MininetE: A Lightweight Emulator for Space Information Networks

Tao Lin(✉), Fa Chen, Kanglian Zhao, Yuan Fang, and Wenfeng Li

School of Electronic Science and Engineering,
Nanjing University, Nanjing 210023, China
mf1723029@smail.nju.edu.cn, zhaokanglian@nju.edu.cn

Abstract. With the continuous development of Space Information Networks (SIN), there is an increasing demand for a cost-effective and high-fidelity tool to carry out research on related networking technologies. Neither the real testbed which is expensive nor the simulators which lack realism can meet our requirements. Therefore, this paper introduces MininetE to emulate space networking allowing relatively high-fidelity experiments that can run on the constrained resources of a single laptop. MininetE enhances the well-known Mininet emulator with adequate isolation and implements dynamic topology control with the original SDN capabilities. The validity of MininetE is verified by results of a Delay-/Disruption-Tolerant Networking (DTN) experiment.

Keywords: Space Information Networks · Emulation · Linux namespace · DTN

1 Introduction

Space Information Networks (SIN) [1] are network systems based on space platforms, e.g. various orbit satellites, stratospheric balloons, space probe etc. In the immediate future, SIN will expand into deep space along with the human exploration of other planets in the solar system. Therefore, it is essential to carry out research on new networking techniques as well as evaluate protocols and algorithms for SIN. Compared with terrestrial network, SIN comprise a wide variety of different environments including long propagation delays, frequent link disruptions, channel-rate asymmetry and dynamic changes of the network topology. To this end, an appropriate network experimentation platform is required to support not only accurate link property modeling for space communications, but also space networking technologies such as Delay-/Disruption-Tolerant Networking (DTN) [2].

The currently available approaches of network-related research have their pros and cons. Large scale testbeds (e.g. Emulab, GENI) give researchers the ability to replicate network experiments at the highest possible level of fidelity, but they are expensive and not always readily available for most researchers. Simulators, such as OPNET and NS-3, use mathematical formulas to create a theoretical and entirely abstract model of a network. They lack realism: They

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
Q. Wu et al. (Eds.): WiSATS 2020, LNICST 357, pp. 48–57, 2021.
https://doi.org/10.1007/978-3-030-69069-4_5

use discrete events to simulate network events such as packet loss and delay in chronological order, and they cannot generate real data flow among the nodes as the realistic network does. Additionally, most emerging networking technologies like DTN, Software Defined Networking (SDN) [3] and Information-Centric Networking (ICN) are not well supported in these simulation tools.

In contrast to simulation, emulation uses actual implementations of protocols and runs real network applications on each virtual node. Emulation aims at accurately reproducing the behavior of a real network and leading to more accurate test results by exploiting as much as possible the same software that would be used on real devices. Emulators can be divided into three major categories by different virtualization methods including full virtualization, para-virtualization and OS-level virtualization [4]. Both full virtualization platforms and para-virtualization platforms provide a high level isolation between virtual nodes but are too heavyweight: the significant consumption of hardware resources limits the emulated network's scale to only a handful of virtual nodes on a single physical machine. By contrast, container-based emulators [5] which leverage OS-level virtualization techniques such as FreeBSD jails, Linux namespaces or OpenVZ are in support of much larger-scale scenarios and are becoming increasingly popular nowadays. This situation can be attributed to containers' ability to consume significantly fewer system resources and instantiate emulated network topologies quickly.

Owing to the features of relatively high fidelity and low overhead offered by container-based emulation, it's an appealing option for testing various network systems and evaluating the performance of protocols. Among these container-based emulators, we eventually choose Mininet [6], an open-source emulator, which permit a variety of network topologies to be emulated on the constrained resources of a single laptop. Mininet provides a straightforward and extensible Python API for arbitrary custom topologies creation and experimentation. Moreover, Mininet integrates well with the Open vSwitch software switch which supports OpenFlow for highly flexible custom routing and Software-Defined Networking. Unfortunately, through our initial experiments we found that Mininet lacked the necessary isolation to support execution of some software (such as ION-DTN and Quagga) which are necessary to emulate SIN. In order to address these issues, this paper presents MininetE, a fork of Mininet extended to support space networking emulation by adding adequate isolation.

The rest of the paper is structured as follows: Sect. 2 presents an overview of Mininet and introduces modifications to the original Mininet. In Sect. 3, the implementation of dynamic topology control is shown in details. In Sect. 4, a sample experiment is presented along with experimental results. At last, a conclusion is given in Sect. 5.

2 MininetE Architecture

The motivation driving us to improve Mininet was that when we performed DTN emulation, we found that Mininet did not support multiple instances of

ION-DTN software on multiple virtual nodes. Afterwards, we set out to start a more in-depth study on why Mininet does not support our needs [7], and then we extend Mininet with a certain components including the following:

- Extensions to the core "mnexec.c" program that provides the container-isolation features.
- Patches and extensions to Mininet Python code, such as Host Class and Net Class.
- Minor modifications to Mininet Python code including definitions and methods of link creation.

In the following subsections, the detailed implementation of MininetE will be introduced in detail.

2.1 Overview of Mininet

Mininet is a widely used container-based emulator when it comes to the experimentation of SDN and the OpenFlow protocol. More than this, by combining lightweight, OS-level virtualization with an extensible CLI and API, Mininet provides a rapid prototyping work flow to customize and create various network topologies.

Figure 1 illustrates a basic network created with Mininet, including virtual hosts, switches, controllers, and links. A host in Mininet is essentially a shell process (e.g. bash) moved into its own network namespace and mount namespace with the unshare system call. Therefore, each host has exclusive network resources, including virtual network interface, network protocol stack and routing tables. Each host has a pipe to the parent Mininet process, mn, which sends commands and monitors output.

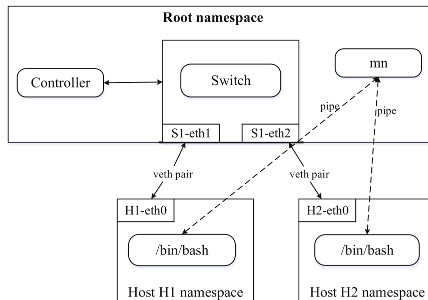


Fig. 1. A Mininet network

Mininet switches are either Open vSwitch instances, Linux bridges, or other types of virtual switch. As for links, a Mininet link is a virtual Ethernet pair (veth pair) which acts like a wire connecting two virtual interfaces. Packets

sent through one interface are delivered to the other. The Linux traffic control program (TC) is used to emulate the communication channel by setting link parameters such as packet loss, delay, and channel bandwidth. Last, controllers are just SDN controllers running on the local server or a remote server.

Up to now, many historical networking experiments has been successfully recreated to replicate results, which proves the high fidelity of Mininet.

2.2 Modifications to Mininet

Due to its efficiency and scalability advantages over full-system virtualization and its inherent support for SDN/OpenFlow as mentioned above, we select Mininet as a basis and extend it into the MininetE tool which is capable of instantiating multiple complex networking software instances (e.g. ION-DTN, Quagga and IPsec).

Our first step in the implementation of MininetE is to add necessary isolation to the "mnexec.c" program that uses the Linux unshare system call to create each virtual host. Apart from the namespace isolation (including Network namespace and Mount namespace) provided by Mininet, MininetE adds support for isolation of the Process Identifier (PID) namespace, Inter-Process Communication (IPC) namespace, and the UNIX Timesharing System (UTS) namespace.

Currently, Linux implements six different types of namespaces [8] as shown in Table 1. The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Containers implemented with Linux namespaces can provide a group of processes with the illusion that they are the only processes in the system.

Table 1. Summary of namespace isolation

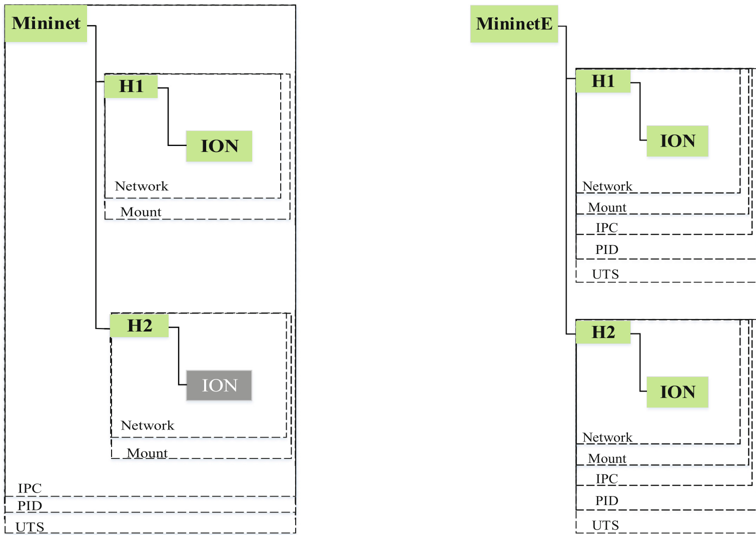
Namespace	Isolate resource	Mininet	MininetE
UTS	Hostname and NIS domain name		✓
IPC	System V IPC, POSIX message queues		✓
PID	Process IDs		✓
User	User and group IDs		*
Mount	Mountpoints and Filesystem	✓	✓
Network	Network devices and Protocol stack	✓	✓

The PID namespace provides processes with an independent set of process IDs (PIDs) from other namespaces. With the introduction of PID namespace, each container can run its own tree of processes and cannot even know of the existence of processes in other PID namespaces. However, processes in the parent PID namespace have a complete view of processes in the child PID namespace, which means that it's convenient to configure and interact with containers by

accurate process IDs in the root namespace. The PID namespace is critical for supporting ION-DTN and some common routing daemons, such as Quagga and IPsec, because there is only one daemon permitted without PID namespace isolation. Moreover, Linux's process control mechanism may interfere or even accidentally kill processes in other containers.

The IPC namespace is usually used along with PID namespace. IPC namespace isolates certain interprocess communication resources, namely, semaphores, message queues, and shared memory. By this way, only processes created in the same IPC namespace are visible to each other. On the contrary, within the original Mininet which lacks the support of IPC namespace isolation, two applications in separate virtual hosts may communicate locally via IPC without going over the emulated network. What's more, the IPC namespace is of significant importance for the ION-DTN software to run successfully, because it's a multi-process program which need to make full use of IPC to transfer and exchange information and make this group of processes work together.

UTS namespace provides the isolation of host name and domain name, so that each container can have an independent host name and domain name, which can be regarded as an independent node on the network instead of a process on the host machine. This can be useful for initialization and configuration scripts of these containers based on their names.



(a) Mininet (b) MininetE

Fig. 2. Comparison of ION in Mininet and MininetE

Aside from the namespace isolation, we make some corresponding modifications to the python code. In order not to destroy the original functions of

Mininet, we decided to extend this part code of the network element classes (e.g. Host Class and Net Class) instead of modifying them directly.

Additionally, when a link is created in Mininet, a pair of two virtual interfaces are instantiated in one of the connected node's network namespace, and one of the interfaces is transferred to the other node's network namespace. Since we isolate PID namespace, this part of code is invalid because these two nodes don't even know that each other exists. To solve this issue, we firstly create the veth pair in root namespace and then move both endpoints into the corresponding namespaces.

After all the work is done, MininetE is able to run complex networking software such as ION- DTN software and Quagga routing daemons. Figure 2 shows the comparison of networking software running on Mininet and MininetE, using ION-DTN as an example. As illustrated in Fig. 2(a), within Mininet, after the first instance of ION-DTN is started in the virtual host H1, H2 fails in initiating another ION-DTN instance because they share the same PID and IPC namespace. The operating system will report an error and inform you that the ION-DTN software has been launched when you try to do so. By contrast, Fig. 2(b) shows the same two node emulated in MininetE. Since both the IPC and PID namespace are isolated, H1 and H2 is invisible to each other, thus both H1 and H2 can successfully run an instance of ION-DTN without mutual interference. Similarly, it is no big deal for MininetE to execute routing daemon such as Quagga and IPsec in each virtual host, which is far beyond the capability of original Mininet.

3 Dynamic Topology Implementation

Although the Linux traffic control program (TC) is used to emulate link properties such as bandwidth, latency, and packet loss, Mininet can only emulate wired links, because link properties are configured and cannot be changed after a link is created. However, other features such as intermittent space links and dynamically changing topology due to satellite motion, which are essential for space networking emulation, are not supported in Mininet. Fortunately, with its inherent support for SDN/OpenFlow, we develop a scheduling method to control link on-off with the help of SDN controller.

Our topology control scheme is based on a centralized emulation architecture, as shown in Fig. 3. Firstly, all virtual hosts are connected with an Open vSwitch to form a star topology. Then the SDN controller controls link-up/down between hosts by controlling the flow table of the switch. The following describes in detail the implementation process from the dynamic topology description information to the scheduled intermittent connectivity.

Figure 4 shows the flow diagram of the topology control implemented by the SDN controller. At the beginning of the scenario, the link information data is read in advance and stored in the controller cache. Then controller issue the Address Resolution Protocol (ARP) flood flow table. The ARP flood flow table enables all ARP packets passing through the switch to be flooded, so that all nodes can obtain the IP addresses and MAC addresses of neighboring nodes.

Next, the connection flow tables of the static links are issued, which survive the entire emulation period. Meanwhile, the flow table of dynamic link is issued by a timer within the SDN controller. At the time of starting the emulation, a multi-threaded timer is started, and the time points of link on/off are added to the timer structure. When the timer ends, the link connection function or the link disconnection function will be executed to deliver the corresponding connected flow table or delete the corresponding flow table.

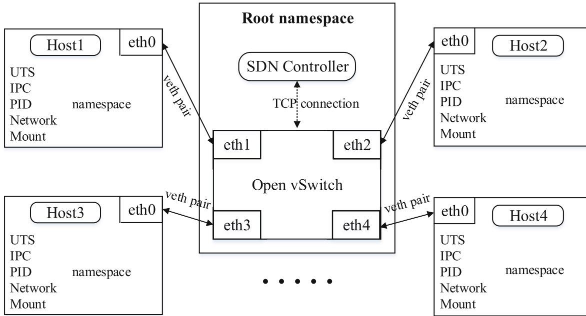


Fig. 3. Centralized emulation architecture

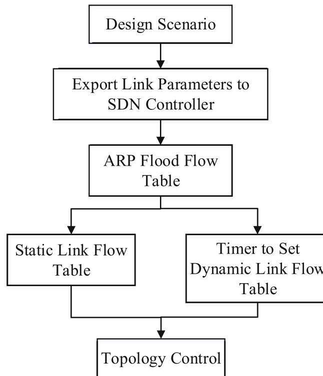


Fig. 4. Dynamic topology control

4 Experimental Validation

MininetE could meet requirements of emulating various networks. It is an ideal emulation tool to provide a credible approach to test and evaluate space networking in a single laptop. In the following subsections, we briefly present a sample work, along with experimental results obtained.

4.1 Specific Scenario

In order to evaluate the reliability of MininetE, we reproduce a DTN experiment presented in [9] with MininetE. The topology of this scenario is shown in Fig. 5. It consists of four DTN nodes: a Moon lander, a satellite orbiting the Moon (SAT), an auxiliary terrestrial Gateway Station (GW), Mission Control Centre (MCC). Intermittent space links are denoted by dotted lines, terrestrial wired links by continuous ones. The purpose of the experiment is to assess the ability of DTN in the deep space communication environment.

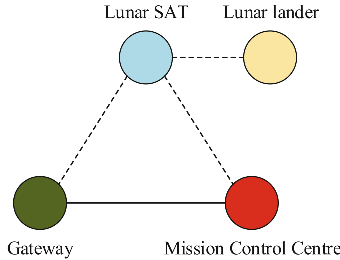


Fig. 5. Topology of the Moon to Earth scenario

Table 2. Contact plan

Link	Contact	Start-Stop time(s)	Speed(downlink)	Latency(s)
Lander-SAT	1	20–40	128 kbit/s	0
	2	100–120	128 kbit/s	0
SAT-GW	1	70–80	1 Mbit/s	1.3
	2	160–170	1Mbit/s	1.3
SAT-MCC		150–180	1 Mbit/s	1.3
GW-MCC		0–180	10 Mbit/s	0

The contact plan and link characteristics are summarized in Table 2. Note that the links between Moon and Earth have a propagation delay of 1.3 s, while the delay of other links is negligible. Besides, losses have been assumed to be negligible on all links.

4.2 Experimental Results

In the scenario summarized above we transfer ten bundles of 50 kB from the Moon lander to the MCC. Figure 6 presents a comparison between some results presented in [9] and those obtained with the MininetE. At the beginning, ten bundles are generated and taken into custody on the Lander. When the first

Lander-SAT contact starts (at 20 s), the first six bundles are transferred to SAT and taken in custody. Then they are transferred to GW when the SAT-GW contact opens (at 70 s), as the GW-MCC link is continuous, they are immediately delivered to the MCC. The rest four bundles are transferred to SAT during the second Lander-SAT contact (begins at 100 s) and taken into custody as before. Finally, they are directly delivered to MCC when the SAT-MCC contact first opens (at 150 s).

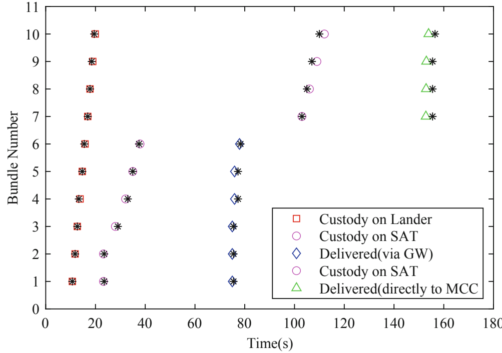


Fig. 6. Bundle transfer from Lander to MCC (Markers: from [9]; x-crosses: MininetE)

5 Conclusion

With MininetE, it is able to run multiple instances of complex network emulation software on each virtual host separately. This, together with our method of dynamic topology control, provides an approach to emulate SIN with accurate link characteristics. Emulation performances prove that MininetE is capable of testing DTN networks. More experiments will be performed in the future to prove that MininetE can meet requirements of emulating various networking techniques.

References

1. Bai, L., de Cola, T., Yu, Q., Zhang, W.: Space information networks. IEEE Wirel. Commun. **26**(2), 8–9 (2019)
2. Burleigh, S., et al.: Delay-tolerant networking: an approach to interplanetary internet. IEEE Commun. Mag. **41**(6), 128–136 (2003)
3. Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2014)
4. Salopek, D., Vasić, V., Zec, M., Mikuc, M., Vašarević, M., Končar, V.: A network testbed for commercial telecommunications product testing. In: 2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 372–377. IEEE (2014)

5. Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., McKeown, N.: Reproducible network experiments using container-based emulation. In: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, pp. 253–264 (2012)
6. 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, pp. 1–6 (2010)
7. Barnes, J.L., Clark, G.J., Eddy, W.: Cogswel: A network emulator for cognitive space networks. In: 34th AIAA International Communications Satellite Systems Conference, p. 5763 (2016)
8. Biederman, E.W., Networx, L.: Multiple instances of the global linux namespaces. In: Proceedings of the Linux Symposium, vol. 1, pp. 101–112. Citeseer (2006)
9. Caini, C., Fiore, V.: Moon to earth DTN communications through lunar relay satellites. In: 2012 6th Advanced Satellite Multimedia Systems Conference (ASMS) and 12th Signal Processing for Space Communications Workshop (SPSC), pp. 89–95. IEEE (2012)