Scalable Emulator for Software Defined Networks

by

Arup Raton Roy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Arup Raton Roy

Abstract

Since its inception, Software Defined Network (SDN) has made itself a very appealing architecture for both Data Center and Wide Area networks by offering more automated control through programmability and simplified network operations and management with its centralized control plane. However, extensive rollout of SDN in the production environment requires thorough validation. Thus, there is a compelling need for SDN emulators that facilitate experimenting with new SDN-based technologies (e.g., SDN-based routing and traffic engineering schemes). Valuable insights on these technologies can be gained from real trace-driven experiments on an emulator platform. Accordingly, network operators can gain confidence in these technologies without jeopardizing their infrastructures and businesses.

Mininet, the de facto standard SDN emulator, allows users to emulate an OpenFlow-based SDN on a single server. Due to the physical resource limitations of a single machine, Mininet fails to scale with large network size and high traffic volume. To address these limitations, we developed **D**istributed **O**penFlow **T**estbed (DOT), a highly scalable emulator for SDN. DOT distributes the emulated network across multiple physical machines to scale with large network sizes and high traffic volumes. It also provides guaranteed compute and network resources for the emulated components (i.e., switches, hosts and links). Moreover, DOT can emulate a wider range of network services compared to other publicly available SDN emulators and simulators.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Raouf Boutaba, for introducing me to the field of Software Defined Networks. He always gave me freedom in pursuing what excites me and kept faith in me during my toughest times. My thesis would never come to an end without his motivation, encouragement, and patience. I would like to extend my thanks to the readers, Prof. Martin Karsten and Prof. Khuzaima Daudjee for providing their feedback and comments on this work.

I am immensely grateful to my parents and always feel indebted for their unconditional love, support, and sacrifices. My mom has always been my source of strength. I do not have enough words to thank my loving and caring sister Toma. She took over so many responsibilities so that I can pursue higher studies far away from home. I am also thankful to Choto pishi, Majo pishi, Boro pishi, and Boro pisha for their countless contributions in my life.

I would like to thank Prof. Reaz Ahmed for managing the DOT project and maintaining a meticulous attitude towards the work that significantly improved its quality and usability. I am also thankful to Ian Forster and Thomas Ng, my two mentors at Ericsson Inc., for providing me the internship opportunity during Fall 2013. Their insightful comments, direction, and advice enhanced my critical thinking and grew my interest in networking industry.

I am grateful to Mohamed Faten Zhani for his friendship, and invaluable advices and the mentoring that I received from him from time to time. I express my gratitude to Faizul Bari for closely working with me in all of my research projects and also dragging me back on the track when any of my plans went wrong. In fact DOT, my thesis work, is his brainchild. I am also grateful to Shihabur Rahman Chowdhury from whom I learned a lot about algorithms, development tools, and standard coding practices. I would also express my sincere thanks to my collaborators Qi Zhang and Yeongrak Choi. Moreover, I would also like to thank present and past members of the SAVI project, particularly Md Golam Rabbani, Maissa Jabri, Ahmed Amokrane, and Aimal Khan.

I would like to express my sincere gratitude to Prof. Mohammad Kaykobad, Prof. Mohammed Eunus Ali, and Prof. Sohel Rahman for motivating me to pursue graduate studies. I am also thankful to Tonmoy Bhai for reviewing my application materials when I was struggling to meet the application deadlines for my graduate studies.

I would like to thank Dollar Bhai, Zulfa, Shahed, Nashid, Swastie, Nahian, Kashpia Bhabi, Shapla Bhabi, Liza Bhabi for their invaluable supports during my stay here. I am grateful to my friends: Niloy da, Mun, Biswa, Faiyaz, Swarna, Adnan, Raihan, and Dip.

Without their support, it would have been really difficult for me to leave my family back in Bangladesh.

Finally, I thank the University of Waterloo for providing me with adequate financial and technical support and maintaining an excellent academic environment throughout my stay. This work was supported by the Natural Science and Engineering Council of Canada (NSERC) in part under its Discovery program and in part under the Smart Applications on Virtual Infrastructure (SAVI) Research Network.

Dedication

This is dedicated to my family.

Table of Contents

List of Tables

List of Figures

Publications

Most of the portions of this thesis appeared in the following publications:

- Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba, "Design and management of DOT: A Distributed OpenFlow Testbed" in IEEE Network Operations and Management Symposium (NOMS), 5-9 May 2014
- Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba, "DOT: Distributed OpenFlow Testbed". SIGCOMM Computer Communication Review 44, 4 August 2014, 367-368.

Chapter 1

Introduction

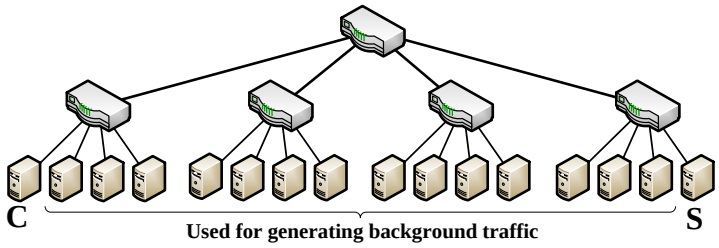
Software Defined Networks (SDNs) have gained a lot of attention from the industry and academia in the last few years. SDN promises to simplify the complexities in network management by adopting two concepts: (i) clean separation of the control and data planes, and (ii) a logically centralized control leveraging a global view of the network.

Improved operational efficiency, lower capital and operational expenses, and higher resource utilization are some key motivations behind the adoption of SDN in the industry [1]. Network equipment and service vendors have developed a number of solutions centered around SDN [2, 3, 4, 5, 6]. Many of these networking solutions have been reported to be deployed on large scale data center and wide area networks [43, 7].

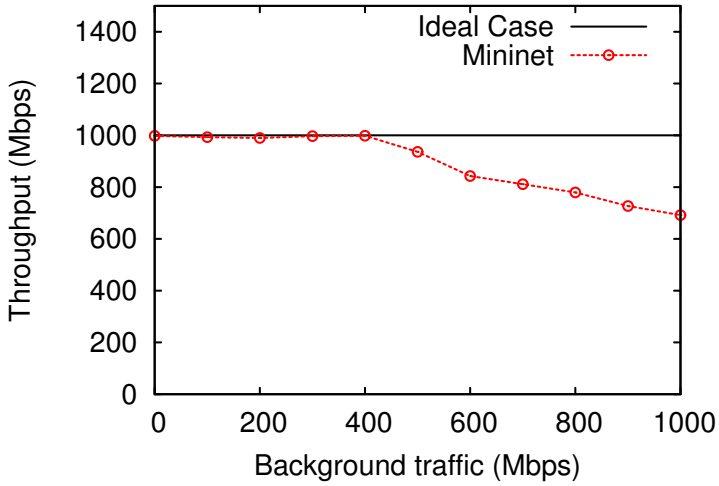
With the growing adoption of SDN solutions, there is a compelling need for SDN emulators that facilitate experimenting with new SDN-based technologies (*e.g.*, SDN-based routing and traffic engineering schemes). Valuable insights can be gained from real trace-driven experiments on an emulator platform. Accordingly, network operators can gain confidence in these technologies without jeopardizing their production networks and businesses.

In this thesis, we present a *scalable* SDN emulator called **D**istributed **O**penFlow **T**estbed (DOT). DOT provides guaranteed CPU time and bandwidth for the emulated components (*i.e.*, hosts, switches, and links). Most notably, DOT is designed for distributed deployment of an emulation from ground up. It deploys a virtual infrastructure (*i.e.*, hosts, links, and switches) of an emulation across multiple physical machines to scale with network size and traffic volume. It also provides built-in support for configuring and monitoring the emulated components.

1.1 Motivation



(a) Network topology



(b) Impact of background traffic

Figure 1.1: Limitation of Mininet

The necessity for a highly scalable emulator rises from the fact that network operators tend to evaluate the performance and behavior of their network under different conditions (*e.g.*, different failure scenarios, with various traffic metrics, *etc.*). Producing *close-to-real-world* emulation results for all possible scenarios is a challenging task from a resource management perspective. If the emulated components are not provided with sufficient isolated physical resources, the resulting resource contention might end up capturing unexpected behaviors that are different from those of real world deployment [50].

Mininet [42] is the *de facto standard* SDN emulator. It emulates entire network on a single machine. To show the motivation for developing a highly scalable emulator, we use

Mininet as a contrasting platform. Here, we present a simple experiment to provide an insight why scalability is a major concern in emulation.

Mininet emulates an OpenFlow based SDN in a single server running all virtual components like virtual hosts(VHs), virtual switches (VSs), and virtual links (VLs). Each node runs as a system process, which consumes small amount of system resources. This allows Mininet to emulate a large network. However, the amount of traffic Mininet can emulate depends on the hardware configuration of the physical server. For example, a dual quad-core 2.4GHz Intel Xeon E5620 processor (12-GB RAM) server can emulate 2.2Gbps traffic, whereas a dual-core 2.1GHz Intel i5 processor (8-GB RAM) laptop can simulate only 976Mbps traffic.

We emulated a fat-tree topology as shown in Fig. 1.1(a) with 16 hosts and 5 switches. We started an UDP `iperf` server on host S and an UDP `iperf` client on host C, generating traffic at 1000 Mbps rate. Then we started 7 iperf client-server pairs in the other 14 hosts uniformly at random to introduce background traffic and measured the throughput of the foreground traffic between S and C. Fig. 1.1(b) reports the result of this experiment.

We can see in the figure that the foreground traffic initially stays at the desired value and gradually decreases with increase in the background traffic. This issue severely limits the applicability of single machine emulators like Mininet. Another point worth noting is that, during this experiment the accumulated traffic is always within the maximum switching capacity of the physical machine (2.2 Gbps), but the foreground traffic keeps decreasing with increase in background traffic.

Traffic can be scaled down to overcome the aforementioned limitation. However, arbitrarily shrinking a network and its traffic to fit into available resources has several limitations. Particularly, it fails to capture the behavior of both the forwarding plane and control plane of SDN. A highly scalable emulator like DOT can eliminate these limitations and produce close-to-real-world behavior of SDN-based technologies.

1.2 Challenges

The design and implementation of DOT involved overcoming many challenges that are described briefly as follows:

1.2.1 Realism

Virtualized instances provisioned for emulation share the resources (CPU, Memory, Bandwidth) of the underlying infrastructure. Thus, absence of proper resource allocation and isolation might result in resource contention. This, in turn, might show unrealistic network behavior that is manifested only during emulation. On the other hand, a real deployment could suffer from issues that may not be apparent in the emulator. Realism ensures that the traits and trends determined in the emulation are comparable to those in a real world deployment.

1.2.2 Scalability

The simplest approach for network emulation is to deploy all the virtual components (*i.e.*, VHs, VSs, and VLs) inside a single physical machine. However, this method affects the performance fidelity of the emulation, thus limits both the network size and traffic volume that can be *faithfully* emulated. Particularly, ensuring realism of an experiment requires resource guarantees for all virtual components. An emulation platform that can run only on a single machine can provide such guarantee only for a small scale emulation. Therefore, scalability is a very attractive feature of an emulation platform for experimenting with large network size and high traffic volume.

1.2.3 Transparency

DOT partitions the virtual infrastructure and deploys it across multiple physical machines. In such partitioning, a virtual link might be mapped to a physical path and might pass through one or more physical switches. Though this partitioning scheme achieves scalability, it brings forth technical challenges in topology discovery for the SDN controller. Specifically, an SDN controller (*e.g.*, Floodlight [8]) generates periodic LLDP messages to discover switch to switch connectivity (Details in Section 2.2). These link layer discovery messages need to be exchanged without any modification between neighboring switches even when they are provisioned in different physical machines. Transparency ensures that no modification is required in the switching engine as well as in the SDN controller for the distributed deployment of a virtual infrastructure.

1.3 Contributions

Our key contributions in this thesis are summarized as follows:

- DOT provides guaranteed allocation of physical resources (*i.e.*, CPU, Memory, and Bandwidth) to the virtual components (*i.e.*, VHs, VSs, and VLs). We address the problem of optimizing the embedding of the emulated topology into a physical infrastructure in presence of this strict resource guarantee. We formulate the embedding problem as an Integer Linear Program (ILP). Then we propose a heuristic algorithm that minimizes both the number of virtual links crossing the network and the required number of physical machines.
- Our emulation platform leverages multiple physical machines to meet the resource requirement of the emulated topology. Each physical machine hosts a partition of the virtual topology, and we propose a technique to stitch these partitions together. We also introduce specialized components to keep this distributed deployment hidden from the users. As a result, all VLs (embedded inside a single physical host or passing through the psychical switches) appear indistinguishable to the user.
- We design and develop a distributed management system that orchestrates the deployment of an emulated network over a cluster of machines. Additionally, we provide centralized access to the deployed emulation for reconfiguration, traffic generation, and event logging through CLIs and APIs.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents an overview of SDN and a summary of prior works and their limitations in emulating large scale SDN. Chapter 3 describes the features, design principles, architecture, and resource management schemes of DOT. Chapter 4 provides performance evaluation of the emulator and its embedding algorithm. We conclude in Chapter 5 by summarizing our contributions and enumerating some possible future research directions. The Appendix describes how to install DOT and explains the steps to run an emulation on it.

Chapter 2

Background

In this chapter, we first briefly introduce SDN in Section 2.1. Then, we explain the topology discovery technique used by the SDN controllers in Section 2.2. Finally, we discuss different approaches for network emulation and list some notable SDN emulators in Section 2.3.

2.1 Software Defined Networks

SDN decouples the control logic from the network devices and places it in a central location in the network. A simplified view of an SDN architecture is shown in Fig. 2.1. Next, we present a brief description of different terminologies in the context of SDN.

-Forwarding Plane

An SDN forwarding device is *dumb* in the sense that it does not implement any routing logic. It treats all the incoming packets according to some *flow* rules installed by a remote controller. If there is no matching rule, a switch contacts its controller through the controller's *Southbound* interface.

The flow rules in a switch can be installed by a controller prior to receiving any packet of a particular flow. This method is known as *proactive* flow rule installation. In the *reactive* approach, as the name suggests, a switch contacts its controller when the first packet of a new flow arrives. The controller then installs the corresponding rule(s) in the switch.

-Control Plane

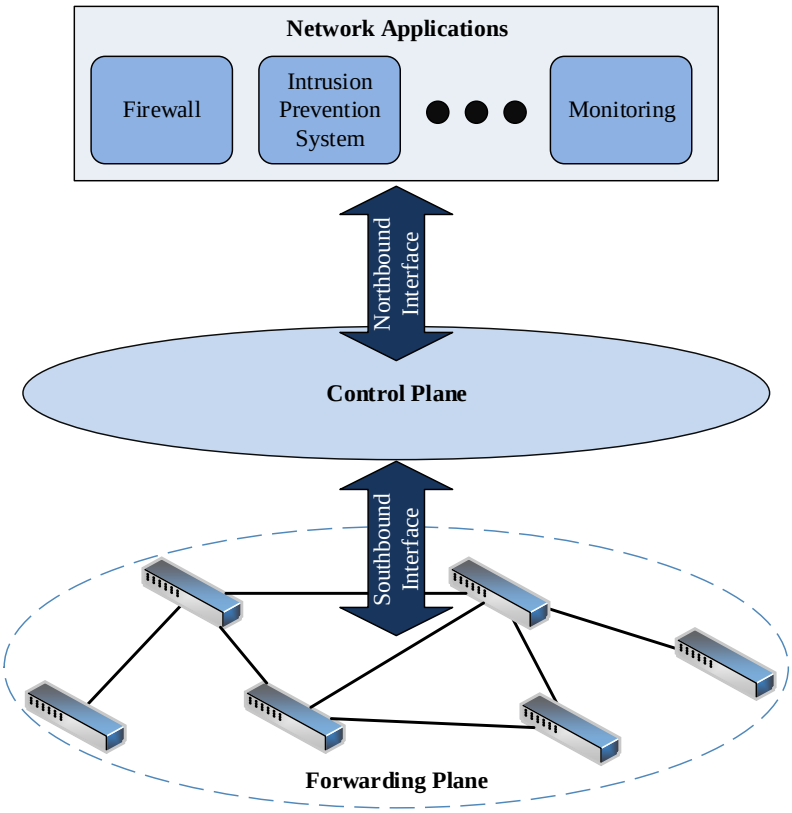


Figure 2.1: SDN Architecture

The network control logic (*e.g.*, routing, packet filtering) of SDN is implemented in a centralized controller. This controller maintains the flow rules of multiple forwarding devices together, periodically collects statistics from the devices, and can make optimized decision leveraging its network wide knowledge.

-Control Channel

The control channel is the network that carries the control traffic (*e.g.*, OpenFlow messages) between the switches and the controller(s). A dedicated control network can be deployed to carry the control traffic. This type of control channel is known as *out-of-band* control channel. On the other hand, *in-band* control channel utilizes the data network without provisioning a separate control network.

-Southbound Interface

Southbound interface comprises a set of well defined APIs and the corresponding communication protocol to exchange messages between a switch and a controller. OpenFlow [23] is the *de facto* southbound protocol for SDN.

-Northbound Interface

The controller exposes a set of APIs for developing network applications (*e.g.*, routing, monitoring). Unlike southbound interface, APIs of northbound interface have not been standardized yet.

2.2 Topology Discovery in SDN

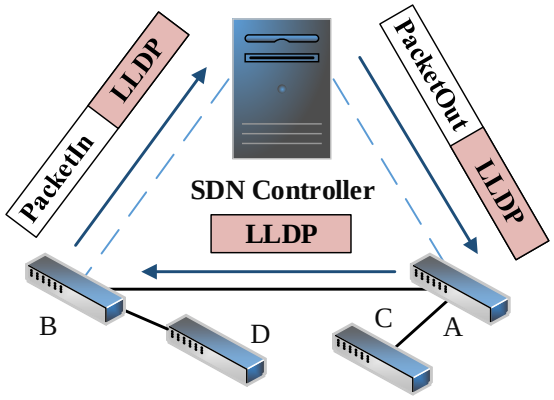


Figure 2.2: Topology Discovery

Here, we explain how the layer 2 connectivity is discovered by an SDN controller. To the best of our knowledge, all the publicly available SDN controllers (Floodlight [8], OpenDaylight [9], POX [10], Ryu [11] *etc.*) use this technique.

A switch is discovered by the controller whenever the transport channel between them is first established. The controller then discovers the physical ports of a connected switch along with its other features using `FeatureReq` and `FeatureRes` OpenFlow messages. Next, the controller determines the switch-to-switch adjacency by generating *periodic* LLDP messages.

At first, the controller creates an LLDP message, puts it in the payload of an OpenFlow `PacketOut` message, and sends it to one of its connected switches (switch A in Fig. 2.2).

The `action` field in this `PacketOut` message tells the receiving switch to forward the LLDP packet to a particular port. The switch (switch B in Fig. 2.2) at the other end of this link (if there is any) receives this packet, encapsulates it in an OpenFlow `PacketIn` message, and sends the encapsulated packet to its controller. The controller then decapsulates the packet and determines the originating switch (*i.e.*, switch A), and discovers the switch-to-switch adjacency (here, link A-B). Note that the LLDP messages traverse only one hop in this technique.

2.3 Emulating SDNs

2.3.1 Network Emulation

Network emulation is a technique to validate new networking architectures and protocols in a *sandboxed* environment before their deployment in a real network. Unlike simulation, emulation runs real code (*e.g.*, real TCP/IP protocol stack) and captures more realistic network behavior. However, this technique requires more resources and might fail to capture the actual performance if proper resources are not allocated for the emulated instances [51].

Network emulators leverage virtualization to emulate end hosts. Some of them use full virtualization, and thus run full-fledged virtual machine (VM) as an end host on top of a hypervisor like XEN [26], KVM [12]. DieCast [40], DOT [51, 52] are examples of such emulators. As each VM runs its own kernel, full virtualization is resource hungry.

On the other hand, emulators like Mininet [45], NetKit [46] uses OS level virtualization. Each VM runs as a system process known as *container*, which leaves very little resource footprint. However, all of them share the same host kernel. This property restricts the types of end hosts that can be emulated over these platforms.

2.3.2 SDN Emulators

Mininet [45], Maxinet [57] and EstiNet [56] are emulators for SDNs. Unfortunately, Mininet emulates the entire network on a single machine, and thus fails to scale with network size and traffic volumes [51]. Maxinet proposes to distribute Mininet over multiple machines, but cannot provide resource guarantee. EstiNet also provides distributed emulation across multiple machines with an added feature of time dilation. However, no details are available regarding the technique used by EstiNet as it is a close-source commercial project.

Chapter 3

DOT: Distributed OpenFlow Testbed

In the previous chapter, we showed that the state-of-the-art network emulators have limitations in experimenting with large scale SDNs. To address these limitations, we developed **D**istributed **O**penFlow **T**estbed (DOT). In this chapter, we present DOT, a horizontally scalable SDN emulator that provisions virtual infrastructure across a cluster of machines.

The remainder of the chapter is organized as follows. We present the key features of DOT in Section 3.1. In Section 3.2, we explain its system architecture. Then, we describe how a virtual infrastructure is provisioned across multiple machines in Section 3.3. Next, Section 3.4 shows how inter-partition traffic is transferred over physical switching fabric. We then present our resource management scheme in Section 3.5. Finally, we provide a summary of this chapter in Section 3.6.

3.1 DOT Features

DOT provides several attractive features that make it more usable for different classes of network emulation. These features also make the deployment and configuration of the emulation easier. A brief overview of DOT features is given below:

- **Scalability:** DOT can be deployed across multiple physical machines. It can be horizontally scaled to meet the requirements of the emulated topology. This feature provides DOT an edge over other emulators.
- **Guaranteed performance:** Our embedding algorithm provides resource guarantee for all emulated components (*i.e.*, VHs, VSs, and VLs). If the physical infrastructure

is incapable of meeting the resource requirement of the emulation, the algorithm does not embed the request. In this way, the performance of the emulated network is kept close to that of a real network.

- **Automated Embedding and Configuration:** A user provides the resources (number of physical machines, their CPU, memory, bandwidth *etc.*) of the physical infrastructure and the requirements (virtual topology, *etc.*) of the emulation in a configuration file. The embedding algorithm then takes this configuration file as input, determines the optimal placement of the virtual components, and deploys them across the cluster of machines. At the same time, DOT also configures different parameters (*e.g.*, propagation delay and bandwidth of the VLs, control plane of the VSs *etc.*) of the deployed virtual infrastructure.

- **SDN from the Ground Up:** DOT supports SDN from the ground up. It creates switching fabric using *Open vSwitch* [13] that has support for the OpenFlow protocol. Additionally, DOT supports both *in-band* and *out-of-band* SDN control planes. A user can deploy her/his preferred SDN controller(s) and sets the required parameters (*e.g.*, IP Address of the controller) in the configuration file. Moreover, VM images with pre-installed popular SDN controllers (*e.g.*, Floodlight [8], POX [10]) are also made available to use within the emulation.

- **Transparency:** DOT inter-connects different partitions of a virtual infrastructure using IP tunnels. It introduces several specialized components to facilitate the distributed deployment and emulation. All these components are kept hidden from the user as well as from the SDN controller(s). Thus, both of them are given the illusion that only the requested virtual infrastructure is being emulated on DOT.

- **Broader Applicability:** Full fledged virtual machines (VMs) can be deployed as end hosts on DOT. In contrast to container based emulation [42], this feature eliminates the dependency on the kernel of the host physical machine and allows user to include any available OS images in the VMs. Although full fledged VMs require more resources than containers [14], they broaden the applicability of DOT. Specifically, they provide flexibility in emulating different classes of network services and algorithms [60, 48, 37, 62, 47, 53, 34, 49, 33, 27, 35, 28, 61, 25, 29, 38]. For example, different publicly available middlebox images [15, 16] can be used on DOT to evaluate different technologies of Network Function Virtualization (NFV) [32].

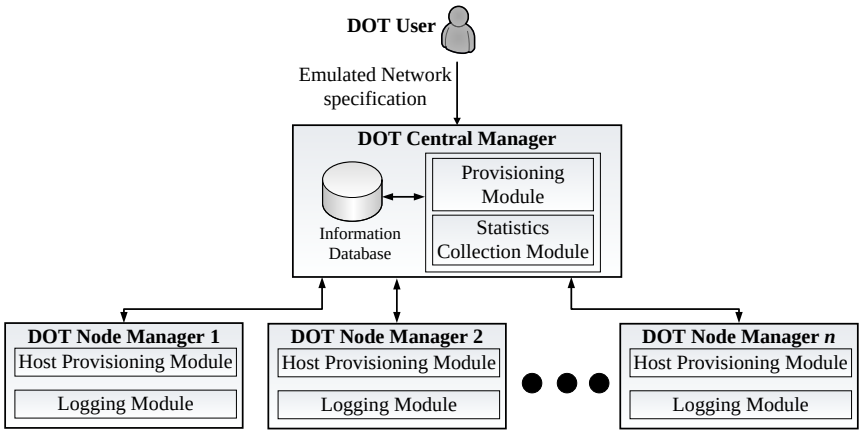


Figure 3.1: DOT management architecture

3.2 System Architecture

In this section, we describe the architecture and different components of DOT. We have two types of physical machines in our system: DOT Manager and DOT Node (Fig 3.3(c)). The DOT Nodes host the virtual instances (VMs, VSs, and VLs) of a user's emulated infrastructure. The operations of the DOT Nodes are orchestrated by the DOT Manager. We developed a distributed management framework for instantiating, configuring and monitoring the emulated components. We describe this framework in Section 3.2.1. Then, we explain the software stack at each physical machine in Section 3.2.2.

3.2.1 DOT Management Architecture

The DOT management architecture (Fig. 3.1) consists of two types of components, namely the central DOT manager, and the DOT node manager (located at DOT Node). In what follows, we provide more details about these components.

1) DOT Central Manager. The DOT Central Manager is responsible for allocating resources for the emulated network as specified by a DOT user. It has two modules, namely the *provisioning* module and the *statistics collection* module. The provisioning module is responsible for running an embedding algorithm that maps the emulated network components to physical resources (*e.g.*, servers and networks). Once the placement of the virtual components is determined, the information is conveyed to the selected nodes. The statistics collection module gathers diverse types of information from logging modules installed on

each of the nodes. Finally, the *information database* stores testbed management information including the current utilization of the cluster, virtual to physical resource mapping as well as collected statistics.

2) DOT Node Manager. A DOT Node Manager is installed on each physical machine and has two modules: *host provisioning* module and *logging* module. The host provisioning module is responsible for allocating and configuring the required resources (*e.g.*, instantiation of VSs, VLs, VMs and tunnels). The logging module, on the other hand, collects multiple local statistics including resource utilization, packet rate, throughput, delay, packet loss, and OpenFlow messages.

3.2.2 DOT Software Stack

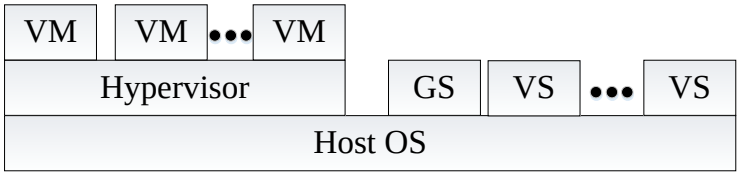


Figure 3.2: Software stack of a DOT node

A DOT node contains VSs and VMs that are responsible for emulating an OpenFlow network. As shown in Fig. 3.2, these virtual components along with the hypervisor, and host operating system compose the software stack of a DOT node. Specifically, a DOT node contains the following components:

- **Hypervisor:** This layer allows to provision multiple VMs on a single physical machine. It also allows to connect VMs to virtual switches using virtual interfaces.
- **Virtual Machine (VM):** Virtual machines are under users' control. A user can deploy OpenFlow controllers or applications (*e.g.*, traffic generation scripts for testing purposes, Web servers *etc.*) on these VMs.
- **Virtual Switch (VS):** Virtual switches are used for emulating OpenFlow switches that belong to the emulated network. Their forwarding rules are populated by the user (or possibly by a controller deployed by the user).
- **Gateway Switch (GS):** Gateway Switch is a special switch created on each physical machine. Its role is to forward packets between virtual switches located at different physical machines. Gateway switches are completely transparent to a DOT user.

3.3 Provisioning a Virtual Infrastructure

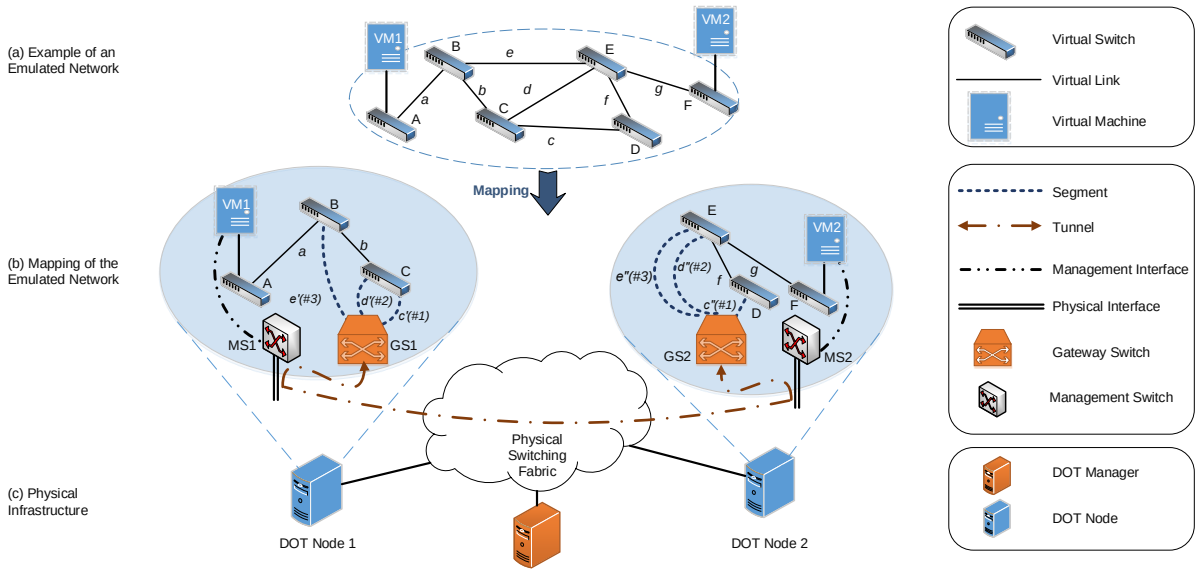


Figure 3.3: Provisioned Virtual Infrastructure

In this section, we explain how DOT distributes the virtual infrastructure across a cluster of machines. Our embedding algorithm (described in Section 3.5) considers the resource capacity of each DOT node and finds an embedding that minimizes physical bandwidth usage and the number of physical machines. The embedding algorithm partitions the emulated network and maps each partition to a particular physical machine.

When the algorithm embeds virtual links, two cases may arise due to partitioning. If the virtual link connects two virtual switches embedded in the same physical machine, then this link is provisioned inside that machine, and hence we call it an *intra-host virtual link*. In turn, a virtual link connecting two virtual switches residing at different physical machines (hereafter called a *inter-host virtual link*) is mapped onto a path passing through the physical network.

Fig. 3.3(a) illustrates an example of an emulated network consisting of 6 VSs, 7 VLs, and 2 VMs. This topology is embedded into 2 physical hosts (Fig. 3.3(b)). Specifically, VSs A, B, and C are allocated in DOT Node 1 and D, E, and F in DOT Node 2. Virtual

links a, b , f, and g are examples of intra-host virtual links, whereas links c , d, and e are inter-host virtual links.

An intra-host virtual link is emulated by instantiating two virtual Ethernet interfaces (`veth(s)`) within the same physical machine (using Linux `add ip link` command). The virtual link bandwidth and delay are emulated using the `tc` and `netem` commands, respectively.

For inter-host virtual links, we create a particular switch called the Gateway Switch (GS) in each physical node for forwarding inward and outward traffic. The process of embedding a inter-host virtual link goes through three steps as follows:

1. We create virtual IP links (with the Linux command `add ip link`) to attach each virtual switch with the gateway switch. In Fig. 3.3, for the inter-host link e, a segment e' is created between virtual switch B and $GS1$ in DOT Node 1 and another segment e'' is created between virtual switch E and $GS2$ in DOT Node 2.

2. Next, a GRE tunnel is created between the physical hosts. A unique identifier is assigned to each inter-host virtual link. In Fig. 3.3, for inter-host virtual link e the identifier 3 is assigned. The GRE tunnel tags every packet forwarded through it with the corresponding identifier. This allows the GS at the other end of the tunnel to uniquely identify the virtual switch that sent the packet.

3. After setting up the tunnels, static flow-entries are created in the GSs at the physical hosts to knit the segments of e together.

An emulated network in DOT also includes a separate (out-of-band) management network. Each VM is equipped with a virtual management interface. This interface is attached to a management switch (`MS1` and `MS2` in Fig. 3.3(b)) of the respective physical host. The management network allows direct access (using `ssh`) to the VMs and ensures that management traffic does not interfere with emulation traffic in any way that affects the outcome of an experiment.

3.4 Traffic Forwarding

In this section, we show how the traffic is forwarded between two remote VMs running on DOT platform. This will provide a precise idea how transparency is ensured by DOT.

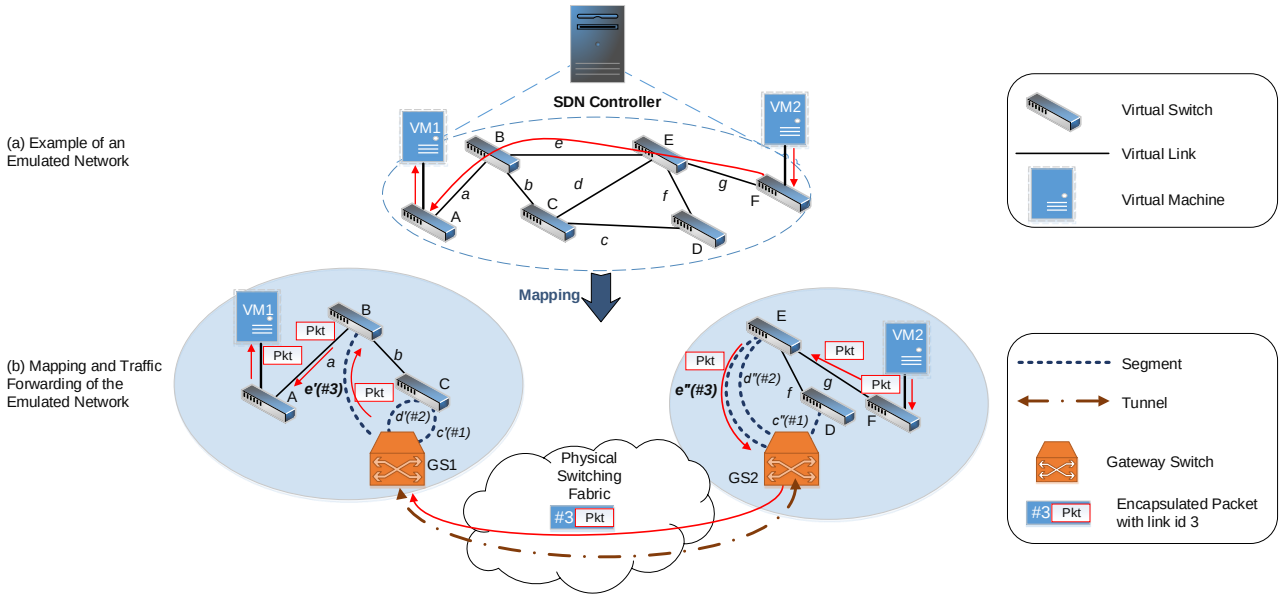


Figure 3.4: Traffic Forwarding

Let us start with a very high level description of the forwarding technique. Whenever a GS receives a packet from one of the segments, it encapsulates the packet within a `GRE` header, tags this packet with the identifier of the segment, and unicasts it to the destination physical machine. The receiving GS (at the other end of the tunnel) decapsulates the packet, inspects the tag, and forwards the packet to the corresponding segment. We pre-install forwarding rules in the GSs to enable the aforementioned mechanism.

Fig. 3.4 explains a scenario where a packet is sent from VM_2 to VM_1 through the path $F \rightarrow E \rightarrow B \rightarrow A$. The switch F first receives the packet and then queries the SDN controller about the route. Here, we assume that it is the first packet of a flow. Now, after detecting the route, the SDN controller installs the forwarding rules at the VSs along the path, particularly at F, E, B, and A. Recall from Section 3.3, the SDN controller is unaware of the responsible GSs for forwarding this packet and does not install any rule in them.

After receiving the forwarding rule, F forwards the packet to E. Now, E forwards the packet through segment e'' thinking that it will go to switch B. However, in reality, this packet reaches $GS2$. Now, $GS2$ inserts the cross-host virtual link identifier for e (here 3) in the `key` field of the GRE header and forwards the packet through the tunnel.

Next, $GS1$ receives this packet from the tunnel and looks at the `key` field in the GRE header. Based on the value of the key (here, 3), $GS1$ selects the segment e' and sends this packet towards switch B. Now, B perceives that it receives this packet directly from E. The tunneling scheme is completely hidden from the VSs also. Next, B forwards the packet to A, and finally A delivers it to VM_1.

3.5 Resource Management

In this section, we address the problem of finding the optimal mapping of an emulated network onto the physical infrastructure. We first formulate the emulated network embedding problem as an Integer Linear Program (ILP). Then, we propose a heuristic algorithm that minimizes both translation overhead and the number of active servers.

3.5.1 Problem Formulation

Table 3.1: Notations

$\tilde{N}$	Set of physical hosts
R	Set of resource types
$\tilde{c}_p^r$	Capacity of physical host $p \in \tilde{N}$ for resource type $r \in R$
$\tilde{b}_p$	Bandwidth of network interface of physical host $p \in \tilde{N}$
$\tilde{\delta}_{pq}$	Propagation delay between physical host p and q
$G = (N, E)$	The virtual infrastructure to be emulated
N	The set of virtual switch
E	The set of virtual links
H	The set of VMs
c_i^r	Resource required by a virtual switch $i \in N$ for resource type $r \in R$
b_e	Bandwidth of virtual link $e \in E$
δ_e	Propagation delay of virtual link $e \in E$
z_i^e	Boolean variable indicating a switch $i \in N$ is one of the ends of link $e \in E$
v_i^h	Boolean variable indicating a VM $h \in H$ is attached to a virtual switch $i \in N$
g_h^r	Resource required by a VM $h \in H$ for resource type $r \in R$

Let $\tilde{N}$ denote the set of physical hosts and $R = \{1...d\}$ the set of resource types (*i.e.*, CPU, memory and disk) offered by each of them. Each physical host $p \in \tilde{N}$ has a capacity

$\tilde{c}_p^r$ for resource type $r \in R$. We denote by $\tilde{b}_p$ the bandwidth of the network interface of physical host $p \in \tilde{N}$. Let $\tilde{\delta}_{pq}$ denote the propagation delay between physical hosts p and q.

In our model, we assume that the physical infrastructure has full bisection bandwidth (e.g., VL2 [39]) and that there is a single path between each pair of nodes. These assumptions simplify the virtual link embedding process. Now, in order to check whether it is possible to embed a virtual link onto a path between two physical hosts p and q, it suffices to check whether there is enough residual bandwidth at the server level. In other words, since the network has full bisection bandwidth, we do not need to check the available bandwidth at the upper layers of the data center network topology.

We model the emulated network as an undirected graph $G = (N, E)$ where N is the set of virtual switches and E is the set of virtual links connecting them. A virtual switch $i \in N$ has a requirement c_i^r for each resource type $r \in R$. Every virtual link $e \in E$ is characterized by its bandwidth b_e and propagation delay δ_e. We define z_i^e as a Boolean variable that indicates whether virtual switch i is one of the ends of link e.

Furthermore, we define H as the set of VMs, and v_i^h as a Boolean variable that indicates whether or not VM h is attached to virtual switch i. We denote by g_h^r the resource requirement of VM $h \in H$ for each resource type $r \in R$. The notations are summarized in Table 3.1.

The problem of emulated network embedding boils down to finding an assignment matrix $X = [x_{ip}]_{|N|\times|\tilde{N}|}$ and a binary vector $Y = \langle y_p \rangle_{p\in\tilde{N}}$, where x_{ip} and y_p are Boolean variables. The variable x_{ip} is equal to 1 if virtual switch i is assigned to physical host p. The variable y_p indicates whether or not physical host p is active (*i.e.*, hosting at least one of the emulated network components). In the following, we focus on computing the resources that need to be allocated in order to accommodate the emulated network to be embedded.

- Resources required by virtual switches

The amount of resources (*i.e.*, CPU, memory, disk) required to accommodate a virtual switch depends on many factors including number and capacity of virtual links connected to it, number of forwarding rules and the amount of traffic it carries. According to the experiments we have conducted, we found that, among these factors, the most determining one is the amount of traffic crossing the virtual switch. Hence, we consider the virtual switch requirements to be proportional to the sum of bandwidth capacities of all virtual links connected to it. Accordingly, the required resources can be expressed as:

$$c_i^r = \sum_{e\in E} z_i^e b_e \rho_r \tag{3.1}$$

where ρ_r is determined empirically through experiments. It is worth noting that it is part of our future work to develop more sophisticated models to capture the relationship between virtual resource requirements and the amount of physical resources to be allocated. It is then straightforward to update our formulation by replacing Eq. 3.1 with the new model.

Furthermore, we should also consider resources required for running the VMs attached to the virtual switches. Indeed, a VM has to reside in the same physical host as the virtual switch to which it is attached. Therefore, we must ensure that there is enough resources in the physical machine to host the virtual switch and its attached VMs. Thus, when embedding a virtual switch, we consider the aggregated resource requirement that encompasses its own requirements and that of its attached VMs. Let $\hat{c}_i^r$ denote the aggregated resource requirement of virtual switch i for resource type r. It can be expressed as:

$$\hat{c}_i^r = c_i^r + \sum_{h \in H} \sum_{i \in N} v_i^h g_h^r \tag{3.2}$$

- Resources required by gateway switches

DOT requires to install a Gateway Switch in each of the active physical hosts to forward the traffic towards other physical nodes. Hence, we need to account for the resources required by the Gateway switch. In our experiments, we found that these resources (mainly CPU) are proportional to the bandwidth capacities of all virtual links going outward from the physical machine (*i.e.*, virtual links connecting two virtual switches hosted by two different machines). Let f_p^r denote the requirement of a gateway switch located at host p for resource type $r \in R$. Hence, we can estimate resource requirement for gateway switches located on physical host p. It can be written as follows:

$$f_p^r = \sum_{i \in N} \sum_{j \in N} \sum_{q \in \tilde{N}} \sum_{e \in E} x_{ip}(1 - x_{jp}) x_{jq} z_i^e z_j^e b_e \rho_r \tag{3.3}$$

- Translation overhead

Packets sent from one physical machine to another undergo encapsulation at the gateway switch. In order to minimize this translation overhead, we need to minimize the number of virtual links using physical network interfaces. In other words, whenever possible, we try to place communicating virtual switches within the same physical host. Thus, translation overhead can be written as:

$$\mathcal{C}^T = \sum_{i \in N} \sum_{j \in N} \sum_{p \in \tilde{N}} \sum_{q \in \tilde{N}} \sum_{e \in E} x_{ip}(1 - x_{jp}) x_{jq} z_{ei} z_{ej} \tag{3.4}$$

- Number of used physical nodes

The number of physical nodes used to embed emulated networks can be expressed as follows:

$$\mathcal{C}^E = \sum_{p \in \tilde{N}} y_p \tag{3.5}$$

Minimizing this number is important for different reasons. First, this allows to reduce resource fragmentation, and thereby makes room for more emulated networks to be embedded. Second, using less physical nodes results in reduced energy consumption.

- Objective function

Given the system model described above, the objective of our optimization problem is to minimize the translation overhead and the number of used physical nodes (Eq. 3.4 and 3.5). It can be written as follows:

$$\mathcal{C} = \alpha \mathcal{C}^T + \beta \mathcal{C}^E \tag{3.6}$$

In Eq. 3.6, α and β are weights used to adjust the importance of individual objectives. Furthermore, the following constraints must be satisfied :

$$\sum_{p \in \tilde{N}} x_{ip} = 1, \ \forall i \in N \tag{3.7}$$

$$x_{ip} \leq y_p \ \forall i \in N, p \in \tilde{N} \tag{3.8}$$

$$f_p^r + \sum_{i \in N} x_{ip} \hat{c}_i^r \leq \tilde{c}_p^r \ \forall p \in \tilde{N}, r \in R \tag{3.9}$$

$$\sum_{i \in N} \sum_{j \in N} \sum_{q \in \tilde{N}} \sum_{e \in E} x_{ip}(1 - x_{jp}) x_{jq} z_{ei} z_{ej} b_e \leq \tilde{b}_p \ \forall p \in \tilde{N} \tag{3.10}$$

$$x_{ip}(1 - x_{jp}) x_{jq} z_{ei} z_{ej} \delta_e \geq \tilde{\delta}_{pq} \ \forall i, j \in N, p, q \in \tilde{N}, e \in E \tag{3.11}$$

$$x_{ip} \in \{0, 1\} \ \forall i \in N, p \in \tilde{N} \tag{3.12}$$

$$y_p \in \{0, 1\} \ \forall p \in \tilde{N} \tag{3.13}$$

Constraint 3.7 guarantees that each virtual switch is assigned to exactly one physical host. We also ensure that a physical node is active if it hosts at least one virtual switch (Eq. 3.8). Furthermore, Eq. 3.9 ensures that physical host capacities are not exceeded. Constraint 3.10 indicates that the sum of bandwidth requirements of inter-host virtual links using the same network interface should not exceed its bandwidth capacity. Finally, Eq. 3.11

ensures that if a virtual link is mapped onto a path between two different physical nodes, its delay requirement is satisfied.

This optimization problem is $\mathcal{NP}$-hard as it generalizes the *Multi-dimensional Bin-packing Problem* [31]. Hence, in the following, we provide a simple yet effective heuristic to solve it.

3.5.2 Heuristic solution

In this section, we present the heuristic algorithm used by DOT for embedding emulated networks. The goal is to find a feasible mapping that minimizes the translation overhead and the number of used physical hosts as dictated by the objective function (Eq. 3.6). This is illustrated by Algorithm 1. This embedding algorithm guarantees maximum bandwidth requirement of each link, and thus it is oblivious of traffic type.

Algorithm 1 Emulated Network Embedding

$\tilde{N}_a \leftarrow$ Set of active physical hosts
$N_u \leftarrow N$ {Set of unassigned switches}
while $N_u \neq \emptyset$ **do**
  Sort N_u in decreasing order according to $\mathcal{R}_i$ (Eq. 3.17)
  $i \leftarrow$ first node in N_u
  $\tilde{N}(i) \leftarrow$ hosts $\tilde{N}_a$ satisfying resource requirements of i.
  if $\tilde{N}(i) \neq \emptyset$ **then**
    Sort $\tilde{N}(i)$ in decreasing order according to $\mathcal{F}_{ip}$ (Eq. 3.20)
    $p \leftarrow$ first node in $\tilde{N}_i$
  else {Need to switch on a physical machine}
    Activate physical host p that satisfies resource requirement of i, and if not possible set $p = -1$.
  end if
  if $p \neq -1$ **then**
    $\tilde{N}_a \leftarrow \tilde{N}_a \cup \{p\}$
    Assign virtual switch i to physical machine p
    $N_u \leftarrow N_u \setminus \{i\}$
  else
    return the emulated network is not embeddable
  end if
end while

Given an emulated network, virtual switches are selected one by one according to some policy. Each selected switch is assigned to one of the active hosts that satisfies the switch requirements in terms of CPU, memory, disk, bandwidth and propagation delay (between the virtual switch under consideration and previously embedded ones). A new host is turned on if active nodes are not able to satisfy these requirements. However, the whole emulated network is rejected if there is no feasible embedding for all of its components. In the following, we provide more details on the policies used to decide the embedding order of virtual switches and to designate the hosting physical machines.

- Virtual switch selection. In order to decide on the embedding order of the virtual switches, we define multiple guiding policies. For instance, it is intuitive that virtual switches with high connectivity are difficult to embed. Hence, it is better to embed them first since this might increase chances to embed their neighbors within the same physical machine, resulting in less number of inter-host links. Thus, we define the degree ratio of virtual switch i as:

$$\mathcal{R}_i^D = \frac{\sum_{e \in E} z_i^e}{\max_{j \in N} \sum_{e \in E} z_j^e} \tag{3.14}$$

The second policy we define to characterize a virtual switch is the resource ratio given by:

$$\mathcal{R}_i^C = w_b \frac{\sum_{e \in E} z_i^e b_e}{\max_{j \in N} \sum_{e \in E} z_j^e b_e} + \sum_{r \in R} w_r \frac{\hat{c}_i^r}{\max_{j \in N} \hat{c}_j^r} \tag{3.15}$$

The intuition here is that it is always more difficult to accommodate high resource demanding virtual switches. Hence, the need to consider their embedding first. The resource ratio value may be adjusted using the weights w_b and w_r that should reflect the scarcity of the resource.

Furthermore, we also try to embed first virtual switches whose neighbors are already embedded. This increases the likelihood that they are hosted within the same machine, and thereby reduces the number of inter-host links and the consumed physical bandwidth as well. We define this locality ratio as:

$$\mathcal{R}_i^N = \frac{\sum_{j \in N} \sum_{e \in E} \sum_{p \in \tilde{N}} z_i^e z_j^e x_{jp}}{\sum_{j \in N} \sum_{e \in E} z_i^e z_j^e} \tag{3.16}$$

Finally, switch selection is based on the ranking computed as the weighted sum of the aforementioned ratios as follows:

$$\mathcal{R}_i = \gamma_D \mathcal{R}_i^D + \gamma_B \mathcal{R}_i^B + \gamma_N \mathcal{R}_i^N \tag{3.17}$$

In Eq. 3.17, γ_D, γ_R and γ_N are weights used to adjust the influence of each factor.

Algorithm 1 evaluates R_i for all unembedded virtual switches and selects to embed first the one providing the highest value. In the next step, we select the physical machine that is going to host the selected virtual switch.

- Physical host selection.

Once the virtual switch to be embedded is selected, the hosting physical machine is chosen based on two criteria. The first criterion is to select the host with lowest residual capacity computed for each host p as follows:

$$\mathcal{F}_{ip}^R = \sum_{r \in R} w_r \frac{\min_{q \in \tilde{N}} u_{iq}^r}{u_{ip}^r} \tag{3.18}$$

In Eq. 3.18, u_{iq}^r is the estimated residual capacity for resource r in physical node q when it is hosting virtual switch i. Minimizing residual capacities of physical machines results in less resource fragmentation and higher machine utilization.

Furthermore, in order to minimize the number of inter-host links, we try to place selected virtual switch at a physical node hosting the maximum number of its neighbors, *i.e.*, maximizing the locality ratio:

$$\mathcal{F}_{ip}^N = \frac{\sum_{j \in N} \sum_{e \in E} \tilde{x}_{jp} z_i^e z_j^e}{\max_{q \in \tilde{N}} \sum_{j \in N} \sum_{e \in E} \tilde{x}_{jq} z_i^e z_j^e} \tag{3.19}$$

In Eq. 3.19, i is the selected virtual switch and p is a physical node. Finally, machine selection is based on its ranking computed as the weighted sum of the aforementioned ratios, as follows:

$$\mathcal{F}_{ip} = \lambda_R \mathcal{F}_{ip}^R + \lambda_N \mathcal{F}_{ip}^N \tag{3.20}$$

In Eq. 3.20, λ_R and λ_N are the weights of each factor in the selection criterion. Specifically, the physical machine p with the highest F_{ip} is selected to host the virtual switch i.

The worst case running time of this algorithm is $O(|N|^2 \log |N| + |N||\tilde{N}| \log |\tilde{N}|)$, where N and $\tilde{N}$ are the number of virtual switches and active physical hosts, respectively. The while loop at line 3 takes at most $|N|$ rounds, and the worst case complexity of the two sorts at lines 4 and 8 are $O(|N| \log |N|)$ and $O(|\tilde{N}| \log |\tilde{N}|)$, respectively.

3.6 Summary

DOT is a horizontally scalable SDN emulator supporting emulation of a wider class of network services compared to other publicly available SDN emulators and simulators. To emulate a large scale SDN, DOT partitions the virtual infrastructure and deploys each partition in separate physical machine. Our embedding algorithm ensures resource guarantee of the virtual infrastructure and aims to optimize physical resource allocation. It rejects an embedding request, if it fails to satisfy its resource requirement. DOT uses IP tunneling to forward the inter-physical machine traffic. Introduction of gateway switch in each physical machine keeps this tunneling hidden from the SDN controller, thus ensuring transparency.

Chapter 4

Implementation and Evaluation

In this chapter, we first describe the properties of the testbed where the subsequent experiments are run in Section 4.1. Section 4.2 compares DOT with Mininet in terms of resource guarantee. In Section 4.3, we evaluate the performance of our embedding heuristic. We demonstrate the capabilities of DOT in reproducing results from a number of networking research papers in Section 4.4. Finally, we present a summary of our evaluation in Section 4.5.

4.1 Implementation Details and Testbed Setup

All software components used in DOT are open source. For emulating virtual switch, we use Open vSwitch version 1.9. We use Linux `tc` command to simulate bandwidth limit and link delay on the virtual links. We use KVM [12] for machine virtualization and Libvirt 1.0.0 [17] library to provision VMs. Each VM runs *tiny core* Linux [18] that offers a minimalistic flavor of Linux and has a very small resource footprint. We use Floodlight [8] controller for the experiments in this thesis. However, any existing OpenFlow controller (*e.g.*, POX [10], OpenDaylight [9]) can be deployed on DOT.

We developed a management module in C++ that processes the input network topology and provides an efficient embedding using the proposed heuristic (Algorithm 1). We choose the values of α and β as 1 and 100, respectively. The rest of the parameters (γ_D, γ_B, γ_N, λ_R, and λ_N) are all set to 1. This module is publicly available at https://github.com/dothub/dot.

We have deployed DOT on 10 physical machines organized in two racks. Each of the machines has dual quad-core 2.4 GHz Intel Xeon E5620 processor and 12-GB of RAM. The DOT manager has been deployed on a separate host connected to the same network. All machines run Ubuntu 12.04 as the host OS.

4.2 Resource Guarantee

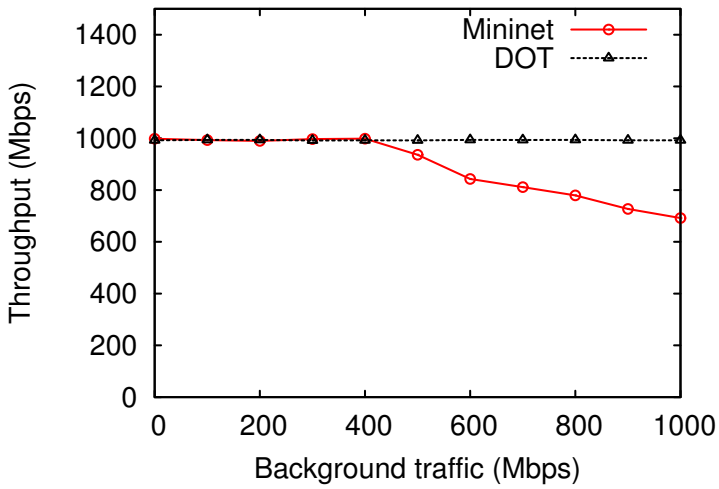


Figure 4.1: DOT vs. Mininet

In our first experiment, we deployed the fat-tree topology shown in Fig. 1.1(a) on DOT (spanning two physical hosts) and performed similar experiment as in Fig. 1.1(b). We started an UDP `iperf` server on host S and an UDP `iperf` client (sending at 1000 Mbps) on host C. Then we started 7 iperf client-server pairs on the other 14 hosts and measured the throughput of the traffic between C and S. The result of the experiment is shown in Fig. 4.1. As we can see from the figures, and unlike Mininet, foreground traffic in DOT is not affected by the background traffic. The embedding process of DOT ensures that every physical host has enough resources to accommodate the compute, memory, and bandwidth requirements of embedded virtual hosts and switches.

Table 4.1: Characteristics of the simulated ISP Topologies

Topology	# of Switch	# of Link
AS-1221	108	306
AS-1239	315	1944
AS-1755	87	322
AS-3967	79	294

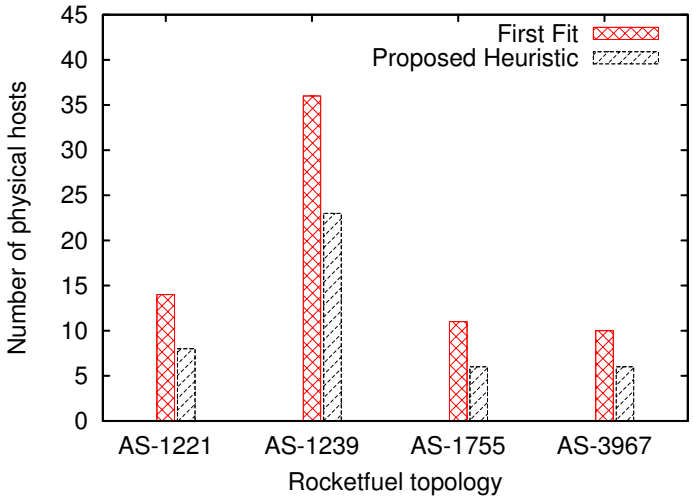


Figure 4.2: Number of physical hosts

4.3 Performance of Embedding Algorithm

Here, we evaluated the performance of our proposed heuristic algorithm against that of *First Fit* bin packing heuristic [44]. We embedded four ISP topologies (Table 4.1) from the RocketFuel repository [54]. We computed the number of physical hosts required to embed these topologies. The result is shown in Fig. 4.3. For AS-1221, FF requires 14 hosts whereas our heuristic requires only 8 hosts. For AS-1239, FF requires 36 hosts and our approach requires only 23 hosts. Similarly, for the other topologies our heuristic consistently requires much less number of physical hosts than FF.

To show the effectiveness of our proposed heuristic over FF, we measure the number of cross-host links and total bandwidth of cross-host links for each physical host. If a host has fewer cross-host links then we have to provision less compute resources for the gateway switch (Eq. 3.4) and as a result more virtual switches can be embedded on the

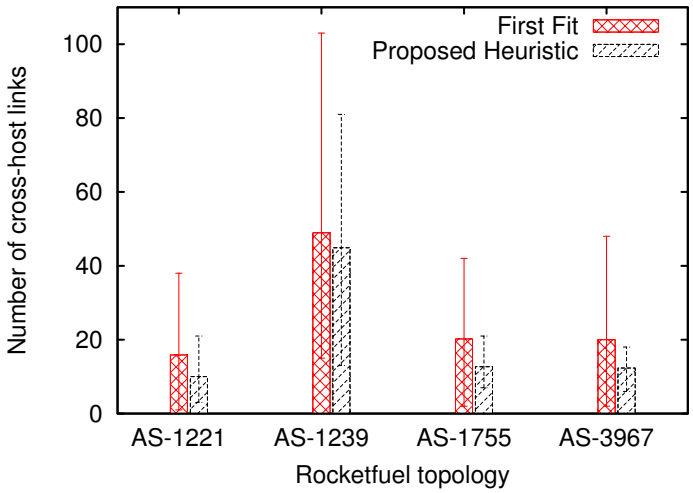


Figure 4.3: Number of inter-host virtual links

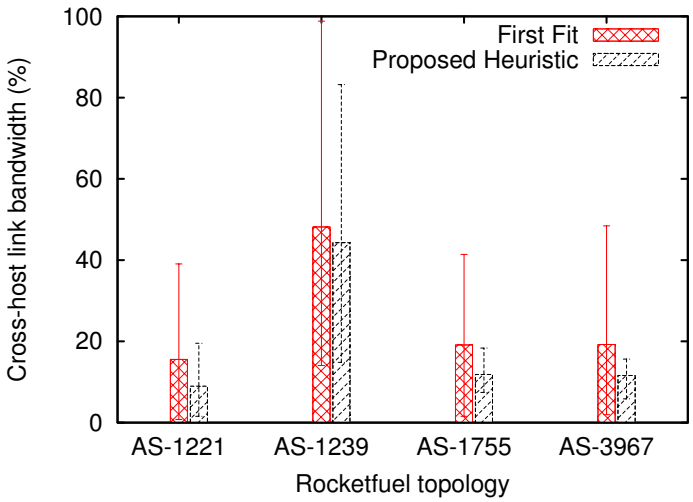


Figure 4.4: Amount of inter-host bandwidth

same physical host leading to a more compact embedding. Fewer cross-host links also indicate that the embedding process is able to embed highly connected portions of input topology on the same physical host.

Fig. 4.3 reports the average, minimum, and maximum number of cross-host links for both FF and our proposed heuristic. We can see from the figure that for all four topologies

our proposed heuristic produces embedding configurations that require much less cross-host links than FF. For AS-1221, AS-1239, AS-1755, and AS-3967 the reduction in average number of cross-host links is 37%, 8.3%, 37.2%, and 38.3%, respectively. The maximum number of cross-host links over all physical hosts for AS-1221, AS-1239, AS-1755, and AS-3967 is 38, 103, 42, and 36, respectively for FF. However, for our proposed heuristic the maximum is only 21, 81, 21, and 18, respectively, which corresponds to up to 50% reduction. The number of links of AS-1239 is extremely high (1944 links). Hence, average node degree for each node is also high. For this reason, neighboring nodes cannot be always embedded in the same physical host due to resource unavailability (CPU, memory, and bandwidth). As a result, the number of cross-host links cannot be significantly reduced. However, a point to be noted here is that our heuristic improves upon the FF approach while using less number of physical hosts.

Fig. 4.3 reports the average, minimum, and maximum percentage of physical bandwidth used by cross-host links for both FF and our proposed heuristic. We can see from this figure that for all four topologies our proposed heuristic produces embedding configurations that require much less cross-host link bandwidth than FF. For AS-1221, AS-1239, AS-1755, and AS-3967 the reduction in average amount of cross-host link bandwidth proportion is 42.3%, 8%, 38% and 40%, respectively. The maximum percentage of cross-host link over all physical hosts for AS-1221, AS-1239, AS-1755, and AS-3967 is 39, 98.8, 41.4, and 48.4, respectively for FF. However, for our proposed heuristic the maximum is only 19.5, 83.2, 18.3, and 15.6, respectively. Our proposed heuristic can achieve up to 50% reduction. For AS-1239, FF produces embedding with 36 physical nodes with a maximum bandwidth usage of 98%. However, our heuristic embeds the same topology with only 23 physical hosts and bounds the maximum bandwidth usage within 83%.

4.4 Reproducing Network Experiments on DOT

We demonstrate the functional capabilities of DOT by reproducing results from a number of networking research papers, by running the corresponding experiments on our platform. For this purpose, we have chosen to reproduce results from the following research works: (i) relation between the number of states and memory usage of Bro [15] intrusion detection system [36], (ii) network link utilization monitoring in SDN using FlowSense [59] , and (iii) throughput characteristics of a data center network topology by scheduling the flows using best fit approach in Hedera [24].

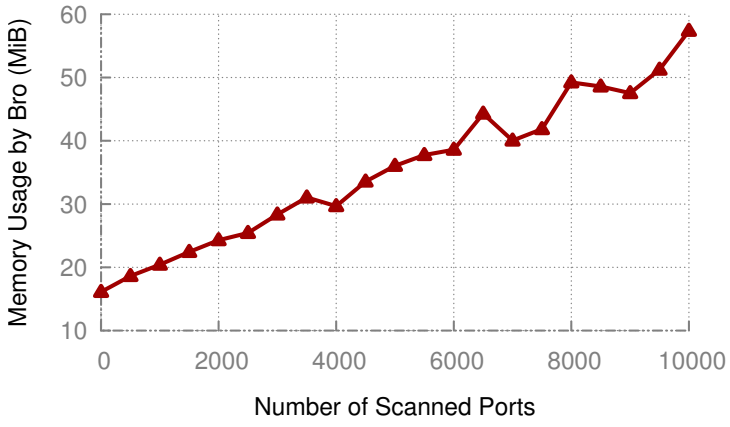


Figure 4.5: Bro IDS Memory Usage Study

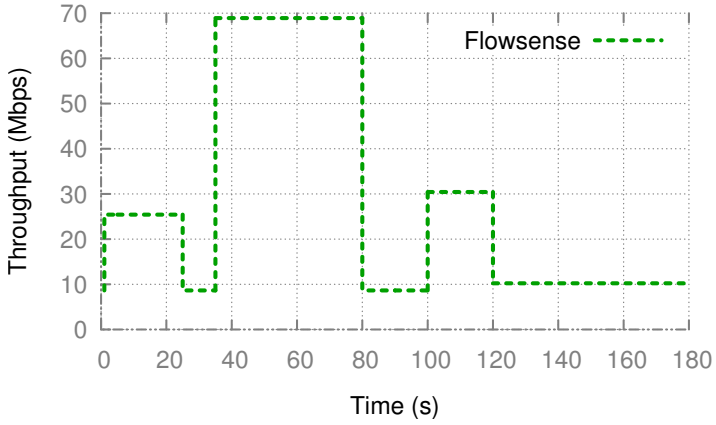


Figure 4.6: Link Usage Measurement using FlowSense

4.4.1 Performance of Bro IDS

The authors reported a linear growth of memory usage of Bro with increasing number of internal states. To reproduce the results we took the approach described in [22]. We instantiated two virtual hosts in DOT connected to a single switch. We run Bro on one of the virtual hosts, and run nmap [19], a security scanner that can scan range of ports on a destination host.

Fig. 4.4 shows results for memory usage of Bro. We changed the number of scanned ports from 1 to 10000, in increments of 500, and report the memory consumption by

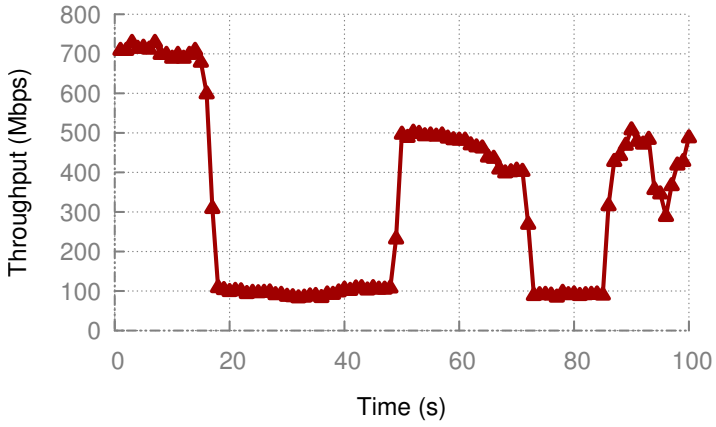


Figure 4.7: Hedera

Bro. We compute the memory consumption in the same way as [22], *i.e.*, from the minor page faults caused by Bro. Fig. 4.4 shows a linear growth in memory usage of Bro with increasing number of connections it needs to keep track of. This linear growth conforms with the original paper [36], as well as with the results reported in [22].

4.4.2 Network Monitoring using FlowSense

For the network monitoring experiment, we setup a similar scenario as described in the original paper [59]. We connected two SDN virtual switches and connected one and two hosts to the switches, respectively. Then we played a time varying traffic similar to the one described in the original paper [59]. We implemented the FlowSense algorithm as a module in Floodlight. We measured the utilization of the link between the two switches and reported the utilization against time.

Then in Fig. 4.4, we report the utilization of the link connecting the two switches. FlowSense was able to properly capture the shape of the traffic along the link and was able to reproduce the results from the original paper.

4.4.3 Evaluating Hedera

For the final experiment, we reproduce results on the impact of scheduling flows using Hedera. We implemented the global first fit flow scheduling policy from the Hedera paper

in Floodlight controller. Our experiments were run on a three level multi-rooted tree topology. The three levels correspond to the edge, aggregate, and core switches in the network. Each edge switch (8 of them) has two virtual hosts connected to it. We measured the throughput of flows by using a *Stride* pattern as described in the original paper.

For the Hedera experiment, the result reported in Fig. 4.4, does not exactly conform to the values of throughput. However, as the figure shows, the throughput follows the same shape as described in the original Hedera paper.

4.5 Summary

Experimental results show that DOT overcomes scalability issues of Mininet and guarantees the required resources for the emulated network. Moreover, the proposed embedding algorithm performs significantly better than a first fit strategy. Compared to the first fit strategy, our algorithm introduces about 50% lesser cross-host links and requires about 50% lesser bandwidth on physical links. Finally, DOT is able to replicate the results of three prominent networking research papers.

Chapter 5

Conclusion

Since its inception, SDN has made itself a very appealing networking architecture for both data center and Wide Area networks by offering more automated control through programmability and simplified network operations and management with its centralized control plane. Moreover, NFV has been receiving a lot of attention from the industry as well for the last few years. Many companies are trying to claim their share of this SDN and NFV market with their products and solutions [20]. However, extensive rollout of these new technologies in production environments with confidence requires thorough validation. Thus, a network emulator facilitating all the domain specific features is a dire necessity.

Emulating SDN was the major driving force behind development of DOT. However, full virtualization with KVM also made DOT a competent candidate for evaluating NFV solutions. We envision DOT as a comprehensive emulation platform for such newer network technologies and services. Next, we outline the contributions of this thesis (Section 5.1) and briefly describe potential research avenues that can be pursued from here (Section 5.2).

5.1 Summary of Contributions

The contributions of this thesis are listed below:

- We address the scalability issues for emulating a large scale SDN and design a distributed emulator DOT to overcome them. DOT allocates guaranteed resources to all virtual components of an emulation during provisioning. We formulate the problem of optimal allocation of physical resources to the virtual components as an ILP. Then,

we design an efficient yet simple heuristic to solve the $\mathcal{NP}$-hard embedding problem. We also design very simple but novel inter-physical machine traffic forwarding technique to ensure transparency and minimize the number of broadcast messages in the physical switching fabric.

- We develop our emulation platform with several open source tools. A user can deploy an emulation and later can interact with it from a central location in the network (*i.e.*, from the DOT Manager). DOT is available as an open source software at www.dothub.org. In addition to the source code, this portal provides installation scripts to easily deploy our platform on a cluster of machines. Moreover, it also contains a comprehensive tutorial to explain the steps to run network emulations on DOT.

- Experimental results show that DOT is able to overcome scalability issues and to guarantee the required resources for the emulated network. Moreover, we present the performance of our heuristic by showing its superiority over First Fit approach. While embedding a virtual infrastructure on DOT, our algorithm uses less physical resources. Finally, we provide our experience in reproducing results of known network experiments by running them on DOT. These results are comparable to the results reported in the original papers.

5.2 Future Works

Here, we present the following possible extensions to the thesis work:

- **Hybrid network:** In addition to Open vSwitch, DOT can be easily extended to include other virtual switch stacks (*e.g.*, Quagga [21]) which support traditional routing protocols like BGP, OSPF, *etc.* This feature will facilitate users to experiment with both OpenFlow and *non*-OpenFlow protocols on the same platform. Thus, the user can compare and contrast an SDN solution with its traditional counterpart with ease.

- **Physical switch integration:** Physical switches can be included in an emulated network in DOT. This feature enables flexible experimental setup to aid gradual rollout of SDN technology in the production network by emulating one part of the network in DOT while keeping the other parts running on actual hardware.

- **Multi-tenant Testbed:** DOT can be extended to provide access to multiple users to run simultaneously their emulations on the same physical infrastructure. This can create an opportunity of a new multi-tenant testbed with inherent support for SDN and NFV. We envision virtualization at the core of such testbed. Several standard techniques used by the cloud providers (*e.g.*, VM migration and consolidation) can be leveraged in this *emulation as a service* domain to better utilize the underlying physical infrastructure. Per user namespace isolation (*e.g.*, two users can use the same name for their VMs without noticing each other), performance and completion time guarantee for each emulation, and physical failure mitigation are some notable challenges for providing this service. It is worth mentioning that none of the existing publicly available testbeds (*e.g.*, GENI [30], OFELIA [55], Emulab [58], *etc.*) supports all of these features.

- **Time dilation:** Recall from Section 3.5, DOT embedding fails when there is no sufficient physical resources to meet the requirement of an emulation. For such situations of resource scarcity, the technique of *time dilation* has been proposed by Gupta *et al.* [41]. Specifically, time dilation slows down the time of a VM by altering time management mechanism of the hypervisor. In other words, a VM maintains a separate *virtual time* which is not synchronized with the time of its host machine. When the time is slowed down, a VM perceives more resources (particularly, CPU and bandwidth) than their actual allocation. Obvious downside of this approach is that the emulation takes longer time to finish. For an SDN emulator, maintaining virtual time only for the VMs is not sufficient. The switching fabric and the remote control plane of SDN should be dilated alongside. Incorporating time dilation in DOT is another potential extension to this thesis.

APPENDICES

Appendix A

Installation

DOT runs on a set of physical machines. One of these physical machines orchestrates the whole emulation process. We call this physical machine the DOT Manager. The other physical machines are called DOT Nodes. These machines contain virtual switches and virtual machines used to build the emulated topology. The recommended platform for DOT Nodes and the DOT Manager is Ubuntu 12.04.4 LTS. The kernel 3.5.0-54-generic.

The DOT Manager must be setup before any DOT Node. The username for all DOT Nodes must be same. We recommend using a separate physical machine for DOT Manager.

A.1 DOT Manager Installation

DOT Manager runs the DOT embedding and provisioning modules. It also provides centralized access to the deployed VMs in different DOT nodes. To setup a DOT manager, the guideline below must be followed:

- Launch a new terminal and enter into privileged mode:

  ```
  sudo -s
  ```

- Download and extract the following script:

  ```
  wget http://dothub.org/downloads/dot_scripts_1_0.tar.gz
  tar xvzf dot_scripts_1_0.tar.gz
  ```

- Change the installation script privileges, run the script and follow the on-screen instructions:

```
chmod a+x dot_manager_install.sh
./dot_manager_install.sh
```

A.2 DOT Node Installation

A DOT node utilizes Open vSwitch, libvirt and KVM to emulate a given topology. Hosts in the emulated topology are instantiated as VMs running either Tiny Core or Ubuntu. This image is preconfigured to assign IP addresses from the 10.254.0.0/16 subnet to the primary interface (i.e., eth0). The lowest 16 bits of the IP address are set equal to the lowest 16 bits of the MAC address of the primary interface to ensure uniqueness. Guidelines to setup a DOT Node are as follows:

- Launch a new terminal and enter into privileged mode:

```
sudo -s
```

- Download and extract the following script:

```
wget http://dothub.org/downloads/dot_scripts_1_0.tar.gz
tar xvzf dot_scripts_1_0.tar.gz
```

- Change the installation script privileges, run the script and follow the on-screen instructions:

```
chmod a+x dot_manager_install.sh
./dot_node_install.sh
```

The script creates a user account that is used for emulation purposes. The default username is **dot**. To use the default username just press ENTER when prompted for a username, otherwise enter a valid username of your preference. A password for the DOT user is also needed. The script will also prompt for user details (e.g., full name, address, location, etc.). One can choose the default value just by pressing ENTER at each prompt. However, one must press the Y key when asked for permission to create the DOT user

account. If the script is able to successfully install all required components, it will inform that the installation is successful and will prompt to reboot the machine.

After configuring all DOT Nodes, copy the ssh key of the DOT Manager to them by following these instructions:

```
wget http://dothub.org/downloads/dot_scripts_1_0.tar.gz
sudo -s
tar xvzf dot_scripts_1_0.tar.gz
chmod a+x dot_manager_keycopy.sh
./dot_manager_keycopy.sh
```

Appendix B

Tutorial: Emulating a topology on DOT

In this chapter, we describe the methods to emulate an SDN topology on DOT with an example. Here, we also describe the emulation configuration file and introduce DOT console and APIs to deploy, monitor, and control an emulation.

B.1 Sample Setup

B.1.1 Physical Environment

The physical environment should contain a dedicated machine for deploying the DOT Manager and one or more machines for deploying the DOT Node(s). The DOT configuration file should contain the hostname, IP address and resource (CPU, memory, disk and interface bandwidth) capacities of each machine.

For example, in Fig. B.1, three machines are connected using a physical switching fabric. Here, `rsg-pc145` is used as the DOT Manager and `rsg-pc161`, `rsg-pc146` are used as DOT Nodes.

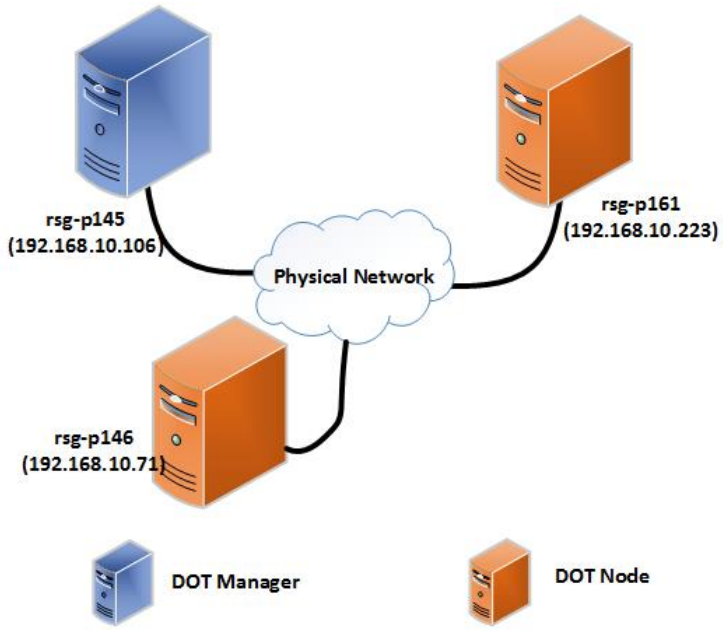


Figure B.1: Physical Environment

B.2 Deploying Logical Topology

A logical topology usually consists of virtual switches, virtual links and virtual machines. DOT Manager distributes the logical topology across the DOT Nodes by considering available physical resources. You should specify the bandwidth and delay of a virtual link connecting two virtual switches. Moreover, you have to provide the resource requirements (CPU, memory, disk and interface bandwidth) for each VM in the DOT configuration file. The Fig. B.2 shows a logical topology with three switches, two links and five virtual machines.

B.3 How to use DOT

B.3.1 Create the configuration file

An emulated topology can be specified using a configuration file. Here, we explain its format and content. We continue with the example presented above. Each section of this configuration file is explained below:

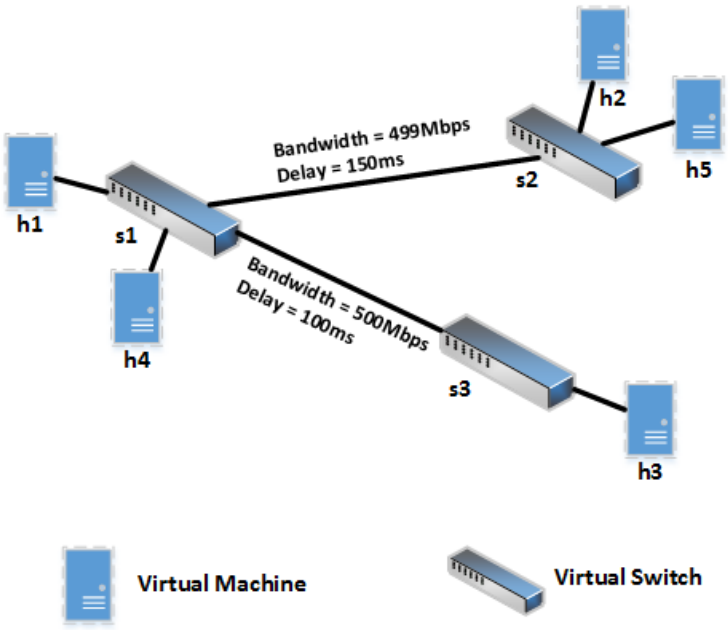


Figure B.2: Logical Topology

The first section specifies a user defined name for the emulation. Note that we follow the standard convention of putting section name in parenthesis as `[SectionName]`

```
[EmulationName]
example_wan
```

The next section specifies the physical topology. The first line in this section states the number of the physical machines, and the subsequent lines describe their properties. Each physical machine is described using a tuple: `<node type, hostname, IP address, CPU (number of cores), memory (MB), disk (GB), interface name, interface bandwidth (Mbps)>`. For the DOT Manager, the node type is manager and for the DOT Nodes, it is node. For the manager, information about CPU, memory, disk, interface, and bandwidth is not required. Modify this section according to your physical infrastructure.

```
[PhysicalTopology]
#First line: Number of physical machines
#Then, each line specifies a tuple:
#type name ip cpu memory disk external_interface bandwidth
```

```
3
manager rsg-pc145 192.168.10.106
node rsg-pc161 192.168.10.223 4 512 8 eth0 1024
node rsg-pc146 192.168.10.71 5 512 8 eth0 1024
```

The estimated propagation delay between two physical machines can be provided in the following section. The first line indicates the number of pairwise delays that are mentioned in the subsequent lines. Each line mentions the hostnames of the physical machines and the propagation delay (in `ms`) between them. If the delay for any pair is omitted, then a default value of 0 ms is considered.

```
[PhysicalTopologyDelay]
#Pairwise delay between any physical vms
#If no delay between a pair is specified, default value will be considered
#Default value of delay is 0
#First line: Number of pair of machines have delay
#Next, each line specifies a tuple
#machine1, machine2, delay (ms)
0
```

The next section presents the logical topology. The number of switches and links are specified in the first line. Each subsequent line describes the properties of a bidirectional logical link by a tuple: `<switch 1, switch 2, bandwidth (Mbps), propagation delay (ms)>`. Each switch is prefixed with letter s followed by a number (between 1 and the total number of switches).

```
[LogicalTopology]
#First line: Number of switches, number of links
#Each switch is represented by 's#'
#Switch # should start with 1
#Next, each line specifies a bidirectional link with the following format:
# s# s# bandwidth delay
3 2
s1 s2 499 150
s1 s3 500 100
```

The following section specifies the VM images used for provisioning the end hosts. The first line indicates the number of images. Each subsequent line describes the properties of a image by a tuple: `<image id, type, name>`. Each image is prefixed with letter i followed by a number (between 1 and the total number of images).

```
[Images]
#Number of images
#each line image id, type, location
2
i1 tc base_tc.qcow2
i2 ubuntu base_lubuntu_mini.qcow2
```

The following section describes the virtualization platform to be used with DOT. Currently, DOT supports KVM hypervisor with Libvirt Virtualization API. Do not change the values of the `Hypervisor` and `Library`. The `NetworkFile` key points to the network configuration file used by DOT to connect a VM to an Open vSwitch instance.

```
[VirtualMachineProvision]
#Start marker of this section. Don't remove
Hypervisor=kvm
Library=lib-virt
NetworkFile=resources/provisioning/libvirt/libvirt_network.xml
#End marker of this section. Don't remove
```

Then, the configuration for the VMs are provided. The first line specifies the number of VMs to be instantiated by DOT. Subsequent lines describe the connection between the VMs and switches. Each line also contains the vCPU (virtual CPU) requirement (number of cores), the memory (MB), the interface bandwidth (Mbps), and the image to be used for provisioning the VM. A VM is prefixed with letter h followed by a number (between 1 and the total number of hosts). The MAC and IP addresses of the VMs are automatically generated from their identifiers using a predefined rule. For example, VM with identifier `h1` is assigned the MAC address $00:00:00:00:00:01$ and IP address 10.254.0.1.

```
[VirtualMachines]
#First line: Number of VMs
#Each switch is represented by 'h#'
#VM# should start with 1
```

```
#Next, each line specifies
#a VM, the switch it is attached, CPU, B/W(Mbps), Memory(MB) and image:
# VM# s# CPU B/W Memory i#
5
h1 s1 1 300 64 i1
h2 s2 2 400 512 i2
h3 s3 1 300 64 i1
h4 s1 2 400 64 i1
h5 s2 1 300 512 i2
```

The number of SDN controllers and their IP addresses and port numbers are provided in the following section. Each controller is prefixed with c and a number (between 1 and the total number of controllers).

You can emulate both *in-band* and *out-of-band* control plane using DOT. The *out-of-band* SDN controller should be reachable from all DOT Nodes.

In addition, both POX and Floodlight controllers are pre-loaded in the supplied Ubuntu image with DOT. Thus, any VM using this image can run an SDN controller. In this configuration, `c1` is an in-band controller hosted by `h2`. This is specified by using the IP address 10.254.0.2. Remember that DOT assigns IP and MAC addresses based on the name of the VM, which in this case is `h2`. The in-band controller will not be initialized automatically and needs to be started separately once the emulated network is deployed (explained in the subsequent section).

```
[Controllers]
#First line: Number of controllers
#Each controller is represented by 'c#'
#Controller # should start with 1
#Next, each line a tuple:
# c# ip port
2
c1 10.254.0.2 6633
c2 192.168.10.106 6633
```

Switch to controller association is specified in the next section. The first line mentions the number of switches and each subsequent line indicates the controller of a switch.

```
[Switch2Controller]
#First line: Number of switches
#Next, each line specifies a switch and its controller
# s# c#
3
s1 c2
s2 c2
s3 c2
```

The next section specifies the credentials (username) to access the DOT nodes from the DOT manager.

```
[Credentials]
#User name of the dot nodes
UserName=dotuser
```

In the final section, OFVersion parameter is used to configure the OpenFlow protocol version of the virtual switches. Currently, DOT supports OpenFlow version 1.0 and 1.3. Thus, OFVersion parameter should be set either to 1.0 or 1.3.

```
[OtherConfig]
#
OFVersion=1.0
```

B.3.2 Run DOT Manager

The DOT manager deploys the emulated topology on the physical machines. It deploys the logical topology only if there is a feasible embedding satisfying resource and delay constraints. Otherwise, it exits by showing that the embedding is not possible. To deploy the logical topology, go to the root directory of the DOT installation in the DOT Manager and run the following commands:

```
root@rsg-pc145:~/dot#./run.sh ExampleWANConf1.0.txt
```

Turn the Dubug mode on:

```
root@rsg-pc145:~/dot#./run.sh ExampleWANConf1.0.txt -d
```

B.3.3 Introduction to DOT Console

DOT provides centralized access to the emulated topology from the DOT manager. You can run the following commands (in privileged mode) to start a utility console.

```
root@rsg-pc145:~/dot# cd ongoing_emulation
root@rsg-pc145:~/dot/ongoing_emulation# ./dot_console
```

Now you are at the DOT manager utility console. To get the list of available commands type help:

```
dot>help
For connecting to VM: connect h#
For disconnecting a VM: disconnect h#
For detaching a link: detach s# s#
For reattaching a link: attach s# s#
Ping: ping h# h#
For sending ping between all pair of hosts: PingAll
For running command in a host: run [-b] [-s] h# command/scriptfile
For monitoring a switch: monitor [-s] start/stop s#
For monitoring a controller: monitor [-c] start/stop c#
For cleaning DOT: clean all
To exit: quit
```

Connecting to a VM:

```
dot>connect h2
Connecting h2
```

A console window for the VM will pop up. You can test the connectivity using ping. Press ctrl+alt to release the mouse pointer from the VM terminal window and change focus to the DOT manager.

Disconnecting from a VM:

```
dot>disconnect h2
Disconnecting h2
```

Purging all VMs and Switches:

```
dot>clean all
Cleaning 192.168.10.71 ....
.......
Quitting DOT console
```

To exit the DOT console:

```
dot>quit
Do you want to disconnect all VMs? [Y/n]
Disconnecting h2
Exiting....
```

Entering `n` to the highlighted prompt (see figure below) will keep the connections to the VMs intact. Otherwise, all connections will be teared down. Note that exiting the DOT console does not shut down the VMs.

```
dot>quit
Do you want to disconnect all VMs? [Y/n]
Exiting....
```

B.3.4 Configuring Management interface of a VM

Each VM is equipped with an interface (eth1) attached to the management network. This interface facilitates remote access to the VM directly from the DOT Manager. All DOT console commands except connect, disconnect, attach, detach assume that the management interface is operational.

If there is a functional DHCP server in the physical network, then the management interfaces can get their required configurations from it. If there is no DHCP server in the network, follow these instructions to configure $eth1$ for each VM:

```
dot>connect h2
```

Then, in the console window of each VM, manually assign IP Address to $eth1$. For example, to assign IP Address 192.168.10.3 to $eth1$ of `h2`:

```
ubuntu@h2:~$sudo ifconfig eth1 192.168.10.3 netmask 255.255.255.0
```

Note the IP Address assigned to the management interface should be in the same subnet as that of the DOT Manager. To check whether the management interface is operational, run pingAll command in the DOT Console:

```
dot>pingAll
Pinging h1->h2
Warning: Permanently added 'h1,192.168.10.2' (RSA) to the list of known hosts.
PING h2 (10.254.0.2): 56 data bytes
64 bytes from 10.254.0.2: seq=0 ttl=64 time=307.990 ms
64 bytes from 10.254.0.2: seq=1 ttl=64 time=151.388 ms
64 bytes from 10.254.0.2: seq=2 ttl=64 time=151.460 ms
64 bytes from 10.254.0.2: seq=3 ttl=64 time=151.416 ms

--- h2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 151.388/190.563/307.990 ms

Pinging h1->h3
Warning: Permanently added 'h1,192.168.10.2' (RSA) to the list of known hosts.
PING h3 (10.254.0.3): 56 data bytes
64 bytes from 10.254.0.3: seq=0 ttl=64 time=202.410 ms
64 bytes from 10.254.0.3: seq=1 ttl=64 time=100.570 ms
64 bytes from 10.254.0.3: seq=2 ttl=64 time=100.561 ms
64 bytes from 10.254.0.3: seq=3 ttl=64 time=100.675 ms
```

It is recommended to run a DHCP server (e.g., dnsmasq) in the physical network to avoid this manual configuration after each provisioning of the emulated network. Note the primary interface of a VM (i.e., eth0), which is attached to the data network, is pre-configured by DOT during provisioning. So, this interface does not require any manual configuration.

B.3.5 Running SDN Controller

The configuration file explained before assumes that the SDN controller is deployed at the DOT Manager (Check Controllers and `Switch2Controller` sections of the configuration

file). However, even in the absence of a SDN controller, the VMs should be reachable from each other, i.e., the `pingAll` command should be successful. Any SDN controller can be used with DOT. Next, we will explain how to use two opensource SDN controllers: POX and Floodlight with DOT.

To use POX with DOT, follow these instructions to install it in the DOT Manager. Next, navigate to the POX root directory and run the following command:

```
./pox.py forwarding.l2_learning
```

On the other hand, to use Floodlight with DOT, use these instructions to install it in the DOT Manager. Then, go to the floodlight root directory and run the following:

```
java -jar target/floodlight.jar
```

Now, run `pingAll` command in the DOT Console to verify the packet forwarding using SDN controller:

```
dot>pingAll
Pinging h1->h2
Warning: Permanently added 'h1,192.168.10.2' (RSA) to the list of known hosts.
PING h2 (10.254.0.2): 56 data bytes
64 bytes from 10.254.0.2: seq=0 ttl=64 time=307.990 ms
64 bytes from 10.254.0.2: seq=1 ttl=64 time=151.388 ms
64 bytes from 10.254.0.2: seq=2 ttl=64 time=151.460 ms
64 bytes from 10.254.0.2: seq=3 ttl=64 time=151.416 ms

--- h2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 151.388/190.563/307.990 ms

Pinging h1->h3
Warning: Permanently added 'h1,192.168.10.2' (RSA) to the list of known hosts.
PING h3 (10.254.0.3): 56 data bytes
64 bytes from 10.254.0.3: seq=0 ttl=64 time=202.410 ms
64 bytes from 10.254.0.3: seq=1 ttl=64 time=100.570 ms
64 bytes from 10.254.0.3: seq=2 ttl=64 time=100.561 ms
64 bytes from 10.254.0.3: seq=3 ttl=64 time=100.675 ms
```

B.3.6 Logging

DOT is capable of logging the OpenFlow messages exchanged between the SDN controller and the switches. By using `start` commands, you can start monitoring specific switch or controller.

To start switch (*e.g.*, `s1`) monitoring:

```
monitor -s start s1
```

To start controller (*e.g.*, `c1`) monitoring:

```
monitor -c start c1
```

Later, you can stop all monitoring using the `stop` commands. These log messages will be stored in the `log` directory of the DOT Manager.

To stop switch (*e.g.*, `s1`) monitoring:

```
monitor -s stop s1
```

To stop controller (*e.g.*, `c1`) monitoring:

```
monitor -c stop c1
```

Additionally, you can run `tcpdump` to capture packets in the end hosts. For example, to log all packets at `eth0` of `h1`, you can run the following command in the DOT Console:

```
run -b h1 'sudo tcpdump -i eth0 > dump001'
```

In this example, the log messages will be stored at the file name `dump001`.

B.3.7 DOT APIs

DOT provides a number of Python APIs. After deploying a virtual infrastructure, a user can write a script to emulate different network behaviors. The following code snippet shows how to simulate link failure using DOT APIs:

```
from ongoing_emulation.api.dot import Dot
emu = Dot()
emu.load()
result = emu.ping('h1', 'h2')
print result
emu.detach('s1', 's2')
result = emu.ping('h1', 'h2')
print result
emu.attach('s1', 's2')
result = emu.ping('h1', 'h2')
print result
```

B.3.8 Running a control application: UDP Blocker

Here, we will show how to run a simple firewall in DOT using Floodlight controller. To enable the firewall run the following command:

```
root@rsg-pc145:~# curl http://localhost:8080/wm/firewall/module/enable/json
```

The default behavior of this firewall module is to discard every packet which can be observed from running ping command in the DOT console:

```
dot>ping h2 h1
pinging h1
Warning: Permanently added 'h2,192.168.10.3' (ECDSA) to the list of known hosts.
PING h1 (10.254.0.1) 56(84) bytes of data.
From h2 (10.254.0.2) icmp_seq=10 Destination Host Unreachable
From h2 (10.254.0.2) icmp_seq=11 Destination Host Unreachable
```

To allow all packets through s1 and s2 use the following commands:

```
root@rsg-pc145:~# curl -X POST -d '{"switchid": "1"}'
                  http://localhost:8080/wm/firewall/rules/json
root@rsg-pc145:~# curl -X POST -d '{"switchid": "2"}'
                  http://localhost:8080/wm/firewall/rules/json
```

Now, `h1` becomes reachable from `h2`

```
dot>ping h2 h1
pinging h1
Warning: Permanently added 'h2,192.168.10.3' (ECDSA) to the list of known hosts.
PING h1 (10.254.0.1) 56(84) bytes of data.
From h2 (10.254.0.2) icmp_seq=1 Destination Host Unreachable
From h2 (10.254.0.2) icmp_seq=2 Destination Host Unreachable
From h2 (10.254.0.2) icmp_seq=3 Destination Host Unreachable
From h2 (10.254.0.2) icmp_seq=4 Destination Host Unreachable
From h2 (10.254.0.2) icmp_seq=5 Destination Host Unreachable
From h2 (10.254.0.2) icmp_seq=6 Destination Host Unreachable
64 bytes from h1 (10.254.0.1): icmp_req=7 ttl=64 time=1331 ms
64 bytes from h1 (10.254.0.1): icmp_req=8 ttl=64 time=331 ms
```

Next, run an `iperf` UDP server at `h2`. This command is run in background (using -b flag) so that subsequent commands can be issued in the console.

```
dot>run -b h2 iperf -s -u
Warning: Permanently added 'h2,192.168.10.3' (ECDSA) to the list of known hosts.
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
```

Then, run an `iperf` UDP client at h1. This client sends UDP datagram to the iperf server running at `h2`.

```
dot>run h1 iperf -c h2 -u -t 10 -i 1
Warning: Permanently added 'h1,192.168.10.2' (RSA) to the list of known hosts.
------------------------------------------------------------
Client connecting to h2, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   110 KByte (default)
------------------------------------------------------------
[  3] local 10.254.0.1 port 53980 connected with 10.254.0.2 port 5001
[  3] local 10.254.0.2 port 5001 connected with 10.254.0.1 port 53980
[  3]  0.0- 1.0 sec    128 KBytes  1.05 Mbits/sec
```

```
[  3]  1.0- 2.0 sec    129 KBytes  1.06 Mbits/sec
[  3]  2.0- 3.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  3.0- 4.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  4.0- 5.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  5.0- 6.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  6.0- 7.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  7.0- 8.0 sec    129 KBytes  1.06 Mbits/sec
[  3]  8.0- 9.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  9.0-10.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.040 ms    0/  893 (0%)
[  3] WARNING: did not receive ack of last datagram after 10 tries.
```

As the firewall does not block any traffic, the `iperf` client is able to communicate with the `iperf` server.

Now, apply the following firewall rules in the DOT Manager to block UDP traffic between `h1` and `h2`. Note that IP Addresses of `h1` and `h2` are 10.254.0.1 and 10.254.0.2, respectively.

```
root@rsg-pc145:~# curl -X POST -d '{"src-ip": "10.254.0.1/32",
                  "dst-ip": "10.254.0.2/32", "nw-proto":"UDP", "action":"DENY" }'
                  http://localhost:8080/wm/firewall/rules/json
root@rsg-pc145:~# curl -X POST -d '{"src-ip": "10.254.0.2/32",
                  "dst-ip": "10.254.0.1/32", "nw-proto":"UDP", "action":"DENY" }'
                  http://localhost:8080/wm/firewall/rules/json
```

Finally, again run the `iperf` client at `h1`.

```
dot>run h1 iperf -c h2 -u -t 10 -i 1
Warning: Permanently added 'h1,192.168.10.2' (RSA) to the list of known hosts.
------------------------------------------------------------
Client connecting to h2, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:   110 KByte (default)
```

```
------------------------------------------------------------
[  3] local 10.254.0.1 port 45632 connected with 10.254.0.2 port 5001
[  3]  0.0- 1.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  1.0- 2.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  2.0- 3.0 sec    129 KBytes  1.06 Mbits/sec
[  3]  3.0- 4.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  4.0- 5.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  5.0- 6.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  6.0- 7.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  7.0- 8.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  8.0- 9.0 sec    128 KBytes  1.05 Mbits/sec
[  3]  9.0-10.0 sec    129 KBytes  1.06 Mbits/sec
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 893 datagrams
[  3] WARNING: did not receive ack of last datagram after 10 tries.
```

Here, the output does not contain any report from the `iperf` server. This confirms that the UDP traffic between `h1` and `h2` is blocked.

To disable the firewall:

```
root@rsg-pc145:~# curl http://localhost:8080/wm/firewall/module/disable/json
```

References

[1] http://opennetsummit.org/archives/mar14/site/why-sdn.html.

[2] http://www.vmware.com/ca/en/products/nsx/.

[3] http://www.cisco.com/c/en/us/products/routers/wan-automation-engine/index.html.

[4] http://viptela.com/.

[5] http://www.cloudgenix.com/.

[6] http://www.gluenetworks.com/.

[7] http://www.businesscloudnews.com/2014/09/10/huawei-21vianet-build-largest-commercial-sdn-network/.

[8] http://floodlight.openflowhub.org.

[9] http://www.opendaylight.org/.

[10] https://github.com/noxrepo/pox.

[11] http://osrg.github.io/ryu/.

[12] http://www.linux-kvm.org/.

[13] http://openvswitch.org.

[14] http://linuxcontainers.org/.

[15] http://www.bro.org/.

[16] http://www.pfsense.org/.

[17] http://libvirt.org.

[18] http://distro.ibiblio.org/tinycorelinux.

[19] http://nmap.org/.

[20] http://www.sdxcentral.com/nfv-sdn-companies-directory/.

[21] Quagga routing suite. http://www.nongnu.org/quagga/.

[22] Reproducable network research. https://reproducingnetworkresearch.wordpress.com /2014/06/03/cs-244-14-bro-network-intrusion-detection-system-performance-analysis/.

[23] OpenFlow switch specification 1.3.1. Open Networking Foundation, September 2012.

[24] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[25] Ahmed Amokrane, Mohamed Faten Zhani, Qi Zhang, Rami Langar, Raouf Boutaba, and Guy Pujolle. On satisfying green slas in distributed clouds. In *IEEE/ACM/IFIP International Conference on Network and Service Management (CNSM)*, Rio de Janeiro, Brazil, November 2014.

[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[27] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. Policycop: An autonomic qos policy enforcement framework for software defined networks. In *IEEE Software Defined Network for Future Network and Services 2013*, November 2013.

[28] Md. Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani Zhani, Reaz Ahmed, and Raouf Boutaba. Dynamic controller provisioning in software defined networks. In *IEEE/ACM/IFIP International Conference on Network and Service Management (CNSM)*, Zurich, Switzerland, October 2013.

[29] Md. Faizul Bari, Mohamed Faten Zhani, Qi Zhang, Reaz Ahmed, and Raouf Boutaba. CQNCR: Optimal VM migration planning in cloud data centers. In *IFIP Networking*, Trondheim, Norway, 2014.

[30] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds – Part I.

[31] Chandra Chekuri and Sanjeev Khanna. On multi-dimensional packing problems. In *Annual ACM-SIAM symposium on Discrete algorithms*, SODA, 1999.

[32] Margaret Chiosi and Don Clarke et al. Network functions virtualisation introductory white paper. Darmstadt, October 2012.

[33] Mosharaf Chowdhury, Muntasir Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In *INFOCOM 2009, IEEE*, pages 783 –791. IEEE Press, 2009.

[34] Mosharaf Chowdhury, Fady Samuel, and Raouf Boutaba. Polyvine: policy-based virtual network embedding across multiple domains. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, VISA '10, pages 49–56, New York, NY, USA, 2010. ACM.

[35] Shihabur Rahman Chowdhury, Md. Faizul Bari, Reaz Ahmed, and Raouf Boutaba. Payless: A low cost network monitoring framework for software defined networks.

[36] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 2–11. ACM, 2004.

[37] Rafael Esteves, Lisandro Zambenedetti Granville, Mohamed Faten Zhani, and Raouf Boutaba. Evaluating allocation paradigms for multi-objective adaptive provisioning in virtualized networks. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.

[38] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. On Orchestrating Virtual Network Functions in NFV. *ArXiv e-prints*, March 2015.

[39] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2:

A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[40] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.

[41] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.

[42] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *ACM CoNEXT 2012*.

[43] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, 2013.

[44] David S Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

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

[46] Maurizio Pizzonia and Massimo Rimondini. Netkit: network emulation for education. *Software: Practice and Experience*, pages n/a–n/a, 2014.

[47] Md Golam Rabbani, Rafael Esteves, Maxim Podlesny, Gwendal Simon, Lisandro Zambenedetti Granville, and Raouf Boutaba. On tackling virtual data center embedding problem. In *Proceedings of the 13th IFIP/IEEE Integrated Network Management Symposium (IM 2013)*, Ghent, Belgium, may 2013.

[48] Md Golam Rabbani, Mohamed Faten Zhani, and Raouf Boutaba. On achieving high survivability in virtualized data centers. *IEICE Transactions on Communications*, 97(1):10–18, 2014.

[49] Muntasir Rahman, Issam Aib, and Raouf Boutaba. Survivable virtual network embedding. In Mark Crovella, Laura Feeney, Dan Rubenstein, and S. Raghavan, editors, *NETWORKING 2010*, volume 6091 of *Lecture Notes in Computer Science*, pages 40–52. Springer Berlin / Heidelberg, 2010.

[50] Arjun Roy, Kenneth Yocum, and Alex C Snoeren. Challenges in the emulation of large scale software defined networks. In *Proceedings of APSYS*, 2013.

[51] Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Design and Management of DOT: A Distributed OpenFlow Testbed. In *14th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, May 2014.

[52] Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Dot: Distributed openflow testbed. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 367–368, New York, NY, USA, 2014. ACM.

[53] Fady Samuel, Mosharaf Chowdhury, and Raouf Boutaba. Polyvine: policy-based virtual network embedding across multiple domains. *Journal of Internet Services and Applications*, 4(1):6, 2013.

[54] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with rocketfuel. In *ACM SIGCOMM*, 2002.

[55] Marc Suñé, Leonardo Bergesio, Hagen Woesner, Tom Rothe, Andreas Köpsel, Didier Colle, Bart Puype, Dimitra Simeonidou, Reza Nejabati, Mayur Channegowda, et al. Design and implementation of the ofelia fp7 facility: the european openflow testbed. *Computer Networks*, 61:132–150, 2014.

[56] Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. Estinet openflow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, 2013.

[57] Philip Wette, Martin Draxler, and Arne Schwabe. Maxinet: distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[58] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[59] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *Passive and Active Measurement*, pages 31–41. Springer, 2013.

[60] Qi Zhang, Mohamed Faten Zhani, Meyssa Jabri, and Raouf Boutaba. Venice: Reliable virtual data center embedding in clouds. *IEEE International Conference on Computer Communications (INFOCOM)*, 2014.

[61] Qi Zhang, Mohamed Faten Zhani, Yuke Yann, Raouf Boutaba, and Bernard Wong. Prism: Fine-grained resource-aware scheduling for mapreduce. *Cloud Computing, IEEE Transactions on*, PP(99):1–1, 2015.

[62] Mohamed Faten Zhani, Qi Zhang, Gwendal Simon, and Raouf Boutaba. Dynamic migration-aware virtual data center embedding for clouds. In *Proceedings of the 13th IFIP/IEEE Integrated Network Management Symposium (IM 2013)*, Ghent, Belgium, may 2013.