SoftwareX 10 (2019) 100248

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

NEMO: A flexible and highly scalable network EMulatOr

Luca Veltri [a,*], Luca Davoli [a], Riccardo Pecori [b], Armando Vannucci [a], Francesco Zanichelli [a]

[a] Department of Engineering and Architecture, University of Parma, Italy
[b] SMARTEST Research Centre, eCAMPUS University, Novedrate (CO), Italy

ARTICLE INFO

Article history:
Received 18 January 2019
Received in revised form 6 May 2019
Accepted 6 May 2019

Keywords:
Network emulation
Protocol stack
Java

ABSTRACT

Evaluating novel applications and protocols in realistic scenarios has always been a very important task for all stakeholders working in the networking field. Network emulation, being a trade-off between actual deployment and simulations, represents a very powerful solution to this issue, providing a working network platform without requiring the actual deployment of all network components. We present NEMO, a flexible and scalable Java-based network emulator, which can be used to emulate either only a single link, a portion of a network, or an entire network. NEMO is able to work in both real and virtual time, depending on the tested scenarios and goals, and it can be run as either a stand-alone instance on a single machine, or distributed among different network-connected machines, leading to distributed and highly scalable emulation infrastructures. Among different features, NEMO is also capable of virtualizing the execution of third-party Java applications by running them on top of virtual nodes, possibly attached to an emulated or external network.

Code metadata

Current code version	Version 1.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_12
Legal Code License	Apache 2.0
Code versioning system used	git
Software code languages, tools, and services used	Java 1.8 or greater
Compilation requirements, operating environments & dependencies	
Link to developer documentation/manual	http://netsec.unipr.it/project/nemo
Support email for questions	luca.veltri@unipr.it

1. Motivation and significance

Network simulators and network emulators are important tools used by network architects to investigate issues and performance before actually deploying novel applications or protocols. Although they share many similarities, network simulators may have some drawbacks since they depend on the particular model of the network and on its own accuracy. On the other hand, network emulators allow their users to test real systems within network platforms involving only some network elements, possibly exhibiting the same real characteristics like bandwidth, latency, or degraded network conditions.

There is a wide range of network emulators that have been developed for specific network scenarios. For example, the Mobile Network Emulator (MNE) [1] focuses on IPv4 and IPv6 MANET networks. Likewise, QOMET is a wireless network emulator proposed in [2] using a multi-stage approach to convert a real-world scenario into a sequence of network-condition descriptors [3]. A very simple network emulator written in Java was proposed in [4], in which the behavior of a network link (considering its bandwidth, delay, and loss) is emulated by properly re-implementing the socket classes used by the tested applications. NCTUns [5] was an important tool providing both simulation and emulation capabilities and one of the first software exploiting kernel re-entering principles, i.e., packets were really processed by a Linux kernel. More recent network emulators use the Linux

* Corresponding author.
E-mail address: luca.veltri@unipr.it (L. Veltri).

https://doi.org/10.1016/j.softx.2019.100248

network namespaces and file system as containerization mechanisms for creating virtual nodes and networks. For example CORE (Common Open Research Emulator)[1] is a tool for emulating networks on one or more machines, based on Linux network namespaces. IMUNES (Integrated Multi-protocol Network Emulator/Simulator)[2] uses Docker containers and Open vSwitch. Mininet,[3] the main network emulator used in experiments with SDN, also exploits Linux network namespaces for creating virtual nodes.

Despite the availability of several network emulators, they often lack in usage flexibility: they could be too specific or they could be difficult to modify. Therefore we developed a new network emulator, called NEMO, whose design intends to maximize simplicity, re-usability, and flexibility.

Unlike other network emulators using Linux virtualization or containerization technologies, NEMO is independent of the underlying OS, providing a complete fresh re-implementation of the IP stack. A core of general network elements, a layered architecture, together with a large set of built-in protocols, make NEMO a general purpose emulator that can be used to experiment with standard IP-based networks, and to develop new components or protocols. It runs on any platform and OS that supports Java (Linux, Windows, macOS, Android, etc.), without any platform-specific settings, guaranteeing complete replication of the experiments. Emulations can be performed using either virtual or real time. In the former case the actual execution time is independent of the emulated scenario (long executions last only the time required by the CPU), and only the actual emulated events are taken into account with nanosecond precision, resulting in a very high accuracy of time evaluation. On the other hand, using a real clock allows users to set up networked scenarios where the emulated network or nodes interact seamlessly with both external applications and network elements. External systems can be attached through proper connectors, therefore allowing users to set up hybrid networks, including both internal and external components and applications. The emulated network can be run on either a single machine or a set of machines, in a larger distributed emulation architecture.

Due to its simple internal structure, NEMO scales very well, as demonstrated in [6]. It can successfully emulate very large networks, composed of up to millions of nodes, on the same machine. NEMO has already been employed for different networking analysis and tests: traffic engineering through IPv6 Segment Routing [7,8], Network Function Virtualization and Service Chaining [9], novel anonymity protocols [10], and VoIP networks. Furthermore, the flexibility of the Java language and the possibility to work at user-space greatly simplify both development and testing phases.

NEMO is easily usable since it only requires a Java VM installed on the host machine. It can be downloaded from its website[4] as a single ZIP archive containing all source code, scripts, and documentation, and can be simply launched at command line. Some built-in examples are included in the archive and described in the online tutorial.

2. Software description

2.1. Software architecture

NEMO is highly structured around many modules making the change or creation of new components very simple and intuitive. It relies on a layered architecture, as shown in Fig. 1.

The core of NEMO is composed of a set of basic network components such as `Packet`, `Address`, `NetAddress`, `NetInterface`, `Node`, `Link`, `RoutingFunction`. A `NetInterface` may wrap physical network cards or virtual interfaces at different layers. Two or more nodes can be connected by attaching each node to a common `DataLink` using a `LinkInterface`. The relationships among the set of basic components described above are shown in Fig. 2.

Above the basic network components, a large set of built-in protocols are present. In particular, the entire TCP/IP protocol stack has been re-implemented in Java (IPv4, ARP, ICMP, IPv6, ICMPv6, UDP, TCP), together with some Data-Link protocols (like Ethernet, SLIP, PPP, TUN), configuration and routing protocols (DHCP, SPF). Nevertheless, other protocols and data formats can be easily imported from other open-source projects, some of which have been already integrated and tested. On top of TCP and UDP there is also a brand new implementation of the standard Java socket API (*java.net*), based on the NEMO TCP/IP protocol stack. This simplifies the development of upper layer protocols, benefiting from a well-known programming API, besides allowing the integration of any third-party Java application without requiring code changes.

Three different types of *connectors*, by means of JNI libraries, provide the fundamental interaction with external systems. The *tuntap library* is the main method and provides a simple and transparent interface to the underlying OS networking by means of standard TUN or TAP interfaces that are layer-three and layer-two virtual interfaces that can be set-up for connecting an application to the OS networking (see Section 3.2). The *rawsocket library* allows direct access to the OS standard POSIX sockets and to the host network interfaces. Finally the *netfilter library*, in case of a Linux-based OS, allows the user to connect a NEMO element to the *netfilter* interface of the Linux kernel; this, in turn, allows the user to intercept and process any packet passing through the underlying machine.

The built-in *tunnel hub* component allows the interconnection of different NEMO networks or nodes running on different machines, possibly attached to networks behind a NAT and/or a firewall.

2.2. Software functionalities

This subsection details some of the main functionalities of NEMO.

2.2.1. Creating new protocols

The independence of the underlying system and network stack, together with a protocol abstraction and a layered structure, allows for easily changing current network elements, and adding new protocols. New protocols can be added either by importing them from external projects (concerning upper layer protocols, such an action is simplified by a NEMO transport layer providing the same standard Java networking `java.net` API), or by developing them starting from the available basic components and the already implemented protocols.

2.2.2. Building network topologies

NEMO allows the user to form any network topology by creating nodes and their attached links directly.

In addition, there are also some tools allowing the user to automatically create some standard topologies such as linear, square (Manhattan), star, or three networks, or a combination thereof.

Concerning node configuration, it can be performed either manually or automatically. For automatic IP node configuration, it can be done easily by using `nextAddressPrefix()` and

[1] https://www.nrl.navy.mil/itd/ncs/products/core.
[2] http://www.imunes.net.
[3] http://www.mininet.org.
[4] http://netsec.unipr.it/project/nemo/.

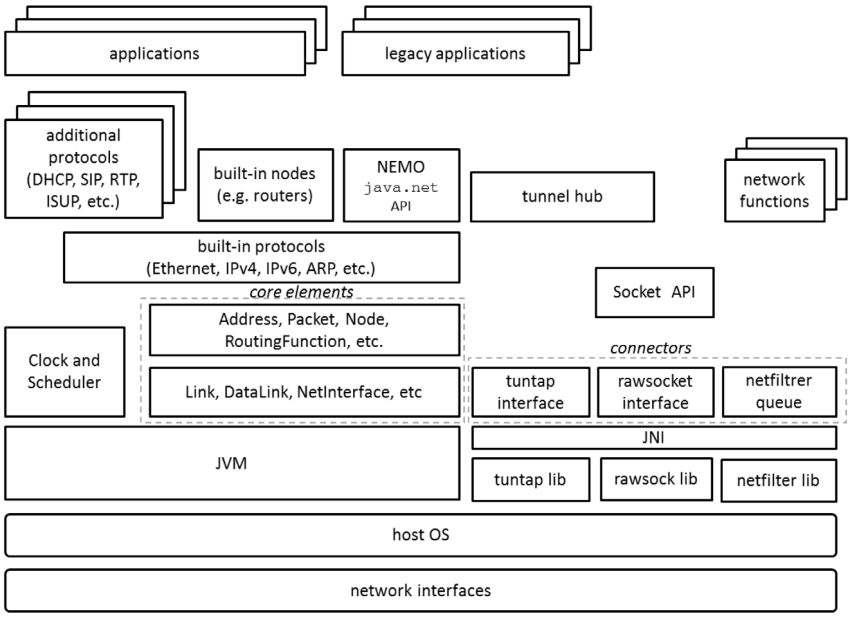


Fig. 1. NEMO software architecture including core objects, built-in protocols, network elements, connectors, additional protocols and applications.

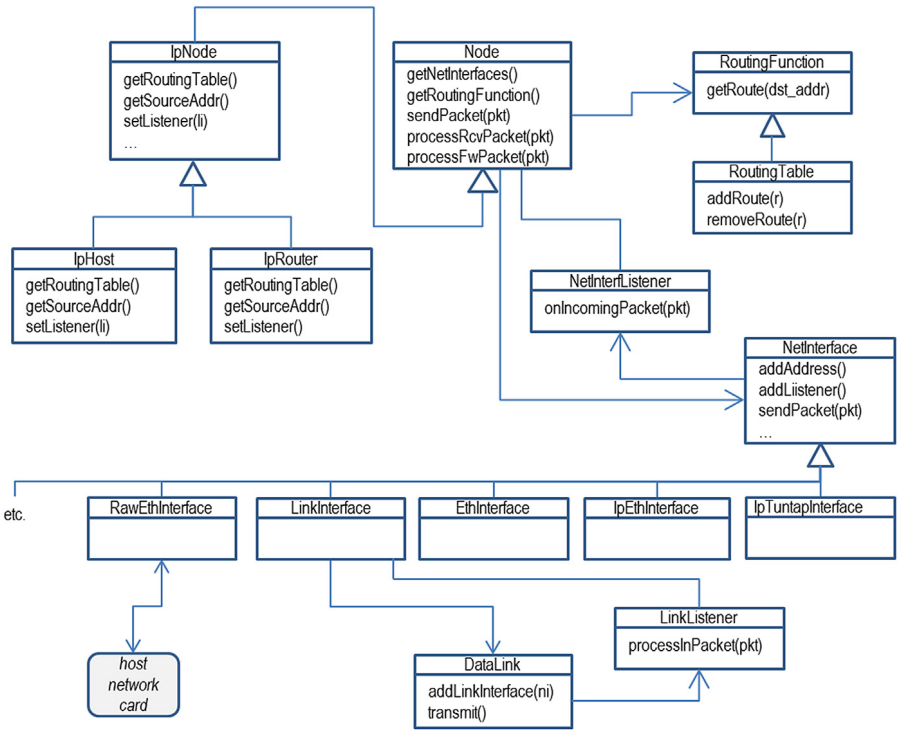


Fig. 2. Relationships between NEMO core objects.

`getRouters()` functions of `IpLink` objects, or through the standard DHCP protocol. Dynamic routing table configuration is also possible by using either:

- a centralized controller, in a SDN approach, collecting the link-state information received by all nodes, computing the shortest paths according to a given routing algorithm (different algorithms like Dijkstra, Bellman–Ford and Floyd–Warshall are available) and configuring the routing tables;
- a distributed approach using a routing protocol. Currently a light version of the OSPF routing protocol is available.

2.2.3. Capturing traffic traces

One of the advantages of NEMO is that all communications between nodes correspond to actual exchanges of packets traversing intermediate links. Therefore, real exchanged packets can be captured and analyzed through standard tools. To this aim, NEMO includes mechanisms for packet interception and trace exportation into standard *libpcap* files, the same format of Linux *tcpdump* and *Wireshark*. This allows the user to capture traces in different points of the network and perform further packet analysis using external tools like *Wireshark*.

2.2.4. Working with external components

NEMO is capable of connecting to and interacting with external systems (network links, nodes, and applications). This can be easily done by using one of its different *connectors*.

2.2.5. Distributed platform

NEMO can run on several interconnected machines, creating larger emulated networks. This is an important feature since it allows the user to distribute nodes and processing load on several hosts.

2.2.6. Virtualizing

NEMO is also capable to virtualize the execution of Java applications by creating virtual nodes, and running the applications on top of these nodes without the need for other virtualization tools or application code modifications.

3. Illustrative examples

In the following, we present three illustrative examples. The third one is also an example of scalability evaluation.

3.1. Capturing traffic

Whenever users want to intercept packets and process them, they can use a simple `Sniffer`. In particular, should users want to capture packets on a given link *link1*, they would just have to create a `Sniffer` object by passing a `LinkInterface` attached to *link1* and the packet processing function to the class constructor.

Fig. 3 illustrates the output of a simple practical example where router *R1* interconnects two networks (10.1.1.0/24 and 10.2.2.0/24) and host 10.1.1.2 pings host 10.2.2.2. The sniffer is located on *R*1 and, as it may be inferred from Fig. 3.b, all six ICMP packets (three ping requests/responses) are correctly captured.

In addition to process captured packets directly in Java, it is also possible to save traffic traces into a standard *libpcap* file for further analysis. NEMO provides a simple way to do that through the `LibpcapSniffer` class that saves all captured packets into a *libpcap* file. This standard file can be opened, for example, with *Wireshark* as shown in Fig. 4.

3.2. Using the tuntap connector

The simplest way for connecting a NEMO network to the local OS and/or to the external network is through the *tuntap* connector that exploits standard TUN or TAP interfaces, possibly supported by the underlying OS. In particular, Linux natively supports both TUN and TAP, macOS supports a variant of TUN (named UTUN), while in Windows only TAP can be set-up by means of the (de-facto standard) TAP implementation provided by the OpenVPN[5] research team. Fortunately NEMO *tuntap* connector provides to the user a uniform interface to all these different OSs, through the `TuntapInterface` and `Ip4TuntapInterface` classes.

In the example shown in Fig. 5, a simple virtual network is connected to the external network by using the *tuntap* connector.

With a few lines of code (Fig. 5.a) it is possible to create an emulated network (`link1` in the example), featuring an `Ip4Router` (*R*1) with a second interface of type `Ip4TuntapInterface` attached to a `tun0` interface. If a TUN or TAP interface and the routing table are correctly set-up, any application running on the underlying OS can communicate with all nodes of the emulated network and vice versa. Moreover, by enabling the IP forwarding it is possible to make the NEMO network routable and reachable by external nodes as well. The resulting topology is depicted in Fig. 5.b. In that figure, an example of PING from an external node 192.168.1.33 to the internal node 10.2.2.2 is shown as well.

[5] https://openvpn.net.

Table 1
Network scalability tests.

	Binary tree	4-ary tree
Tree height	21	10
Routers	4,194,303	1,398,101
Access links	2,097,152	1,048,576
Core links	4,194,302	1,398,100
Number of hops	44	22
Virtual time	3.52 ms	1.76 ms
Real time	6 ms	5 ms

3.3. Deploying large topologies

One of the main characteristics of NEMO is its high scalability. It can run very large networks on a single machine and/or different interconnected machines leading to a distributed emulation platform. Creating a distributed execution is very simple by configuring some network nodes with a virtual interface connected to an external *tunnel hub*. More than one *tunnel hub* can be set up, allowing the user to distribute the network topology amongst any number of connected machines.

Below we focus on a single-machine execution, testing the number of NEMO nodes and links that can be run on the same machine. We use two tree-topologies with different degrees, as detailed in Table 1, reporting: i) the dimension, in terms of tree height, number of routers, and links; ii) the number of hops; iii) the virtual time; and iv) the real time.

We ran our experiments on a common PC, with 16 GB RAM, Intel i7 2.70 GHz CPU, and 64-bit Windows 7 OS. We used the standard Oracle JDK SE 1.8 64-bit JVM, and we allocated 12 GB of RAM for the JVM.

Table 1 shows that we succeeded to emulate networks with 4,194,303 nodes, in the case of degree 2, and with 1,398,101 in the case of degree 4, respectively.

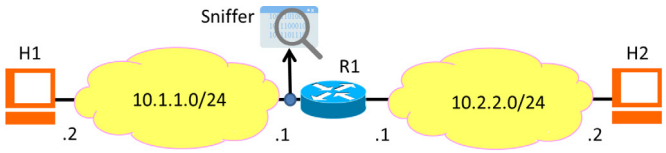
We have also performed some throughput evaluations. We used a 8×8 Manhattan topology composed of 64 routers interconnected through $2(8^2-8) = 112$ core links, and with $4 \cdot 8 = 32$ access links connected to the border routers. All links have been considered with capacity 100 Mb/s. Fig. 6.a reports the virtual time ($T_{virtual}$) and real time (T_{real}) spent to exchange packets between two end-points located at the most top-left and bottom-right access networks (hence, the two nodes were separated by 15 routers and 16 hops). We streamed an increasing number of packets at the maximum link bit rate, and considered two packet sizes: $P_{size} = 50$ bytes and $P_{size} = 1000$ bytes, including both IPv6 and UDP headers. As one can see in the graph, the real time spent by the emulator experiences a linear increase with the number of packets, and no critical conditions can be noted. The real time for transferring small packets (50 B) and the real time for transferring large packets (1000 B) are almost the same and the corresponding curves in the graph (the blue and green lines respectively) are overlapping.

Finally, in Fig. 6.b the memory usage for running a binary tree network is plotted as a function of the total number of nodes. As it is possible to see, the memory usage grows linearly with the number of nodes.

4. Impact

To the best of our knowledge, NEMO is the first network emulator providing all the following features at the same time:

1. It is an open emulator not focusing on either a specific network scenario or a network protocol. It completely re-implements all protocols (rather than just emulating them), without using the protocol stack of the underlying




```
long bit_rate=1000000; // 1Mb/s
IpLink link1=new IpLink(bit_rate,new Ip4Prefix("10.1.1.0/24"));
IpLink link2=new IpLink(bit_rate,new Ip4Prefix("10.2.2.0/24"));

Ip4Router r1=new Ip4Router(new IpLink[]{link1,link2});
Ip4Host host1=new Ip4Host(link1);
Ip4Host host2=new Ip4Host(link2);

// capture all packets sent through link1
new Sniffer(new PromiscuousLinkInterface(link1), new SnifferListener(){
@Override
public void onPacket(Sniffer sniffer, NetInterface ni, Packet pkt) {
System.out.println("Captured  packet:  "+ProtocolAnalyzer.exploreInner(pkt));
}
});

host1.ping((Ip4Address)host2.getAddress(),3,System.out);
```

(a)

```
PING 10.2.2.2 20 bytes of data:
Captured packet:  ICMP 10.1.1.2 > 10.2.2.2 type=8 code=0 msglen=28
20 bytes from 10.2.2.2: icmp_sqn=0 ttl=127 time=12 ms
Captured packet:  ICMP 10.2.2.2 > 10.1.1.2 type=0 code=0 msglen=28
Captured packet:  ICMP 10.1.1.2 > 10.2.2.2 type=8 code=0 msglen=28
20 bytes from 10.2.2.2: icmp_sqn=1 ttl=127 time=2 ms
Captured packet:  ICMP 10.2.2.2 > 10.1.1.2 type=0 code=0 msglen=28
Captured packet:  ICMP 10.1.1.2 > 10.2.2.2 type=8 code=0 msglen=28
20 bytes from 10.2.2.2: icmp_sqn=2 ttl=127 time=6 ms
Captured packet:  ICMP 10.2.2.2 > 10.1.1.2 type=0 code=0 msglen=28

--- 10.2.2.2 ping statistics ---
3 packets transmitted, 3 received, 0.0% packet loss, total time 2006ms
```

(b)

Fig. 3. Example of capturing traffic traces. On the left (a), the network topology, together with the corresponding code to create the network and attach a sniffer. On the right (b), the output of the sniffer.

```
// capture all packets sent through link1
new LibpcapSniffer(new PromiscuousLinkInterface(link1),LibpcapHeader.LINKTYPE_IPV4,"trace.pcap");
```

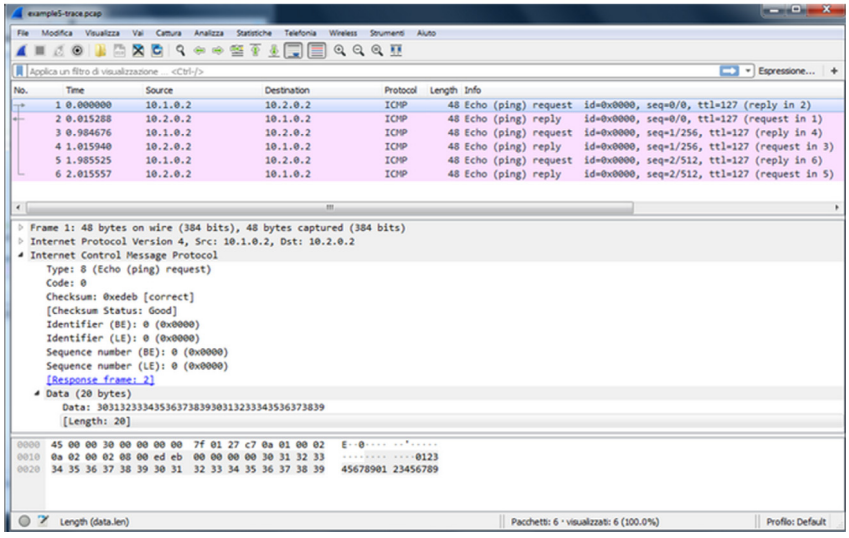


Fig. 4. Example of code for capturing and exporting traffic traces, and screenshot of Wireshark showing the exported trace.

system (like other Linux-based emulators do). All protocols are implemented according to the corresponding IETF and ITU-T standards, allowing the interaction with whatever external nodes and applications;

2. It is highly scalable. NEMO can run very large networks, up to millions of nodes on a single machine, or it can be deployed over different interconnected machines creating completely-distributed larger emulated networks;
3. It is portable. All protocols are implemented in Java and run in user space; this guarantees high portability and the possibility to replicate exactly the same tests and evaluations on different OS platforms, without requiring any specific configuration of the OS;
4. It supports both real time and virtual time emulations. By using a virtual clock it is possible to measure network events with nanosecond accuracy;
5. The interconnection with external systems is easy. By using one of the three available NEMO connectors, it is possible to interact with the underlying machine or to connect an emulated network to external nodes and networks;
6. It can be easily integrated with third-party Java applications. It is possible to run Java applications on top of an emulated node replacing the application network stack, transparently, without any modification of either the code of the application or the underlying OS.

Features 1 and 3 simplify the modification of existing protocols (e.g., new extensions of IPv4 or IPv6, variants of the current

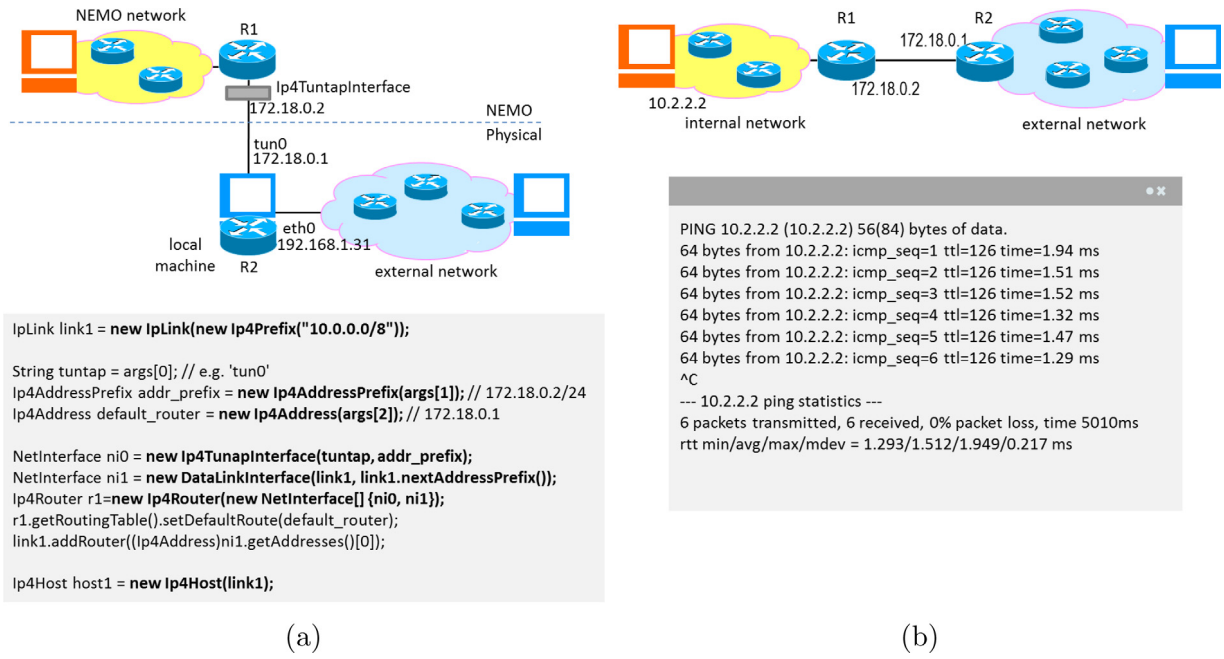


Fig. 5. Example of tuntap use. On the left (a), the code to create and attach a tuntap interface, together with the resulting network topology. On the right (b), the equivalent network topology, together with the output of a simple PING test.

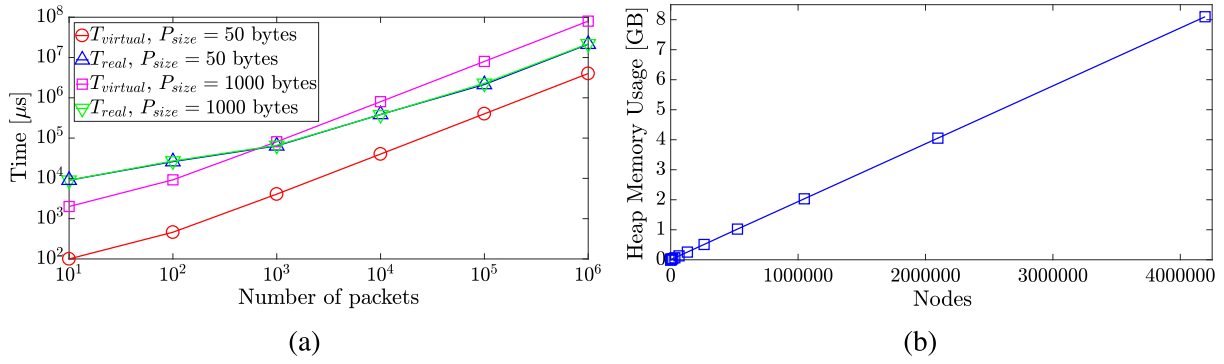


Fig. 6. Transmission time and memory usage in large topologies. On the left (a), the transmission time vs. an increasing number of packets for a 8 × 8 Manhattan network. On the right (b), the heap memory usage vs. the total number of nodes in a binary tree network. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

transport protocols, new routing mechanisms, etc.) as well as designing and developing new protocols. The use of Java technology and of object-oriented user-space programming also accelerates development and testing phases, allowing users to replicate the experiments independently of the underlying OS. The replacement of the standard `java.net` API makes also the development of new networked applications easier. Feature 2 is important when very large network topologies need to be tested, such as when testing new routing protocols or networking paradigms (like SDN or ICN). The possibility of distributing the emulation amongst different machines removes any constraint the underlying host may have. Concerning feature 4, on one hand, using a virtual clock allows the user to perform experimentations with high time precision and with perfect replication of the results, notwithstanding the time-precision and CPU load of the underlying machine; on the other hand, using the real clock allows the emulator to seamlessly interact with external systems and protocols. Finally, features 5 and 6 allow the user to create interesting hybrid scenarios, where the emulator is used to replace only a portion of the network and/or to integrate existing network elements or applications. In particular, the possibility to transparently run existing (Java) applications with a new network stack, without modifying either the application or the underlying OS, is a completely new feature, something no other network emulator provides.

Although NEMO is being publicly released with this software publication (it has been presented in a preliminary version in [6]), it has already been used, in collaboration with other research groups, to carry out different experimentations. It has been used for studying new networking paradigms (Traffic Engineering with SDN and IPv6 Segment Routing) [7], Network Function Virtualization [8], and to test a novel anonymity protocol [10].

5. Conclusions

The network emulator herein presented is a novel, powerful and flexible tool that can be used in different networking scenarios. It can be used for emulating a single link, a portion of a network, or an entire large network. The core of the emulator is platform and protocol independent: it can be used to emulate either standard IP-based networks or new protocols. It can be executed on a single machine, regardless the underlying OS, or on a distributed set of machines. External applications can be easily

attached, either as external network applications, or as applications running on top of the NEMO network stack. Considering all these advantages we hope that NEMO may become a common and widespread tool for network researchers and administrators to test and implement their own applications and products.

Within this article, we have only described some enlightening use cases and a simple scalability evaluation, showing that NEMO can support up to millions of nodes running on the same host machine. A more complete description of the software and all its possible usages can be found on the official website. NEMO is an ongoing project: we plan to improve its performance by increasing the number of supported protocols, to extend its features and functionalities, and to test it in other complex network environments.

Acknowledgment

The work of Luca Davoli was funded by the University of Parma, under "Iniziative di Sostegno alla Ricerca di Ateneo" program, "Multi-interface IoT sYstems for Multi-layer Information Processing (MIoTYMIP)" project.

The authors would like to thank Mr. Antonio Enrico Buonocore for carefully proofreading the paper.

References

[1] Macker JP, Chao W, Weston JW. A low-cost, IP-based mobile network emulator (MNE). In: IEEE military communications conference, 2003, vol. 1; 2003. p. 481–6. http://dx.doi.org/10.1109/MILCOM.2003.1290150.

[2] Beuran R, Nakata J, Okada T, Nguyen LT, Tan Y, Shinoda Y. A multi-purpose wireless network emulator: QOMET. In: 22nd international conference on advanced information networking and applications; 2008. p. 223–8. http://dx.doi.org/10.1109/WAINA.2008.111.

[3] Khanduri R, Rattan SS. Performance comparison analysis between IEEE 802.11a/b/g/n standards, vol. 78(1); 2013.

[4] Avvenuti M, Vecchio A. Internet emulation for java applications through socket factories. In: Proceedings 26th annual international computer software and applications; 2002. p. 111–6. http://dx.doi.org/10.1109/CMPSAC.2002.1044540.

[5] Wang S-Y, Lin Y-B. NCTUns network simulation and emulation for wireless resource management. Wirel Commun Mobile Comput 2005;5(8):899–916.

[6] Davoli L, Protskaya Y, Veltri L. NEMO: A flexible java-based network emulator. In 2018 26th international conference on software, telecommunications and computer networks; 2018. p. 1–6.

[7] Davoli L, Veltri L, Ventre PL, Siracusano G, Salsano S. Traffic engineering with segment routing: SDN-based architectural design and open source implementation. In: Proceedings of the 2015 fourth european workshop on software defined networks. Washington, DC, USA: IEEE Computer Society; 2015, p. 111–2. http://dx.doi.org/10.1109/EWSDN.2015.73.

[8] Salsano S, Veltri L, Davoli L, Ventre PL, Siracusano G. PMSR - poor man's segment routing, a minimalistic approach to segment routing and a traffic engineering use case. In NOMS 2016 - 2016 IEEE/IFIP network operations and management symposium; 2016. p. 598–604. http://dx.doi.org/10.1109/NOMS.2016.7502864.

[9] Abdelsalam A, Clad F, Filsfils C, Salsano S, Siracusano G, Veltri L. Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure. In: 2017 IEEE conference on network softwarization; 2017. p. 1–5. http://dx.doi.org/10.1109/NETSOFT.2017.8004208.

[10] Davoli L, Protskaya Y, Veltri L. An anonymization protocol for the internet of things. In: 2017 international symposium on wireless communication systems; 2017. p. 459–64. http://dx.doi.org/10.1109/ISWCS.2017.8108159.