Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 6540207, 15 pages
http://dx.doi.org/10.1155/2016/6540207

Research Article

EmuStack: An OpenStack-Based DTN Network Emulation Platform (Extended Version)

Haifeng Li, Huachun Zhou, Hongke Zhang, Bohao Feng, and Wenfeng Shi

School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing 100044, China

Correspondence should be addressed to Haifeng Li; 13111026@bjtu.edu.cn

Received 21 August 2016; Accepted 25 October 2016

Academic Editor: Xiaohong Jiang

With the advancement of computing and network virtualization technology, the networking research community shows great interest in network emulation. Compared with network simulation, network emulation can provide more relevant and comprehensive details. In this paper, EmuStack, a large-scale real-time emulation platform for Delay Tolerant Network (DTN), is proposed. EmuStack aims at empowering network emulation to become as simple as network simulation. Based on OpenStack, distributed synchronous emulation modules are developed to enable EmuStack to implement synchronous and dynamic, precise, and real-time network emulation. Meanwhile, the lightweight approach of using Docker container technology and network namespaces allows EmuStack to support a (up to hundreds of nodes) large-scale topology with only several physical nodes. In addition, EmuStack integrates the Linux Traffic Control (TC) tools with OpenStack for managing and emulating the virtual link characteristics which include variable bandwidth, delay, loss, jitter, reordering, and duplication. Finally, experiences with our initial implementation suggest the ability to run and debug experimental network protocol in real time. EmuStack environment would bring qualitative change in network research works.

1. Introduction

The current Internet is based on a number of key assumptions on communication system, including a long-term and stable end-to-end path, small packet loss probability, and short round-trip time. However, many challenging networks (such as sensor/actuator networks and ad hoc networks) cannot satisfy one or more of those assumptions. Excited enough, there have been increasing efforts to support these challenging networks on some special delay and interrupt scenes [1, 2]. In particular, in order to adapt Internet to these challenging environments, Fall proposes Delay Tolerant Networks (DTN) [3]. The key idea of DTN is custody transfer [4] which adopts the hop-by-hop reliable delivery to guarantee the end-to-end reliability. DTN was initially invented for the deep space communication, while currently it has been gradually applied in wireless sensor networks, ad hoc networks, and even satellite networks.

In DTN areas, related research works such as routing and congestion control strategies have obtained many achievements [5, 6] along with a number of DTN implementations such as DTN2, ION, and IBRDTN [7–9]. However, many problems [10, 11] such as security and contact plan design have not been resolved yet.

In order to further study DTN architecture, many experimental platforms have been designed. Koutsogiannis implements a testbed to evaluate space-suitable DTN architectures and protocols with many deep space communication scenarios [12]. The DTN testbed can support about ten nodes experimental topology. Based on the generic-purpose wireless network bench, Beuran designs a testbed named QOMB [13]. QOMB has a good support for emulating a large-scale mobile networks, but it wastes lots of hardware resources since none of virtual computing technology is employed. Thus, QOMB lacks a monitoring system; the experimental fidelity cannot be guaranteed especially in the large-scale scene. Komnios introduces the SPICE testbed [14] for researching space and satellite communication. SPICE is equipped with special hardware and it can accurately emulate the link characteristics between the space and ground stations. However, due to the introduction of professional hardware, SPICE is hard to be imitated by other researchers.

Meanwhile, without using network virtualization technology, the emulation topology of SPICE is fixed and will be changed difficultly.

With the advancement of network and compute virtualization technology, it becomes much easier to design and implement a scalable and flexible emulation platform than before. In this work, EmuStack, a network emulation platform for DTN, is introduced. Our design objective is enabling EmuStack to support a large-scale, real-time, and distributed network emulation and provide synchronous and dynamical precise management for topology and link characteristics. For example, Docker container technology [15] is utilized as the compute virtualization technique into efficiently virtualize several physical emulation nodes into hundreds of virtual emulation nodes. By integrating Linux Traffic Control (TC) utility [16] with OpenStack [17], EmuStack can achieve more fine-grained control of the virtual topology and link characteristics. Meanwhile, OpenStack is composed of various independent modules; thus it possesses a good support for the development of the other functionalities in EmuStack. To improve the performance of EmuStack, many OpenStack subprojects are adopted. An example is Ceilometer [18] which is developed lightly and integrated into EmuStack for ensuring experimental fidelity and monitoring, alarming, and collecting relevant data.

As we have a deeper insight into our initial work [19], in this paper, we further present details of controlling link characteristics and analyze the reason for link rate-limiting difference between the Ethernet device of virtual emulation node and the TAP device of physical emulation node. Moreover, we further introduce EmuStack scalability and performance and discuss their main factors. Additionally, we provide one more DTN experiment to better evaluate and demonstrate the performance of EmuStack.

The remainder of this paper is organized as follows. In Section 2 we introduce the related work. In Sections 3 and 4, we present architectural design, implementation of EmuStack and thoroughly discuss performance of EmuStack. Then we reproduce two published classic DTN experiments and compare and analyze the key experimental results in Section 5. Finally, in Section 6, we conclude this paper along with future works.

2. Related Work

Recently, with the advancement of container virtualization technology, network researchers show their interest in employing container to construct their experimental platforms to support their large-scale topology experiments. Emulab [20] is one of the well-known testbeds using the container virtualization in Linux. Due to the efficiency of container, Emulab possesses a good support for scalability. Although these technologies introduced in Emulab are not the latest now, the design philosophies are still helpful for current researchers to design large-scale test bed. Additionally, Lantz et al. [21] designed Mininet based on container virtualization technique including processes and network namespaces technique. Mininet can support SDN and run on a single computer. Handigol et al. [22] further improved Mininet performance with enhancements to resource provisioning, isolation, and monitoring system. Besides, Handigol replicated a number of previously published experimental results and proved that Linux Container (LXC) [23] technology is not only lightweight but also possesses a good fidelity and performance. In order to perform an in-depth performance evaluation of LXC, Xavier et al. [24] conducted a number of experiments to evaluate various compute virtualization technologies and finally proved that LXC virtualization has a near-native performance on CPU, memory, disk, and network. Therefore, in EmuStack, we employ Docker container (based on LXC) as compute virtualization technology.

OpenStack is an open-source reference framework mainly for developing private and public cloud, which consists of loosely-coupled components that can control hardware pools of compute, network, and storage resources. OpenStack is composed of many different independent modules, and anyone can add additional components into OpenStack to meet their requirements. Therefore, OpenStack is definitely a good choice for developing emulation platform.

3. Architectural Design

This section describes the overall architecture design of EmuStack from the perspective of hardware and software.

3.1. Hardware. Figure 1 shows EmuStack hardware structure (where gray rectangles stand for primary services installed). EmuStack hardware can be composed of only several physical nodes (general-purpose computer). There are two types of physical nodes: network emulator and physical emulation nodes. Network emulator is the core hardware which is a physical node equipped with multiple NICs in EmuStack and it plays multiple roles. It is not only an OpenStack controller node which manages compute and network resources and an OpenStack network node which manages virtual emulation networks, but also an emulation orchestrator which is responsible for creating emulation parameters and orchestrating the whole resources of CPU, memory, and network. In addition, physical emulation node is a compute node of OpenStack, which hosts all virtual emulation nodes and executes the emulation control commands from network emulator.

In EmuStack, there are two types of physical networks, namely, the management network and emulation network. Management network carries management traffic which consists of lightweight control information and usually does not become the determinant of performance. Emulation network transfers emulation data which consumes much bandwidth and would vary greatly with different DTN protocol experiments. Therefore, the physical emulation network possibly becomes the main limitation of EmuStack. For several physical nodes system of EmuStack, adopting the star structure can solve the emulation data traffic bottleneck problem, as shown in the bottom right of Figure 1. In this structure, all emulation NICs of physical emulation nodes are directly connected to those of network emulator. NICs of network emulator are attached to an Open vSwitch bridge, where the “internal” device named after itself is assigned an IP address belonging to the emulation network. In practice, this physical emulation

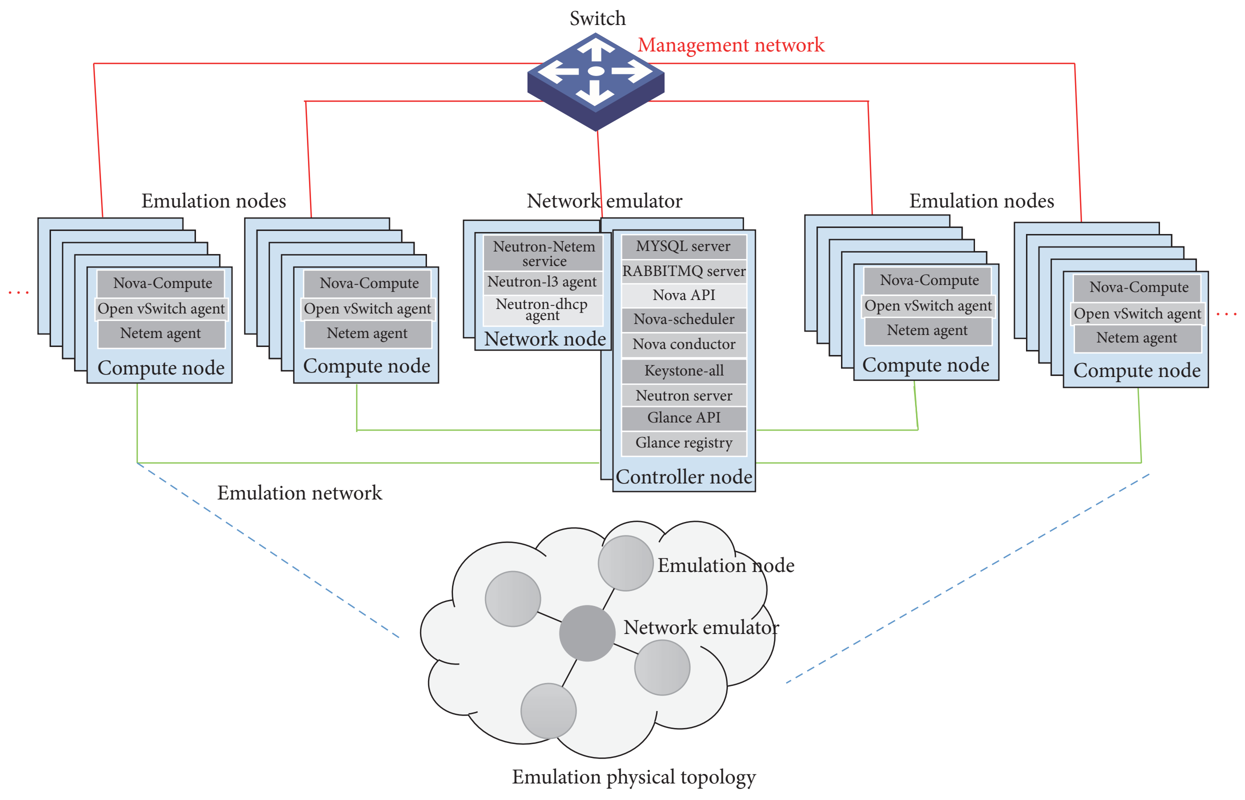


FIGURE 1: EmuStack hardware structure.

network structure can meet most requirements of our DTN research works; however if researchers want to construct the EmuStack system that consists of dozens of physical nodes, this structure would become infeasible since network emulator would not have enough NICs to directly connect to all the emulation NICs of physical emulation nodes. For system with dozens of physical nodes, physical emulation network can employ several physical switches to carry the emulation data as management network does. In this scheme, as the first step, we need to determine which one of physical NICs on network emulator (and physical emulation nodes) will carry management traffic. Then we connect all the remaining NICs of network emulator (and physical emulation nodes) to those physical switches. Those physical switches ports will need to be specially configured to allow trunked or general traffic. Finally, for EmuStack system with hundreds of physical nodes, as a part of the future work, we will extend network emulator to support distributed processing and enable multiple network emulators to exist in EmuStack.

3.2. Software. Figure 2 describes EmuStack software synopsis involving network emulator, physical emulation node, and virtual emulation node. As the key component of EmuStack, network emulator carries many open-source services and customized service extensions. Nova service and the core plugin in Neutron service is attended to initialize virtual emulation nodes and virtual emulation network, respectively. Additionally, these services also have the ability to create, modify, and delete virtual emulation nodes and virtual emulation network. Neutron-Netem service is responsible for generating experimental parameters and data to dynamically control experimental program, topologies, and link characteristics. Meanwhile, in order to provide sufficient fidelity and reduce experimental complexity at the same time, we adopt Telemetry Management (Ceilometer) [18] service to monitor and collect hardware resources and experimental data. In addition, Keystone [25], Horizon [26], and Glance [27] are utilized to provide the support for managing authentication, authorization, service catalog, web interface, and image services. Besides, as a part of the future work, on the basis of OpenStack Heat service, we will develop the orchestrator to more efficiently and flexibly orchestrate the distributed hardware resource management [28]. Most of those services are open-source projects and available in OpenStack; hence we only need to integrate them to meet most EmuStack design requirements. In order to implement synchronous, dynamic, precise, and real-time emulation control service, we design and implement the Neutron-Netem service and Neutron-Netem agent, which will be further discussed in Section 4.

As shown in the bottom left of Figure 2, physical emulation node is regarded as a compute node in OpenStack where virtual emulation nodes are hosted. Physical emulation node runs Nova-Compute service driven by the Docker hypervisor to manage virtual emulation nodes and Open vSwitch agent to execute the managing emulation network commands (including create, modify, and delete function) from network emulator. Open vSwitch agent employs two Open vSwitch (OVS), “OVS for emulation” and “OVS for

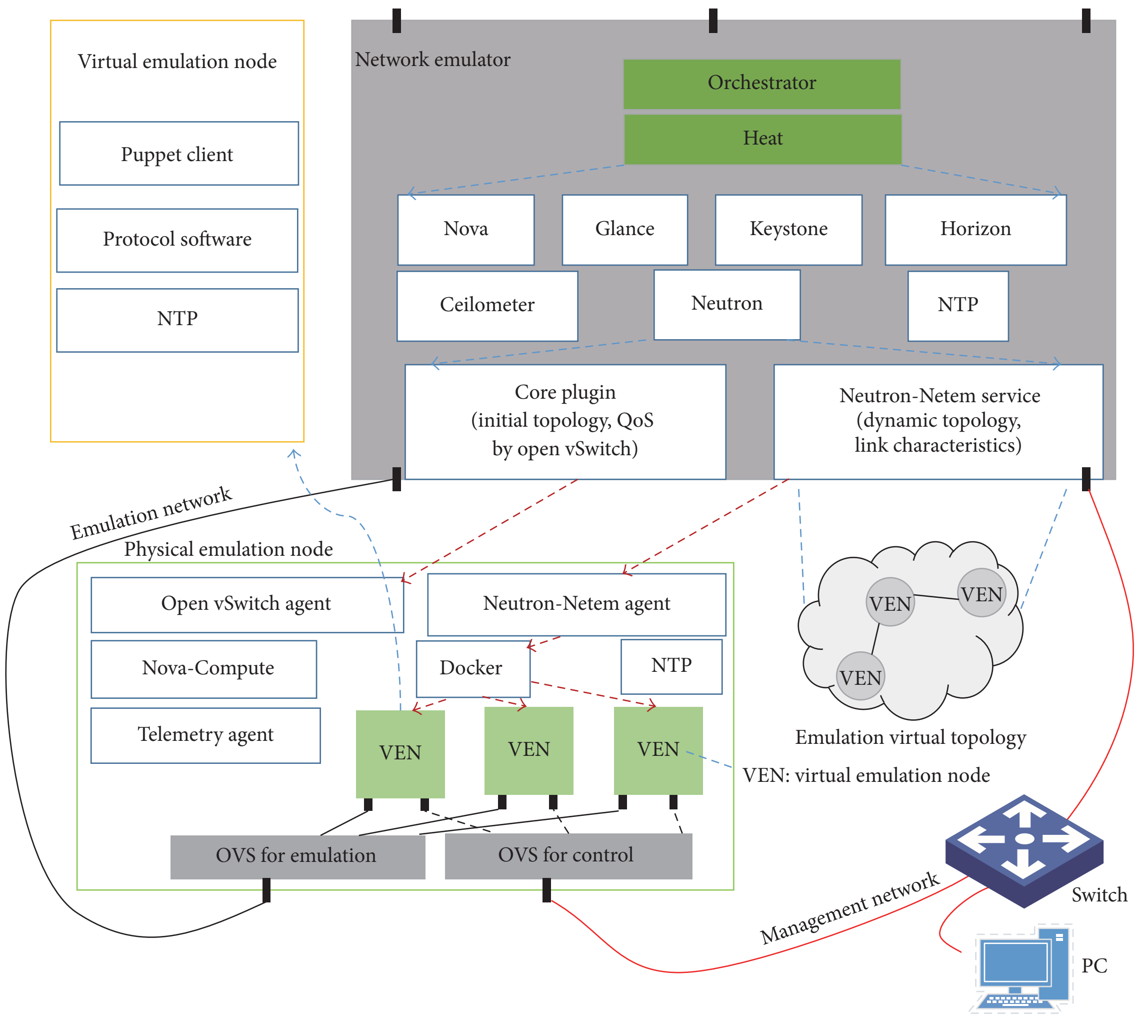


FIGURE 2: Synopsis of EmuStack software.

control" to manage virtual emulation networks and virtual management networks, respectively. Open vSwitch agents manage virtual networks by configuring flow rules on the above two OVS. Moreover, as the agent of Ceilometer service in network emulator, Telemetry Agent is responsible for publishing collected data to network emulator through the management network and creating alarms once collected data breaks the monitoring rules. Finally, Neutron-Netem agent is designed to precisely and dynamically control emulation topologies and link characteristics, which will be further introduced in Section 4.

As shown in the upper left of Figure 2, virtual emulation node (VEN) is a Docker virtual machine which is hosted in physical emulation node. It is spawned from the operating system image where Network Time Protocol (NTP) service, custom network protocol software, and Puppet client service can be installed. In particular, Puppet client service can be utilized by virtual emulation nodes to receive control information from network emulator or physical emulation nodes.

Note that time synchronization is very essential for EmuStack. The DTN bundle protocol depends on absolute time to determine whether received packets are expired. Furthermore, EmuStack must ensure the experimental program in different virtual emulation nodes which can be exactly synchronously executed in the correct time sequences. Therefore, Chrony [29], an implementation of NTP [30], is installed in all nodes to provide the properly synchronizing services. In detail, network emulator is configured to reference accurate time servers while physical and virtual emulation nodes refer to network emulator. In our local area network (LAN) of EmuStack, the time synchronization precision reaches as high as 0.1 milliseconds, which meets the requirements for most emulation experiments.

4. Implementation

This section describes the details of EmuStack core modules (Neutron-Netem service and Neutron-Netem agent). Firstly, in order to sketch the outline of EmuStack implement, EmuStack emulation workflow is described in Section 4.1. Secondly, Sections 4.2, 4.3, and 4.4 present the details of emulation synchronous control, topology control, and customization of

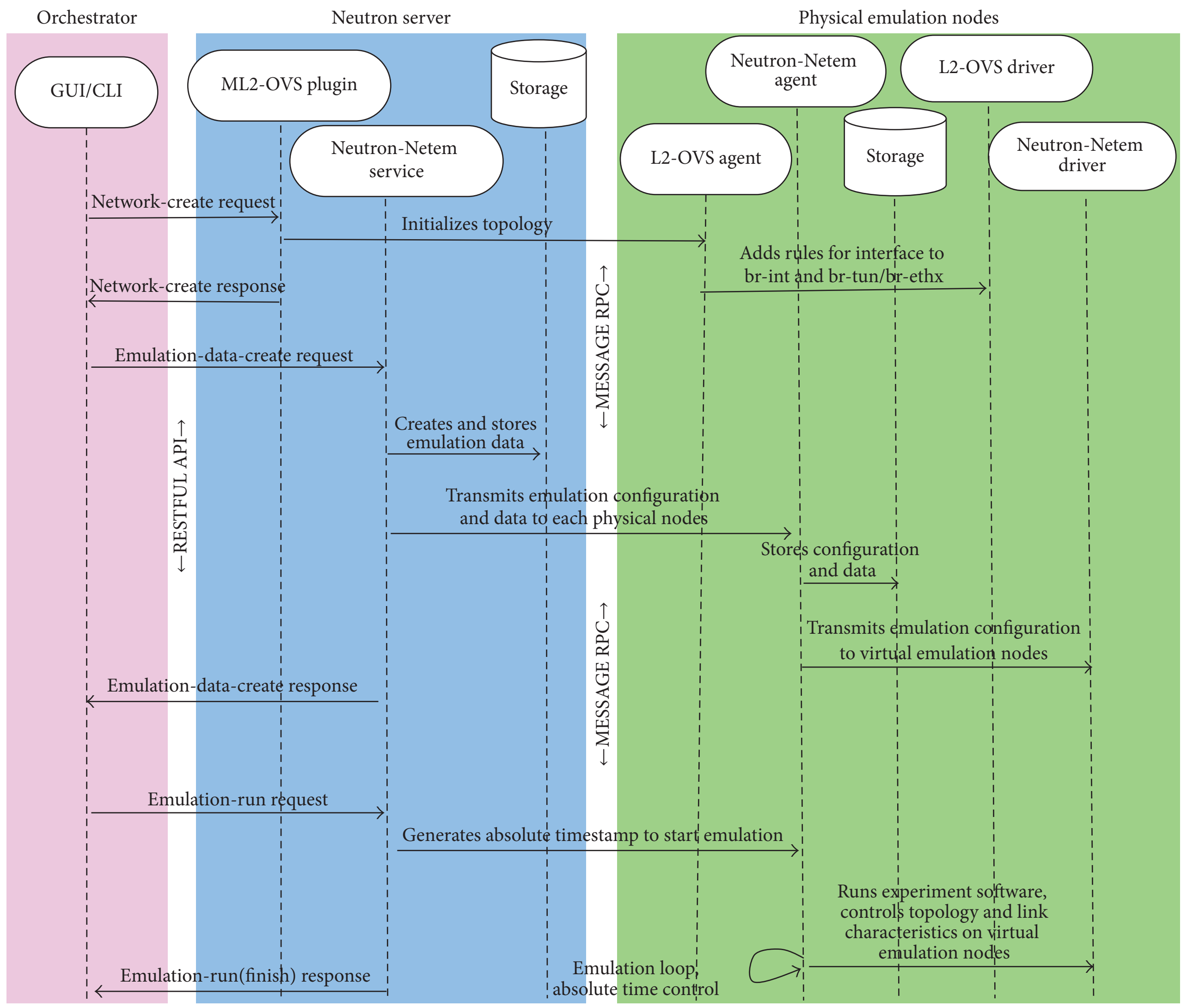

Orchestrator
Neutron server
Physical emulation nodes
GUI/CLI
ML2-OVS plugin
Storage
Neutron-Netem service
Neutron-Netem agent
L2-OVS driver
L2-OVS agent
Storage
Neutron-Netem driver
Network-create request
Initializes topology
Adds rules for interface to br-int and br-tun/br-ethx
Network-create response
Emulation-data-create request
←MESSAGE RPC→
←RESTFUL API→
Creates and stores emulation data
Transmits emulation configuration and data to each physical nodes
Stores configuration and data
Transmits emulation configuration to virtual emulation nodes
Emulation-data-create response
←MESSAGE RPC→
Emulation-run request
Generates absolute timestamp to start emulation
Runs experiment software, controls topology and link characteristics on virtual emulation nodes
Emulation loop, absolute time control
Emulation-run(finish) response

FIGURE 3: Process flow of emulation network.

link characteristics, respectively. Finally, the scalability and performance of EmuStack are discussed in Section 4.5.

4.1. Emulation Workflow. Before the beginning of emulation, we first create a virtual machine image, where special software and shell scripts should be installed to fulfill the specific experimental requirements. For example, you must install an SSH server (or Puppet client) into the image and ensure that it starts up on boot with the correct configuration, or you may install shell scripts to collect some experiment results. Next, we create virtual networks before launching virtual emulation nodes. Virtual networks are composed of two types of networks, namely, management network and emulation network. Management network is Neutron flat network in OpenStack, where all nodes (including virtual, physical emulation nodes, and network emulator) reside on the same network and no VLAN tags are created. Emulation network involves one or more private virtual networks. Moreover, one virtual emulation node could belong to either one or more virtual emulation networks. After creating virtual network, we launch a sufficient number of virtual emulation nodes and initialize virtual networks, right before running the emulation.

Unlike a simulator running in virtual time based on discrete event, EmuStack runs in real time and cannot pause a node's clock to pend for events. For a distributed real-time emulation platform, it is difficult to ensure that every control command can be executed synchronously in the different physical nodes due to the stochastic communication delay and background system load. In order to avoid communication delay, especially the control information transmission delay, EmuStack stores the control information in the local-storage before emulation starts to run.

We can now introduce the process flow of emulation network described in Figure 3. Note that the ML2-OVS plugin, L2-OVS agent, and L2-OVS driver are components of the core plugin in Neutron service. As with OpenStack, EmuStack firstly initializes emulation topology by launching instances together. Secondly, after a successful initialization, the orchestrator requests Neutron-Netem service to run mobility module to create topology and link characteristics data. Meanwhile, in order to support the requirements of those who evaluate the same experimental protocol with different protocol parameters and the same model data, Neutron-Netem service stores the generated model data in the persistent storage. Thirdly, Neutron-Netem service dispatches the emulation data to each agent residing in every physical emulation node. The emulation data is split into different parts for each agent and every agent just only receives its own part and stores it. Relatively, every agent can transmit experimental configuring parameters to virtual emulation nodes by invoking Puppet server API. In each virtual emulation node, Puppet client works in kick-mode and starts to receive configuration (or command) once triggered by Neutron-Netem agent. Finally, after dispatching the emulation data, the orchestrator sends a request to Neutron-Netem service to start emulation; then Neutron-Netem service delivers an absolute timestamp to every agent. Once the staring time is up, agents will start to emulate the experiment, and therefore, the starting timestamp has to be a little (such as sixty seconds) larger than current timestamp, and that extra time is left for Neutron-Netem agents receiving the starting timestamp.

In EmuStack, Neutron-Netem service is organized into separate submodules such as storage and mobility modules. In particular, Neutron-Netem service provides a simple plugin mechanism to enable users to extend different mobility modules. Thus mobility modules can be individually built as researchers' own experimental purposes. The various mobility modules are intended to provide required realistic network emulation environment for different experimental network protocol development. Besides, Neutron-Netem service provides the inheritance mechanism that a mobility module can be developed based on the others. The primary functionality of a mobility module is to create data for dynamically controlling emulation topology and link characteristics. In Section 5, we will employ two mobility modules for DTN large file transmission experiment and the DTN routing protocol comparison experiment of Probabilistic Routing with Epidemic, respectively.

4.2. Synchronous Control. Algorithm 1 describes the synchronous control of Neutron-Netem agent. As shown in lines (2) to (4), Neutron-Netem agents all are asleep and synchronously start emulation once the starting time comes. The time synchronization accuracy depends on the sleeping time SLEEP_TIME and the NTP synchronization precision. Since the NTP synchronization precision is as high as 0.1 milliseconds in our platform environment, the synchronization accuracy is only up to SLEEP_TIME. In fact, the SLEEP_TIME is a trade-off between synchronization precision and system load. In practice, we set SLEEP_TIME to 100 (milliseconds) to satisfy the requirements of most experiment with lightweight CPU load.

With the coming of starting time, Algorithm 1 goes into the outer loop as shown in lines (11) to (23). This outer loop takes advantage of absolute time to control its cycles. As shown in line (13), LOOP_CYCLE (loop cycle) is an important parameter for this loop system. The topology and link characteristics are updated every LOOP_CYCLE. The control operation delay (lines (14) to (16)) plus sleeping time (lines (20) to (22)) is around equal to LOOP_CYCLE. However, due to system load and other unknown factors, the control operation delay may be larger than LOOP_CYCLE by accident; this will lead to synchronous control failure. To help users evaluate the fidelity of the experiments, this failure information all is logged (lines (17) to (19)). Besides, the exceeded time will force future cycles of the loop to reduce the sleeping time; this will enable platform to synchronize again. After the end of outer loop, Neutron-Netem agents kill all experiment processes to get ready for next experiment.

```
(1)  INIT protocol software
(2)  WHILE current_time < starting_time
(3)          sleep SLEEP_TIME miliseconds
(4)  END WHILE
(5)
(6)  INIT topology control
(7)  INIT link characteristics control
(8)  START state collection
(9)
(10) SET counter to 0
(11) WHILE counter < CONTROL_PERIOD
(12)     increment counter
(13)     next_time = start_time + \
                  LOOP_CYCLE * counter
(14)     control topology
(15)     control link characteristics
(16)     control protocol software
(17)     IF current_time – next_time > THRESHOLD
(18)         collect error log
(19)     END IF
(20)     WHILE current_time < next_time
(21)         sleep SLEEP_TIME miliseconds
(22)     END WHILE
(23) END WHILE
(22) KILL all experiment processes
```

Algorithm 1: Synchronous control on Neutron-Netem agent.

4.3. Controlling Topology. Figure 4 provides details on the controlling topology and link characteristics. As shown in the right of Figure 4, Neutron-Netem service delivers the control information to Neutron-Netem agents in advance. According to the received control information, Neutron-Netem agents invoke their driver to dynamically control the emulation experiment once the starting time is coming. In particular, as a part of this control information, the topology control information is described by connection matrix in EmuStack, as shown in Figure 5. In fact, a network topology, no matter how complex it is, can be represented by a connecting relationship between any two nodes. An example for a three nodes topology is shown in Figure 5, where "1" corresponds to connection between two nodes and "0" means disconnection.

In EmuStack, the connection matrix along with time sequences is generated by mobility module. According to connection matrix, Neutron-Netem agents periodically

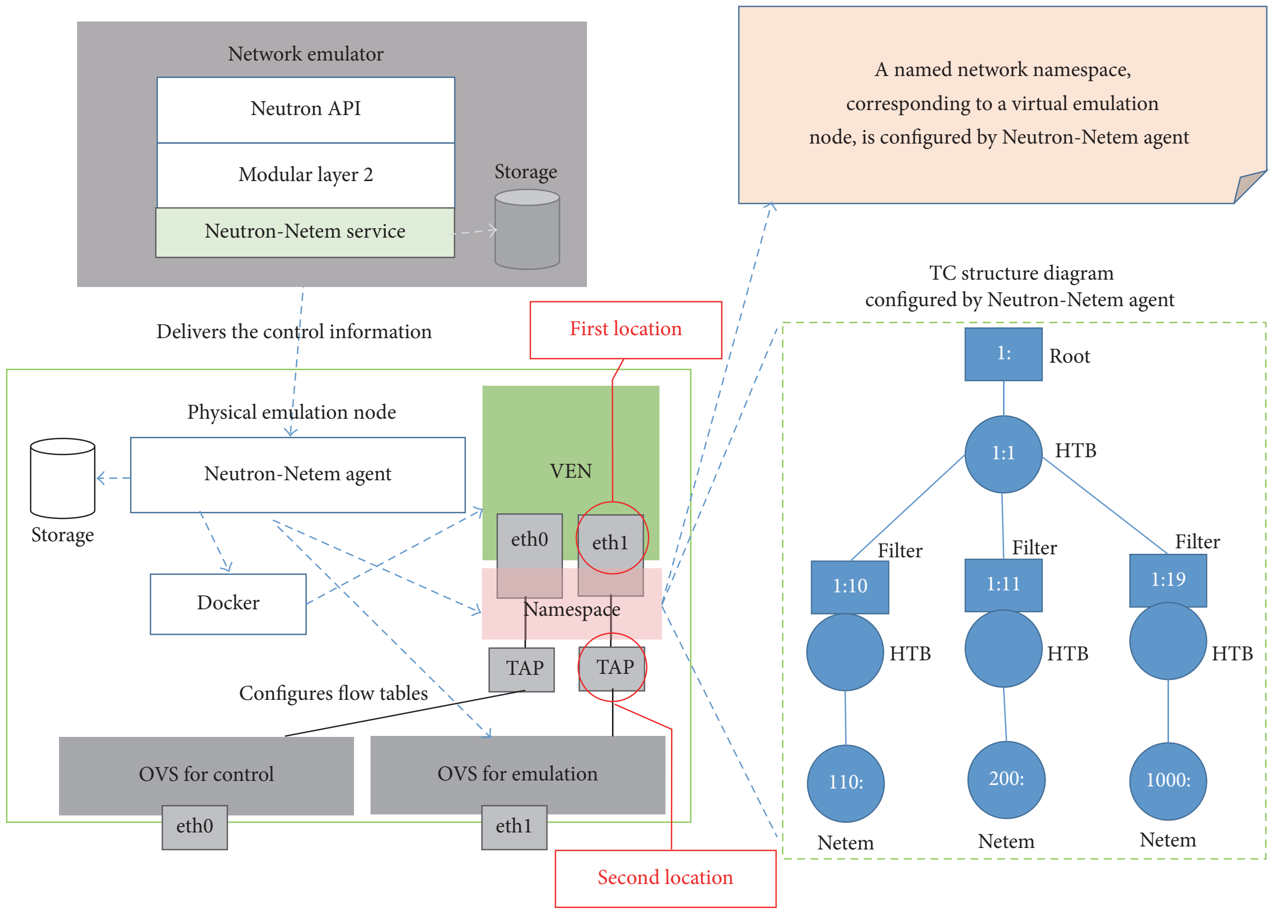
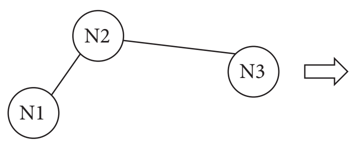


FIGURE 4: Topology and link characteristics control.

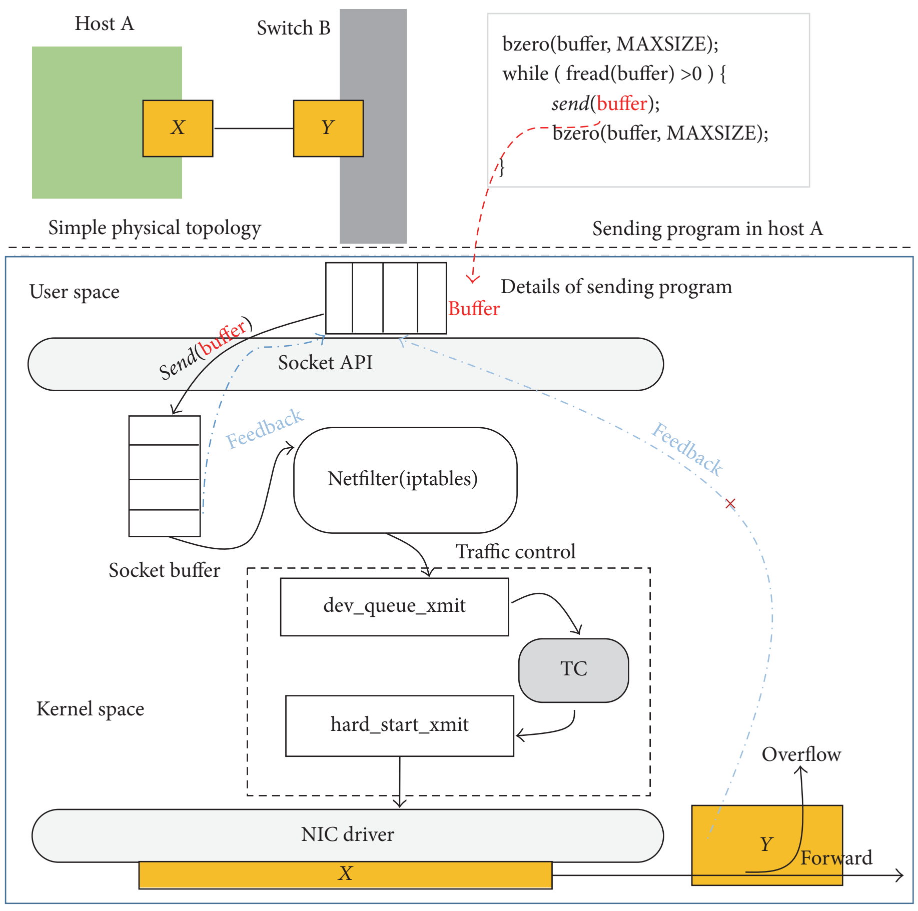


$$\begin{array}{c|ccc} & N1 & N2 & N3 \\ \hline N1 & 1 & 1 & 0 \\ N2 & 1 & 1 & 0 \\ N3 & 1 & 0 & 1 \end{array}$$

FIGURE 5: Simple topology and connection matrix.

invoke their drivers to dynamically change emulation topology during the emulation. There are two ways to dynamically controlling emulation topology: one is based on Open vSwitch and the other is to depend on iptables. Neutron-Netem agents can control virtual emulation topology by configuring flow tables on "OVS for emulation." Managing virtual emulation topology in this way is similar to how Neutron-Open vSwitch agent manages virtual topology in OpenStack, but Neutron-Netem agents can do these more efficiently and quickly. Meanwhile, Neutron-Netem agents can achieve higher synchronous precision since they have already store the emulation control information into local-storage, while Neutron-Open vSwitch needs to get this control information by Remote Procedure Call (RPC) services which take a long-term delay. Additionally, Neutron-Netem agents can dynamically control virtual emulation topology by configuring iptables entries in the special named network namespace. This namespace is corresponding to the virtual emulation node as shown in the top right of Figure 4. In the initial implement of EmuStack, the second way to control topology is implemented in Neutron-Netem agent driver, whose performance will be discussed in Section 4.5. As to the first method, we would take it into consideration in the future work.

4.4. Controlling Link Characteristics. In Linux, system offers a very rich set of tools for traffic control. The Traffic Control, TC, utility is one of the most famous tools. TC is good at shaping link characteristics which include link bandwidth, latency, jitter, packet loss, duplication, and reordering. Besides, it allows users to set queuing disciplines (QDiscs) within network namespace. There are two types of QDiscs in TC: one is classful queuing disciplines which have filters attached to them and allow traffic to be directed to particular classed queues or subqueues; the other is classless queuing disciplines which can be used as primary QDiscs or inside a leaf class of a classful QDiscs. As shown in the bottom right of Figure 4, Hierarchical Token Bucket (HTB) [31] is classful QDiscs, and Netem [32] is classless. In EmuStack,



```
bzero(buffer, MAXSIZE);
while ( fread(buffer) >0 ) {
        send(buffer);
        bzero(buffer, MAXSIZE);
}
```

Figure 6: Rate-limiting difference between two locations.

Neutron-Netem agents use HTB to control link rate, attaching filters to HTB QDiscs to distinguish different virtual emulation links. Meanwhile, Netem is used inside HTB leaf classes to emulate variable delay, loss, reordering, and duplication.

In telecommunications, a link is a communication channel that connects two communicating devices (such as network interfaces); a media access control address (MAC address) is a unique identifier assigned to network interface for communications. Hence, in EmuStack, we can use source-destination MAC addresses to configure filter rules to distinguish different virtual emulation links. In particular, due to the high link asymmetry in most DTN experiments, EmuStack adopts the source-destination ordered pairs to distinguish the difference between uplink and downlink. Meanwhile, we elaborately design control policies since TC QDiscs are only good at shaping outgoing traffic. For example, assuming a link between node A and node B, for A, EmuStack handles A's uplink at one end of the link (on A) and controls A's downlink at the other end of the link (on B); then emulation data can be shaped bilaterally. In addition, EmuStack also can create one or more special intermediate virtual nodes for all virtual emulation nodes of the same physical emulation node to shape their downlink traffic.

We can limit link rate at both locations as shown at the middle of Figure 4. The two locations marked with red circles stand for two different network devices. The first location stands for network interfaces in virtual emulation nodes. All network interfaces are corresponding to those of named network namespaces. The second location represents TAP devices which are paired with those network interfaces mentioned above and attached to Open vSwitch ("OVS for emulation"). Limiting link rate at both locations is feasible, but there are some notable differences. Assuming experimental network protocols (such as UDP) do not have any congestion control algorithms, then any rate-limiting at the second location will lead to a large number of packet loss, but this will not happen at the first location. In most DTN experiments, rate-limiting leading to much packet loss probably is not what we want, and we mostly expect that rate-limiting and packet loss do not interfere with each other.

Figure 6 describes the rate-limiting difference between the two locations with a simple topology and a sending program. In this simple topology, X device is at the first location and Y device is at the second location. Assume that the sending program calls UDP socket API. When sending program sends application data, Linux kernel copies application data from user space buffer to socket buffer. If socket buffer

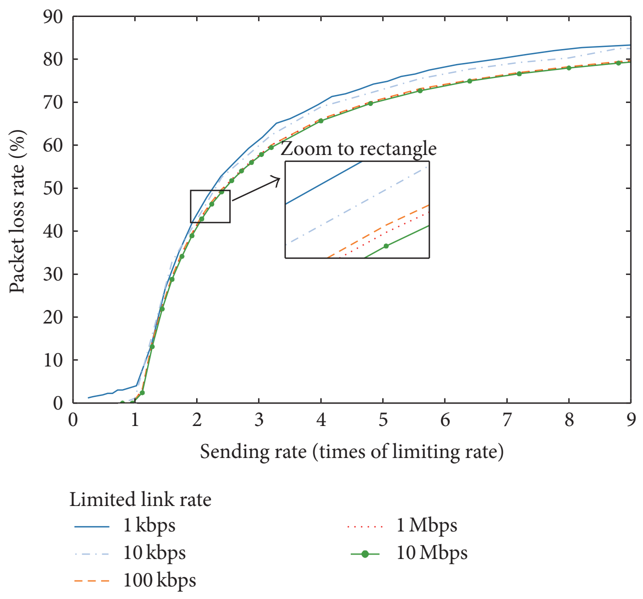


Figure 7: Feature of rate-limiting at the second location.

ever gets full, the blocking socket will put the program in sleep state until the socket buffer has enough space, or the nonblocking socket will return the error "Operation Would Block" immediately. Therefore, no matter which mode (blocking or nonblocking) the socket works in, sending program always receives a "feedback," and this prevents the socket buffer from overflowing as shown in Figure 6. These packets are delivered to QDiscs buffer to shape them and finally transmitted to link by NIC driver. In brief, TC QDiscs consume packets in socket buffer and clear socket buffer, and then sending program can send application data to socket buffer again. As a result, TC indirectly affects the transmission of sending program; a sufficient TC buffer and the feedback mechanism ensure packet loss does not to happen. As to rate-limiting at Y (the second location), since there is not feedback between Y and sending program, Y ingress buffer will overflow and drop most application data as shown in Figure 7.

Figure 7 presents the relationship between packet loss rate and sending rate for rate-limiting at the second location. As expected, when rate-limiting is fixed, the larger the sending rate (times), the more packets the link drops. Meanwhile, setting the sending rate as constant and HTB buffer as default, the larger the rate-limiting, the smaller the packet loss rate. For most DTN scenarios, this is not what we want to see except for testing congestion control algorithms. For example, assuming NIC bandwidth is 90 Mbps and rate-limiting is 10 Mbps, packet loss rate will be up to 80 percent.

In current EmuStack version, we implement all link characteristics control at the first location but only achieve rate-limiting function at the second location by configuring ingress policing rules in Open vSwitch (OVS for emulation). Although rate-limiting at the second location has been implemented in QoS (Quality of Service) plugin of OpenStack Neutron service, it is implemented in centralized model and the synchronous precision is too low. Hence we reimplement the function with the distributed model and obtain that higher synchronous precision is achieved.

4.5. Scalability and Performance. We deploy EmuStack in our experimental platform consisting of nine physical nodes. Each physical node is an identical Dell™ PowerEdge™ R720 2U rack server with one 2.4 GHz Intel Xeon E5-2609 processor (with 4 cores), 10 M of L3 cache per core, 32 GB RAM, and Broadcom 5720 Quad Port 1 GbE BASE-T. In particular, network emulator is integrated with four more Intel EXPI9402PT Dual Port NICs. All management network interfaces of nine physical nodes are interconnected by TP-LINK TL-SF1024D Ethernet switch. All emulation network interfaces of eight physical emulation nodes are linked to those of network emulator. The Ubuntu 14.04 LTS Linux distribution is installed on the all physical nodes and the NetworkManager service is not allowed to start up upon boot, since NetworkManager always repeatedly invokes the useless dhclient program and occupies an amazing number of CPU resources whenever EmuStack launches Docker containers. In addition, operating system kernel version is 3.19.0-31, iptables version is 1.4.21, iprouter2 version is ss131122, and Docker version is 1.10.1. Based on these platform environments, we analyze the emulation scalability and performance as follows.

Compute (CPU), memory (RAM), and network (NIC) are the three chief factors of EmuStack scalability. To make efficient use of CPU and RAM, EmuStack adopts Docker container as virtualization technology instead of kernel-based full virtualization solutions. Docker containers share the same operating system kernel so that they can consume fewer CPU and RAM resources. For example, our platform launches sixty containers on a single machine with about nine percent of CPU usage and ten percent of RAM usage, which serve as virtual emulation nodes which are installed with Ubuntu 14.04 LTS and start up with OpenSSH server and Puppet client. Additionally, in order to ensure that emulation network does not hit network bottleneck easily, EmuStack dispatches compute requests to as few as possible physical emulation nodes for the same experiment, so most virtual emulation nodes are interconnected by the internal bridge (OVS for emulation) and communications between them can consume the least bandwidth of physical emulation network. Meanwhile, all emulation network interfaces on the network emulator are attached to a Linux bridge to improve the bandwidth of physical emulation network. All of these enable EmuStack to support hundreds of nodes with nine physical nodes.

The major factor of EmuStack performance is the updating delay, the time consumed by changing experimental emulation topology or link characteristics for one time. In Algorithm 1, the updating delay determines the LOOP_CYCLE parameter presented on line (13). The minimum LOOP_CYCLE should be no less than the maximum updating delay; otherwise EmuStack will fails to synchronize. In addition, current EmuStack version employs iptables and TC utility to dynamically control the virtual emulation network, respectively. Hence the iptables and TC performance directly impact EmuStack performance and their processing

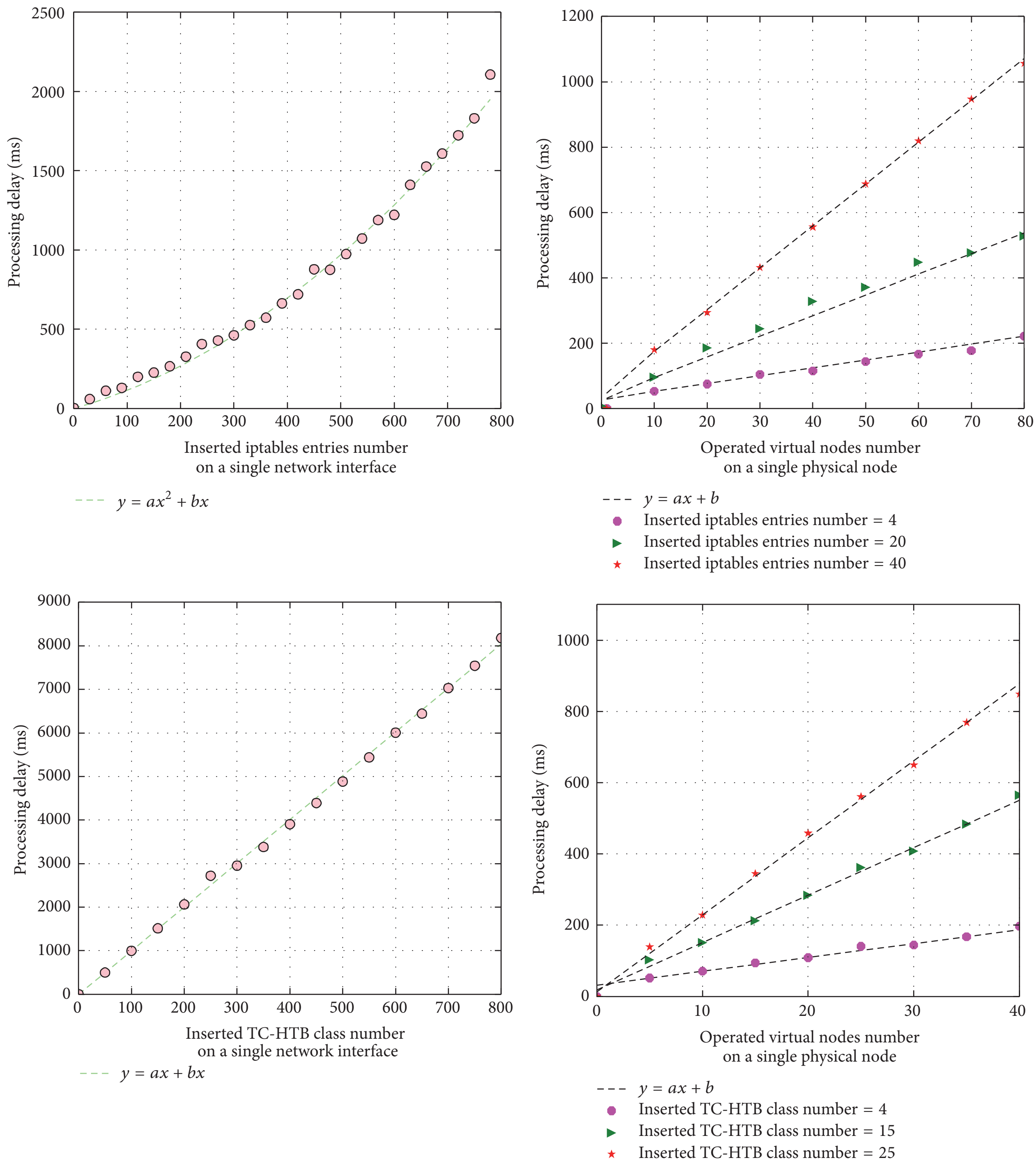


FIGURE 8: The average performance of iptables and TC.

delay has direct impact on the updating delay. The performance of iptables and TC is analyzed as follows.

Figure 8 shows the average performance of iptables and TC, where average performance stands for updating delay trend. The left of Figure 8 describes the performance for operating at a single network interface. Interesting enough, for iptables, the average processing delay can be represented by quadratic function of inserted entries number; for TC-HTB, the relationship between average processing delay and inserted entries number can be well described by a linear function. Hence EmuStack can estimate processing delay with both functions of law. The right shows the performance of concurrently operating multiple virtual nodes in a single physical node. The processing delay of iptables and TC all

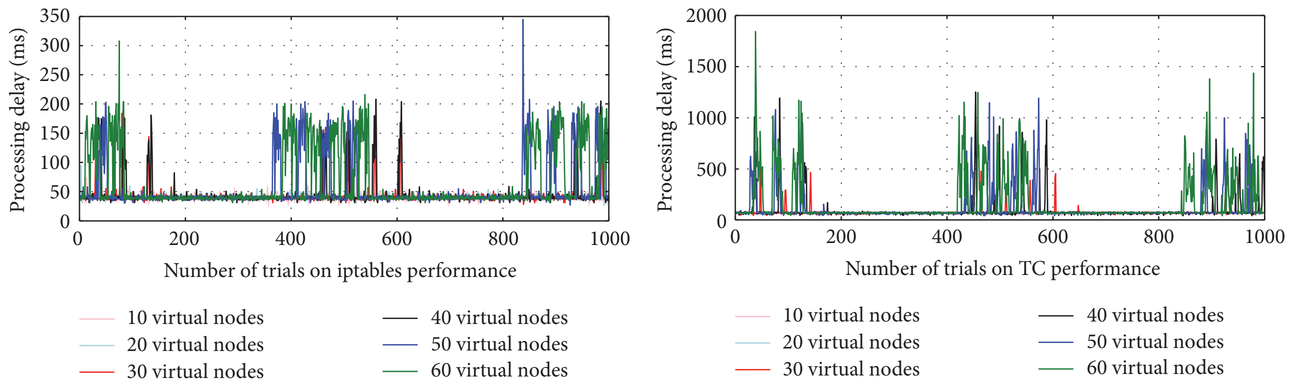


Figure 9: The real-time performance of iptables and TC.

grow linearly when the inserted iptables entries (or TC-HTB class) number is fixed, and this is influenced by serialization, contention, and system load.

Figure 9 shows the real-time performance where the processing delay is the time it takes to insert ten iptables entries (or TC-HTB classes) into a single network interface. The processing delay starts to fluctuate violently with the increasing number of virtual nodes in a single physical node (each virtual node has a network interface which is paired with TAP device in host namespace and linked to Open vSwitch). For example, when there are less than thirty virtual nodes in a single physical node, the processing delay remains stable throughout one thousand trials. However, when the number of virtual nodes increases up to sixty, the fluctuation scope gets wider, with an iptables maximum scope that reaches about 350 milliseconds. TC maximum scope is up to 1800 milliseconds which is five times more than that of iptables. Hence the updating delay of link characteristics (TC processing delay) is probably the most serious limitation in EmuStack.

By analyzing the feature of the real-time performance, we can estimate the maximum updating delay and obtain the minimum LOOP_CYCLE for a specific scale experiment in simple experimental environment (single user). However, it is hard to do that in complex experimental environment (multiuser), and this will raise a lot of complex problems such as virtual nodes orchestration. Because of the limited space, we do not get into details and have a deeper insight into such topic here, leaving this part to be discussed in the future work.

5. Experimental Evaluation

To evaluate and demonstrate EmuStack, this section reproduces key results from two published DTN experiments. One is the DTN large file transmission experiment that applies Low Earth Orbit (LEO) [33, 34] and the other one is the DTN routing protocol comparison experiment of Probabilistic Routing with Epidemic [35, 36]. The goal of the first experiment is to prove that results obtained on EmuStack can match with the results measured on hardware. The goal of the second experiment is to demonstrate that EmuStack can dynamically change a large-scale topology and precisely support a large-scale experiment.

5.1. Large File Transmission Using LEO Satellite. One type of LEO satellite is the remote sensing satellite. Generally, remote sensing satellite transfers a lot of sensing data to the ground station, and these data are usually in a large scale. For example, a single raw picture created by earth observation satellite is usually in a room of hundreds of megabytes (MB) or even more. Unfortunately, only about 10 minutes contact time is allowed for LEO when passing over a ground station in one orbital cycle. Additionally, the LEO transmission rate is low; taking the UK-DMC satellite [34] as an example, there is a downlink of 8.134 Mbps and uplink of 9600 bps. Therefore, it is almost impossible that LEO can transfer a large file to the ground during the period that it passes over a single ground station. Actually, three passes are needed to transfer the complete file to the ground as shown in Figure 10. During each pass, LEO transfers one segment of the total file to Earth Control Center via one Earth Gateway (GW), and once the job of transferring the complete image file is finished, it has been reassembled at the Earth Control Center.

To test whether the results obtained by EmuStack can match those of hardware, we created the experimental topology both in EmuStack and in real hardware as shown in Figure 10. The real hardware environment is built by seven physical nodes, where we utilize TC shell scripts to dynamically control the topology and link characteristics. All the parameters of real hardware that are related to the large file transmission are the same as those of EmuStack, which is described in the following passage.

To ensure reality of experiment process and data, firstly we use Satellite Tool Kit (STK) [37] to model the LEO link characteristics and topology as described in the Table 1. Based on the parameters generated by STK, we write this experimental mobility module in Neutron-Netem service. The mobility module can create the emulation topology and link control information according to the requirements of the large file transmission. Secondly, the OpenStack virtual machine image equipped with the DTN network protocol software ION-3.3.1 [38] is built. ION-3.3.1 uses CFDP [39] program to fragment and reassemble the 258 MB image file from LEO to Earth Control Center. CFDP is configured with 32 kilobytes (kB) bundle [40], 128 kB block of Licklider Transmission Protocol (LTP) [41], and Contact Graph Routing (CGR) [42] protocol. Additionally, it is worth noting that TC

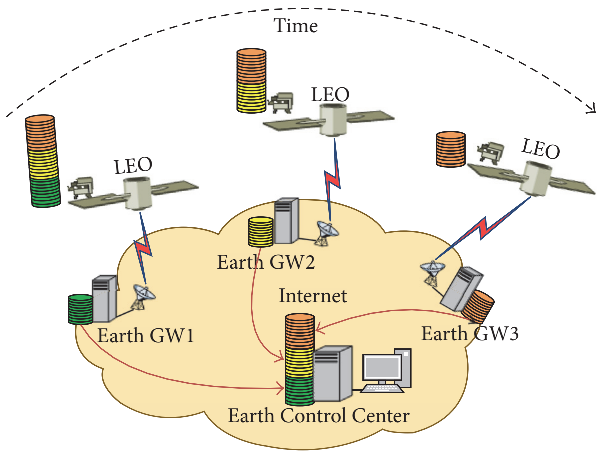


FIGURE 10: LEO block transmission scenario.

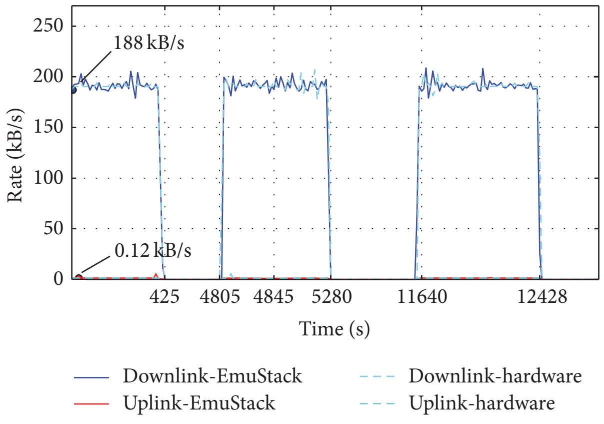


FIGURE 11: Downlink rate and uplink rate in the Earth Control Center.

TABLE 1: LEO block transmission scenario contact plan.

Contact or idle	Duration	Rate	Delay	Jitter	Packet loss rate
LEO → GW1	425 s	200 kB/s	15 ms	2 ms	0.3%
Idle period	73 min				
LEO → GW2	475 s	200 kB/s	16 ms	4 ms	1.1%
Idle period	106 min				
LEO → GW3	788 s	200 kB/s	13 ms	3 ms	0.8%

Netem delay is limited by the frequency (HZ) of the Linux system clock (the tick rate), and the system clock should run at 1000 HZ to allow Netem delays in increment of 1 millisecond.

Figure 11 shows the downlink rate and uplink rate in the Earth Control Center. Due to the effectiveness of ION scheduled, LTP starts a transmission as soon as the link is available. During the whole transmission, LEO first transmits about 79 MB block of image file to Earth Control Center via earth GW1. After about 73 minutes' disconnection, LEO secondly establishes the connection with earth GW2 and transmits another 79 MB block to Earth Control Center. Finally, with 106 minutes' break, LEO transmits the rest of image file to Earth Control Center via earth GW3. In this experiment, experimental results show that downlink unitization is high (about 94%), and the ratio between downlink rate and uplink rate is 1600 : 1. These results prove that DTN protocol family has a good support for intermittent and asymmetric links. Thus EmuStack can be employed to achieve significant results in advance of (or possibly without) setting up a hardware testbed. Meanwhile, since the results of EmuStack closely match with those of the hardware, it indicates that EmuStack has good support for experimental fidelity.

5.2. Comparison of PROPHET Routing with Epidemic. Vahdat and Becker present a routing protocol for DTN called Epidemic routing [36]. This routing protocol allows nodes to exchange summary vectors (an index of their own messages) and request messages which were not owned once they encounter each other. This means messages will spread through the network like an Epidemic, as long as buffer is large enough and the possibility exists. Lindgren et al. propose PROPHET, a Probabilistic Routing Protocol using History of Encounters and Transitivity [35]. The operation of PROPHET is similar to that of Epidemic routing. When two hosts meet, they exchange summary vectors which also contain delivery possibility. Relying on this predictability data, each node calculates the new delivery possibility, which is used to decide which messages to request from the other node.

To evaluate the ability of EmuStack precise control in large-scale experiment, we emulate the simulation experiment described in the PROPHET paper [35] and compare PROPHET with Epidemic in the community scenario. The community scenario consists of a $3000 * 1500$ m area and fifty-six virtual emulation nodes as shown in Figure 12. The area is split into twelve subareas: eleven communities (C1–C11) and one "gathering place (G)." Every community contains five nodes (the same color circles as shown in Figure 12): one fixed node acting as the gateway of the community and four mobile nodes; all nodes treat the community as their home community. The four mobile nodes of every community select a destination, move there with a speed between ten and thirty miles per second, pause there for a moment, and select a new destination and speed. The probabilities of different destinations are chosen according to the current location of mobile nodes. In the experiment, a warm-up period of 500 seconds is used to initialize protocols, and 3000 seconds is used to create and delivery massages, and another 8000 seconds is used for allowing more messages to be delivered.

In order to emulate the above community mobility scene in EmuStack, we develop the community mobility model into a mobility module. Before the beginning of experiment, we first create the experimental virtual image which is equipped with IBRDTN [9]. IBRDTN supports the Epidemic routing and PROPHET routing whose "link_request_interval" parameter is set as 1000 milliseconds since the community model is updated every second. The other model parameters are configured the same as those in PROPHET paper [35].

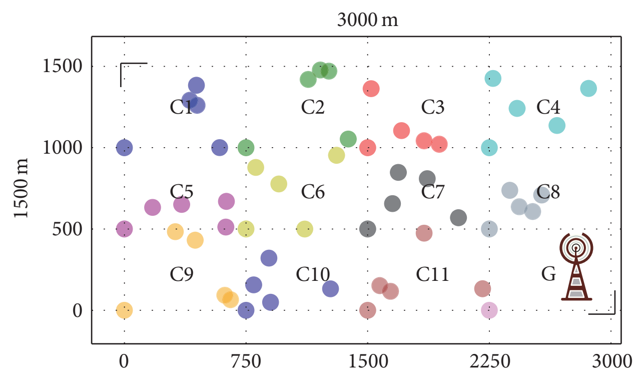


Figure 12: Community mobility model.

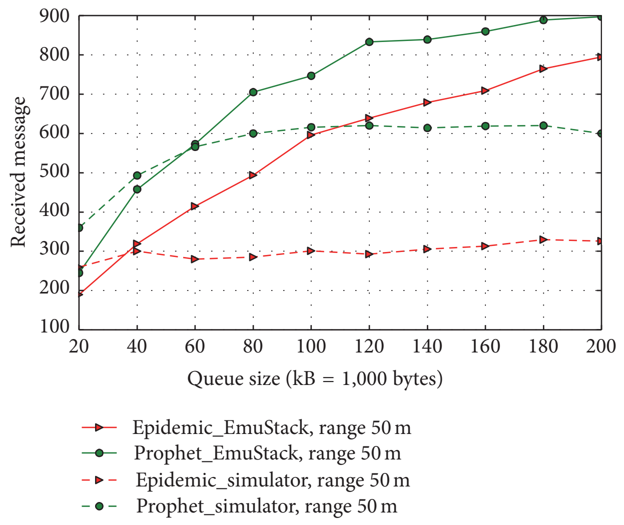


Figure 13: The average delivery rates in community scenario.

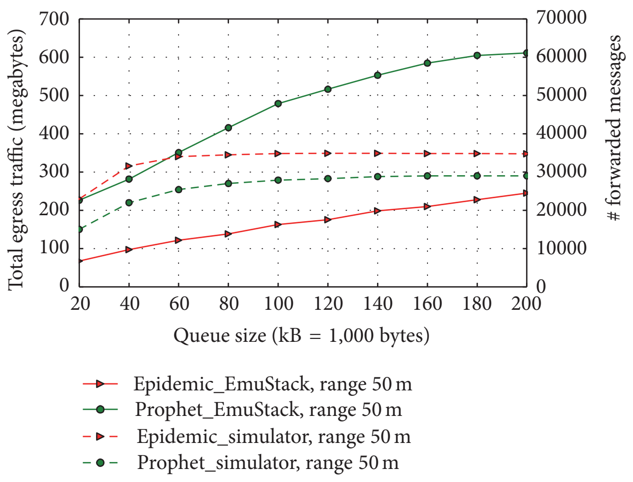


Figure 14: The consumption of the network resources in the community scenario. In the simulator, Lindgren utilizes the number of forwarded messages to indirectly evaluate the consumption; in EmuStack, we employ the value of the total egress traffic to directly measure the consumption.

Note that LOOP_CYCLE is set as one second in the experiment. We attempt to dispatch the fifty-six virtual nodes to different numbers of physical nodes; then EmuStack performs the above experiment for several times with the different configurations of the IBRDTN “limit_storage” parameter (namely, the queue size). At the end of experiments, we check Neutron-Netem agents logs for synchronizing errors. We find that even though all the fifty-six virtual nodes are orchestrated into a single physical node, no synchronizing errors were thrown in EmuStack. This indicates the ability to precisely control large-scale experiment in EmuStack. We further discuss the details of experimental results in the following passage.

Figure 13 shows the average delivery rates in both EmuStack and the simulator described in the PROPHET paper (Hop count = 11). The Epidemic and PROPHET routing protocol show the similar performance in both EmuStack and the simulator. For example, with the increasing size of the queue, the number of messages which eventually reach destination goes up. It is obvious that they can be buffered for longer time and get more opportunities to be delivered successfully, since the larger queue size would enable more messages to be cached and less be dropped. Meanwhile, as shown in Figure 13, the PROPHET routing protocol has a much better performance compared with the Epidemic routing protocol in terms of the delivery rate, and the results of EmuStack matches with those of the simulator. All these results can demonstrate that both PROPHET and Epidemic routing protocols run normally in EmuStack. EmuStack can emulate the large-scale experiments.

Figure 14 presents the consumption of the network resources in the community scenario. In the simulator [35], Lindgren utilizes the number of forwarded messages that occur when nodes encounter each other to indirectly evaluate the consumption; in EmuStack, we employ the value of the total egress traffic to directly measure the consumption. The egress traffic is composed of the forwarded messages and routing overhead; hence it can achieve the more comprehensive evaluation for the consumption of network resources. As described in Figure 14, in EmuStack, PROPHET has a much higher network overhead than Epidemic, as opposed to that in the simulator. This is because the Epidemic routing protocol has been optimized by IBRDTN [9]. IBRDTN already has replaced the summary vectors of the basic Epidemic with the efficient Bloom-Filter mechanism and manages a purge vector as an extension of the Epidemic routing protocol which ensures the bundles delivered successfully to be deleted throughout the network. Therefore Epidemic can consume fewer network traffic than the origin PROPHET described in [35].

Finally, Figure 15 describes the average delivery delay in both EmuStack and the simulator. There are two ways of calculating the average delay. One way is by dividing the sum of the delay of the messages successfully delivered by the number of those (delay 1). The other way is by dividing the sum of the delay of all the messages successfully and unsuccessfully delivered by the number of those (delay 2). The delay of those unsuccessfully delivered messages is defined as

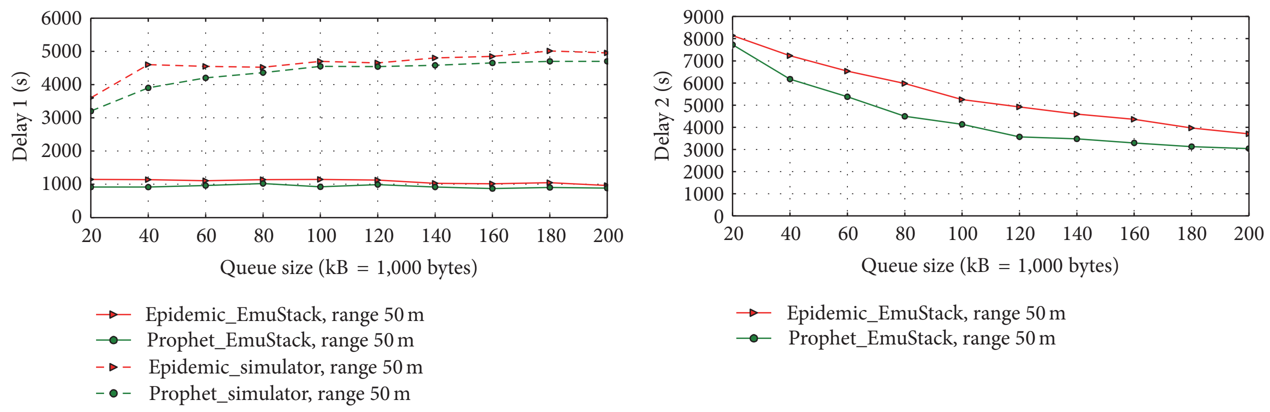


Figure 15: The average delivery delay in community scenario.

the value of subtracting the messages' sending time from the experimental ending time.

The delay 1 is utilized to evaluate the average delay of massages in [35]. As shown in the left of Figure 15, the value of the delay 1 fluctuates back and forth with the increase of queue size. As we all know, the larger queue can shorten the delivery delay of the messages which would be successfully delivered even if the queue is relatively small, and it can also enable messages which would be unsuccessfully delivered to reach their destination nodes, while the value of the delivery delay of these messages become larger compared with the origin zero. Therefore, there are increases and decreases in the delivery delay value, which result in the value of delay 1 fluctuating in small scope.

Due to the above phenomenon, we argue that the first way of calculating delivery delay may be unreasonable. Hence we attempt to evaluate the average delivery delay of the forwarded massages by the second way where we take the delivery delay of unsuccessfully delivered messages into consideration when calculating the average delivery delay. As shown in the right of Figure 15, with the increasing size of the queue, there is an obvious decrease in the average delivery delay (delay 2) for both routing protocols. It is intuitive that the value of delay 2 decreases since larger queue leads to more messages delivered successfully and quickly. In short, no matter which method is used to calculate the average delivery delay, PROPHET always has shorter delivery delay than Epidemic in both EmuStack and the simulator.

As we expected, all the above results demonstrate that EmuStack can reproduce the key results of the large-scale DTN experiment described in [35] and achieve more details of the experimental network protocols than the simulator, which is helpful for us to further improve the design of the experimental network protocol.

6. Conclusion

In this work, we present a real-time, distributed, and scalable emulation platform based on OpenStack for DTN. Firstly, we discuss hardware, software deployment, the design architecture, and implementation. Specially, we present the details of control of link characteristics and topology. Secondly, we analyze the platform scalability and performance. Finally, we evaluate and demonstrate the emulation platform with two classical DTN experiments.

In order to have a thorough evaluation, as a part of the future work, we will create more realistic mobility and link characteristic models to emulate more complex DTN experiments. Meanwhile, we will also evaluate these effects with different virtual computing, virtual network technologies and complex experimental environment (multiuser orchestration).

Competing Interests

The authors declare that there are no competing interests regarding the publication of this paper.

Acknowledgments

This work was supported in part by NSFC of China under Grant no. 61271202, NSAF of China under Grant no. U1530118, National High Technology of China ("863 program") under Grant no. 2015AA015702, and National Basic Research Program of China ("973 program") under Grant no. 2013CB329101.

References

[1] H. Zhang, P. Dong, W. Quan, and B. Hu, "Promoting efficient communications for high-speed railway using smart collaborative networking," *IEEE Wireless Communications*, vol. 22, no. 6, pp. 92–97, 2015.

[2] W. Quan, C. Xu, J. Guan et al., "Social cooperation for information-centric multimedia streaming in highway VANETs," in *Proceedings of the IEEE 15th International Symposium on World of Wireless, Mobile and Multimedia Networks (WoWMoM '14)*, pp. 1–6, IEEE, Sydney, Australia, June 2014.

[3] K. Fall, "A delay-tolerant network architecture for challenged internets," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 27–34, ACM, August 2003.

[4] K. Fall, W. Hong, and S. Madden, "Custody transfer for reliable delivery in delay tolerant networks," IRB-TR 03-030, 2003.

[5] A. P. Silva, S. Burleigh, C. M. Hirata, and K. Obraczka, “A survey on congestion control for delay and disruption tolerant networks,” *Ad Hoc Networks*, vol. 25, pp. 480–494, 2015.

[6] M. Liu, Y. Yang, and Z. Qin, “A survey of routing protocols and simulations in delay-tolerant networks,” in *Wireless Algorithms, Systems, and Applications*, vol. 6843 of *Lecture Notes in Computer Science*, pp. 243–253, Springer, Berlin, Germany, 2011.

[7] Delay Tolerant Networking Research Group, DTNReference Implementation, March 2016, https://sites.google.com/site/dtnres-group/home/code.

[8] S. Burleigh, “Interplanetary overlay network an implementation of the DTN bundle protocol,” in *Proceedings of the 2007 4th Annual IEEE Consumer Communications and Networking Conference (CCNC ’07)*, pp. 222–226, IEEE, Las Vegas, Nev, USA, January 2007.

[9] S. Schildt, J. Morgenroth, W. B. Pöttner et al., “IBRDTN: a lightweight, modular and highly portable Bundle Protocol implementation,” *Electronic Communications of the EASST*, vol. 37, 2011.

[10] M. J. Khabbaz, C. M. Assi, and W. F. Fawaz, “Disruption-tolerant networking: a comprehensive survey on recent developments and persisting challenges,” *IEEE Communications Surveys & Tutorials*, vol. 14, no. 2, pp. 607–640, 2012.

[11] J. A. Fraire and J. M. Finochietto, “Design challenges in contact plans for disruption-tolerant satellite networks,” *IEEE Communications Magazine*, vol. 53, no. 5, pp. 163–169, 2015.

[12] E. Koutsogiannis, S. Diamantopoulos, and V. Tsaoussidis, “A DTN testbed architecture,” in *Proceedings of the International Conference on Ultra Modern Telecommunications & Workshops (ICUMT ’09)*, pp. 1–2, St. Petersburg, Russia, October 2009.

[13] R. Beuran, S. Miwa, and Y. Shinoda, “Network emulation testbed for DTN applications and protocols,” in *Proceedings of the 32nd IEEE Conference on Computer Communications (INFOCOM ’13)*, pp. 3441–3446, Turin, Italy, April 2013.

[14] I. Komnios, I. Alexiadis, N. Bezirgiannidis et al., “SPICE testbed: a DTN testbed for satellite and space communications,” in *Testbeds and Research Infrastructure: Development of Networks and Communities: 9th International ICST Conference, TridentCom 2014, Guangzhou, China, May 5–7, 2014, Revised Selected Papers*, vol. 137 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 205–215, Springer, Berlin, Germany, 2014.

[15] Docker, https://www.Docker.com/.

[16] “TC,” http://www.lartc.org/lartc.html.

[17] OpenStack, http://www.openstack.org/.

[18] “Ceilometer,” https://launchpad.net/ceilometer.

[19] H. Li, H. Zhou, H. Zhang et al., “EmuStack: an openstack-based DTN network emulation platform,” in *Proceedings of the IEEE International Conference on Networking and Network Applications (NaNA ’16)*, pp. 387–392, Hakodate, Japan, July 2016.

[20] M. Hibler, R. Ricci, L. Stoller et al., “Large-scale virtualization in the emulab network testbed,” in *Proceedings of the USENIX Annual Technical Conference (USENIX ’08)*, pp. 113–128, Boston, Mass, USA, June 2008.

[21] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Monterey, Calif, USA, October 2010.

[22] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” in *Proceedings of the 8th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT ’12)*, pp. 253–264, ACM, Nice, France, December 2012.

[23] “LXC,” https://linuxcontainers.org/.

[24] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP ’13)*, pp. 233–240, March 2013.

[25] “Keystone,” https://wiki.openstack.org/wiki/Keystone.

[26] “Horizon,” https://wiki.openstack.org/wiki/Horizon.

[27] “Glance,” https://wiki.openstack.org/wiki/Glance.

[28] F. Song, D. Huang, H. Zhou, H. Zhang, and I. You, “An optimization-based scheme for efficient virtual machine placement,” *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 853–872, 2014.

[29] “Chrony,” http://chrony.tuxfamily.org/.

[30] “NTP,” http://www.ntp.org/.

[31] “HTB,” http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm.

[32] NetEm, http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[33] W. Ivancic, W. M. Eddy, L. Wood et al., “Delay/disruption-tolerant network testing using a LEO satellite,” in *Proceedings of the NASA Earth Science Technology Conference (ESTC ’08)*, University of Maryland, June 2008.

[34] C. Caini and R. Firrincieli, “Application of contact graph routing to LEO satellite DTN communications,” in *Proceedings of the IEEE International Conference on Communications (ICC ’12)*, pp. 3301–3305, IEEE, Ottawa, Canada, June 2012.

[35] A. Lindgren, A. Doria, and O. Schelén, “Probabilistic routing in intermittently connected networks,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 7, no. 3, pp. 19–20, 2003.

[36] A. Vahdat and D. Becker, “Epidemic routing for partially-connected ad hoc networks,” Tech. Rep. CS-20000, Duke University, 2000.

[37] “STK,” http://www.agi.com/products/stk/.

[38] “ION-DTN,” http://sourceforge.net/projects/ion-dtn/.

[39] S. C. Burleigh, “CFDP for interplanetary overlay network,” *NASA. Tech Briefs*, vol. 35, no. 3, p. 36, 2011.

[40] K. L. Scott and S. Burleigh, Bundle protocol specification, 2007.

[41] R. Wang, S. C. Burleigh, P. Parikh, C.-J. Lin, and B. Sun, “Licklider transmission protocol (LTP)-based DTN for cislunar communications,” *IEEE/ACM Transactions on Networking*, vol. 19, no. 2, pp. 359–368, 2011.

[42] S. Burleigh, Contact graph routing, 2010.