Received January 11, 2018, accepted February 27, 2018, date of publication March 14, 2018, date of current version April 18, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2815844

Evaluation of a Wireless Transport Network Emulator Used for SDN Applications Development

ALEXANDRU STANCU[1,2], ALEXANDRU VULPE[1], (Member, IEEE), AND SIMONA HALUNGA[1], (Member, IEEE)

[1]Telecommunications Department, Faculty of Electronics, Telecommunications and Information Technology, Politehnica University of Bucharest, Bucharest, Romania

[2]Ceragon Networks, Bucharest, Romania

Corresponding author: Alexandru Stancu (alex.stancu@radio.pub.ro)

This work was supported in part by the Ministry of Innovation and Research, UEFISCDI, Project 5 Sol/2017 within PNCDI III, Integrated Software Platform for Mobile Malware Analysis (ToR-SIM), in part by OFDM System based on FFT with non-integer argument (FractOFDM) under Contract 213PED/2017, and in part by MultiMonD2 under Grant PN-III-P1-1.2-PCCDI-2017-0637.

ABSTRACT Software Defined Networking (SDN) is a paradigm that emerged in the networking industry recently. Many standardization activities are still ongoing and having the right tools to support these efforts is important. The Wireless Transport Emulator (WTE) was designed and implemented for supporting the standardization endeavors of the Wireless Transport Project, part of the Open Networking Foundation (ONF). WTE uses different technologies in order to simulate a wireless transport network, consisting of emulated Network Elements, that expose two information models proposed by ONF: the Microwave Information Model, TR-532, and the Core Information Model, TR-512. The tool is also extremely useful for SDN application developers that want to create applications using the aforementioned information models, because it eliminates their need of owning real, expensive, wireless transport devices in order to test the functionality that they are developing. This paper describes the architecture of the WTE and then evaluates the simulator with regards to some characteristics. Based on the measurements, conclusions about the capabilities of the simulator are drawn.

INDEX TERMS Software-defined networking, 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 [1]. 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 that time, thus their adoption in the industry was not possible until now [2].

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 [3], [4]. 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 SDN controller. This has numerous advantages, from being able to much more easily introduce new policies in the network through software, to the ability of central configuration of all network devices, instead of individual configuration. In this way, SDN can provide enhanced mechanisms for network management and configuration.

SDN can also 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 [5], [6]. 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 [7], [8]. As expressed in [9]–[12], significant attention from the research community is given to advancing SDN in all aspects of a network and in different network types, such as wireless sensor, satellite, optical or vehicular networks.

Wireless Transport Group is part of the Open Networking Foundation (ONF) and is focused 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 ([13]–[16]), 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 [17]. The main author contributed to both the TR-532 and the PoCs, as expressed in the Contributions section in [17].

The main contribution of the paper is the development and implementation of a Wireless Transport Emulator (WTE) and then its evaluation, with regards to several characteristics. WTE uses different technologies in order to simulate a wireless transport network, consisting of emulated Network Elements, that expose the Microwave Information Model, TR-532, and the Core Information Model, TR-512 [18]. 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. WTE could be used also by operators for testing SDN applications or interactions between such applications, prior to deploying them to production networks. Because of its flexibility and modularity, the simulator could be extended to accommodate other information models. Its utility was already validated in the 4th WT SDN PoC organized by ONF, as described in [16], where it was successfully used for preparing the PoC and can still be used for demonstrating the proposed use cases, even though the PoC has ended, being installed in the ORBIT environment [19]. It is called the ***Wireless*** Transport Emulator because it can be used to expose the information models proposed by ONF (especially the Microwave Information Model). The only characteristic of the wireless medium that is being emulated at the moment is the bandwidth of a link, which is influenced by the channel bandwidth and the modulation technique used (attributes that can be modified through the SDN controller). The WTE can be further enhanced to emulate other wireless medium characteristics as well, such as the influence of the frequencies on the connection, or the wireless signal strength, but it is not straightforward and is therefore not in the scope of this work.

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, then section V describes the methodology used for evaluating the implementation and presents the evaluation results. Section VI presents some advantages and disadvantages of the WTE, based on the measurements and a comparison with *mininet*. Finally, section VII 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 [20], 100G SDN Testbed, provided by EsNet, as pointed out in [21] and [22], GENI [23], [24], ORBIT [25], 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 network emulation. It differs from the simulation approach through the real network applications or real-time operating systems (OS) 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* [26], [27]. 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 ([28]), and it does not support other southbound protocols (e.g. NETCONF [29]).

Another network simulator that can be used in the context of SDN is ns-3 [30]. It provides support only for the OpenFlow southbound protocol, but, as stated in [31], 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. This simulator can be transformed in an emulator by utilizing it in combination with virtualization tools like QEMU, such that a NETCONF interface could also be provided. The main drawback of this approach is that the user is not allowed to utilize a specific proprietary YANG model for his emulated device.

Another available tool for software-defined networks emulation is EstiNet [31]. 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 mainly OpenFlow as a southbound interface. There are only a few tools that emulate networks in the context of SDN and provide a NETCONF southbound interface, but none of them allow the user to expose its own YANG models. One possible cause can be that SDN is not a mature field yet and the standardization process is still ongoing. SDN focused until recently on altering the flow tables from network devices. This implied using protocols like OpenFlow, which is specifically tailored for this matter. NETCONF, on the other side, can be compared with protocols like SNMP (Simple Network Management Protocol), being designed mostly for configuring the attributes of network elements.

The importance and popularity of the model driven programmability and management of networks, achieved through the NETCONF protocol, is increasing recently, as expressed also by Medved *et al.* [32] . This is reflected also in the research work done in many Standards Developing Organizations (e.g. ONF, IETF, ETSI, etc.), which focuses lately on developing data models in this context.

Only recently, Yang [33] information models that represent network devices have emerged and can be used by NETCONF. For example, the YANG model used in the Wireless Transport Network Emulator was just released end of December 2016, by the Wireless Transport project, as TR-532. The Information Modeling project defined the Core Information Model (TR-512) in March 2015. These two models represent the main pillars of the WTE, and it allows SDN application development based on them, without the need of owning real and expensive wireless transport equipment. The YANG model representing TR-512 contains information about the network element itself, it defines its interfaces, internal cross-connections, details about the equipment (cards, serial numbers, etc.). The YANG model associated with TR-532 contains attributes that provide details about the interfaces of the NE, such as: capabilities, configuration, status, current problems, current and historical performance values.

The authors in [34] depict the main concepts behind SDN in transport networks. These are based on architectural models of network elements and transport networks themselves. Because such networks are large and complex, having multiple components, it is of great importance to define a network model that covers all the aspects, is technology agnostic and comprises functional entities, for being able to design and control them. The transport networks representation is simplified by layering it into a number of independent transport layer networks. A client-server relationship exists between such layer networks, the client being the signal being carried, and the server being the layer network providing its transport. Also for simplification purposes, the networks can be partitioned into smaller disjoint subnetworks, that are interconnected by links. The transport networks can be divided into several components: topological components, transport processing functions and transport entities. The Core Information Model (TR-512), proposed by the Information Model project, defined such a model, in which topological components and transport processing functions are modeled using the ForwardingDomain and LogicalTerminationPoint objects, while transport entities are represented using the ForwardingConstruct object class. The Core Information Model can be extended with fragments for specific technologies, which is exactly the case of the wireless transport networks, extended through the Microwave Information Model (TR-532).

It was chosen not to extend the existing *mininet* implementation with a NETCONF interface, because of the lack of flexibility that it provides when defining the topology. WTE allows the user to define the topology to be simulated in a simple JSON file, having a fixed format. With this approach, one can easily define the network elements to be emulated, along with interfaces for each, at different layers (as defined by the Core Information Model) and links between them. The two tools are complementary, *mininet* focusing on the OpenFlow protocol, and WTE using NETCONF. Nevertheless, they could be used in parallel and they could even be linked (since they are both using *veth* pairs for representing links between elements) such that, for example, traffic between two *mininet* switches can pass through a wireless transport network emulated with WTE.

III. ARCHITECTURE

The Wireless Transport Emulator was designed for simulating, on a single Linux host, a wireless transport network topology, using different tools, while exposing the information models developed by ONF (TR-532 and TR-512). WTE is relying on a configuration file which specifies the topology that the user wants to simulate. This file has a fixed JSON format, which was defined according to the transport layers of the Logical Termination Point (LTP) objects defined in the Core Information Model. The format will be detailed later on in the paper.

Several tools are used in the architecture of WTE, that add up to the simulator solution. Each Network Element (NE) is simulated through a docker container, which runs a Linux image along with the NETCONF server, represented by the Default Values Mediator (DVM) that was previously used in WT SDN PoCs organized by ONF [14], [15] and detailed by Stancu *et al.* [35]. This approach was chosen in order to achieve an isolation at the file system level, so that multiple DVM instances can run without issues on the same machine. The interfaces that the user defines in the topology file are represented as Linux interfaces inside each docker container, and are used for creating the links between simulated devices. Every NE has a management interface which connects to the SDN controller. For obtaining isolation between these interfaces, so that the data traffic will not be forwarded through this interfaces, docker networks are used. All these elements form the architecture of the WTE. The usage of this tool in the SDN context is illustrated in Fig. 1.

Docker is a tool which allows creating software containers incorporating applications that can be run in an isolated environment with regards to the applications from the host operating system, which launches those containers [36]. An application and all its dependencies can be packed inside a docker container. The resource consumption of such a container is lower than that of a virtual machine, because the operating system is not replicated, but only the libraries and processes needed by the application being virtualized. As highlighted in [37], docker can be utilized for generating reproducible research activity. For these reasons, this tool is used also in

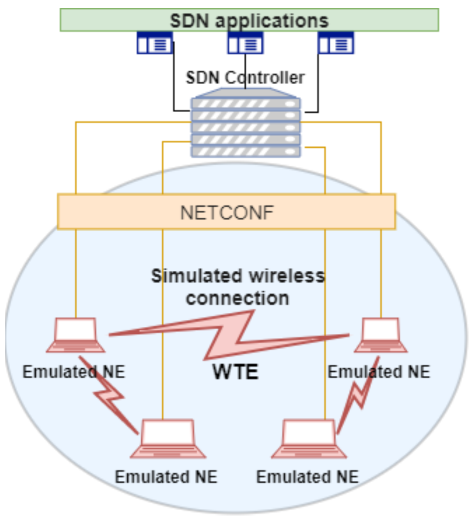


FIGURE 1. **Usage of the WTE in the SDN context.**

the case of WTE, for obtaining the isolation in the file system level, of the application that implements the NETCONF server which exposes the desired information models, DVM.

Docker networks represent a feature provided by docker which can isolate also the network stack associated with a docker image. Thus, different containers can be run inside separate docker networks, achieving network isolation for them. The user can choose the details of such docker networks, like the network address or network mask.

For representing the links between simulated NEs a simple method was chosen: virtual Ethernet pairs (*veth* pairs). This is actually a tunnel between two Linux interfaces: the traffic that is sent to one of the pair interfaces is forwarded to the second interface in the pair, and vice versa. This represents a frequently used method in working with containers [38].

From the design phase of the WTE a simple workflow emerged: when the simulator is initialized, it analyses the topology file. Then, it creates the docker networks associated to each network element defined in the topology and afterwards it starts the necessary docker containers. The next step is creating the Linux interfaces associated with different transport layers of the LTP objects, as defined in the topology file. The last step is creating links between those interfaces, according to what is described in the topology, and running the Command Line Interface (CLI).

The major components of the WTE are: the DVM, which was adapted so it can be run in the environment proposed by WTE, the JSON file containing the topology that we want to simulate, another JSON configuration file containing several parameters that detail how WTE should be configured and a Python framework that glues everything together and implements the logic inside the simulator. This last component is actually the core of the WTE.

The Python core of WTE is responsible for implementing the infrastructure that the simulator needs and it is designed to be modular and flexible. Is was developed using an object oriented manner, having classes for every important component: a general framework, network elements, interfaces, links, topologies, etc. This allows extending the WTE, for example, with some other NETCONF server implementation. Because the architecture of the WTE is highly flexible and modular, it can also be extended to accommodate other YANG information models.

IV. IMPLEMENTATION

The implementation of the WTE is based on Python code for the core and on C code for the NETCONF server. The previous DVM solution, that was developed for the WT SDN PoCs organized by ONF [14], [15], was adapted to comply with this new approach. The code is open-source and can be found on GitHub [39].

The implementation is presented gradually, in order to construct the full picture of the WTE. First, the topology and configuration files are presented. These files are parsed to determine the parameters of the emulation and the topology to be simulated. Then, we detail the component that implements the NETCONF server (DVM), which exposes the information models (TR-532 and TR-512). After that, the component that glues everything together is described: the Python framework, or the core of the WTE. Next, more details about the emulator are given: how the interfaces defined in the topology are represented, how the links between those interfaces are constructed, including how the bandwidth of a wireless connection is emulated, and finally how traffic can be passed through the simulated topology.

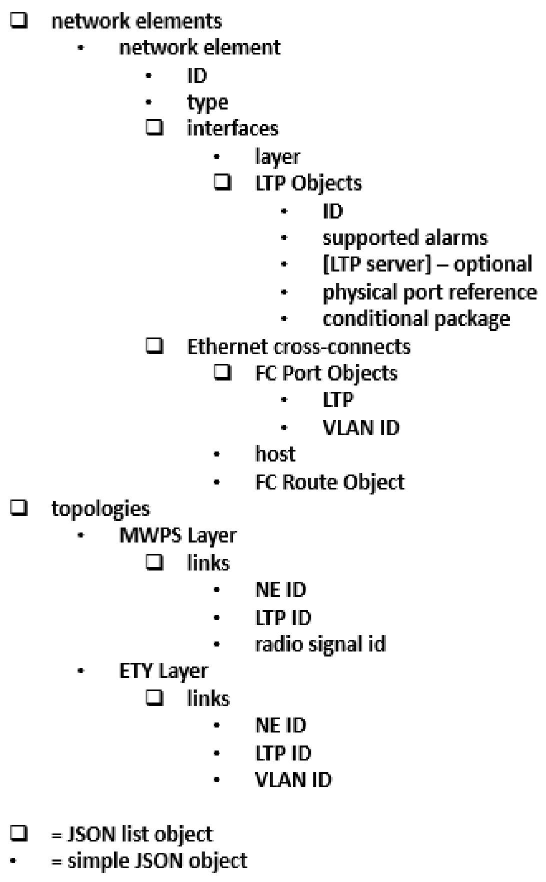
A. CONFIGURATION FILES

WTE uses configuration files having a fixed format for offering a simple interface to the users. Two JSON files exist: *topology.json*, that describes the topology that the user wants to simulate and *config.json*, which contains the configurable parameters of the simulator. The fixed structure of the topology file is highlighted in Fig. 2.

For simulating a topology, the required information is composed by the network elements, their interfaces and the links between them. Having this in mind, the topology file will contain a Network Elements JSON list object, which represents a collection of NEs. Each NE will have its interfaces, on different LTP layers defined, along with several details needed by the information models. In Wireless Transport networks, in the simplest case, we can have wireless or Ethernet links between the devices. In the Core and Microwave Information Models terminology, this translates to Microwave Physical Section (MWPS) layer, or the Ethernet Physical (ETY) layer respectively. This leads to another JSON list object in the topology file, which contains the links between interfaces, at each of these layers.

B. DVM FOR WTE

The Default Values Mediator represents the NETCONF server implementation that exposes the desired information models: TR-512 and TR-532. The DVM implementation that already existed was transformed into a docker image.



- ❑ network elements
  - network element
    - ID
    - type
    - ❑ interfaces
      - layer
      - ❑ LTP Objects
        - ID
        - supported alarms
        - [LTP server] – optional
        - physical port reference
        - conditional package
    - ❑ Ethernet cross-connects
      - ❑ FC Port Objects
        - LTP
        - VLAN ID
      - host
      - FC Route Object
- ❑ topologies
  - MWPS Layer
    - ❑ links
      - NE ID
      - LTP ID
      - radio signal id
  - ETY Layer
    - ❑ links
      - NE ID
      - LTP ID
      - VLAN ID

❑ = JSON list object
• = simple JSON object

FIGURE 2. Topology JSON file structure.

Because each docker container will represent a different NE, a method was required for distinguishing between the separate containers, from an SDN controller point of view, after the simulator is initialized. This was done by modifying the DVM to use the *startup datastore* capability provided by the NETCONF server. This means that each server can load its initial configuration from an XML file, at boot time. The consequence in the WTE implementation was that, based on the topology file, one XML file containing the details of that specific network element was created and copied inside the docker container for the NETCONF server to load when initializing. This ensured that each simulated device starts with a configuration as described in the topology file and the SDN controller that will communicate with it will see that configuration accordingly.

The DVM was capable of generating dummy NETCONF notifications, so that SDN application developers could test this channel of communication. This ability was improved for the WTE implementation. The previous implementation would read the details of an alarm from a configuration file and, at a specific interval, it would send that notification to whoever subscribed to this event. This mechanism was improved now significantly. The DVM which runs inside the docker container has now the ability to choose a random interface that will send an alarm, from the interfaces defined on that specific NE. Afterwards, still randomly, it will choose an alarm, from the *supported alarms* YANG attribute of that interface, and, according to its previous state (if the alarm was raised, it will be cleared, or vice versa), the notification will be sent. This provides SDN application developers a more realistic interaction with the simulated network from the alarms point of view. At a specific interval, a problem notification will be generated, based on the interfaces present in the network and their supported alarms list, so this communication channel can be tested.

C. WTE CORE

The core of the simulator is represented by the code that offers the infrastructure that, along with the other tools, provides the desired functionality: emulating a wireless transport network, while exposing through a NETCONF interface the Core and Microwave information models. It is implemented in Python, using an object oriented approach, allowing extensibility in a simple manner. This method is similar with the one used in mininet [26], [40]. There are two main reasons for which we chose not to extend the mininet solution, but to develop a new one: (i) mininet is based on the OpenFlow protocol, while we wanted a solution based on NETCONF, that could expose the Core and Microwave information models, and (ii) the topology specification - in mininet, this is done either through the predefined topologies available in the command line or using the provided Python API; these approaches were not flexible for the needs of WTE.

The class diagram of the WTE core is presented in Fig. 3. The purpose of the design was creating a modular and flexible architecture, that would allow readability of the code and an easy extensibility. This is the reason that several classes exist: the *NetworkElement* class is used for representing a wireless transport device. According to the topology file, several instances of this class will be created. Each will handle the creation of its associated docker container, XML file containing its configuration and its interfaces. Several classes for the interfaces exist, one for each of the layers present in the information models. The *Topology* class is used to represent the topologies defined in the configuration file, at the different layers: MWPS and ETY. The *Link* class represents a link between two interfaces.

A simplified sequence diagram that describes the initialization of the simulator is illustrated in Fig. 4. The user will issue a command for starting the simulator, giving the necessary JSON files as input. After that, the framework of the WTE will create the needed *NetworkElement* objects associated with the devices that the configuration file contains. The next step is creating the associated interfaces for each of the NEs, according to the described topology. Afterwards, the XML file associated with the device is created and then the docker networks, docker container and the Linux interfaces are created. The last point is building the topologies described by the user, which translates into creating the links between the docker containers. WTE will then wait for commands in a CLI.

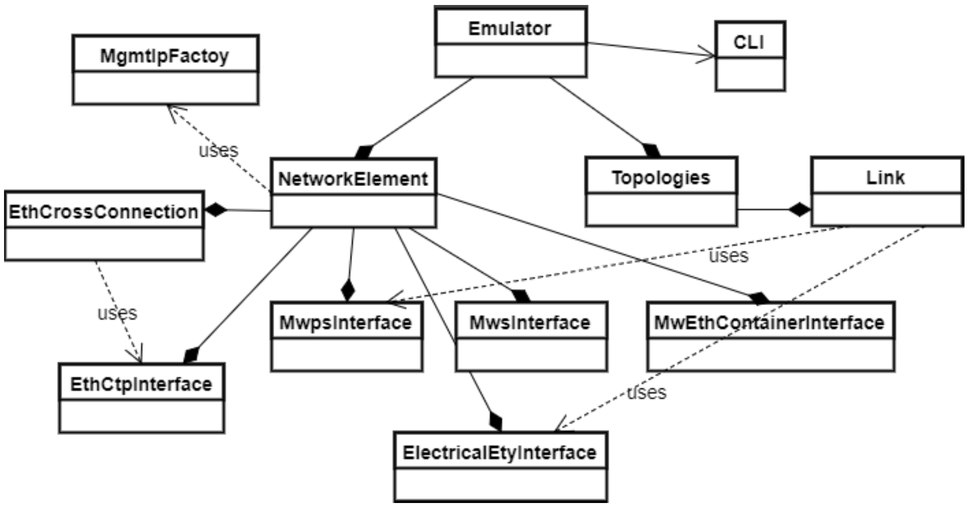


FIGURE 3. Class diagram of the WTE core.

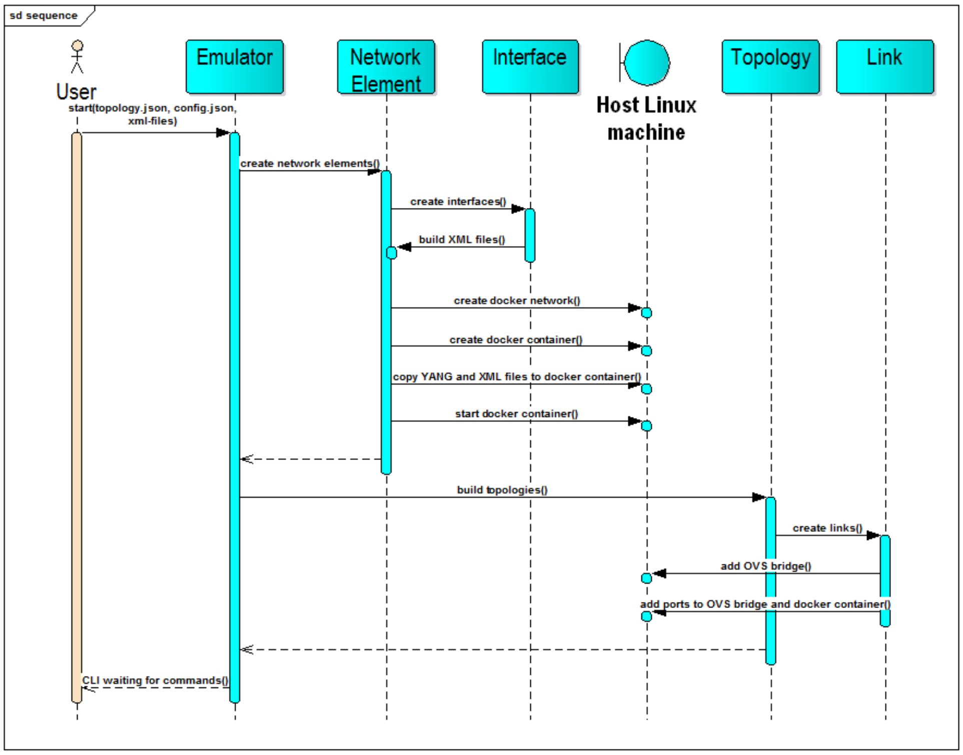

sd sequence
User
Emulator
Network Element
Interface
Host Linux machine
Topology
Link
start(topology.json, config.json, xml-files)
create network elements()
create interfaces()
build XML files()
create docker network()
create docker container()
copy YANG and XML files to docker container()
start docker container()
build topologies()
create links()
add OVS bridge()
add ports to OVS bridge and docker container()
CLI waiting for commands()

FIGURE 4. Simplified sequence diagram for the initialization of the WTE.

D. INTERFACE REPRESENTATION

Each interface of the NEs, or, in Core and Microwave Information Models terminology, the LTP objects associated with each device, regardless of their transport layer, will be represented as a Linux interface inside the docker container associated with the simulated network element.

All LTP objects present in the topology file can be added as Linux interfaces using the ***ip*** tool offered by Linux [41]. The MWPS or ETY objects that are not used in a link between two NEs are represented as *dummy* interfaces. If they are part of a link, they will be defined as an interface from the veth pair associated with that link. The LTP objects from the other transport layers will mandatorily have a client-server relationship, as defined in the Core Model. In order to represent this inside the docker container, *bond* interfaces are used. These represent logical points that can aggregate several interfaces, mapping perfectly with the client-server relationship defined in TR-512. It is the responsibility of the

WTE core to decide what type of Linux interface to create and how to make the connections with the others, according to the topology file.

E. LINKS BETWEEN NEs

The links between devices are represented, as stated previously, through virtual Ethernet pairs (veth pairs). Each link will basically be a tunnel between two interfaces that are part of separate docker containers, ensuring thus connectivity between them. With this approach, if one interface representing one end of the link is disabled (set administratively *down*), this will be reflected in the remote side, the link being *down*. This behavior is consistent with real networks.

In order to make the WTE a real emulator of wireless transport links, for the moment only the bandwidth characteristics of the connection are altered in such a manner that it will represent a link in the wireless medium. In the future, other characteristics of this medium could be implemented. As defined in TR-532 [17], the capacity of an interface is influenced by several parameters that are present in the Microwave Model, and can be seen in (1).

$$\begin{aligned} & txCapacity \\ & \quad = txChannelBandwidth \\ & \qquad * log_2(modulationCurrent) * codeRate * symbRate \end{aligned} \tag{1}$$

where:

- *txCapacity* is the capacity of the transmitter of the interface, in Mbps
- *txChannelBandwidth* is the bandwidth of the transmit channel, in kHz
- *modulationCurrent* is the current modulation used, in number of symbols
- *codeRate* is the percentage of the useful information from the transmitted data (typically between 85% and 95% [42])
- *symbRate* represents the typical symbol rate for wireless transport devices and has a typical value of 85% [42].

This formula is being used inside the DVM. When one of these attributes are changed, the new capacity of the respective interface is computed and the interface bandwidth is altered using the ***tc*** utility offered by Linux [43]. In this manner, the user can utilize SDN Applications from the controller and modify these values, and the capacity of the link will be automatically changed in the simulator. This was implemented and tested with the *iperf3* tool [44], [45], which is part of every docker container, allowing the injection of traffic in the simulated network.

F. INJECTING TRAFFIC IN THE SIMULATED NETWORK

For offering a complete simulation tool, including the validation of the topology and links that were created, WTE provides the possibility of injecting traffic in the network, between two points. For this ability, the tool *iperf3* is installed and used in each docker container.

All the connections that were described before are implemented at Layer 2. This means that the Linux interfaces inside the docker containers do not use IP addresses (besides the one used for management). The traffic generation is done between two interfaces, one being the server and the other being the client.

iperf3 allows generating TCP or UDP traffic, at the same time providing statistics about the packet latency, jitter, bandwidth of the channel used for communication or the percentage of lost packets.

V. EVALUATION AND RESULTS

For evaluating the WTE, four characteristics were considered for measuring: the processing power that the simulator uses, the RAM memory needed by the simulated topology, the initialization time and the disk space that the WTE uses. The methodology about the measurements and the results, as well as a comparison with mininet will be covered in this section. The intention of the measurements is not to give the exact precise values that the simulator requires, but to give some insight about what part of the WTE (docker containers, Linux interfaces, veth tunnels) influences the most the characteristics that we measure and affects its scalability.

A. METHODOLOGY

For allowing the aforementioned measurements, some solutions were developed in the simulator, in the WTE core. Measuring the boot time implied using a timer provided by Python. The measurement started after the configuration files were read, and stopped after all the elements in the topology were created, just before starting the CLI. This measured time represents the initialization duration of the simulator and is expressed in seconds.

A similar approach was used when measuring the disk space used by the WTE. Before the simulator starts the initialization phase, it queries the operating system about the free space available on disk. After the initialization is finished, a new interrogation is done to the operating system. The difference between these two values represents the disk space occupied by the WTE. We need to mention that when running the evaluation, no other applications besides the simulator were running in the systems (apart from the OS applications). The disk spaced occupied by the WTE is given by the need of the docker containers to represent their file systems inside the host operating system. The computed value of the disk space used by the simulator is expressed in MB.

The percentage of the used RAM is computed by the core of the WTE. In calculating it, it neglects some values (e.g. the memory needed by the core itself), because they are not significant with regards to other values. The neglected values are comparable with the others only when emulating small topologies (e.g. under 20 simulated interfaces), and do not increase with regards to the topology size, this being the reason they are not considered for measuring. The only memory allocations taken into account are represented by the ones associated with the docker containers that are created by the simulator. In the CLI of the WTE, a new command that computes this percentage was implemented. The computation uses the information reported by the command *docker stats*, provided by docker. This returns the RAM percentage utilized by each running docker container and was used because its results are accurate. The implementation of the CLI command will add these values and return the sum. Thus, the value returned will represent the RAM percentage used for simulating the topology.

The computing of the CPU percentage used by the WTE is triggered by the same command as for the RAM percentage. Its implementation uses also the *docker stats* command provided by docker, but the approach is slightly different. The difference appears because the CPU power is not used continuously by the docker containers associated with the simulated network elements. If, in the moment that we run the command to query the percentage of the CPU used by the containers, they did not have CPU time allocated, they could have zero percent usage. A subsequent query might reveal different results, other CPU percentages being allocated to the docker containers. In order to make the measured value relevant, a number of 10 queries were made with the *docker stats* command, and an average of the returned results was computed. Empirically we observed that if this number of samples was increased, for example to 100, the difference between the results was insignificant (around 10^{-3}). It must be noted that these values can fluctuate, because the docker engine can allocate CPU time to the docker containers according to their needs, and this is not deterministic, because the CPU usage can be slightly different for each emulated NE at the moment of the measurement.

The result returned by the *docker stats* command is with regards to a single CPU core. This means that, in the case of intense utilization, we could have had returned values larger that 100%. For this reason, the result is normalized to the number of CPU cores of the machine where the simulator runs. In a nutshell, the CPU percentage is computed with this method: the OS is queried about the CPU percentage used by the docker containers with the *docker stats* command, and the sum of these percentages is calculated. This operation is repeated 10 times, taking 10 samples, and the average is computed. The value obtained is divided by the number of CPU cores, resulting the final value.

The CPU usage and RAM percentage used were measured only after the simulated devices were connected to the SDN controller and the NETCONF connections were established in the network. All of the aforementioned characteristics were measured in different topologies that are typical for wireless transport networks: ring, tree or mesh.

For the ring networks, topologies having a number of devices from 10 to 200 were simulated, with a step size of 10 elements. In this situation, each device would need two air interfaces, for connecting to its neighbors. We chose to add two Ethernet interfaces to each device, for granting the ability of injecting traffic in the network. According to the Core and Microwave information models, this resulted in 8 LTP objects associated with each NE, so a total of 8 Linux interfaces in each docker container.

In the case of tree topologies, binary trees were considered, so each network element needed three air interfaces (one for the connection towards the root and two for the connections to the leafs). An Ethernet interface was added to each device, for providing the ability of injecting traffic in the network. In this scenario, every docker container associated with an NE will contain 10 Linux interfaces. In the simulations associated with this topology, the depth of the tree was varied from 3 (meaning a total of 7 devices) to 7 (meaning 127 simulated network elements).

For mesh topologies, the number of simulated network elements varied from 3 to 10. Full mesh topologies differ from the previous two cases with regards to the total number of simulated interfaces: it does not depend anymore in a linear manner on the number of NEs, but has a quadratic increase. For a full mesh network with N network elements, we simulated a total number of $N(N-1)$ interfaces, and $N(N-1)/2$ number of data links, respectively. Thus, for N elements, each simulated device had $N-1$ air interfaces. Also, an Ethernet interface was added to each device, for providing the ability of injecting traffic in the network. This resulted in varying the number of simulated interfaces from 6 to 90.

TABLE 1. System characteristics of the systems where the WTE evaluation was done.

Characteristic	*Local machine*	*ORBIT Cloud*	*DT Cloud*
OS	Linux	Linux	Linux
Kernel version	4.4.0-93-generic	4.4.0-83-generic	4.4.0-89-generic
CPU architecture	x86_64	x86_64	x86_64
RAM	4 GB	8 GB	8GB
Disk storage	32 GB	80 GB	80 GB
CPU frequency	2591.59 MHz	2 GHz	2500 MHz
CPU cores	4	4	4
Hypervisor	VirtualBox	VMware	VMware

B. EVALUATION ENVIRONMENTS

The measurements associated with the evaluation were done in three different environments where WTE was installed: locally, on a laptop having a Linux virtual machine (VM), and in two cloud environments.

The first cloud environment was ORBIT [19], [25], [46] and access to it was provided by AT&T. ORBIT is a wireless network testbed designed for the research community and used for achieving reproducible experiment results.

The second cloud environment was the one used in the 4th WT SDN PoC organized by ONF and was provided by Deutsche Telekom (DT) [16]. This private DT Cloud, located in Prague, offered virtual machines for the SDN controllers and for the simulator environment.

The characteristics and resources that those system have are described in Table 1.

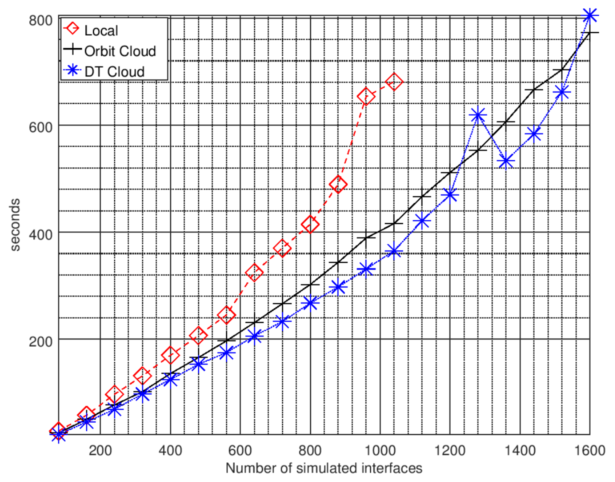


FIGURE 5. Boot time versus the number of simulated interfaces in a ring topology.

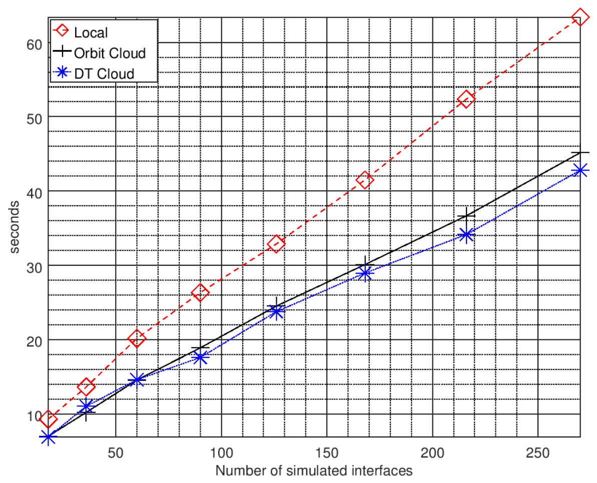


FIGURE 7. Boot time versus the number of simulated interfaces in a mesh topology.

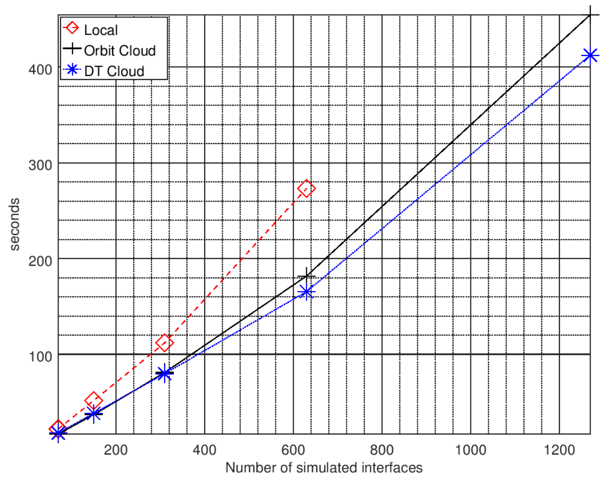


FIGURE 6. Boot time versus the number of simulated interfaces in a tree topology.

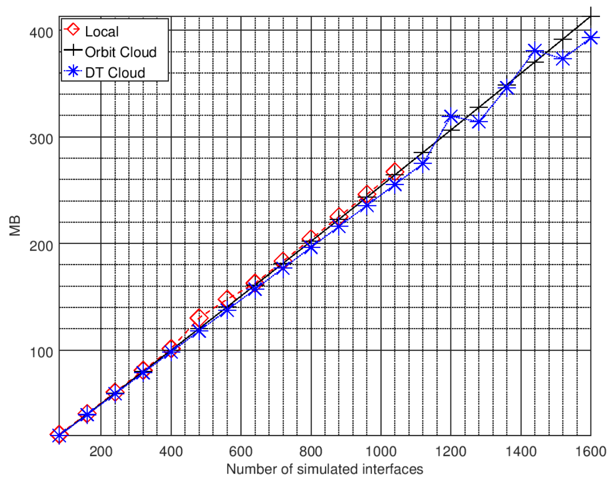


FIGURE 8. Used disk storage versus the number of simulated interfaces in a ring topology.

C. INITIALIZATION TIME

The initialization time represents the duration from the moment that the simulator is started, until all the simulated elements are created. The local machine has only 4GB of RAM, so the simulations stop at 130 elements in the case of the ring topology, because the emulator does not have sufficient memory to further increase the number of NEs (needed by the *docker* containers and simulated interfaces).

The variation of the boot time in seconds versus the number of interfaces simulated can be seen in Fig. 5, 6 and 7.

It can be observed that, even though the frequency of the CPU in the local system is higher than in the other systems, the boot time is higher. This situation arises because of the different hypervisor used in the different environments: on the local machine, VirtualBox is used, while in the two cloud environments, VMware, a solution tailored for server virtualization, thus more capable, is used.

The initialization time depends in a linear manner on the number of interfaces that the simulated topology contains. Even if the values measured are high, reaching approximately 300 seconds for 1000 interfaces, they are acceptable, being only the initialization values.

D. DISK STORAGE

The disk storage that the WTE needs was the next characteristic measured in the evaluation process. This is given by the docker containers that represent their file systems in the host operating system, but also by the Linux interfaces simulated. The measurement results are illustrated in Fig. 8, 9 and 10.

The results emphasize that the used disk storage depends in a linear manner on the number of the interfaces that are being simulated. In the ring topology case, for 960 emulated interfaces, the disk storage used by the WTE reaches 240 MB, in the case of the tree topology, for 1270 Linux interfaces, approximately 330 MB are used. For the mesh networks, the simulation reveals almost 65 MB used disk storage for a number of 270 simulated interfaces. All these measurements

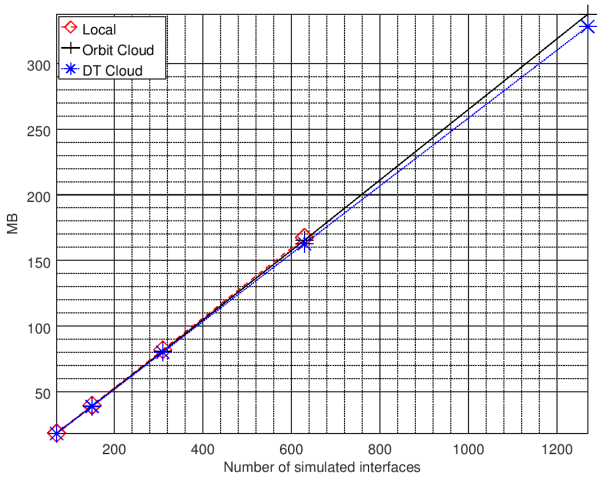


FIGURE 9. Used disk storage versus the number of simulated interfaces in a tree topology.

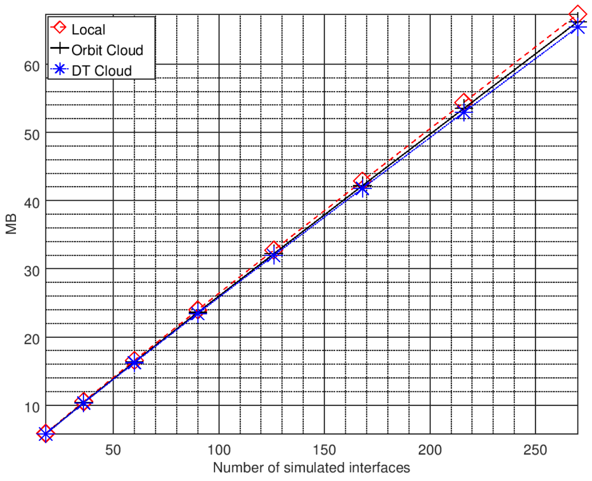


FIGURE 10. Used disk storage versus the number of simulated interfaces in a mesh topology.

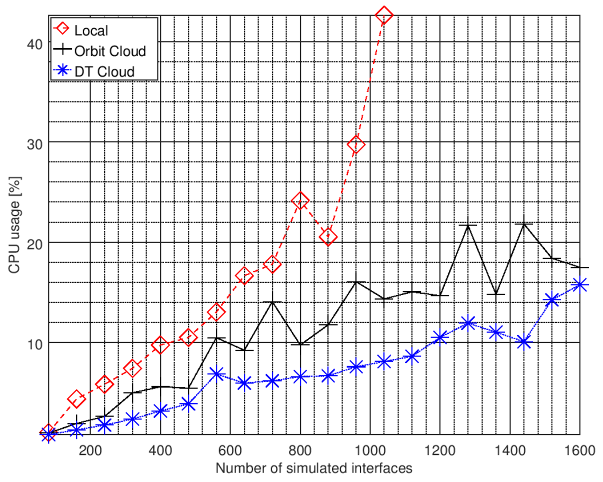


FIGURE 11. CPU usage versus the number of simulated interfaces in a ring topology.

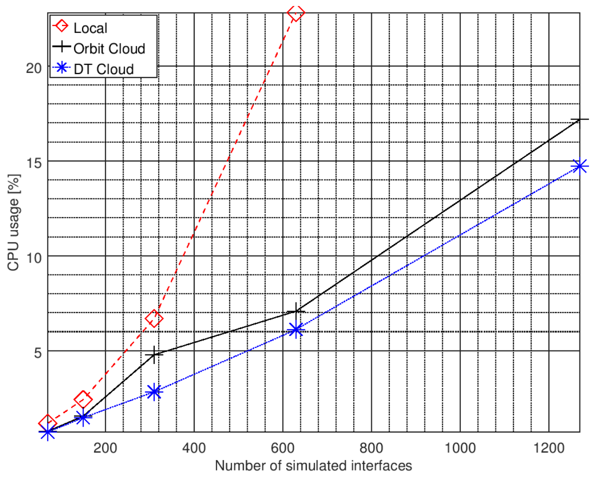


FIGURE 12. CPU usage versus the number of simulated interfaces in a tree topology.

reveal an average of 0.25 MB per each Linux interface created.

E. CPU USAGE

The previously measured characteristics are not relevant after the simulator was initialized. The CPU and RAM usages are more important, because they limit the topologies that can be simulated on a system.

The CPU usage is represented as the percentage of the used processing power from the total system processing power. For example, if the WTE would be using 25% CPU power in a system with 4 cores, it would mean that a core is entirely used by the simulator. The measurement results for the CPU usage, versus the number of simulated interfaces are highlighted in Fig. 11, 12 and 13.

The results reveal that, even if the method of taking multiple samples was used and an average was computed, the CPU usage does not have a perfectly linear dependency on the number of simulated interfaces. The trend, however, is linear. We can observe that the two cloud environments have a similar behavior. Some issues arise in the local measurements, these inconsistencies appearing in the charts because of the less capable hypervisor.

For 250 interfaces simulated, in a full mesh topology, the CPU usage reaches approximately 3%. For 1200 interfaces, in the case of the ring topology, the CPU usage reaches approximately 12% and in the case of tree topology, for the same number of interfaces, the simulator utilizes almost 14% of the CPU power. The results reveal that WTE does not use a very high amount of CPU power, even in the case of large topologies, in a close to idle state. If the SDN Controller will start requesting data from all the simulated devices, then the CPU usage will start becoming noticeable.

F. RAM USAGE

The last measured characteristic in the evaluation process is represented by the RAM usage. This is described as the percentage of the memory used with respect to the total amount

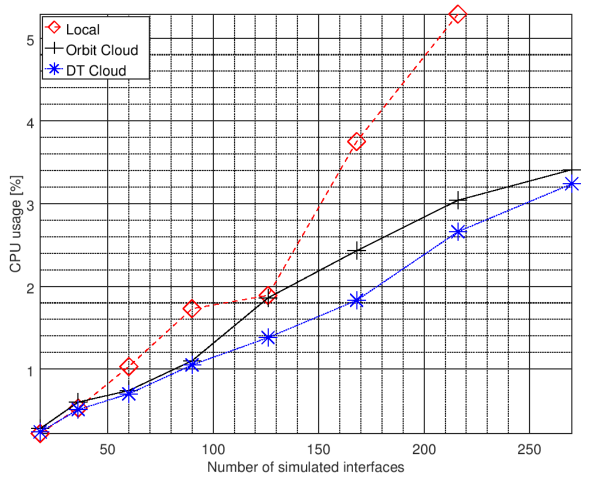


FIGURE 13. CPU usage versus the number of simulated interfaces in a mesh topology.

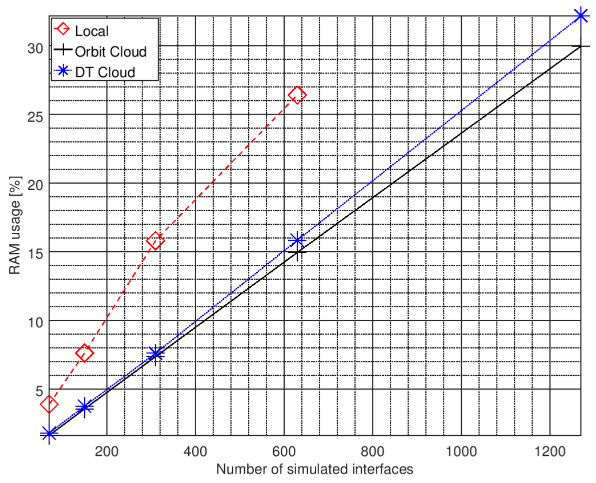


FIGURE 15. RAM usage versus the number of simulated interfaces in a tree topology.

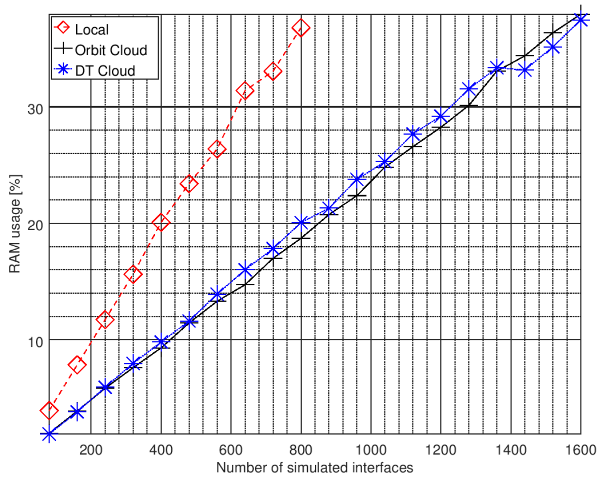


FIGURE 14. RAM usage versus the number of simulated interfaces in a ring topology.

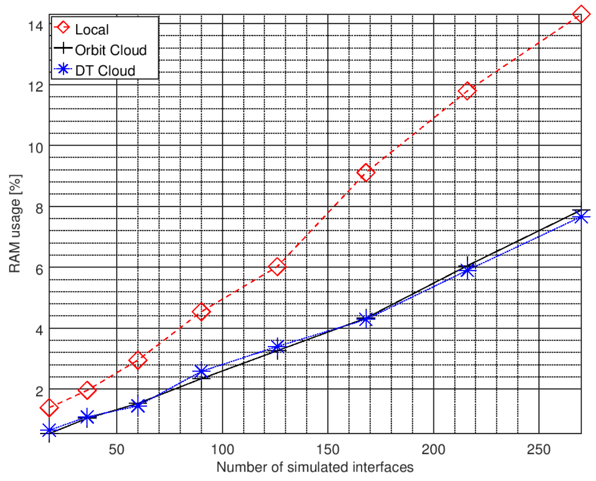


FIGURE 16. RAM usage versus the number of simulated interfaces in a mesh topology.

of RAM. This is the most important characteristic, which will limit the scalability of the simulator. If in the case of the CPU usage, if the WTE needs high processing power, it will eventually get it from the OS and the simulator processes will be further executed, even if the execution of the entire system will be slow. In the case of the RAM, if the simulator will further request memory and the system is unable to fulfill its request, the execution of the WTE will be stopped by the OS and the simulated topology will crash. The results of the RAM usage versus the number of simulated interfaces are highlighted in Fig. 14, 15 and 16.

The charts reveal a linear dependency between the RAM used and the number of simulated interfaces. In the mesh topology, for a number of 200 Linux interfaces, the RAM usage reaches almost 5%. The tree topology reveals a percentage of approximately 24 for the RAM usage, when simulating 1000 interfaces. In the ring topology, for the same number of Linux interfaces, the RAM usage reaches approximately the same value, of 24%. We can conclude that, regardless of the simulated topology, WTE requires approximately 0.025% of RAM for each simulated interfaces, and since we considered the measurements from a system with 8 GB of RAM, this would represent approximately 2 MB of RAM for each Linux interface.

VI. DISCUSSION

A. ADVANTAGES AND DISADVANTAGES

The evaluation results highlight some interesting conclusions about WTE. The variable that influences the most the behavior of the simulator it is not the number of simulated network elements, as one might find intuitive, but the number of interfaces that the topology contains. The larger this number is, the higher will be the initialization time, RAM used and disk storage needed.

The initialization time of the simulator is rather high for large topologies, reaching hundreds of seconds for networks containing thousands of interfaces. This time is not critical, though, because, once running, the simulator does not need to be reinitialized, unless the user wants to change the topology. What this influences mostly is the user experience. This time might not be optimizable any further, because the docker engine does not allow parallel creation of containers.

The disk storage needed by WTE does not represent a problem in current systems, even when simulating very large topologies. For example, when emulating a topology containing 4000 interfaces, approximately 1 GB of storage will be used on disk, which should not be an issue in nowadays machines.

The CPU usage is the measured characteristic that varies the most and is the least predictable. We notice an increase in usage with the number of network elements and with the number of simulated interfaces. The CPU usage is not very predictable, because the docker engine is responsible for allocating processing power to each of the running containers, according to its needs. During execution, these needs might be influenced also by the SDN applications that are running in the SDN controller and are requesting information from the simulated devices.

The size of the topologies that can be simulated is influenced mainly by the RAM available in the system where WTE was installed. In a system having 4 GB of RAM, about 1000 interfaces can be simulated. Even though this would mean, according to the previously described results, using only 2 GB of the RAM, the simulator will not be able to use more memory, because the operating system and maybe some other applications will need RAM as well. For more capable systems, that reach 8 GB of RAM, the number of simulated interfaces can exceed 2500, because the percentage of RAM needed by the OS is lower than in the previous case, so the emulator has more to use. This number refers to the Linux interfaces associated with the LTP objects from the ONF information models. If we are referring only to interfaces that are used for links between network elements (such as air interfaces or Ethernet interfaces), considering that such interfaces have also other LTP objects associated, according to the information models, we can conclude that around 800 such interfaces could be simulated.

B. COMPARISON WITH MININET

Most of the other simulators used in the context of SDN, including mininet, rely on the OpenFlow protocol. Since the Microwave Information Model was just recently released, in December 2016, no other simulators offers it yet. We will nevertheless compare WTE with mininet, even though their utilizations are slightly different.

From an architectural point of view, the two simulators are very much alike. They are both based on a Python framework that handles the simulation environment: creating hosts, switches or links between them, through virtual Ethernet pairs. If, in the case of mininet, hosts are simulated as processes, which are then connected to the network switches, WTE emphasizes the simulation of network elements and their functionality provided through the NETCONF interface.

Both mininet and WTE use, after initialization, a command line interface that waits for user commands. This CLI has knowledge about the simulated topology and can send commands to the emulated devices or can inform the user with regards to details about their state.

Even if the protocols exposed are different, mininet using OpenFlow and WTE basing on NETCONF, both simulators use the concept of virtualization based on Linux containers [47]. In mininet, they are used natively, directly from Linux. WTE uses docker for creating the Linux containers that represent the network elements.

From the topology description point of view, the approaches are different in the two simulators. Mininet offers the possibility of starting the simulator from a terminal and to specify through certain parameters the type of topology to simulate (having some predefined topologies). The other possibility is using its Python API and creating the topology directly in the Python code. WTE has a different approach: describing the topology to be simulated in a configuration file, using a predefined JSON format. This eliminates the need for the user to have previous programming experience with Python.

Functionally, both simulators use the same method for representing network links, virtual Ethernet pairs, regardless of the points that define the link: a host and a switch or two switches, in mininet, or two network elements, in WTE.

In WTE, every device is represented as a docker container which runs the NETCONF server that exposes the ONF information models. In mininet, for representing the OpenFlow switches, the Open Virtual Switch (OVS) solution is used, which is executed natively in the environment where mininet is installed (without using virtualization), or the user can choose other OpenFlow software switch to be used. For this reason, the initialization time, but also the resources needed by WTE are larger that the ones used by mininet.

VII. 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 ecosystems. 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.

We have proven that WTE represents a viable solution for SDN application developers. It allows simulating different topologies, which, depending of the capabilities of the machine where the simulation environment is installed, can have thousands of interfaces. In this manner, SDN application developers can execute also scalability tests on their applications. Even though real production networks can contain thousands of elements and tens of thousands of interfaces, WTE is a first step in emulating such Wireless Transport networks. WTE could be used also by network operators, for testing new SDN applications for their networks, or even test the interactions between several SDN applications and their effect on the network. Future research can focus on optimizing the WTE so that it can simulate even larger topologies, or finding a solution of running it in a distributed manner on several systems.

REFERENCES

[1] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Communication Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[3] D. Kreutz, F. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[4] A. Stancu, S. Halunga, G. Suciu, and A. Vulpe, "An overview study of software defined networking," in *Proc. 14th Int. Conf. Inform. Econ. (IE)*, 2015, pp. 50–55.

[5] I. F. Akyildiz, P. Wang, and S.-C. Lin, "Softair: A software defined networking architecture for 5g wireless systems," *Comput. Netw.*, vol. 85, pp. 1–18, Jul. 2015.

[6] H. Wang, S. Chen, H. Xu, M. Ai, and Y. Shi, "SoftNet: A software defined decentralized mobile network architecture toward 5G," *IEEE Netw.*, vol. 29, no. 2, pp. 16–22, Mar./Apr. 2015.

[7] R. Vilalta, A. Mayoral, R. Muñoz, R. Casellas, and R. Martínez, "Hierarchical SDN orchestration for multi-technology multi-domain networks with hierarchical ABNO," in *Proc. Eur. Conf. Opt. Commun. (ECOC)*, Sep./Oct. 2015, pp. 1–3.

[8] R. Vilalta *et al.*, "Hierarchical SDN orchestration of wireless and optical networks with E2E provisioning and recovery for future 5G networks," in *Proc. Opt. Fiber Commun. Conf. Exhib. (OFC)*, Mar. 2016, pp. 1–3.

[9] H. I. Kobo, A. M. Abu-Mahfouz, and G. P. Hancke, "A survey on software-defined wireless sensor networks: Challenges and design requirements," *IEEE Access*, vol. 5, pp. 1872–1899, 2017.

[10] T. Li, H. Zhou, H. Luo, I. You, and Q. Xu, "SAT-FLOW: Multi-strategy flow table management for software defined satellite networks," *IEEE Access*, vol. 5, pp. 14952–14965, 2017.

[11] J. Wu, Z. Ning, and L. Guo, "Energy-efficient survivable grooming in software-defined elastic optical networks," *IEEE Access*, vol. 5, pp. 6454–6463, 2017.

[12] R. D. R. Fontes, C. Campolo, C. E. Rothenberg, and A. Molinaro, "From theory to experimental evaluation: Resource management in software-defined vehicular networks," *IEEE Access*, vol. 5, pp. 3069–3076, 2017.

[13] ONF. (Sep. 2015). *Wireless Transport SDN Proof of Concept White Paper*. [Online]. Available: https://rs.opennetworking.org/wiki/download/attachments/262144003/1st_Wireless%20Transport_PoC_White_Paper.pdf?api=v2

[14] ONF. (Jun. 2016). *Wireless Transport SDN Proof of Concept 2 Detailed Report*. [Online]. Available: https://rs.opennetworking.org/wiki/download/attachments/262144003/2nd_Wireless%20Transport_PoC_White_Paper.pdf?api=v2

[15] ONF. (Dec. 2016). *Third Wireless Transport SDN Proof of Concept White Paper*. [Online]. Available: https://rs.opennetworking.org/wiki/download/attachments/262144003/3rd_Wireless%20Transport_PoC_White_Paper.pdf?api=v2

[16] ONF. (Aug. 2017). *Fourth Wireless Transport SDN Proof of Concept White Paper*. [Online]. Available: https://rs.opennetworking.org/wiki/download/attachments/262144003/4th_Wireless_Transport_PoC_White%20Paper.docx?api=v2

[17] ONF. (Dec. 2016). *TR-532 Microwave Information Model, Version 1.0.* [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2013/05/TR-532-Microwave-Information-Model-V1.pdf

[18] ONF. (Mar. 2015). *TR-512 Core Information Model (CoreModel), Version 1.0.* [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/Core_Information_Model_V1.0.pdf

[19] D. Raychaudhuri *et al.*, "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols," in *Proc. IEEE Wireless Commun. Netw. Conf.*, vol. 3. Mar. 2005, pp. 1664–1669.

[20] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 255–270, Dec. 2002.

[21] K. Roberts, Q. Zhuge, I. Monga, S. Gareau, and C. Laperle, "Beyond 100 Gb/s: Capacity, flexibility, and network optimization," *J. Opt. Commun. Netw.*, vol. 9, no. 4, pp. C12–C24, 2017.

[22] ESnet. (Oct. 2017). *100G SDN Testbed*. [Online]. Available: https://www.es.net/network-r-and-d/experimental-network-testbeds/100g-sdn-testbed/

[23] M. Berman *et al.*, "GENI: A federated testbed for innovative network experiments," *Comput. Netw.*, vol. 61, pp. 5–23, Mar. 2014.

[24] L. Liu *et al.*, "Experimental demonstration of OpenFlow-based dynamic restoration in elastic optical networks on GENI testbed," in *Proc. Eur. Conf. Opt. Commun. (ECOC)*, Sep. 2014, pp. 1–3.

[25] M. Ott, I. Seskar, R. Siraccusa, and M. Singh, "ORBIT testbed software architecture: Supporting experiments as a service," in *Proc. 1st Int. Conf. Testbeds Res. Infrastruct. Develop. Netw. Commun. (Tridentcom)*, Feb. 2005, pp. 136–145.

[26] B. Heller, "Reproducible network research with high-fidelity emulation," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2013.

[27] B. Lantz and B. O'Connor, "A mininet-based virtual testbed for distributed SDN development," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 365–366, 2015.

[28] ONF. (Dec. 2009). *OpenFlow Switch Specification, Version 1.0.0.* [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

[29] *Network Configuration Protocol (NETCONF)*, document RFC 6241, Jul. 2011.

[30] G. F. Riley and T. R. Henderson, "The *ns*-3 network simulator," in *Modeling and Tools for Network Simulation*. Berlin, Germany: Springer, 2010, pp. 15–34.

[31] S.-Y. Wang, C.-L. Chou, and C.-M. Yang, "EstiNet openflow network simulator and emulator," *IEEE Commun. Mag.*, vol. 51, no. 9, pp. 110–117, Sep. 2013.

[32] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a model-driven SDN controller architecture," in *Proc. IEEE 15th Int. Symp. World Wireless, Mobile Multimedia Netw. (WoWMoM)*, Jun. 2014, pp. 1–6.

[33] *YANG—A Data Modeling Language for the Network Configuration Protocol (NETCONF)*, document RFC 6020, Oct. 2010.

[34] ONF. (Mar. 2016). *SDN Architecture for Transport Networks*. [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/SDN_Architecture_for_Transport_Networks_TR522.pdf

[35] A. Stancu, A. Vulpe, O. Fratu, and S. Halunga, "Default values mediator used for a wireless transport SDN proof of concept," in *Proc. IEEE Conf. Standards Commun. Netw. (CSCN)*, Oct. 2016, pp. 1–6.

[36] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, p. 2, 2014.

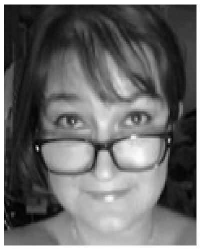
[37] R. Chamberlain and J. Schommer. (2014). *Using Docker to Support Reproducible Research*. [Online]. Available: https://ndownloader.figshare.com/files/1590657

[38] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2016, pp. 713–717.

[39] A. Stancu. (2017). *Wireless Transport Emulator*. [Online]. Available: https://github.com/Melacon/WirelessTransportEmulator
[40] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. 9th ACM SIGCOMM Workshop Hot Topics Netw.*, 2010, p. 19.
[41] A. N. Kuznetsov, "Ip command reference," Inst. Nucl. Res., Moscow, Russia, Tech. Rep. iproute2-ss020116, 1999.
[42] G. Kizer, *Digital Microwave Communication: Engineering Point-to-Point Microwave Systems*. Hoboken, NJ, USA: Wiley, 2013.
[43] W. Almesberger, "Linux network traffic control—Implementation overview," Ecole Polytechnique Federale de Laussane, Laussane, Switzerland, Tech. Rep., 1998.
[44] (2017). *IPERF—The Ultimate Speed Test Tool for TCP, UDP and SCTP*. [Online]. Available: https://iperf.fr/
[45] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. (2005). *IPERF—The TCP, UDP and SCTP Network Bandwidth Measurement Tool*. [Online]. Available: http://dast.nlanr.net/Projects
[46] ORBIT. *Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT)*. Accessed: Nov. 22, 2017. [Online]. Available: http://www.orbit-lab.org/
[47] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. 8th Int. Conf. Emerg. Netw. Exper. Technol.*, 2012, pp. 253–264.

ALEXANDRU STANCU received the B.S. degree in electronics and telecommunications from the University Politehnica of Bucharest, Romania, in 2012, and the M.S. degree in electronics and telecommunications from the University Politehnica of Bucharest in 2014, specializing in networks and software for communications, where he is currently pursuing the Ph.D. degree in electronics and telecommunications, with a thesis topic on software-defined networking. He is a Software Engineer with Ceragon Networks. His research interests include software-defined networking, computer networks, security, wireless transport, and mobile communications.

ALEXANDRU VULPE (M'12) received the Ph.D. degree in electronics, telecommunications and information technology from the University Politehnica of Bucharest, Romania, in 2014. His research interests include wireless sensor networks, mobile communications, security, quality of service, radio resource management, and mobile applications. His publications include over 50 papers published in journals or presented at international conferences. He has participated as a Researcher in a number of national or international projects in the area of security and Internet of Things, such as Reconfigurable Interoperability of Wireless Communications Systems, NATO Science for Peace Research project from 2007 to 2010, eWALL—eWall for Active Long Living (FP7 project, from 2013 to 2016), and Optimization and Rational Use of Wireless Communication Bands, NATO Science for Peace Project, from 2013 to 2015.



SIMONA HALUNGA (M'00) received the M.S. degree in electronics and telecommunications in 1988 and the Ph.D. degree in communications from the University Politehnica of Bucharest, Bucharest, Romania, in 1996. From 1996 to 1997, she followed post-graduate courses in management and marketing organized by the Romanian Trade and Industry Chamber and the University Politehnica of Bucharest, in collaboration with Technical Hochschule Darmstadt, Germany. From 1997 to 1999, she was a Visiting Assistant Professor with the Electrical and Computer Engineering Department, University of Colorado at Colorado Springs, USA. Since 2006, she was a Full Professor with the Telecommunications Department, Faculty of Electronics, Telecommunications and Information Technology, University Politehnica of Bucharest. She has published over 180 papers in different scientific journals or in the proceedings of different scientific conferences and participated in several nationally-funded, NATO Science for Peace and European research projects. Her research interests include multiple access systems and techniques, digital communications, communications systems, and digital signal processing for telecommunications. She was the Director of the Scalable Radio Transceiver for Instrumental Wireless Sensor Networks-Sarat nationally-funded research project from 2012 to 2016.