An IoT Network Emulator for Analyzing the Influence of Varying Network Quality

Stefan Herrnleben[1(✉)], Rudy Ailabouni[2], Johannes Grohmann[1], Thomas Prantl[1], Christian Krupitzer[1], and Samuel Kounev[1]

[1] University of Würzburg, Würzburg, Germany
{stefan.herrnleben,johannes.grohmann,thomas.prantl,
christian.krupitzer,samuel.kounev}@uni-wuerzburg.de
[2] Bosch Engineering GmbH, Stuttgart, Germany
rudy.ailabouni@de.bosch.com

Abstract. IoT devices often communicate over wireless or cellular networks with varying connection quality. These fluctuations are caused, among others, by the free-space path loss (FSPL), buildings, topological obstacles, weather, and mobility of the receiver. Varying signal quality affects bandwidth, transmission delays, packet loss, and jitter. Mobile IoT applications exposed to varying connection characteristics have to handle such variations and take them into account during development and testing. However, tests in real mobile networks are complex and challenging to reproduce. Therefore, network emulators can simulate the behavior of real-world networks by adding artificial disturbance. However, existing network emulators often require a lot of technical knowledge and a complex setup. Integrating such emulators into automated software testing pipelines could be a challenging task. In this paper, we propose a framework for emulating IoT networks with varying quality characteristics. An existing emulator is used as a base and integrated into our framework enabling the user to utilize it without extensive network expertise and configuration effort. The evaluation proves that our framework can simulate a variety of network quality characteristics and emulate real-world network traces.

Keywords: Mobile networks · Network quality · Network emulation · Application development

1 Introduction

In recent years, the number of IoT devices has increased rapidly [13]. These devices can be found in many different areas, such as smart homes, health care, smart buildings, vehicles, agriculture, smart cities, and industrial automation [8,12,17]. Since many IoT devices are mobile, communication over wireless networks is often necessary. For this, WiFi, mobile networks, as well as proprietary radio standards are used [15]. At the application level, IoT protocols such as MQTT or CoAP are often employed [3]. However, wireless networks are

H. Song and D. Jiang (Eds.): SIMUtools 2020, LNICST 370, pp. 580–599, 2021.
https://doi.org/10.1007/978-3-030-72795-6_47

exposed to significantly more interference factors than wired networks. Buildings, the velocity with which the device moves, weather, and the distance to the mobile radio station influence the connection quality [10,16,19]. These influencing factors can lead to packet delays, bandwidth restrictions, packet loss, or jitter.

IoT devices often require a stable connection to internet services. Temporary performance limitations could lead to undesired application behavior and dissatisfied users. Application developers should, therefore, test their applications in networks with different quality characteristics. However, tests in real-world networks are time-consuming, expensive, and not suitable for repetition. Network emulations can simulate the quality characteristics of networks and add artificial interference. The impact of varying network quality on the application can be simulated and tested by the developer through emulated networks. Unfortunately, network emulators often require time-consuming setup, complex configuration, and technical knowledge of networks. Furthermore, network emulators are often not designed for integration into software testing environments.

In this paper, we present a framework to emulate the characteristics of IoT networks. An existing network emulator is embedded in our framework to provide artificial interference at the network interface layer. By this, our emulator supports varying the maximum bandwidth, the packet delay, the packet loss rate, the packet duplication rate, and the jitter value. Besides the disturbance of the link, connection failures can be emulated as well. A scheduler adjusts the connection quality at predefined times to reconstruct a temporal behavior. The definition of time events enables the framework to simulate the movement of devices, changes in the environment, or a variation of disruptive factors. Besides manually specifying the quality characteristic changes and times, network measurement traces can also be imported into the event queue. Our work provides the following core contributions. First, the developed framework can be easily instantiated for efficient use by application developers. Second, measurements from real-world networks can be imported into the scheduler to reconstruct realistic scenarios. Third, a provided REST API for configuring and operating the emulator makes it ideal for integration into automated software test pipelines.

To ensure that the quality characteristics are correctly emulated, the first part of the evaluation focuses on these individual factors. The timing of the scheduler's events is also analyzed within this part of the evaluation. In the second part of the evaluation, a real-world network with Message Queuing Telemetry Transport (MQTT) traffic is measured and emulated. The evaluation will show that real network traces can be imported and emulated realistically.

The remainder of this work is structured as follows. Section 2 introduces existing emulators and their differences. Next, Section 3 presents our developed framework for the emulation of IoT networks and discusses the modules and features. Section 4 evaluates the emulator. The first part of the evaluation examines the precise appliance of limited quality characteristics on a link; the second part presents the test results of a real-world network's emulation. Section 5 concludes the work, identifies some limitations and states future work.

2 Related Work

Network emulators are already widely used to artificially limit networks in certain characteristics and execute measurements and benchmarks on the restricted networks. Most network emulators have been developed for specific purposes and fulfill different requirements. In this section, we introduce the open-source network emulators well-known in science. Generally, network emulators can be divided into the following two categories [11]: *Network Link Emulators (NLEs)* and *Virtual Network Emulators (VNEs)*.

2.1 Network Link Emulators (NLEs)

NLEs are lightweight and simple to use network emulators. They are usually limited to the link layer and are used to manipulate packets in such a way as to emulate specific network conditions. Most NLEs can influence the bandwidth, latency, packet error rate, and jitter of a network link but are limited to the machines they run on.

NISTnet [4] is a Linux based network emulation package that was originally developed and released by the American *National Institute of Standards and Technology (NIST)* in 2003. NISTnet can be set up on a single Linux computer and allows this computer to run as a virtual router. This virtual router can then be used to emulate variable network conditions by influencing different network parameters, such as packet delay distribution, congestion, loss, bandwidth limitation, as well as packet reordering and duplication. NISTnet is implemented as a Linux kernel extension and offers an X-Window System-based user interface, which can be used to monitor the network traffic and manage the emulation. Furthermore, NISTnet supports replaying real-world network traces. NISTnet is nowadays no longer maintained and actively developed. Most of its functionality has been incorporated into NetEM and the Linux *iproute2* kit.

NetEm [7] is currently one of the most popular NLEs and is included in the Linux kernel. Stephen Hemminger developed it at the Linux Foundation, and has incorporated many of the functionalities of NISTnet. NetEm is part of the Linux iproute2 kit, a collection of tools and utilities used to control TCP/IP networks and traffic in Linux. NetEm is an enhancement of the Linux *Traffic Control (TC)* system. It allows manipulating incoming and outgoing packets to a specific network interface by configuring delays, packet loss rates, and duplications rates, and packet reordering. Users of NetEm can either add discrete artificial delays or add delay distribution models. Furthermore, NetEm allows interfering traffic by adding random noise to a specific percentage of packets. Although NetEm is a powerful NLE, it has some limitations. One of the main issues with NetEm is related to Linux's timer granularity, since Linux does not run in real-time [7].

Similarly to Linux's TC, the Free-BSD based *Dummynet* [14] network emulator can also manipulate the kernel's IP queue to emulate variable network quality parameters. Dummynet allows emulating finite and bounded-size queues, bandwidth limitations, delays, and packet losses.

For individual use cases, NLEs are powerful network emulators, especially when multiple ones are combined, as is the case with the iproute2 of Linux. However, one of the main shortcomings of NLEs is their limitation to a single computer. This makes it difficult to emulate larger network topologies. Since most NLEs manipulate the packets coming into and leaving a specific network interface, one would need to use some virtualization technology to scale up the emulation. This makes the tools more complex and more challenging to use.

2.2 Virtual Network Emulators (VNEs)

VNEs aim to emulate entire network topologies. In VNEs, users can create all kinds of network topologies and can practically control every aspect of a network stack. However, this complexity often results in them being complicated and difficult to use.

EMULAB [18] is a network emulation platform that is run and hosted by the Flux Research Group, which is part of the School of Computing at the University of Utah in the United States. EMULAB can be used for research, development, and educational purposes. However, it is a time- and space-shared platform, which means not everyone can use it. EMULAB has multiple installations around the world and requires those who want to use it to apply in advance, stating the number of nodes they require and for how long. Multiple criteria are considered when granting access to EMULAB, such as being associated with an academic institution, having a clear deadline for a conference paper submission, and needing to perform experiments as part of a publication that will be peer-reviewed.

The *Integrated Multiprotocol Network Emulator/Simulator (IMUNES)* [20] is an open-source and free general-purpose IP network emulation tool that was developed in 2004 by the University of Zagreb in Croatia. It is available for both the FreeBSD and Linux operating systems and aims to offer realistic network topology emulation and simulation. It uses kernel-level virtualization techniques to instantiate multiple lightweight virtual nodes, which can be connected using kernel-level links and bridges to instantiate complex network topologies. Networks emulated using IMUNES can have a large number of virtual nodes all running on a single physical machine. IMUNES comes bundled with a console to configure the emulated network.

Mininet [6,9] is one of the best well known open-source network emulators and enables the creation of a virtual network containing virtual hosts, switches, and links on a single machine. Hosts are virtual machines running standard Linux network software. Mininet can create custom network testbeds that enable sophisticated topology testing without having to connect to physical networks. As Mininet can run real code, any network applications developed and tested in Mininet, specifically network control and routing software, can be migrated to a real system with minimal changes.

The *Common Open Research Emulator (CORE)* [1] is an open-source network emulator that originally bases on IMUNES. It has forked from IMUNES in

2004 and further expanded the emulator to add new functionality, such as support for wireless networks, mobility scripting, distributed emulation over multiple computers, and more. The Boeing company originally released it in 2008, and it is still further maintained and developed by a community of developers to this day on GitHub[1]. Like IMUNES, CORE uses the operating systems' virtualization technology to instantiate large network topologies. CORE allows for instantiating lightweight virtual networks and virtual nodes, e.g., Ethernet ports, routers, and PCs connected using the host operating system's kernel. These virtual machines are created using the Linux network namespace virtualization technology and are connected using Linux Ethernet bridging. This allows each virtual node to share resources such as the memory, CPU time, and the file system with the host system, but have their own network stack and process environment. Real network interfaces can also be connected to the virtual nodes to connect real and virtual worlds, which allows for even more powerful network emulation [2]. Users can launch a terminal from each of the emulated nodes, which in turn permits running any command and launching any application over the emulated network.

More recently, the NetSec Research Group from the Department of Engineering and Architecture at the University of Parma in Italy has developed *NEMO* [5]. NEMO is written in Java and aims to be a highly flexible and portable network emulator. It seeks to be platform-independent by taking advantage of Java's virtual machine and runs completely at the user-space. While NEMO can be used to develop and test new communication protocols, it also comes bundled with an extensive collection of standard protocols. It includes an implementation of the TCP/IP stack that is entirely independent of the underlying operating system. NEMO allows connecting real networks with the virtual network and can run emulations that are distributed among multiple physical machines.

Although many of the VNEs partly offer graphical tools for configuration, the creation of emulated networks is often complex. A lot of configuration effort is also required when connecting virtual and physical networks. During software testing, influencing quality parameters usually requires knowledge of the emulator and may require in-band reconfiguration of the emulators.

3 Approach

This section describes our framework for network emulation focusing on IoT communication. Our network emulator aims to test mobile IoT applications under varying quality characteristics at the development stage. The emulator artificially influences the quality of network connections so that the application can be tested under degraded quality characteristics.

Most emulators introduced in Sect. 2 offer great features and can influence networks in several aspects. However, using these emulators for application developers, especially in automated test environments, often requires much technical

[1] https://github.com/coreemu/core.

knowledge and manual intervention. We define the following requirements for a network emulator focusing on IoT application development:

1. Simple and easy to use: Application developers do not have to be network experts to use the emulator. Already defined disturbance scenarios can be reused to generate reproducible results.
2. Emulation of the temporal behavior of real-world networks: The quality characteristics should be changeable during a running emulation. This enables, e.g., simulating a road trip by a car passing different networks (e.g., 3G, LTE, or 5G).
3. Connection with real-world physical networks and devices: Both virtual and physical networks and devices should be able to be integrated into the emulator. For example, an IoT system benefits from including a real MQTT Message Broker.
4. Remote control: The network and connection characteristics can be controlled via API calls. This allows using the emulator in automated test environments.
5. Running programs and applications over the emulated network: The emulator can start applications and execute commands on the devices.

As indicated in Sect. 2, a large number of network emulators already exist, which in particular already supports many of the network characteristics to be controlled. However, to the best of our knowledge, there is currently no network emulator that meets the previously defined requirements. This is especially true for the easy integration of the emulator into software development pipelines. In this paper, we extend an existing emulator with these features.

Our research indicates that the *CORE* network emulator best suits as a foundation for our implementation. It partly fulfills the first requirement since it is a lightweight network emulation platform. Judging the ease of use is subjective as it is specified by many factors, such as the users' skills in writing Python scripts and their experience in network engineering. CORE can connect emulated virtual networks to real-world physical ones. However, this requires some complex configuration effort. Currently, CORE does not automatically support the emulation of the temporal behavior of networks and the ability to control the emulator remotely.

Figure 1 shows the architecture of our network emulator, including the components of the CORE network emulator. The emulator consists of four modules. The central part of our network emulator, denoted by *Network Emulator* in Fig. 1, abstracts the technical details from the users and combines all of the other modules into a working network emulator. On top of that, our emulator contains the modules *TopologyConfigurator*, *Scheduler*, and a *REST API*, which we describe in the following. Our emulation framework is available as open source and can be downloaded from our Git repository[2].

The *TopologyConfigurator* module is responsible for instantiating and managing the topology. It contains all the logic and methods required to manage

[2] https://gitlab2.informatik.uni-wuerzburg.de/descartes/iot-and-cps/iot-network-emulator.

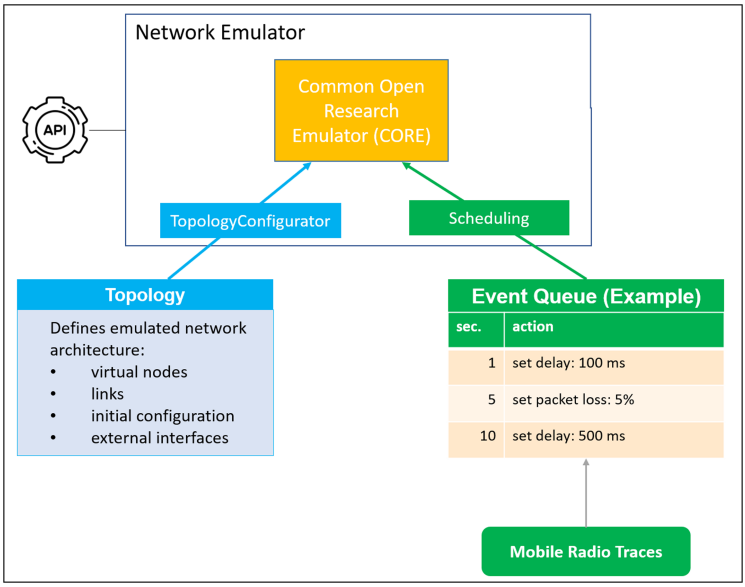

Fig. 1. The architecture overview of the network emulator.

the node and link entities. The module furthermore provides interfaces to add, modify, and remove these nodes and links. The hosts, connected to the emulator, can be virtual nodes, switches, hubs, or physical nodes, connected to Ethernet interfaces. The virtual nodes are deployed running an integrated SSH server by default to manage them and execute scripts. Starting traffic generators like *Iperf* or executing measurement scripts are some examples of using the SSH interface. If physical nodes connect to the emulated network, it is sufficient only to provide the network interface name. Our emulation framework instantiates all required virtual bridges between the physical and virtual networks. This is a great strength of our emulator compared to the original version of CORE, where these tasks have to be done manually. The module further provides the functionality to update the connection quality characteristics of links. It can add artificial network delay, jitter, packet loss rate, packet duplication rate, and limit the bandwidth of a link. The *TopologyConfigurator* module abstracts the technical complexity of network emulation from the user to easily integrating our emulator. At starting the emulator, users have the option to define a topology configuration as a YAML file. The topology configuration file includes fundamental information and parameters regarding the emulation. It defines the logical network addresses and the virtual nodes that will run in the emulation and the links between them. API calls can generate the emulated network entirely if no topology configuration is provided.

The *Scheduler* module emulates the temporal behavior of networks by scheduling the execution of certain events at specific times. Supported events are adding, removing, or modifying connection quality characteristics. This allows, for example, to add a delay later or to change the maximum supported bandwidth multiple times. Furthermore, the events can execute commands on virtual nodes, to, e.g., start programs or execute additional configuration tasks on the nodes. The events can be defined as an event queue in a configuration file. Each event is specified by the event type, the time when it should be executed, and additional optional execution arguments. One of the greatest strengths of the scheduler is the emulation of the temporal behavior of real-world networks. For this, the scheduler can import network measurement trace files. An event with the new value is created when a change in quality characteristics occurs in the measurement from the real-world network. A possible use-case for this is testing software for vehicular applications. The quality characteristics such as maximum bandwidth, delay, and packet loss can be varied along the route depending on network coverage and the mobile network used. Developers of such software can measure the internet connection quality on a certain route in advance and then use our emulator to reconstruct the entire route's network conditions. Developers can then perform experiments and investigate their software's behavior under these connection quality conditions, without having to drive the route again.

The *REST API* enables interaction with the network emulator at runtime. Upon starting our network emulator, a web server listening to HTTP requests is started in the background. Instead of configuration files, users can configure the topology and the events via REST API calls. Through REST API calls, users can further modify the emulation, i.e., adding or removing nodes and links. Also, the connection quality of specific links can be influenced (e.g., by adding artificial delays, bandwidth limitations, or increasing the packet loss and duplication rates) as well as the link jitter value. Further, users can retrieve information on the emulator's status via the REST API, including the current topology (i.e., nodes, their attributes such as IP addresses and interfaces), and the links between the nodes. Users can also get detailed information about a specific node, such as its neighbors and connections. Also, information about the links, including their current quality characteristic, can be queried. The REST interface is a powerful feature to integrate the network emulator into a software test and build pipelines, as both the instantiation and the execution of the emulator can be fully automated.

4 Evaluation

This section evaluates our emulator in two dimensions. Section 4.1 measures and validates the influence of quality characteristics on network traffic. Also, the temporal behavior of the scheduler is evaluated in this section. In Sect. 4.2, the quality characteristics of a real-world network are captured, and the communication behavior is then emulated in a virtual environment. Section 4.3 briefly discusses the validity of the measurements.

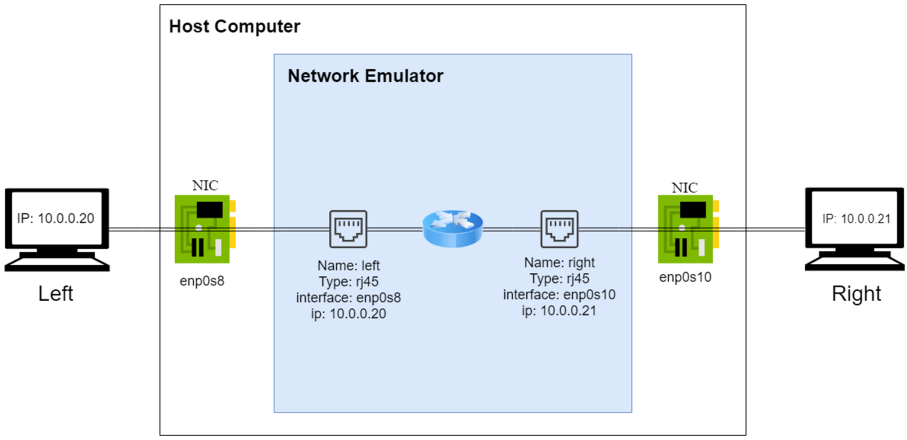

Fig. 2. Testbed setup for connection quality characteristic evaluation.

4.1 Connection Quality Characteristics

Our network emulation framework can manipulate different network quality characteristics: packet delay, bandwidth, packet loss rate, packet duplication rate, and jitter. In this evaluation, we verify that our network emulator correctly manipulates the quality of network links based on those five parameters. For each of these parameters, we perform an individual test using a common testbed.

Testbed Setup. All of the following experiments are performed on a notebook with an Intel Core i7-8550U CPU running at 1.80 GHz with a maximum turbo frequency of 4.0 GHz with 16 GB RAM. The network emulator is executed on a Linux virtual machine within the device and has access to four out of eight processor cores and 6 GB of RAM. The virtual machine runs Ubuntu version 18.04.2 LTS. For the hypervisor, we use version 5.2.22 of VirtualBox.

Figure 2 illustrates the experiment setup. Our topology consists of two *RJ45* interfaces connected to a virtual switch within the emulator. The two physical interfaces are connected to two virtual machines, simulating the connectivity to two other physical nodes. Each of the external virtual machines also runs Ubuntu version 18.04.2 LTS and have access to 2 GB of RAM, and two of the CPU cores.

Delay. The delay measurements verify that (i) the emulator applies the set delay correctly, and (ii) the scheduler executes the change at the correct time. To evaluate the network delay, we use an echo request/reply (ping) test. In this test, a packet is sent from one node to another, and upon receipt, a reply is sent back to the sending computer. The time it takes for the packet to be delivered and returned is called *Round Trip Time (RTT)*.

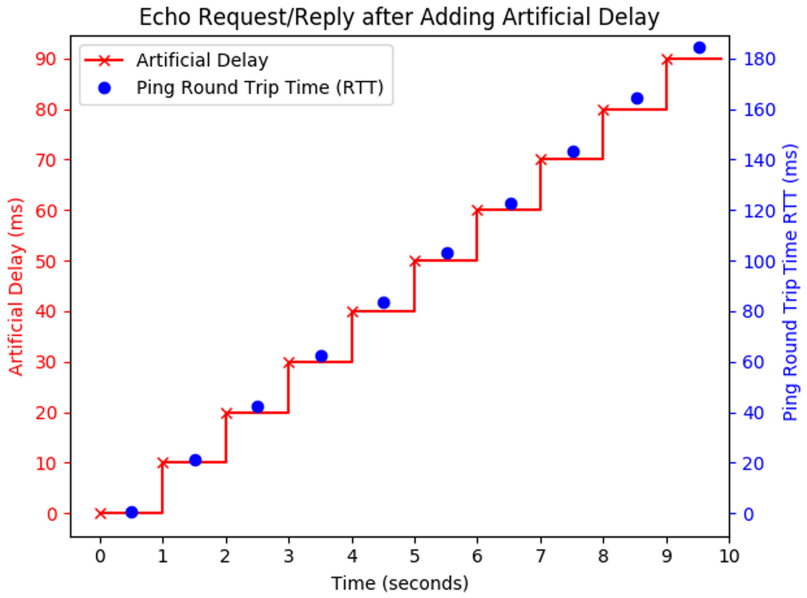

Fig. 3. The results of the delay measurements.

For the measurement, we send 50 echo requests sequentially per second from the right computer (ref. Fig. 2) to the left one. We run the test for a total of ten seconds. To do this, we schedule an echo request test every second.

To evaluate our emulator's scheduler module and the ability to variate the network delay, we defined an event queue containing ten link update events. Each of these events occurs every second and increases the delay value on the link between the left and right computers by ten milliseconds, starting by zero milliseconds at second zero.

The measurement results are depicted in Fig. 3. The x-axis of the graph shows the elapsed time of the experiment in seconds. The left y-axis shows the added artificial delay values in milliseconds. Likewise, the red step plot also denotes the added artificial delay values. More specifically, the red 'x' marks the time when the delay was set. Each delay value is stable for one second until the next delay value is set. The right y-axis in blue represents the RTT and has a linear relationship with the left y-axis. The RTT of a ping refers to the total time it takes for the echo request to be sent from one computer to another and the time for the echo reply back. Therefore, the RTT is expected to be twice the network delay/latency plus any required processing time. The y-value of the blue dots represents the average RTT of the 50 echo requests sent in that second. The x-value refers to the time interval of when those 50 echo requests were sent.

Optimally, each blue dot lies slightly above each step, on the red line. That means that the RTT is precisely double the delay plus the required processing time. Some of the RTT values are slightly higher than others, which could be

due to other events happening in the system and the fact that we were sending 50 packets each second at very short intervals. This means that the kernel and the network interface have to buffer some packets before sending them out. From this experiment for delay investigation, it can be concluded that both setting the correct link delay and scheduling the events works as expected.

Bandwidth. In this part of the evaluation, we investigate influencing the bandwidth of a particular link within a network. For this measurement, we use the same testbed setup described in Fig. 2 and measure the throughput capacity between the left and right computers. To determine the bandwidth of a link, we use the *iperf3*[3] Python library. This library provides a wrapper for the *iperf3* tool. *Iperf* is a well known and widely used tool for actively measuring the maximum achievable bandwidth on networks. The Python wrapper provides convenience methods for performing tests using *iperf3* and parsing their results. To perform the test, we run *iperf3* in server mode on the left computer, and we start the emulator on the host computer, without providing an event configuration file. On the right computer, we execute a Python script that first performs API calls to set the appropriate bandwidth limits.

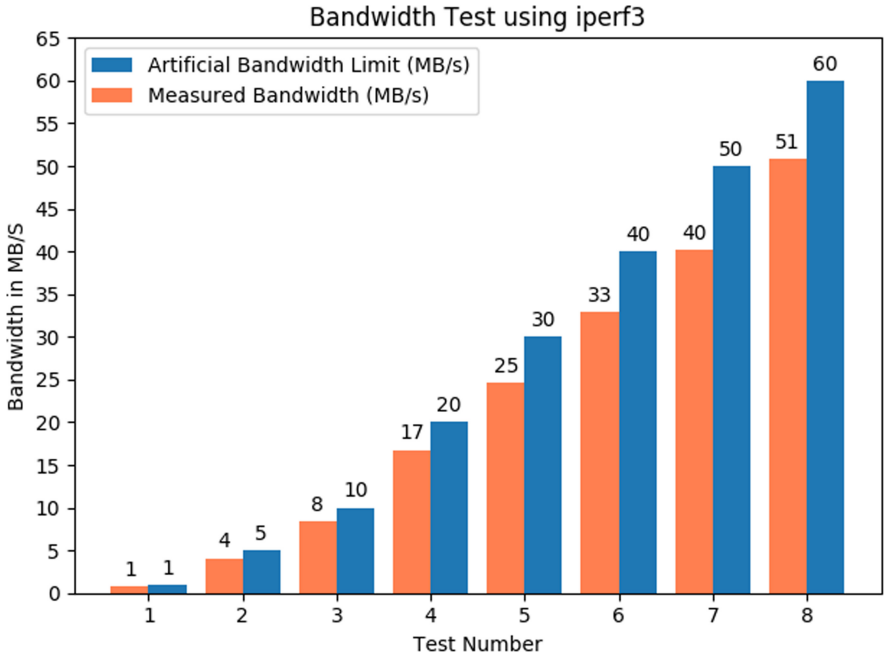

Fig. 4. Results of the bandwidth measurements. (Color figure online)

[3] https://pypi.org/project/iperf3/.

We run eight different measurements, each with an artificial limited emulated bandwidth of 1 MB/s, 5 MB/s, 10 MB/s, 20 MB/s, 30 MB/s, 40 MB/s, 50 MB/s, and 60 MB/s. For each bandwidth limit, the throughput between the left and right computer is measured for a total of 60 s. This measurement is repeated a total of eight times for each bandwidth limit. All of the throughput values from the eight measurements for each bandwidth limit are averaged.

The results of the measurement are depicted in Fig. 4. The x-axis denotes the measurement number. The y-axis shows bandwidth value in Megabytes per second. The blue bars refer to the bandwidth limit set using the emulator and denote the expected results. The orange bars represent the measured throughput itself, specifically the average of all eight measurements performed for each bandwidth limit. The numbers above the bars represent their values and are rounded down to the nearest integer.

From this investigation, it can be concluded that the actual throughput never exceeds the maximum defined bandwidth. Furthermore, it can be seen that by increasing bandwidth, a difference between the artificial limit and the measured bandwidth occurs. We assume that TCP's window size causes this difference. However, we asserted that our emulator appropriately modified the QDISC using NetEm.

Packet Loss Rate. This evaluation investigates the influence of our emulator on the packet loss rate of a specific link. The packet loss rate refers to the number of lost packets divided by the total number of sent packets. We use the testbed setup shown in Fig. 2, and evaluate the packet loss rate between the left and right computers by using echo requests and replies.

For the test, we perform five measurements, each with different packet loss rates of 20%, 40%, 60%, 80%, and 100%. Each test sends 1000 echo requests and is repeated 30 times. The packet loss rate is set via API call within the measurement script.

The results of our measurements can be seen in Fig. 5. The x-axis in the graph denotes the measurement number. The y-axis represents the packet loss rates in percent. The blue bars refer to the packet loss rate that was applied by the emulator and represent the expected results. The orange bars depict the average packet loss rate of all 30 measurements for each of the packet loss rates that we applied using the emulator. The numbers above the bars represent the bars' height, i.e., the expected values, respectively, the measured values and are rounded two decimal places.

It can be seen that the average packet loss rate of the measurements is very close to the rate that was applied by the emulator. From this, we can conclude that our network emulator is appropriately applying the packet loss rate parameter.

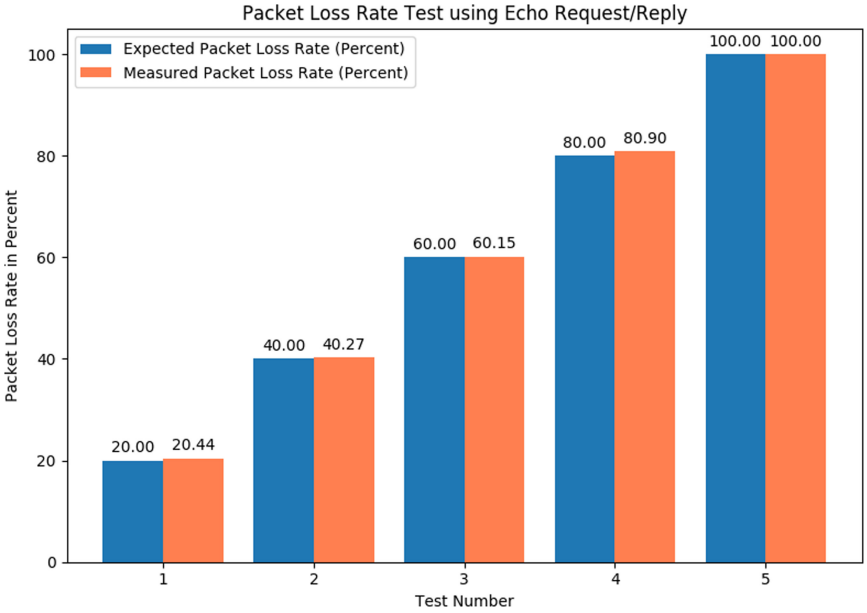

Fig. 5. The results of the packet loss rate measurements.

Packet Duplication Rate. This measurement verifies whether the set packet loss rate is correctly adopted. For this, the measurement setup from the previous section, including the echo reply/request measurement method, is used. The unique sequence number of the ICMP message identifies duplicate packets in the response log. We apply the packet duplication rates of 20%, 40%, 60%, 80%, and 100%. Each measurement sends 1000 echo requests and is repeated 30 times, before averaging the packet duplication rates.

The results of our measurements can be seen in Fig. 6. Like in the previous experiment, the X-axis represents the number of the measurement, and the y-axis represents the packet duplication rates in percent. The blue bars denote the packet duplication rate that we applied using our emulator and the results we expect to see. The orange bars represent the actual measurements, specifically the average of each of the 30 measurements for each of the five packet duplication rates. The numbers at the top of the bars are rounded to two decimal places.

It can be seen that the measured packet duplication rate of the emulated network matches the set value. From this, we can conclude that our network emulator is appropriately adjusting the packet duplication rate of network links accurately.

Jitter. This measurement evaluates the setting of an artificial jitter on a link. Jitter is defined as the variance in the time that data packets take to traverse a network path. For this measurement, we use the setup from Fig. 2 and measure the connection quality between the left and right computer using echo

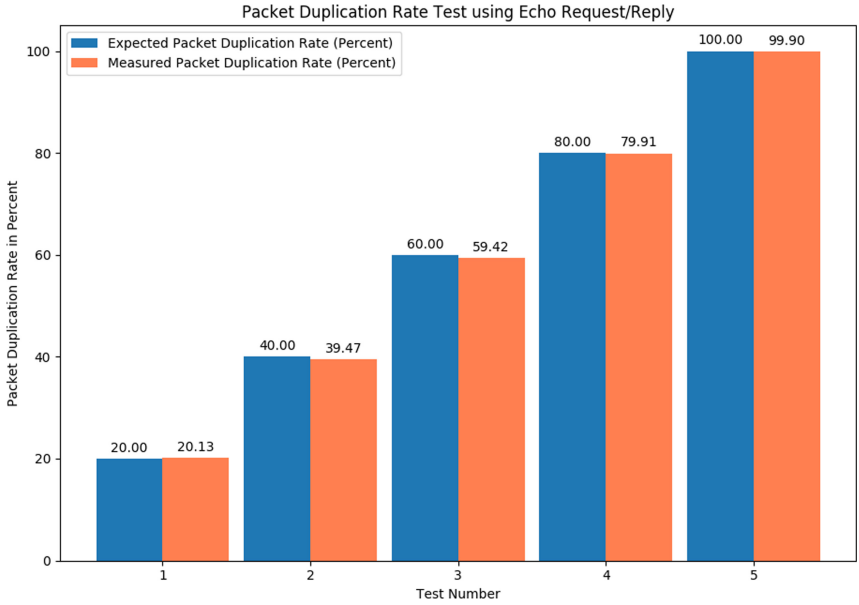

Fig. 6. The results of the packet duplication rate measurements.

requests/replies. As jitter is related to the delay value in a way, we first have to use the emulator to apply an artificial delay to the link between the right and left computers. Explicitly, we set the delay value to 100 ms. We then used the emulator to apply a jitter value of 50 ms to the link between the right computer and the switch.

For the measurement, we send echo requests for 60 s, with an interval of 500 ms between each request. From the individual timestamps of the ICMP replies, we calculate the standard deviation of the RTT. Since we set a link delay value of 100 ms, provided there is no network jitter, the average RTT of the link between the left and right computers should be approximately 200 ms plus some processing time.

The results of the measurement can be seen in Fig. 7. The x-axis denotes the elapsed time of the experiment. The y-axis represents the RTT values in milliseconds. The blue plot represents the variations between the RTT values. The orange line in the middle represents the theoretical average RTT if the jitter value was 0. The green line represents the maximum theoretical RTT value we should expect to see, i.e., the average theoretical RTT value plus the jitter value. Likewise, the red line at the bottom represents the minimum expected value, i.e., the average RTT minus the jitter.

In total, 120 echo requests were transmitted, 114 replies were sent back, and six were lost caused by a timeout of 60 s for the whole test. Figure 7 shows that most measurement results reflect our expectations, i.e., RTTs between 150 ms and 250 ms. The expected interval must be set to twice the set jitter value because

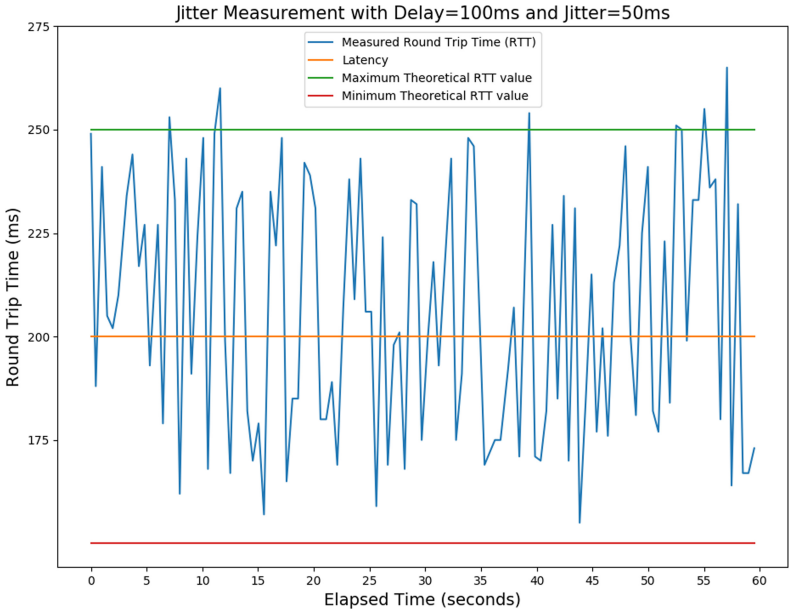

Fig. 7. The results of the jitter measurements. (Color figure online)

the packets pass through the link twice during an echo request and reply. The six values slightly above 250 ms can be explained by the processing time, which is not subtracted from the measured transmission time.

4.2 Evaluation of Temporal Behavior Emulation

This section evaluates the temporal behavior of the emulation framework within a real-world IoT use-case. For this, we employ a publish/subscribe communication scenario using the *Message Queuing Telemetry Transport (MQTT)* protocol for communication between two clients and one message broker. To emulate a real-world network's temporal behavior, we first need to capture the characteristics of such a network over a certain period. The capturing is done in a first experiment, using a public MQTT message broker. In a second experiment, we emulate the network to the public broker by the captured network trace and replace the broker by a local one.

Measuring the Temporal Behavior of a Real-World Network. In this measurement, we capture the temporal behavior of a real-world network in terms of network latency. The measurement setup is depicted in Fig. 8. Two virtual machines running an MQTT client, a subscriber and a publisher, communicate with each other over a public MQTT message broker[4].

[4] https://test.mosquitto.org/.

Fig. 8. The setup used to gather data of the network quality between the MQTT clients and the public MQTT Mosquitto broker.

For the publisher and subscriber clients, we developed an MQTT application using the *Eclipse Paho* library. The publisher application connects to the MQTT broker and publishes the system's current time along with a message sequence to a topic called "time" every second. With each published message, the sequence is incremented. We can control the time interval between each publication and the total number of messages that the application should publish. The application running on the subscriber client subscribes to the topic "time" and receives all messages related to this topic from the broker. Upon receiving a message, the subscriber subtracts the time in the message from the current time. By doing this, we can approximate the time it took for the message to be sent by the publisher, processed by the broker, forwarded to the subscriber, and processed by the subscriber. To obtain accurate results regarding each message's transport duration, we synchronized the clocks of the two MQTT clients directly before starting the experiment. Since in our setup, the publisher is publishing a message each second, we can use each message's sequence number to derive the sending time of the message. Upon calculating the transport duration of the various messages at the subscriber end, a trace file is generated, including the sequence number of each message and its measured transport duration. The total running time for the experiment is 60 s.

Emulating the Temporal Behavior of a Real-World Network. In this part of the evaluation, we emulate the network with the characteristic captured in the previous part. For the experiment, we replace the public MQTT message broker by a local broker, running within our the emulated network. The clients now connect to the local broker. The setup is shown in Fig. 9.

We use the trace file from the real-world measurement and import it into the event configuration so that the fluctuations are represented as events. For the evaluation, one client again publishes the current time and a sequence number, which is subscribed by the other client. As in the first part of the experiment, the subscriber measures the transport duration of each message and stores the duration along with each messages sequence number.

The data, captured from the real-word network and the measured data from the emulated network, are depicted in Fig. 10. The x-axis denotes the elapsed

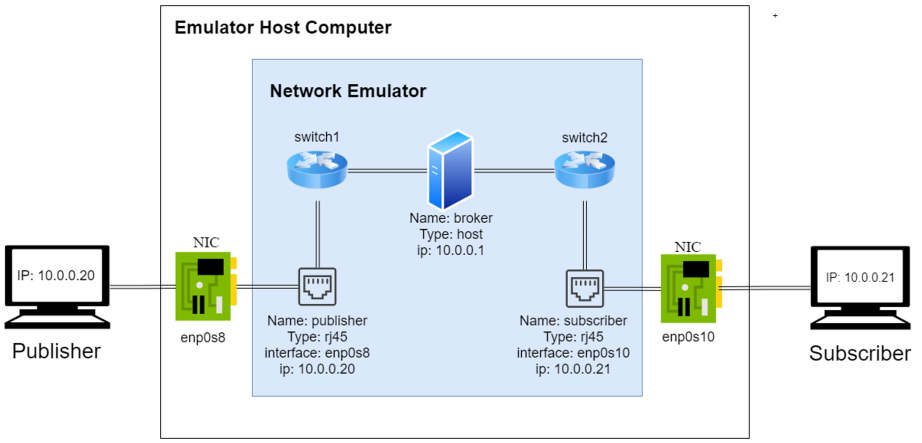

Fig. 9. Testbed setup for the MQTT scenario.

time in seconds. The y-axis indicates the transport duration of each message in milliseconds. The orange 'x' markers represent the transport duration of each message in the real-world scenario, i.e., when communicating with the public *Mosquitto* broker. The blue plot represents the transport duration of the messages in the emulated scenario.

It can be seen that our emulation framework can reconstruct the behavior of the real-work network accurately. The value measured from the emulation is always very close to the value of the real-world network. The minimal deviations can be explained by the delays in the local message broker. From this measurement, we can conclude that our emulation framework can accurately emulate the behavior of a real-world network.

4.3 Threats to Validity

Although the measurements of the evaluation were performed with great care, they might contain some inaccuracies. In measurements where time behavior such as RTT or jitter was measured, the processing time for the response was not included. Therefore, the reported time corresponds to the set delay plus the processing time. This explains the slightly discrepancy between expected value and measured value in Fig. 3 as well as Fig. 7. However, since the processing time of an echo request compared to the set delay is very short, this can be neglected. A similar consideration applies to the measured interference in the emulated network. In both virtual and physical networks, the connection quality is influenced by, e.g., packet queueing. Since our measurements for accuracy were performed on a single notebook and exclusively in a virtual network, this can also be neglected. Finally, in the real-world scenario, the transmission time was measured as the time for transport and processing in the MQTT message broker and emulated as an artificial delay. The local MQTT message broker also

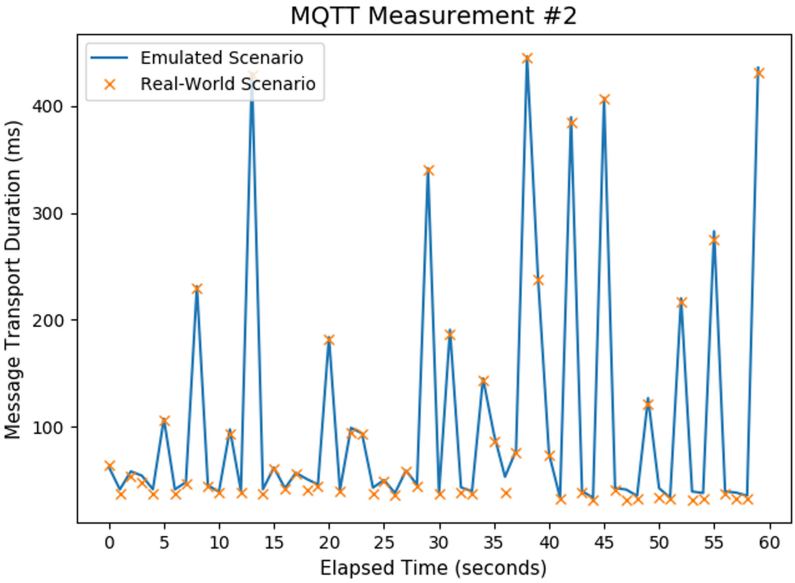

Fig. 10. Comparison of the connection quality with the public *Mosquitto* broker and the emulated connection over a period of 60 s. (Color figure online)

introduces a processing time, and this time is added to the measured transmission time, which explains the small offset in Fig. 10. Since the values for the real network's delay were very high, and the broker's processing time was very short, this aspect was not considered in depth.

5 Conclusion

Network emulators can artificially influence the quality characteristics of network connections. They can be powerful tools in the development of mobile IoT applications to test the application's behavior even if the connection quality of wireless networks varies. However, existing emulators are often complex to configure, require technical knowledge about networks, and are challenging to integrate into automated test environments.

In this paper, we presented a novel network emulation framework that can be easily integrated into software test pipelines. Our emulator does not require extensive expertise so that developers of IoT applications can focus on application development. The emulator supports changes in the interference parameters over time so that a mobile network's temporal behavior, e.g., during a car ride, could be simulated. In the evaluation, the correct influence of the quality characteristics delay, bandwidth, packet loss rate, packet duplication rate, and jitter value, as well as their temporal adjustment, is validated. Furthermore, a measurement of a real-world network with MQTT traffic is imported into the emulator,

and the temporal behavior of the real-world network is simulated against an application.

Although our framework can ideally be used to simulate a temporal behavior, there is a limitation inaccuracy. Our network emulator itself does not directly influence the quality characteristics of the network connection but uses an existing emulator, which uses tools of the Linux kernel like NetEm or Traffic Control. Thus, the accuracy of our emulator depends on the accuracy of the underlying libraries. If a higher accuracy or even real-time is required, other libraries could be included instead.

To further simplify our emulator's operation, future work could extended our framework by a graphical user interface. The elements of CORE, such as the visualization of the network, can be reused, and, for example, a representation of the event queue can be added. Furthermore, temporal behaviors could be emulated by assigning probability distributions to quality characteristics. Probability distributions represent a variation of the values over time, as they are present in some real-world scenarios. In addition to investigating network influences of a single application, the number of applications on virtual instances could be scaled using the emulator. This would enable the investigation of the influence of interference factors in the network within even more complex peer-to-peer networks with a large number of participants.

Acknowledgements. This work was funded by the German Research Foundation (DFG) under grant No. (KO 3445/18-1).

References

1. Ahrenholz, J., Danilov, C., Henderson, T.R., Kim, J.H.: CORE: a real-time network emulator. In: MILCOM 2008–2008 IEEE Military Communications Conference, pp. 1–7, November 2008. https://doi.org/10.1109/MILCOM.2008.4753614
2. Ahrenholz, J.: Comparison of CORE network emulation platforms. In: 2010-Milcom 2010 Military Communications Conference, pp. 166–171. IEEE (2010)
3. Bormann, C., Castellani, A.P., Shelby, Z.: CoAP: an application protocol for billions of tiny internet nodes. IEEE Internet Comput. **2**, 62–67 (2012)
4. Carson, M., Santay, D.: NIST net: a linux-based network emulation tool. ACM SIGCOMM Comput. Commun. Rev. **33**(3), 111–126 (2003)
5. Davoli, L., Protskaya, Y., Veltri, L.: NEMO: a flexible java-based network emulator. In: 2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1–6, September 2018. https://doi.org/10.23919/SOFTCOM.2018.8555769
6. De Oliveira, R.L.S., Schweitzer, C.M., Shinoda, A.A., Prete, L.R.: Using Mininet for emulation and prototyping software-defined networks. In: 2014 IEEE Colombian Conference on Communications and Computing (COLCOM), pp. 1–6. IEEE (2014)
7. Hemminger, S., et al.: Network emulation with NetEm. In: Linux Conf AU, pp. 18–23 (2005)
8. Herrnleben, S., et al.: Towards adaptive car-to-cloud communication. In: Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), Kyoto, Japan (2019)

9. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, p. 19. ACM (2010)
10. Luomala, J., Hakala, I.: Effects of temperature and humidity on radio signal strength in outdoor wireless sensor networks. In: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE (2015)
11. Nussbaum, L., Richard, O.: A comparative study of network link emulators. In: Proceedings of the 2009 Spring Simulation Multiconference, San Diego, CA, USA, pp. 85:1–85:8. SpringSim 2009. Society for Computer Simulation International (2009). http://dl.acm.org/citation.cfm?id=1639809.1639898
12. Porter, M.E., Heppelmann, J.E.: How smart, connected products are transforming competition. Harv. Bus. Rev. **92**(11), 64–88 (2014)
13. Ray, P.P.: A survey on Internet of Things architectures. J. King Saud Univ. Comput. Inf. Sci. **30**(3), 291–319 (2018)
14. Rizzo, L.: Dummynet: a simple approach to the evaluation of network protocols. ACM SIGCOMM Comput. Commun. Rev. **27**(1), 31–41 (1997)
15. Roux, J., Alata, E., Auriol, G., Nicomette, V., Kaâniche, M.: Toward an intrusion detection approach for IoT based on radio communications profiling. In: 2017 13th European Dependable Computing Conference (EDCC), pp. 147–150. IEEE (2017)
16. Sabu, S., Renimol, S., Abhiram, D., Premlet, B.: Effect of rainfall on cellular signal strength: a study on the variation of RSSI at user end of smartphone during rainfall. In: 2017 IEEE Region 10 Symposium (TENSYMP), pp. 1–4. IEEE (2017)
17. Shahid, N., Aneja, S.: Internet of things: vision, application areas and research challenges. In: 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), pp. 583–587. IEEE (2017)
18. Stoller, M.H.R.R.L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale virtualization in the emulab network testbed. In: USENIX Annual Technical Conference, Boston, MA, pp. 255–270 (2008)
19. Ukhurebor, K., Abiodun, C.: Assessment of Building Penetration Loss of Cellular Network Signals at 900 MHz Frequency Bands in Otuoke, Bayelsa State, Nigeria 119 (06 2018)
20. Zec, M., Mikuc, M.: Operating system support for integrated network emulation in imunes. In: Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (1; 2004) (2004)