Design and Management of DOT: A Distributed OpenFlow Testbed

Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba
David R. Cheriton School of Computer Science, University of Waterloo
Email: {ar3roy, mfbari, mfzhani, r5ahmed, rboutaba}@uwaterloo.ca

***Abstract*—With the growing adoption of Software Defined Networking (SDN), there is a compelling need for SDN emulators that facilitate experimenting with new SDN-based technologies. Unfortunately, Mininet [1], the de facto standard emulator for software defined networks, fails to scale with network size and traffic volume. The aim of this paper is to fill the void in this space by presenting a low cost and scalable network emulator called Distributed OpenFlow Testbed (DOT). It can emulate large SDN deployments by distributing the workload over a cluster of compute nodes. Through extensive experiments, we show that DOT can overcome the limitations of Mininet and emulate larger networks. We also demonstrate the effectiveness of DOT on four Rocketfuel topologies. DOT is available for public use and community-driven development at *dothub.org*.**

I. Introduction

Software Defined Networking (SDN) has gained a lot of attention from industry and academia in the last few years. To simplify the network operation and management, SDN separates the control plane from the forwarding devices and moves it to a conceptually centralized controller [2].. The centralized control of SDN introduces immense scope of innovation in the way networks are programmed and managed. By maintaining a global view of the network, the controller allows to easily implement and deploy innumerable control applications (*e.g.*, routing, firewalls, traffic shaping).

With the growing adoption of SDN solutions, there is a compelling need for SDN emulators that facilitate experimenting with new SDN-based technologies. In this context, Mininet [1] has been proposed as a software emulator for prototyping a network on a single machine. It allows users to create, control and customize an emulated network on which then can run and test new control applications like SDN-based routing and traffic engineering schemes. Unfortunately, Mininet emulates the entire network on a single machine, and thus fails to scale for large emulated networks and traffic volumes as we shall show in the next section.

To address these limitations, in this paper we propose *Distributed OpenFlow Testbed* (DOT), a highly scalable emulator for SDN. DOT provisions the emulated network across a cluster of machines. Unlike Mininet, DOT provides guaranteed compute and network resources for the emulated components (*i.e.*, switches, hosts and links). By distributing the emulated network components across multiple machines, DOT can easily scale with network size and traffic volume, allowing to emulate large datacenters and wide-area networks. Researchers can use DOT for testing and evaluating new control applications, SDN-based policy enforcement platforms and monitoring frameworks [3]–[5].

978-1-4799-0913-1/14/$31.00

Our main contributions in this paper are summarized as follows:

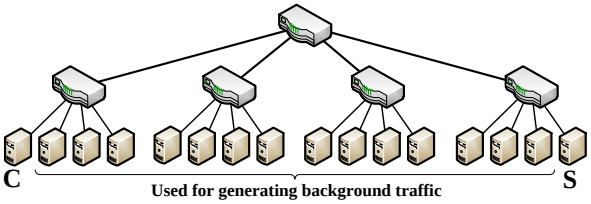
- DOT management architecture: we propose a distributed management architecture that manages the deployment of an emulated network over a cluster of machines. We also propose technical solutions to emulate network components (*i.e.*, virtual switches, virtual machines and virtual links) and to guarantee the requested performance.
- Emulated network embedding: we address the problem of optimizing the embedding of the emulated network into a physical infrastructure. We formulate the problem as an Integer Linear Program (ILP). We then propose a heuristic algorithm that minimizes both the number of virtual links crossing the network and the required number of physical machines.
- Experimental evaluation: we compare the performance obtained with DOT to that of Mininet. We also evaluate the performance of the proposed embedding algorithm and compare it with the First Fit (FF) algorithm.

The rest of this paper is organized as follows. In Section II, we present the background and motivation behind DOT. Details on DOT architecture are presented in Section III. The problem formulation of emulated network embedding problem as well as the proposed heuristic algorithm are then provided in Section IV. Evaluation results are described in Section V. Finally, we conclude in Section VI.

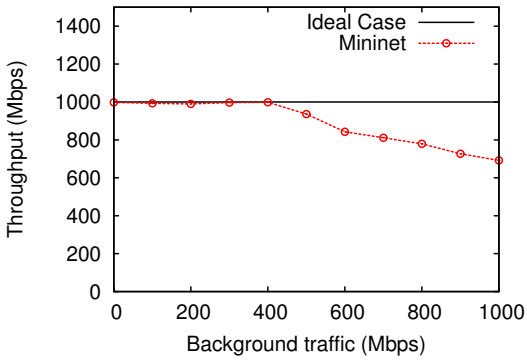
II. Background and Motivation

Mininet [1] is the state-of-the-art OpenFlow based network emulator. It emulates an OpenFlow network in a single server by running all virtual nodes (virtual hosts and switches) in separate network namespaces. Each node runs as a system process, which consumes small system resources. This allows Mininet to emulate a large network. However, the amount of traffic Mininet can simulate 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 simulate 2.2Gbps traffic, whereas a dual-core 2.1GHz Intel i5 processor (8-GB RAM) laptop can simulate up to 976Mbps traffic only.

We simulated a fat-tree topology (generated by using the Mininet command `mn --topo tree,depth=2,`



(a) Network topology



(b) Impact of background traffic

Fig. 1. Limitation of Mininet

`fanout=4`) as shown in Fig. 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(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 Mininet. Another point to be noted 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 problems of its own. More specifically, such an approach can show poor 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. These limitations have also been identified and explained in detail by Arjun Roy *et al.* in [6]. Apart from emulators, there are also large scale dedicated OpenFlow testbeds, *e.g.*, GENI [7] and OFELIA [8]. The difference between DOT and these testbeds is the basic distinction between a testbed and an emulator. A user can deploy a large number of OpenFlow switches with arbitrary topologies in DOT, which is not possible within a testbed, where users are bounded by the physical span and hardware capacities of the testbed.

In DOT, we provide guaranteed bandwidth between switches and hosts of an emulated network. Moreover, we can distribute the load over multiple machines to scale dynamically with network size and traffic volume. We have used various software libraries and components for host, switch and link virtualization. Next we explain some of these libraries and components.

Open Virtual Switch (OVS) [9], [10] is a production quality, multi-layer, virtual switch that supports numerous management protocols and interfaces along with OpenFlow. We use OVS to emulate switches. Hosts are emulated by deploying user supplied VMs. We use the `add ip link` Linux command to emulate a link having both endpoints (*i.e.*, switch or host) within the same physical machine. On the other hand, we use Generic Routing Encapsulation (GRE) [11] tunnel if the endpoints of a link reside in different physical machines.

We use the Linux `tc` command to configure traffic characteristics in Linux kernel. The traffic `SHAPING` argument of the `tc` command is used to cap transmission rate between virtual nodes and thereby simulate a specific link bandwidth. Furthermore, we use the `netem` kernel component in conjunction with the `tc` command to simulate propagation delay in a physical link.

In the last few years, a good number of OpenFlow controllers have been developed. There exists single core [12], single threaded [13] controller implementations for research purposes. While multi-core, multi-threaded [14], [15] high yield controller implementations are available for industrial usage. Controllers have also been developed with a wide variety of programming languages like python, Java and C/C++. DOT does not impose any requirements on the type of controller. So, a user can deploy any OpenFlow controller. DOT provides a user with general purpose VMs where he can run any program supported by the VM's operating system.

III. DOT: Distributed OpenFlow Testbed

In this section, we provide an overview of the proposed Distributed OpenFlow Testbed (DOT). Virtual switches and VMs are distributed over a physical network. Hence, we developed a distributed management framework for instantiating, configuring and monitoring the emulated components. We describe this framework in Section III-A. Then we explain the software stack at each physical machine in Section III-B. Finally, we explain the technical approaches for resource provision and configuration in DOT (Section III-C).

A. DOT Management Architecture

The DOT management architecture (Fig. 2) consists of two types of components, namely the central DOT manager and the DOT node manager (located at each physical machine). 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 concerned nodes. The Statistics

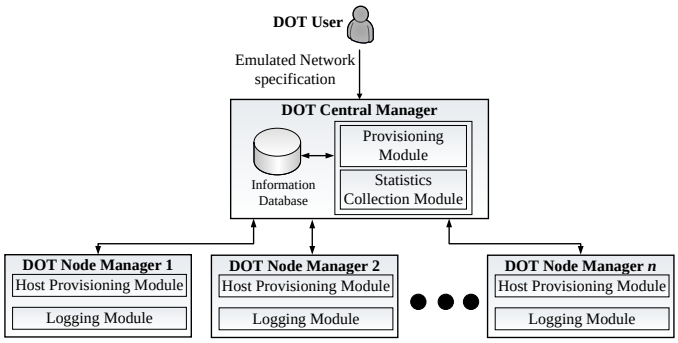


Fig. 2. DOT management architecture

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 virtual switches, links, 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.

B. DOT Software Stack

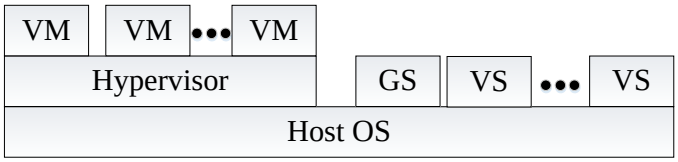


Fig. 3. Software stack of a DOT node

A DOT node (*i.e.*, a physical machine) contains virtual switches and virtual machines that are responsible for emulating an OpenFlow network. As shown in Fig. 3, 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 facilitates provisioning multiple VMs in a single physical machine. It also allows to connect VMs to virtual switches using virtual interfaces.
- **Virtual Machine (VM):** Virtual machines are put under users' control. A user can 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 filled by the user (or possibly by a controller deployed by the user).
- **Gateway Switch (GS):** Gateway Switch is a special switch created in each physical machine. Its role is to forward packets between virtual switches located at different physical machines. It is worth noting that the gateway switches are completely transparent to a DOT user.

C. DOT Resource Provisioning and Configuration

In the following, we provide more details on the configuration, interconnection and provisioning of the emulated network components. Specifically, we focus on the method of allocating virtual switches and VMs across the available machines.

A virtual switch and the VMs connected to that switch are collocated at the same physical machine. Two cases may arise while mapping a virtual link between two virtual switches. The first case arises when a virtual link connects a pair of virtual switches residing on the same physical machine. Consequently, it is totally provisioned inside one physical machine, and hence we call it an *intra-host virtual link*. The second case appears when the virtual link connects two virtual switches located at different hosts (hereafter called a *cross-host virtual link*). As a result, it should be mapped onto a path that passes through the network and connects the two ends.

Fig. 4(a) shows an example of an emulated network consisting of 8 virtual switches, 11 virtual links, 2 VMs, and 1 SDN controller. This topology is embedded into 3 physical hosts (Fig. 4(b)). Specifically, virtual switches A, B, and C are allocated in machine 1; F, G, H in machine 2; and D, E in machine 3. Virtual links (A,B) and (B,C) are examples of intra-host virtual links, whereas links (B,F) and (B,D) are cross-host virtual links.

In what follows, we describe how the different types of virtual links are provisioned and emulated.

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

- Cross-host virtual link. Provisioning this type of link requires creating multiple segments forming a path between the two end points of the link. However, since traffic is sent outside the physical machines, we create a particular switch called the Gateway Switch (GS) in each physical node for forwarding inward and outward traffic. It is worth noting that this particular switch is completely transparent to the DOT users. The emulated network controller(s) are only aware of the logical topology. The partitioning scheme and GSs are completely transparent to the controllers. The process of embedding a cross-host virtual link goes through three steps that are explained below:

1) We create virtual ip links (with the Linux command `add ip link`) to attach each virtual switch with the gateway switch. In Fig. 4, for the cross-host link d, a segment d' is created between virtual switch B and $GS1$ in physical host $PH1$ and another segment d'' is created between virtual switch F and $GS2$ in physical host $PH2$.
2) Next, a GRE tunnel is created between the physical hosts. A unique identifier is assigned to each cross-host

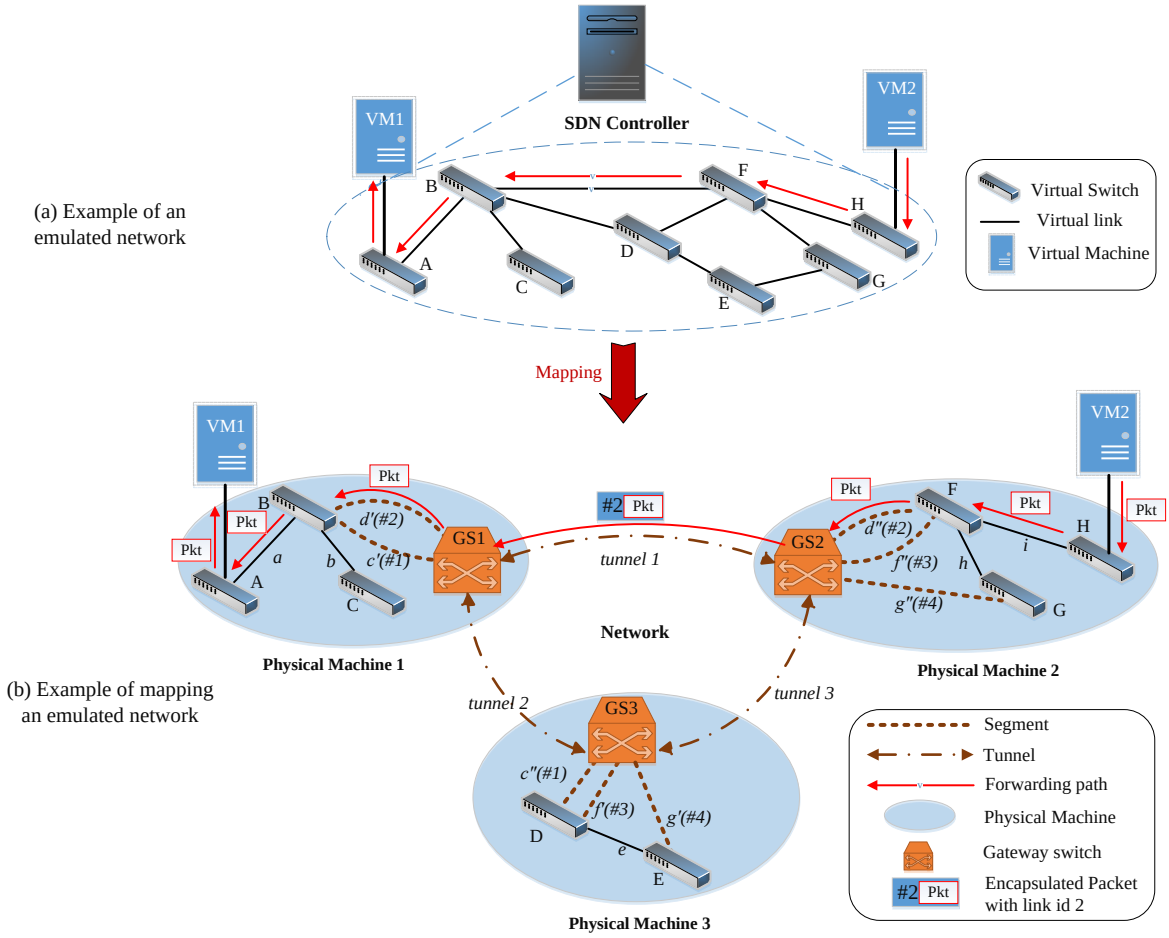


Fig. 4. Emulated network embedding and traffic forwarding in DOT

virtual link. In the figure, for cross-host virtual link d the identifier 2 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 d together.

For example, if a packet is sent from VM_2 to VM_1 through the path $H \rightarrow F \rightarrow B \rightarrow A$, then switch H first receives the packet and forwards it to F. Now, F forwards the packet through port p thinking that it will go to switch B. Here, we have connected the cross-host-segment d'' at exactly the same port p. So, the packet goes through d'' and reaches $GS2$. Now, $GS2$ inserts the cross-host virtual link identifier for d (here 2) in the `key` field of the GRE header and forwards the packet through *tunnel 1*. Next, $GS1$ receives this packet from tunnel 1, looks at the `key` field in the GRE header and based on the value forwards the packet through the link d' towards switch B. Switch B perceives this packet to be received from switch F directly. The tunneling scheme is completely hidden from the virtual switches. Next, B forwards the packet to A and then A delivers it to VM_1.

IV. Network Embedding

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

A. Problem Formulation

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 [16]) and that there is a single path between each pair of nodes. This simplifies the virtual link embedding process, i.e., in order to check whether it is possible to embed a virtual link into 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, we don't need to check the available bandwidth at the upper

layers of the data center network topology (since the network has full bisection bandwidth).

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 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 has 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 noticed that, among those 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. Hence, the required resources can be expressed as:

$$c_i^r = \sum_{e\in E} z_i^e b_e \rho_r \tag{1}$$

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. 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 written as:

$$\hat{c}_i^r = c_i^r + \sum_{h\in H}\sum_{i\in N} v_i^h g_h^r \tag{2}$$

- 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:

$$f_p^r = \sum_{i\in N}\sum_{j\in N}\sum_{q\in\tilde{N}}\sum_{e\in E} x_{ip}(1-x_{jp})x_{jq}z_i^e z_j^e b_e \rho_r \tag{3}$$

- 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:

$$\mathcal{C}^T = \sum_{i\in N}\sum_{j\in N}\sum_{p\in\tilde{N}}\sum_{q\in\tilde{N}}\sum_{e\in E} x_{ip}(1-x_{jp})x_{jq}z_{ei}z_{ej} \tag{4}$$

- Number of used physical nodes

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

$$\mathcal{C}^E = \sum_{p\in\tilde{N}} y_p \tag{5}$$

Minimizing this number is important for different reasons. First, this allows to reduce resource fragmentation, and thereby make 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. 4 and 5). It can be written as follows:

$$\mathcal{C} = \alpha\mathcal{C}^T + \beta\mathcal{C}^E \tag{6}$$

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

$$\sum_{p\in\tilde{N}} x_{ip} = 1,\ \forall i \in N \tag{7}$$

$$x_{ip} \le y_p\ \forall i \in N, p \in \tilde{N} \tag{8}$$

$$f_p^r + \sum_{i\in N} x_{ip}\hat{c}_i^r \le \tilde{c}_p^r\ \forall p \in \tilde{N}, r \in R \tag{9}$$

$$\sum_{i\in N}\sum_{j\in N}\sum_{q\in\tilde{N}}\sum_{e\in E} x_{ip}(1-x_{jp})x_{jq}z_{ei}z_{ej}b_e \le \tilde{b}_p\ \forall p \in \tilde{N} \tag{10}$$

$$x_{ip}(1 - x_{jp})x_{jq}z_{ei}z_{ej}\delta_e \geq \tilde{\delta}_{pq} \ \forall i, j \in N, p, q \in \tilde{N}, e \in E \quad (11)$$

$$x_{ip} \in \{0, 1\} \ \forall i \in N, p \in \tilde{N} \quad (12)$$

$$y_p \in \{0, 1\} \ \forall p \in \tilde{N} \quad (13)$$

Constraint (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. 8). Furthermore, Eq. (9) ensures that physical host capacities are not exceeded. Constraint (10) indicates that the sum of bandwidth requirements of cross-host virtual links using the same network interface should not exceed its bandwidth capacity. Finally, Eq. (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* [17]. Hence, in the following, we provide a simple yet effective heuristic to solve it.

B. Heuristic solution

In the following, 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. 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. 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 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.

Algorithm 1 Emulated Network Embedding

```
1: Ñ_a ← Set of active physical hosts
2: N_u ← N {Set of unassigned switches}
3: while N_u ≠ ∅ do
4:    Sort N_u in decreasing order according to R_i (Eq. 17)
5:    i ← first node in N_u
6:    Ñ(i) ← hosts Ñ_a satisfying resource requirements of i.
7:    if Ñ(i) ≠ ∅ then
8:       Sort Ñ(i) in decreasing order according to F_ip (Eq. 20)
9:       p ← first node in Ñ_i
10:   else {Need to switch on a physical machine}
11:      Activate physical host p that satisfies resource requirement of i, and if not possible set p = −1.
12:   end if
13:   if p ≠ −1 then
14:      Ñ_a ← Ñ_a ∪ {p}
15:      Assign virtual switch i to physical machine p
16:      N_u ← N_u \ {i}
17:   else
18:      return the emulated network is not embeddable
19:   end if
20: end while
```

- 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 cross-host links. Thus, we define the degree ratio of virtual switch i as:

$$\mathcal{R}_i^D = \frac{\sum_{e \in E} z_i^e}{\max_{j \in N} \sum_{e \in E} z_j^e} \quad (14)$$

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

$$\mathcal{R}_i^C = w_b \frac{\sum_{e \in E} z_i^e b_e}{\max_{j \in N} \sum_{e \in E} z_j^e b_e} + \sum_{r \in R} w_r \frac{\hat{c}_i^r}{\max_{j \in N} \hat{c}_j^r} \quad (15)$$

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 cross-host links and the consumed physical bandwidth as well. We define this locality ratio as:

$$\mathcal{R}_i^N = \frac{\sum_{j \in N} \sum_{e \in E} \sum_{p \in \tilde{N}} z_i^e z_j^e x_{jp}}{\sum_{j \in N} \sum_{e \in E} z_i^e z_j^e} \quad (16)$$

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

$$\mathcal{R}_i = \gamma_D \mathcal{R}_i^D + \gamma_B \mathcal{R}_i^B + \gamma_N \mathcal{R}_i^N \quad (17)$$

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

Algorithm 1 evaluates $\mathcal{R}_i$ for all unembedded virtual switches and selects the one with the highest value to embed first. The next step is to select the physical machine which will accommodate 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:

$$\mathcal{F}_{ip}^R = \sum_{r \in R} w_r \frac{\min_{q \in \tilde{N}} u_{iq}^r}{u_{ip}^r} \quad (18)$$

where u^r_{iq} is the estimated residual capacitive for resource r in physical node q when it is hosting virtual switch i. Minimizing residual capacities of physical machines result in less resource fragmentation and higher machine utilization.

Furthermore, in order to minimize the number of cross-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:

$$\mathcal{F}^N_{ip} = \frac{\sum_{j \in N} \sum_{e \in E} \tilde{x}_{jp} z^e_i z^e_j}{\max_{q \in \tilde{N}} \sum_{j \in N} \sum_{e \in E} \tilde{x}_{jq} z^e_i z^e_j} \tag{19}$$

where 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:

$$\mathcal{F}_{ip} = \lambda_R \mathcal{F}^R_{ip} + \lambda_N \mathcal{F}^N_{ip} \tag{20}$$

where λ_R and λ_N are the weights of each factor in the selection criterion. Specifically, the physical machine p with the highest $\mathcal{F}_{ip}$ 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.

V. Evaluation

In this section we provide performance evaluation of our proposed heuristic compared to the *First Fit (FF)* approach. We also show that in contrast to Mininet, DOT can provision adequate resources to ensure guaranteed switching capacity for each virtual switch. In the rest of this section we first describe our experimental setup and then we present results obtained from a DOT deployment in our cluster.

A. Experimental Setup

We have deployed DOT on 10 physical machines organized in two racks. The central DOT manager has been deployed on a separate host connected to the same network. All machines run Ubuntu 12.04 as the host OS. 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 for machine virtualization and Libvirt 1.0.0 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 [19] controller for the experiments in this paper. However, any existing OpenFlow controller 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 also generates the system commands to deploy DOT on a distributed infrastructure. We compared the performance of our proposed heuristic with the first fit (FF) approach. This approach randomly chooses a physical machine for each virtual switch, and embeds it if adequate residual resources are available in that machine.

TABLE I. 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

B. Results

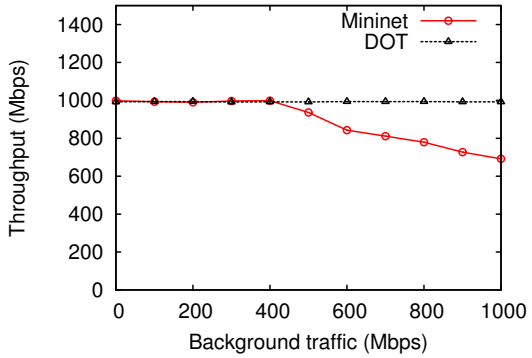
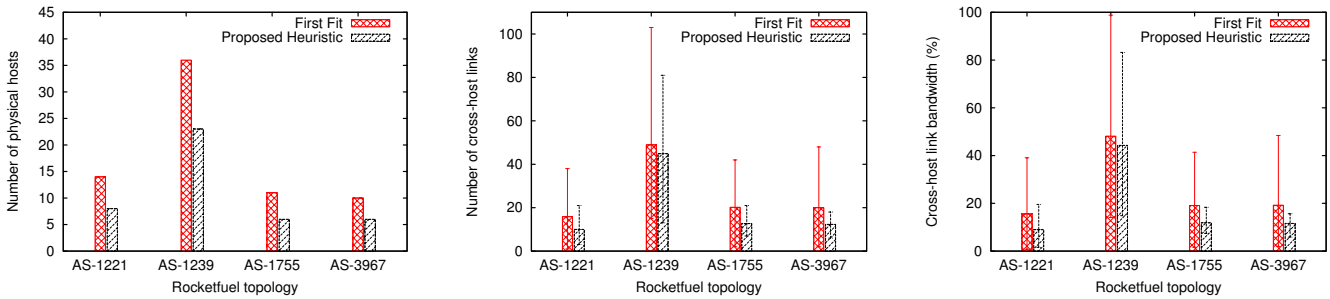


Fig. 6. DOT vs. Mininet

In our first experiment, we deployed the fat-tree like topology shown in Fig. 1(a) on DOT (spanning two physical hosts) and performed similar experiment as in Fig. 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. 6. 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.

Next, we evaluate the performance of our proposed heuristic against that of FF. We embedded four ISP topologies (Table I) from the RocketFuel repository [20]. We computed the number of physical hosts required to embed these topologies. The result is shown in Fig. 5(a). For AS-1221, FF requires 14 hosts whereas our heuristic require 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 (Equation 4) and as a result more virtual switches can be embedded on the same physical host leading to a more compact embedding. Fewer cross-host links also indicate that the embedding process is



(a) Number of physical hosts (b) Number of cross-host links (c) Amount of cross-host bandwidth

Fig. 5. Evaluation results

able to embed highly connected portions of input topology on the same physical host.

Fig. 5(b) 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 link 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 is 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 requirements (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. 5(c) 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 the 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%.

VI. Conclusion

In this paper we presented the design and management of DOT. To the best of our knowledge, DOT is the only distributed emulator for SDN. To emulate a given network and traffic load, DOT distributes the network components over the available physical machines. DOT provides guaranteed resources (computation and bandwidth) for each emulated component (*i.e.*, switches, hosts and links). It has built-in support for configuring and monitoring the emulated components from a central management module. Furthermore, we formulated a mathematical model and proposed a heuristic algorithm to optimize resource utilization while embedding an emulated network into the cluster of machines. Experimental results show that DOT is able to overcome scalability issues of Mininet and to guarantee 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.

Currently, DOT is capable of using a fixed number of physical machines to emulate a given network. However, our framework can be easily extended to dynamically add hardware resources on the fly to scale up or down with the changes in the emulated networks. We intend to extend DOT with this dynamic scalability feature. We also want to add multi-user support for running multiple emulations on the same physical infrastructure simultaneously. We will improve the usability of DOT by implementing a generic RESTful API for remote monitoring and management. Finally, we will incorporate a configurable logging facility for tracking OpenFlow messages during the emulation. These features will significantly improve the usability of DOT.

Acknowledgement

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.

References

[1] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, Mar. 2008.

[3] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic controller provisioning in software defined networks," in *International Conference on Network and Service Management (CNSM)*, 2013.

[4] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "PolicyCop: an autonomic QoS policy enforcement framework for software defined networks," in *Software Defined Networks for Future Networks and Services (SDN4FNS)*, 2013.

[5] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A Low Cost Netowrk Monitoring Framework for Software Defined Networks," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2014.

[6] A. Roy, K. Yocum, and A. C. Snoeren, "Challenges in the emulation of large scale software defined networks," in *Asia-Pacific Workshop on Systems (APSYS)*, 2013.

[7] http://www.geni.net/.

[8] http://fp7-ofelia.eu.

[9] http://openvswitch.org.

[10] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending Networking into the Virtualization Layer," in *Workshop on Hot Topics in Networks (HotNets)*, 2009.

[11] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Generic Routing Encapsulation (GRE)," RFC 2784, Internet Engineering Task Force, Mar. 2000.

[12] https://github.com/noxrepo/nox.

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

[14] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012.

[15] A. Voellmy and J. Wang, "Scalable software defined network controllers," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 289–290, Aug. 2012.

[16] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proceedings ACM SIGCOMM*, 2009.

[17] C. Chekuri and S. Khanna, "On multi-dimensional packing problems," in *Annual ACM-SIAM symposium on Discrete algorithms (SODA)*, 1999.

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

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

[20] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," in *ACM SIGCOMM*, 2002.