Marionnet: a virtual network laboratory and simulation tool

Jean-Vincent Loddo
LIPN
99, avenue J.B. Clément
93430 Villetaneuse, France
loddo@lipn.univ-paris13.fr

Luca Saiu
LIPN
99, avenue J.B. Clément
93430 Villetaneuse, France
saiu@lipn.univ-paris13.fr

ABSTRACT
We present Marionnet1, a high-level simulation tool allowing to accurately reproduce the behavior of physical computer networks made by computers, hubs, switches and routers; virtual machines run unmodified GNU/Linux binaries for the x86 architecture at nearly native speed.

Individual virtual devices can be dynamically created, destroyed or modified while the rest of the virtual network is running, providing many opportunities for experimentation without any need for clumsy hardware setup.

Marionnet has a very intuitive graphical user interface also suitable to inexperienced users, and is being used with success for teaching computer networks at Université Paris 13.

We believe its adoption can be extended to other specialists in the field of network, in order to ease the design and debug of computer networks, applications and protocols.

This work presents Marionnet from a user point of view, complementing its description with a practical use scenario showing an example of how the application could be profitably employed in the industry.

Categories and Subject Descriptors
I.6.7 [Simulation Support Systems]; I.6.3 [Applications]; H.4 [Information Systems Applications]; Miscellaneous

General Terms
Experimentation

Keywords
simulation, computer networks, GUI, interactive, UML

1. INTRODUCTION
One of the main purposes of Marionnet is enabling users to test computer network cablings, network protocols, services and applications. The application allows to easily define and configure a virtual network complete with hubs, switches and routers, and then to actually run unmodified GNU/Linux x86 software on virtual computers.

Marionnet is a free software application running on GNU/Linux built on UML2 and VDE3.

Not only Marionnet benefits from the efficiency and stability of Linux, but its free software license has been the very reason making Marionnet possible in the first place: we actually modified and adapted User Mode Linux ([17], [5], [6]) and VDE ([4]).

Network devices like hubs, switches, routers and cables also need to be simulated to be able to closely reproduce the behavior of a physical network.

In this paper we describe Marionnet from the point of view of a user, providing only some brief hints to the implementation. [12] describes the inner workings of Marionnet at a reasonable level of detail.

2. BASIC USAGE
Marionnet is an interactive software with a graphical user interface based on GTK+ ([13]). If exposes to the users the usual project-oriented metaphore: in order to start using the application the user can create a new empty project or open an existing one. In both cases a dialog window pops up, asking to choose a file name. Once a project is created or opened the user can freely navigate through the functionalities offered by the interface: the device palette, the graphical network representation (henceforth the network graph) and the panes providing advanced functionalities (Interfaces, Anomalies, Filesystem history) become active. Figure 1 shows the interface as it looks right after the creation of a new project.

1Marionnet is financed by Université Paris 13.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Proceedings of SimulationWorks March 4, 2008, Marseille, France
Copyright 2008 ACM X-XXXX-XX-X/XX/XX ...$5.00.

2User Mode Linux, by Jeff Dike et al., is a port of the Linux kernel to the GNU/Linux system consisting in a guest kernel which runs on the host operating system as a normal user-level process, but exposing the complete Linux kernel API to its guest processes.

Even at the cost of an excessive simplification, in this paper we make the conscious decision of not distinguishing among such a solution, emulation, simulation and paravirtualization: the internal architecture of the software components we build upon is not relevant for the application.

3Virtual Distributed Ethernet, by Renzo Davoli et al., is a low-level communication infrastructure which we heavily reused for implementing virtual devices. Marionnet internals, including our modification to VDE, are described in [12].
The interface how it looks right after the creation of a new project

Figure 1:

2.1 Devices palette

Network devices can be created, modified and controlled from the device palette within the Hardware pane.

The user is freed from the burden of physically placing device icons in the network graph two-dimensional space: placement is automatic, but several parameters (for example the length of edges and the size of icons) can be tuned if needed (see Figure 1, on the right).

The network graph is automatically updated at each device state change (such as startup, pause or resume) to reflect the current state.

The device palette offers two kinds of functionalities:

1. virtual network editing, including definition, modification and removal of individual devices.
2. virtual network control features: each device can be started-up, paused, resumed, shutdown and powered-off.

Eight types of virtual network components are currently provided:

1. computer
2. Ethernet hub
3. Ethernet switch
4. IP router
5. “straight” Ethernet cable
6. crossover Ethernet cable
7. Ethernet cloud
8. Ethernet socket

We are now going to briefly review them, showing how the devices of each type can be tuned.

2.2 Virtual computer

The virtual computer device represents a computer running a GNU/Linux operating system. Just like a physical computer a virtual computer can be off or running. As an “extension” to what it is possible in a physical network we also provide the possibility of pausing computers: in the paused state computers do not react to incoming messages. Pausing allows to experiment with dynamic routing protocols, making a machine temporarily unreachable.

Different icons represent a virtual computer in off, running and paused state in the network graph:

Parameters

When a user adds or modifies a virtual computer the dialog window shown in Figure 2 allows to choose its name and to set several parameters.

The following parameters are hardware-related:

- **RAM**: the quantity of RAM reserved by the host system to the guest, seen by the guest as its physical memory. The default 48Mb setting is adequate for comfortably running even graphical applications like Firefox or Ethereal/Wireshark.
  
  Of course swapping is also provided on virtual machines.

- **Number of Ethernet cards** (1 by default)

The user can also tune the software running on the virtual machine:

- the particular GNU/Linux distribution, chosen among the ones provided for guest systems by the Marionnet installation\(^4\).

- a variant represents a modification to the filesystem of the selected GNU/Linux distribution. The COW (Copy On Write) technology supported by UML allows a very efficient implementation of this feature, involving a single sparse file containing only the blocks which are different from the unmodified distribution. A typical COW file takes only a few megabytes of disk space on the host.

- setting the terminal type allows to choose whether the machine displays its X clients using the host X server or an Xnest server of its own.

- the particular guest kernel, chosen from several versions of Linux compiled with different features enabled.

- when a machine has its terminal type set to host the user interacts with it in text mode using a simple virtual terminal window, and can launch graphical applications which draw their clients on the same X display.

\(^4\)It is quite easy to add support for more distributions: see the UML documentation for more information about how to create guest filesystem images.
2.3 Virtual hub

A *hub* is a very simple electronic device reproducing the signals it receives from one of its port into all the other ports. All the network nodes connected to a hub belong to the same Ethernet collision domain.

Despite nowadays being mostly disregarded as obsolete variants of switches (see Section 2.4), hubs are often convenient for intercepting and analyzing network traffic: this can be easily realized by running a *sniffer* application such as *wireshark* or *tcpdump* on a virtual computer directly connected to a hub.

VDE (see [4] and [3]) allows to simulate an Ethernet switch, but some modifications to its C source code have been necessary in order to realize the behavior needed in Marionnet.

The following icons represent a hub in its **off**, **running** and **paused** state:

```
[Image]
```

*Parameters*

A very simple dialog window allows to set the **name** and few other parameters of each virtual hub:

- an optional **label** to be shown on the network graph near the hub icon,
- the **number of Ethernet ports**, 4 by default.

2.4 Virtual switch

An Ethernet *switch* also allows to relay several Ethernet frames through a network, but differently from a hub it outputs data only to the intended receiving node.

A virtual switch is represented by the following icons in Marionnet:

```
[Image]
```

*Parameters*

The dialog window allowing to edit a switch has no notable differences from the one for editing a hub (see Section 2.3): only a **name**, an optional **label** (to be shown in the network graph) and the **number of ports** can be chosen.

2.5 Virtual router

An IP *router* is a device directing packets from a local network to another local network. The main purpose of a router is to find the next node of a network through which a packet should be sent to reach its final destination in the minimum time. It is worth to emphasize that, differently from hubs and switches which operate at the **link layer**, routers work at the **network layer**, and their behavior is considerably more complex. *Routing tables* can be set either statically or dynamically.

Virtual routers are implemented with the *Quagga* software (see [10], [2]; Quagga is derived from the Zebra project: [http://www.zebra.org/](http://www.zebra.org/) *running on a UML virtual machine*). Quagga allows both static and dynamic routing (in the latest case supporting different protocols such as RIP, OSPF, BGP and ISIS).

Each router port can be configured from the Marionnet **Interface** pane.

Once it is started, a virtual router can also be set up with the *teutet* protocol.

Virtual routers are identified by the following icons in the network graph:

```
[Image]
```

*Parameters*

Router interfaces can be configured from the **Interfaces** pane.

The dialog window allowing to setup or modify routers is similar to the ones for hubs and switches.

2.6 Virtual cable

Ethernet cables allow to physically connect nodes in the network. Marionnet simulates the most common sort of cables deployed today, twisted pair with “RJ45” connectors. In any case we tend to abstract over such low-level details, which are typically not relevant at the high-level where Marionnet works.

Marionnet currently supports only fully wired virtual networks; support for wireless networks is planned for a future version. Some work in this direction using UML already exists: see [9]. Bandwidth limitation (allowing for example to restrict bandwidth to 100Mb/s over some cables and
“Straight” cable editing dialog

Figure 3: “Straight” cable editing dialog

to 1000Mb/s on others) would be easy to implement, and is planned for the next version. The distinction between “straight” and crossover cables can be enforced by setting a global option.

The control actions provided by the devices palette are limited to connect and disconnect in the case of cables. Cables are always connected by default, but they can be disconnected and reconnected at will by the user, as it is common while making tests on the network at several different levels.

Cables are represented in the network graph as solid lines when connected, and as dashed lines when disconnected.

Parameters
The user can set the name, and optional label to be shown in the network graph, and the identity of each endpoint for each cable, as seen in Figure 3.

2.7 Virtual Ethernet “cloud”

An cloud represents an Ethernet network composed of hubs, switches and cables, with exactly two endpoints an unspecified internal structure. The only externally observable effects of a cloud consist in delays and other anomalies in the relaying of frames from one endpoint to the other.

This “device” is particularly useful for the simulation of dynamic routing (see Section 2.5).

Parameters
The cloud definition dialog is not particularly interesting, as it only allows to set a name and an optional label to be shown in the network graph.

Anomalies can be set from the Defects pane with a very fine level of detail; see Section 5.2.

2.8 Virtual external socket

Using the components presented up to this moment it is possible to build a virtual network made of computers, hubs, switches, routers and clouds connected by “straight” and crossover Ethernet cables. Such a virtual network is a possibly interesting and useful but completely closed system, isolated by the outside world.

The external socket device represents a “female Ethernet wall socket”, opening a breach in this apparent closure: when connected to an external socket other components can access the same (non-virtual) network to which the host belongs.

External sockets provide several useful opportunities:

• connecting virtual computers to the Internet
• easily installing additional software on virtual machines, for example using apt-get install on a debian distribution
• making virtual computers clients of services offered by the host or its network: some examples include DHCP, DNS, NFS, and NTP

It is also possible to use external sockets to connect several virtual networks, possibly running on different hosts.

The implementation depends on the bridging functionality in Linux.

Parameters
The user can only specify a name and an optional label.

3. NETWORK GRAPH

When any virtual device changes state or a Marionnet project is opened the network graph image is regenerated to reflect the current situation.

The configuration parameters settable by the user from the palette next to the graph image in the Hardware pane are divided into four categories.

5Of course the speed of the processor or processors would remain an artificial limit, but limiting the maximum bandwidth is easy with VDE.

6When this policy is enforced an “incorrect” cable, for example a crossover cable connecting a computer with a switch, simply does not transmit any frames.

7There would be very little point in stretching the start-up/shutdown/poweroff/pause/resume metaphor to also apply to cables, as the behavior associated to each of these actions would always exactly mimic either the connect or disconnect action. Simply providing the means for temporarily disconnecting a cable is intuitive and covers all the useful cases.

8There is a hidden communication channel from each virtual computer to the host system, used for making virtual machines exchange messages with the host X server (be it native, switches, Xnest). The implementation of this feature is discussed in [12]. Nonetheless this communication is not directly accessible to the user and definitely not “general-purpose”.

9This functionality is implemented with runtime calls to dot, from the free software package Graphviz ([1]), dot can generate an image by placing icons in the two-dimensional space in a completely automatic way, but it also supports many “hints” to tune the picture appearance. All the user settings described in this section are implemented with (often quite complex) dot commands.
**Figure 4:**
A typical example of simulation with a very simple network. $m_2$ is ping-ing a remote machine reachable through the *external socket*. $m_3$ has an *Xnest* interface; instead $m_1$ has a terminal interface, and its *xeyes* is shown by the host X server.

---

**Nodes**

Three settings are available:

1. a slider allows to tune the icons’ size
2. pressing the “dice” button randomly rearranges nodes
3. pressing the “slashed dice” button resets the nodes arrangement to the initial state

**Edges**

Four settings:

1. pressing the “right arrow” button regenerates the image so that its main spine is drawn horizontally
2. pressing the “down arrow” button redraws the image with a vertical main spine
3. a slider allows to set the *minimum edge length*
4. pressing the “circular arrow” button pops up a menu allowing to swap the ends of an edge in the network graph image

**Labels**

A slider allows to set the *distance* between a label and the icon representing the node it describes.

**Surface**

A slider allows to set the *size of the canvas* containing the whole image.

4. **EXECUTION AND CONTROL**

The Marionnet user interface provides two different styles of controlling simulation: the device-oriented *local style* and the network-oriented *global style*.

4.1 **Local vs. global control**

The device palette provides for *local control*: the user first chooses a *device type* (e.g. the router); then a popup menu appears displaying the possible *actions*. When the user chooses one (e.g. “power-off”), a sub-menu pops up displaying the list of *device name* of the selected type on which the selected action is currently possible. The action is performed as soon as the user clicks on a device name. All menus are *dynamically generated* according to the current simulation state: for example the “resume” action is only possible on currently “paused” devices; and of course the possibility of creating and destroying devices *even during the simulation* implies that the list of existing devices itself varies through time.

By contrast, the large buttons always visible on the bot-
tom pane permit a form of global control: pressing one of
them executes the same action on all the devices which are
currently in a state allowing it; the possible global actions are
startup everything, shutdown everything, and power-off
everything.

4.2 “Shutdown” vs. “power-off”

Within Marionnet the term “power-off” means brusquely
unplugging the “power cord” of a virtual device\textsuperscript{10}. By con-
trast the term “shutdown” refers to the action of “gracefully”
terminating the activity of a device before turning it off.
The difference between power-off and shutdown is rele-
vant for “stateful” virtual devices which contain some kind
of disk which supports asynchronous writes, like computers
and routers; just like for physical devices, a brusque power-
off can leave filesystems in an inconsistent state\textsuperscript{11}.

5. ADVANCED USAGE

Marionnet also provides several advanced functionalities,
the most important of which we now enumerate:

5.1 Interface pre-configuration

In the Interfaces pane (see Figure 5) the user can configure
the network interfaces of virtual computers and routers by
setting the following parameters:

- MAC address
- MTU
- IPv4 address
- IPv4 broadcast address
- IPv4 netmask

\textsuperscript{10}This is implemented by sending a \texttt{SIGKILL} signal to all the
processes simulating a virtual device, for most devices.
\textsuperscript{11}But damages are typically very limited with modern filesys-
tems: since UML is a complete port of Linux it includes sup-
port for journaled filesystem like \texttt{ext3} and \texttt{reiserfs}, which of
course can also be used in virtual computers and routers.

- IPv6 address\textsuperscript{12}

Although all these parameters can also be set after startup
by logging in the virtual computer or router and invoking
\texttt{ifconfig}, this interface provides a convenient shortcut. By
setting a global option IPv4 and IPv6 addresses can also be
automatically generated by the application upon the cre-
ation of a port.

5.2 Defects

The communication infrastructure can manifest several
types of “faulty” behavior:

- frame loss
- frame duplication
- flipped bits
- transmission delay\textsuperscript{13}

As shown in Figure 6 defects can be set up in the Anoma-
lies pane, with the granularity of the single electric line, i.e.
the direction (\texttt{in-to-out} or \texttt{out-to-in} for ports and \texttt{left-to-right}
or \texttt{right-to-left} for cables).

For each direction of each cable or port of each device the
user can individually set any defect, by entering a probabil-
ity or, in the case of delays, a time in millieconds.

Defects settings can be updated “hot”, i.e. while the net-
work is running: the behavior is immediately affected.

5.3 Filesystem history

For each virtual computer or router, a complete history
of the disk states is available (see Figure 7): each state is
saved just before startup.

A machine or router can be started up in the most recent
state (which is the default behavior), or in any previously
saved state.

\textsuperscript{12}IPv6 addresses also contain information about the netmask
and broadcast; there is no need for three distinct parameters
in the case of IPv6.
\textsuperscript{13}The delay follows a normal distribution in the current im-
plementation. The parameters “min” and “max”, settable by
the user, represent $\mu - 3\sigma$ and $\mu + 3\sigma$. 
saved state. This allows users to freely experiment with potentially “dangerous” filesystem modifications, as each change is reversible.

For each machine or router the filesystem history displays a tree structure keeping track of the “parent-child” derivation relation of states.

States can also be deleted or exported as variants, to be used for new machines or routers in the same or even in different projects: see subsection 2.1 and Figure 2 for some details about variants.

6. FILE FORMAT

The Marionnet project file format is a tar archive containing some OCaml marshalled objects and UML cow files.

The format has been very carefully designed to be back-and forward-compatible: newer versions of Marionnet can read project saved by older versions and vice-versa: when Marionnet finds some information which it doesn’t “understand”, the system simply ignores it. If instead some needed field is lacking then a default value is generated.

We hope this to become a conventional exchange format for people who desised to share projects and “prepackaged” networks.

7. A POSSIBLE USAGE EXAMPLE

As one example of a usage scenario for Marionnet being employed in the industry we outline the development of a peer-to-peer application-level protocol, using a simulated network.

7.1 Scenario

We assume the protocol to be based on an existing transport layer, so that its implementation does not need to reside in the kernel; a reasonable choice for the transport is, of course, UDP.

14Marionnet could also be used for testing kernel-level implementations based on Linux, but this would limit its application to running virtual machines using experimental UML kernels. The kernel-level development itself could also be performed within Marionnet, but without particular benefits: in particular (virtual) computer reboots would be still

7.2 Development

Setting up a development environment in which programmers can be productive and at ease may involve a considerable investment of time and energy; our proposal is explicitly aimed at not disrupting this environment. In particular the development itself, intended as editing and compiling, can continue to happen on the physical machines already employed by programmers.

We assume the adoption of a typical modern infrastructure for development, in particular the availability of a network file system like NFS or SMB.

A shared directory on the network file system will host the compiled, executable files, so that they are available to all virtual computers through an Ethernet socket device (see Subsection 2.8).

One virtual computer should be configured as a peer machine, installing on its virtual filesystem all the needed software, including external services and libraries; the packaging system of the GNU/Linux distribution can be used for this. This setup should then be made easily reusable by saving the machine’s filesystem state as a variant; in this way other identical peer virtual machines can be later created, at will.

7.2.1 Discovery

The first task in this project would probably consist in developing some strategy to discover new peer machines joining the network. This is easy to test by creating a virtual network with just three or four computers connected to a switch. Each of them will run the same executable from the shared directory.

7.2.2 Fault tolerance

Any reasonable peer-to-peer protocol should of course be fault-tolerant, and react in a correct way to the sudden disconnection of some machine. The pause and power-off actions can be employed on virtual computers in the same setup outlined above.

In order to stress the protocol’s resilience to the lack of reliability of the network, one or more cloud devices (see Subsection 2.7) can be added to the virtual network. The application can be tested in arbitrarily adverse conditions, by tuning the defect probabilities16 (see Subsection 5.2) at runtime, while the application is running.

If the application has also an on-disk state and its fault-tolerance needs to be tested, single virtual computers can be powered-off while running, just for this purpose. If some “interesting” inconsistent disk state is reached in this way the filesystem state can be saved as a variant (see Subsection 2.2) so that the problem can be studied and debugged in repeatable conditions.

7.2.3 Firewalls, NATs and multicast

If the peer-to-peer protocol needs to work across firewalls needed. Here we want to present, by constrast, a case of development with rapid turnaround.

15If not already available such a service can be installed on the Marionnet host computer, when the same machine is also used for development.

16Doing the same type of test with a loopback network device or on a physical network would be extremely hard, because the number of faults would be too small in practice, and however not directly modifiable.
and NAT subnetworks, the virtual network can be modified by adding routers (see Subsection 2.5), or different virtual computers configured as gateways: virtual computers can be equipped with the exact same firewall software which could run on physical networks.

A possible optimization of the protocol consists in supporting multicast to transmit the same message to multiple peers: such a feature, and its interaction with routing, is much easier to test in a virtual environment.

7.2.4 Interoperability among different versions

At some time in the application life time, the protocol will inevitably have to be modified. When interoperability among different versions is desired, tests can be prepared by setting up one virtual machine per released protocol version (not using the shared network directory in this case), and making variants.

If this is desired, machines created from such variants can be added to all future test cases, and made interact with the new versions of the protocol.

If some older version which the developer desires to use for testing is not readily available, it can even be fetched from the revision control system and compiled on the virtual machine itself: the only requirements are, of course, the availability of a network-accessible revision control system on the host network, and the compiling (not necessarily editing) environment on virtual computers.

8. CONCLUSIONS

We implemented a general purpose interactive simulation system for computer networks, allowing to accurately simulate complete networks, including the software running on computers.

Marionnet is currently being employed with success for teaching computer networks to students at Université Paris 13. We believe its use can be profitable extended to other cases of simulation and debugging or networks and applications.

Some rough edges and “wishlist features” remain, like UI internationalization. The problem of supporting wireless networks, despite being nontrivial, has been tackled with success by others using UML: see [9].

Marionnet is free software, released under the GNU GPL ([8]): anyone is free to share and modify it.

Marionnet is written using the elegant functional language OCaml ([11]), and the implementation itself is quite interesting; we are trying to build a strong “community” around it, and we welcome contributions to expand it in new directions.

The Marionnet web site is http://www.marionnet.org.

9. ACKNOWLEDGEMENTS

We wish to thank Jeff Dike for UML, Renzo Davoli for VDE, the author of OCaml and of course the free software community, for making this work possible. Marionnet is financed by Université Paris 13.

10. REFERENCES


