Wireless Transport Network Emulator for SDN Applications Development

Alexandru Stancu*†, Martin Skorupski ‡, Alexandru Vulpe*, and Simona Halunga*
*POLITEHNICA University of Bucharest, Faculty of Electronics, Telecommunications and Information Technology
Telecommunications Department, Bucharest, Romania
†Ceragon Networks, Bucharest, Romania
‡highstreet technologies, Berlin, Germany
Corresponding author: alex.stancu@radio.pub.ro

***Abstract*—Software Defined Networking is a paradigm that emerged in the networking industry recently. This technology is not mature yet, but is gaining momentum, driven by the fact that traditional networks began to show their limitations. Many standardization activities are still ongoing, given the fact that there are many aspects of a network that SDN touches upon, and having the right tools to support these efforts is important. The Wireless Transport Emulator (WTE) was designed for supporting the standardization endeavors of the Wireless Transport Group, part of the Open Networking Foundation (ONF), a consortium that aims to promote SDN adoption through defining open standards. WTE uses different technologies in order to simulate a wireless transport network, consisting of emulated Network Elements, that implement a Microwave Information Model, TR-532. It consists of a NETCONF server implementation that advertises the TR-532 information model, and the link representations based on Open vSwitch (OVS) bridges and a Python framework that glues everything together. The tool is also extremely useful for SDN application developers that want to create applications using the aforementioned information model, because it eliminates their need of owning real, expensive, wireless transport devices in order to test the functionality that they are developing. It is being used at the moment for preparing the next Wireless Transport PoC.**

***Index Terms*—Software-Defined Netowrks, Wireless Transport, Open Networking Foundation**

I. Introduction

Computer networks have become, nowadays, complex and increasingly challenging from the configuration and setup point of view. Therefore, the need for key architectural changes to the paradigm of networking has risen. Software-Defined Networking (SDN) emerged around the year 2009, from the work that was done in Stanford University in the context of the OpenFlow project. It is a revolutionary approach in networking, which focuses on mitigating the limitations proven by traditional networks. The concepts proposed by this paradigm are not new, some being even 25 years old, but the timing was not right at the time, thus their adoption in the industry was not possible then.

SDN proposes a novel network architecture, where the forwarding state of the data plane is managed by a distant control plane, decoupled from the data plane [1]. In this way, network devices become simple packet forwarding devices, while the control logic or the control plane is implemented in what is called the controller. This has numerous advantages, from being able to much more easily introduce new policies in the network through software, to being able to centrally configure all network devices instead of configuring individually each one. This way SDN can provide enhanced mechanisms for network management and configuration.

SDN can be used for optimizing the radio (e.g. remote radio units - RRUs and baseband units - BBUs) and transport (e.g. optical cross connects, microwave links) resources in future 5G systems. These resources can be managed by centralized controllers, on top of which an orchestrator may be placed. Therefore the SDN orchestrator has to be exposed to an adequately detailed abstraction of these resources.

Wireless Transport Group is part of the Open Networking Foundation (ONF) and is focuses on the development of a microwave information model that would abstract the characteristics of any wireless transport device. Several Proofs of Concept (PoCs) were conducted by the group ([11], [12] and [13]), where the model was tested and several use-cases that prove the utility of the model were implemented successfully. This led to the emergence of the first version of the Microwave Information Model, which is a technical recommendation by ONF, called TR-532 [10]. The main author contributed to both the TR-532 and the PoCs.

The main contribution of the paper is the design of a Wireless Transport Emulator (WTE). WTE uses different technologies in order to simulate a wireless transport network, consisting of emulated Network Elements, that implement a Microwave Information Model, TR-532. This tool is extremely useful for SDN application developers that want to create applications using the aforementioned information model, because it eliminates their need of owning real, expensive, wireless transport devices in order to test the functionality that they are developing.

This paper is organized as follows: section II makes an overview of some tools that relate to WTE, but are used in other types of networks, section III defines the architecture of the emulator, section IV provides high level details about the implementation and the technologies used and, finally, section V concludes the paper.

II. Related work

Development and testing of network applications or protocols can be done using different approaches. The first one, but

also the most expensive, is to use an experimental testbed. This consists of a small network consisting of real equipment to be used for the testing purposes. Several testbeds exist: Emulab [2], 100G SDN Testbed, provided by EsNet, as pointed out in [3] and [6], GENI [4], etc. As stated earlier, the main drawback of this approach is that building such a network is expensive.

The second approach for testing network applications and protocols is network simulation. This approach is usually simple and easy to use and can be used on a laptop or personal computer. It is flexible and scalable, the operations of real devices and interaction between them being modeled and run in a software program. The main drawback in this case is the fidelity and the replicability of the results in the same simulation conditions.

The third approach is the network emulation. It differs from the simulation approach through the real network applications or real-time operating systems that are used inside the emulation environment. As opposed to simulations, where the experiments are either faster or slower than real-time, emulations are executed in real-time.

Not many network simulation or emulation tools exist in the context of SDN. The most notorious and widely used software-defined network emulator is Mininet [5], [7]. It has the ability to emulate hosts, OpenFlow switches and links between them. It is also able to use its own SDN controller or to connect to a remote one. It is easy to use and has a Python API that be utilized in order to customize a network. Its main focus is the OpenFlow protocol and it does not support other southbound protocols (e.g. NETCONF).

Another network simulator that can be used in the context of SDN is ns-3 [8]. It provides support only for the OpenFlow southbound protocol, but, as stated in [9], it is limited to an old OpenFlow version and not developed anymore, because of the need to implement an SDN controller inside the environment, instead of being able to work with an external one.

Another available tool for software-defined networks emulation is EstiNet [9]. It is also based, as the previous two, on OpenFlow as the single supported southbound protocol. It is mainly used for SDN application performance testing, through its ability to provide such performance results in a correct, accurate and repeatable manner.

All of the above tools, though, provide only OpenFlow as a southbound interface. No tools that emulate networks in the context of SDN and provide a NETCONF southbound interface exist yet. This is probably because SDN is not a mature field yet and the standardization process is still ongoing. Only recently, YANG information models that represent network devices have emerged. For example, the YANG model used in the Wireless Transport Network Emulator was just released end of December, by the Wireless Transport Group, as TR-532. This is the main pillar of the WTE, as it allows SDN application development based on TR-532, without the need of owning real and expensive wireless transport equipment.

III. Architecture

The Wireless Transport Emulator is designed to run on a single Linux host and emulate there a specific topology described in a file, in a JSON format, which is given as a parameter when starting the WTE. Its architecture contains several components that rely on different tools. Each Network Element (NE) is simulated as a NETCONF server, which exposes the TR-532 information model, representing a wireless transport device.

The NETCONF server implementation is based on the OpenYuma framework and it has its roots in a tool that was used in the Proofs of Concept that were conducted by the Wireless Transport Group, called the Default Values Mediator (DVM), which is described in [14]. To achieve isolation for each of the simulated device, the NETCONF server is ran inside a *docker* container. It includes an Ubuntu Lunix image and the OpenYuma application that represents the server.

Each NE has a management interface on which the NETCONF server listens, and, for making this possible, each docker container is ran inside a *docker network*. This solution provides a full network isolation between the simulated devices and makes the docker containers available from the inside or from outside the host only through their management interface.

Another tool used by the WTE is *Open vSwitch* (OVS). For emulating a microwave link between two NEs, an OVS bridge is created in the host Linux machine and inside each container a Linux interface is created and attached to that bridge. This ensures a layer 2 connectivity between the two docker containers, with the help of the bridge.

Everything is glued together through a Python framework, which parses the configuration files needed (a topology JSON file and a configuration JSON file, which will be detailed later) and manages the instantiation of the docker networks and docker containers needed by the WTE. A high level overview of the architecture, detailed for a simple line network topology containing three network elements, is presented in Figure **??**.

The Python framework that is responsible for the implementation of the infrastructure needed by the emulator is very flexible and modular. It is implemented in an object oriented way, having classes for each of the important aspects that are needed: the emulator framework, network elements, interfaces, links, topology etc. This offers the possibility of extending it, for example, with a different NETCONF server implementation, by creating a NE from a different docker container.

The WTE also offers the possibility for each created NE to register itself automatically to an OpenDaylight (ODL) SDN Controller. This is an ODL specific implementation and it is done by issuing an HTTP POST request with specific parameters. Because it is implemented as a separate module, it can be easily extended to accommodate other controllers as well. The details about the SDN controller, like IP address, port and authentication details are part of the configuration JSON file. The controlled can reside on a remote machine, the only important thing to note here is that the host Linux machine

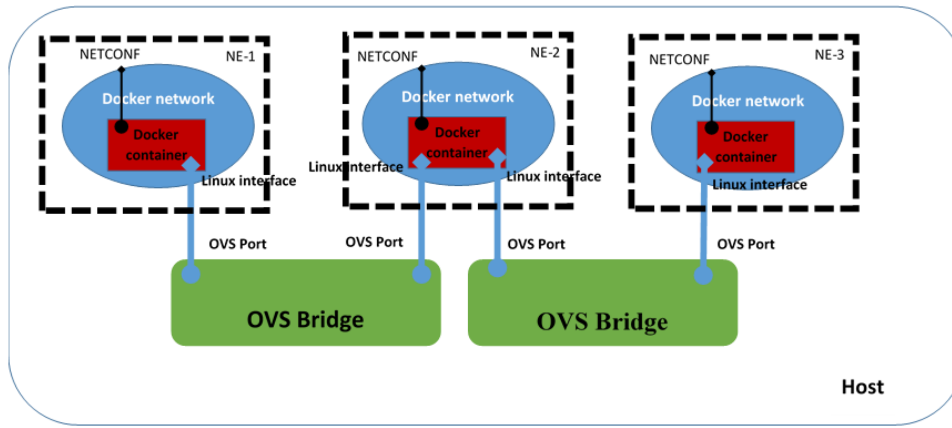

Fig. 1. High-level architecture of the WTE.

and also the docker containers need to have connectivity to that machine, meaning that they must be part of the same layer 3 network. More details about this connectivity will be given in the next section.

IV. Implementation

The implementation of the WTE is a combination of C and Python code and can be divided into several parts that together offer the functionality previously described: the Python framework that glues everything together, the NETCONF server implementation that advertises the TR-532 information model, which is actually a docker image used to represent an NE and the link representations based on OVS bridges. These will be detailed in the next subsections. The code is open-source and can be found in the GitHub [17].

A. NETCONF Server

The NETCONF server used in the current version of the WTE is a C implementation based on the OpenYuma framework. This framework was chosen for its advantages over other open-source frameworks, as described in [15]. The starting point for the development of this server is represented by the YANG files of the ONF TR-532, which contain the information model that describes a wireless transport device. Besides that, the microwave model refers also to the Core Information Model, ONF TR-512 ([16]), which is also supported, in part, by the server.

Using a tool provided by the OpenYuma framework, C code is automatically generated from the YANG files. This code generator has been altered in order to fulfill our needs. In its base implementation, it generates stub callbacks for each of the attributes defined in the YANG model, and then the user needs to implement these callbacks in order to get the desired functionality. In our case, we modified the code generator so that it automatically fills the stub callbacks with a generic function that is used to read the values of the attributes from an XML file. The format of the XML file is based on the tree representation of the YANG model and it only contains the status attributes (read-only). This approach provides flexibility in the sense that the status information provided by the NETCONF server is easily customizable, without the need of altering the C implementation and eliminating the need of code recompilation. By modifying the XML file provided to each server implementation, one can conveniently provide different status information for different simulated network elements.

The approach is somewhat similar also in the case of configuration attributes. We use the *startup* datastore capability of a NETCONF server and provide an XML startup configuration file when the server is started, so that it will contain details about the rad-write attributes that are present in the core and microwave models. This file is also customized for each simulated NE, such that every server instance will have its own Network Element Universally unique identifier (UUID), number of air-interfaces and their details or any other needed parameter. The values for these attributes are provided by the user in the JSON topology file needed by the emulator when it starts. As stated before, this approach offers resilience to the emulator.

Another important feature of the C implementation of the server is the ability to generate NETCONF notifications. This represents a big advantage, because the SDN application developers can develop and test applications that receive alarms from the simulated devices and they can also be informed by the NEs when a configuration attribute is changed. There are two approaches used for notification generation. The first one is, again, altering the C code generator so that in the callbacks generated for configuration attributes, a function that triggers a NETCONF notification is added. Every time the value for such an attribute is changed, this callback function is invoked and, from it, the call to the notification generation is done, using the current timestamp of the system, the name of the attribute that is being changed and its new value. This helps every SDN application that is subscribed to NETCONF notifications to be informed about configuration changes. The other approach used for NETCONF notifications is randomly triggering a problem notification. In the microwave information model, each air-interface has a *supported-alarms* attribute, which contains comma separated values of alarm names that are

supported by that interface of the device. The model also defines a minimum number of six alarms to be defined. In the WTE, the liberty of defining alarm names for those interfaces is given to the user, through the topology JSON file. The server implementation has the ability to randomly generate a problem notification, containing the alarm name from the user defined values, the current timestamp of the system and the severity of the alarm. The server stores internally an array of those alarms, so if an alarm was already raised previously, the server will not send it the second time, but will clear it. The time period between two consecutive NETCONF notifications, in this case, is provided by the user, through the JSON configuration file that the emulator takes as a parameter when starting. If that period is zero, the server will not send problem notifications.

A few other tweaks were implemented in the NETCONF server. For example, since an air-interface present in the microwave model is represented as a Linux interface in the docker container where the server runs, the status parameter *link-is-up*, which is *true* in case the communication is established with the remote side is read directly from Linux, using the *ip* tool. Also, the configuration parameter *power-is-on* directly alters the interface configuration (*up* or *down*) of the corresponding Linux interface.

Everything is packed inside a docker image. This docker image runs in every docker container that represents a network element and provides isolation (e.g. in terms of the XML configuration files used by the server) from other simulated devices. The docker container associates a port from the host Linux machine to a port inside the container (e.g. port 8300 from the host is associated with port 830 - standard NETCONF port - from inside the docker image, and port 2200 from the host is associated with port 22 - standard SSH port, needed for connecting to the NETCONF server). The SDN controller can then use these host ports and the management IP address of the NE for its NETCONF connectivity.

B. Links using Open vSwitch

For being able to emulate a complete wireless transport network topology, thus being able to pass traffic between the docker containers, links between such containers are also modeled. The information about the connectivity between network elements is provided by the user through the topology JSON file. A connection is represented by a JSON object containing the two connection points of a link. The connection point is defined by the NE UUID and the air-interface UUID. Having only the air-interface UUID would have sufficed, since this ID is by definition unique, but this is not enforced anywhere in the emulator framework, so it was chosen to have both attributes present in the topology JSON file for defining one side of a microwave link.

For each link defined in the topology JSON file, an Open vSwitch bridge, having a unique name, is created in the host Linux machine. Afterwards, the *ovs-docker* utility is used for both sides of the link. This is adding a connection between the docker container and the OVS bridge. After both sides of the link are added, the two docker containers have layer 2 connectivity through the OVS bridge. Optionally, an IP address can be assigned to each of the interfaces from inside the docker containers, and thus a layer 3 connectivity can be achieved between the two emulated devices, but this is not needed.

C. Python framework

The WTE contains a framework implemented to put all of the pieces together. This pieces are represented by the docker containers that run the docker image of the NETCONF server, which depict the wireless transport devices, and the links between them, modeled by connections through an OVS bridge. The framework has a similar approach as the one used in *mininet*, containing several classes representing objects in the emulation environment, such as: the emulation environment object, the network element object, the interface object, the link object etc. The workflow is depicted in a brief sequence diagram in Figure 2.

The emulation environment object contains the details about the configuration files needed by the WTE: the topology JSON file (*topology.json*, which has a specific format and describes the network elements with all their details (UUID, a list of air-interfaces along with all their details - UUID, supported alarms etc.) and the links between them. Several layers for these interfaces exist in the Core Information Model, and they can be represented in the JSON file and have a correspondence in the WTE, but we will not detail all of them. We will focus only on the MWPS (Microwave Physical Section), which depicts a physical radio port of a network element. Every information that is NE specific and is desired to be unique for each device will be added here in the next releases. The second file which is needed by the WTE is *config.json*, a JSON file containing several configuration parameters of the emulator, such as the details about the SDN controller, the period notification timeout, the network address to be used for the management IP addresses of the emulated devices and a boolean that describes if the NEs should automatically try to register to the SDN controller. Every attribute that is global and needed by the emulation environment itself will be added here in future releases. The next parameters needed by the WTE are the pointers to the skeleton XML files needed by the NETCONF server: the XML file containing the configuration attributes defined in the YANG models, that will be altered by the framework for each simulated device and then used by the server as a startup datastore and the status XML file which contains the values to be read by the server for the read-only attributes. Also, the emulator is required to have the YANG files of TR-532 and TR-512, so that they can be added to the docker container in order for the NETCONF server to advertise them. The emulator environment can also take a *--clean* parameter. When this is given by the user, the emulator only tries to clean the host Linux machine from any remaining old junk objects (suck as docker containers, docker networks or OVS bridges) that might have remained from a previous emulator run.

The network element object depicts a simulated wireless transport device. It is a class with many attributes and methods.

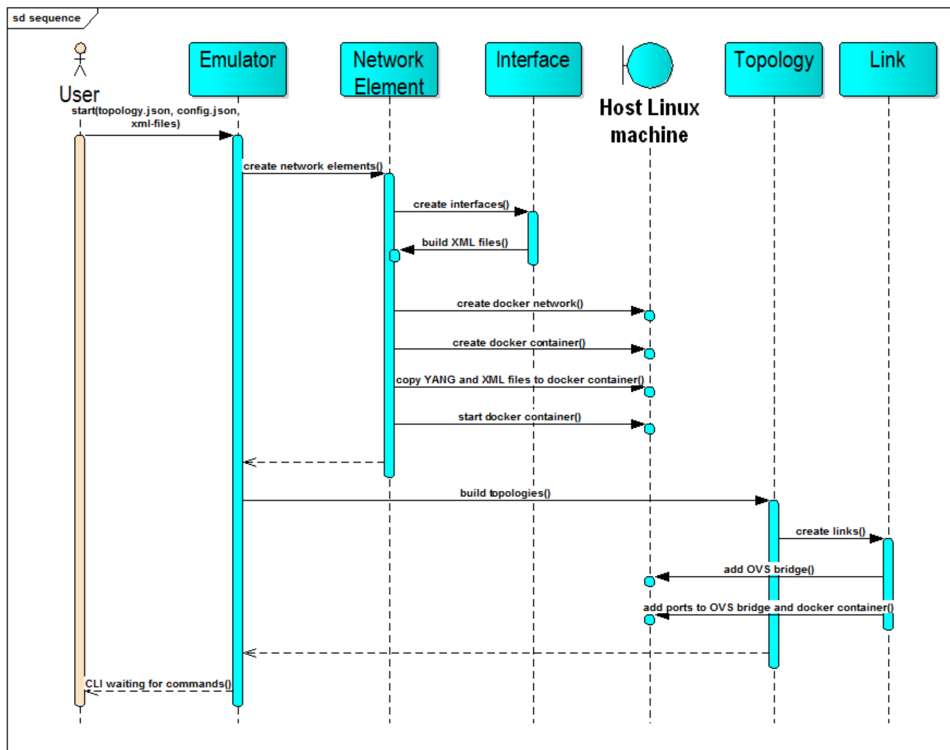

Fig. 2. Sequence diagram for starting the WTE.

One of the most important attributes of the NE is the management IP address. As stated before, the user has the liberty of specifying the network address of the devices. A management IP factory exists, that slices the network address given by the user into smaller /30 subnets. This is needed in order to ensure the separation of docker networks. Each NE is part of a docker network. A /30 netmask was chosen for the docker networks because it wastes the smallest number of IP addresses from the user specified network. The /30 netmask will provides two host addresses: one will be part of the host Linux machine and the other one will be part of the docker container itself. Supposing that the user specified the management network 192.168.0.0/16, the first emulated NE will have the docker network 192.168.0.0/30, with the IP 192.168.0.1 residing in the host machine and the IP 192.168.0.2 being part of the first docker container that is started inside that network. The second emulated device will have the docker network 192.168.0.4/30, with the IP 192.168.0.5 in the host machine and the IP 192.168.0.6 in its corresponding docker container. This will ensure that the NETCONF servers are reachable from the host machine via the IP addresses 192.168.0.1 and respectively 192.168.0.5. From an SDN controller point of view, it will see two different NEs, one accessible via NETCONF at 192.168.0.1:8300 and the other at 192.168.0.5:8300, but the docker containers will not have connectivity on the management interface, as desired. The important thing to note here is that the SDN controller must have connectivity to any of the management IP addresses of the devices, meaning it

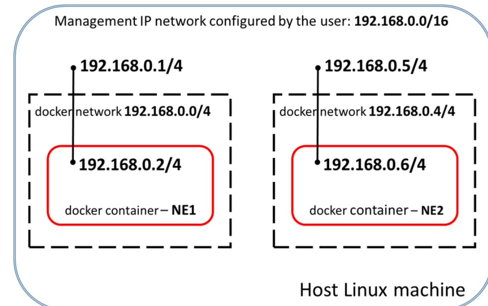

Fig. 3. Example management IP address allocation for two emulated NEs.

needs to be part of the 192.168.0.0/16 network, or even part of a supernet of that network address, so that its IP address will not conflict with any of the devices management IP addresses, given any number of NEs to be simulated. A simple figure depicting two devices along with their associated IP addresses can be seen in Figure 3.

The network element object is also responsible for altering the skeleton XML files used by the NETCONF server, with its specific values that are defined in the topology JSON file. It actually parses those XML files and finds the relevant nodes there, modifying their value with whatever it needs. It is also in charge of creating the list of interfaces it finds in the topology JSON file associated with its UUID. After that, it creates its docker container based on the common image that was built prior to starting the emulation, then it copies the

XML and the YANG files needed by the NETCONF server and it starts the docker container. When the container is started, the NETCONF server boots and waits for connection on its specific port.

The interface object is responsible for maintaining the details about its parameters. For example, as stated previously, it contains a *supported-alarms* attribute, which needs to have a minimum of six alarms defined, as enforced by the YANG model. It has a reference to its parent network element and uses it to alter the XML configuration and status files, which are part of the NE object. Altering means that the interface is responsible for adding a new XML node containing the details about that specific interface in the XML files. This object does not do any modifications inside the docker container itself, it only changes the XML files. The NE object is responsible for making the necessary configuration inside the docker container, after it starts it, based on its interface list.

After the docker containers representing the network elements are started, the emulator environment goes to the next step, which is the topology creation. It creates a topology object which is responsible for parsing the *links* object from the topology JSON file. This approach was chosen because in the topology can be multiple layers as well: the MWPS, which represents the connections between air-interfaces and another layer, ETH, which can represent connections between two Ethernet ports from the simulated NEs. We will not detail the ETH layer here, but relies on the same principles as the MWPS. For each link found in the topology JSON file, the topology object will create a link object, which is responsible for maintaining the details about the link. It verifies that the link connection points are valid and the corresponding interface objects are already created. If this is valid, then it proceeds to the actual emulation of the link: it creates an OVS bridge for that specific link and it connects the two interfaces into that bridge, ensuring layer 2 connectivity between the two docker containers.

V. Conclusion

SDN is a paradigm that is gaining momentum in the networking industry, driven by organizations that focus on accelerating its adoption through development of open standards and encourage open-source software platforms. The remote programming of the forwarding plane is encouraged through protocols like OpenFlow and NETCONF, having well defined interfaces and information models that abstract the underlying network.

Having the right tools to test the developed information models or the SDN applications that are based on those models is a key enabler for maintaining the momentum and accelerate the adoption of SDN in the industry. *Mininet* is an important tool for emulating networks that support the OpenFlow protocol, but the industry was lacking an emulator that would support also the NETCONF protocol. WTE tries to close that gap by proposing an emulating framework that exposes a NETCONF interface. It is focusing for the moment on the wireless transport devices, implementing the YANG models of ONF TR-532 and TR-512, but because of its modularity and flexibility it can be extended to accommodate any YANG information model or any other NETCONF server implementations.

There are many aspects in which WTE may be improved. For example, the links between two docker containers could be modeled through Linux *veth pairs*, instead of creating an OVS bridge between the containers. This will solve also an issue that can arise in the current implementation: if one side of the link is disabled (Linux interface is configured to *down*), the other side would not be affected, since it is another port in the OVS bridge. By using veth pairs, disabling one side of the link would be automatically seen in the remote side as well.

References

[1] A. Stancu, S. Halunga, G. Suciu, and A. Vulpe, “An Overview Study of Software Defined Networking”, Informatics in Economy (IE 2015), 2015 14th International Conference on, 2015, pp. 50-55.

[2] B. White, et. al. “An integrated experimental environment for distributed systems and networks”, ACM SIGOPS Operating Systems Review, 2002, vol. 36, number SI, pp. 255-270.

[3] K. Roberts, et. al. “Beyond 100 Gb/s: capacity, flexibility, and network optimization [Invited]”, Journal of Optical Communications and Networking, 2017, vol. 9, number 4, pp. C12-C24.

[4] M. Berman, J.S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, I. Seskar “GENI: A federated testbed for innovative network experiments”, Computer Networks, 2014, vol. 61, pp. 5-23.

[5] Brandon Heller “Reproducible Network Research with High-Fidelity Emulation”, PhD Thesis, Stanford University, 2013.

[6] 100G SDN Testbed [Online] Available athttps://www.es.net/network-r-and-d/experimental-network-testbeds/100g-sdn-testbed/.

[7] Bob Lantz and Brian OConnor“A Mininet-based Virtual Testbed for Distributed SDN Development”, ACM SIGCOMM Computer Communication Review, 2015, vol. 45, number 5, pp. 365-366.

[8] George F. Riley and Thomas R. Henderson“The ns-3 Network Simulator”, Springer Berlin Heidelberg, 2010, pp. 15-34.

[9] Shie-Yuan Wang, Chih-Liang Chou and Chun-Ming Yang“EstiNet openflow network simulator and emulator”, IEEE Communications Magazine, 2013, vol. 51, number 9, pp. 110-117.

[10] ONF TR-532 - “Microwave Information Model”, Version 1.0 [Online] Available athttps://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR-532-Microwave-Information-Model-V1.pdf.

[11] Wireless Transport SDN Proof of Concept White Paper, 2015 [Online] Available athttps://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/ONF_Microwave_SDN_PoC_White_Paper%20v1.0.pdf.

[12] Wireless Transport SDN Proof of Concept 2 Detailed Report, 2016 [Online] Available athttps://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/Wireless_Transport_SDN_PoC_White_Paper.pdf.

[13] Third Wireless Transport SDN Proof of Concept White Paper, 2016 [Online] Available athttps://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/Third-Wireless-Transport-SDN-Proof-of-Concept-White-Paper.pdf.

[14] A. Stancu, A. Vulpe, O. Fratu, S. Halunga“Default values mediator used for a wireless transport SDN Proof of Concept”, 2016 IEEE Conference on Standards for Communications and Networking (CSCN).

[15] A. Stancu, A. Vulpe, G. Suciu, E. Popovici“Comparison between several open source Network Configuration protocol Server implementations”, 2016 International Conference on Communications (COMM).

[16] ONF TR-512 - “Core Information Model”, Version 1.1 [Online] Available athttps://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/ONF-CIM_Core_Model_base_document_1.1.pdf.

[17] Wireless Transport Emulator GitHub [Online] Available athttps://github.com/Melacon/WirelessTransportEmulator.