NEmu: A distributed testbed for the virtualization of dynamic, fixed and mobile networks

Vincent Autefage, Damien Magoni

NEmu: a Distributed Testbed for the Virtualization of Dynamic Fixed and Mobile Networks

Vincent Autefage, Damien Magoni*

University of Bordeaux, LaBRI
351, Cours de la Liberation
33400 Talence, France

Abstract

Experimentation is typically the last step before launching a network application on a large production scale. However, it is often difficult to gather enough hardware resources for experimenting with a reasonably sized distributed application inside a controlled environment. Virtualization is thus a handy technique for creating such an experimentation testbed. We propose a tool called *NEmu* designed to create virtual dynamic networks by using emulation for testing and evaluating prototypes of networked or distributed applications with a complete control over the network topology and link parameters. *NEmu* leverages system emulators such as QEMU for virtualizing the hosts and the routers. It uses *vnd* for virtualizing components such as links and switches. In addition, *NEmu* allows users to create such customized topologies with limited hardware resources and without any administrative rights. We validate *NEmu* by replicating two network experiments and by showing that *NEmu* gives results very similar to the original ones.

Keywords: Emulation, mobile, network, testbed, virtualization.

1. Introduction

Experimentation is important to realistically and accurately test and evaluate network applications. Experimentation on algorithms is usually made by *simulation*. This technique is available through well known software like *ns* [1] or *OMNeT++* [2] and enables to evaluate the efficiency and the scalability of algorithms or protocols. Experimentation on a real program, i.e. implementation, is different to the extent that it is more focused on execution time, processor usage, memory consumption, network properties, etc.

It can be a difficult task when trying to experiment with a network application involving dozens of machines or more. Moreover, the mobility or the dynamics of scenarios can drastically increase the difficulty of experimentation. Using the Internet as a test bed is impractical as no parameters can be controlled. Setting up a hardware test bed is expensive and cumbersome. Furthermore, network applications can have very different ways of connecting hosts to each others and changing the network topology and network parameters of a hardware test bed is time consuming and error prone. Virtualization techniques for creating such an experimentation test bed can save resources and ease manipulations. It is a proven method for reducing the equipment and space costs as well as the energy consumption of using physical hosts [3].

Our solution to overcome the above hardware constraints is thus to build a test bed able to set up virtualized networks. A virtual network uses virtual machines instead of physical hosts and connects them with virtual links in order to build a virtual network topology. The virtual machines of a virtual network can be hosted on one or several physical hosts depending on the number of virtual machines needed and the resources capacities of the physical ones.

We propose a tool designed to create virtual networks for testing and evaluating prototypes of applications on the top of static, dynamic or mobile networks with a complete control over the network topology and link properties (bandwidth, delay, bit error rate, etc.) and the mobility of nodes. The goal of our tool is to enable the creation of reasonably sized virtual networks while minimizing the number of necessary physical hosts and network equipment needed. It can build host-based overlay networks by using emulators such as QEMU [4]. We have called our tool *NEmu* which stands for ***N**etwork **E**mulator for **m**obile **u**niverses* because it is able to create both fixed and mobile emulated networks. It is also a tribute to the name of the QEMU software which is a powerful machine emulator heavily used by *NEmu*. The contributions of our work are as follows:

- A detailed description of our *NEmu* software which is able to manage a distributed set of virtual nodes

*Corresponding author: phone: +33-5-4000-3540, fax: +33-5-4000-6669

Email addresses: autefage@labri.fr (Vincent Autefage), magoni@labri.fr (Damien Magoni)

and links for emulating any arbitrary static, dynamic or mobile network topology (Section 2).

- A detailed description of our *nemo* tool which implements the mobility inside *NEmu* and enables to create a complete autonomous mobile network by following a predefined scenario (Section 4).
- A performance validation experiment by making a comparison between *NEmu* and *Mininet* [5] (Section 5).
- A state of the art on related and previous work targeted at networking emulation and a comparison of the features provided by *NEmu* with the ones offered by similar alternative virtual networking testbeds (Section 6).

This paper is an extended and revised version of our previous work published in [6].

2. Description of *NEmu*

2.1. Overall Design

NEmu is a python program consisting of 6000 lines of code which allows to build a dynamic and distributed virtual network infrastructure. It is based on the concept of *Network Virtualization Environment* (NVE) introduced by N. Chowdhury and R. Boutaba in [7]. The main characteristic of a NVE is that it hosts multiple *Virtual Networks* (VN) that are firstly not aware of one another, and that are secondly completely independent of each other. A VN is a set of *virtual nodes* connected by *virtual links* in order to form a virtual topology. *NEmu* provides the possibility of creating several virtual network topologies with the central property that a VN is strictly disjoint from another in order to ensure the integrity of each VN.

Thus, *NEmu* integrates characteristics that are fundamental to a NVE: First, the *flexibility and heterogeneity* allows the user to construct a customized topology, with custom virtual nodes and virtual links. The *scalability* allows different virtual nodes to be hosted by different physical hosts in order to avoid limitations of a unique physical machine. The *isolation* decouples the different virtual networks which run on the same infrastructure. It also guaranties a strict separation between the host and the virtual networks. The *stability* ensures that faults in a virtual network would not affect another one. The *manageability* ensures that the virtual network and the physical infrastructure are completely independent. Therefore, a VN created on an infrastructure *A* can be deployed on another infrastructure *B*. The *legacy support* ensures that the NVE can emulate former devices and architectures. Finally, the *programmability* provides some optional network services to simplify the use of the virtual network (such as DHCP, DNS, etc.). It also implies that the user can develop and integrate his own additional services.

In addition, *NEmu* includes three important extra properties:

- The *accessibility* which means that *NEmu* can be fully executed without any administrative rights on the physical infrastructure. Indeed, the major part of public infrastructures, like universities and laboratories, does not provide administrative access to their users in order to ensure the security and the integrity of the whole domain. Therefore, the user execution would allow most people to use *NEmu* freely.
- The *dynamicity* of the topology enables node hot-connections which means that a virtual node can join or leave the topology dynamically without perturbing the overall virtual network.
- The *mobility* of nodes provides a way to create a self defined topology evolution through time and space. In other words, it is possible to create an autonomous connectivity scenario.
- The *community aspect* of the virtual network provides the possibility for several people to supply virtual sub-networks in order to build a community network like the Internet is.

2.2. Network Elements

NEmu is a distributed virtual network environment which allows users to create arbitrary and dynamic topologies. To this end *NEmu* is based on different building blocks. *NEmu* uses *virtual nodes* connected by *virtual links* in order to create a virtual network topology. A virtual topology can be hosted by one or several physical hosts. The part of the virtual topology laying on a given physical host represents a *NEmu session* which is configured by the *NEmu manager*.

2.2.1. Virtual Node

A *virtual node* for *NEmu* is an emulated machine that requires a hard disk *image* to work. This image is typically provided as a regular file on the physical host machine. Two types of *virtual nodes* currently exist in *NEmu*:

- A `VHost` is a virtual *host* machine (i.e., end-user terminal) on which the hardware properties and the operating system can be fully configured by the user.
- A `VRouter` is a virtual *router* directly configured by *NEmu* and provides ready-to-use network services.

Each virtual node uses a *virtual storage* which can be either a real media (cdrom, hard drive, etc.), a *raw* file or a host directory. A raw file can be privately dedicated or shared by several other virtual nodes. A modification of a shared file by one virtual machine will affect the others which may be troublesome if the file contains the operating system. To solve the problem, *NEmu* uses *Copy-on-Write* (CoW) operations on the original file. A *CoW file* (also

known as a *sparse file*) only stores the differences with its original file. The advantage, compared to a regular copy, is that the CoW file is much smaller. In addition, *NEmu* can use a regular directory on the physical host (without building a CoW), as a storage media in four different ways:

- by making a *Sparse* file which only stores the differences with its original file;
- by making a *Squash* file system which is a read-only raw image;
- by using a *FAT16* emulated interface which enables a direct access to a host's file system;
- by using a *Virtio* interface [8] which also enables a direct access to a host's file system;
- by using a *Network Block Device* which enables a virtual node to remotely access to a block device through the real IP network [9];
- by using a *SSH* tunnel which enables a virtual node to remotely access to a block device through a secured connection.

As said before, a `VHost` needs a disk image which must be supplied by the user. This image must be prepared prior to creating the virtual network. Furthermore, one image can be used by many `VHosts` by using *sparse files*. *NEmu* provides a network topology visualization option by processing its topology data file through *Graphviz* [10].

A `VRouter` is directly configured by *NEmu* and provides several services to simplify the virtual network management: DHCP, DNS, NFS, HTTP, SSH, NTP, Netfilter, dynamic routing protocols (RIP and OSPF), and QoS management with *Traffic Control* [11]. Moreover, it is easily possible to add some new services through a plug-in system available in *NEmu*. A router is running a customized image version of *TinyCore* which is a lightweight and highly configurable Linux distribution [12]. Such a system typically requires about ∼30 MBytes on disk and ∼100 MBytes in memory with all services running. Services provided by a `VRouter` are optional and can be enabled or disabled before or during runtime.

2.2.2. Virtual Link

A *virtual link* for *NEmu* is an emulated network connection between *virtual nodes*. This emulated connection can either be performed inside the machine emulator of a node (the link thus being attached to this node) or be performed by a dedicated emulation program (not running any system image in this case). Three types of *virtual links* currently exist in *NEmu*:

- A `VLine` is a virtual point-to-point *link* interconnecting two nodes.
- A `VHub` is a virtual multi-point *hub* emulating a physical Ethernet hub and interconnecting several nodes.
- A `VSwitch` is a virtual multi-point *switch* emulating a physical Ethernet switch and interconnecting several nodes.

Virtual links typically carry Ethernet frames from one virtual Network Interface Card (NIC) to one or more other virtual NICs. This Ethernet traffic is tunneled between virtual nodes by using TCP or UDP connections. *NEmu* can also use a *VDE* [13], a virtual switch which interconnects virtual machines through the shared memory system inside the Linux kernel, in order to create a local multi-point *switch*. Alternatively, *NEmu* can use our *vnd* program to emulate a network component. *vnd* stands for *v*irtual *n*etwork *d*evice. The *vnd* is a C++ program which consists in 6k lines of code that can emulate a `VLine`, a `VHub` or a `VSwitch` (defined as modes). The advantages of using a *vnd* is that the user can set the bandwidth, delay, jitter and bit error rate on any interface in any mode whereas QEMU offers no control over its hub emulation. In addition, *NEmu* also provides a `Slirp` which is a special type of link whose purpose is to provide an Internet access to the virtual node. It is an emulation of a NATed access to the real Internet by using the physical host NIC. Also, *NEmu* is able to interconnect a virtual NIC to a TUN/TAP kernel interface or to any UNIX socket.

Figure 1 shows an example of a *NEmu* managed virtual network. On the left side, two `VHosts` are connected to a `VRouter` through a `VSwitch` by using UDP tunnels. On the right side, two `VHosts` are connected to the above `VRouter` through a `VHub` by using TCP tunnels. Here, *virtual links* are created and managed inside *vnd* processes.

2.3. Management of Virtual Networks

We explain below the notion of a *NEmu* session and we describe the physical resources needed to run a virtual network.

2.3.1. Session

As already said above, a *NEmu session* represents a complete configuration of a network topology which lays on a physical host (storages, virtual nodes configurations and links). A distributed virtual network on n physical hosts consists in n *NEmu sessions* at least. A *session* is represented by an auto-generated directory in order to be saved and re-used. A *session* can be saved as a *sparse archive* which compresses all elements and which is compatible with *sparse files* unlike traditional archives.

2.3.2. Manager

The *NEmu manager* is the command line user interface to manipulate a *session*. *Sessions* are independent even if they are part of the same network topology. The *manager* can be used in three ways :

- as a python module to be integrated in another script or program;
- as a dynamic python interpreter;

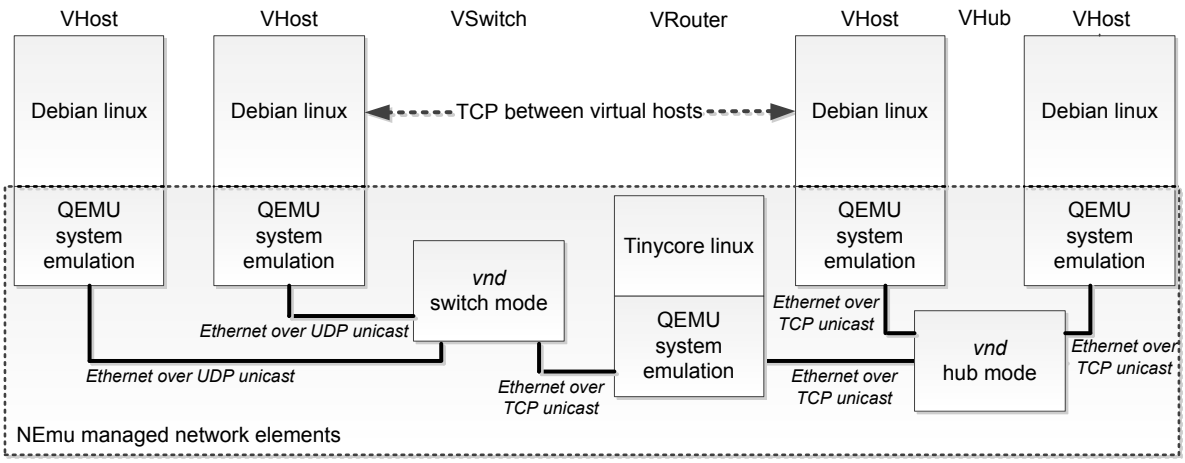

Figure 1: Network elements in action.

- as a python script launcher.

The *NEmu* manager provides a remote accesses, through SSH connections, to manipulate *NEmu sessions* laying on other distant hosts. The python language is upgraded in order to interact with other distant *sessions*.

2.4. Example of a Topology

We present in Figure 2 the Python script that generates the network topology previously shown in Figure 1.

2.5. Accuracy and Scalability

NEmu does not currently provide any specific primitives or tools for measuring backend side performances or for collecting results from all the nodes in the virtual network. The measurement tools available are all the usual system tools that can be installed and run either inside the virtual machines or on the physical hosts. Typical network performance evaluation tools include: `iperf`, `netperf`, etc. The accuracy of *NEmu* is mainly limited by the underlying network characteristics (e.g., bandwidths, delays, error rates, etc) of the backend connections between the physical hosts. It is obvious that the virtual bandwidth set between two virtual nodes will never be able to be above the real physical bandwidth available between the physical machines hosting those virtual nodes. Finally, the scalability of *NEmu* is mainly limited by QEMU's requirements for the virtual machines. The needs for each VM is typically at least one dedicated core (if possible) and at least a few hundred megabytes of RAM per VM. Thus, on a single regular machine, a virtual network could scale to a dozen nodes. On a group of regular machines or on a big server or cluster, it could scale to a hundred nodes.

```
# Creates a new session
InitNemu()

# New switch with 3 ports
VSwitch('switch', niface=3)

# New hub with 3 ports
VHub('hub', niface=3)

# New router with DHCP and SSH services
VRouter("router", nics=[VNic(), VNic()], services=[
      Service("ipforward"),
      Service("ifup", "0:192.168.1.1", "1:192.168.2.1"),
      Service("dnsmasq", domain="local1", net="192.168.1.0/24",
                  start="192.168.1.10", end="192.168.1.20"),
      Service("dnsmasq", domain="local2", net="192.168.2.0/24",
                  start="192.168.2.10", end="192.168.2.20"),
      Service("sshd")])

# New host configuration (French keyboard, SDL display and 512MB of RAM)
VHostConf('chost', sdl=None, k='fr', m=512)

# New hosts with the chost common configuration and 2 NICs
VHost('a', conf='chost', hds=[VFs('debian.img', type='cow'], nics=[VNic()])
VHost('b', conf='chost', hds=[VFs('debian.img', type='cow'], nics=[VNic()])
VHost('c', conf='chost', hds=[VFs('debian.img', type='cow'], nics=[VNic()])
VHost('d', conf='chost', hds=[VFs('debian.img', type='cow'], nics=[VNic()])

# Connects nodes to links
Link('router:0', 'switch:0')
Link('router:1', 'hub:0')
Link('a', 'switch:1')
Link('b', 'switch:2')
Link('c', 'hub:1')
Link('d', 'hub:2')

# Starts the virtual network
StartNemu()
```

Figure 2: A example of a topology script written for *NEmu*.

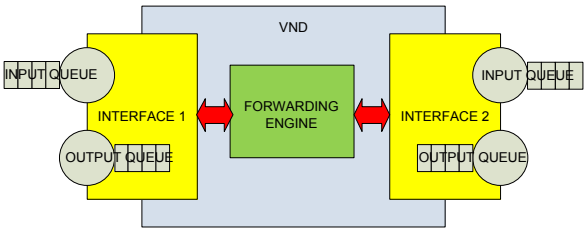

Figure 3: Architecture of a *vnd*.

3. The Virtual Network Device (*vnd*)

This section presents our software program called *vnd*. It is a program which is able to emulate network devices such as a link, hub, switch or an access point from a high level point of view. This is by far not the first software able to emulate network devices but it has some unique features which may prove useful in the network virtualization domain:

- it runs as a lightweight stand-alone process and can fail without killing virtual machines,
- it can support dynamic connections and reconnections as well as disconnections and is immune to the failures of virtual machines,
- it provides many networking backends, such as the *sockets* API, which is available on any platform, to connect to the virtual machines,
- it can dynamically set the link properties such as bandwidth, delay, jitter and bit error rate,
- it can coarsely emulate wireless interface cards in infrastructure and ad hoc modes as well as access points.

3.1. Architecture

A *vnd* contains an engine and several interfaces. It can contain any number of interfaces as long as system memory is available. Interfaces can be created and destroyed at runtime. Each interface owns an input queue and an output queue. Each queue has a number of buffers which can be set at runtime. Interfaces are internally connected through the engine. Figure 3 shows the architecture of a *vnd* with two interfaces. Data coming in or out of a *vnd* can be interpreted in two ways:

- `raw`: data is considered as an uninterpreted flow of bytes and each buffer can contain data bytes up to its maximum size,
- `Ethernet`: data is considered as Ethernet frames and each buffer can contain only one frame whose size shall be less or equal than the buffer's maximum size.

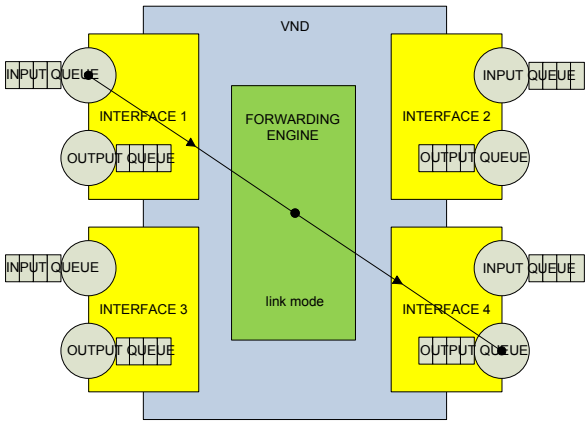

Figure 4: Link mode.

A *vnd* can be set to one of six different working modes depending on the network component that it emulates. The first four modes are typical network components which are independent from any virtual machine. The last two modes are used to emulate a Wireless Interface Card (WIC) in either infrastructure or ad hoc mode. Thus in the last two modes, the *vnd* is not used as a separate network component but it is used in conjunction with a virtual machine to form a mobile node. When the *vnd* is used as a wireless card emulator, it is connected to its virtual mobile node by a specific and unique interface called a `direct` interface. If the mobile node is considered using infrastructure mode (BSS or ESS), then the *vnd* also has another specific and unique interface called an `access` interface which is connected to the access point that the mobile node is currently associated with. A *vnd* can be set to one of the six possible modes:

1. `link`: each interface is directly bound to another interface, which means that any data going into the input of the first interface is forwarded to the output of the second interface in this given direction (i.e., it is one way as shown on Figure 4),
2. `hub`: each interface is bound to all others, which means that any data going into the input of an interface is forwarded to the output of all the other interfaces except itself as shown on Figure 5,
3. `switch`: any frame going into the input of an interface is forwarded to the switch engine which uses a forwarding table to determine the output interface leading to the device having the same address as the frame's destination address as shown on Figure 6,
4. `access point`: any frame going into the input of an interface is forwarded to the switch engine which uses a forwarding table to determine the output interface leading to the device having the same address as the frame's destination address (see Section 4),
5. `infrastructure interface`: any frame going into the input of the `access` interface is forwarded to the output of the `direct` interface leading to the mobile

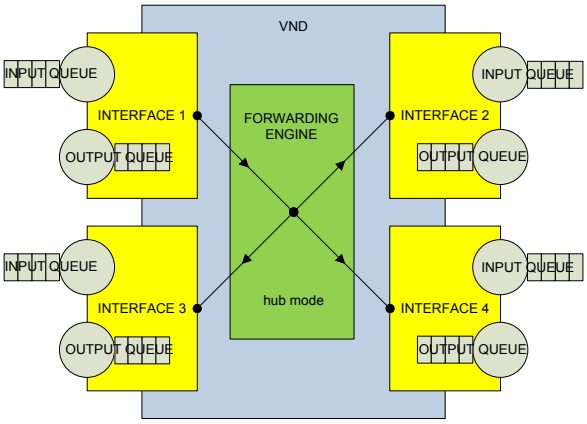

Figure 5: Hub mode.

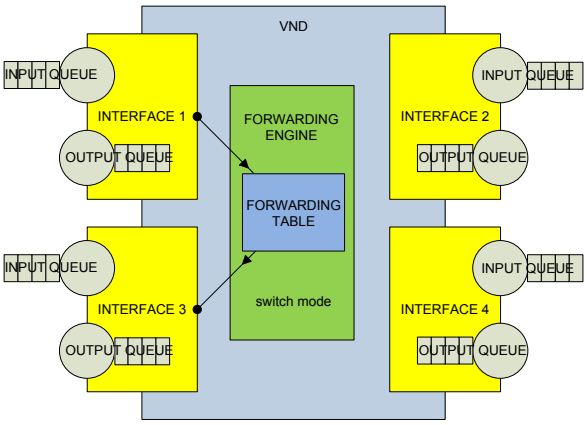

Figure 6: Switch mode.

node itself, and any frame going into the input of the `direct` interface is forwarded to the output of the `access` interface leading to the access point (see Section 4),

6. `ad hoc interface`: any frame going into the input of any interface which is not the `direct` interface is forwarded to the output of the `direct` interface leading to the mobile node itself, and any frame going into the input of the `direct` interface is forwarded to all the other output interfaces except itself (see Section 4).

The last four modes only make sense when the data is interpreted as Ethernet frames as MAC addresses are needed. In order to emulate the IEEE 802.11 protocols, a pseudo header is added to any frame coming from an access point or emulated WIC.

The forwarding table is filled as in a hardware switch having auto-learning capability. When a frame is received by an interface, the *vnd* checks if the source MAC address is associated with this interface. If yes nothing is done, if no, the *vnd* stores this association in the forwarding table. When a frame is transmitted, the engine looks up the destination MAC address of the frame in the forwarding table and forward the frame to the interface associated with that address. Currently, the forwarding table does not remove entries depending on a given lifetime and thus the table must be manually cleared if needed. The *vnd* supports port-based VLANs in `hub` and `switch` modes. The *vnd* does not yet implement the Spanning Tree Protocol, thus it is up to the user to avoid making loops in the topology of the virtual network.

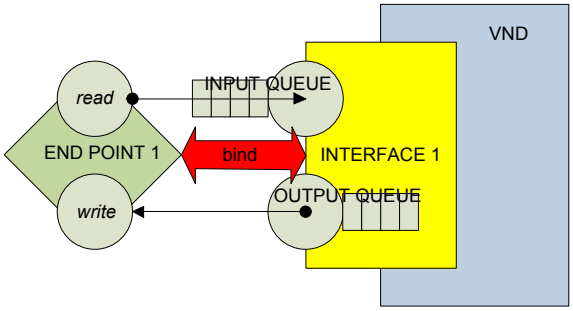

Figure 7: Bind between an emulated interface and a network backend in a *vnd*.

3.2. Implementation

In the domain of virtualization, the term *network backend* is often used to designate the software part of an emulator that enables the connection of the emulator to the other emulators either on the same physical machine or on different ones. Network backends on UNIX are usually implemented with `TAP` interfaces, VDE [13], *sockets* or *slirp* (which provides a full TCP/IP stack implementing a virtual NATed network).

The *vnd* currently provides Internet and UNIX local *sockets* backends as well as TAP and VDE backends. All these backends are implemented in an object called `endpoint`. To be useful, a network backend must be tied to a virtual network interface in a machine or a *vnd*. This tie is implemented in the code of emulators in more or less flexible ways. In order to support the dynamic features presented at the beginning of this section, the *vnd* implements the tie in a flexible way by separating the virtual interface from the endpoint. This tie can be dynamically created or destroyed by using the `bind` command as shown on Figure 7. Thus the failure of a network backend connection does not impact a virtual interface except for the loss of traffic. An endpoint can also be rewired to another interface if needed although data can be lost in the process.

As *NEmu* currently only uses QEMU for system emulation, we show on Figure 8 an example of a TCP connection between the network backends of a QEMU virtual machine and a *vnd*. QEMU defines a local VLAN object to associate the virtual `eth0` interface to the socket backend called `socket.0`. The difference between the *vnd* `bind` and the QEMU VLAN is that a `bind` creates a bijection

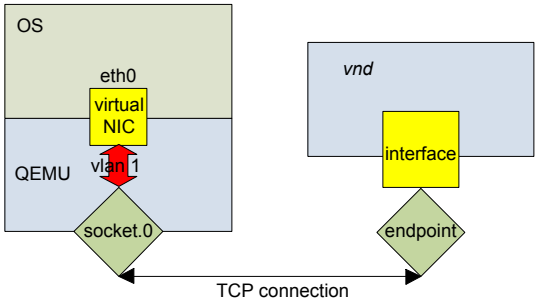

Figure 8: TCP connection between the network backends of a QEMU and a *vnd*.

between an interface and an endpoint whereas a VLAN can connect several backends to the same interface thus actually acting as a simple VLAN inside the emulator (everything is broadcasted inside though).

3.3. Performances

In order to be useful and realistic, the *vnd* program must have acceptable performances. We have carried out several measurements to evaluate its performances and compare them to other emulators, namely QEMU and Dynamips, as they also provide socket-based network backends. The scenario was interconnecting two QEMU virtual machines via a virtual network device (line, hub and switch) emulated by either a QEMU, a Dynamips or a *vnd* process. All network backend connections were TCP connections made on the loop-back interface of the physical machine which was an Intel Core 2 equipped desktop PC. An FTP session was established between the two virtual machines and a 1GB file was transferred and timed to compute the bandwidth. Table 1 shows the throughput of the various possible emulations of virtual network devices. We can see that our *vnd* program performs at least as well as the others when emulating any device. We can also see that QEMU limits the throughput of the virtual machines at around 34MB/s.

To measure the maximum throughput of the *vnd* itself, we have used two *netcat* processes interconnected by a *vnd* in raw mode. The throughput amounts to 682 Mbps to be compared with the 910 Mbps obtained by a direct connection between the two *nc*. Thus the *vnd* only achieves 75% of the throughput achieved by the direct connection. This loss is due to the pipelining of the two TCP connections as well as the queueing and the processing time inside the *vnd*.

Finally we have also done measurements to evaluate the accuracy of the *vnd* bandwidth and delay parameters. Concerning the bandwidth, we can deduce from the above results that the *vnd* can at most emulate 100 Mbps speeds. We have observed by varying the bandwidth parameter from 100 kbps to 100 Mbps that the difference between the value of the bandwidth parameter (set on the interface of the *vnd* by the user) and the value of the measured bandwidth does not exceed 2%.

Table 1: Throughput of the switch emulators

Network Component	Emulation Program	Measured Throughput
Baseline	nc to nc link	910 Mbps
	vnd (raw connections)	682 Mbps
`VLine`	QEMU to QEMU direct link	272 Mbps
	vnd (link mode)	240 Mbps
`VHub`	QEMU hub	160 Mbps
	vnd (hub mode)	240 Mbps
`VSwitch`	Dynamips switch	12 Mbps
	vnd (switch mode)	156 Mbps

4. The Network Mobilizer (*nemo*)

4.1. Design

NEmu can emulate mobile networks. Thus it is possible to create a virtual network topology that evolves in time. In order to manage mobility, *NEmu* uses a special mobility engine called *nemo*. *nemo* [14] is a lightweight C++ program which can generate connectivity scenarios for mobile networks. A connectivity scenario is a time stamped list of wireless link connection and disconnection events between mobile nodes. Indeed, *nemo* is based on a specific use of the *vnd* [15] software, which can on-the-fly create virtual links having dynamically set characteristics. *nemo* is able to send orders to *NEmu* in real time which enables to emulate the changes of connectivity between mobile nodes by creating, destroying or changing the characteristics of the links at the appropriate time. *nemo* works behind the scene and is entirely controlled by *NEmu* which acts as the user interface. *nemo* is implemented in C++, contains around 3000 lines of code, and is using some Boost [16] libraries including the powerful asynchronous input/output library called *asio*. It is a lightweight program using around 1MB in RAM and it is portable thanks to Boost (on the majority of UNIX and Windows variants). *nemo* is composed of two parts : one part based on a simulated time scheduler and another part based on a real time scheduler. The source code is available at [14].

4.2. Simulated Time Scheduler

The simulated time scheduler is the heart of the simulation part of *nemo*. It can generate connectivity scenarios for the real time scheduler. Three steps are necessary to generate a connectivity scenario:

- generate a map;
- generate a mobility scenario on this map;
- generate a connectivity scenario from the mobility scenario.

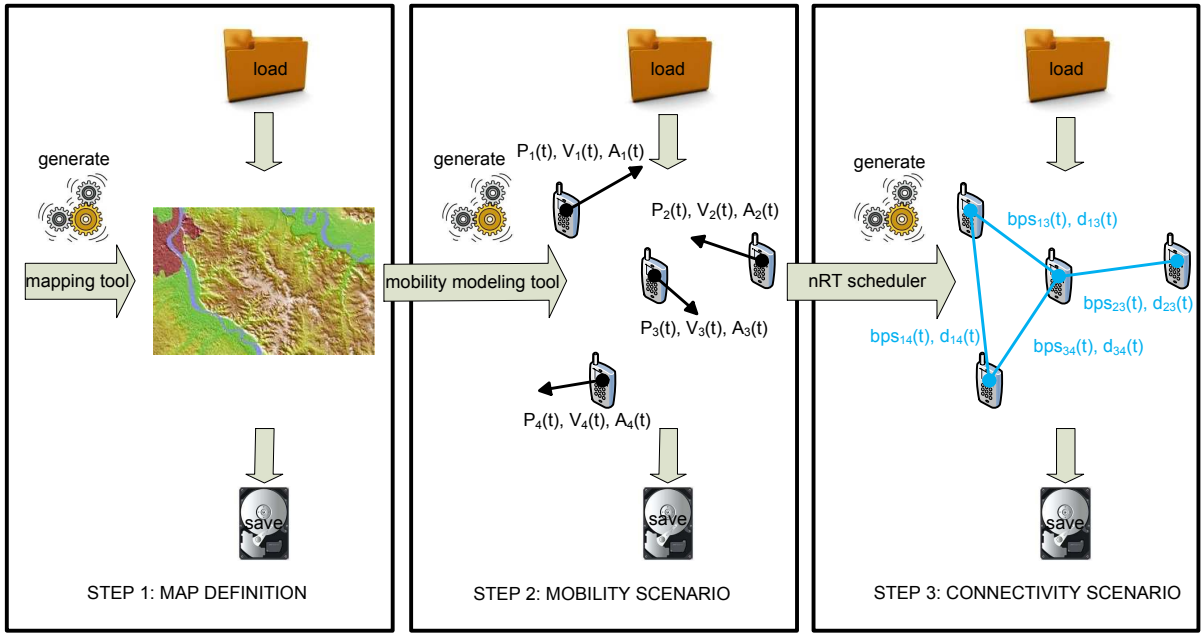

Figure 9: Generation of connectivity scenarios with *nemo*.

At each step, the results of the step can be saved on disk in order to be loaded at a later time to avoid recomputation. The simulated time scheduler runs the mobility scenario and at each time interval (set by the user), it computes the distances and the possible wireless connections between all the pairs of mobile nodes. The steps are illustrated on Figure 9. Being able to generate connectivity scenarios is an advantage over using a network simulator interconnecting real applications with taps, because the latter must compute the mobility at every run and this computation could be too heavy to enable the real-time execution of the applications. Up to now, *nemo* generate rectangular maps and purely random mobility scenarios. This is useful for carrying functional tests. *nemo* is also capable of importing *ns-2* formatted mobility files produced by tools such as Bonnmotion [17] providing realistic mobility models such as the *Gauss-Markov Mobility Model* [18] or the *Reference Point Group Mobility Model* [19]. In the future, *nemo* will be able to load more elaborate 2D or 3D maps containing attenuation information.

The simulated time scheduler suffers from several limitations:

- it may require intensive computation of the order of $O(n^2)$ (that is why it is written in C++);
- it requires the user to make a tradeoff between the temporal precision (the time interval between each connectivity evaluation), the computation time and the number of events detected.

4.3. Real Time Scheduler

The real time scheduler is the heart of the emulation part of *nemo*. It executes the connectivity events at their exact time stamp, set with respect to the start of the scenario. The temporal precision used in the real time scheduler is equal to the one set during the processing of the mobility scenario by the simulated time scheduler. The interaction of the real-time scheduler with *NEmu* is illustrated on Figure 10. It shows that *NEmu* plays the role of a central controller for the other processes. In a virtual mobile network, one *vnd* is used to emulate each wireless network interface card (WIC). Thus there is one *vnd* per virtual mobile node and inside it are instanciated the real network backend links (i.e., TCP or UDP tunnels). *NEmu* transmits the orders of the user (e.g., *start, stop*, etc) to *nemo*. When the real time scheduler is running, *NEmu* also recovers the connectivity events generated by *nemo* and retransmit them to the various *vnd* corresponding to the WICs of the virtual mobile nodes in order to make the network topology evolve. The real time scheduler can be paused and resumed at any moment by the user.

The real time scheduler suffers from several limitations:

- as opposed to *NEmu*, *nemo* is centralized;
- it is better to execute all the sessions and nodes on a unique physical host to avoid reducing the performances;
- the *sockets* used to connect the *vnd* introduce delays and reduce the temporal precision. The latter will be at best of the order of the millisecond.

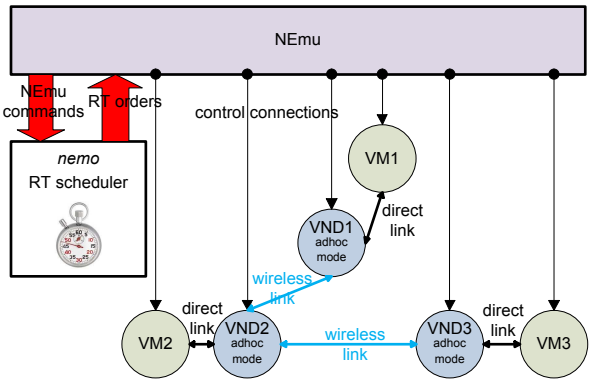

Figure 10: Emulation of virtual mobile networks with *nemo*.

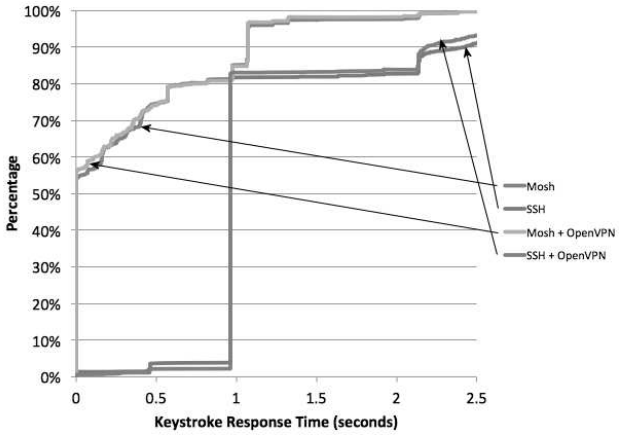

Figure 11: Original Mosh results obtained by mininet.

5. Experimentation

We present the results of two experiments that we replicated in order to show the accuracy of *NEmu*. The first experiment was initially done with the Mininet container-based emulator, the second was done with the JBotSim simulator. In both cases, we were able to accurately replicate the results with *NEmu*.

5.1. Mosh Experiment Replication

In order to validate the accuracy of experimentation results obtained with *NEmu*, we reproduce a performance benchmark of *Mosh* [20]. Mosh is a remote terminal application which is more tolerant to connectivity break than SSH by using the SSP protocol and a predictive algorithm.

The experimentation consists in measuring the average keystrokes response time for Mosh and SSH. This experiment has been previously carried out on *Mininet* [5], another network emulator which is well known for its degree of realism in experimental conditions [21].

We reproduce the exact experimentation described in a Stanford network lecture [22] and which has been officially supported by Mosh developers. In this experimentation the client is connected to a switch through an emulated 3G network, and the server through an emulated Wi-Fi network. Authors consider the following experimental network conditions:

- 3G:
  - packet loss rate: 0.01;
  - bandwidth: 1 Mbps;
  - delay: 450ms.
- Wi-Fi:
  - packet loss rate: 0.08;
  - bandwidth: 25 Mbps;
  - delay: 30ms.

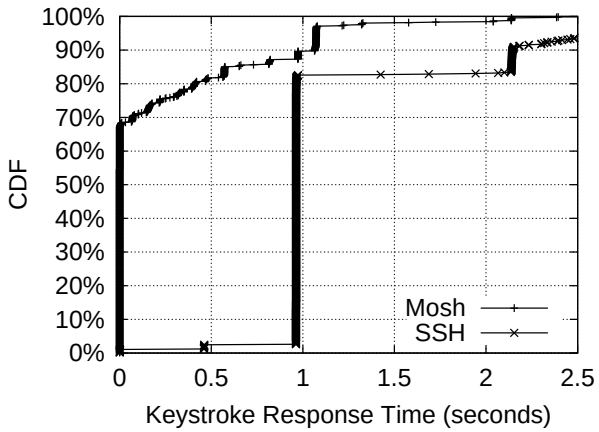

Figure 12: Mosh results obtained by *NEmu*.

Thanks to our *vnd* program, we configure the network properties as detailed above. Original results are presented in Figure 11. Our results are illustrated in Figure 12. We can notice that both results are nearly identical. Those results imply that *NEmu* can offer a similar degree of realism than *Mininet*.

5.2. AMiRALE Experiment Replication

In order to validate the accuracy of experimentation with mobile devices performed with *NEmu*, we replicate performance results obtained by simulation of a multi-agents system called AMiRALE [23]. Emulation is performed with *NEmu* while simulations are carried out with JBotSim [24], a Java library which enables the design of low and high level communication scenarios and behaviors of heterogeneous mobile nodes.

AMiRALE is a distributed system which enables several autonomous vehicles to perform common tasks collaboratively. The application scenario consists in a team of ground robots which has to collect a given number of garbage in a park, each one being a target for the cleaning robots. We evaluate the output data rates generated by AMiRALE as a function of the number of targets to

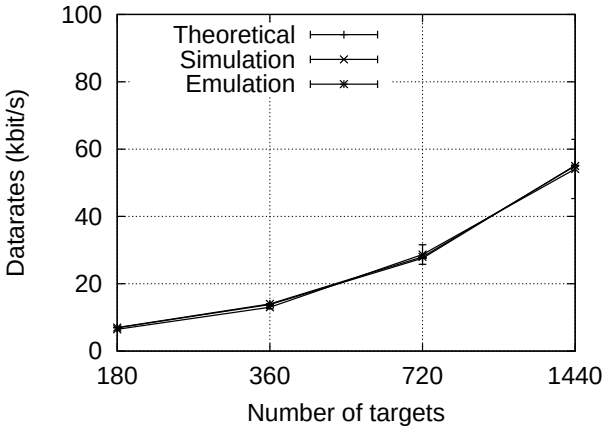

Figure 13: AMiRALE results obtained by JBotSim and *NEmu*.

process. Each robot is specialized which means that it can only clean one kind of garbage. When a robot finds a garbage which it is not able to collect itself, it generates a new *mission* in order to inform other robots of the existence of this garbage. This strategy enables a robot to clean a garbage which has been discovered by another robot.

Figure 13 shows the results of our experiment as a function of the number of targets. Standard deviation values are also shown on the plots. Since all missions are broadcast without any restriction, we can calculate the theoretical data rates by multiplying the number of missions by the size of an unique mission and divide the result by the frequency of broadcasts. This result is provided by the *theoretical* plot. The *emulation* plot shows results from the *NEmu* experiments while the *simulation* plot shows results from the JBotSim simulations. This figure shows that theoretical, simulation and emulation results are very similar which implies that *NEmu* provides coherent performance results for this mobile devices' scenario.

6. Related Work

6.1. Node Emulation Systems

Currently, *NEmu* uses QEMU virtual machines as virtual network nodes. Despite the fact that a lot of solutions of host virtualization exist, we chose QEMU which is a generic and open source machine emulator and virtualizer [4]. QEMU runs without any administrative rights and emulates a lot of various hardware architectures [25]. Therefore, instantiating a QEMU virtual machine as a virtual node allows the user to configure freely its hardware and software layers which fills perfectly with the flexibility and heterogeneity properties of a NVE.

Others systems such as *VMware* [26] or *Xen* [27, 28] have better I/O performances but can only emulate x86 and x64 architectures [29, 30] which compromises the legacy support defined in Section 2. Moreover, *Xen* is too close to the system which means that it requires administrative rights to be configured properly.

Virtualization systems such as *VirtualBox* [31] and *Hyper-V* [32] are also limited to x86 and x64 architecture emulation.

Regarding *UML* [33], the software is now unmaintained and only can emulate Linux operating systems.

OpenVZ [34] can be see as the successor of UML but also only supports Linux operating systems.

LXC [35] is a Linux kernel system which can encapsulate several process and a sub-file system in a virtual container. This solution cannot be consider as a real virtualization system and does not enable any hardware configuration.

Dynamips [36] is an emulation system dedicated to CISCO systems. Therefore, it cannot emulate standard user machines.

6.2. Link Emulation Systems

6.2.1. Virtual Switches

NEmu uses a program called *vnd* in order to emulate customized virtual links. Nevertheless, other systems enable to inter-connect several virtual machines.

VDE [13] is a virtual switch which inter-connects virtual machines through the shared memory system inside the Linux kernel. Such a system cannot be distributed on several physical machines. Moreover, VDE does not include any mechanism in order to manipulate link properties like bandwidth, delay, etc.

Open vSwitch [37] is an open source project which enables to instantiate virtual switches with a high customization of virtual links. However, this software relies on virtual network interfaces inside the Linux kernel which can only be created by a system administrator of the physical infrastructure. The accessibility defined in Section 2 would be impossible with Open vSwitch.

Vnet [38] is a distributed inter-connection system which enables to link several virtual machines which lay on different physical hosts. Even if the system is distributed, it does not provide any link customization mechanism.

Click [39] is a Linux kernel framework which allows to create software defined routers (i.e. at network layer 3). Therefore, this solution is not suitable for our needs.

6.2.2. Link Properties Manipulation

Our program *vswitch* includes the link properties customization in order to configure bandwidth, delay, jitter and bit error rate. Several other solutions exist in order to make this job. For instance *NetEm* [40], which relies on the Linux kernel tools *Traffic Control* [11], enables to manipulate same properties. However, it requires root privileges on the real infrastructure.

Dummynet [41] enables to create some customizable virtual links between two entities. Therefore, it cannot play the role of a multi-point device like a hub or a switch. Moreover, this solution operates directly in the kernel which is not compliant with the accessibility target.

A similar project called *netpath* [42] uses the Click library in order to create customize virtual links.

Another project called *Trickle* [43] can be used without any administrative rights in order to fix the maximum date rates for a process. However, it can only fix this property for the entire process. Thus our virtual switch is much more configurable.

6.3. Network Virtualization Environment

Dynagen [44] is for Dynamips, the equivalent of *NEmu* for QEMU. Dynagen manages fleet of Dynamips machines and their inter-connections. However, the dynamicity of the topology is strongly limited, adding network services is quite impossible and there is not any community aspect.

GNS [45] is an open source software which allows to build a virtualized network topology with Dynamips, VirtualBox and QEMU virtual machines. However, it does not provide the possibility to build a community network and adding network services is as complicated as Dynagen. Finally *GNS* is hardly usable without any graphical interface making difficult the creation of a complex network.

Velnet [46] is a virtual environment dedicated to teaching which uses VMware virtual machines. The complete topology can only run on a single host which implies strong limitations on the size of the virtual network.

ModelNet [47] emulates a distributed virtual network but this one remains static at runtime. Thus, the dynamicity is not ensured with ModelNet. Further, the management of this system is fully centralized on an unique physical machine which disables the community aspect.

Vagrant [48] uses *VirtualBox* virtual machines in order to emulate virtual network. The topology is hosted on a single physical machine and remains static at runtime. Finally, the inter-connections are built inside the host kernel making a *flat network*, i.e. a network which not relies on standards ways of addressing and routing.

VINI [49] is a distributed virtual network which overhangs the *PlanetLab* testbed [50] which is an international distributed cluster system. VINI uses UML virtual machines which strongly limits operating systems for nodes. Moreover, connections between nodes are made with virtual networks interfaces inside the kernel of the physical machine which makes the configuration impossible without administrative rights.

Violin [51] is similar to VINI but provides some virtual routers which hosted different services like *NEmu*. However the use of UML and the need of an existing overlay limits the use scope of this solution.

NetKit [52] relies also on UML and VDE switches which do not require any administrative rights. Such a system cannot be distributed.

Marionnet [53] is a virtual environment dedicated to teaching. It provides several network services and the community aspect but relies on UML.

Virconel [54] uses OpenVZ virtual machines which also strongly limits operating systems for nodes. Moreover, The topology is static during runtime and the interconnections between virtual machines are made in the host kernel which requires special rights on the physical system.

VNUML [55] is a static virtual system relying on UML and which requires administrative rights.

VNX [56] is the successor of VNUML is a compatible with other virtualization systems. However, as VNUML, it requires administrative rights and does not include any link property mechanism.

Cloonix [57] is a dynamic virtual network environment. Nevertheless, it does not include link property mechanism.

Mininet [5] is a Container-Based Emulator which allows to create a custom and dynamic topology on a unique physical host. The second version of this system is highly demanding about fairness of experimentation conditions [21].

CORE [58] is a graphic tool which enables to emulate virtual mobile networks. This system relies on a framework called IMUNES [59] which runs over the FreeBSD [60] operating system. Nodes and links are managed inside the operating system kernel which implies strong limitations in terms of flexibility.

MobiEmu simulates mobile networks through ns3 for the mobility and LXC containers for nodes. This tool does not enable NVE possibilities.

OpenNebula [61] provides a powerful solution to manage a virtual cloud on a wide physical infrastructure with an important number of nodes. OpenNebula manages the whole cloud domain from a single access point. Storage medias are centralized and are accessible through the network which implies the use of a NAS or NFS.

NET [62] is a powerful hardware-based infrastructure which allows to perform realistic experimentation on mobile networks. However, this solution uses real network inter-connection devices (i.e. switches, etc.) in order to build the virtual network.

PdP [63] is partial NVE implementation which focuses itself on flexibility, isolation, high-speed data rates and low cost. It uses OS level virtualization nodes such as OpenVZ.

Onelab2 [64] is a well known emulation tool over PlanetLab. It also relies on DummyNet which strongly limits the flexibility of this solution.

IP-TNE [65] is an original solution which enables hosts and real network to interact with a virtual mobile network. It is not really an emulation solution since it provides only the mobility properties of nodes and not the real node virtualization.

Research platforms such as *PlanetLab* [50], *GENI* [66] and *FEDERICA* [67] supply virtual infrastructure built on the top of *slices* of third parties owned hardware. This approach leads to the concept of Testbed as a Service (TaaS), exemplified by FanTaaStic [68]. Federations of testbeds have also been recently emerging, such as the EU-driven *OpenLab* federation [69] as well as the *OneLab* federation [70], managed by UPMC. In all these cases, the user needs a specific account, must comply to specific us-

age policies and has to use the tools, services and APIs of these testbeds.

Table 2 exhibits several properties of those previous solutions compared to *NEmu*. We see that *NEmu* can cover all usages (test and proof, performance evaluation as well as learn and teach). It can achieve a high realism as a research tool by managing dynamic virtual networks and by being able to be distributed over several physical machines. Furthermore, it can be easily used and deployed as no special rights are required on the physical machines. The creation of a complex network is facilitate by the important number of available network services and the easy way to add another ones. Finally, it offers a new feature called *community aspect* that enables several users to merge their virtual networks together in order to build a single larger network.

Table 2: Comparison of network virtualization tools.

	Test and proof	Evaluation	Teaching	Dynamic	Distributed	Mobility	Community aspect	Network services	Regular privileges	Configurable links	NVE+ compliant
Dynagen	•	○	•	○	•	○	○	•	•	○	○
GNS3	•	○	•	○	•	○	○	•	•	○	○
Velnet	○	○	•	○	○	○	○	•	•	○	○
ModelNet	•	•	○	○	•	○	○	○	•	•	○
Vagrant	•	○	•	○	○	○	○	•	•	○	○
VINI	•	•	○	•	•	○	○	○	○	•	○
Violin	•	•	○	•	•	○	○	•	•	•	○
NetKit	•	○	•	•	○	○	○	•	•	○	○
Marionnet	•	○	•	•	○	○	•	•	•	○	○
Virconel	•	•	•	○	•	○	○	•	○	○	○
VNUML	•	○	•	○	○	○	○	○	○	○	○
VNX	•	•	•	○	•	○	○	○	○	○	○
Cloonix	•	•	•	•	○	○	○	○	•	○	○
Mininet	•	•	•	•	○	○	○	○	•	•	○
CORE	•	•	○	•	○	•	○	•	○	•	○
MobiEmu	•	•	•	•	○	•	○	○	○	•	○
NET	•	•	○	•	•	•	○	○	○	•	○
PdP	•	•	○	○	•	○	○	○	○	•	○
Onlab2	•	•	○	○	•	○	○	○	○	•	○
NEmu	•	•	•	•	•	•	•	•	•	•	•

• *Yes* ○ *No*

7. Conclusion

NEmu and its associated programs *vnd* and *nemo* enable the creation and management of dynamic, heterogeneous and mobile virtual networks. They provide a good compromise between ease of use, low cost and realism. Such virtual networks can be distributed over several physical hosts and be controlled without any administrative rights. They can also evolve in real time with *nemo* by following a pre-calculated connectivity scenario.

We have compared *NEmu* to Mininet with regard to the Mosh experiment and have shown that the results are nearly identical thus demonstrating that *NEmu* can be used for similar purposes. The advantage being that it can be distributed over several physical machines unlike Mininet. This is important as *NEmu* uses full system emulation and not container based emulation. We have compared *NEmu* to JBotSim with regard to the AMiRALE experiment and the obtained results are similar, thus validating its use for virtual mobile devices' experimentation. *NEmu* can therefore overcome the cost of a physical testbed infrastructure while enabling the evaluation of real applications thanks to its low level emulation of network and system components. *NEmu* can also emulate mobile ad hoc networks which, as far as we know, is a unique feature among network emulators.

As envisioned by Conti *et al.* [71], the Internet of the future will be *polymorphic*, i.e., it will allow various specific networking environments to coexist, thanks to *virtualization* and *federation*. The extended concepts of NVE presented in this paper clearly embrace those two properties. Our software is a first step towards building such virtual networks. Its current use is mainly for research experimentation and teaching but its long term use could include production. The variety of supported *backends* could indeed lead our software to provide new network services.

Several next steps are already planned for our future work on *NEmu*. They consist in the following tasks by order of priority:

- the integration of migration capabilities for virtual machines in order to enable load balancing;
- the integration of new services inside the virtual router;
- the implementation of more sophisticated map generation algorithms;
- the improvement of the accuracy of the *vnd*.

8. References

[1] T. Henderson, M. Lacage, G. Riley, C. Dowell, J. Kopena, Network simulations with the ns-3 simulator, ACM SIGCOMM demonstration.

[2] A. Varga, et al., The omnet++ discrete event simulation system, in: Proceedings of ESM, Vol. 9, 2001.

[3] B. Yamini, D. Selvi, Cloud virtualization: A potential way to reduce global warming, in: Proceedings of IEEE RSTSCC, 2010, pp. 55–57.

[4] F. Bellard, QEMU, a fast and portable dynamic translator, in: Proceedings of USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.

[5] B. Lantz, B. Heller, N. McKeown, A network in a laptop: Rapid prototyping for software-defined networks, in: 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, pp. 19:1–19:6.

[6] V. Autefage, D. Magoni, Network emulator: a network virtualization testbed for overlay experimentations, in: 17th IEEE International Workshop on Computer-Aided Modeling Analysis and Design of Communication Links and Networks, 2012, pp. 38–42.

[7] N. Chowdhury, R. Boutaba, Network virtualization: state of the art and research challenges, IEEE Communications Magazine 47 (7) (2009) 20–26.

[8] KVM, Virtio, http://www.linux-kvm.org/page/Virtio.

[9] NBD, Network Block Device, http://nbd.sourceforge.net.

[10] Graphviz, Graph Visualization Software, http://www.graphviz.org (1988).

[11] B. Hubert, G. Maxwell, R. Van Mook, M. Van Oosterhout, P. Schroeder, J. Spaans, Linux advanced routing & traffic control, in: Ottawa Linux Symposium, 2003, pp. 213–222.

[12] R. Shingledecker, TinyCore Linux, http://tinycorelinux.net (2008).

[13] R. Davoli, Vde: Virtual distributed ethernet, in: Proc. of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities, 2005, pp. 213–220, http://vde.sourceforge.net.

[14] D. Magoni, Network Mobilizer, http://www.labri.fr/perso/magoni/nemo (2012).

[15] D. Magoni, Virtual Network Device, http://www.labri.fr/perso/magoni/vnd (2012).

[16] B. Dawes, al., Boost C++ Libraries, http://www.boost.org.

[17] N. Aschenbruck, E. Gerhards-Padilla, M. Gerharz, M. Frank, P. Martini, Modelling mobility in disaster area scenarios, in: 10th ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems, 2007, pp. 4–12.

[18] B. Liang, Z. J. Haas, Predictive distance-based mobility management for pcs networks, in: Proceedings of IEEE INFOCOM, 1999, pp. 1377–1384.

[19] X. Hong, M. Gerla, G. Pei, C.-C. Chiang, A group mobility model for ad hoc wireless networks, in: Proceedings of the 2nd ACM MSWiM, 1999, pp. 53–60.

[20] K. Winstein, H. Balakrishnan, Mosh: An interactive remote shell for mobile clients, in: USENIX Annual Technical Conference, 2012, pp. 177–182.

[21] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, N. McKeown, Reproducible network experiments using container-based emulation, in: 8th International Conference on Emerging Networking Experiments and Technologies, 2012, pp. 253–264.

[22] A. Aljunied, Evaluation of Mosh performance results, http://reproducingnetworkresearch.wordpress.com/2013/03/13/cs244-2013-evaluation-of-mosh-mobile-shell-performance-results.

[23] V. Autefage, S. Chaumette, D. Magoni, A mission-oriented service discovery mechanism for highly dynamic autonomous swarms of unmanned systems, in: Autonomic Computing (ICAC), 2015 IEEE International Conference on, 2015, pp. 31–40.

[24] A. Casteigts, Jbotsim: a tool for fast prototyping of distributed algorithms in dynamic networks, in: Proceedings of the 8th International Conference on Simulation Tools and Techniques, 2015.

[25] A. Ribiere, Emulation of obsolete hardware in open source virtualization software, in: Proceedings of the 8th IEEE INDIN, 2010, pp. 354–360.

[26] EMC, VMware, http://www.vmware.com (2004).

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: Proceedings of the 19th ACM SOPS, 2003, pp. 164–177.

[28] M. Bourguiba, K. Haddadou, G. Pujolle, Packet aggregation based network i/o virtualization for cloud computing, Computer Communications 35 (3) (2012) 309–319.

[29] J. Che, Y. Yu, C. Shi, W. Lin, A synthetical performance evaluation of openvz, xen and kvm, in: Proceedings of IEEE APSCC, 2010, pp. 587–594.

[30] P. Domingues, F. Araujo, L. Silva, Evaluating the performance and intrusiveness of virtual machines for desktop grid computing, in: Proceedings of IEEE IPDPS, 2009, pp. 1–8.

[31] Oracle, VirtualBox, https://www.virtualbox.org (2007).

[32] Microsoft, Hyper-V, www.microsoft.com/hyper-v-server (2008).

[33] J. Dike, A user-mode port of the Linux kernel, in: Proceedings of Linux Showcase and Conference, Vol. 2, 2000.

[34] Parallels, OpenVZ Linux Containers, http://wiki.openvz.org (2006).

[35] D. Lezcano, LXC, http://lxc.sourceforge.net (2008).

[36] C. Fillot, Dynamips, https://github.com/GNS3/dynamips (2007).

[37] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, Extending networking into the virtualization layer, Proceedings of ACM HotNets workshop.

[38] A. I. Sundararaj, A. Gupta, P. A. Dinda, Dynamic topology adaptation of virtual networks of virtual machines, in: Proceedings of the 7th LCR Workshop, 2004, pp. 1–8.

[39] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The click modular router, ACM Transactions on Computer Systems 18 (3) (2000) 263–297.

[40] S. Hemminger, et al., Network emulation with netem, in: Linux Conf Au, 2005, pp. 18–23.

[41] M. Carbone, L. Rizzo, Dummynet revisited, SIGCOMM Computer Communications Review 40 (2) (2010) 12–20.

[42] S. Agarwal, J. Sommers, P. Barford, Scalable network path emulation, in: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '05, IEEE, 2005, pp. 219–228.

[43] M. Eriksen, Trickle: A userland bandwidth shaper for unix-like systems, in: Proceedings of USENIX Annual Technical Conference, FREENIX Track, 2005, p. 43.

[44] G. Anuzelli, Dynagen, http://dynagen.org (2006).

[45] GNS3, Graphical Network Simulator, http://www.gns3.net (2007).

[46] B. Kneale, A. Y. De Horta, I. Box, Velnet: virtual environment for learning networking, in: Proceedings of the 6th Australasian Conference on Computing Education, Vol. 30, 2004, pp. 161–168.

[47] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kosti, J. Chase, D. Becker, Scalability and accuracy in a large-scale network emulator, in: The 5th ACM SIGOPS Operating Systems Review, 2002, pp. 271–284.

[48] M. Hashimoto, J. Bender, Vagrant, http://vagrantup.com (2010).

[49] A. Bavier, N. Feamster, M. Huang, L. Peterson, J. Rexford, In vini veritas: realistic and controlled network experimentation, in: Proceedings of ACM SIGCOMM, 2006, pp. 3–14.

[50] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, M. Wawrzoniak, Operating system support for planetary-scale network services, in: Proceedings of the 1st USENIX NSDI, 2004, pp. 253–266.

[51] X. Jiang, D. Xu, Violin: Virtual internetworking on overlay infrastructure, in: Proceedings of the 2nd Springer ISPA, 2003, pp. 937–946.

[52] M. Pizzonia, M. Rimondini, Netkit: easy emulation of complex networks on inexpensive hardware, in: Proceedings of the 4th IEEE TridentCom, 2008, pp. 1–10.

[53] J.-V. Lodo, L. Saiu, Marionnet, http://www.marionnet.org (2007).

[54] Y. Benchaib, A. Hecker, Virconel: A network virtualizer, in: Proceedings of the 19th IEEE MASCOTS, 2011, pp. 429–432.

[55] DIT, VNUML, http://dit.upm.es/vnumlwiki (2003).

[56] DIT, VNX, http://dit.upm.es/vnxwiki (2008).

[57] V. Perrier, Cloonix, http://clownix.net (2007).

[58] J. Ahrenholz, C. Danilov, T. Henderson, J. Kim, Core: A real-time network emulator, in: Proceedings of IEEE MILCOM, 2008, pp. 1–7.

[59] Z. Puljiz, M. Mikuc, IMUNES, http://imunes.tel.fer.hr (2003).

[60] FreeBSD, FreeBSD, http://www.freebsd.org (1993).

[61] C. Labs, OpenNebula, http://www.opennebula.org.

[62] S. Maier, D. Herrscher, K. Rothermel, Experiences with node virtualization for scalable network emulation, Computer Communications 30 (5) (2007) 943–956.

[63] Y. Liao, D. Yin, L. Gao, Network virtualization substrate with parallelized data plane, Computer Communications 34 (13) (2011) 1549–1558.
[64] M. Carbone, L. Rizzo, An emulation tool for planetlab, Computer Communications 34 (16) (2011) 1980–1990.
[65] R. Simmonds, B. W. Unger, Towards scalable network emulation, Computer Communications 26 (3) (2003) 264–277.
[66] N. Van Vorst, M. Erazo, J. Liu, Primogeni: Integrating real-time network simulation and emulation in geni, in: Proceedings of IEEE PADS, 2011, pp. 1–9.
[67] C. GARR, FEDERICA, http://www.fp7-federica.eu (2008).
[68] A. Willner, S. Albrecht, S. Covaci, F. Schreiner, T. Magedanz, S. Avessta, C. Scognamiglio, S. Fdida, U. Bub, Fantaastic: Sustainable management of future internet testbed federations, in: Network Operations and Management Symposium (NOMS), 2014 IEEE, 2014, pp. 1–4. doi:10.1109/NOMS.2014.6838379.
[69] S. Fdida, T. Korakis, H. Niavis, S. Salsano, G. Siracusano, The express sdn experiment in the openlab large scale shared experimental facility, in: Science and Technology Conference (Modern Networking Technologies) (MoNeTeC), 2014 International, 2014, pp. 1–7. doi:10.1109/MoNeTeC.2014.6995584.
[70] L. Baron, C. Scognamiglio, M. Rahman, R. Klacza, D. Cicalese, N. Kurose, T. Friedman, S. Fdida, Onelab: Major computer networking testbeds open to the ieee infocom community, in: Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on, 2015, pp. 3–4. doi:10.1109/INFCOMW.2015.7179314.
[71] M. Conti, S. Chong, S. Fdida, W. Jia, H. Karl, Y.-D. Lin, P. Mhnen, M. Maier, R. Molva, S. Uhlig, M. Zukerman, Research challenges towards the future internet, Computer Communications 34 (18) (2011) 2115 – 2134. doi:http://dx.doi.org/10.1016/j.comcom.2011.09.001.