Auto-Mininet: Assessing the Internet Topology Zoo in a Software-Defined Network Emulator

Marcel Großmann, Stephan J.A. Schuberth
Faculty of Information Systems and Applied Computer Science
Otto-Friedrich University,
Bamberg, Germany
Email: marcel.grossmann@uni-bamberg.de
stephan-johannes-albert.schuberth@stud.uni-bamberg.de

***Abstract*—Among all virtualization approaches, Software-Defined Networking exploits this paradigm on the communication level and offers an "operating system" for networks. Therefore, the control logic of packet processing devices is moved onto external controllers and these decoupled control planes offer more reliability, scalability and performance compared to purely distributed systems. This fact is moving the network towards the application layer and offers applications direct interaction within the network. However, the requirements to build a virtualized network are fairly hard to accomplish and therefore, efficient testbed environments should reveal the success. In this field the container-based emulation used by Mininet is able to simulate large topologies on a single computer. We develop an automatic initialization of Mininet topologies and provide the ability to build up real-world test-suites at small scale. To achieve this, we use graphical formats of the Internet Topology Zoo to generate networks according to the Mininet syntax. Additionally, we develop a distributed test suite for evaluation purposes that uses a distributed Internet traffic generator. A statistical analysis of the generated traffic reveals the performance of Mininet for the topologies.**

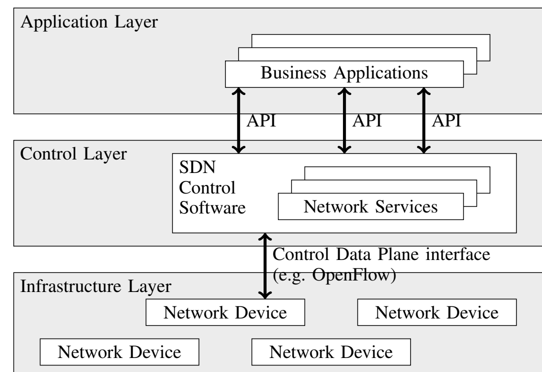


Fig. 1. A SDN approach to separate several layers and introduce transparency of the network to the business applications. The application layer communicates through an API with the control software of the SDN that is providing required network services. Via a control data plane the controller are connected to the infrastructure layer and can change network device configurations at runtime [2].

I. Introduction

Over the last decades virtualization revolutionized the structures of the computing area. However, communication infrastructures have remained virtually untouched, but there are upcoming approaches to virtualize this segment too. Software-Defined Networking (SDN) is a paradigm that uses controller instances to supervise the hops in a network topology. Figure 1 depicts several layers of a SDN approach. The *application* layer is holding the business applications and introduces transparency of the network to the end users. The application layer communicates through an application programming interface (API) with the control software of the SDN that is providing required network services. The controllers are connected to the *infrastructure* layer through a control data plane and can change network device configurations at runtime. Among other approaches McKeown *et al.* [1] introduce the *OpenFlow* protocol that is an essential element for developing SDN solutions. Here, it is the first vendor-independent communication interface that is placed between the control and forwarding layers of a SDN. Therefore, OpenFlow is maintained by the Open Networking Foundation (ONF), which is a user-driven organization that is implementing SDN with open standards [2].

A major problem arises by proving the functionality of such a system. Setting up environments is too cost intensive, especially for deploying the system on a complete hardware based testbed. Here is, where virtualization is part of the game again. With Open vSwitch [3] being a software-defined network stack without the need of hardware-based infrastructure a topology is virtualizable within a single machine. Lantz *et al.* [4] introduce Mininet, a framework that runs on a single computer and is able to simulate software-defined network topologies with respect to their conditions.

Now one question arises: How do we represent real-world topologies in such a testbed? In our paper we define a methodology to build up Mininet configurations based on graphical maps of the Internet Topology Zoo (ITZ) [5]. This is a platform that provides several topologies of the real-world in a graphical notation, containing the geographical location of the switches or routers, in the following called hops, and their interconnections. The parser uses these graphs to generate a topology for the testbed and you can migrate all existing topologies to Mininet onto your computer.

For the evaluation of the Mininet topologies, we used the distributed Internet traffic generator (D-ITG), Avallone *et al.* [6] developed. It generates traffic in several ways and is able to collect all relevant statistics. Within a simple test topology we reveal, how Mininet performs in setups with or without cross-traffic and which effect a breakdown of a hop induces.

II. Software-Defined Networking

Traditional networks are shaped by their hardware components and the software that runs on these components. Ideally, hardware itself should satisfy the following requirements. First, it is *simplicity*, because it should be inexpensive to build hardware to run networks. Second, *vendor-neutrality* describes that parts of several vendors should work together in a mixed setup. Third, it is a *future-proof* concept since there is no need to suddenly upgrade hardware to support future innovations. Besides, the software side of networking basically has a single requirement: It is *flexibility* to easily update the control plane to support inexpensive changes to the network by its operators. However, not a single goal is reached by any of the popular technologies today. Initially, network deployments are unnecessarily expensive, difficult to change in runtime, expensive to maintain and not very user-friendly with maintenance downtimes. In the past, several approaches have been tried, with almost all of them having failed to reach any significant importance.
Network design depends on two major influence factors. On the one hand, network *requirements* arise by its users and operators. Users usually want their data (their packets) to arrive at the desired destination with several Quality of Service (QoS) attributes being satisfied. However, network operators have different needs like traffic engineering, virtualization, tunneling or network separation, which are often transparent to the users or even irrelevant. On the other hand, network *interfaces* define the network through technical implementation details. The packet header of a given protocol broadcasts the users' requirements from their hosts to the network and is transparent to any kind of job that is performed. For the operators a more tedious and time-intensive configuration of the network needs to be fulfilled in order to get each network node running. Finally, a packet identifies to a switch through specific data in its header.

In the original Internet operator requirements are negligible. Only packets flowing from source to destination hosts were concerned, while each hop updates the routing information to forward the packet. Due to the fact that there is no network interface for the operator, the hardware is manually programmed at each hop. The host-network and packet-switch are basically one interface.
Multiprotocol Label Switching (*MPLS*) [7] started differentiating the "core" and the "edge" of the network. A label is assigned to packets arriving at the network edge that is used to forward them, when they are traversing the core. The labels serve both, the purpose of forwarding inside the network as well as satisfying operator requirements like traffic engineering or tunneling. The label information itself is decoupled from the host protocol, i.e. IPv4, which expresses the host requirements to the network. While the Internet Protocol (IP) remains the interface to express host requirements, the label is the interface that is responsible for the packet switching. For broadcasting the requirements of operators nothing changed compared to the original Internet.

The biggest change SDN brings is the introduction of a programmable interface for network operators. The control plane is decoupled from the data plane, this means the topology of nodes that transfer data is different from the one that is used to propagate control changes. Figure 2 shows the separation of planes that is achieved through the coexistence of two different topologies. The *controller plane* is basically a tree with the controller as root node being connected to all other switches and routers. All attached hosts cannot influence the controller plane, because it is transparent to the outside of the network. The network topology is represented by the actual *data plane* that is not taking the controller into account. There exist several network designs with this level of separation, i.e. Ethane [8] or OpenFlow [1].
The defacto standard protocol in the SDN field is OpenFlow and will be discussed in more detail here. Hops are constructed to export an extra interface solely to a controller that is used to manage forwarding, which is done through application of mappings in forwarding tables between packet header fields and actions that are taken upon encounters of matching packets. Header parts to be matched can be part of standard protocols that are widely used today, like IP or the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Actions are i.e. sending a packet to a specific port or rewriting its protocol header data.

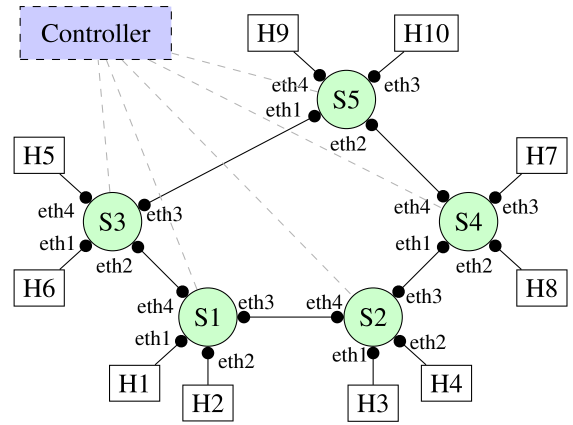


Fig. 2. Separation of two planes in SDN. $S1$ to $S5$ are the hops of the network. They construct the network topology which is depicted with the lines being attached to the switches via eth interface. The dashed lines represent the control plane that connects every hop to the controller via a secure channel.

This architectural design has several advantages. The network operator controls the hardware from a single point and rolls out new setups and changes by programming the network's controller, instead of programming every single hardware instance manually. The flexibility of SDN to collectively control the data flows via forwarding table entries can be used to implement requirements such as an isolation of different networks.
However, SDN design has also a few disadvantages. SDN in the OpenFlow implementation does not simplify network hardware. The flow table matching to apply rules provided by the controller still requires to parse all headers, which is simpler in the MPLS approach. If external network protocols change (i.e. IPv4 to IPv6), also the matching behavior in a SDN has to be changed, to keep on working. This is due to the fact that header informations and matching rules are no longer fitting, which can be improved by a MPLS approach on top. With it, only the network edge has to be updated, to switch the network back to a working state.

Header Fields	Counters	Actions

TABLE I. REQUIRED FIELDS FOR A FLOW TABLE ENTRY [1].

In Port	Ethernet			VLAN ID	IP			TCP/UDP	
	SA	DA	Type		SA	DA	Proto	SRC	DST

TABLE II. THE HEADER FIELDS THAT ARE MATCHED IN AN OPENFLOW SWITCH [1].

A. *OpenFlow*

McKeown *et al.* [1] describe, how they enable innovation in campus networks, using the OpenFlow specification. The inspiration for OpenFlow was the identification of a common set of functions that are implemented by several routers and switches. Thus, this open protocol allows network administrators to partition traffic into different flows, since the functionality is transparent to the operator of the network.

Like depicted in Table I, each flow table consists of three fields. A packet *header* that defines the flow, an *action*, which defines how packets should be processed and statistics. The latter are *counters* that keep track of the number of packets and bytes for each flow and the time since the last packet matched the flow, mainly to help with the removal of inactive flows. For the matching procedure the header fields of Table II are concerned. The data path of an OpenFlow Switch presents a clean flow table abstraction; each flow table entry contains a set of packet fields to match and an action (such as send-out-port, modify-field, or drop). When an OpenFlow Switch receives a packet it has never seen before, for which it has no matching flow entry, it sends this packet to the controller. The controller then decides on how to handle this packet. It can drop the packet or it can add a flow entry directing the switch on how to forward similar packets in the future.

A controller is attached to the network via a separate secure channel, like Figure 2 shows. There exist several implementations of such controllers. A first implementation of an OpenFlow controller is *NOX* [9], which is written in C++ and Python. A purely C implementation that has a multi-threaded infrastructure at its core is *MūL* [10]. It is designed for performance and reliability, which is needed from the beginning of the deployment in mission-critical networks. The *ovs-controller* [11] is a trivial reference controller that comes packaged with Open vSwitch [3]. A variation to use Ruby is given with *Trema* [12], which is a full-stack framework for developing OpenFlow controllers in Ruby and C. *POX* [9] as a successor of NOX is a high-level SDN API, since it is programmed in Python and therefore platform independent. POX offers a queriable topology graph and support for virtualization. Another Python based approach is *Ryu* [13], which is an open-source Network Operating System (NOS) that supports OpenFlow. Finally, some Java controllers are missing. *Jaxon* [14] depends on NOX and is not platform independent at all. *Maestro* [15] is capable to orchestrate network control applications. *Beacon* [16] supports event-based and threaded operations. Originally, the Apache-licensed OpenFlow controller *Floodlight* [17] was forked from the Beacon controller. It is supported and improved by a community of developers and it supports a broad range of virtual and physical OpenFlow switches. It is chosen for controlling our networks in Section III. Furthermore, with *NodeFlow* there are JavaScript approaches to implement an OpenFlow controller for Node.JS [18].

Besides of the listed controllers, there are also special purpose controllers. *RouteFlow* [19], e.g., is an open source project to provide virtualized IP routing services for OpenFlow enabled hardware, which is mainly written in C++ and Python. RouteFlow is composed of an OpenFlow controller application, an independent *RouteFlow* server and a virtual network environment that reproduces the connectivity of a physical infrastructure and runs IP routing engines. For using multiple OpenFlow controllers *Flowvisor* [20] acts as a transparent proxy. Simple network access control (*SNAC*) [21] is an OpenFlow controller built on NOX, which uses a web-based policy manager to manage the network. *Oflops* [22] is a standard controller that benchmarks various aspects of an OpenFlow Switch.

There are options available, but none so far was able to satisfy the needs to prototype networks. Hardware-based setups were too expensive and thus lacked flexibility, scalability, shareability and affordability. Prior software-based solutions like a network of virtual machines lacked scalability due to being too heavyweight and overly resource intense and so lacked also shareablity. Software testbeds had a mismatch with simulation code not being deployable and also lacked interactivity. Other approaches tried to fix the shortcomings with distributed testbeds, thus being able to support large topologies and also making these shareable. Still having a solution to run on only a single workstation would further fix scalability, shareability and affordability for people not having access to these test networks. With the advent of SDN another solution arose, called Mininet. It promises to satisfy all the above needs at once using SDN technology. Evaluation of this claim will be done in Section IV.

B. *Mininet*

For prototyping networks further requirements need to be fulfilled. To increase *flexibility*, topologies should be defined software-wise and not on a hardware level, which would reduce costs during the setup and development time. Network designs created on the software level should be *deployable* on actual hardware without having to alter code or configurations. *Interactivity* is given by running, managing and testing designs in real-time. *Scalability* is needed to depict real-world networks, where a single workstation simulates networks with hundreds of nodes, if not thousands. In a testbed protocols and characteristics of networks should be expressed as close as possible to deployed network instances in a productive environment. Network designs should be *shareable* between collaborators without complicated setups and configurations. Testbeds should be cheap enough, to be *affordable* for people needing those, either in academia or at work in companies.

The magic behind Mininet's illusion is a set of features built into Linux that allow a single system to be split into a bunch of smaller containers, each with a fixed share of the processing power, combined with virtual link code that allows links with accurate delays and speeds. Internally, Mininet employs lightweight virtualization features in the Linux kernel, including process groups, CPU bandwidth isolation, and network *namespaces*, and combines them with link schedulers

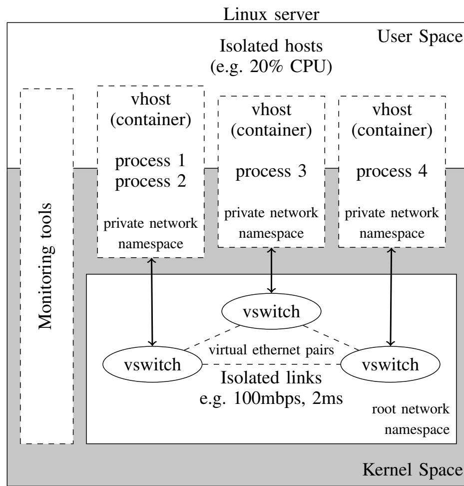


Fig. 3. Container based emulation in Mininet based on different namespaces. Every vhost is a container, which provides its own private network namespace and is participating in the network with a connection to a virtual switch [23].

and virtual Ethernet links. These features yield a system that starts faster and scales to more hosts than emulators, which are based on full virtual machines. For example, $1,057$ nodes allocate $492MB$ of memory and start up at about $817s$ [4]. Therefore, a Mininet network consists of three elements [23], as depicted in Figure 3.

- *Virtual hosts*, which are a group of user-level processes moved into a network namespace. Network namespaces themselves are containers for the network state and provide process groups with exclusive ownership interfaces, ports and routing tables. For example, two web servers in two network namespaces can coexist on one system, both listening to private Ethernet interfaces on the same port. To guarantee fairness, Mininet limits the processor bandwidth for each process group to a fraction of the CPU time available.
- *Virtual switches* that typically use the default Linux bridge or Open vSwitch are running in kernel mode to switch packets across interfaces. Moreover, switches and routers may run in the kernel space to speed up the packet switching or in the user space to modify them easily.
- *Isolated links* are defined with an explicit data rate, which is enforced by Linux traffic control that consists of several packet schedulers to shape traffic to a configured rate. A virtual Ethernet pair acts like a wire connecting two virtual interfaces or virtual switch ports. Packets sent through one interface are delivered to the other and each interface appears as a fully functional Ethernet port to all system and application software.

III. Topologies in Mininet

The SDN emulator from Subsection II-B needs topologies that are defined in Python for its execution. Though, the ITZ, from Subsection III-A, provides a database with many real-world topologies, they are defined in a graphical notation. To make them available in Mininet version 2.0.0, a possibility is to use the topology parser from Subsection III-B.

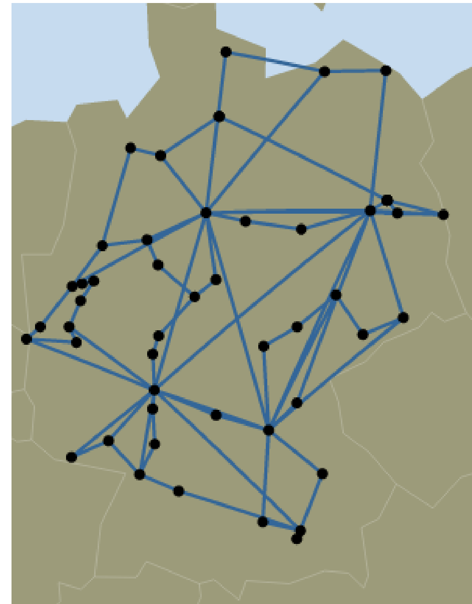


Fig. 4. DFN network topology that was published in January 2011. All routers/switches of the topology are depicted as black circles, while the links are painted in blue lines.

A. The Internet Topology Zoo

The ITZ [5] is a store for data of network topologies. A lot of attention was given through the study of network topologies, but it stagnated by a lack of accurate data. Methods for measuring topologies have flaws, such that all arguments circled around the methods and have overcome the important questions about network structures. Since operators publish information about their networks, the ITZ database contains topologies from *AboveNet* to *Zamren*. Therefore, the ITZ is the most accurate collection of networks that is publicly available and includes meta data, which is not measurable. In our examples of Subsection III-B we use the German research network (DFN[1]) topology as depicted in Figure 4.

All topologies are in a graphical format that uses the extensible markup language (XML) as description basis. Validity of each topology in the `GraphML` format is given by the XML schema definition (XSD) for GraphML[2] files. The graphical format provides enough information to build up testbed networks with respect to real world topologies. Listing 1 depicts the important elements to structure the network. To identify, which purpose the `key` elements are used for, the `attr.name` field contains the relevant information. Here, especially the *label* for a node and its *longitude* and *latitude* positions are needed to rebuild the network in Mininet. Moreover, the *edge* element specifies the connections between the nodes and is needed to link the switches in Mininet.

[1] "Deutsches Forschungsnetz"
[2] `http://graphml.graphdrawing.org/xmlns`

```xml
<?xml version="1.0" encoding="utf-8"?>
<graphml ...>
  <key attr.name="key" attr.type="int" for
     ="edge" id="d36" />
   ...
  <graph edgedefault="undirected">
    <data key="d0">2/01/11</data>
    <data key="d1">Germany</data>
    <data key="d2">Country</data>
    <data key="d3">DFN</data>
     ...
    <data key="d28">1</data>
    <node id="0">
      <data key="d29">1</data>
      <data key="d30">50.83333</data>
      <data key="d31">Germany</data>
      <data key="d32">0</data>
      <data key="d33">12.91667</data>
      <data key="d34">CHE</data>
    </node>
     ...
    <edge source="0" target="1">
      <data key="d35">e52</data>
      <data key="d36">0</data>
    </edge>
     ...
  </graph>
</graphml>
```

Listing 1. Selection of important values of the *graphml* format for the DFN topology depicted in Figure 4.

```python
#!/usr/bin/python

from mininet.topo import Topo
...

class GeneratedTopo( Topo ):
    def __init__( self, **opts ):
        # Initialize Topology
        Topo.__init__( self, **opts )
        # switches first
        CHE = self.addSwitch( 's0' )
        ...
        # and now hosts
        CHE_host = self.addHost( 'h0' )
        ...
        # add edges between switch and
            corresponding host
        self.addLink( CHE , CHE_host )
        ...
        # add edges between switches
        self.addLink( CHE , LEI, bw=10,
            delay='0.348009502ms')
        ...
topos = { 'generated': ( lambda:
    GeneratedTopo() ) }
...
if __name__ == '__main__':
    sshd( setupNetwork() )
```

Listing 2. The transformation of Listing 1 to Python for establishing the topology in Mininet.

B. The Topology Generator for Mininet

Since all topologies available in the ITZ have a quite similar structure, parsing them to generate executable Mininet topologies is possible with ease. Topologies to be used in Mininet are executable/loadable Python classes interfacing with the Mininet API. So each usable Mininet topology is similar, having the same content in the *head* and the *tail* of a file. The parts, where the files basically differ is the definition of

- *host* entries,
- *switch* entries and
- *links* between switches and hosts.

The difference between *executable* and *loadable* files is in the code at the end of the Python script. If the topology is executed from a Linux shell, Mininet is automatically started with the topology defined and secure shell (SSH) access available. *Loaded* means that Mininet was started alone from the Linux shell and the topology was given as a calling argument via the *topo* parameter. The code that defines SSH access to the topology nodes is not executed, only the part defining that the topology is used. In return this means using the topology just by *loading* SSH access is not available. So the preferred usage is to directly execute the topology from a Linux shell. There are several command line arguments available for using the ITZ Parser. Through them you can define a GraphML input file for the topology that should be parsed to a Python topology that Mininet can use. An adjustment of the bandwidth in $Mbps$ is also possible by setting the `bw` value of all edges to the given value. Further, the parser requires the files to be located in the same directory and without specifying input parameters the program will terminate. However, some values can be omitted, like the bandwidth limitation, which is otherwise initialized to $10Mbps$. If omitted, the remote controller IP is initialized with "10.0.2.2", which is the standard IP for the host OS when using Oracle Virtualbox for virtualization. Additionally, the delay for Mininet is calculated by using the geographical coordinates of the ITZ topologies.

$$dist(SP, EP) = \arccos\{\sin(La_{EP}) \cdot \sin(La_{SP}) + \cos(La_{EP}) \cdot \cos(La_{SP}) \cdot \cos(Lo_{EP} - Lo_{SP})\} \cdot r \quad (1)$$

In the topology distances are calculated by the spherical law of cosines that is described by Equation 1, where the radius is assumed about $r = 6,378,137m$. SP is the starting point, while EP describes the end point. La is the latitude value and Lo the longitude value of a geo-coordinate, which must be given as radian values [24].

$$t_L = \frac{dist(SP, EP)}{v_L} \quad (2)$$

Moreover, the signal speed is approximated with the speed of light and the reflective factor of 1.52 for optical fiber, such

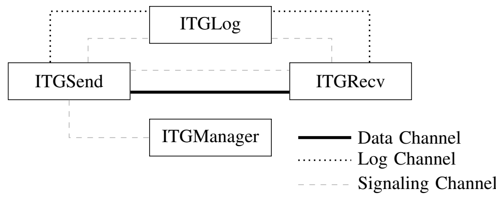


Fig. 5. The architecture of the D-ITG traffic generation. Every module is connected through several communication channels to other modules. `ITGSend` performs the traffic generation and sending processes, while `ITGRecv` receives it. `ITGLog` is a storage to collect all log files, while `ITGManager` is used for the remote control [6].

that $v_L = \frac{3 \cdot 10^8}{1.52} \frac{m}{s} = 1.97 \cdot 10^8 \frac{m}{s}$ [25]. Therefore, the estimated latency time t_L is calculated by Equation 2.

For example, Listing 1 provides in the fields `d30` the latitude and in `d33` the longitude value of a geolocation, such that we can determine the delay between two points from this graphical format. First, we extract the geocoordinates for `CHE`, which are $La_{\texttt{CHE}} = 50.83333°$ and $Lo_{\texttt{CHE}} = 12.91667°$. The connection is built up to `LEI` with $La_{\texttt{LEI}} = 51.33962°$ and $Lo_{\texttt{LEI}} = 12.37129°$. The distance between `CHE` and `LEI` is calculated by Equation 1, such that $dist(CHE, LEI) = 68557.871953m$. In the same way the delay is calculated by Equation 2, such that $t_L = \frac{68557.871953m}{1.97 \cdot 10^5 \frac{m}{ms}} = 0.348009502ms$. The latency is written into the python script of Listing 2 for initializing the edge between `CHE` and `LEI`.

IV. Evaluation Suite for Mininet

The evaluation of Mininet topologies is another concern. After being able to establish all topologies of the ITZ, the evaluation of several metrics needs to be concerned. For traffic generation, we use the D-ITG in Subsection IV-A and finally a few trials are performed with a Mininet topology in Subsection IV-B.

A. Distributed Internet Traffic Generator

To evaluate the performance of Mininet D-ITG in version 2.8.0-rc1 was used: "Distributed Internet Traffic Generator (D-ITG) is a platform capable to produce traffic that accurately adheres to patterns defined by the inter departure time between packets (IDT) and the packet size (PS) stochastic processes" [6]. Therefore, it offers a rich variety of probability distributions for the traffic generation and uses some models proposed to emulate sources of various protocols. With it, it is possible to generate various packet streams and collect statistics with a logging server. In Figure 5 all important modules of the D-ITG are depicted. The `ITGSend` module is responsible for the traffic generation, while the `ITGRecv` module is the sink for the packets, which are delivered over a *Data Channel*. To collect logging information both, the ITGSend and ITGRecv are communicating via a *Log Channel* with the `ITGLog` module. For remote control the `ITGManager` offers the functionalities to adjust parameters of ITGSend through the *Signaling Channel*.

Besides the modules of Figure 5 the D-ITG decoder (`ITGDec`) analyzes the results collected by the `ITGLog` module. It calculates the packet loss, throughput, jitter and delay, both the one-way delay (OWD) and the round-trip time (RTT). Moreover, it can analyze log information in real-time, e.g., if the sender is instructed by a controller entity to adapt the transmission rate based on channel congestion and receiver capacity.

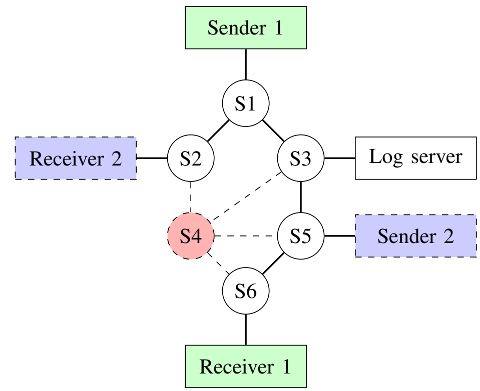


Fig. 6. The topology for the measurement trials. $S1$ to $S6$ are the switches in the topology, while Sender/Receiver denote the hosts that are handling generated traffic. Besides, the log server collects the relevant statistical data of the hosts. $S4$ is shutdown for a few measurement trials.

B. Measurement Trials

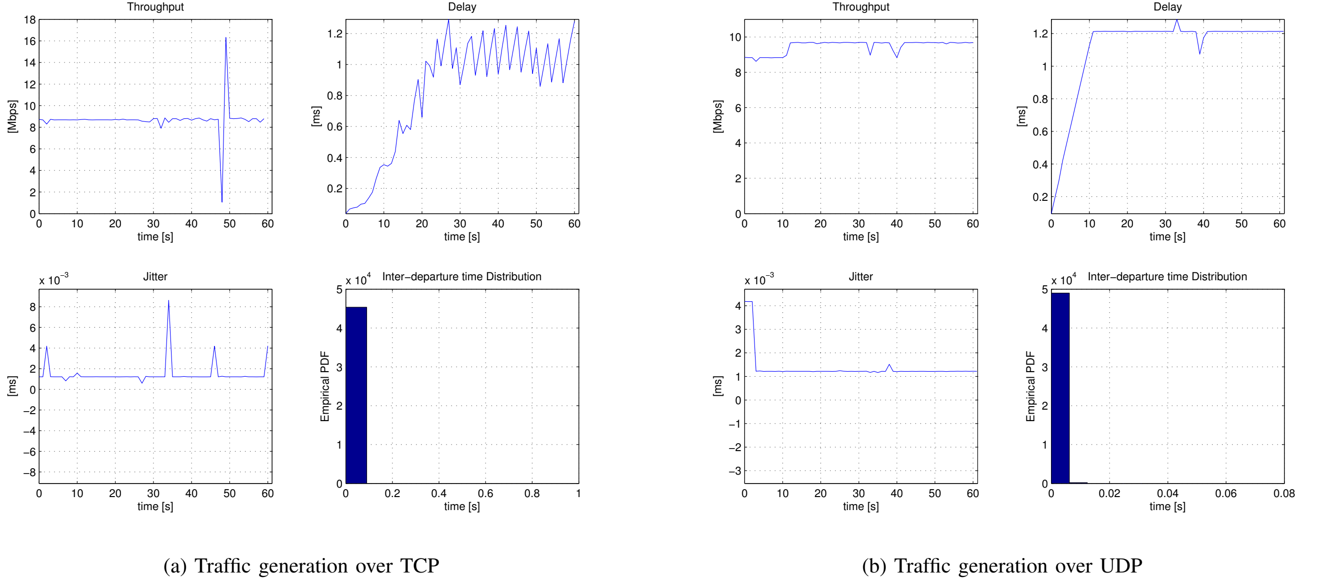
Figure 6 depicts the topology of our measurement trials. $S1$ to $S6$ are the switches in the topology, while Sender/Receiver denote the hosts that are handling generated traffic. On each sender a `ITGSend` process is called to generate the traffic, while on each receiver `ITGRecv` handles the receipt of the packets. Besides, the log server collects the relevant statistical data of the hosts by running an instance of `ITGLog`. $S4$ is shutdown for a few measurement trials and therefore all dashed links are unavailable. To sum up, in total we performed four trials, each for TCP and UDP with a duration of one minute for each trial in the following setups:

1) Only *Sender 1* and *Receiver 1* are responsible for the traffic on the topology without switch $S4$.
2) *Sender 1* and *Receiver 1* are handling the traffic on the full topology.
3) Both sender/receiver pairs are responsible for the traffic without switch $S4$.
4) Both sender/receiver pairs are handling the traffic on the full topology.

C. Results

The Mininet [4] virtual machine was running on a HP Proliant DL380p Gen8 server with Vmware[3] virtualization, while the Floodlight controller [17] is executed on a standard computer. For the trials the bandwidth of the SDN hops is limited to $10Mbps$. In D-ITG the traffic generation through `ITGSend` is setup with the maximum payload size for one packet, such that the traffic rate is adjusted by varying the IDT of two consecutive packets. The *test* bitrates reach from $8Mbps$ to $12Mbps$ in steps of $0.5Mbps$. For probing nine different bitrates a trial for one protocol and one setup has a duration of nine minutes. Therefore, it takes 18 minutes for two protocols and 72 minutes for the complete trial.

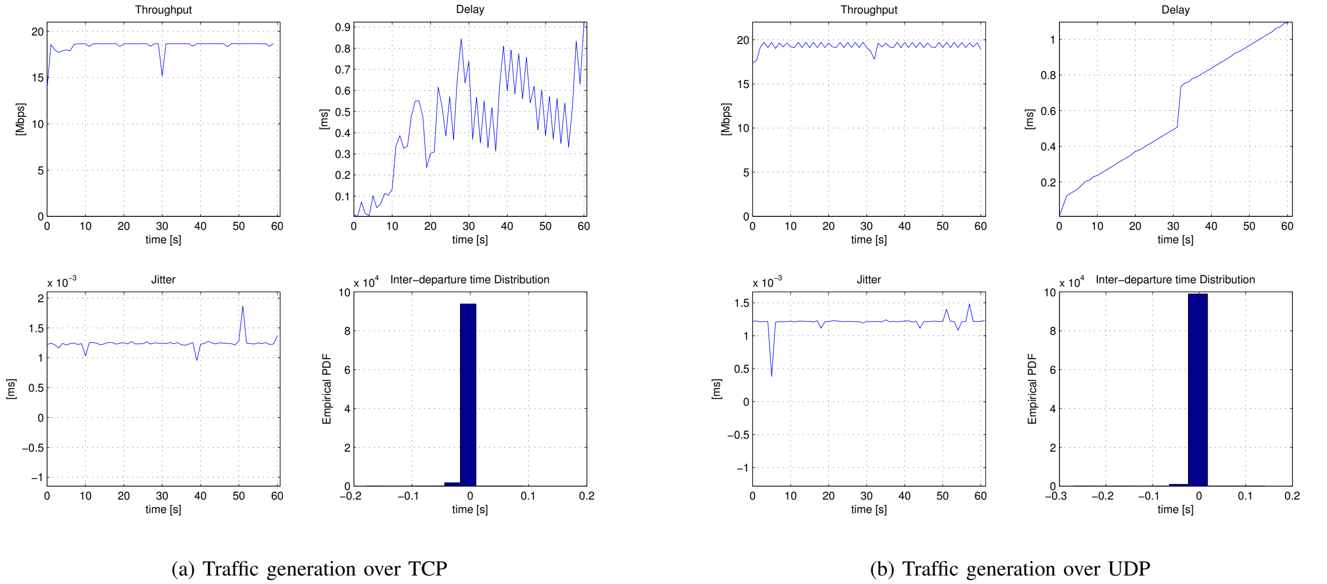
[3]`http://www.vmware.com`



(a) Traffic generation over TCP

(b) Traffic generation over UDP

Fig. 7. Evaluation of traffic with a constant *test* bitrate of $10Mbps$. Compared to Figure 6 the trial is setup with Sender/Receiver pair 1 and without switch $S4$. The delay within the edges of the SDN topology is $1ms$.



(a) Traffic generation over TCP

(b) Traffic generation over UDP

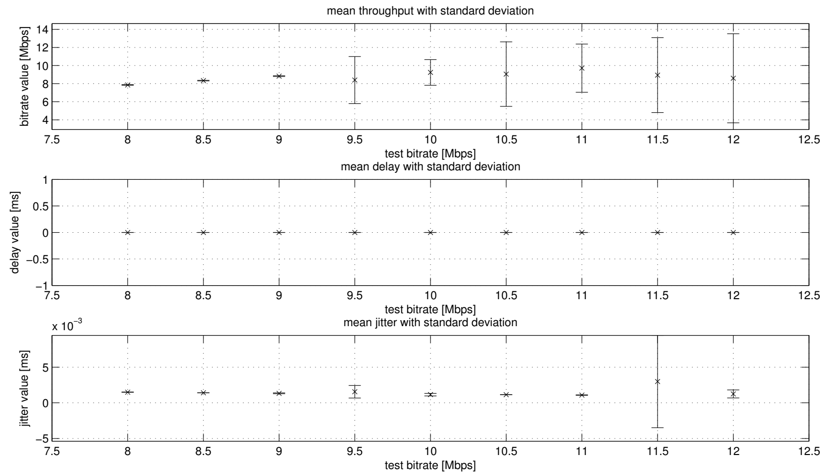
Fig. 8. Evaluation of traffic with a constant *test* bitrate of $10Mbps$ for each sender. Compared to Figure 6 the trial is setup with both Sender/Receiver pairs generating traffic to all switches of the topology. The delay within the edges of the SDN topology is $1ms$.

The D-ITG decoder provides data files that can be analyzed with MatLab [26]. Initially, every trial is evaluated and plotted as shown in Figure 7 and 8. In the plots, the first three graphs show specific characteristics in the time sequence from $0s$ to $60s$. In more detail, the upper left plot depicts the throughput in $Mbps$, while the upper right shows the delay in ms. The lower left plot is evaluating the jitter value in ms. It is calculated according to RFC 4689 [27], where D expresses the forwarding delay and i the order of the packets (Equation 3). The delay is further calculated by Equation 4, where S is the send time of a packet and R the receive time.

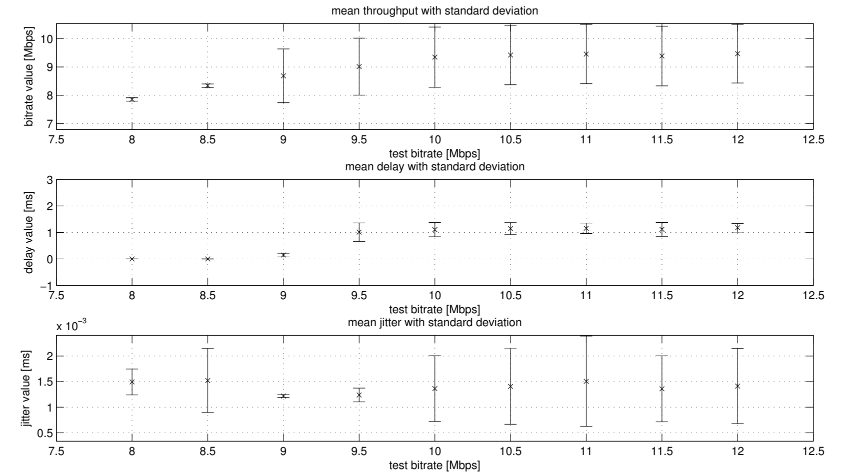
$$AvgJitter = \frac{\sum_{i-1}^{n} |D_i - D_{i-1}|}{n} \tag{3}$$

$$D_i = R_i - S_i \tag{4}$$

The lower right graph depicts the distribution of the inter-departure times (IDT).

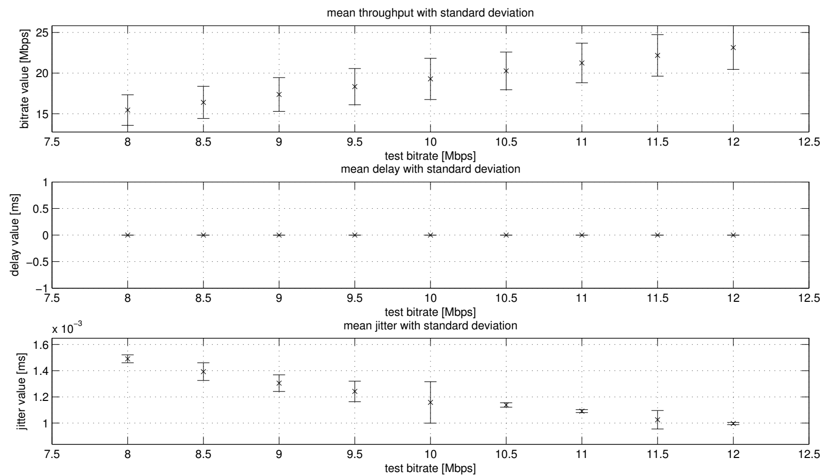


(a) Sender $S1$

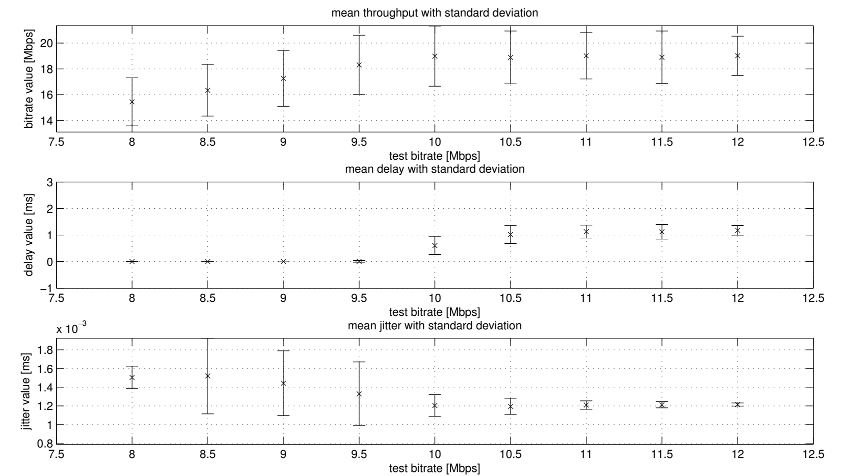


(b) Receiver $R1$

Fig. 9. Evaluation of UDP by increasing the *test* bitrate of the generated traffic. Compared to Figure 6 the trial is setup with one Sender/Receiver pair and without switch $S4$ and the constant bitrates are increased from $8Mbps$ to $12Mbps$. Each plot depicts the mean value and the corresponding standard deviation expressed through two bars. The first graph plots the throughput in $Mbps$, the second graph the delay in ms and the third graph the jitter value in ms.

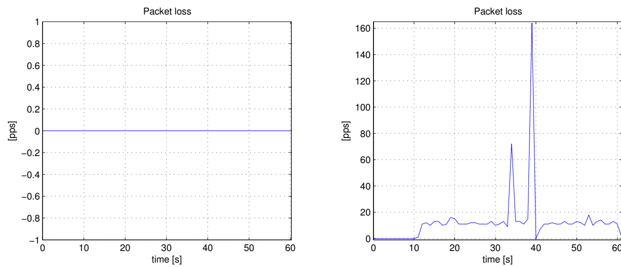


(a) Sender $S1$ and $S2$



(b) Receiver $R1$ and $R2$

Fig. 10. Evaluation of UDP by increasing the *test* bitrate of the generated traffic. Compared to Figure 6 the trial is setup with one Sender/Receiver pair and the constant bitrates are increased from $8Mbps$ to $12Mbps$. Each plot depicts the mean value and the corresponding standard deviation expressed through two bars. The first graph plots the throughput in $Mbps$, the second graph the delay in ms and the third graph the jitter value in ms.



(a) Evaluation of packet loss with a constant *test* bitrate of $9Mbps$

(b) Evaluation of packet loss with a constant *test* bitrate of $10Mbps$

Fig. 11. Evaluation of packet loss of the topology of Figure 6, which is setup with Sender/Receiver pair 1 and without switch $S4$. The delay within the edges of the SDN topology is $1ms$ and the bandwidth limitation of the SDN switches is $10Mbps$.

In Figure 7 the trial runs on the topology with sender/receiver pair 1 and without switch $S4$. We observe that the topology performs with a received throughput of about $9Mbps$ for the TCP (Figure 7a) traffic and nearly $10Mbps$ for the UDP (Figure 7b) traffic. Moreover, the TCP congestion control is clearly visible in the delay plot of Figure 7b and tries to find the threshold above $20s$. Furthermore, on the receiver side, depicted in Figure 7a and 7b we observe a linear increase of the delay to a certain threshold, which is about $1.2ms$ for UDP, though the sender is not responsible for this behavior. In comparison to the UDP loss graphs of Figure 11 the achievement of the $1.2ms$ delay of Figure 7b occurs at $10s$, which is the same time when Figure 11b shows the first packet loss in packets per second pps. Nevertheless, avoiding the threshold shows that all packets are transmitted, e.g., with a bitrate of $9Mbps$ (Figure 11a).
In Figure 8 the trial runs on the full topology with both sender/receiver pairs. A similar observation occurs again, while both senders generate traffic with a throughput of $10Mbps$ each, the total amount of the received bitrate is about $18.5Mbps$ for TCP (Figure 8a) and nearly $20Mbps$ UDP (Figure 8b). Again, the delay shows a linear increase for the UDP scenario, while there occurs a heavily oscillating increase in the TCP scenario, which is probably caused by overlapping congestions control mechanisms of the cross traffic generation. The IDT distribution depicts negative times which probably occur, when one hops, e.g., $S2$ is fully utilized such that the traffic flow from Sender 1 to Receiver 1 is using $S3$ for the following transmissions. Packets over the new route arrive faster than their predecessors on the old route, which produce negative results in the IDT.

Figure 9 and 10 collect the statistics for several trials in one graph. Therefore, the mean value and the corresponding standard deviation are calculated over the time schedules of the uppermost three graphs of Figure 7 and Figure 8. This values are plotted for different *test* bitrates reaching from $8Mbps$ to $12Mbps$. The first graph, the uppermost, depicts the throughput in $Mbps$, the second one the delay in ms and the third the jitter value in ms.

The sender always generates the constant *test* bitrate in every case of the performed measurements, without introducing additional delay (Figure 9a and Figure 10a), though the variance of the throughput generation is increasing by reaching the threshold. In Figure 9, running the trial of sender/receiver pair 1 without switch $S4$, we observe at the receiver side, Figure 9b, an increase of the throughput until the threshold of $10Mbps$ is achieved. The delay graph expresses a similar behavior until the bitrate achieves $10Mbps$. Later on the mean value of the delay converges to $1.2ms$. Within reaching $10Mbps$ the standard deviation is increasing and holds a similar level thereafter, while the mean value for the received throughput remains a bit below the bandwidth limitations.

In the trial with cross-traffic over the full topology, we observer that the throughput is steadily increasing until it achieves $10Mbps$ for both flows in Figure 10. For example, when both senders generate traffic with a constant bitrate of $10Mbps$ each, the overall received throughput is estimated with about $18.5Mbps$. There is no delay until the flows reach the threshold and thereafter it converges to $1.1ms$. In the jitter plot it is obvious that the more the bitrate increases the lower the mean value of the jitter will be due to the fact that packets are processed with nearly the same IDTs after the threshold.

V. Conclusion & Future Work

With the Internet Topology Zoo parser we lay the foundation to evaluate real-world network topologies in a SDN emulator. Furthermore, you are now able to specify networks in the GraphML syntax and simply migrate them to Mininet, without touching Python code. The simplicity to parse nearly every network to your own laptop reveals new possibilities to test several aspects of SDN behavior. For example, the controller placement problem described by Heller *et al.* [28] could be further analyzed with real-world or self-defined topologies. Additionally, the test suite, which is enabled with SSH access to all hosts, provides the ability to use traffic generation engines to evaluate network behavior to a certain extent. Here, the monitoring of SDNs can be tested in a scale-able environment that represents real-world topologies. According to the SDN open source initiatives, the topology parser and all results of the measurement trials will be made publicly available[4]. With the usage of D-ITG, we evaluated an exemplary topology, which was controlled by Floodlight. Our first measurement trial reveals that Mininet performs as expected within a given range of selected bitrates. In future experiments the network size needs to be increased, especially through using topologies of the ITZ, to evaluate, how many nodes Mininet can handle with acceptable performance. Basically, the SSH access is the foundation to push test modules onto the SDN, which is established within the usage of the ITZ parser. Furthermore, evaluations of different controllers could reveal their application area.

Acknowledgment

The authors would like to thank Walter de Donato for his kind support with the D-ITG framework, especially with the D-ITG decoder.

[4]The topology generator, the D-ITG automation for Mininet and the results of the measurement trials are available at `http://141.13.92.69/index.php/projects/auto-mininet`

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[2] Open Networking Foundation (ONF), "Sotware-defined networking: The new norm for networks," Tech. Rep., April 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[3] "Open vSwitch," May 2013. [Online]. Available: http://openvswitch.org

[4] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[5] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet Topology Zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[6] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, "D-ITG distributed Internet traffic generator," in *Proceedings of the First International Conference on the Quantitative Evaluation of Systems. QEST 2004.*, 2004, pp. 316–317.

[7] Z. Ren, C.-K. Tham, C.-C. Foo, and C.-C. Ko, "Integration of mobile IP and multi-protocol label switching," in *IEEE International Conference on Communications*, vol. 7, 2001, pp. 2123–2127.

[8] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *Proceedings of the 2007 Conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1282380.1282382

[9] "Noxrepo.org," May 2013. [Online]. Available: http://www.noxrepo.org

[10] "Mul," May 2013. [Online]. Available: http://sourceforge.net/projects/mul

[11] "ovs-controller," May 2013. [Online]. Available: http://openvswitch.org/cgi-bin/ovsman.cgi?page=utilities%2Fovs-controller.8

[12] "Trema: Full-Stack OpenFlow Framework in Ruby and C," May 2013. [Online]. Available: http://trema.github.io/trema

[13] "Ryu: Python-based OpenFlow controller but we aim for bigger pictures," May 2013. [Online]. Available: http://osrg.github.io/ryu

[14] "Jaxon: Java-based openFlow Controller," May 2013. [Online]. Available: http://jaxon.onuos.org

[15] "Maestro-platform: A scalable control platform written in Java which supports OpenFlow switches," May 2013. [Online]. Available: https://code.google.com/p/maestro-platform

[16] "Beacon," May 2013. [Online]. Available: https://openflow.stanford.edu/display/Beacon/Home

[17] "Project floodlight," May 2013. [Online]. Available: http://www.projectfloodlight.org/floodlight

[18] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *Internet Computing, IEEE*, vol. 14, no. 6, pp. 80–83, 2010.

[19] "RouteFlow," May 2013. [Online]. Available: https://sites.google.com/site/routeflow/

[20] "Flowvisor," May 2013. [Online]. Available: https://github.com/OPENNETWORKINGLAB/flowvisor/wiki

[21] "Simple network access control (SNAC)," May 2013. [Online]. Available: http://www.openflow.org/wp/snac/

[22] "Oflops," May 2013. [Online]. Available: http://www.openflow.org/wk/index.php/Oflops

[23] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th international Conference on Emerging networking experiments and technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 253–264. [Online]. Available: http://doi.acm.org/10.1145/2413176.2413206

[24] P. Eittenberger, M. Großmann, and U. Krieger, "Doubtless in Seattle: Exploring the Internet Delay Space," in *8th EURO-NGI Conference on Next Generation Internet (NGI)*, 2012, pp. 149–155.

[25] E. Hecht, *Optics (4th Edition)*. Addison-Wesley, 2001.

[26] "Matlab," May 2013. [Online]. Available: http://www.mathworks.de/products/matlab

[27] S. Poretsky, J. Perser, S. Erramilli, and S. Khurana, "Terminology for Benchmarking Network-layer Traffic Control Mechanisms," RFC 4689 (Informational), Internet Engineering Task Force, Oct. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4689.txt

[28] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *Proceedings of the first workshop on Hot topics in software defined networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342444