Virtualization Toolset for Emulating Mobile Devices and Networks

Vincent Autefage, Damien Magoni, John Murphy

▶ To cite this version:

Vincent Autefage, Damien Magoni, John Murphy. Virtualization Toolset for Emulating Mobile Devices and Networks. IEEE/ACM International Conference on Mobile Software Engineering and Systems, May 2016, Austin, United States. 10.1145/2897073.2897087 . hal-01281879

HAL Id: hal-01281879

https://hal.science/hal-01281879

Submitted on 4 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Virtualization Toolset for Emulating Mobile Devices and Networks

Vincent Autefage
Univ. Bordeaux, LaBRI
351, Cours de la Liberation
33405 Talence, France
autefage@labri.fr

Damien Magoni
Univ. Bordeaux, LaBRI
351, Cours de la Liberation
33405 Talence, France
magoni@labri.fr

John Murphy
Univ. College Dublin, PEL
Belfield, Dublin 4
Dublin, Ireland
j.murphy@ucd.ie

ABSTRACT

With the ubiquitous usage of mobile devices, most communications are now impacted by the users' mobility. Therefore, applications and services must be designed to cope with network dynamics produced by those mobility patterns. Software research and development would benefit from taking device mobility into account. However, implementing and testing software on real devices is costly and cumbersome to perform. Virtualization is a widely used technique for avoiding these issues. In this paper, we propose three tools for creating and managing networks with mobile devices. Both network devices and user devices are emulated, the latter by using the QEMU system emulator. We implemented a virtual network device that can emulate access points and wireless interfaces, a real-time mobility engine that controls the dynamics of the connections and a control and management tool. Our toolset, called *NEmu*, can create both infrastructure and adhoc virtual networks for testing and evaluating applications with a fine-grained control over the network topology and link parameters. Results show that *NEmu* gives similar results as container-based virtualization and discrete event-based simulation.

CCS Concepts

•Networks → Logical/virtual topologies; Programmable networks;

Keywords

Mobile computing; mobile devices; network emulation; network virtualization

1. INTRODUCTION

Mobile devices' hardware is usually not based around the typical x86/x64 processor architecture commonly found on laptops, desktops and servers. Virtualization is thus a practical and efficient way to help develop software on those specific platforms. However, mobile devices also have a mobility behavior that was not relevant for regular applications running on laptops or desktops and that should now be taken into account. Setting up a hardware-based network testbed to do this is expensive and cumbersome. In particular, changing the network topology and the network parameters of a hardware testbed is time consuming and error-prone. Virtualizing the device is thus a first step but being able to virtualize the network where the device sits in would be a substantial improvement as network dynamics could then be emulated with ease. Developers would then be able to test and evaluate their applications in a dynamic environment where connectivity would vary over time depending on network conditions. In this paper, we propose a set of tools designed to create virtual networks for testing and evaluating applications running on mobile devices located in fixed infrastructure or mobile ad hoc networks with a complete control over the network topology, link properties (bandwidth, delay, bit error rate) and the mobility of nodes. The goal of our toolset is to enable the creation and management of reasonably sized virtual networks that are completely configurable and that can emulate the mobility of some or all of the nodes.

The contributions of our work are as follows:

- A detailed description of our toolset which is able to manage a distributed set of virtual nodes and links for emulating any arbitrary dynamic fixed and/or mobile network topology (Section 2).
- A validation experiment of a dynamic scenario by replicating with *NEmu* the *Mosh* [40] evaluation carried out with *Mininet* [27] (Section 3.1).
- A validation experiment of a mobile scenario by replicating with *NEmu* the *AMiRALE* [4] evaluation carried out with *JBotSim* [11] (Section 3.2).
- A description of the state of the art on related and previous work targeted at dynamic and mobile networking emulation tools (Section 4).

Our previous work concerning *NEmu* as a standalone tool for emulating fixed wired networks was published in [5]. The official website for *NEmu* can be found at: http://nemu.valab.net.

MobileSoft'16, May 16-17 2016, Austin, TX, USA

ACM ISBN 978-1-4503-4178-3/16/05.

DOI: http://dx.doi.org/10.1145/2897073.2897087

2. TOOLSET DESCRIPTION

2.1 Overall Design

The toolset is composed of three different programs:

- a C++ program called *virtual network device* (*vnd*), that can emulate links, hubs and switches but also access points and wireless interface cards (WIC),
- a C++ program called *network mobilizer* (*nemo*), which contains an event processor and a real-time mobility engine that controls the dynamics of the connections between the devices over time,
- and a Python program called *Network Emulator for Mobile Universes* (*NEmu*), which controls the two programs above as well as manages the virtual devices and the virtual machines running on QEMU.

NEmu also provides a Command-Line Interface (CLI) for the users of our toolset and as such, we also call the whole toolset *NEmu*.

NEmu is based on the concept of *Network Virtualization Environment* (NVE) introduced by Chowdhury and Boutaba in [12]. 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 guarantees 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 four 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 hot connections of nodes 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 Emulator for Mobile Universes

NEmu as a tool is a Python program consisting of 8000 lines of code which allows users to build dynamic virtual network infrastructures including mobile devices and mobile ad hoc networks. To this end *NEmu* is based on several 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 Nodes

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 `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* [23]. Moreover, it is easily possible to add some new services through a plug-in system available in *NEmu*. A `VRrouter` is running a customized image version of *TinyCore* which is a lightweight and highly configurable Linux distribution[1]. 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 Links

A *virtual link* for *NEmu* is an emulated network connection between *virtual nodes*. Many types of *virtual links* currently exist in *NEmu*. Here are the most relevant:

- A `VHub` is a virtual *hub* emulating a physical Ethernet hub and interconnecting several nodes.

[1] http://tinycorelinux.net

- A `VSwitch` is a virtual *switch* emulating a physical Ethernet switch and interconnecting several nodes.
- A `VRemote` represents a special point-to-point link which inter-connects a virtual element to a distant one (i.e., residing on another physical host) without breaking the isolation of the whole virtual network.
- A `VSlirp` represents a special point-to-point link which inter-connects a virtual node via a NAT'ed virtual interface to a real network interface card on the physical host.
- A `VTap` represents a special point-to-point link which inter-connects a virtual element to the physical machine through a Linux `tap` interface.
- A `VAirWic` represents a wireless interface connector.
- A `VAirAp` is a wireless access point for wireless/wired connections.

Virtual links are configured by the user in *NEmu* commands or scripts. *Virtual links* carry streams or frames from one virtual node to one or more others. The traffic is tunneled between virtual nodes by using so-called *backend* connections. *NEmu* uses our *vnd* program to emulate network devices. 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 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.

2.2.3 Sessions & Manager

As already said above, a *NEmu session* represents a complete configuration of a network topology residing on a physical host (i.e., storage, 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. The *NEmu manager* is the CLI 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;
- as a Python script launcher.

The *NEmu* manager provides remote access, through SSH connections, to manipulate *NEmu sessions* residing on other distant hosts. The network topology can be visualized through the *Graphviz* software suite [20].

2.3 Example of a Mobile Topology

We present in Figure 2 the Python script that generates a network topology with three drones as shown in Figure 1.

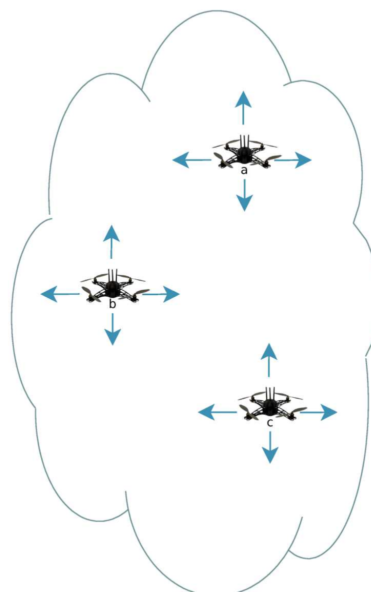

Figure 1: Example of a mobile ad hoc network topology with three drones.

```
# Initializing a NEmu session
InitNemu()

# Creating wireless cards
VAirWic('awic')
VAirWic('bwic')
VAirWic('cwic')

# Creating virtual nodes
VHost('a', hds=[VFs('drone.img', type='cow')]
         , nics=[VNic()])
VHost('b', hds=[VFs('drone.img', type='cow')]
         , nics=[VNic()])
VHost('c', hds=[VFs('drone.img', type='cow')]
         , nics=[VNic()])

# Linking nodes to wireless cards
Link('a', 'awic')
Link('b', 'bwic')
Link('c', 'cwic')

# Setting wireless cards in adhoc mode
SetAirMode ('awic', 'adhoc')
SetAirMode ('bwic', 'adhoc')
SetAirMode ('cwic', 'adhoc')

# Creating a mobility scenario with wireless cards
MobNemu('mob', nodes=['awic', 'bwic', 'cwic'])

# Setting the mobility scenario :
# . map of 1000x1000
# . duration : 120 seconds
# . number of mobility events (e.g., changing way) : 100
# . wireless properties changed every 5 seconds
GenMobNemu('mob', width=1000, height=1000
                , time=120, events=100, step=5)

# Starting all virtual components
StartNemu()

# Starting the mobility scenario
StartMobNemu('mob')
```

Figure 2: Corresponding *NEmu* script for generating the mobile topology.

2.4 Virtual Network Device

This section presents our *vnd* software. It is a program which is able to emulate network devices such as links, hubs, switches or access points. 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 emulate wireless interface cards in infrastructure and ad hoc modes as well as access points.

2.4.1 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. Data coming in or out of a *vnd* can be interpreted in three 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 II frames and each buffer can contain only one frame whose size shall be less or equal than the buffer's maximum size.
- `Pseudo 802.11`: data is considered as IEEE 802.11 frames, although only a pseudo header is used (i.e., not all header fields are present, only MAC addresses and to/from DS fields), and each buffer can contain only one frame whose size shall be less or equal than the buffer's maximum size.

A *vnd* can be set to one of the six different working modes available, depending on the network device that it emulates. The first four modes are typical network devices 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. Indeed, to our knowledge, no emulators currently support wireless cards. The main idea is to use the *vnd* to emulate an interface card of its own. Thus, in the last two modes, the *vnd* is not used as a separate network device but it is used in conjunction with a virtual machine to form a mobile end-user device (e.g., smartphone, tablet, autonomous vehicle). 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 `wic` interface. This creates a direct link between

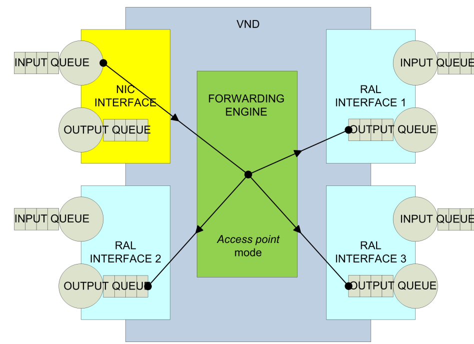

Figure 3: Access point mode.

the *vnd* and the virtual mobile device. Communications between wireless devices (access points and wireless cards) are virtualized by connections between interfaces called `ral` (for Radio Access Link). If the mobile node is considered using infrastructure mode (BSS or ESS), then the WIC, emulated by the *vnd*, will be connected to the access point by a `ral` interface. Because radio interfaces are virtualized, we could avoid broadcasting frames to all the neighbors in range but we do it to mimic the real behavior of a shared radio medium and to allow promiscuous listening.

The six possible modes and their behavior are:

- *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).
- *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.
- *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.
- *access point*: any frame going into the input of a `nic` interface is forwarded to any output of a `ral` interface and any frame going into the input of a `ral` interface is forwarded to the output of the `nic` interface (as shown on Figure 3).
- *infrastructure interface*: any frame going into the input of a `ral` interface is forwarded to the output of the `wic` interface leading to the mobile node itself, and any frame going into the input of the `wic` interface is forwarded to the output of any `ral` interface (as shown on Figure 4), one of them necessary leading to the access point when in infrastructure mode.
- *ad hoc interface*: same as above.

The last four modes only make sense when the data is interpreted as Ethernet or 802.11 frames, as MAC addresses are needed. In order to emulate the IEEE 802.11 protocol, a pseudo header is added to any frame sent by a `ral` interface and removed from any frame received by a `ral` interface.

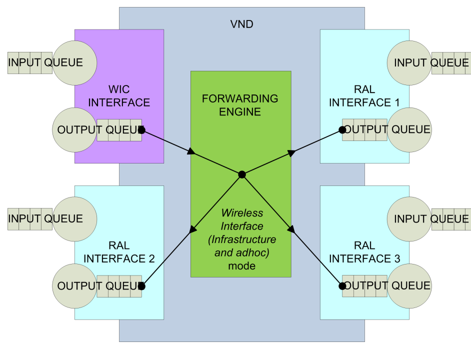

Figure 4: Infrastructure/ad hoc WIC mode.

In those modes, 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.

2.4.2 Implementation

The *vnd* is implemented in C++, contains around 6000 lines of code, and uses the following libraries:

- Boost libraries [14]: *thread*, *chrono*, *system*, *regex*, *bind*, and the asynchronous input/output library called *asio*.
- SSL libraries (for creating SSL backend connections).
- Linux-specific libraries: VDEplug (for connecting to VDE switches).

The *vnd* can be used directly through a rudimentary CLI or can be entirely controlled by *NEmu* which then acts as the user interface. It is a lightweight program using around 250 KBytes in RAM and it is portable on the majority of UNIX and Windows variants. The code is open source[2] and licensed under the LGPLv3.

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 virtual machine or a *vnd*. This tie is implemented by emulators in more or less flexible ways. In order to support the dynamic features presented at the beginning of this section, the *vnd* implements this tie in a flexible way by separating the virtual interface from the endpoint. This tie can be dynamically created or destroyed. 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.

[2]http://www.labri.fr/perso/magoni/vnd

2.5 Network Mobilizer

Our toolset can emulate mobile devices and networks. Thus, it is possible to create a virtual network topology that evolves over time such as a Mobile Ad-hoc Network (MANET). In order to manage mobility, *NEmu* uses the *nemo* mobility engine. *nemo* is a lightweight C++ program which can generate connectivity scenarios for mobile devices and networks. A connectivity scenario is a time-stamped list of wireless link connection and disconnection events between wireless devices (e.g., mobile nodes, access points). In order to emulate device mobility and network dynamics, *nemo* leverages some features of our *vnd* software, such as the ability to create or delete virtual links on-the-fly and to dynamically modify link characteristics. *nemo* is able to send orders to *NEmu* in real time for emulating the connectivity changes between mobile nodes by creating, destroying or changing the characteristics of the links at the appropriate time. *nemo* contains two parts that are executed sequentially in that order: the first part is based on a simulated-time discrete-event processor and the second part is based on a real-time scheduler.

2.5.1 Discrete Event Processor

The discrete event processor contains a *simulated* time scheduler which is the heart of the simulation part of *nemo*. The processor can generate connectivity scenarios that can be later fed to the real-time scheduler. Three steps are necessary to generate a connectivity scenario. First, the generation of a map (i.e., size, granularity). Second, the generation of a mobility scenario on this map. A mobility scenario contains mobility events at a given time (i.e., start time, start position, velocity and acceleration). Third, the generation of a connectivity scenario (i.e., start time, device ids, connection status (start, update, stop), data rate, distance) from the aforementioned mobility scenario. At each step, the results of the step can be saved on disk in order to be loaded at a later time and thus to avoid re-computation. The discrete event processor runs the mobility scenario and at each *time interval*, whose value is initially set by the user, it computes the distances and the possible wireless connections between all the pairs of mobile nodes. The inverse of the time interval is defined as the *sampling frequency* of the scenario. 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. For the moment, *nemo* generates rectangular maps and purely random mobility scenarios which is useful for carrying out functional tests. *nemo* is also capable of importing *ns-2* formatted mobility files. In the future, *nemo* will be able to generate and load more elaborate 2D or 3D maps containing, for instance, attenuation information to represent physical obstacles and/or

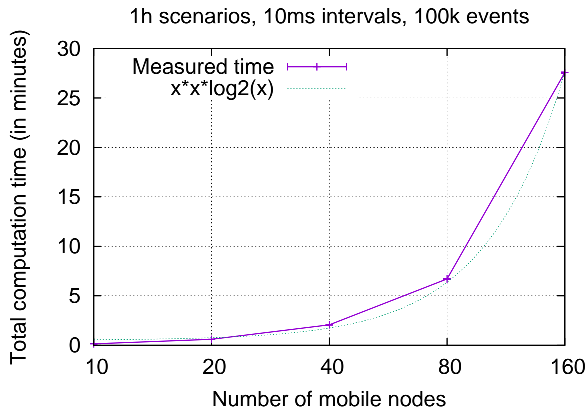

Figure 5: Computation complexity of the discrete event processor.

relief elevation on the map.

The processor requires the user to make a trade-off between the temporal precision provided by the sampling frequency (i.e., the time interval between each computation of the overall connectivity), the computation time (which is dependent on the number of nodes) and the number of events detected. Indeed, a low frequency may miss some short-lived connectivity events while a high frequency may add much computation time without adding any more events. The selection of the sampling frequency is thus in part dependent on the respective speeds of the nodes. Also, as the scenario will later on be executed in real time, a very high sampling frequency would be meaningless. As a rule of thumb, a time precision of 1 ms is probably the best that can be achieved in real time, thus an upper limit for the sampling frequency would be around 1000 computations per second.

The code that evaluates the connectivity between all nodes at each time interval has a worst-case computation complexity of $O(n^2 \times log_2(n))$, with n being the number of nodes in the scenario. However, as n is usually small and as the processor is written in C++ for performance, the total computation time remains usually below the simulated time duration of the scenario. Plus, this computation is done only by the processor in the first phase of the *nemo* usage in order to generate the connectivity scenario. It is not done when launching the real-time scheduler.

To assess the computation complexity of the connectivity computation, we have generated random scenarios for 5 different network sizes, with a time interval of 10 ms (i.e., a sampling frequency of 100 calls/second), a scenario duration of 1 hour and a total number of mobility events of 100000. The results are shown on Figure 5. Each point is the average of 10 runs. As standard deviation values are all below 1.5% of their respective means, the y-error bars cannot be seen of the plot. A fit of the plot with the function `f(x) = a + b*log2(x)*x**2` has been made by using the nonlinear least-squares Marquardt-Levenberg algorithm provided by `gnuplot` and is also shown on the figure. Experimental results are coherent with the fitting function. As can be seen, a typical 1-hour connectivity scenario with 40 nodes will take approximately 2 minutes to be generated from a mobility scenario containing 100k events, while a 160 nodes one will take around 28 minutes.

2.5.2 Real-time Scheduler

The real-time scheduler is the heart of the emulation part of *nemo*. It executes any connectivity event at its exact timestamp, set with respect to the start of the scenario. The discrete event processor sets the start time of the mobility scenario at 0 second. However, when executed in real-time by the scheduler, the start time of the connectivity scenario is set to the current clock time and all timestamps are then offset with this value. The temporal precision available in the real-time scheduler is given by the *high resolution clock* provided by the Boost chrono library (which is itself based on the system time library). The accuracy thus depends on the OS of the physical host. For most modern OS, accuracy is typically acceptable for values no smaller than a few milliseconds. However, events themselves will of course always fall on multiples of the sampling time interval set during the processing of the mobility scenario by the simulated time scheduler of the discrete event processor.

In a virtual mobile network, one *vnd* is used to emulate each wireless network interface card (WIC) as explained in Section 2. Thus, there is one *vnd* per virtual mobile node and inside it are instantiated the backend links towards the mobile device's virtual machine and all the other *vnd* of the other mobile nodes. *NEmu* transmits the orders of the user (e.g., *start, stop*) to *nemo*. When the real-time scheduler is running, *NEmu* also recovers the connectivity events generated by *nemo* and retransmits 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/resumed at any time by the user.

As opposed to *NEmu*, *nemo* is not distributed (i.e., , only one program instance of *nemo* may be running a given connectivity scenario). Indeed, *nemo* must know the positions of all the devices at any given time. It is recommended that all the virtual machines emulating mobile devices should reside on the same physical host in order to avoid degrading performance, as sending orders over the physical network may add significant delay. Some network backends, such as *sockets*, used by the *vnd* introduce delays and may reduce the temporal precision even on a single physical machine. The latter will be at best of the order of the millisecond as previously mentioned. As our toolset targets the development and testing of non-safety-critical applications, we assume that this precision is enough.

2.5.3 Implementation

nemo can be used directly through a rudimentary CLI or can be entirely controlled by *NEmu* which then acts as the user interface. *nemo* is implemented in C++, contains around 3000 lines of code, and is using the following Boost libraries: *thread*, *chrono* and *system*. Remote control of *nemo* also requires the *regex*, *bind* and *asio* libraries. It is a lightweight program using around 1 MByte in RAM and it is portable on the majority of UNIX and Windows variants. The code is open source[3] and licensed under the LGPLv3.

In order to emulate wireless communications between `ral` links, *nemo* implements the two-ray ground-reflection model and mimics the bitrates *vs* sensitivity levels of three WICs: Cisco Aironet, 3Com Xjack and Lucent Wavelan. *nemo* does not yet currently take into account the MAC layer access and contention mechanisms.

[3]http://www.labri.fr/perso/magoni/nemo

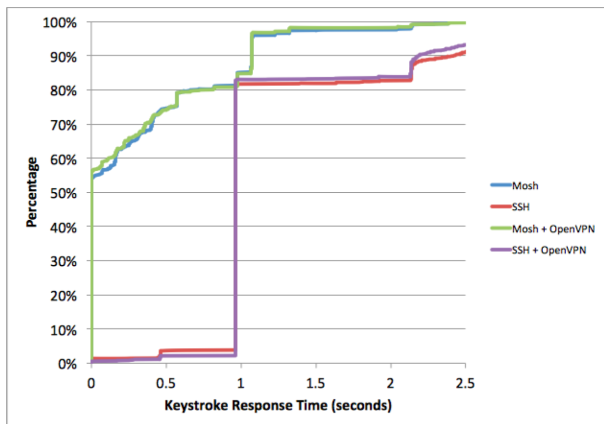

Figure 6: Original Mosh results with Mininet.

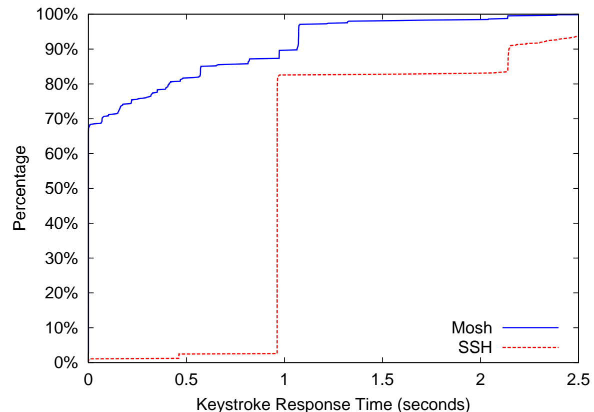

Figure 7: Mosh results with *NEmu*.

3. EXPERIMENTATION

In this section, we present two experiments that illustrate the accuracy of our tools used in two different scenarios. The first one evaluates roaming connectivity issues and compares Mosh (Mobile shell) to SSH. The second one evaluates data traffic in a mobile ad hoc network composed of moving ground robots running the AMiRALE collaboration system.

3.1 Dynamic Network Experiment

In order to validate the accuracy of experimentation results obtained with *NEmu*, we reproduce a performance benchmark of *Mosh* [40]. 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 keystroke response time for Mosh and SSH. This experiment has been previously carried out on *Mininet* [27], 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 [2] and which has been officially supported by the Mosh developers. In this experimentation, the server is connected to a switch through an emulated WiFi network, and the client through an emulated 3G network. The authors consider the following experimental network conditions for the 3G connection: packet loss rate: 0.01, bandwidth: 1 Mbps, delay: 450 ms; and for the WiFi connection: packet loss rate: 0.08, bandwidth: 25 Mbps, and delay: 30 ms.

Thanks to our *vnd* program, we configure the network properties as detailed above. Original results obtained with *Mininet* are presented in Figure 6. Our results are illustrated in Figure 7. We have not emulated the OpenVPN scenarios as they were very already close to the regular ones in the original experiment. We can notice that results are nearly identical. Those results imply that *NEmu* can offer a degree of realism similar to *Mininet*.

3.2 Mobile Network Experiment

In order to validate the accuracy of experiments with mobile devices performed with *NEmu*, we compare several performance results obtained by simulation and emulation on a multi-agents system called *AMiRALE* [4]. Emulations are performed with *NEmu* while simulations are carried out with *JbotSim* [11], 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 (e.g., drones, ground robots) to perform common tasks collaboratively. This system is only based on asynchronous one-way broadcast messages. Consequently, if the density of nodes is very high, network congestion can appear; this phenomenon is commonly called the *broadcast storm problem* [31]. In order to avoid, or at least to reduce the effects of this phenomenon, AMiRALE includes several user mechanisms called *filters* which enable, notably, to limit the re-emission of the same message.

We evaluate the output data rates generated by AMiRALE as a function of the filters' configuration. The application scenario consists in a team of ground robots which has to collect a given number of items of garbage in a park. Garbage and robots are placed randomly in the park at the beginning of the scenario. Mobile ground robots are moving by following the *random way-point* mobility model [9]. Each robot is specialized, which means that it can only clean one kind of garbage (e.g., glass, paper, compost). When a robot finds an item of 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 item of garbage. This strategy enables a robot to clean an item of garbage which has been discovered by another robot. Missions are broadcast by robots every 5 seconds. When a robot collect an item of garbage, the relative mission is marked as *end*. We have run many instances of our scenario, both with *NEmu* and JbotSim with different numbers of garbage items in the park and several filter configurations.

Figure 8 shows the results of our experiment as a function of the number of items of garbage in the park (thereafter called *targets*). Here, no filter is used which means that a mission is broadcast from its creation to the end of the scenario (i.e., when no garbage remains in the park). 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 a unique mission and divide the result by the frequency of broadcasts. This result is provided on the theoretical plot. This figure shows that theoretical, simulation and emulation results are very similar which implies that *NEmu* provides coherent performance results on

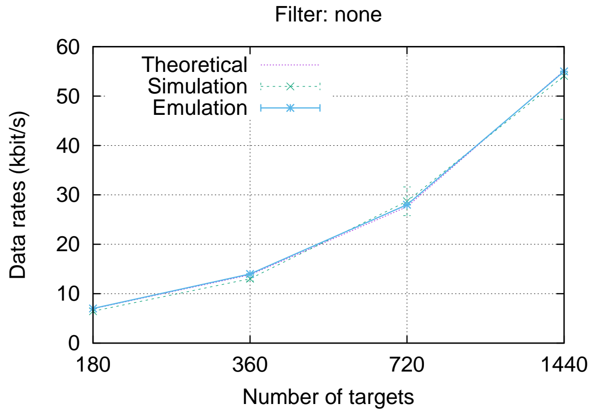

Figure 8: Data rates obtained without filter *vs* the number of targets.

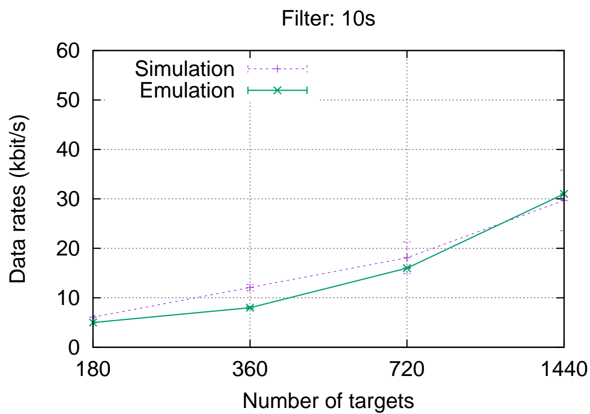

Figure 9: Data rates obtained with a 10s filter *vs* the number of targets.

this mobile scenario.

Figure 9 and Figure 10 show the same scenario results, where filters are configured to avoid a mission to be broadcast if it has been completed (i.e., marked as *end*) for more than 10 seconds and 60 seconds respectively. Since the creation pace of the missions changes as a function of the mobility of the robots and the initial placement of the garbage and robots, we cannot provide any theoretical result for those plots. We can see that simulation and emulation experiments provide similar results with overlapping confidence intervals except at points for 360 targets, where the simulations give higher data rates for reasons not yet investigated.

Collectively, those results imply that network emulation with *NEmu* can offer a satisfying degree of realism both with fixed and mobile experimentation.

4. RELATED WORK

Network virtualization has been proposed and studied for quite some time and many tools are currently available. *Dynamips* [17] is a CISCO hardware emulator. *Dynagen* [3] is for Dynamips, the equivalent of what *NEmu* is for QEMU. Dynagen manages fleet of Dynamips machines and

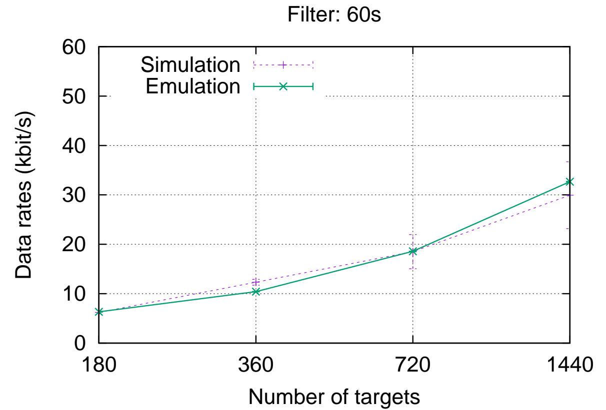

Figure 10: Data rates obtained with a 60s filter *vs* the number of targets.

their inter-connections. However, the network dynamics of the links of the topology are strongly limited (mainly on/off by the user) and adding network services is quite difficult. *GNS* [19] 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 spread on several physical machines and adding network services is as complicated as in Dynagen. Finally *GNS* is hardly usable without any graphical interface making difficult the creation and management of a complex network. *Velnet* [25] 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* [38] 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 a unique physical machine which disables the community aspect. *Vagrant* [22] 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 does not rely on standard ways of addressing and routing. *VINI* [7] is a distributed virtual network which overhangs the *PlanetLab* testbed [6] 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* [24] 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* [34] relies also on UML and VDE switches which do not require any administrative rights. Such a system cannot be distributed. *Marionnet* [29] is a virtual environment dedicated to teaching. It provides several network services and the community aspect but relies on UML. *Virconel* [8] uses OpenVZ virtual machines which also strongly limits operating systems for nodes. Moreover, the topology is static during runtime and the interconnections be-

tween virtual machines are made in the host kernel which requires special rights on the physical system. *VNUML* [15] is a static virtual system relying on UML and which requires administrative rights. *VNX* [16] 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* [32] is a dynamic virtual network environment. Nevertheless, it does not include link property mechanism. *Mininet* [27] is a Container-Based Emulator which allows to create a custom and dynamic topology on a unique physical host. It is quite efficient as it does not use full virtualization but that limits the creation of heterogeneous networks (with ARM devices for instance) [21]. *OpenNebula* [26] 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 media are centralized and are accessible through the network which implies the use of a NAS or NFS. *PdP* [28] 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* [10] is a well known emulation tool over PlanetLab. It also relies on DummyNet which strongly limits the flexibility of this solution. *IP-TNE* [36] 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* [6], *GENI* [39] or *FEDERICA* [18] supply virtual infrastructure built on top of *slices* of hardware owned by third parties. Thus, the user needs a specific account, must comply to a usage policy and has to use the tools, services and APIs of these testbeds. They are useful for emulating long distance links as their physical infrastructure is usually nationwide.

Virtualizing *mobile* devices and networks is much more difficult than regular networks and only a handful of tools are currently available. *CORE* [1] is a graphic tool which enables to emulate virtual mobile networks. This system relies on a framework called IMUNES [35] which runs over the FreeBSD 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. However, it may happen that, due to the processing inherent in the execution of simulation events, the simulator cannot keep up with real-time execution as indicated in the ns3 manual. Furthermore LXC containers cannot emulate mobile devices typically running on different processor types (i.e., ARM instead of Intel based). *NET* [30] 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.

Similarly to our *vnd* software, some other existing tools enable the creation of emulated customized virtual switches. *VDE* [13] is a virtual switch which inter-connects virtual machines through the shared memory system inside the Linux kernel. VDE does not include any mechanism in order to manipulate link properties like bandwidth, delay, etc. *Open vSwitch* [33] 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 thus lacking flexibility in terms of backend connections. *Vnet* [37] 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. In addition, all these virtual switch solutions do not enable access point emulation.

5. CONCLUSIONS

Our toolset composed of *NEmu* and its associated programs *vnd* and *nemo* enable the creation and management of virtual heterogeneous and mobile devices and networks. They provide a good compromise between ease of use, low cost and acceptable realism. Such virtual networks can 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 JBotSim with regard to the Amirale experiment, and have shown in both cases that the results are nearly identical thus demonstrating that *NEmu* can be used for similar purposes. The advantage being that *NEmu* can emulate mobile devices such as smartphones and tablets thanks to the QEMU system emulator (which can emulate ARM processors for instance) unlike Mininet which uses container virtualization. *NEmu* can therefore overcome the cost of a physical testbed infrastructure while enabling the evaluation and testing of real applications thanks to its low level emulation of network and system devices. *NEmu* can also emulate mobile ad hoc networks which, as far as we know, is a unique feature among network emulators using full system virtualization. Several next steps are already planned for our future work on *NEmu*. They consist in the following tasks by order of priority:

- The improvement of the performance and accuracy of the *vnd*.
- The implementation of more sophisticated map generation algorithms inside *nemo*.
- The implementation of a more realistic wireless card emulation, taking into account better radio propagation models, CSMA/CA access method, etc.
- The adaptation of *NEmu* in order to use other system emulators (e.g., VirtualBox, Dynamips).

NEmu is a free open-source software available under the LGPLv3 license. Tutorials as well as a full documentation can be found on its website. The latest version of the source code can be downloaded from the *NEmu* website[4].

6. ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Research Centre (www.lero.ie).

7. REFERENCES

[1] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim. Core: A real-time network emulator. In *Proc. of IEEE MILCOM*, pages 1–7, 2008.

[4] http://nemu.valab.net

[2] A. Aljunied and A. Atreya. Evaluation of mosh performance results. http://reproducingnetworkresearch.wordpress.com/2013/03/13/cs244-2013-evaluation-of-mosh-mobile-shell-performance-results, 2013.
[3] G. Anuzelli. *Dynagen*, 2006. http://dynagen.org.
[4] V. Autefage, S. Chaumette, and D. Magoni. A mission-oriented service discovery mechanism for highly dynamic autonomous swarms of unmanned systems. In *Proc. of the Int'l Conf. on Autonomic Computing*, pages 31–40, 2015.
[5] V. Autefage and D. Magoni. Network emulator: a network virtualization testbed for overlay experimentations. In *Proc. of the 17th Int'l Workshop on Computer-Aided Modeling Analysis and Design of Communication Links and Networks*, pages 38–42, 2012.
[6] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of the 1st USENIX NSDI*, pages 253–266, 2004.
[7] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In vini veritas: realistic and controlled network experimentation. In *Proc. of ACM SIGCOMM*, pages 3–14, 2006.
[8] Y. Benchaib and A. Hecker. Virconel: A network virtualizer. In *Proc. of the 19th MASCOTS*, pages 429–432, 2011.
[9] C. Bettstetter, H. Hartenstein, and X. Pérez-Costa. Stochastic properties of the random waypoint mobility model. *Wireless Networks*, 10(5):555–567, 2004.
[10] M. Carbone and L. Rizzo. An emulation tool for planetlab. *Computer Communications*, 34(16):1980–1990, 2011.
[11] A. Casteigts. Jbotsim: a tool for fast prototyping of distributed algorithms in dynamic networks. In *Proc. of the 8th Int'l Conf. on simulation tools and techniques*, 2015.
[12] N. Chowdhury and R. Boutaba. Network virtualization: state of the art and research challenges. *IEEE Communications Magazine*, 47(7):20–26, 2009.
[13] R. Davoli. VDE: Virtual Distributed Ethernet. In *Proc. of the 1st Int'l Conf. on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220, 2005.
[14] B. Dawes and al. *Boost C++ Libraries*. http://www.boost.org.
[15] DIT. *VNUML*, 2003. http://dit.upm.es/vnumlwiki.
[16] DIT. *VNX*, 2008. http://dit.upm.es/vnxwiki.
[17] C. Fillot. *Dynamips*, 2007. https://github.com/GNS3/dynamips.
[18] C. GARR. *FEDERICA*, 2008. http://www.fp7-federica.eu.
[19] GNS3. *Graphical Network Simulator*, 2007. http://www.gns3.net.
[20] Graphviz. *Graph Visualization Software*. http://www.graphviz.org.
[21] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *8th Int'l Conf. on Emerging Networking Experiments and Technologies*, pages 253–264, 2012.
[22] M. Hashimoto and J. Bender. *Vagrant*, 2010. http://vagrantup.com.
[23] B. Hubert, G. Maxwell, R. Van Mook, M. Van Oosterhout, P. Schroeder, and J. Spaans. Linux advanced routing & traffic control. In *Ottawa Linux Symposium*, pages 213–222, 2003.
[24] X. Jiang and D. Xu. Violin: Virtual internetworking on overlay infrastructure. In *Proc. of the 2nd ISPA*, pages 937–946, 2003.
[25] B. Kneale, A. Y. De Horta, and I. Box. Velnet: virtual environment for learning networking. In *Proc. of the 6th Australasian Conf. on Computing Education*, volume 30, pages 161–168, 2004.
[26] C. Labs. *OpenNebula*. http://www.opennebula.org.
[27] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *9th Workshop on Hot Topics in Networks*, pages 19:1–19:6, 2010.
[28] Y. Liao, D. Yin, and L. Gao. Network virtualization substrate with parallelized data plane. *Computer Communications*, 34(13):1549–1558, 2011.
[29] J.-V. Lodo and L. Saiu. *Marionnet*, 2007. http://www.marionnet.org.
[30] S. Maier, D. Herrscher, and K. Rothermel. Experiences with node virtualization for scalable network emulation. *Computer Communications*, 30(5):943–956, 2007.
[31] S. Ni, Y. Tseng, Y. Chen, and J. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proc. of the Int'l Conf. on Mobile Computing and Networking*, pages 151–162, 1999.
[32] V. Perrier. *Cloonix*, 2007. http://clownix.net.
[33] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. *Proc. of ACM HotNets workshop*, 2009.
[34] M. Pizzonia and M. Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *Proc. of the 4th TridentCom*, pages 1–10, 2008.
[35] Z. Puljiz and M. Mikuc. *IMUNES*, 2003. http://imunes.tel.fer.hr.
[36] R. Simmonds and B. W. Unger. Towards scalable network emulation. *Computer Communications*, 26(3):264–277, 2003.
[37] A. I. Sundararaj, A. Gupta, and P. A. Dinda. Dynamic topology adaptation of virtual networks of virtual machines. In *Proc. of the 7th LCR Workshop*, pages 1–8, 2004.
[38] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *ACM Operating Systems Review*, pages 271–284, 2002.
[39] N. Van Vorst, M. Erazo, and J. Liu. Primogeni: Integrating real-time network simulation and emulation in geni. In *Proc. of PADS*, pages 1–9, 2011.
[40] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *USENIX Annual Technical Conf.*, pages 177–182, 2012.