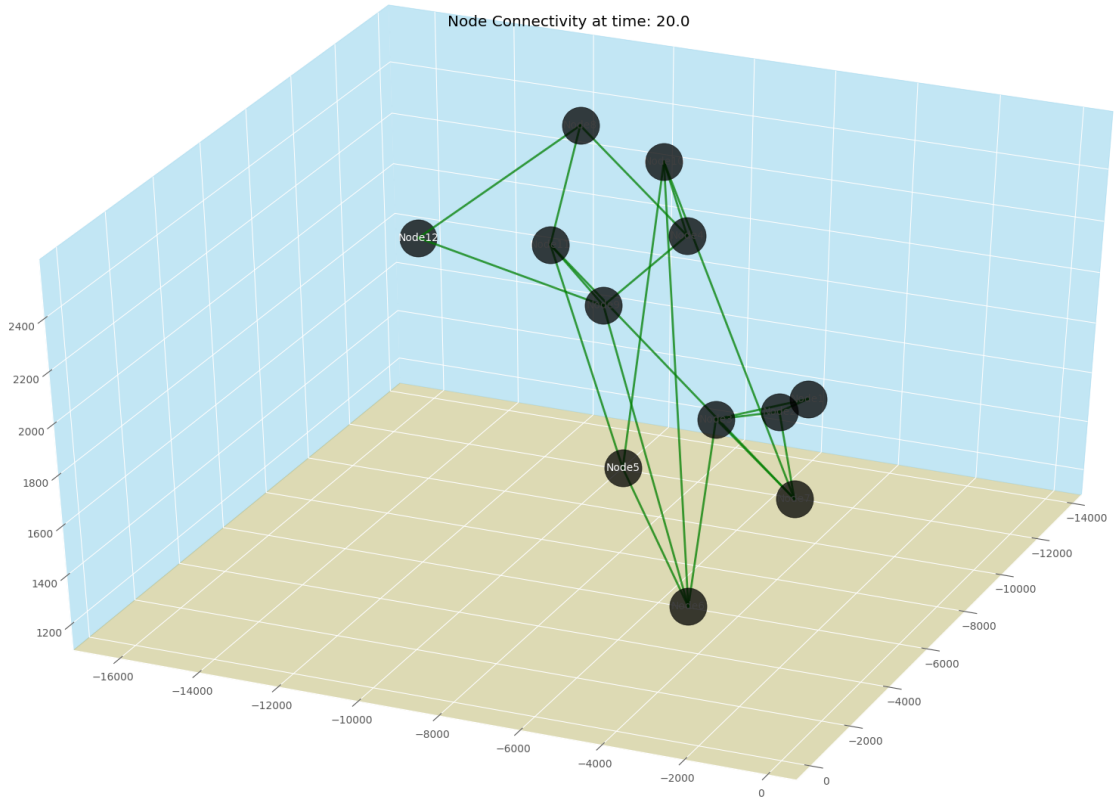

Figure 3.9. MANET node connectivity display for twelve nodes after 20 seconds.

OS. The CLI is shown in Figure 3.8. In the figure, the term *NestedNet* refers to the emulator. The CLI constructs the required number of nodes with GNABs and iPHYs, run simulations for a specified period, and then deletes the emulator.

3.3.6.1 Node-Connectivity Display

A Python 3 script using the *matplotlib* library was developed to visualize connections between nodes. The visualization is changed as visibility is updated. The script extracts current connectivity information from the CA and plots the graph for the user. An example of the display is shown in Figure 3.9. The image can be rotated for a three-dimensional view. The figure shows a network of twelve nodes connected after a 20 second emulation run.

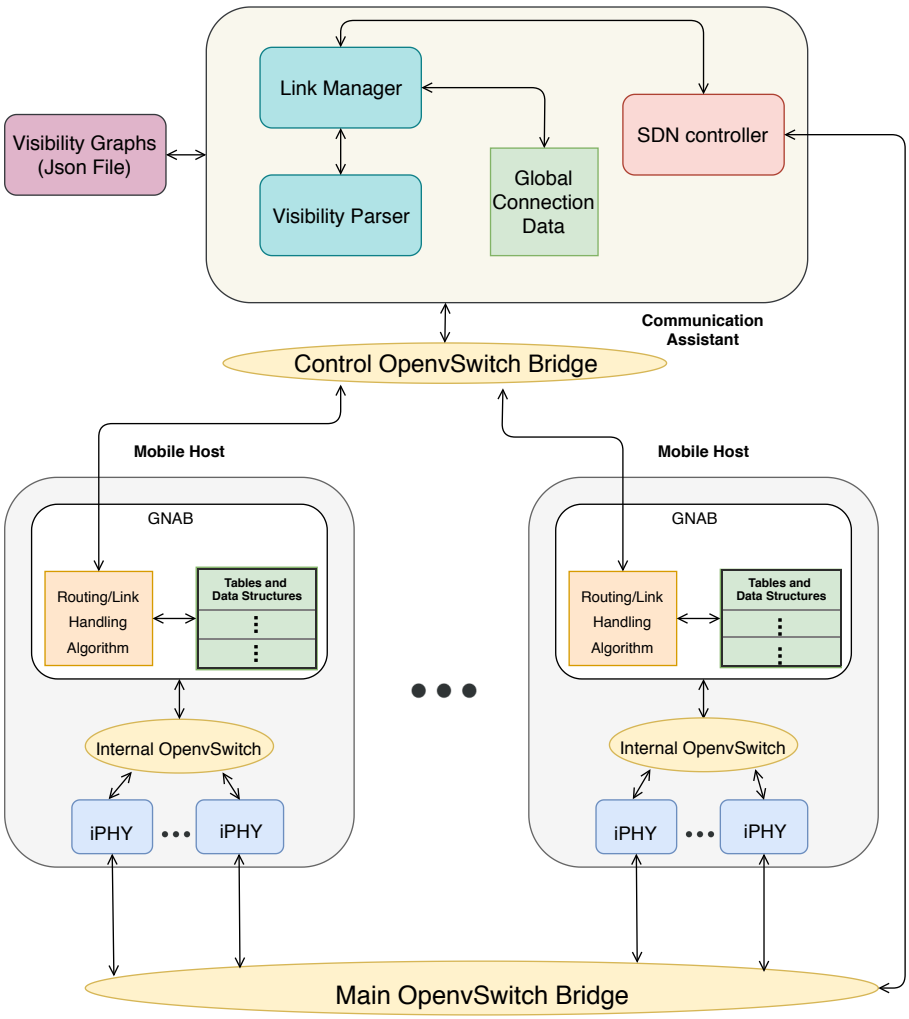

Figure 3.10. Implementation of mobile nodes, the communication assistant (CA) and their interaction. GNABs communicate to the CA via a control OVS. The CA parses the JSON file and maintains visibility and link information in tandem with the GNABs.

3.4 Emulator Operation

After a successful emulator setup, the testing of custom, user-defined routing protocols can begin. The CLI starts processes for all GNABs and iPHYs. During emulator generation, the source code and binaries for applications are copied into the individual namespace filesystems of the containerized units. Figure 3.10 shows a system level view of the emulator.

```
{
 "MobilecomScnDef" : {
  "simDuration": 10.0,
  "MobileNodeDef" : [
    {
      "nodeName":"BIIF-1", "nodeID":36585, "PosX":-15777.768500, "PosY":-4847
        .702200 ,"PosZ":-21031.090300, "Time":0.000000,
      "visibilityGraph":[
        {"nodeName":"BIIF-1","nodeID":36585,"LoS":true,"PosX":-15777.768500
          ,"PosY":-4847.702200,"PosZ":-21031.090300},
        {"nodeName":"BIIF-2","nodeID":37753,"LoS":true,"PosX":6908.784700
          ,"PosY":25485.215000,"PosZ":-1440.555000}
      ]
    },
    {
      "nodeName":"BIIF-2", "nodeID":37753, "PosX":6908.784700, "PosY":25485
        .215000 ,"PosZ":-1440.555000, "Time":0.000000,
      "visibilityGraph":[
        {"nodeName":"BIIF-1","nodeID":36585,"LoS":true,"PosX":-15777.768500
          ,"PosY":-4847.702200,"PosZ":-21031.090300},
        {"nodeName":"BIIF-2","nodeID":37753,"LoS":true,"PosX":6908.784700
          ,"PosY":25485.215000,"PosZ":-1440.555000}
      ]
    }
 }
}
```


Figure 3.11. An example JSON visibility graph file. Visibility to other nodes represented as Boolean values in the *LoS* field.

The first component to start is the CA, followed by the nodes. The JSON-based visibility information for each node is then parsed by the CA using the *visibility parser* function. The JSON file includes a timestamp and visibility information for nodes represented as Boolean values, as illustrated in Figure 3.11. In this example, nodes with names BIIF-1 and BIIF-2 exist. Each node has visibility to other nodes (*LOS* field) for a given timeframe indicated by the *Time* field. *PosX*, *PosY*, *PosZ* indicate the geographical position of the node at the given time.

A GNAB polls its iPHYs to receive information about the other components within its node. Each GNAB then sends a HELLO message to the CA via the control OpenvSwitch Bridge asking for acknowledgement to join the network. This bridge interfaces all the GNABs to the CA. The CA replies with a unique ID and IP address for the GNAB. Once all GNABs have received IPs, the CA sends node visibility information to the node's GNAB. The GNAB then receives the visibility information

Algorithm 1: Algorithm for handling a link request

```
Data: Link request with Src Node ID: s, Dst Node ID: d, Src iPHY: p
Result: Create virtual link to connect mobile nodes
if iPHY p is not occupied in Node s then
    if link between Node s and Node d exists then
        Reply Node s the LINK_FAIL
    else
        if at least one suitable iPHY m is free in Node d then
            Create virtual link between s and d
            Add link information to the link table
            Reply Node s the LINK_OK
            Inform Node d the LINK_OK and the iPHY m is used
        else
            Reply Node s the LINK_FAIL
        end
    end
else
    Disregard current link request
end
```

from the communication assistant and stores it in its local cache, shown in Figure 3.9 as *Tables and Data Structures*. Based on this visibility and iPHY resource information, the GNAB sends requests for links using a *Routing/Link Handling Algorithm*. The algorithm allows iPHYs to connect with its visible peers. After the peer-to-peer mesh network setup, the GNAB broadcasts its ID through the iPHYs and maintains routing information about the nodes to which it can communicate. The CA continuously monitors received link requests, established connections, and maintains global connection data.

While the high-level use of the CA to configure connections in the Main Open vSwitch Bridge was discussed in Section 3.3.4, more details are provided here. If a GNAB has a free iPHY, it will request a new node-to-node connection. Based on visibility information, the GNAB will send a link request to the CA to connect with a remote node. The link allocation algorithm used by the CA is shown in detail in Algorithm 1. The CA evaluates the request received from the GNAB and adds a flow to the main Open vSwitch bridge using a simple Linux SDN controller. The addition

of the flow signifies an established link between two mobile nodes via two selected iPHYs. A link is only created if the iPHYs in both nodes are available. The CA then sends a LINK_OK message to the requesting GNAB to confirm the link. The GNAB adds this link to its routing table and forwards it to its components. In some cases, the GNAB may request a link, but the CA is unable to install one. In the following cases it may not be possible to allocate a link:

- A link already exists between the requested mobile nodes via iPHYs.
- The destination node requested the same link and the request is received before the current request.
- Before the request could be processed, all the iPHYs were occupied in the destination node.

In such cases, the CA will send a LINK_FAIL message to the GNAB, which will induce the GNAB to try to create another node-to-node connection. This concludes the network generation phase of the emulator after which data flow may begin.

After a time interval, the CA processes new visibility information from the input JSON file. If some nodes are no longer visible to each other, the CA deletes their flows and informs the affected GNABs that they no longer exist. The GNABs may then request new links based on the updated visibility information.

The period between two consecutive visibility changes and the frequency of the changes may be controlled via the visibility JSON file. Emulation can be performed over multiple time periods of minutes or hours.

3.5 Initial Evaluation

In this section, we describe an initial evaluation of our emulation framework. The emulator is set up in a VirtualBox virtual machine (VM) installed on a 14-core Intel

Xeon workstation (2.6 GHz, 128 GB). The VM consists of eight processor cores and 32 GB of memory. The operating system (OS) is Ubuntu 16.04.6 LTS. Each mobile node is assumed to contain one GNAB, four iPHYs, and an internal Open vSwitch for internal interconnection in the mobile node. A series of tests were performed to evaluate the emulator in terms of memory usage, initialization time and performance.

The emulator was initially evaluated for memory requirements. Docker containers provide a tool *Docker stats* [38] to display a live stream of container(s) resource usage statistics. This tool and *top* [8] were used to diagnose the emulator memory requirements.

Table 3.1. System Memory Usage for NestedNet, the Mobile ad-hoc Emulator. Each node contains one GNAB and four iPHYs

Emulator Component	System Memory Steady State (Avg.)
Parent Container	568 KB
Child Container	404 KB
Docker daemon	38 MB
Internal Open vSwitch	76 MB
Main Open vSwitch Bridge	86 MB
Single Mobile Node	128 MB
Four Node Emulator	638 MB

Table 3.1 shows the system memory usage for a test run of the emulator. A four-node system was created for this evaluation. A Docker container uses about 400-500 KB and an Open vSwitch instance consumes just over 80MB. This is the minimal memory usage for Docker container execution. Each node-level Docker container is instantiated with its own Docker daemon which consumes about 38MB of memory. This result indicates the memory usage for emulating a single mobile node.

- One Docker daemon on the host to create parent (node) container ($\tilde{3}$8MB)
- One parent container (568KB)
- One internal Open vSwitch (76MB)

- One Docker daemon inside the parent container (38MB)
- Five child containers (four iPHYs and one GNAB) (404KB each).

Excluding the Docker daemon on the host, close to 128MB are needed to represent a single mobile node with containers and daemons. A total of 168MB are needed to represent a one-node emulator including the Docker software running on the host. These figures can be extended to estimate memory usage for a larger emulator. For instance, the memory for a four-node emulator may be calculated as follows:

- Four times the memory for a single node (128 MB)
- Memory for the Docker daemon on host (38 MB)
- Memory for the main Open vSwitch bridge (86MB)

The total indicates a memory requirement of 638MB for a four-node emulator.

The initialization of the MANET emulator begins when the communication assistant sends the visibility graph to the GNABs. It then configures the main Open vSwitch bridge to create direct connections between iPHYs according to the link requests received from the GNABs. When all iPHYs have been used or all visible mobile nodes are connected, the GNABs stops sending link requests. Thus, initialization time is defined as the time taken for the creation of all the containers and Open vSwitches followed by the initial link creation. The emulator initialization time relative to mobile node count was measured.

Figure 3.12 shows the time in seconds taken by the emulator to complete the network generation. The number of mobile nodes to be generated is specified via the Command Line UI. It is seen that the initialization time shows nearly linear growth with the number of mobile nodes. The primary reason for this trend is the link request processing time take by the CA. Hence, the time cost is proportional to the number

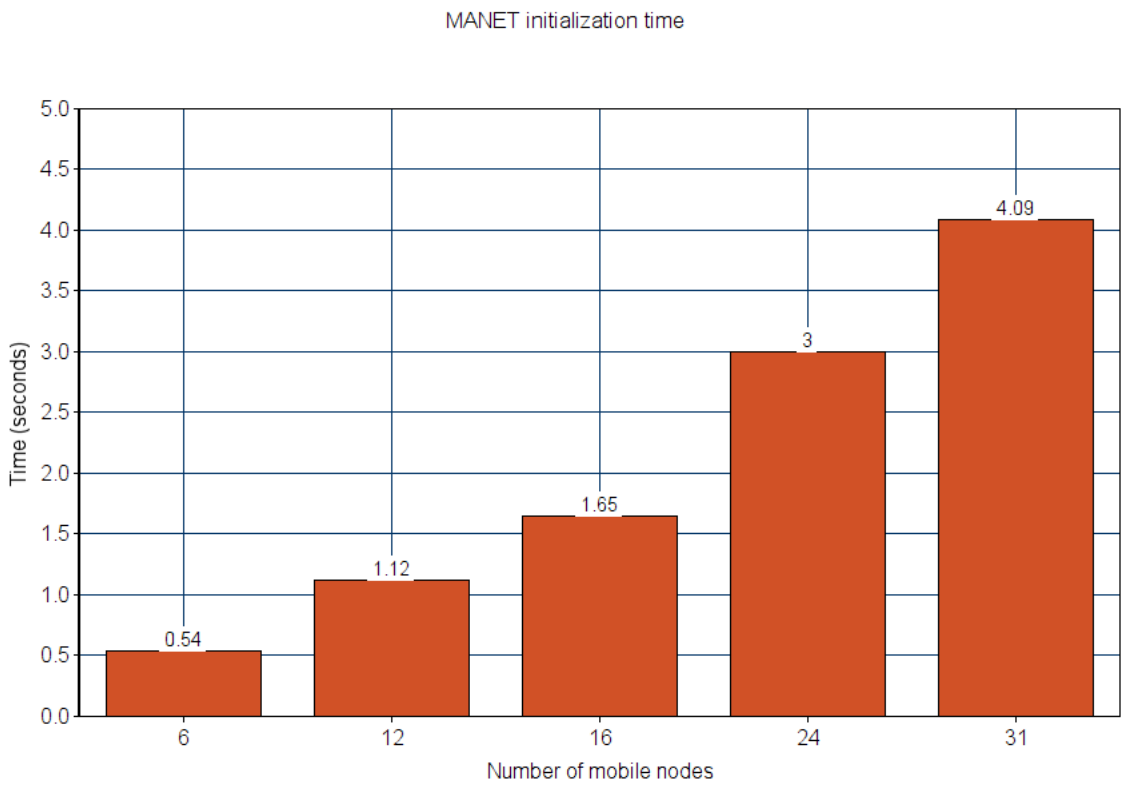

Figure 3.12. Time taken for MANET initialization. The emulator is initialized with a 6, 12, 16, 24 and 31 nodes separately and the time taken for each initialization is noted. Nodes contain a GNAB and four iPHYs.

of direct links that must be established. Duplicate link requests and the link requests that are unable to be established may increase time overhead.

The emulator was evaluated for intra-node and inter-node communication throughput and latency. Intra-node connections are GNAB-to-iPHY and inter-node connections are iPHY-to-iPHY across nodes. The tests were conducted using *qperf* [31] for a variety of mobile node counts.

Table 3.2 shows inter-node and intra-node throughput for an increasing number of emulator nodes. The test was conducted to determine the bandwidth between child containers (GNAB-iPHYs) in the same node and between child containers in different parent containers (iPHY-iPHY). A single connection per parent (node) was used. A

Table 3.2. Bandwidth for Mobile ad-hoc Emulator

No. Nodes	Inter-node Bandwidth (Gb/sec)	Intra-node Bandwidth (Gb/sec)
6	35.0	34.5
12	34.5	34.4
16	34.7	34.1
24	34.8	34.1
31	34.4	33.8

```
root@011a154ca492:/# docker exec phy1 /bin/sh -c "iperf -c 10.0.1.11 -i 1 -t 10"
------------------------------------------------------------
Client connecting to 10.0.1.11, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.13 port 50600 connected with 10.0.1.11 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  3.98 GBytes  34.2 Gbits/sec
[  3]  1.0- 2.0 sec  4.15 GBytes  35.7 Gbits/sec
[  3]  2.0- 3.0 sec  3.76 GBytes  32.3 Gbits/sec
[  3]  3.0- 4.0 sec  3.94 GBytes  33.8 Gbits/sec
[  3]  4.0- 5.0 sec  4.38 GBytes  37.7 Gbits/sec
[  3]  5.0- 6.0 sec  4.35 GBytes  37.3 Gbits/sec
[  3]  6.0- 7.0 sec  3.78 GBytes  32.5 Gbits/sec
[  3]  7.0- 8.0 sec  4.04 GBytes  34.7 Gbits/sec
[  3]  8.0- 9.0 sec  5.36 GBytes  46.0 Gbits/sec
[  3]  9.0-10.0 sec  4.97 GBytes  42.7 Gbits/sec
[  3]  0.0-10.0 sec  42.7 GBytes  36.7 Gbits/sec
```

Figure 3.13. Inter-node average bandwidth across nested containers in a single emulator node

bandwidth of about 34 Gbps was achieved for both inter-node and intra-node cases. Packets are sent within a virtual environment using virtual interface devices (Veth) between containers. Since both types of connections use Veth for communication via the same kernel space with just one Open vSwitch hop, the performance difference is negligible.

In Table 3.2, the inter- and intra-node bandwidth remains consistent regardless of node count. This result implies that increasing the number of containers, i.e. mobile node count in the emulator setup, does not effect individual link performance. Rows one and five show that a 6-node emulator and a 31-node emulator both display

comparable bandwidth performance of 34 Gbps for both inter and intra-node cases, indicating scalability.

Figure 3.13 shows the results of an intra-node bandwidth performance test between a GNAB container and an iPHY container in a single node. We used *iperf* [55] to measure the maximum network throughput between the two child containers by establishing a TCP connection. One container acts as a client while the second container runs the TCP server. The server uses its default interface to bind itself to a TCP socket. The client then uses this IP address to send data streams with a default TCP packet size of 1,500 bytes and a window size of 85KB. The time-stamped report of the amount of data transferred and the throughput measured between the two containers is shown in the figure. This test was run for ten seconds with bandwidth calculated at one second intervals.

Latency values, measured in μs and shown in Table 3.3, were generated with qperf. Latency was calculated by sending fixed size TCP packets (1,500 bytes) between a qperf client and server such that the packets bounce off the IP address. The client qperf daemon then determines the time elapsed for a packet to make the round trip.

Table 3.3. Latency for Mobile ad-hoc Emulator

No. Nodes	Inter-node latency (μs)	Intra-node latency (μs)
6	30.1	29.3
12	30.3	30.4
16	30.1	29.6
24	30.1	30.5
31	30.0	30.5

From the table, we observe that delay remains almost constant in the range of 30 μseconds for inter and intra-node transit. A typical network delay is measured in milliseconds. The decreased latency in emulation is a consequence of the software-only emulation environment in a single host. Since the packets do not traverse across physical network devices in a physical network, the delay measurement is reduced.

The only delay incurred for a packet in the virtual environment is routing through the kernel space with associated buffer delays and network stack limitations. Node count increase has a negligible effect on latency for both types of connections.

For deeper insight, a reference baseline is needed to highlight the performance benefits of NestedNet. In the following section, another Docker-based emulator, ContainerNet, is used for reference and quantitative comparison.

3.6 A Detailed Comparative Study with ContainerNet

In this section, we evaluate our emulator, NestedNet, with respect to ContainerNet. ContainerNet does not support nested/hierarchical Docker containers, reducing its ability to accurately implement node components. We illustrate the advantages and trade-offs of using nested containers in NestedNet with respect to single-layered ContainerNet.

3.6.1 Overview

ContainerNet is a fork of the popular SDN Mininet network emulator that uses Docker containers as hosts in emulated network topologies. ContainerNet has been designed for experimentation in cloud computing, fog computing, network function virtualization (NFV), and multi-access edge computing (MEC). However, due to its Mininet lineage, it can be used for generic network emulation using Docker containers with the API. ContainerNet includes virtualization technology that is similar to NestedNet The similarities and differences between ContainerNet and NestedNet are as follows:

3.6.2 Similarities

- Use of Docker containers for network nodes.

- Leverage Linux features such as network namespaces [20] and Cgroups [15] to provide isolation.
- Open vSwitch and OpenFlow compatible.
- Veth devices utilized for virtual link emulation and compatibility with Open vSwitch and controller technology.

3.6.3 Differences

- *ContainerNet:* Use of single layer Docker containers to emulate network nodes.

  NestedNet: Use of nested Docker containers to emulate nodes and node sub-components.

- *ContainerNet:* Processes run in each node within the container environment using container network interfaces.

  NestedNet: Processes run in nested child containers with a distinct environment.

- *ContainerNet:* Cannot directly run native binaries for multiple hardware platforms in the same container (For example, it is not possible to execute binaries that share the same IP address, MAC address, and TCP port in a node-level container. They must be in isolated containers).

  NestedNet: Each child container in a node has its own set of interfaces. Each container can run binaries directly and communicate with other processes. Each child isolates its processes from other containers to avoid memory overwrites and allow the use of the same resource (i.e TCP port 8080 for example) without conflicts.

3.6.4 Advantages of ContainerNet with respect to NestedNet

- Fast startup time due to a single layer of containers.

- Single Docker daemon and Docker image on the host for all container instances limits memory usage.

3.6.5 Drawbacks of ContainerNet with respect to NestedNet

- No API available for creating an intra-node environment; Need to use bash script or commands.

- Processes running in a container share an execution environment. The execution of multiple processes in the environment that require different versions of the same library in the same userspace and filesystem can cause conflicts. LDPATH, a Linux environment variable that points to directories where the dynamic loader should look for libraries for each process, must be modified for each process so the proper library can be located. Some libraries cannot be adapted in this way since the path is hard coded in the executable at compile time. At the network level, each process must be configured to use separate ports.

- Multiple processes may compete for the same container resources.

- In NestedNet, an equal share of CPU time can be allotted to each child container. This is managed via Cgroups allowing for the fair sharing of CPU and memory resources.

- In NestedNet, private inter-process communication (IPC) namespaces can be set up for child Docker containers. A POSIX/SysV IPC namespace provides for the separation of named shared memory segments, semaphores and message queues.

- In NestedNet, the PID namespace of each child container allows each subcomponent to have its own init-like process (PID 1), which controls all the processes within it. This supports container shutdown without affecting other

child container operations, similar to hardware implementation. In ContainerNet, a parent container has an init-process. Process shutdown requires knowledge of the process ID. The termination of PID 1 terminates all sub-component processes.

3.6.6 MANET topology framework in ContainerNet

In this section, the use of ContainerNet to implement an emulation environment that is similar to the environment shown for NestedNet in Figure 3.6 is detailed.

Figure 3.14 illustrates a MANET topology framework constructed using ContainerNet. The GNAB and iPHYs applications run as processes in a ContainerNet host. Each node is implemented as a Docker container with five processes communicating using Veth links through an internal Open vSwitch. Each Docker container has its own namespace. External inter-node (iPHY-to-iPHY) connections use the Main Open vSwitch Bridge with Veth links. Internal interfaces are created with bash scripts as the API does not directly support the creation of intra-node links and switches. Like NestedNet, iperf [55] is used to assess GNAB and iPHY emulator performance.

The CA operates in the same plane as the ContainerNet environment (on the host) and assists in the setup of the emulator. The Main Open vSwitch Bridge is controlled by the CA. The bridge is configured to add and delete flows between nodes. It adds rules to the Main Open vSwitch Bridge to attach two ports when two specific iPHYs in the network must be connected. If the CA needs to break a link as a result of a visibility change, it deletes the rule corresponding to the specific link. The CA reads visibility graph JSON files and perform topology updates based on changes in visibility.

3.6.7 Experimentation

Experiments were conducted to evaluate the performance of generic per-link bandwidth and latency and performance degradation caused due to multi-process interfer-

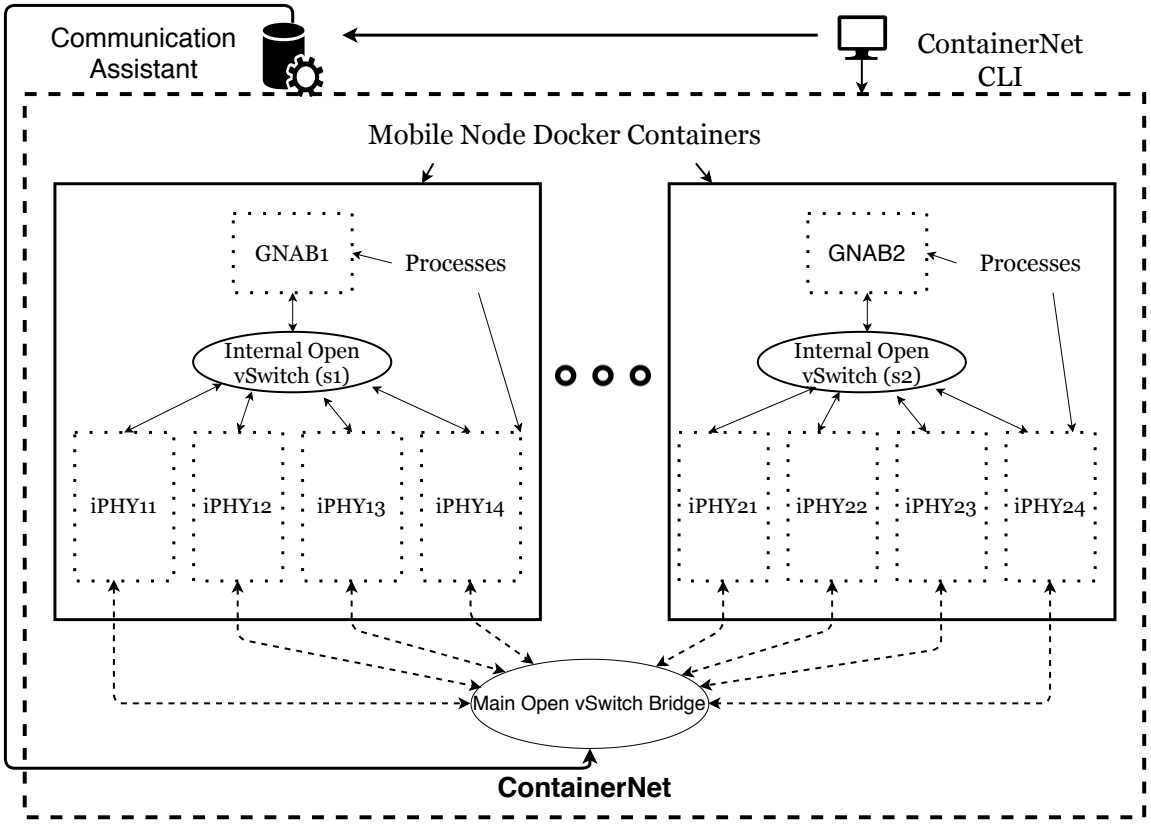

Figure 3.14. Framework of MANET topology implemented with ContainerNet. The GNAB and iPHYs run as processes inside a parent (node) container. The GNAB and iPHY processes share interface, memory and CPU resources with their host node Docker container.

ence. The *iperf* (TCP) traffic generator was used to calculate throughput and *ping* (ICMP) [52] was used to calculate round trip latency. The main observation points were intra-node links (GNAB-iPHYs) and inter-node links (iPHYs-iPHYs) between distinct nodes. Four sets of experiments examined 1) the throughput and latency of intra-node and inter-node communication, 2) the effect of a network intensive process on an iPHY to GNAB communication process in a node, 3) the worst case throughput and latency of intra-node and inter-node communication in a 12-node, 4 iPHY network and, 4) the effect of intra-node component scaling on intra-node link throughput.

3.6.7.1 Evaluation Infrastructure

- *System:* NestedNet is generated using the UI on a VM running Ubuntu 16.04 with 8GB memory and 4 processor cores. The VM runs on a 14-core Intel Xeon PowerEdge server (2.6 GHz, 128 GB). ContainerNet is also launched on a VM with the same CPU and memory resources. The parent containers and VM host are configured with Open vSwitch version 2.11.1. Docker daemon version 18.09.3 is used. All parent node containers are based on *dind*, while the child containers in NestedNet are based on a Ubuntu 16.04 Docker image.

- *Topology:* The topologies for NestedNet and ContainerNet were generated as follows:

  - *NestedNet:* A base MANET topology is created in NestedNet using nested Docker containers, such that each parent container contains 5 child containers, 1 GNAB and 4 iPHYs. $Iperf$ servers and clients are launched to evaluate link performance.

  - *ContainerNet:* For ContainerNet, all containers represent nodes and each container runs multiple processes ($iperf$ server/clients connected via Veth interfaces) representing GNAB and iPHYs.

  We considered assigning pre-defined equal CPU shares to each parent container to obtain a fair division of resources. However, this effort caused severe degradation of throughput and latency results for both emulators. All experiments described subsequently do not include pre-defined CPU shares for the Docker containers.

3.6.7.2 Experimental Setup

- *NestedNet:* Each container has a set of interfaces belonging to two local networks. One is the intra-node network for iPHY-GNAB communication (192.168.0.x),

the other is the network through which all iPHYs are connected via the Main Open vSwitch Bridge (192.168.2.x). For intra-node communication, an *iperf* server is executed on the GNAB container allowing all iPHYs within the node to connect as *iperf* clients. For the inter-node link test, one iPHY runs the *iperf* server and the other executes the *iperf* client. For latency evaluation, there is no need for a client-server relationship. Round-trip time is measured using an ICMP echo with *ping*.

- *ContainerNet:* The ContainerNet topology is created using an API [58]. Veth interfaces in the container are not assigned to a specific namespace. An IP address within a container cannot be repeated and thus each iPHY in the network is assigned different IP addresses to enable inter-node link evaluation. A bash script is used to create a local network that is distinct from the external network using pairs of Veth links. One end of a Veth link is added as a port on the Internal OVS while the other has an IP address of a local LAN and remains open for process binding. For experimentation, an *iperf* server or client can bind to an interface for intra-node and inter-node communication.

3.6.8 Results

3.6.8.1 Experiment 1: Intra-node and inter-node communication performance for a 12-node network

In series of tests, the throughput and latency of NestedNet and ContainerNet are compared. Intra-node and inter-node communication is evaluated by observing intra-node iPHY-GNAB connections and inter-node iPHY-iPHY connections. Performance results are obtained for one, two and four simultaneous network connections for both emulators.

- ***Setup:***

A 12-node network with 4 iPHYs each is used for this experiment. Each intra-node link is a iPHY-GNAB link which consists of a pair of Veth interfaces, interconnected via the internal Open vSwitch. One end of each interface acts as an OVS port, while the other is bound to a process. For inter-node links, iPHYs from two distinct parent containers (nodes) are connected via Veth interfaces, interconnected via the Main Open vSwitch Bridge. Only a single iPHY-to-iPHY connection exists between any two nodes.

- ***Experiment:***

  Bash scripts were used to run the tests in both emulators. A single test entails the transfer of a continuous data stream (TCP or ICMP) between the selected components for a given amount of time. A total of 100 tests of 60 seconds each were conducted to evaluate throughput and delay. For inter-node communication, one of the two iPHYs is randomly selected as an *iperf* server to establish connectivity. Each connection sends continuous TCP data stream of 1,500 bytes packet for 60 seconds at the maximum bandwidth available. Inter-iPHY data transfer is established when the *iperf* server and client are run. Data is sent unidirectionally from one iPHY to another in the experiments. Latency was calculated using *ping* which echoes ICMP packets towards a given IP address for 60 seconds. For intra-node tests, packets were sent from iPHY to GNAB, while for inter-node tests, one of the iPHYs sends data to the remote iPHY.

  - *Intra-node communication:* For an intra-node connection, the bash script selects a random node and an iPHY within it. The *iperf* server and client establish a link to the GNAB. For two intra-node connections, a different node than the one from the previous test is selected and two iPHYs in the node are selected. Two *iperf* clients are connected to the GNAB *iperf* server for simultaneously data transfer. For the four-connection tests, four

Table 3.4. Intra-node communication summary

Intra-Node Metrics (iPHY-GNAB connection)	ContainerNet	NestedNet
Avg. Throughput for 1 connection	30.36 Gbps	31.76 Gbps
Avg. Latency for 1 connection	0.056 ms	0.075 ms
Avg. Throughput for 2 connections	27.46 Gbps	26.29 Gbps
Avg. Latency for 2 connections	0.053 ms	0.072 ms
Avg. Throughput for 4 connections	17.57 Gbps	19.58 Gbps
Avg. Latency for 4 connections	0.052 ms	0.072 ms

Table 3.5. Inter-node communication summary

Intra-Node Metrics (iPHY-iPHY connection)	ContainerNet	NestedNet
Avg. Throughput for 1 connection	26.92 Gbps	32.05 Gbps
Avg. Latency for 1 connection	0.082 ms	0.080 ms
Avg. Throughput for 2 connections	24.11 Gbps	25.42 Gbps
Avg. Latency for 2 connections	0.075 ms	0.074 ms
Avg. Throughput for 4 connections	14.19 Gbps	15.74 Gbps
Avg. Latency for 4 connections	0.074 ms	0.073 ms

iPHYs are selected and links created in a similar fashion. Experimental results over 100 tests are shown in Table 3.4.

– *Inter-node communication:* For a single link inter-node communication, the bash script selects one random iPHY in each of two random nodes and runs the *iperf* test for 60 seconds. For the two node connection test, one iPHY is selected in each of four random nodes. Two iPHYs in separate nodes run the *iperf* server while the other two run the *iperf* client, thus creating two distinct inter-node connections. To create four inter-node connections, eight nodes are randomly chosen. Four nodes have an *iperf* server and four have *iperf* clients. Not more than one connection exists between two nodes. The results of the inter-node experiments are shown in Table 3.5. Every test selects a new random set of nodes and iPHYs to allow for a distribution of samples.

- ***Discussion:***

– *Intra-node communication:* In Table 3.4, it is observed that the intra-node bandwidth of NestedNet is comparable to that of ContainerNet for all three sets of experiments. The latency is about 20μs higher in NestedNet, however this is due to the multiple layers of containers that requires packet processing in the child container, the parent container stack and the destination child container stacks. Increasing the number of network connections communicating simultaneously increases the network processing time in the container network stacks and the CPU and memory usage of the underlying virtualized OS kernel. This effect causes the per-link bandwidth to drop with an increase in the number of intra-node connections.

– *Inter-node communication:* In Table 3.5, the inter-node throughput of NestedNet is slightly better than that of ContainerNet for all three experiments. NestedNet has a higher throughput for a single link, as each process is encapsulated by a container and thus can share the parent network stack evenly, as compared to ContainerNet. This effect can be explained as follows: The VMs of both emulators are allotted four CPU cores, where each core represents 100% CPU. The *Docker stats* command indicates the usage of each container with respect to total available CPU i.e. 400%. It was observed that while running the *iperf* server and client in ContainerNet, the node with the client used 120% of 400% CPU, while the server used only 80%. Meanwhile, in NestedNet, the client used 118%, while the server used 104%. This unevenness in ContainerNet caused the server to react more slowly causing TCP re-transmission and delayed acknowledgement packets. NestedNet provides better resource sharing. Thus, both client and server can match each other to provide lower packet loss, re-transmissions and packet duplication. To validate our findings, CPU restrictions were added to all parent containers (33% usage per parent container for 12 nodes

out of 400% of available CPU). It was observed that both NestedNet and ContainerNet supported similar bandwidth (1 Gbps).

The latency for NestedNet is comparable for all sets of experiments. For inter-node communication, the ICMP packets in ContainerNet must cross different container network stacks via the Main Open vSwitch and thus similar latency is seen. Similar to intra-node connections, increasing the number of connections simultaneously in the network worsens the network processing and CPU and memory usage leading to packets getting delayed and backlogged. Per-link throughput drops to 15Gbps in both the emulators as we increase the number of connections in the network.

3.6.8.2 Experiment 2: Background stress test on intra-node processes

In this experiment, the benefits of isolating node components in containers is explored via an intensive network test. A network intensive background process (misbehaving process) is executed in a node to consume resources and affect other iPHY and GNAB processes. The misbehaving process bombards the network stack and increases system load. It occupies the network bandwidth/network stack and increases the CPU/memory usage of the container, thus degrading the performance of other executing processes. An *iperf* client that generates concurrent 120 TCP threads is used to represent the misbehaving process. The *loopback* interface, a dummy Linux interface that bounces packets off to imitate sending/receiving packets by a intra-node component is used. This interface exists in the same container as the misbehaving process. For testing, a standard intra-node link is established as a baseline and the effects of the misbehaving process are observed. An *iperf* client sends 1,500 bytes of TCP data to an *iperf* server using a single thread to form the "well-behaved" process. Linux processes are scheduled to run using prioritized round robin scheduling. In the case of TCP-based *iperf* processes, slight delays at the data receive/send queues

cause the scheduler to temporarily preempt a process. For instance, if the time slice for a process receiving data ends before the receiving buffer's lock is released, the lock will remain until the next time slice is allocated to the process. The time when the process resumes execution is directly dependent on system load. The misbehaved process increases the system load by introducing multiple parallel TCP streams. The introduction of such a process should not have an effect on the behavior of isolated containers. If the process is enclosed in a child container it does not share locked resources.

We define the performance degradation of the well-behaved process in regards to the misbehaving process using the following metric. Degradation D is defined as D = (T1-T2/T1), where T1 is the throughput of the well-behaved process before introducing the misbehaving process and T2 is the throughput when the misbehaving process is running.

- ***Setup:***
  - *ContainerNet:* The misbehaving process executes in the same container as the well-behaved process.
  - *NestedNet:* The misbehaving process is in one child container while the well-behaved process executes in a different child container.
- ***Experiment:***

  Experiments for ContainerNet and NestedNet were conducted for a period of 40 seconds such that the misbehaving process starts at the 10 second point and ends at the 20 second point, as shown in Figure 3.15. The effect on the throughput of the well-behaved process is observed.
- ***Discussion:***

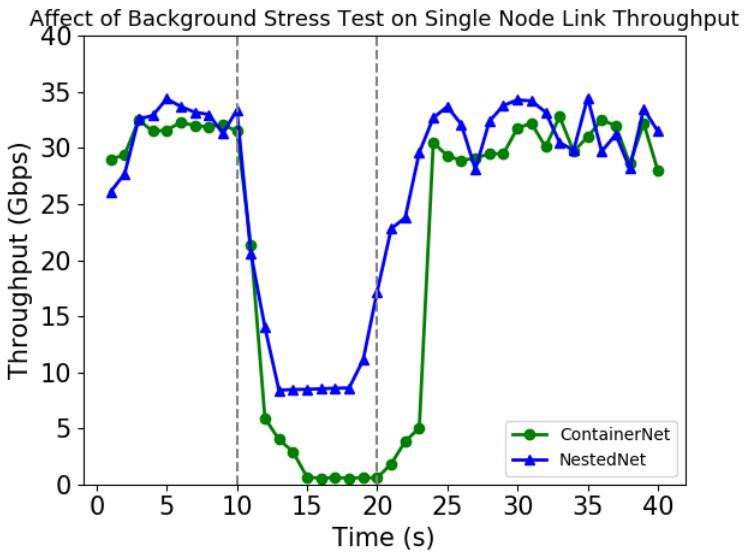

Figure 3.15. Effects of stress test on the well-behaved process throughput. The misbehaving process executes from 10s to 20s. A large throughput drop is observed for the well-behaved process in ContainerNet (98%) versus NestedNet (72%).

– *ContainerNet:* In Figure 3.15, a significant throughput drop of 30Gbps to 500Mbps is observed in ContainerNet, i.e. D = 98%. The well-behaved process executes in the same environment (parent container) as the misbehaving process. They share the same IPC namespace, interfaces and network stack. Delays in process scheduling due to irregular CPU sharing between processes can occur, as observed in Experiment 1. This effect may cause packet processing delays due to higher system load exerted by the misbehaving process. Concurrent bombardment of packets by the multi-threaded process further complicates timely packet processing of the well-behaved process by the network stack of the parent container.

– *NestedNet:* In Figure 3.15, a throughput drop from 30Gbps to 8Gbps is observed in NestedNet, i.e. D = 72%. The processes execute in distinct child containers with dedicated network stacks, interfaces and namespaces. The containers enable fairer CPU and memory sharing amongst the pro-

cesses such that the well-behaved process can send and receive packets in a timely manner. The 72% drop can be attributed to the load on the underlying virtualized kernel OS of the parent container and the host.

To summarize, degradation for the emulators are: ContainerNet: D = 98% ; NestedNet: D = 72%. This amounts to 26.5% lower degradation in NestedNet.

3.6.8.3 Experiment 3: Worst-case evaluation for a 12-node network

In this experiment, a 12-node MANET environment is evaluated under the condition that all inter-node and intra-node links are occupied simultaneously. This provides a worst-case estimation for the link bandwidth and latency of the emulator. Unlike Experiment 1, multiple inter-node connections are created between the same nodes and all inter- and intra-node communication occurs concurrently.

- ***Setup:***

  A 12-node network with four iPHYs and a GNAB in each node is used. The intra-node and inter-node connections are established as described in Experiment 1. Intra-node and inter-node communications are evaluated by observing intra-node iPHY-GNAB connections and inter-node iPHY-iPHY connections, respectively. For inter-node connections, the Main Open vSwitch Bridge is configured to connect all iPHYs from a node to the iPHYs of another node.

- ***Experiment:***

  An experiment consisted of two tests, one for throughput evaluation using an *iperf* server and clients and one for latency using *ping*. A test entails continuous data transfer (TCP or ICMP) between all intra-node and inter-node components for a period of 90 seconds. For inter-node connections, the iPHYs in even-numbered nodes were chosen to run the *iperf* server and the ones in

Table 3.6. Throughput comparison summary

Metric (Mbps)	Containernet	NestedNet
Avg. intra-node throughput	1383.99	861.01
Stdev. of intra-node throughput	87.75	164.99
Avg. inter-node throughput	1827.92	1340.24
Stdev. of inter-node throughput	250.97	324.68

Table 3.7. Latency comparison summary

Metric (ms)	Containernet	NestedNet
Avg. intra-node latency	0.042	0.091
Stdev. of intra-node latency	0.007	0.006
Avg. inter-node latency	0.081	0.090
Stdev. of inter-node latency	0.020	0.007

odd-numbered nodes execute the client. The data flow occurs as described in Experiment 1 for inter-node and intra-node components. A total of 100 tests were conducted to create a performance distribution. The average and standard deviation of both metrics were computed. The containers were set up without any specified CPU and memory sharing as that approach would significantly degrade the available throughput. Docker can effectively isolate and share resources among the containers. ContainerNet is at a disadvantage due to the absence of nested child containers.

- ***Discussion:***

  Throughput and latency results of the experiment are charted in Table 3.6 and Table 3.7, respectively.

  - *Throughput:* In Table 3.6, it is observed that average throughput per intra- and inter-node link is better in ContainerNet than NestedNet. NestedNet's extra container layers exacerbates packet processing time. Moreover, running 72 *iperf* connections simultaneously (48 internal links and 24 external links) significantly increases CPU and memory load on the parent containers. The CPU/memory scarcity can cause network performance

degradation resulting in lower throughput and higher standard deviation due to the more difficult allocation of resources.

– *Latency:* The worst case latency per link is comparable in both emulators (Table 3.7). NestedNet's multiple layers can cause a delay of about 50μs more than ContainerNet. A *ping* application occupies much less of the CPU (0.06%) than an *iperf* (7%) application. Thus, the system load from executing 72 connections is much lower. The inter-node links show a slightly higher deviation than NestedNet.

  In ContainerNet, a parent container executes four intra-node and four inter-node *ping* processes at the same time, sharing the same network stack. Thus, packets may wait until the memory resource is allocated to the process to send a ICMP packet or a reply. A busy CPU attending to other packets can cause this reply to be delayed, or even lost. Isolating the processes in child containers mitigates this issue. An iPHY running in a child container can better manage two processes (one intra-node and one inter-node) to receive and send packets as it always has its share of CPU and memory.

3.6.8.4 Experiment 4: Performance scalability evaluation for intra-node components

In this experiment, internal processes in a two-node environment are scaled to observe the effect on the throughput of intra-node links. The use of more than five nested containers is considered. This experiment examines framework limits and quantifies the performance of nodes with a large number of internal components.

- ***Setup:***

The MANET topology is regenerated for each experiment with an increasing number of iPHYs per node, ranging from two to twenty-five. The test and observation points are iPHY to GNAB links.

- ***Experiment:***

  For a given number of iPHYs per node, a total of 100 tests with TCP streams running for 90 seconds were conducted. Each test entails *iperf* clients (iPHYs) sending constant TCP data of 1,500 bytes to the GNAB *iperf* server. The throughput was averaged over all the iPHYs per node. The standard deviation obtained signifies the variation in throughput amongst the intra-node links. For each subsequent experiment, the emulator was deleted and reconstructed with an increment of 5 iPHYs per node.

- ***Results:***

  The effect of scaling on the GNAB-iPHY link throughput is shown in Figure 3.16. The green and blue graphs indicate the average link throughput per node for ContainerNet and NestedNet for a intra-node. The red and black graphs show the increase in standard deviation of throughput per node as a percentage of the average. Each point is averaged over 100 trials.

- ***Discussion:***

  - *ContainerNet:* In Figure 3.16, it is observed that increasing the number of iPHYs significantly drops the throughput per intra-node link. The iPHYs and GNAB processes execute in the parent container resulting in network processing delays and affecting the throughput. The deviation of throughput increases by 5% to 13%, i.e. each link may fluctuate 5-13% from the average throughput. This can be attributed to the processes competing for the CPU and network resources to cause difficult allocation. As seen in

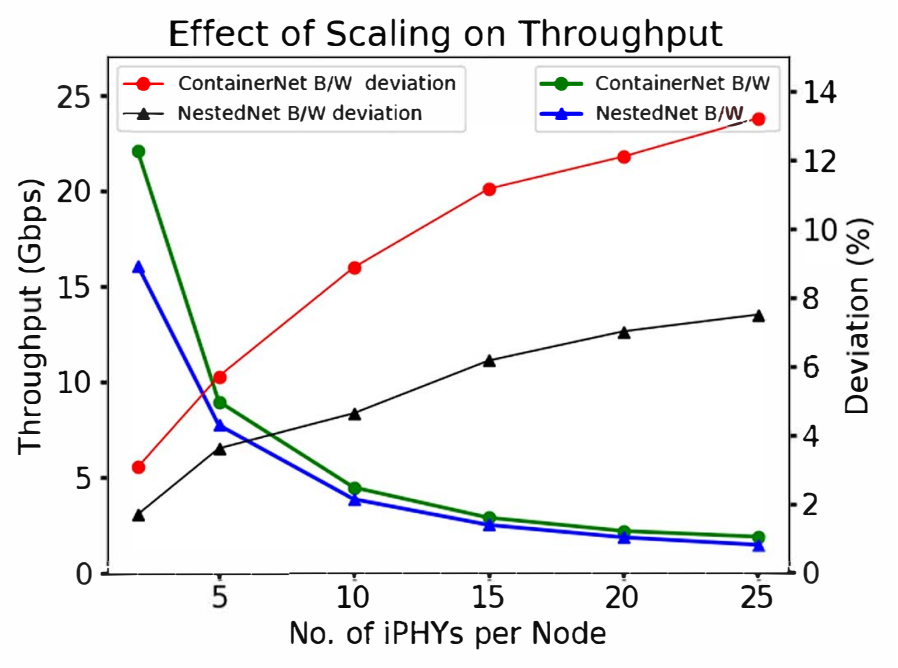

Figure 3.16. Effect of scaling intra-node components in a two-node network. The average GNAB-iPHY throughput per node and corresponding percentage deviation from average with increasing number of iPHYs per node.

Experiment 1, each client (process) may receive intermittent access to CPU resources, resulting in a delay in transmission/re-transmission. Moreover, the GNAB server, also present in the same environment, may encounter delays in sending acknowledgment due to the lack of CPU time consumed by the clients. This may lead to TCP re-transmission, duplicate packets and packet loss.

– *NestedNet:* NestedNet shows a similar drop in intra-node throughput. Increasing TCP application usage increases overall system load, stressing the underlying kernel of the parent and the host. However, this issue can be alleviated with more hardware resources such as CPU/memory and is not limited by the emulator function. The deviation of link throughput is less in NestedNet, varying from 3% for 1 iPHY per node to 8% for 25 iPHYs per node. Thus, each intra-node connection can maintain a more

stable throughput. Fair division of CPU shares amongst iPHY containers during execution mitigates the problem of a haphazard allocation of resources between processes. Hence, each iPHY *iperf* process is more likely to transmit and receive packets in a timely manner. Similarly, the GNAB *iperf* server can send acknowledgements to all connected clients.

3.6.9 Summary

- *The advantages of NestedNet can be summarized as follows:*

  The isolation provided by NestedNet can provide more stability for intra-node processing. A misbehaving process can cause performance degradation of 98% in a well-behaved link of ContainerNet versus 72% in NestedNet. Scaling the number of internal process is less disruptive for intra-node throughput in NestedNet, with a deviation of up to 8% as opposed to 13% in ContainerNet. NestedNet represents better management of intra-node process due to the modular, hierarchical design such that network resources can be reused in same network node by different processes.

Table 3.8. Container Startup Time (seconds)

Nodes	iPHYs	Containernet	NestedNet
12	4	28s	98s
2	25	9s	93s

- *The drawbacks of NestedNet against ContainerNet can be summarized as follows:*

  The start-up time for NestedNet is significantly higher (Table 3.8) than ContainerNet. A 12-node environment with five sub-components (Four iPHYs and one GNAB) needs over a minute and a half to build. The primary overhead is related to the child container image loading and creation. Increasing the sub-components does not add significant overhead.

CPU usage and memory usage of bare nested containers in NestedNet is more than in ContainerNet. An idle NestedNet node needs 0.8% CPU and 128MiB memory, while a ContainerNet node uses 0.15% CPU and 48MiB memory. This is due to multiple nested daemons (Docker and Open vSwitch) that are required to instantiate child containers. As a single process, ContainerNet requires only one Docker daemon per host and an Open vSwitch inside each container. Thus, there is no overhead of creating an additional Docker daemon for child containers.

CHAPTER 4

NETWORK LINK DYNAMIC EMULATION TESTBED

4.1 Introduction

Emulation methodologies have evolved to an extent where platform-building and application execution are not enough. Real-time distributed systems place significant demands on a network [13]. Moreover, the introduction of new algorithms and protocols exacerbate the need to provide real-time quality of service (QoS) configurations [80].

Network emulation should provide a means to evaluate non-functional properties of implemented protocols. However, it is challenging to emulate networked systems accurately, especially wireless networks subject to many parameters that affect the behaviour of a channel/link [9]. This behaviour necessitates that the emulation tool provide a way to introduce network impairments such as bandwidth limitation, delay and packet loss according to a user-defined model to test the protocols and applications properly.

This chapter describes the design and evaluation of a network emulator testbed, using technologies [2][81] that can provide an interface to alter link properties. Linux based algorithms [21][41][60] are used to change link properties using user-defined input. As shown in Figure 4.1, the emulator resides between two routers or hosts of a network to provide parameter throttling for several links at the same time. A hardware and software testbed is presented that is used to validate the performance of this emulator.

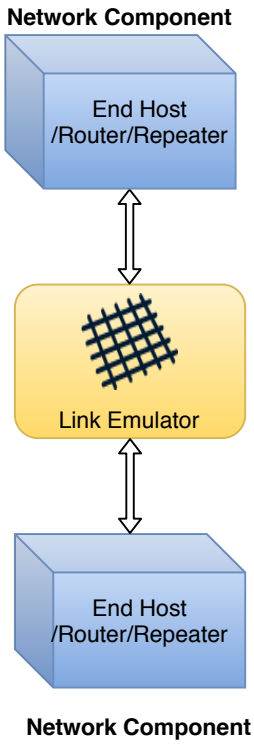

Figure 4.1. A link emulator prototype. An emulator may function as an intermediary node between two network components such as a repeater, router or end host.

4.1.1 Bandwidth Limitation

An important property of a network link or a connection is the maximum bandwidth it can support. A point-to-point connection between two hosts may have several segments of links stacked end-to-end. A bandwidth constraint at one segment can thus inhibit the entire link [32].

A prominent approach uses a leaky bucket algorithm [37] to limit bandwidth. A leaky bucket controls outgoing traffic to the specified bandwidth by storing the surplus traffic in a FIFO queue. When the queue is full, incoming packets are dropped. Linux maintains traffic shapers that use packet dropping algorithms. A well-known shaper is the Token Bucket Filter (TBF) [12]. TBF slows down packets to a specific rate and accepts a *limit* option indicating the maximum number of packets to queue. However, such leaky bucket options are not an ideal active queue management option even though they are fast. The biggest disadvantage is that being a "classless" shaper, it cannot prioritize one TCP stream over another.

Hierarchical Token Bucket (HTB) [45] improves on this approach by allowing the filtering of specific traffic to prioritized queues. However, unlike TBF, HTB doesn't allow queue length specification. This issue causes HTB to slow packets rather than to drop them which can have implications on the delay.

Another Linux algorithm which is prominently used to limit bandwidth is HFSC (Hierarchical Fair Service Curve [72]). HFSC allows classification of traffic like HTB without the disadvantage of dropping packets. HFSC uses filters provided by the Linux traffic control API (tc) to decide packet class.

Two-way link bandwidth can be regulated on a per-direction basis or using common bandwidth limitations. Bandwidth might also be shared by all participants of a multi-point connection. Bandwidth limitations are primarily set to bits/second or bytes/second.

4.1.2 Delay

In a network, packet delay between two connected hosts is comprised of diverse components such as: propagation delay, medium access delay, and queuing delay [82]. The propagation delay is characterized by the type and length of the propagation media. It may also include delays introduced by network components that are on

link segments such as repeaters. A static system will have a constant propagation delay. Moreover, for a wireless network, the delay may vary with positional changes. The emulation tool must support these changes.

Our emulation approach involves setting link delay properties that are an aggregation of all delays over the link. This action can be performed using point configuration in which a single point/interface to be regulated is chosen for the complete link. Ideally, a point close to the destination is selected. Point selection can be determined by the user. Multiple points can also be considered. However, this choice may introduce the overhead of maintaining consistent parameter values on the link interfaces.

Linux provides an API in the form of a scheduler to configure interface latency. NetEm is a network emulator included in the Linux kernel [36]. It provides emulation functionality to test protocols and applications by presenting an API to emulate network properties such as packet loss, duplication and packet corruption. NetEm consists of a queuing discipline known as "qdisc" [27]. It has been integrated as a part of the Linux kernel since version 2.6.8. The delay parameters for a link can be described using an average value (μ), a standard deviation (σ), and correlation (ρ). NetEm allows for the specification of a given average time for packet delay. It also allows for a random variation in average time delay with a correlation in %. An example of such a configuration is:

```
tc qdisc add dev eth0 root netem delay 10ms 5ms 10%
```

where the constant delay is 10ms, 5ms is variation σ and 10% is the correlation ρ. NetEm uses a uniform distribution (μ +/− σ) by default. A qdisc may be classless or classful. Classful qdiscs contain classes and provide a handle to identify a class. Handles can then be used to attach filters for packet-based QoS configuration. In our approach, classful qdiscs are used since they allow for addition of bandwidth, and delay and packet loss configuration simultaneously. Delays are described in milliseconds (ms) in the network domain.

4.1.3 Packet Loss

A tunable packet loss parameter is required for specific applications. There are two main reasons for packet loss: packets are dropped as a consequence of link congestion or impaired due to transmission errors [44]. The bandwidth limitation approach mentioned in the previous subsection drops surplus packets in the case of congestion in the emulated link. Thus packet loss in our approach primarily signifies transmission errors. Packet loss is described in percentage (%) of packets received. Such errors are introduced in wired and wireless links. Wireless links typically have greater packet loss that must be modeled [67]. Packet loss is defined in terms of probability, usually as a single loss probability value in a model [66]. In this work, a simplistic emulation model is assumed that can be characterized by a single value of packet loss. NetEm emulates packet loss by randomly dropping the specified percentage of packets before they are queued [36]. The model manipulates loss probability over different link interfaces along with (if the scenario requires it) a loss correlation. However, a realistic emulation model should provide a more flexible approach.

4.1.4 Dynamic Parameters

Network properties, including those mentioned above, propagation delay and round trip time (RTT), must be accurately modeled in an emulation platform. RTT delay can determine system performance for networked systems [77]. Bandwidth limitations may be a consequence of protocols and applications. Bandwidth allocation in server systems [51], link aggregations [34] and adaptive bandwidth in 802.11 protocols [79] are some of examples where bandwidth throttling may be necessary. Packet loss determination inherently requires a dynamic model that can be parameterised with a mean probability and correlation. The model itself may change over time depending on physical conditions, especially for wireless networks. Therefore, a real-

istic emulation model must support both fixed values and dynamic models of these parameters.

4.2 Mininet-based emulator

Mininet [10] is a popular simulation software that brings together several network tools such as namespacing [20], Veth [18] and Open Flow [49] to build custom network models. It allows parameter tuning on physical interfaces. However, the use of virtual interfaces is a much cleaner approach. Mininet is used to generate our custom software environment, with support from Python [29] and Open vSwitch (OVS) to allow smooth migration and deployment.

Mininet presents different API levels to create a custom network with abstract processes as hosts and Veth links as cables. It supports virtual switches such as Linux Switch [76] and Open vSwitch. In this work, the lower level API is used to generate an environment with an Open vSwitch bridge which provides a platform to configure link parameters. Mininet and Open vSwitch both support traffic control APIs for qdiscs and NetEm. However, performing dynamic link reconfiguration using the Mininet API is not straightforward and requires modification of the virtual platform.

The Mininet-based emulator uses the "Mininet-VM" virtual machine (VM) provided by its developers. The VM can be run on Oracle VirtualBox to construct a custom environment on any host. For a simple link emulator integrated with a hardware testbed, a single Open vSwitch that connects the two network ends is sufficient. For testing purposes, the emulator is used to test a simple network, i.e. a client-server topology. The topology consists of several clients connected to a server via multiple links, each of which has unique characteristics. The link emulator sits between the router connecting the clients and the server, establishes the links and throttles link parameters of each link separately based on source and destination Layer-3 addresses.

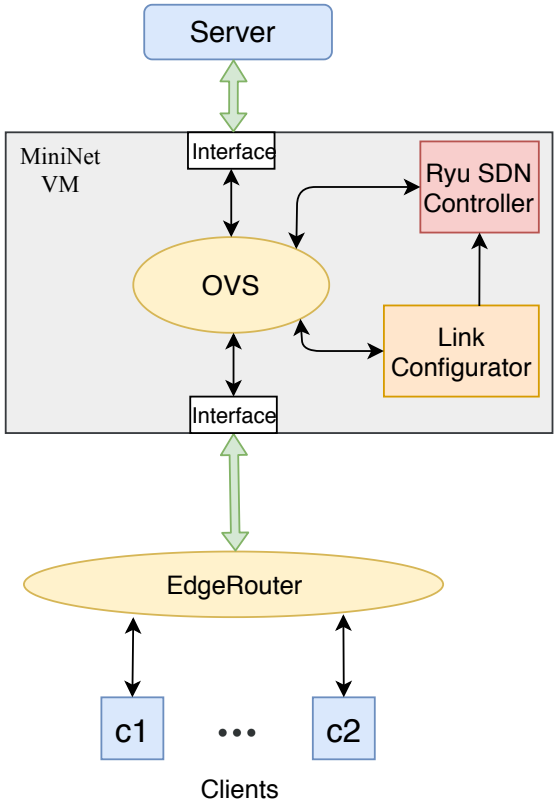

Figure 4.2. Link Emulator Design with Mininet VM, OVS and Ryu-Router. The Mininet VM acts as an intermediary between the physical router (EdgeRouter) and the server workstation.

In our experimentation, the VM is deployed on a Dell Optiplex 7010 machine with two physical interfaces (1G network interface cards). One physical interface of the workstation is connected to one end of the network (client side) and the other physical interface is connected to the server side. The design uses an Open VSwitch as an intermediary black-box that can steer/modify/parameterize traffic that passes through it. The goal is to perform QoS throttling in the black-box.

Figure 4.2 illustrates an abstract view of the emulator and its position as a black-box between the EdgeRouter-X [54] and the server. The EdgeRouter-X is a general purpose physical router that can forward packets from the clients to the workstation running the VM. The addition of the emulator between the two network points is performed by configuring the OVS such that the two interfaces on the host machine

are used as bridged adapters to the Mininet-VM. A bridged adapter is a part of VirtualBox "Bridged Networking" mode wherein it exposes the guest machine to the local network. This is achieved by creating two virtual interfaces that reside in the same sub-network as the physical network. Oracle VM VirtualBox can thus connect to one of the installed NICs and exchanges network packets directly, circumventing the host operating system's network stack.

The virtual interfaces in the VM are configured as ports on the Open vSwitch, allowing the OVS to be used as a *nexthop* point for traffic that traverses paths between the client and server. The presence of the Open vSwitch is not enough to route traffic via the emulator VM. The client side interface is typically connected to a physical router (EdgeRouter) which has multiple clients on other interfaces. The clients may belong to different sub-networks raising the need for routing software to modulate the OVS.

The Open vSwitch resides in the Mininet-VM and can act like a router to connect two ends of a network while simultaneously performing link property tuning. The Open vSwitch is inherently an L2 switch. To enable the SDN capabilities of the OVS, an SDN controller is necessary to process the packets. Ryu [65] is a remote SDN controller developed in Python that is used in conjunction with Open vSwitch to enable packet level filtering.

Ryu presents a general purpose API for custom application design as per user requirements. However, for a client-server scenario, a basic router is sufficient to effectively route packets from the various sub-nets of clients to the server. Ryu-Rest-Router [16] is a pre-designed application that presents a REST-API based interface with a fully functional router. It converts the Open vSwitch into a router that can effectively route packets arriving from the clients from different LANs to the server. It enables rules addition for routing and default gateway addresses via a REST API. The Open vSwitch and Ryu-Router complete the configuration. The OVS acts as a

gateway for the traffic to and from the server. A link reconfigurator function in the VM performs the task of throttling QoS measures.

4.3 Open vSwitch and Ryu-router

Using the Ryu-Router allows support for realistic traffic scenarios. For instance a network with multiple LAN's could be used with the Ryu_Router such that we can emulate links across different local area networks. Each sub-network often has different bandwidth or delay limitations. Visibility, distance and interference influence different wireless system link metrics.

A topology is created in Mininet that includes an Open vSwitch bridge which supports OpenFlow 1.3. A single virtual Mininet host is created that can be used for debugging purposes. A Ryu controller installation is accompanied with a *ryu-manager* that is used to run Ryu specific applications. Once the VM setup concludes, routing rules and gateways are added and the link configurator is initialized.

4.4 Linux *traffic control (tc)* for point link configuration

As described in Section 4.1.1, HFSC is a suitable design choice for bandwidth configurations. Moreover, classful qdiscs allow delay and packet loss parameters for the same interface. Each interface in Linux inherently contains a ingress and egress qdisc. The egress or root qdiscs are commonly used. These features simplify queuing disciplines by using classes and class structures. For the emulator, the two ends of a network connection are formed via two physical interfaces. Hence, two points are available along the link for network configuration. The absence of multiple links between clients is not limited by the design. It is assumed that the emulator is connected to a single physical router. Link configuration is performed at the server side interface for 'upwards' traffic, i.e. client to server, and at the client side interface

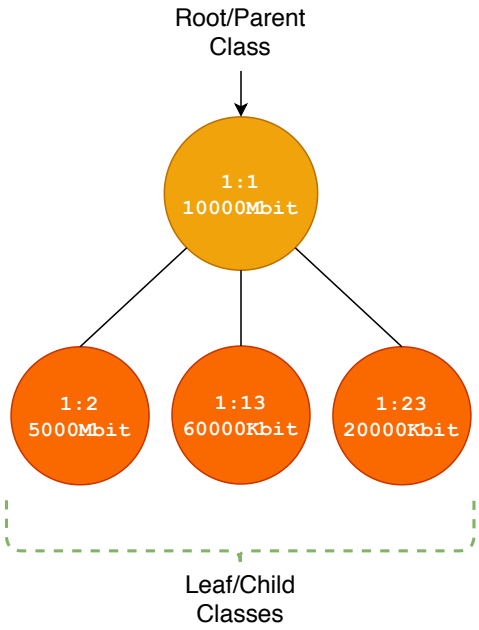

Figure 4.3. HFSC hierarchical class tree for bandwidth. The hierarchical structure consists of a root class, with two levels of leaf classes signified by 1:1 and 1:2. The 1:13 and 1:23 classes reflect bandwidth configuration for two different clients.

for 'downwards' traffic, i.e. server to client, as shown in Figure 4.2. A shell script is used to add the link parameters to the interfaces.

The emulator supports multiple configurations per sub-network on a single interface. The classful feature of HFSC and qdisc allows creation of a tree-like structure for class hierarchy to the same interface. The tree consists of a root which enforces the maximum bandwidth limit for the interface. Leaf classes allow a lower bandwidth which can be subdivided amongst several leaf classes. Figure 4.3 illustrates a hierarchical class tree.

An upper bandwidth limit is applied for all the child classes as a ceiling. A leaf class with an associated parent root class is created and assigned a sequential class number to uniquely identify a bandwidth configuration. Figure 4.4 shows an example of such a configuration. The hierarchical structure consists of a root class followed by two levels of leaf classes signified by 1:1 and 1:2. The 1:1 class forms the ceiling for

```
class hfsc 1: root
class hfsc 1:1 parent 1: sc m1 0bit d 0us m2 10000Mbit ul m1 0bit d 0us m2 10000Mbit
class hfsc 1:23 parent 1:1 leaf 24: sc m1 0bit d 0us m2 20000Kbit ul m1 0bit d 0us m2 20000Kbit
class hfsc 1:13 parent 1:1 leaf 14: sc m1 0bit d 0us m2 60000Kbit ul m1 0bit d 0us m2 60000Kbit
class hfsc 1:2 parent 1:1 sc m1 0bit d 0us m2 5000Mbit ul m1 0bit d 0us m2 5000Mbit
```

Figure 4.4. HFSC classful configuration for bandwidth. The root class 1:1 specifies the upper limit using the sc and ul parameters to 10Gbps, 1:2 class is the default class with a 5Gbps upper limit. The 1:13 and 1:23 classes for two different clients are allocated 60Mbps and 20Mbps, respectively.

the given interface. The configuration is added in the form of a service curve rate (sc) and an upper limit rate (ul). However, a simple setting can be created by using the same bandwidth value for each. The 1:2 class is a child of the 1:1 class and is added as the default class which will be followed if none of the classes are used. The 1:13 and 1:23 classes are the most significant as they reflect bandwidth configurations for two different clients. They are at the same level as the default class with their parent being 1:1. Bandwidth is set to 10Gbps for the parent class and 5Gbps for the default class. The other two bandwidth allocations are 20Mbps and 60Mbps, respectively, for each client.

The next step involves adding a queuing discipline as a leaf for the bandwidth class. A *handle* is assigned to this qdisc such that it can be identified for addition or deletion. The NetEm tool can be applied to a given qdisc to specify the delay for a given branch in milliseconds and packet loss in percentage. The traffic control filter (*tc filter*) is the main tool that filters exiting packets. It uses fields to direct packets to the appropriate qdisc. One can assign a specific configuration branch of bandwidth, delay and packet loss to a given set of packets. This filtering may happen based on any of the fields present in an IP packet. The current application demands filtering based on source and destination IP addresses of the clients and server. The *tc filter* also supports filters based on sub-networks, L2 ports, and MAC addresses.

4.5 Hardware Testbed

To evaluate emulation performance, the clients and server are implemented in hardware. Raspberry Pi Model 3B+ (RPi/Pi) [30] nodes are used to represent the clients and server. The Pi can support multiple applications in its Raspbian Operating System. The Pi is traffic limited by a 100Mbits/second network interface controller (NIC). Tests are performed with traffic within 1-100 Mbits/second.

Figure 4.5 shows the detailed hardware testbed including the features of the Link Configuration Tool and a specific address-based example. Multiple Pis that belong to different LANs are connected to a Ubiquiti EdgeRouter X (EDRX) [39]. The EDRX functions as a physical router connecting all the Pis belonging in different subnets. The EDRX has five interfaces (eth0-eth4), all allowing custom connections. Ubiquiti Networks present a web UI that allows EDRX configuration as a L2 switch or a router, helping define the role of each interface. Two interfaces are connected directly to two Pis to act as separate LAN interfaces. One interface is attached to a WAN interfaced to the emulator via an Ethernet cable. The EDRX wizard enables predefined configuration setup. The emulator VM was implemented on a Dell Optiplex 7010, quad core with 24GB of memory. The system runs Ubuntu Linux and has two 1G NICs. Interface eth0 (Figure 4.5) is connected directly to the physical router (EDRX). Even though the EDRX, ethernet cables and the workstation interfaces support 1G bandwidth, the RPi interface is a bottleneck. The eth1 interface is connected to another RPi, the server. The link configuration function is a Python script that performs multiple functions. The script runs in the VM to parse the *json* file (json parser), adds the IP address information to the IP address constructor, and conveys link properties to another bash script (link parameter tool). These functions will be discussed in Section 4.6. From a user point of view, the emulation setup includes multiple links with unique link properties between the clients and the server even though both clients forward traffic over the same physical interfaces.

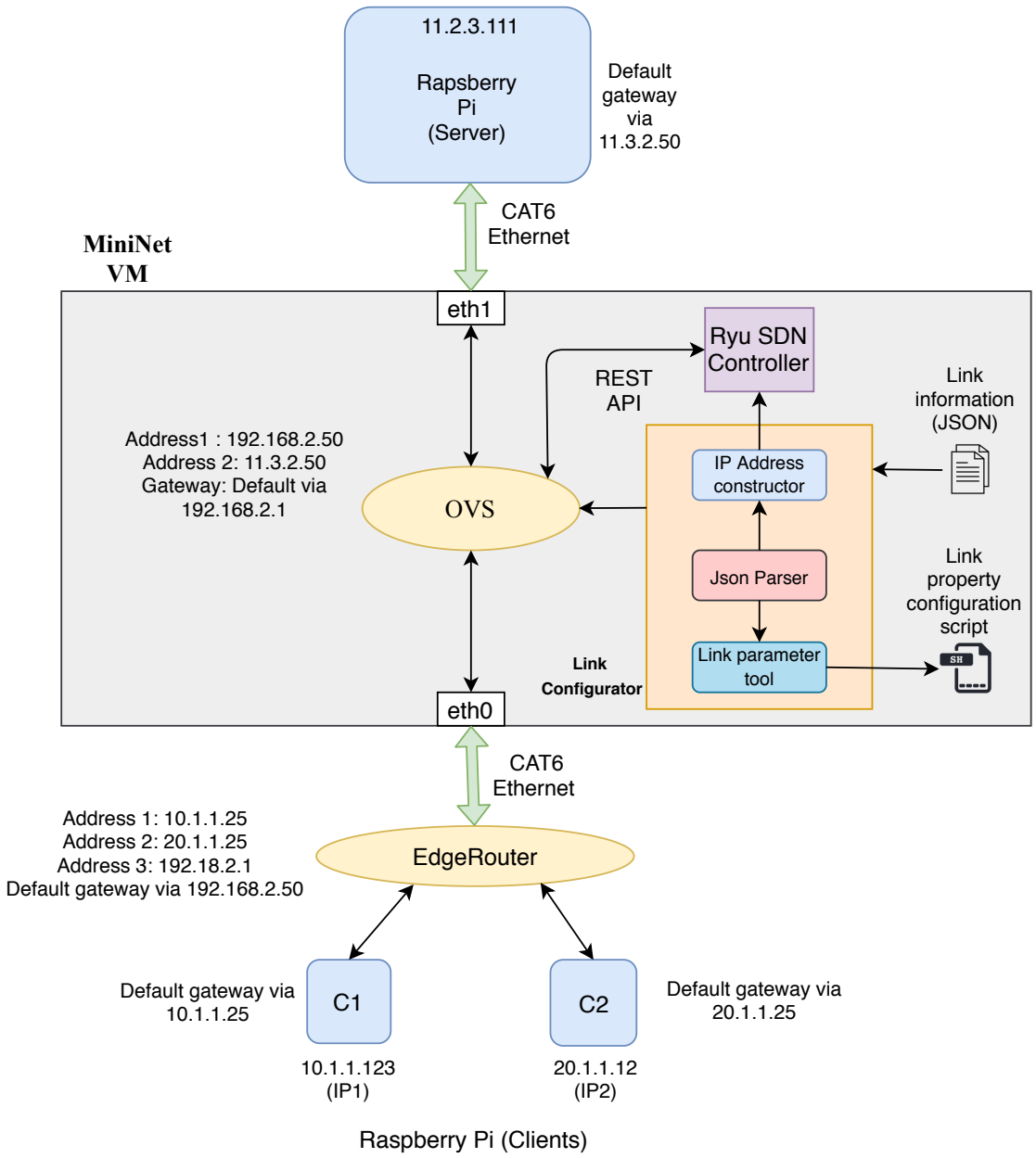

Figure 4.5. Hardware Testbed Diagram with an example scenario. The link configuration tool consists of json parser, IP address constructor and a link parameter tool that interacts with the SDN controller and property configuration scripts. An example scenario with the default gateways and IP addresses are shown.

4.6 Implementation approach

In this section, the details of an emulation test, including a detailed example, are described. A custom Mininet application written with a Python API was used to

generate the topology for emulation. It consists of an OVS bridge with two bridged adapters as ports. It defines the use of a remote SDN controller over TCP port 6033 on the loopback interface of the VM. Once the structure is ready, the *Rest-Router* is launched which recognizes the OVS listening to 6033 port and connects to it. The link configuration tool then starts the process of link construction. A JavaScript Object Notation (*json*) file is used to define the source and destination IP addresses of the end hosts (Figure 4.6). These IP addresses act as the source and destination addresses for packet filtering and provide information to the Ryu-router for configuration. The *json* file also describes link configuration properties. Forward and backward configurations may be required for one-way traffic. Thus, the *json* file defines client Wide Area Network (WAN) IP addresses and the corresponding connected servers in addition to forward and backward path bandwidth, delay and packet loss for each client. An example of the *json* file is shown in Figure 4.6. This example shows a three-client, one server network. The physical router is connected to three clients with IP addresses defined as "Client_IP"s. The "Server_IP" defines the IP address of the server. This information is used by the IP address constructor to configure the Ryu-router.

For each client to server link, the forward and backward path properties are defined in the form: {Link Property Type}_{Client_No}_{direction(forward/backward)}. This information is used by the link property tool and the bash script. To enable routing via the virtual switch (Figure 4.5), gateway addresses are added using the REST-API so that both the physical router and the server can use the OVS as the next hop for all packets. This action is performed by the IP address constructor function of the link configuration tool. For example, the clients have IP addresses 10.1.1.123 and 20.1.1.12 and the server has address 11.3.2.111. The EDRX has a LAN interface with IP address 192.168.2.1 that is added as the default next hop for packets entering the OVS. This establishes a backward path (server-to-client). The Ryu-router

```
{"Linksimulation": {
    "Routerpairs": [{
        "Routerid": 1,
        "Clientcount": 3,
        "Client_IP1": "10.1.1.123",
        "Client_IP2": "20.1.1.12",
        "Client_IP3": "10.2.1.23",
        "Server_IP": "11.3.2.111"
      }

    ],
    "Linkproperties": [{
        "Routerid": 1,
        "properties": [{
          "throughput_1_forward": "40Mbps",
          "delay_1_forward": "1ms",
          "packetloss_1_forward": "0",
          "throughput_1_backward": "0Mbps",
          "delay_1_backward": "0ms",
          "packetloss_1_backward": "0",
          "throughput_2_forward": "50Mbps",
          "delay_2_forward": "5ms",
          "packetloss_2_forward": "0",
          "throughput_2_backward": "0Mbps",
          "delay_2_backward": "0ms",
          "packetloss_2_backward": "0",
          "throughput_3_forward": "30Mbps",
          "delay_3_forward": "3ms",
          "throughput_3_backward": "0Mbps",
          "delay_3_backward": "0ms",
          "packetloss_3_backward": "0"
        }]
```


Figure 4.6. *json* file example for link emulation. The file shows three clients connected to one router. The *Client_IP*s and *Server_IP*s are defined for a given router along with the forward and backward path properties (throughput, delay, packetloss) for each client.

adds an internal IP address 192.168.2.50 in the same LAN as the EDRX. This enables 192.168.2.50 to be the next hop for all incoming packets from the clients on the EdgeRouter. This setup is performed via the EDRX Web UI to establish the forward path for the testbed (client-to-server). The default route configurations for every packet entering the OVS/Ryu-Router can be summarized as follows:

- Default, forward to the EdgeRouter.
- Destination is server, forward to the server
- Destination is client, forward to the EdgeRouter

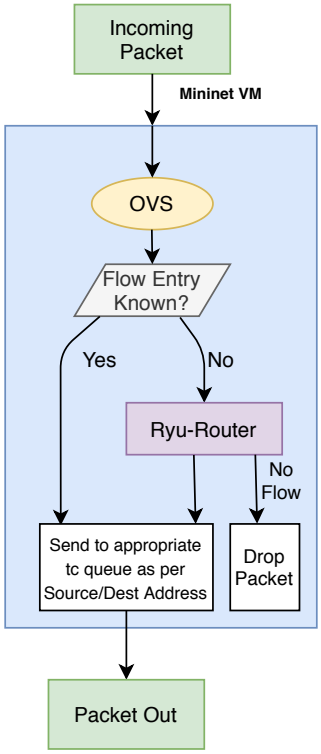

Figure 4.7. Workflow for packet based routing in the Mininet VM through OVS and Ryu-router. Packet received at the OVS is directed to the appropriate queue as per source/destination IP address if flow exists. If flow doesn't exist in the OVS then it is forwarded to the Ryu-router for further processing.

These configurations ensure the establishment of the default forward and backward paths between the client and server. Figure 4.7 shows the function of the Ryu-router for packet based forwarding. When a packet is received at the OVS, if a flow exists, i.e. if the path to the destination is known, it is directed to the appropriate queue designated by the link property configuration script as per the source/destination IP address. If a flow does not exist in the OVS, it is forwarded to the Ryu-router for further processing. The Ryu-router performs routing based on the information in its routing table. If the Ryu-router is unable to find an appropriate destination for the packet, the packet is dropped.

Algorithm 2: Algorithm for configuring links

Data: Link Information with Src IP: src_{IP}, Dst IP: dst_{IP}, Bandwidth: bw, Delay: del, Packet Loss: pl
Result: Add routes/gateway on router and configure link parameters

```
if src_IP to dst_IP route exists then
    Call script to add Link configurations
    if Link parameters previously defined then
        Delete handle used to define the qdisc, netem for del, pl
        Delete hfsc class for old bw
        Add new hfsc class with bw with same classid
        Add new qdisc, netem for del, pl with same handle as the one deleted
    else
        Add new hfsc, qdisc class with bw, del, pl
        Add TC filter for src_IP and dst_IP to the new qdisc class
    end
else
    Add new route/gateway according to src_IP and dst_IP
    Add new hfsc, qdisc class with bw, del, pl for the link
    Add TC filter for src_IP and dst_IP to the new qdisc class
end
```

4.6.1 Link Configuration Tool

The link configuration tool (LC) performs three primary tasks:

- Parse the *json* file (Figure 4.6).
- Extract the source and destination IP addresses and add gateway addresses on the virtual switch via the REST API.
- Extract the link parameters (bandwidth, delay and packet loss) for each set of source and destination IPs. Set them on forward and backward interfaces via script calls.

The link configuration tool was implemented in Python 2.7. It is launched after Mininet topology setup. The next step is the establishment of successful communication between the clients and the server through the EdgeRouter and the link emulator. The addition of link parameters is then performed by invoking a bash script which

completes the initial setup. The link property configuration script for link parameter setup is invoked by the LC link parameter tool.

Any existing configurations on the interface are verified before creating a class, qdisc, and a netm in the TC hierarchy (Algorithm 2). The LC receives the source IP, destination IP and bandwidth, delay and packet loss values. The LC first checks if the route for the destination IP and router already exists. If it does, then the link property configuration script is called. The script then verifies if the link parameters are already defined. If they are, it deletes the NetEm qdisc for the given interface using the "handle". The handle is a unique ID that is defined to identify a qdisc. This handle can be an arbitrary but sequential number for qdisc addition or deletion. Once the qdisc is cleared, the hfsc class is deleted. Then a new leaf class with new link parameters *bw*, class ID, *del* and *pl* is added. A filter is used to redirect packets to a qdisc/hfsc queue uniquely use this class ID. The filter matches the source and destination IP addresses of the packet before forwarding it to the queue.

If the class does not exist, a new leaf class is defined under the root and *bw* is added using the hfsc class. Parameters *del* and *pl* are added using a unique handle such that the hfsc class is a parent. Finally, a TC filter is added to the new qdisc class for *src_IP* and *dst_IP* to establish a flow redirection to the class when a packet is matched. When there are no routes/gateways are present on the Ryu-router, the LC calls the IP address constructor to add the gateways and then follows the same procedure for adding link parameters.

This concludes the initial link setup by the LC. The properties that are currently defined may dynamically be changed in the emulator by the user. In this case, the procedure defined in Algorithm 2 from step 20 to step 23 is followed. Modification using the command line interface is discussed in the next subsection.

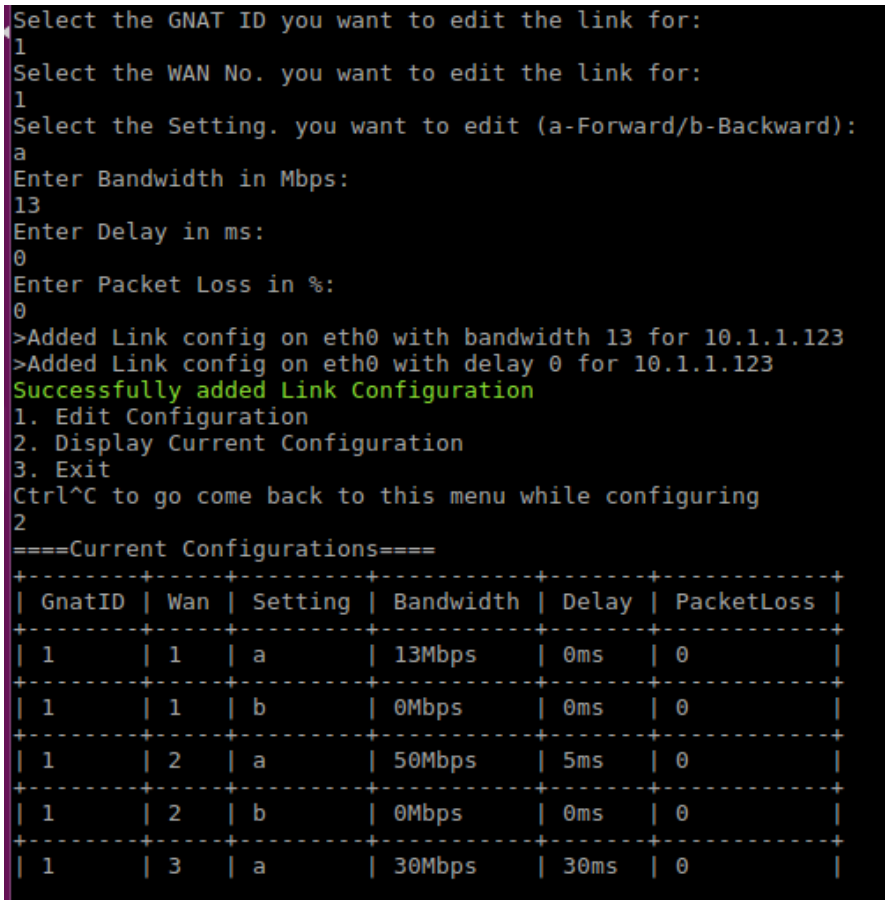

Figure 4.8. Command line interface used for dynamic link property changes. The user can choose from an existing set of client - server pairs and its corresponding forward or backward path. In this case GNAT ID refers to the Router ID and WAN No. refers to the client ID. Setting "a" sets forward link properties and "b" for backward link properties.

4.6.2 Dynamic Link Emulation

There can be two possible modes of operation for dynamic emulation. Either the link parameters are periodically updated via a *json* file input or new parameters are input using a command line interface. The latter approach is followed for preliminary testing since it provides user interaction.

Once the configuration is completed by the LC, a user interface is shown to display and/or change bandwidth, delay, and packet loss parameters. A screenshot of the user command line user interface is shown in Figure 4.8. The user may choose an option

from the existing set of client - server pairs and a corresponding forward or backward path. The LC prompts for user-defined inputs and modifies the emulator interfaces via the TC link configuration scripts described in the previous subsection.

It is important to ensure a modification of either the forward or backward path does not affect the other path. For example, traffic between a client and a server should not affect the workload from another client. If the workload is similar, the bandwidth of the link should be equally shared.

4.7 Hardware-based Evaluation

In this section, results for a set of experiments for the link emulator framework are presented. The emulator is implemented in a VirtualBox VM installed on a 4-core Intel i5-3470 CPU (3.20GHz, 24GB). The Mininet VM runs Ubuntu 16.04.16 LTS and consists of 1 processor and 1 GB RAM. The emulator consumes limited memory since it requires only three main processes, i.e. Open vSwitch, Ryu-Router and Link Configurator. Hence, just a 1 GB RAM is sufficient for the emulation. With all dependent files installed, including OpenvSwitch, Ryu-router and Python, the file consumes about 1.4-1.5 GB of disk space.

The experiments were conducted using the hardware testbed described in the Section 4.5. The testbench consists of two clients, C1 and C2 connected to the server via an EdgeRouter and the link emulator. Tools *iperf* [55] and *ping* [52] were used to evaluate bandwidth and delay metrics. Due to Raspberry Pi interface limitations, the maximum bandwidth is limited to 100Mb/sec. Thus, the operation of the emulator was evaluated in the range of 1-100 Mb/sec. The experiments were performed for each link individually.

4.7.1 Bandwidth Evaluation

Table 4.1 shows the bandwidth configuration for a single link between C1 and the server. In this case, the forward links are configured with a bandwidth setting which is labelled "Bandwidth". The observed bandwidth is noted as "Observed B/W". An *iperf* server is running on the server, while TCP traffic is generated from C1. The client uses the server IP address to send data streams with a default TCP packet size of 1,500 Bytes and a window size of 85KB. The number of transferred bytes and time taken for data transfer are shown in the third and fourth columns as "Bytes Trans." and "Time".

Table 4.1. Bandwidth of the Link Emulator

Bandwidth (Mb/sec)	Observed B/W (Mb/sec)	Bytes Trans. (MB)	Time (s)
1	0.96	1.6	14.2
5	4.78	6.6	11.6
10	9.56	12.2	10.8
20	19.10	25.0	11.0
30	28.70	36.8	10.7
40	38.20	47.8	10.4
50	47.80	59.1	10.4
60	57.40	70.6	10.3
70	66.90	81.8	10.2
80	76.40	93.6	10.3
90	86.00	105.0	10.3
100	93.70	112.0	10.1

The observed bandwidth is slightly lower than the specified amount. This result is primarily due to hardware and memory constraints imposed by the CPU, OS and NIC buffer. Also, the TCP receive window needs time to ramp up speed to the required level which may cause a minor drop in bandwidth. Moreover, the use of queuing disciplines may also cause slight packet loss/corruption.

The test was conducted at requested (configured) bandwidths between 1 Mbps and 100 Mbps. After 100Mbps, link saturation was observed due to the bottleneck caused by the Raspberry Pi hardware interface. A visual representation of the results

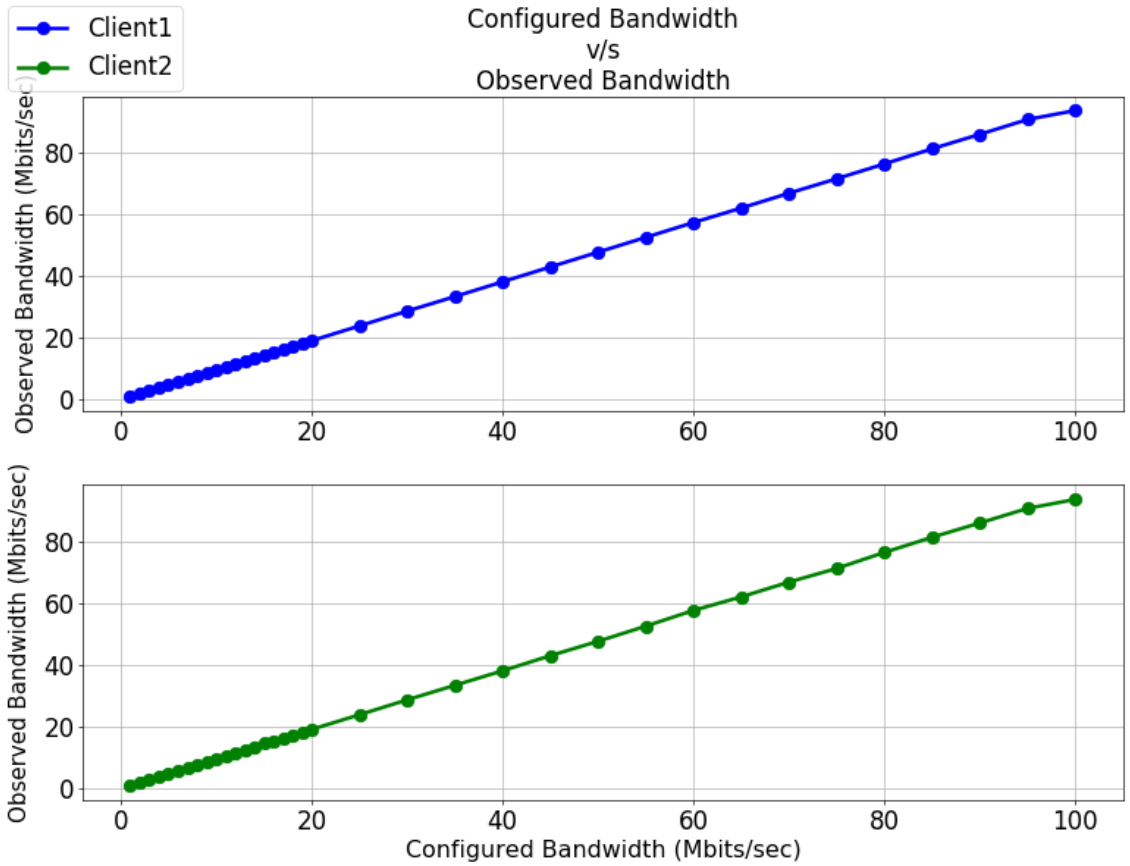

Figure 4.9. Bandwidth (Configured v/s Observed) shown for two clients, C1 and C2. Calculated using *iperf* via TCP connection between each client and the server. A linear trend observed upon reaching saturation after 100Mbps.

are illustrated in Figure 4.9. The bandwidth configuration for C1 and C2 was varied from 1 Mbps to 100 Mbps. At lower data rates, i.e. from 1Mbps to 20Mbps, the granularity of the tests was increased to every 1Mbps to analyse the effect closely. As the figure shows, the configured bandwidth is quite accurately matched by the bandwidth observed via *iperf*. The linear increase in bandwidth with the requested value indicates the accuracy of the link emulator. Both clients show similar results.

Figure 4.10 provides a closer look at the high granularity region in Figure 4.9 for a single client (Client 1). The figure shows the effect of varying the specified bandwidth by 1Mbps steps. The experiment was conducted from 1 Mbps to 20 Mbps. The observed bandwidth shows a linear trend with changes in configured values. An

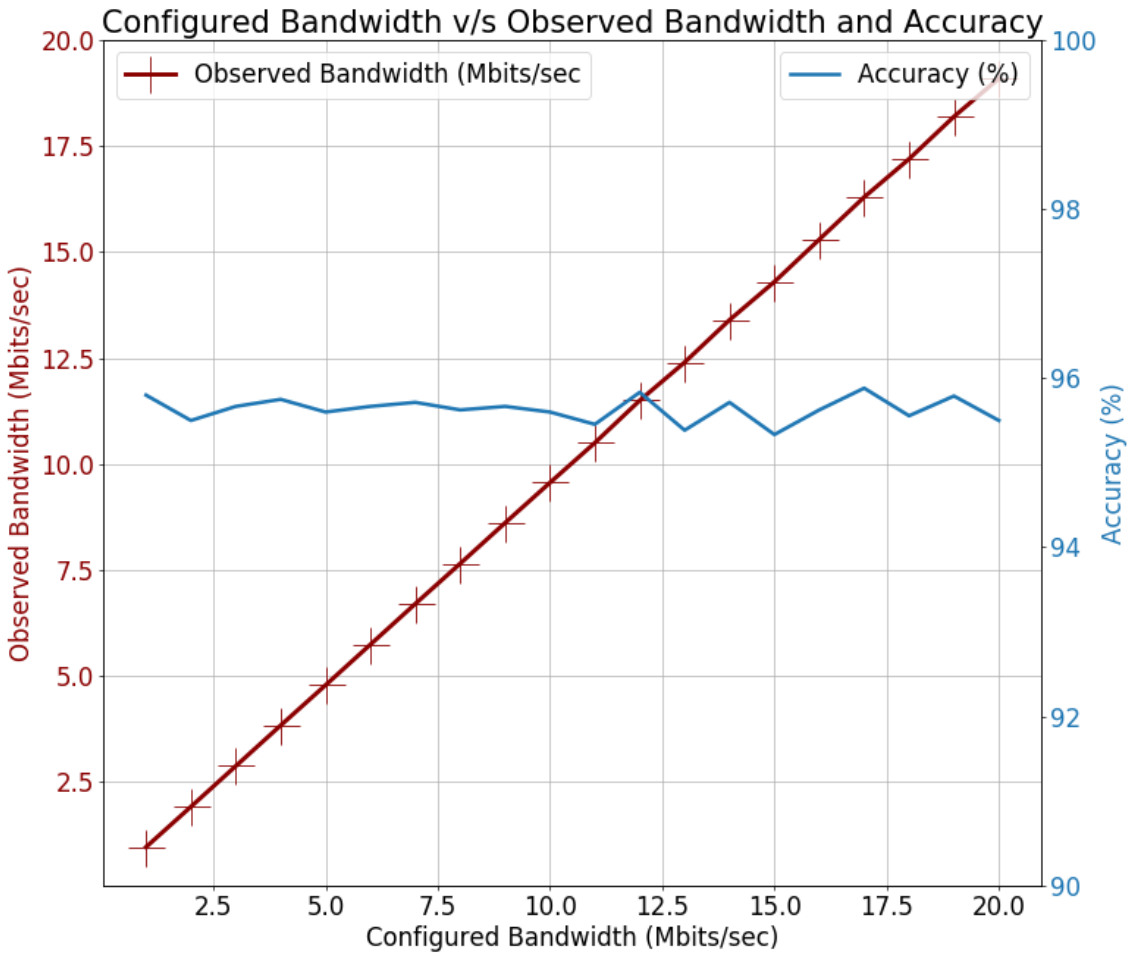

Figure 4.10. Configured Bandwidth v/s Observed Bandwidth and Accuracy. Granular tests from 1 Mbits/second to 20 Mbits/second with an interval of 1 Mbits/second. Ratio of observed to configured bandwidth is around 95%.

accuracy metric is shown in Figure 4.10 on a twin axis. The blue plot signifies the accuracy of the observed to the requested bandwidth based on the following formula:

$$Accuracy = \frac{ObservedBandwidth}{ConfiguredBandwidth} * 100$$

The figure shows a narrow range of accuracy values. The accuracy metric varies with each test from a minimum observed accuracy of 95.3 to a maximum of 95.8. The consistent 5% reduction is likely a consequence of buffer usage and packet dropping.

4.7.2 Delay Evaluation

Latency (delay) and round trip time values were also evaluated with the testbed. The tests were conducted using *ping* commands from the Client 1 Raspberry Pi to the server Raspberry Pi on the hardware testbed. Table 4.2 illustrates the round trip time (RTT) and delay measurements between Client 1 and the server. An overhead of 1.4-1.6 seconds is observed regardless of delay setting. This overhead is a consequence of the hardware testbed, wired links, and packet processing time along the path combined with the routing performed by the SDN controller. This overhead time can be termed as "Inherent Delay".

Table 4.2. Latency values measured by the Link Emulator

Expected Delay (ms)	Observed Delay			
	Min (ms)	Avg (ms)	Max (ms)	stdev (ms)
0	1.46	1.67	2.05	0.135
1	2.61	2.80	3.31	0.258
2	3.75	3.8	3.92	0.061
3	3.74	4.44	6.47	1.044
4	5.64	5.77	6.15	0.204
5	6.66	6.81	7.18	0.216
6	7.68	7.81	8.24	0.168
7	8.62	8.81	9.12	0.148
8	9.66	9.80	10.55	0.282
9	10.61	10.82	11.31	0.217
10	11.51	11.82	12.18	0.211
11	12.66	12.77	12.87	0.132
12	13.68	13.81	13.99	0.107
13	14.65	14.81	14.96	0.168
14	15.53	15.76	15.87	0.090
15	16.70	16.83	17.11	0.205

Configured delay is varied with a 1 ms step. Table 4.2 indicates the delay added on the link by the emulator as "Expected Delay" and the minimum, average, maximum and standard deviation (stddev) of the observed experimental results in milliseconds. Standard deviation is an average of how far each ping RTT is from the mean RTT. The higher the stdev, the more variable the RTT over time. The maximum delay

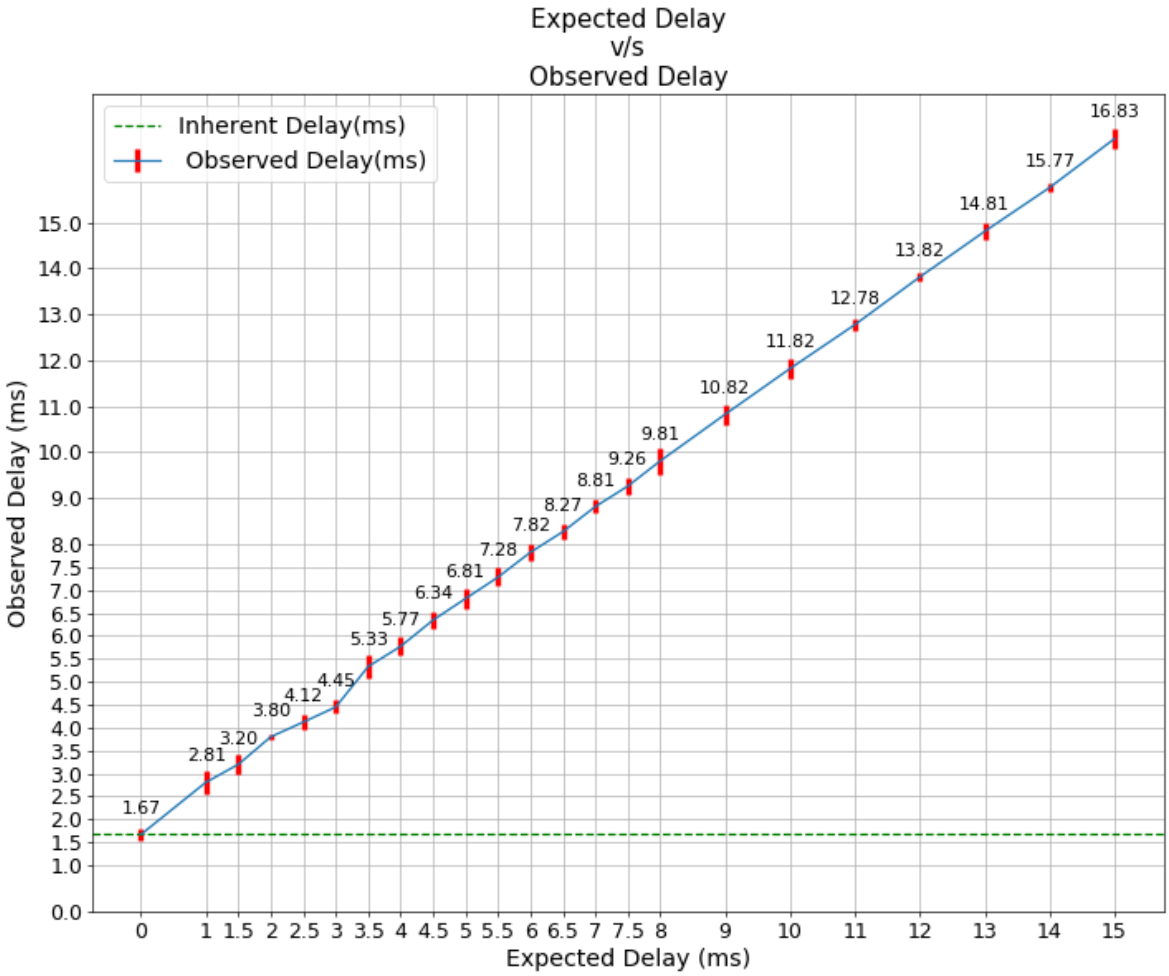

Figure 4.11. Delay Configuration (Expected v/s Observed). Inherent delay exists between the clients and the server due to the presence of hardware network components. There is a linear trend at granular levels (Delay- 0-8 ms) and at higher values. Standard deviation bars for each value are show in red.

is usually the time taken by the first packet to receive a reply since it must find a destination route. Hence the minimum and average values provide a better estimate of actual delay.

For a delay setting of 8ms, an average delay observed was 9.80ms which is 1.8ms over the configured value. Considering an average of 1.67 ms inherent delay, the observed variation is around 130 μ seconds. The standard deviation values measure observed delay fluctuations (Figure 4.11).

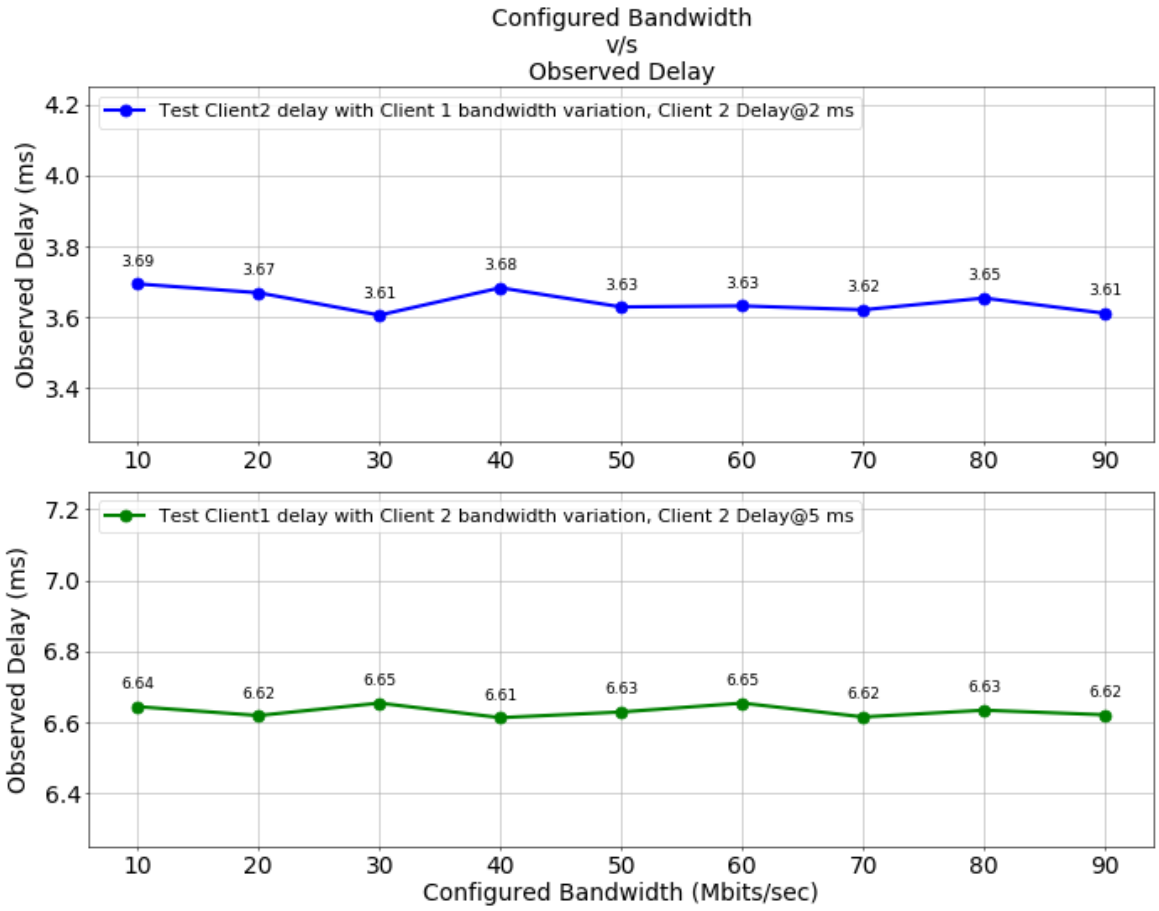

Figure 4.12. Effect of bandwidth variation on observed delay of another client. First plot shows observed delay at Client 2 (which was set constant at 2ms) while increasing the configured bandwidth of Client 1. Second plot shows observed delay at Client 1 (which was set constant at 5 ms) while increasing the configured bandwidth of Client 2.

From 0 ms to 8 ms, experiments were conducted with a step interval of 0.5 ms. The green line indicates the observed delay. The blue line is the average of the observed delay with the expected delay. The red vertical bars for each point indicate the standard deviation for each delay value.

The tests analyzed so far were all conducted for a single link i.e. either client1 - server or client2 - server. To evaluate the effect of running simultaneous traffic from both clients to the server, the following test was performed. One client sends

uninterrupted TCP data streams to the server. At the same time, another client generates and sends ICMP traffic to the server.

The robustness of the link emulator is evident if the observed delay at the second client is unaffected by the TCP traffic sent to the server by the first. For the experiment, Client 1 generates TCP traffic. A static delay value is configured for Client 2. The observed delay at Client 2 is monitored while altering the bandwidth configuration of Client 1.

The results of the experiment are shown in Figure 4.12. The first plot shows the observed delay at Client 2 set at a constant 2 ms, while the configured bandwidth of Client 1 is incrementally increased. The second plot shows the observed delay at Client 1 set at a constant 5 ms, while the configured bandwidth of Client 2 is incrementally increased. The observed delay of both clients under the delay test shows minor delay variation even though the same hardware link to the same server is used. The effect occurs because the hfsc queue and qdisc are separately configured in the link emulator. In the first plot, the delay is around 3.6 ms for a set 2 ms delay, which is appropriate if the inherent delay of 1.6 ms is considered. For the second subplot, the delay is 6.6 ms which is 1.6 ms higher than configured.

4.8 Software Evaluation

In this section, we describe experiments to test the reliability of the link emulator under stressful conditions. The emulator was migrated to a laptop, and the entire system shown in Figure 4.5 was recreated in a VM using software alternatives. VirtualBox VM was installed on a 4-core Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz(16 GiB RAM) laptop. The same Mininet VM as used for hardware-based evaluation is used and runs Ubuntu 16.04.16 LTS and consists of 1 processor and 1 GB RAM and 1 GB disk size.

This evaluation tests the emulator with parallel and bidirectional streams between the clients and server. Parallel and bi-directional streams are prevalent in wired and wireless network emulation applications. Parallel streams can also be found in localised applications that are multi-threaded or transmit different types of data from a client and a server. One example is multiple uploads or downloads occurring from the same computer connected to a centralized office server. Bidirectional streams are prevalent in applications that require constant communication between server and client. An example is controlling a server-based application user interface from a client. In either case, there may be restrictions on certain workstations and users to prevent hogging the common link. Thus, the link emulator can help allot a specific share of throughput for each user.

This approach provides a scenario to test link emulator performance accuracy. Each client should not exceed their allocated throughput. Three different experiments were conducted. First, both the clients communicate with the server in parallel via TCP streams. Second, the number of streams per client are increased and the total allocated bandwidth and fair allocation within the streams is analyzed. Lastly, we run a bi-directional stream from each client for various throughput configurations ranging from 0.1Mbps to 5Gbps.

4.8.1 Experimental Setup

The experiments were conducted using an entirely virtualized testbed to replace the hardware components from Figure 4.5. The software testbed is created in the Mininet VM. It consists of two clients, C1 and C2 which are Mininet virtual hosts. These hosts replace the RPi hardware hosts (Client 1 and Client 2) from Figure 4.5 and are connected to the server, which is another Mininet host, via two software routers. The EdgeRouter is replaced with a software router (Open vSwitch) to connect the two clients, while the Virtual OVS that is used to add the link configurations

remains the same. *iperf* [55] was used to evaluate bandwidth and throughput for the experiments. The single link between the two routers acts as the common trunk link between clients and the server.

Without configuration, the client to server link is found to support 4.77 Gbps bandwidth which is the maximum any client can achieve and will serve as a baseline for the experiments.

4.8.2 Parallel Streams

Two experiments were conducted to evaluate the effect of parallel communication between multiple clients through the link emulator. *iperf* is used to generate TCP streams from the virtual clients to the virtual server. Each stream sends TCP data of size 1,500 bytes per packet at the maximum rate possible.

The first experiment is demonstrated in Figure 4.13. A fixed delay of 1 ms is configured for both clients in all the experiments. The throughput of both clients are then configured to a predefined equal value. An *iperf* test was run from both clients to the server simultaneously to validate the emulator's capability to maintain the allotted bandwidth per-client without affecting each other. This test is performed for four different sets of configured throughputs -(100Kbps-1Mbps), (1Mbps-10Mbps), (10Mbps-100Mbps) and (100Mbps-5Gbps). This provides an insight into the accuracy of the emulator at low granularity and high saturation levels of link configuration requirements, i.e. a throughput of <1 Mbps and >1Gbps.

In Figure 4.13, it is observed that the accuracy of link emulator holds for all levels of granularity and values. Both Client1 and Client2 achieve exact same rate as allotted. Finally for configurations over 2.5Gbps, the link saturates. This is because the available 4.7Gbps of bandwidth is equally divided amongst both clients.

The first experiment only evaluated for one stream per client. The second experiment is demonstrated in Figure 4.14. The number of parallel streams per client are

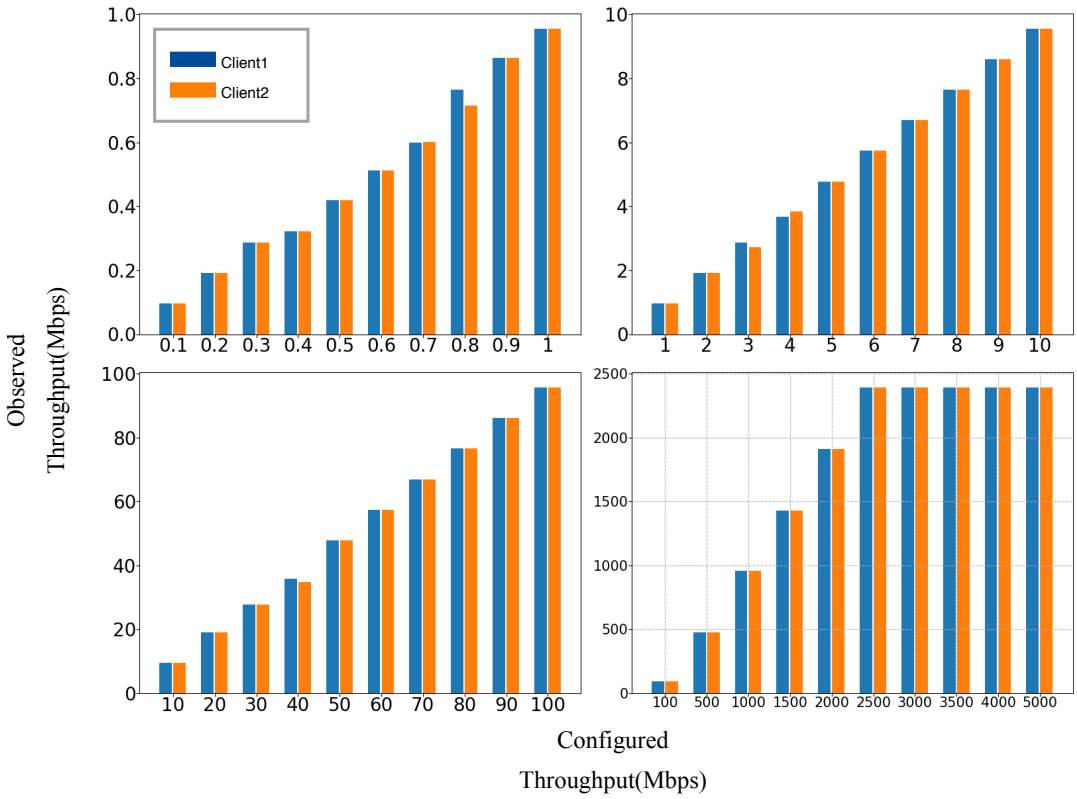

Figure 4.13. A single parallel TCP stream from each client to server working simultaneously. Streams occupy the common link between the two routers. Each client manages to maintain the allocated bandwidth irrespective of each other. Bottom-right plots sees saturation at 2.3Gbps since the total bandwidth of the common link is about 4.7Gbps.

increased sequentially from 1 to 20. While scaling the streams, it is observed that the accuracy per client and fair division of the per-client throughput amongst the TCP streams is maintained. The throughput configuration for both client is preset to 2 Gbps and delay is set to 1 ms. Thus at a time, a client can occupy 2 Gbps of the trunk link with server, which is less than 4.7 Gbps/2.

The fair division of throughput amongst the streams can be represented using a Margin of Error (MoE) metric. This metric provides a overall estimate of how accurately one can justify the fair division of per-client throughput amongst the streams.

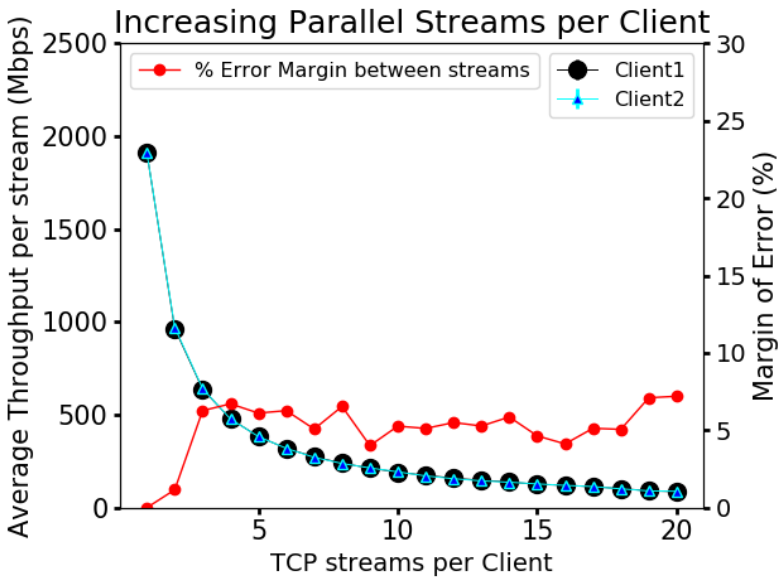

Figure 4.14. Multiple parallel TCP stream from each client to server working simultaneously. Streams occupy the common link between the two routers. Each client manages to maintain the allocated bandwidth, while equally dividing the bandwidth amongst its streams.

For instance, for 10 streams with a client allotted 2 Gbps, the fairest division would have 200 Mbps per stream.

However, in a realistic scenario each of the 10 streams may observe a slightly higher or lower throughput. MoE helps quantify results over several trials, how much each stream may diverge. The lower the MoE, the better the division of resources and the higher the stability amongst the streams.

In Figure 4.14, it is observed that increasing the number of streams from 1 to 20 per client reduces the average per-stream throughput. Both clients follow the same trajectory indicating the fair sharing of link bandwidth irrespective of the number of streams per client. The margin of error, indicated by the red line, hovers around 5-10%. Hence, with an error of 5-10%, each stream achieves an average bandwidth at the ideal division i.e. 2 Gbps / (Number of streams per client). The deviation per stream for both clients increases slightly when the number of parallel streams is

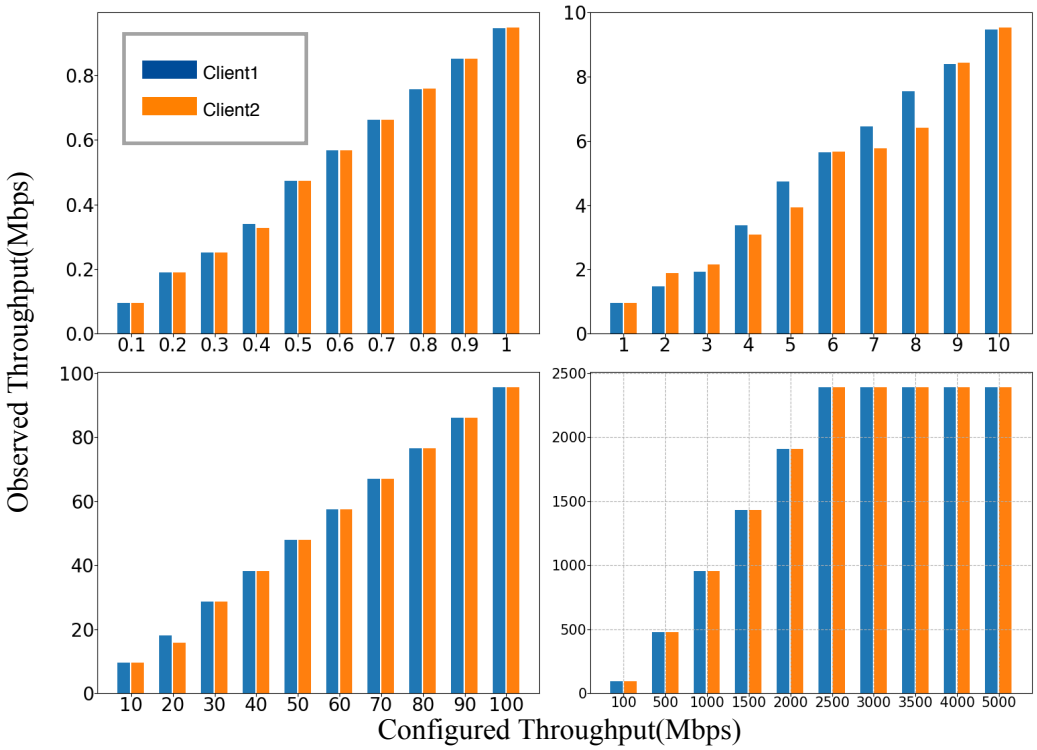

Figure 4.15. Single-bidirectional TCP stream from each client to server working simultaneously. Four streams occupy the common link between the two routers at a time. Each client manages to maintain the allocated bandwidth for its forward stream irrespective of other client and backward stream. Bottom-right plots see saturation at 2.3Gbps since the total bandwidth of the common link is about 4.7Gbps.

increased. This issue might be caused by CPU and memory resource limitations which cause buffer overuse or packet transmission delays for TCP streams. The maximum 7% MoE indicates appropriate bandwidth sharing between the streams.

4.8.3 Bidirectional Streams

In this experiment, bi-directional streams from each client to the server are executed. Thus, for two clients, a total of four streams occupy the trunk link at a time. The structure of the experiment is the same as the structure used for the parallel streams, with the observed throughput compared against configured throughput for different levels of granularity.

Figure 4.15 illustrates the results of bi-directional tests for 4 different sets of configured throughputs -(100Kbps-1Mbps), (1Mbps-10Mbps), (10Mbps-100Mbps) and (100Mbps-5Gbps). The downlink throughput is observed, which provides the true estimation of the achieved throughput.

A linear trend for all sets of allotted throughput vs observed throughput is found for both clients. Moreover, both follow the configuration exclusively. This signifies the stability of the clients under conditions where there is bidirectional data transfer. Finally, for configurations over 2.5Gbps, the link saturates at 2.3Gbps as the available 4.7Gbps of bandwidth is equally divided amongst both clients.

CHAPTER 5

CONCLUSION

5.1 Network Virtualization and Emulation using Docker and Open vSwitch

In Chapter 3, the design and implementation of our emulation framework for mobile ad hoc networks was discussed. The framework provides an encapsulated and isolated environment using nested Docker containers. Open vSwitch is used to create bridges between the nodes and the components of each mobile node. Visibility graphs provide for the dynamic update of links due to positional changes. A communication assistant was developed to facilitate dynamic link changes. A command line interface allows for user interaction to generate and use the emulator.

We have successfully tested its integration with hardware and up to 144 nodes in the MANET topology in the laboratory. Comparisons with ContainerNet shows on-par network performance for typical connections with up to 32 Gbps throughput per link. The use of nested containers has a CPU, memory usage and start-up time overhead, but the nested-containers approach is a suitable model for hierarchies, thanks to improved resource sharing between child containers located within the parent container. Under stressful, high-bandwidth conditions, our emulator showed 26.5% better throughput versus ContainerNet. Overall, NestedNet presents a new paradigm of nested infrastructure in hierarchical network emulation.

5.2 Network Link Dynamic Emulation Testbed

In Chapter 4, the successful design and implementation of a dynamic link emulator was described. The emulator utilizes the traffic control API of Linux OS to implement queuing-based link parameterization. The implementation allows link establishment between multiple subnetworks with each link allotted unique bandwidth, delay and packet loss properties. The Ryu-router and a custom application, the Link Configuration (LC) tool, are used to establish connections and perform link property changes. Host information is fed through a *json* file which contains the source and destination IP addresses of all connections to be created along with their unique bandwidth, delay and packet loss values. The LC's *json* parser changes the specified IP addresses. Link properties are handled via a Linux bash script that uses the Linux *tc* API. The emulator is designed to support link characterization using *tc* filters. The link emulator was evaluated using multiple link property tests.

A fully software-based emulator was created in which all the hosts (clients and server) are virtual. It was tested using parallel and bi-directional TCP streams. Results indicate that under such scenarios, the link emulator can allocate throughput on a per-client basis. For parallel streams, each client is throttled such that all streams share a per-client configured throughput with a deviation of not more than 7%.

CHAPTER 6

FUTURE WORK

6.1 Network Virtualization and Emulation using Docker and Open vSwitch

Our Docker-based emulator provides a solution for sub-component isolation for SDN infrastructure using nested Docker containers. There are additional avenues to explore.

- The scope of this thesis was limited to the emulation of a MANET and its functionality. However, multi-switch, multi-node SDN topologies with heterogeneous nodes and sub-components could be emulated using the nested Docker approach. Since we use Linux, the environmental setup may need to change. Some emulation targets could be a network of complex computer systems or heterogeneous data-center racks.

- A myriad of wired and ad-hoc network protocols exist. These protocols could be evaluated with the emulator.

- A distributed system with hardware switches/routers to accelerate inter-node protocols could be implemented. Currently, the entire emulator is implemented in a server. The emulator could be distributed across multiple workstations to increase scalability and explore more realistic scenarios. The primary development required in this distributed system would be the creation of a synchronized communication assistant.

- Docker containers have mostly been used for virtualization in data-center and microservice applications. Nested containers have generally not been used in the network domain. Fully exploring the use of nested containerization for network emulation has potential.

- Lightweight virtualization technologies such as OpenVZ [43] and Virtuozzo [70] are gaining popularity for network emulation. Replacing Docker with these technologies would be interesting. It is unclear if they support nested virtualization.

- The security of nested containers could be a concern. Each application needs to operate securely in a child container. Security issues for nested containers could be considered for network emulation and other Docker-based virtualization domains.

6.2 Network Link Dynamic Emulation Testbed

The link emulation testbed was designed as a framework to emulate links between network endpoints. The framework was also used to allocate per-user bandwidth and delay to mitigate network congestion. The link emulator could be deployed for other use cases.

- For the the scope of this thesis, two qdisc algorithms were used for link configuration, Hierarchical Fair Service Curve (HFSC) and Netem. Other qdisc algorithms, such as Token Bucket Filter (TBF) [12] and Hierarchical Tocken Bucket (HTB) [21], could be explored. It would be desirable to provide more accuracy than HFSC and support multiple parameters (bandwidth, delay and packet loss).

- It would be interesting to allow for dynamic changes in per link bandwidth. The dynamic change could be time-based or space-based. This change would

allow for the emulation evaluation of path-loss metrics resulting from wireless transmission protocols and models.

- The emulator could support additional network protocols. The IEEE 802.1Q protocol [68] supports VLANs and VxLANS, and is widely used commercially. It would be interesting to evaluate a defined bandwidth/delay allotted per VLAN for centralized server access. Each departmental section may have different bandwidth needs and the entire organization may be divided into smaller VLANs. The emulator could serve as a bandwidth allocation firewall for all server accesses.

- The Ryu SDN controller supports per packet L2-L3 processing in the emulator. This characteristic can be utilized by the emulator as a repeater or a ground station in a network emulation scenario. The work in this thesis assessed the processing of source and destination IP addresses only. Packets received from different sources, with different type of service priority values, VLAN tags, TCP ports, and destinations could be processed differently to support various scenarios.

BIBLIOGRAPHY

[1] Ahrenholz, J. Comparison of CORE network emulation platforms. In *MILCOM 2010 Military Communications Conference* (Oct 2010), pp. 166–171.

[2] Al-Somaidai, Mohammed Basheer, and Yahya, Estabrak Bassam. Effects of linux scheduling algorithms on Mininet network performance. *Communications 3*, 5 (2015), 128–136.

[3] Alvarez, Jose, Maag, Stephane, and Zaidi, Fatiha. Dhymon: a continuous decentralized hybrid monitoring architecture for manets. *arXiv:1712.01676* (2017).

[4] Baclawski, Kenneth. A network emulation tool. In *Simulation of Computer Networks* (1987), vol. 4, Citeseer, pp. 198–206.

[5] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the art of virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[6] Bellard, Fabrice. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), vol. 41, p. 46.

[7] Beshay, J. D., Francini, A., and Prakash, R. On the fidelity of single-machine network emulation in Linux. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (Oct 2015), pp. 19–22.

[8] Binns, Roger. Dynamic real-time view of a running system. `https://linux.die.net/man/1/top`. Last accessed 8 January 2020.

[9] Blywis, Bastian, Günes, Mesut, Juraschek, Felix, and Schiller, Jochen H. Trends, advances, and challenges in testbed-based wireless mesh network research. *Mobile Networks and Applications 15*, 3 (2010), 315–329.

[10] Bob Lantz, Brandon Heller, Nikhil Handigol, and Jeyakumar, Vimal. Mininet: An instant virtual network on your laptop. `http://mininet.org/`, 2020. Last accessed 6 January 2020.

[11] Boettiger, Carl. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review 49*, 1 (2015), 71–79.

[12] Braun, Torsten, Einsiedler, Hans Joachim, Scheidegger, Matthias, Stattenberger, Günther, Jonas, Karl, and Heinrich, J Stüttgen. A Linux implementation of a differentiated services router. In *International Symposium on Networks and Services for the Information Society* (2000), Springer, pp. 302–315.

[13] Carpenter, Tamra, Heyman, Daniel, and Saniee, Iraj. Studies of random demands on network costs. *Telecommunication Systems 10*, 3-4 (1998), 409–421.

[14] Carson, Mark, and Santay, Darrin. Nist Net: a Linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review 33*, 3 (2003), 111–126.

[15] Checconi, Fabio, Cucinotta, Tommaso, Faggioli, Dario, and Lipari, Giuseppe. Hierarchical multiprocessor CPU reservations for the Linux kernel. In *Proceedings of the 5th international workshop on operating systems platforms for embedded real-time applications (OSPERT 2009), Dublin, Ireland* (2009), pp. 15–22.

[16] Community, Ryu SDN Framework. Ryu: A component-based software defined networking framework. `https://osrg.github.io/ryu-book/en/html/rest\_router.html`, 2017. Last accessed 6 January 2020.

[17] De Oliveira, Rogério Leão Santos, Schweitzer, Christiane Marie, Shinoda, Ailton Akira, and Prete, Ligia Rodrigues. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)* (2014), IEEE, pp. 1–6.

[18] Developers, Ubuntu. Virtual Ethernet devices. `http://man7.org/linux/man-pages/man4/veth.4.html`. Last accessed 7 January 2020.

[19] Developers, Ubuntu. Important tools for controlling the network subsystem of the Linux kernel. `https://sourceforge.net/projects/net-tools/`, 2005. Last accessed 7 January 2020.

[20] Developers, Ubuntu. A namespace wraps a global system resource in an abstraction. `http://man7.org/linux/man-pages/man7/namespaces.7.html`, 2020. Last accessed 7 January 2020.

[21] Devera, Martin, and Cohen, Don. HTB Linux queuing discipline manual-user guide. *M. Devera web site, Tech. Rep* (2002).

[22] Docker, Inc. Dind: A nested Docker setup. `https://hub.docker.com/\_/docker`, 2018. Last accessed 7 January 2020.

[23] Docker, Inc. Docker containerization platform. `https://www.docker.com/resources/what-container`, 2020. Last accessed 6 January 2020.

[24] Fielding, Roy. Representational state transfer. `https://restfulapi.net/`, 2000. Last accessed 7 January 2020.

[25] Fink, John. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25 (2014).

[26] Fontes, R. R., Afzal, S., Brito, S. H. B., Santos, M. A. S., and Rothenberg, C. E. Mininet-WiFi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)* (Nov 2015), pp. 384–389.

[27] Foundation, Linux Networking. Classless queuing disciplines. `http://tldp.org/HOWTO/Traffic-Control-HOWTO/classless-qdiscs.html`, 2000. Last accessed 7 January 2020.

[28] Foundation, OpenSource : Linux Networking. Open vSwitch: A distributed virtual multilayer switch. `https://www.openvswitch.org/`, 2009. Last accessed 6 January 2020.

[29] Foundation, Python Software. Python : a programming language that lets you work quickly and integrate systems more effectively. `https://www.python.org/`, 2001. Last accessed 7 January 2020.

[30] Foundation, Raspberry Pi. Single-board computer with wireless LAN and Bluetooth connectivity. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`. Last accessed 13 January 2020.

[31] George, Johann. qperf, measure RDMA and IP performance, 2009.

[32] Gersht, Alexander, and Kheradpir, Shaygan. Real-time bandwidth allocation and path restorations in SONET-based self-healing mesh networks. In *Proceedings of ICC'93-IEEE International Conference on Communications* (1993), vol. 1, IEEE, pp. 250–255.

[33] Guedes, Dorgival, Wundsam, Andreas, Scott, Colin, and McCauley, James. POX: an OpenFlow controller. `https://github.com/noxrepo/pox`, 2007. Last accessed 7 January 2020.

[34] Guo, Chuanxiong, Lu, Guohan, Wang, Helen J, Yang, Shuang, Kong, Chao, Sun, Peng, Wu, Wenfei, and Zhang, Yongguang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference* (2010), ACM, p. 15.

[35] Habib, Irfan. Virtualization with KVM. *Linux Journal 2008*, 166 (2008), 8.

[36] Hemminger, Stephen. Network emulation with NetEm. In *Linux conf au* (2005), pp. 18–23.

[37] Herrscher, Daniel, Leonhardi, Alexander, and Rothermel, Kurt. Modeling computer networks for emulation. In *PDPTA* (2002).

[38] Inc., Docker. Display a live stream of container resource usage statistics. `https://docs.docker.com/engine/reference/commandline/stats/`, 2000. Last accessed 8 January 2020.

[39] Inc., Ubiquiti Networks. A physical configurable router. `https://dl.ubnt.com/datasheets/edgemax/EdgeRouter\_X\_DS.pdf`. Last accessed 8 January 2020.

[40] international standard, ECMA. Javascript programming language. `https://www.json.org/json-en.html`, 1999. Last accessed 7 January 2020.

[41] Kalitay, Hemanta Kumar, and Nambiarz, Manoj K. Designing Wamen: A wide area network emulator tool. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)* (2011), IEEE, pp. 1–4.

[42] Kiess, Wolfgang, and Mauve, Martin. A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Networks 5*, 3 (2007), 324 – 339.

[43] Kolyshkin, Kirill. Virtualization in Linux. *White paper, OpenVZ 3* (2006), 39.

[44] Kurose, James F. *Computer networking: A top-down approach featuring the Internet, 3/E*. Pearson Education India, 2005.

[45] Kwan, Bruce H, Agarwal, Puneet, and Khamisy, Asad. Hierarchical queue shaping, Mar. 6 2012. US Patent 8,130,648.

[46] Lantz, Bob, Heller, Brandon, and McKeown, Nick. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (2010), ACM, p. 19.

[47] Liu, Jianqi, Wan, Jiafu, Wang, Qinruo, Deng, Pan, Zhou, Keliang, and Qiao, Yupeng. A Survey on Position-Based Routing for Vehicular Ad Hoc Networks. *Telecommunication Systems 62*, 1 (2016), 15–30.

[48] Macker, J. P., Chao, W., and Weston, J. W. A low-cost, IP-based mobile network emulator (MNE). In *IEEE Military Communications Conference, 2003. MILCOM 2003.* (Oct 2003), vol. 1, pp. 481–486 Vol.1.

[49] McKeown, Nick, Anderson, Tom, Balakrishnan, Hari, Parulkar, Guru, Peterson, Larry, Rexford, Jennifer, Shenker, Scott, and Turner, Jonathan. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[50] Merkel, Dirk. Docker: lightweight Linux containers for consistent development and deployment. *Linux journal 2014*, 239 (2014), 2.

[51] Mogul, Jeffrey Clifford. System and method for receiver based allocation of network bandwidth, May 6 2003. US Patent 6,560,243.

[52] Muuss, Mike. Test the reachability of a host on an Internet protocol (IP) network. `https://linux.die.net/man/8/ping`, 1983. Last accessed 9 January 2020.

[53] Networks, Nicira. Nox: a C++ OpenFlow controller. `https://github.com/noxrepo/nox`, 2008. Last accessed 7 January 2020.

[54] Networks, Ubiquiti. Physical router. `https://www.ui.com/edgemax/edgerouter-x/`. Last accessed 13 January 2020.

[55] NLANR/DAST. A speed test tool for TCP, UDP and SCTP. `https://iperf.fr/`, 2000. Last accessed 7 January 2020.

[56] Noble, Brian D, Satyanarayanan, Mahadev, Nguyen, Giao T, and Katz, Randy H. Trace-based mobile network emulation. In *ACM SIGCOMM Computer Communication Review* (1997), vol. 27, ACM, pp. 51–61.

[57] Nordstrom, Erik, Gunningberg, Per, and Lundgren, Henrik. A testbed and methodology for experimental evaluation of wireless mobile ad hoc networks. In *First international conference on Testbeds and Research Infrastructures for the Development of Networks and Communities* (2005), IEEE, pp. 100–109.

[58] Peuster, Manuel. ContainerNet API, 2020.

[59] Peuster, Manuel, Kampmeyer, Johannes, and Karl, Holger. Containernet 2.0: A rapid prototyping platform for hybrid service function chains. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)* (2018), IEEE, pp. 335–337.

[60] Rechert, Klaus, McHardy, Patrick, and Brown, Martin A. HFSC scheduling with Linux. *Linux Magazine* (2005), 28–37.

[61] Reina, DG, Askalani, Mohamed, Toral, SL, Barrero, Federico, Asimakopoulou, Eleana, and Bessis, Nik. A survey on multihop ad hoc networks for disaster response scenarios. *International Journal of Distributed Sensor Networks 11*, 10 (2015), 647037.

[62] Rizzo, Luigi. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (1997), 31–41.

[63] Roh, Bongsoo, Han, Myoung-hun, Hoh, Mijeong, Kim, Kwangsoo, and Roh, Byeong-hee. Tactical manet architecture for unmanned autonomous maneuver network. In *MILCOM 2016-2016 IEEE Military Communications Conference* (2016), IEEE, pp. 829–834.

[64] Rosenblum, Mendel. VmWare virtual platform. In *Proceedings of Hot Chips* (1999), vol. 1999, pp. 185–196.

[65] Ryu, Jin-Le, Kwon, Jong-Chan, and Choi, Eun-Ha. Remote controller, May 2015. US Patent App. 29/493,596.

[66] Salsano, Stefano, Ludovici, Fabio, Ordine, Alessandro, and Giannuzzi, D. Definition of a general and intuitive loss model for packet networks and its implementation in the netem module in the linux kernel. *University of Rome* (2012).

[67] Salyers, David C, Striegel, Aaron D, and Poellabauer, Christian. Wireless reliability: Rethinking 802.11 packet loss. In *2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks* (2008), IEEE, pp. 1–4.

[68] Seaman, Mick, Smith, Andrew, Crawley, Eric, and Wroclawski, John. Integrated service mappings on ieee 802 networks. *RFC2815* (2000).

[69] Sharma, S., and Kumar, S. Techniques for real-world implementation of a manet. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)* (Feb 2019), pp. 519–524.

[70] Soltesz, Stephen, Pötzl, Herbert, Fiuczynski, Marc E, Bavier, Andy, and Peterson, Larry. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007* (2007), pp. 275–287.

[71] Staub, Thomas, Gantenbein, Reto, and Braun, Torsten. VirtualMesh: An Emulation Framework for Wireless Mesh and Ad Hoc Networks in OMNeT++. *Simulation 87*, 1-2 (2011), 66–81.

[72] Stoica, Ion, Zhang, Hui, and Ng, TS. *A hierarchical fair service curve algorithm for link-sharing, real-time and priority services*, vol. 27. ACM, 1997.

[73] Suri, Niranjan, Hansson, Anders, Nilsson, Jan, Lubkowski, Piotr, Marcus, Kelvin, Hauge, Mariann, Lee, King, Buchin, Boyd, Mısırhoğlu, Levent, and Peuhkuri, Markus. A realistic military scenario and emulation environment for experimenting with tactical communications and heterogeneous networks. In *2016 International Conference on Military Communications and Information Systems (ICMCIS)* (2016), IEEE, pp. 1–8.

[74] To, Marco Antonio, Cano, Marcos, and Biba, Preng. DOCKEMU–a network emulation tool. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops* (2015), IEEE, pp. 593–598.

[75] Tuteja, Asma, Gujral, Rajneesh, and Thalia, Sunil. Comparative Performance Analysis of DSDV, AODV and DSR Routing Protocols in MANET Using NS2. In *2010 International Conference on Advances in Computer Engineering* (2010), IEEE, pp. 330–333.

[76] Varis, Nuutti. Anatomy of a Linux bridge. In *Proceedings of Seminar on Network Protocols in Operating Systems* (2012), p. 58.

[77] Wang, Zhiheng, Zeitoun, Amgad, and Jamin, Sugih. Challenges and lessons learned in measuring path RTT for proximity-based applications. In *Passive and Active Measurement Workshop* (2003), Citeseer.

[78] Weingartner, Elias, Vom Lehn, Hendrik, and Wehrle, Klaus. A Performance Comparison of Recent Network Simulators. In *2009 IEEE International Conference on Communications* (2009), IEEE, pp. 1–5.

[79] Xu, Kaixin, Tang, Ken, Bagrodia, Rajive, Gerla, Mario, and Bereschinsky, Michael. Adaptive bandwidth management and QoS provisioning in large scale ad hoc networks. In *IEEE Military Communications Conference, 2003. MILCOM 2003.* (2003), vol. 2, IEEE, pp. 1018–1023.

[80] Xue, Qi, and Ganz, Aura. QoS routing for mesh-based wireless LANs. *International Journal of Wireless Information Networks 9*, 3 (2002), 179–190.

[81] Yan, Jiaqi, and Jin, Dong. Vt-mininet: Virtual-time-enabled Mininet for scalable and accurate software-define network emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), ACM, p. 27.

[82] Zhang, Baoxian, and Mouftah, Hussein T. QoS routing for wireless ad hoc networks: problems, algorithms, and protocols. *IEEE Communications Magazine 43*, 10 (2005), 110–117.