Status Report: Marionnet

How to Implement a Virtual Network Laboratory in Six Months and Be Happy

Jean-Vincent Loddo
Laboratoire d'Informatique de l'Université Paris Nord
France
loddo@lipn.univ-paris13.fr

Luca Saiu
Laboratoire d'Informatique de l'Université Paris Nord
France
saiu@lipn.univ-paris13.fr

Abstract

A virtual network laboratory —allowing to emulate a physical network of computers and network devices such as switches or routers *in software*— represents a valuable tool for students, and may also be useful to researchers and system administrators. A tool of this kind, particularly if it aims at being usable by inexperienced students, should offer the same opportunities of configuring and experimenting with components as a physical network, providing also an intuitive graphical user interface for *dynamically* manipulating the network topology and each individual virtual device.

Building such an inherently concurrent system is nontrivial, requiring the integration of many different components written in different languages and a complex control logic. Indeed some projects with similar purposes have been existing for years, and typically use scripting languages such as *Python* and *Bash*; by contrast our system, *Marionnet*[1], has been implemented using the functional language OCaml in just six man-months and yet providing several important features still missing in more mature projects.

We seize the occasion of describing Marionnet to discuss the relevance of the functional style and of advanced type systems for dramatically cutting development time.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.1.3 [*Concurrent Programming*]; D.1.5 [*Object-oriented Programming*]; D.3.3 [*Language Constructs and Features*]: Polymorphism. Classes and objects. Inheritance; C.2.m [*Miscellaneous*]; C.2.m [*Miscellaneous*]; I.6.3 [*Applications*]; I.6.7 [*Simulation Support Systems*]; K.3.2 [*Computer and Information Science Education*]: Computer science education; K.3.1 [*Computer Uses in Education*]: Collaborative learning.

General Terms Design, Languages, Experimentation

Keywords OCaml, static typing, emulation, virtual machine, GUI, User Mode Linux

1. Motivations

Enabling students to practice on network configuration and distributed application development using physical components is cumbersome and expensive, particularly in a crowded classroom environment where the availability of devices such as computers, switches or routers is limited.
It is also completely unrealistic to expect that students are able to do exercises at home on their own, when they typically only have access to a single Internet-connected computer.
The possibility of a direct "hands-on" experience with network protocols is also impaired by the same difficulties, and may end up being undeservedly neglected in a traditional teaching setting.

A straightforward solution to solve this pedagogical problem consists in emulating[2] a whole computer network on a single machine.

Moreover, the scope of such an application may extend well beyond our initial didactic motivation, and a system of this kind can prove itself to be valuable also as a testing tool for network administrators and computer scientists interested in security: many network attacks can be easily and safely emulated, without any need for hardware setup.

2. Introduction

As a first approximation, nesting a whole network into a single computer essentially amounts to:

- *emulating single machines* using one among the several already existing technical solutions[3]; without any pretense of exhaustivity we cannot but cite some free software projects as Bochs ([26]), QEmu ([6]), UML ([14, 15, 16]) and Xen ([4]), and the proprietary product VMWare ([46]).
- *emulating network devices* such as cables, hubs, switches and IP routers; this second problem can also be solved in a variety of ways, all of them consisting in some functionality —of widely varying complexity— built on inter-process communication and operating system features such as `tun/tap` interfaces ([25]).

The differences among all the possible approaches above are not particularly significant for our purposes, although they may influence the performance and applicability of a certain solution.

[1] *Marionnet* is supported as an *e-learning* project by Université Paris Nord.

ML'07, October 5, 2007, Freiburg, Germany.

[2] Here we take the term "emulation" in a very broad sense, including solutions as different as *full hardware simulation*, *paravirtualization*, *user-mode kernels*, and so on.

[3] Such solutions greatly differ in their scope and implementation style: assembly instruction *interpreter* vs. *incremental compiler*, *hardware emulation* vs. *operating system virtualization*. The original intents of them were also varying, including the concurrent use of different operating systems on the same machine, and security applications such as *honeypotting* and *sandboxing*.

We choose UML as a platform because of its maturity, documentation, relatively simple installability and good performance.

UML allows one to manage several kernels as standard userspace processes, and in its turn this enables us to emulate GNU/Linux machines —very suitable to networks— on commonly available x86[4] hardware.

This choice is by no means the "only" possible one: other technologies like Xen would also have been reasonable alternatives.

2.1 An high level architecture: network emulation layers

As shown in Figure 1, the network emulation problem can be abstractly modeled into four layers, all of them nontrivial. Each layer depends on the layers below it for its implementation, and as usual the level of abstraction grows upward.

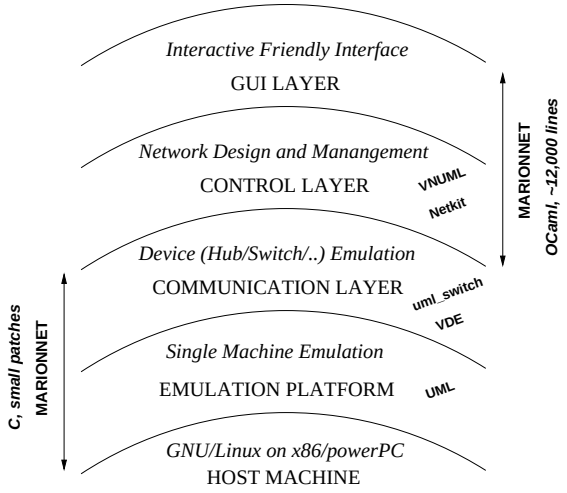

Figure 1. Emulation layers for a virtual network

Emulation platform. At the bottom level an *emulation platform* allows one to create several independent virtual computers running on a single —possibly not even networked— *host* computer. Virtual machines should realistically replicate real machines' behavior, allowing to execute user software with no modification, and to normally read and write a local *virtual filesystem*.

Communication layer. Virtual machines must be able to communicate with one another in some way, employing real network protocols; in practice supporting at least Ethernet is imperative, due to its ubiquity.
Virtual *network devices* such as switches, hubs and IP routers should also be available. Such virtual network devices should be running as host processes, but replicating in the closest possible way the behavior of their physical counterparts. We call this intermediate level the *communication layer*.

This layer might also provide some mean to observe the network traffic: it seems natural to implement such functionality at this level, as the emulation of communication among machines and devices already requires to work with protocols at a low level.

Control layer. A *control layer* allows one to define a virtual network and run it, reasoning with emulated components at a very high level. As a bare minimum a control layer should provide means to *statically* define, then startup and shutdown a whole virtual network.
A more flexible layer may allow one to *dynamically* define network devices, turning on and off *individual* components, and connect or disconnect virtual cables, all of this *while the network is running*.

GUI layer. At the top level the user could be exposed to a user interface, providing an intuitive view of the network and allowing to easily interact with the control layer.
It may be worthy to emphasize that, on one hand, our main target users are inexperienced first- or second-year students, unaccustomed to complex interfaces and still without a deep understanding of how networks work; on the other hand, a full network configuration is quite complex a state, and requires some sophistication in the interface to be presented in a readily understandable way.
The need to balance among these two conflicting needs makes also the interface design nontrivial; and a vast number of features (see section 3) might make the use of a GUI nearly a necessity.

Some applications explicitly aimed at didactics in this field already exist, but to our knowledge none of them implements dynamic control; two project support GUIs, but their sophistication in this aspect is limited by their laying upon a *static* control layer. See section 6 for more information and a detailed comparison with related projects.

2.2 Contributions

Our application, *Marionnet*, is an *OCaml*[5] ([28, 9]) implementation of the topmost two levels, a *fully dynamic control layer* and a *GUI layer*.

The emulation platform consists in *UML*, by Jeff Dike et al., and the communication layer is *VDE*, by Renzo Davoli et al. ([13]). UML and VDE —both written in C— have also been patched for Marionnet (see subsubsections 4.9.1 and 4.9.2), but the very limited scope of such changes makes unreasonable to talk about a mixed OCaml / C implementation: the C components have been re-used essentially as they were.

Our GTK+ ([42]) user interface (see Figure 2) is very simple to use: in our tests "on the field" most students have started to be productive with it in a matter of minutes.

The static part of the interface has been mostly built with Glade ([45]), and the dynamic part with LablGTK ([22]), the OCaml GTK+ binding.

Our OCaml implementation amounts to just 12,000 lines of manually written code, which testifies in favor of the language conciseness and suitability for rapid development: the whole application was built by the authors in just about *six* man-months.

Even if not yet polished the application has proved to be solid, and a preliminary version has already been used by one author in occasion of the *Practice of Network Protocols* exam at IUT[6] of Université Paris Nord in June 2007.

Marionnet is free software built on free software, distributed under the GNU General Public License ([19]). We have taken advantage of the possibility of modifying the source code of some of our components (the Linux kernel and VDE), and in the true spirit of free software we hope that our work may in its turn form a basis for others to build upon.

3. Objectives

We are now going to briefly list application requirements as perceived user expectations, teacher needs and implementation constraints.

[4] We did not test the application on PowerPC yet.

[5] The reasons for our choice of OCaml are discussed in section 4.1.

[6] "Institut Universitaire Technologique".

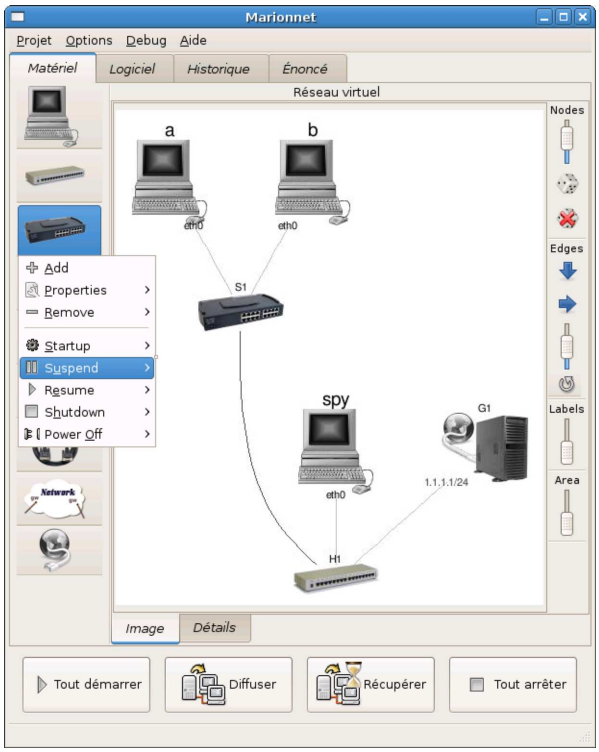

Figure 2. The main window of *Marionnet* showing a simple network with three computers, a switch, a hub and an Internet gateway.

3.1 Emulated network components

It was decided since the beginning that the network devices to be emulated would have been user-configurable at a very fine level of detail.

Computers: for each computer the user should be able to specify the *amount of RAM*, the *number of Ethernet cards and serial ports* (serial ports are also used as network devices), the particular *kernel* and *GNU/Linux distribution* to run, and the way to run *X* ([40]) clients on the emulated machine: either connecting to the host X server, or to a different *Xnest*[7] per machine, or without any support for graphics.

Hubs and switches: for each hub or switch the user can specify the number of ports, and —for switches— whether it should support the *STP* protocol ([35]).

Routers: emulated routers should replicate in a close way the behavior of physical Cisco routers; their configuration is complex and mostly happens at runtime, except for the specification of the number of ports.

Cables: an *Ethernet* cable can be either *straight-thru* or *crossover*. *Serial* cables have also to be supported.

Clouds: a *cloud* represents a slow and noisy IP connection (or a whole network the user has no control upon), with two endpoints: packets enter into one end and (possibly) exit from the other end, generally out of order, with a random *delay* and their *ttl* decreased by a random amount. The user can specify several parameters of the involved probability density functions.

[7] In this case Xnest processes run on virtual machines, but *seen as X clients* they however require a connection to the *host* X server; hence the problems to solve are essentially the same.

Gateways: an emulated network gateway connects the virtual network and the *host* network, and routes the traffic between them, making possible even Internet access for emulated computers. The only specified parameter for gateways is their *IP address*.

3.2 Port and cable defects

Just as physical hardware can fail (and discovering and working around such faults can provide an interesting, if sometimes unforeseen, learning opportunity), it should also be possible to define *defects* in any port or cable of the emulated network, with the granularity of each *direction*: *left-to-right* / *right-to-left* for cables, or *in-to-out* / *out-to-in* for ports. The supported defects are *delay*, percentage of *lost packets*, percentage of *flipped bits* and *bandwidth upper limit*.
As for the *cloud* device, the user should be able to set some parameters of the involved probability density functions.

3.3 Dynamic network reconfiguration

In order to enable users to perform the same kind of experimentation possible with physical networks it was decided from the beginning to allow one to tune *single devices* independently from the rest of the network. The user should be able to change the network topology by adding or removing components *while other components are running*.

Each cable can be temporarily disconnected so that the user can observe how the network works in its absence, and then reconnected; this is particularly useful to experiment with routing protocols.
As a useful "extension" of physical networks behavior it would also be desirable to generalize temporary disconnection to other devices, enabling users to *suspend* them and then *resume* it.

Stateful devices such as machines can be turned off in a clean way or by simply interrupting power[8].

"Hot" reconfiguration was deemed essential even if it inevitably complicates the implementation, raising the level of concurrency.

3.4 Filesystem history

A further extension to what would be possible with a real network consists in —at least *logically*— saving the complete filesystem image at shutdown time for machines and routers. This allows the user to freely experiment with configuration, with the possibility of returning at any moment to a known "working" state.

This feature may also be useful to the teacher, who may wish to inspect a student "configuration history".

3.5 GUI-related functionalities

Of course the interface should offer the usual functionalities of a GUI program such as the possibility of opening and saving projects; a project includes a network graph, the filesystem states forest and, optionally, textual problem statements for students.

This feature is important because a Marionnet project file is also thought as an *interchange format*, particularly to enable teachers to cooperate exchanging exercises[9].

Network graph image. An up-to-date graphical representation of the whole network graph should be available at any time. Such representation should be automatically generated in an understandable form, to spare users the irrelevant burden of placing nodes in a

[8] And of course, as it happens with physical machines this may leave filesystems in a messy state.

[9] This practice happens to be surprisingly less frequent and more cumbersome than it should, because of hardware and software incompatibilities among the machines of different classes. We hope Marionnet might contribute to alleviate such problems.

two-dimensional space; however a toolbar should also be provided to fine-tune some representation parameters such as the *whole image size*, the *distance from each node to its label*, and the *icons size*[10]. Different kind of cables (straight-thru vs. crossover vs. serial) should be drawn as arcs of different colors; detached cables should also be portrayed as such.

Virtual computer interface. For each computer a console should be visible for users to log in and enter commands. Machines should also be able to run graphical X applications, and particularly elaborate network applications such as the graphical sniffers *Ethereal* and *Wireshark*.

Virtual device interface. In the same spirit, other network devices should have a simulated *control panel* showing a grid of blinking LED lights representing port activity, in close resemblance to physical devices. This functionality provides a simple and intuitive way to observe network traffic. See Figure 3.

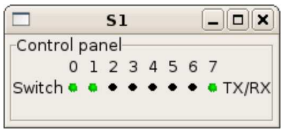

Figure 3. A virtual eight-port switch *LED grid*, with cables connected to the ports 0, 1 and 7.

Filesystem history interface. An interface based on a *treeview* widget should allow one to easily navigate in the filesystem history displayed as a forest, showing which state is derived from which other state for each machine and router, allowing to *delete uninteresting states*, to *add textual comments* and to *boot a device from any saved state*, like in Figure 4.

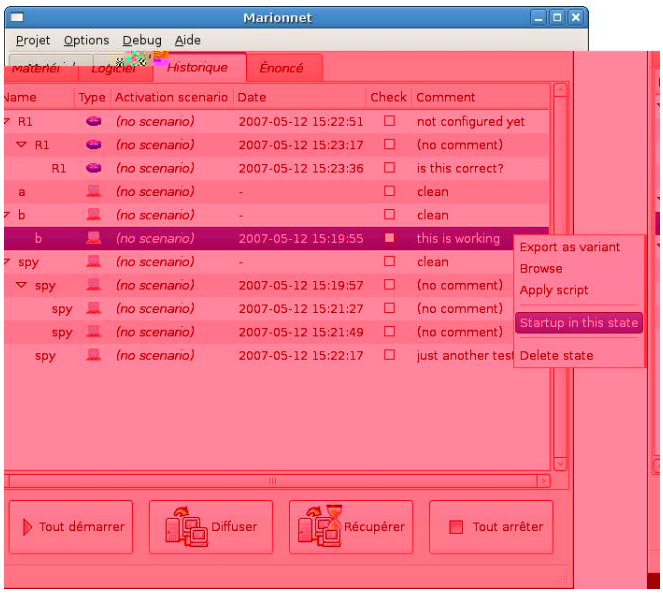

Figure 4. Filesystem history interface

[10] Figure 2 shows a network graph image automatically generated without any fine tuning.

3.6 Classroom functionalities

Some more advanced functionalities for working in a classroom are desirable: for example a teacher should be able to broadcast a project to all the students' computers (*diffusion*), and students in their turn should be able to send modified projects back to the teacher (*retrieving*).

When in *exam mode* the application should also automatically run analysis scripts on all the students' virtual machines at shutdown time; the output of such scripts should be easily available to the teacher.

3.7 Internationalization

The user interface should be fully internationalized, and support some universal character set like *UTF-8* as the *external* representation for strings. Our short-term plans are mainly centered around French localization, but there is no reason why Marionnet should not be usable also in different locales.

3.8 Development time constraints

The bulk of the development effort had to be spent in implementing the two topmost layers of Figure 1 ***in the shortest possible time***. This challenging requirement was motivated by the fact that the authors could not be full-time developers or maintainers, and the application had to be at least usable by the end of the second semester of 2007, in order to be employed for exams at the IUT of Université Paris Nord.

These constraints essentially dictated the need of reusing existing software whenever possible, and using a high-productivity language for all the new code.

4. The solution

4.1 Language choice

We made the choice of using a functional language fairly early, because of our long positive experiences, in one case for teaching and in the other in large implementation projects ([39]).

OCaml was preferred over other functional languages such as Haskell ([36]) because of its natural support for mixed programming styles ([32]). Because of the importance of the imperative side of our project and in the light of our past experiences, we have some doubts about how Haskell *monads* would "match" our problem. Despite their mathematical beauty, we feel that monads leave to be desired as a practical programming construct: introducing some stateful computation or I/O within some code which was initially written as functional invariably implies the need for a cascade of changes in the types of *surrounding* expressions. Said in other words, introducing a monadic construct as a *local* change tends to have *global* effects. While side effects do indeed introduce problems, they appear to function better as a *mean of abstraction* in the sense of [1], section 1.1. This theme is touched in [39].

4.2 Implementation guidelines

In spite of the prevalence of the functional style (subsection 4.7 and subsubsection 4.8.4 show interesting examples) our use of OCaml also takes advantage of the *object-oriented* style, particularly in the control layer (see subsection 4.4), where "objects" in the OO sense tend to represent the *physical* objects to be emulated.

Concurrency makes the implementation substantially more complex, but it is unavoidable to support our dynamic control layer. Anyway our use of threads is controlled and restricted to just a few cases (see subsection 4.4, subsubsection 4.8.3 and especially the code in subsection 5.3), in order to keep the complexity manageable and the development time short.

4.3 Network structure: user-level vs. emulation-level

Due to the strict time constraints forcing us to reuse an existing communication layer we decided fairly soon to use *VDE*, to our knowledge the most powerful way to interconnect UML virtual machines[11].

VDE allows one to create *virtual switches and hubs* connected to the Ethernet interfaces of UML machines[12] and *virtual wires*. All virtual switch and hub ports are implemented via Unix sockets ([41]) or `tun/tap` interfaces [25]. Wires can be destroyed and recreated at runtime without affecting virtual machines. Such functionality may apparently look like a perfect match for our problem.

Unfortunately, however, a couple of implementation choices in VDE and UML prevent a direct application of VDE devices to emulate the devices specified by the user in Marionnet. The first problem is that **when a network interface is defined for an UML machine, a *switch or hub* where to connect it can be specified, but *not a wire***: in other words, some virtual cables are not directly represented as cables in VDE, and are left implicit. This risks to force us to introduce gratuitous asymmetries and particular cases in our code but, more importantly, also creates problems with dynamicity: destroying the virtual hub or switch to which a virtual Ethernet interface is connected makes the interface unusable; by contrast we need "*stable endpoints*" to which we can dynamically connect and disconnect cables which on their turn may be connected to other devices.

VDE also implements defects as we need them[13] but *only in cables*, whereas we require the same functionality also for ports.

To overcome such problems we devised a two-level emulation approach where ***each* device in the *user-level* network is represented, in general, by *several* UML and VDE processes, making up the *emulation-level* network**.

Such dichotomy is of course completely invisible to the user.

Our mapping from *user-level* into *emulation level* may be most easily understood with an example:
a virtual computer with n Ethernet cards is represented by:

- *one* UML process connected to
- n two-port VDE hub processes ("*hublets*" in our jargon and code), in their turn connected to
- n VDE cable processes incorporating port defects, connected to
- n hublet processes, representing user-level Ethernet ports to which virtual wires can be connected at any time.

Figure 5 shows, among other things, how a machine with two Ethernet interface at user-level is represented at emulation level.

Hublets are used as the *stable endpoints* we need for Ethernet connections: they are only destroyed when a virtual device is destroyed or the number of its ports is changed. Virtual cables can be connected or disconnected from hublets at any moment.

Such a solution allows for great flexibility, and also makes the implementation modular and consistent: a virtual switch with four ports, for example, is always represented by the same number and type of processes, wherever it occurs in the user-level network. This convenience comes *at a negligible cost* in runtime performance: traversing one or two switches more has no observable delay, and average local `ping` time between two virtual machines has consistently been measured as well below one millisecond on all our test machines[14].

We adopt the same strategy also for emulating serial ports and cables: in this case we simply map virtual machine serial ports to host `ptys` —which is supported by UML— so that we can **use `ptys` as stable endpoints for serial connections**. We simply employ `cat` processes with input/output redirection to emulate serial cables, dynamically spawning and killing them as needed.

Figure 5 shows how each element of a very simple network is represented at *user-level* and *emulation level*.

Two distinct levels of emulation exist, and this is clearly reflected by the structure of our OCaml code. On one hand a hierarchy of classes represents *processes*, on the other a second distinct hierarchy implements *devices*, using processes as building blocks. Both processes and devices internally represent the current emulation state as a DFA state (including for example *on* and *off* for a device, and *running* and *suspended* for a process: see Figure 6 and the code in subsection 5.1). In both cases methods are provided for following existing DFA transitions, interacting with the external UML, VDE and `cat` processes and updating the emulation state.

4.3.1 Translation of virtual devices into processes

What follows is a quick description of the mapping from the user-level network to the emulation-level network for each kind of device.

Virtual computer with* n *Ethernet cards and* m *serial ports: A UML process u is connected to *n* hublets h_{i1}, and each h_{i1} is connected to another hublet h_{i2} via a cable c_i incorporating the defect of the Ethernet port `eth`i. Each user-level `eth`i —seen as a connection endpoint— is represented by the hublet h_{i2}.

Each serial port j of u is connected to a dynamically allocated `pty`, which represents `tty`j as a connection endpoint.

The special Ethernet interface `eth42`[15] is also always emulated to allow graphical applications to communicate with the host X server. `eth42` is *ghostified*, and connected to a host `tap` interface rather than to a hublet: see subsection 4.9.1 for information about what this means and implies.

Virtual Ethernet cable connecting any pair of Ethernet ports: The two endpoints are simply connected by a VDE cable process. No distinction is needed between *straight-thru* and *crossover* ca-

[11] The simple `uml_switch` by Jeff Dike et al. ([16]) also allows a set of UML machines to communicate, but it does not allow one to link several switches to one another, and especially does not support the *dynamic* behavior we need. `uml_switch2` by Felix Müri ([31]) improves Jeff Dike's `uml_switch` at least by allowing to cascade devices, but does not allow dynamic control. By contrast VDE allows one to cascade virtual switches and hubs, and the possibility of creating and destroying *virtual wires* at runtime is another of its main features ([13]).

[12] VDE can also be used as a communication layer for QEmu and host machines, but this is not important for our purpose.

[13] We modified the "defect" functionality of VDE only in a trivial way: see subsection 4.9.2.

[14] Our smallest test machine was a Pentium III 800MHz with 512Mb RAM. Marionnet itself is quite lightweight for a graphical application; most processor time, and especially memory, is consumed by UML processes. We found that a minimum of 50Mb per UML instance is needed for comfortably running graphical applications on virtual computers (16Mb are enough for many typical non-graphical applications). Hublets and cables have not a significant overhead because they are idle for most of the time, and only slightly increase the emulated network latency when communicating, due to the added context switches and to blink commands (see subsubsection 4.8.3).

[15] The number 42 is "The Answer to the Ultimate Question of Life, the Universe, and Everything" in [2].

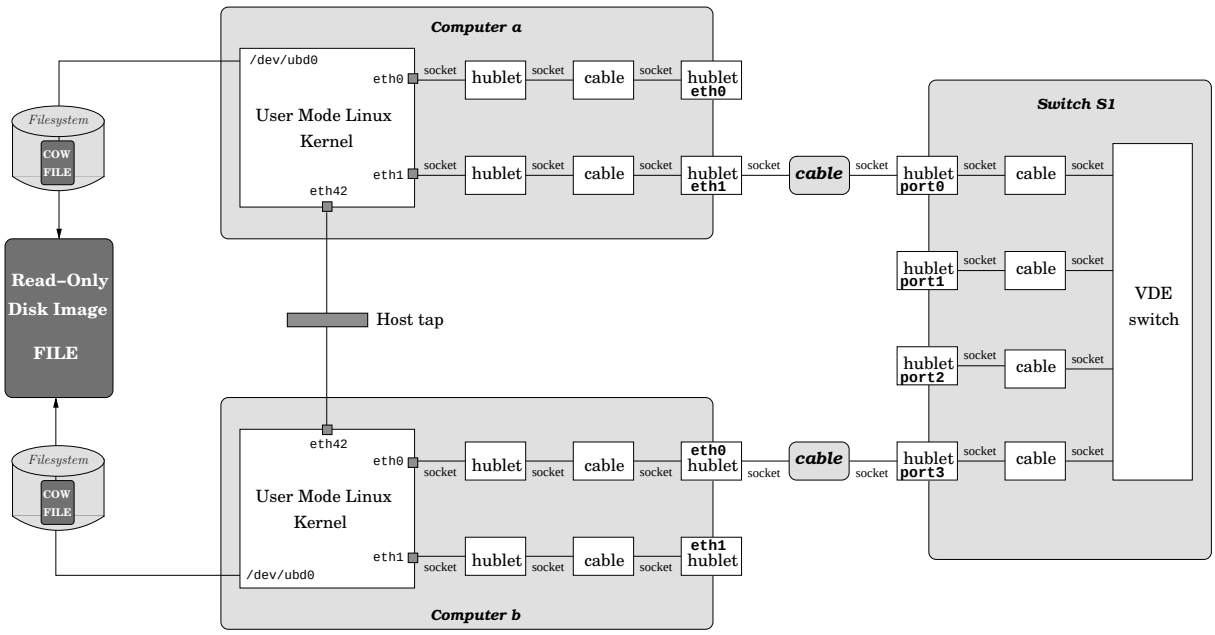

Figure 5. A sample network seen both at *user-level* and at *emulation-level*. The virtual computer *a* is connected to the virtual computer *b* via the four-port virtual switch *S1*. User-level network devices are represented as gray boxes with rounded angles, while emulation-level processes running on the host are shown white; the two user-level virtual cables are particular in being the only cases where the two views exactly match, a user-level device being mapped into an emulation-level process. In this case *a* and *b* are running the same distribution and hence are using the same filesystem image, but of course have different `cow` files. Both virtual computers have an `eth42` interface connected to a host `tap`, for communicating with the host X server.

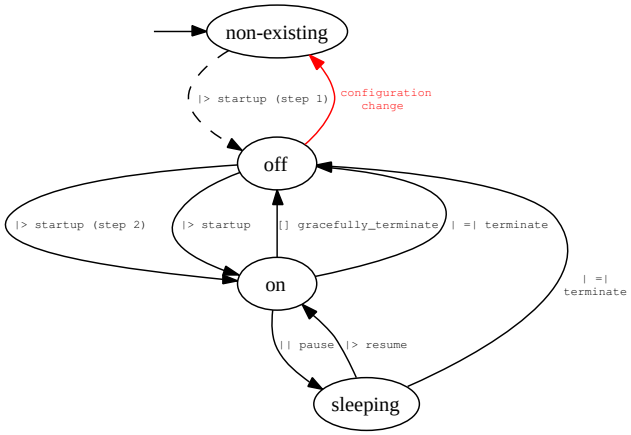

Figure 6. Virtual device DFA. The state *non-existing* is needed for all the cases where a device has been defined but not started yet (hence has no *hublets* or `ptys` to which other processes can connect), or its configuration has been changed (which may alter the number of *hublets* and `ptys`). Also note that the *startup* operation is implemented in two different ways, as a two-step transition from *non-existing* to *on*, and as a single-step transition from *off* to *on*; this difference is not exposed to the user.

bles since the GUI does not allow one to connect cables of the wrong type[16].

[16] However the possibility of connecting cables of the "wrong" type (thus obtaining a non-working connection) could have some educational value, and it is being considered for addition as an optional feature.

Virtual serial cable connecting two machines: A simple `cat` process has its standard input and standard output redirected to the endpoints' `ptys`.

Virtual hub or switch with* n *ports: A main VDE hub or switch m is connected to n cables c_i, each incorporating the defects of `port`i. Each cable c_i is connected to a hublet h_i, representing `port`i as an endpoint.

Virtual router with* n *ports: A router with n ports is represented exactly as a machine with n Ethernet ports and no serial ports. Routing protocols are implemented "in software" on the virtual machine, using the *Quagga* service ([8, 23]).

Cloud: Two hublets h_1 and h_2 are connected by a cable c. All "defects" like lost packets or delay are implemented in c, while the random `ttl` decrease[17] is implemented (by convention) in h_1. The free endpoints of h_1 and h_2 represent the two user-level cloud endpoints.

Gateway: A hublet is connected to a host `tap` on one side, and has the other side available for connecting cables, as the only gateway endpoint in the user-level network.

4.4 Control layer

The control layer is heavily object-oriented. A class hierarchy implements each device as a class, hiding the details of managing the the individual processes involved in the emulation.

The startup order of virtual devices is constrained by the need for hublets to be running *before* any of the VDE cables and UML processes directly connected to them.

[17] Such a functionality was not originally in VDE, but has been easy to add by patching its C source.

In the same way, a cable process can be spawned only when both its endpoints are already running.
Apart from these two constraints, all process spawns can proceed in parallel[18].

For usability's sake it is very important that the GUI does not "freeze" when a relatively long operation takes place, such as starting up a set of devices together. In order to achieve this the whole business of process management is delegated to a *control thread*, with the purpose of asynchronously accepting *tasks* to be enqueued, while executing them in a FIFO fashion. Each request may also internally involve concurrency.

The control thread is also responsible for reacting to *unexpected termination* of emulation-level processes, which may happen because of several conditions such as insufficient memory or explicit termination of a process by the user. Being able at least to terminate related processes in such cases and to keep the internal state consistent improves the application fault-tolerance.
Unexpected termination monitoring is performed by handling `SIGCHLD` signals from spawned processes.

To avoid implementing a command interpreter and ease inter-thread communication we devised a generic message-passing feature relying on OCaml's higher-order functions. The core of its implementation is shown in subsection 5.3.

4.4.1 Cable reference counter

A feature quite useful in practice is the possibility of temporarily *disconnecting* and then *reconnecting* a single virtual cable, without destroying it[19].
This is nontrivial to implement because of the complexity added by the need for a cable process to have both its endpoints (hublets or `ptys`) alive to be started: for example a cable whose endpoints become alive might be currently in disconnected state (hence the cable process should not be spawn), or the configuration of a currently running cable endpoint could be modified, leaving the cable with only one alive endpoint (and hence the cable process should be terminated).

We found a very compact and elegant solution to this problem involving a *reference counter*: each cable object has a mutable integer field `reference_counter`, always in the range $[0, 3]$. `reference_counter` is initialized to 1, and then:

- decremented at each *disconnection*
- incremented at each *reconnection*
- incremented at each *endpoint startup*
- decremented at each *endpoint shutdown*

A cable process has be spawn only when the `reference_counter` of its cable object rises from 2 to 3, and must be terminated as soon as `reference_counter` drops from 3 to 2.

4.5 UML kernels

When defining a virtual computer the user can choose from *several UML kernels* compiled with different configurations[20], particularly with regard to network parameters. As UML kernels are seen by the host as normal processes this poses no particular problems.

When we need to supply some parameters from the OCaml side to a UML instance we simply bind some variables on the kernel command line, and then retrieve them in the emulated computer from the Linux virtual file `/proc/cmdline`.

For example to implement the *exam mode*, we simply invoke UML with the parameter `exam=1`; the GNU/Linux distributions we provide are modified to check at shutdown time whether `exam` is bound in `/proc/cmdline` and, it that is the case, to run an analysis script and saving the machine configuration and other relevant results.

4.6 Virtual filesystems

The user can also choose among *several GNU/Linux distributions* installed on the host disk as filesystem images. The same distribution can be used by different virtual computers at the same time, and each machine must be able to *write* to its virtual filesystem, without interfering with the others. Making copies would be very impractical because of the typically large size of filesystem image, in the order of hundreds of megabytes or even several gigabytes.
Fortunately UML allows one to solve this problem in a simple way, using as a virtual filesystem a *pair* of files:

- a *read only* filesystem image
- a *sparse* file containing only changes relative to the initial image. In UML jargon such a "patch" is called —for understandable reasons— *copy-on-write file*, or *cow*.

Filesystem images can thus be shared without any concern for concurrency, and each machine needs only its *cow* file for running, as shown in the example of Figure 5.

Typical *cow* files take **just few megabytes** on disk but require *sparse file support* on the host filesystem. Such support is in practice always present on GNU/Linux, but it may be lacking on other systems to which the user might want to copy some files. For this reason we work with *cow*s only in temporary directories of the host filesystem, and always save Marionnet project files as compressed *GNU tar* archives. *tar*s may contain sparse files without wasting space and without being sparse themselves, hence they are safe to copy to any filesystem.

To get an idea of a "reasonable" projects file size, the project files delivered by students after their three hours exam mentioned in subsection 2.2 involved three machines (hence at least three *cow* files) and took on average 4Mb each.

4.7 Network graph representation

The network graph is implemented in object-oriented style in a quite straightforward manner, using *lists* for holding together elements of the same type.

Methods for looking up and updating elements by *id*, *name* and *type* are implemented using higher-order functions on lists, typically of linear complexity. This is perfectly acceptable in our case: the small size of the networks which can be practically managed with the GUI makes performance concerns irrelevant.

4.8 Graphical User Interface

GTK+ ([42], [22]) and Glade ([45]) allow one to build aesthetically pleasant interfaces with a "native" feel.

In this spirit we paid attention to respect the usual conventions of GUI applications that users typically expect, such as the presence of the usual *File* menu, status bars and toolbars.

In order not to make the user interface heavy we used *notebook* widgets, which tend to save space on the screen and make some interface elements visible only when requested. It is important that

[18] Such cases of process-level parallelism, very frequent in Marionnet, allow one to easily exploit SMP systems. OCaml, by contrast, can not exploit *thread-level* concurrency for parallel execution on SMPs, due to the current garbage collection design. We hope this limitation will be lifted in a future release.

[19] In order to mirror what happens with physical cables, virtual cables can not be "turned on" or "turned off", and they are "connected" by default.

[20] This simply involves kernels compiled with different `.config` configuration files.

Marionnet windows do not fill the whole screen, as often several other windows are needed, like virtual *computer terminals*, virtual *device LED grids*, plus the windows for all *graphical applications* running on virtual computers.

Figure 7 shows a not particularly complicated scenario of this kind.

Like any event-driven GUI, our interface makes heavy use of callback functions[21]. Being able to use higher order and in particular *partial function application* proved to be a huge advantage for writing callbacks, allowing to specify some parameters *at event connection time* rather than at function definition time.

An example from our filesystem history interface shows this:

```
(* A callback definition: *)
let on_add_row treeview selection file_name () =
  (* body *);;

(* ... From the function creating a popup menu,
   in an environment where treeview, add_row_menuitam
   and selection are visible:  *)
add_row_menuitam#connect#activate
  ~callback:(on_add_row
               treeview
               selection
               "file.text");
```

LablGTK ([22]) —the OCaml binding for GTK+— requires that callbacks have type `unit -> unit` and the partially applied function we connect to the *activate* event has indeed that type, but while `connect`'ing it we are also able to supply *any other parameter* needed in the callback body, and all of this with static type checking.

Such flexibility relies on the language support for closures, hence is simply unthinkable in most imperative languages including GTK+'s "native" C.

As one of the very few open-ended "experiments" we conceded ourselves, we implemented a polymorphic *environment* datatype representing the outcome of all user interactions via dialogs, seen as a set of $< key, value >$ pairs. Using environments allowed for some more modularity in interface code[22].

4.8.1 Network graph image

The graphical representation of the whole network shown in the notebook *Hardware|Image* is automatically built by *Graphviz* ([3]) from a *DOT* specification regenerated by the OCaml code at every network modification. The user is also free to set several parameters such as edge length and icon size, whose effect is forcing and immediate regeneration of the image with the appropriate DOT options.

Reusing the sophisticated functionalities of Graphviz allowed us to save considerable development time, with a negligible performance impact. Algebraic data types and higher-order make OCaml extremely well suited for symbolic manipulation of which this *compilation* is an instance. The availability of such features has made this part of the implementation particularly simple.

4.8.2 Computer terminals

UML easily allows one to use an `xterm` or `gnome-terminal` as the virtual machine console.

Practically the only "customization" we needed consisted in displaying the virtual machine name on the window title bar, to enable users to recognize which console belongs to which machine. This was achieved "in software" by making the GNU/Linux system running on virtual machines print a string with *XTerm terminal control sequences* at startup ([11]).

4.8.3 Device LED grids

For each *running* virtual hub, switch or router a GTK+ window is shown displaying the emulated device LED grid[23], as it can be seen in Figures 3 and 7. This allows one to easily inspect the *connection state* and traffic for each single port, mirroring what could be seen with a physical device.

LED blinking is *exact*, i.e. each blink reflects the transmission or reception of *one* Ethernet frame.

This is implemented with a small patch to the VDE virtual cable `wirefilter` (see subsection 4.9.2) and, at the OCaml side, with a thread waiting for 'blink' commands from `wirefilter` processes. Each command simply contains a LED grid identifier and a port index, encoded in a text string to ease portability.

Inter-process communication is implemented with *datagram sockets* in the `PF_LOCAL` namespace.

LED grids are implemented with a relatively complex combination of GTK+ widgets; in particular each light, which can be *on* or *off*, consists of a *notebook* widget (see [42]) with two hidden tabs, each of which contains a *pixmap*. The same pixmaps are shared by all lights.

GTK+ timers are used to automatically toggle the light state back to the *on* state (i.e. "connected but not communicating") after a fixed time interval for each blink (currently 80 milliseconds), making the interface completely asynchronous.

Despite the occasionally high bandwidth of blink commands this functionality has no noticeable impact on responsiveness, probably due to the efficiency of the implementations of GTK+ and Unix sockets. Garbage collection pauses are hardly perceivable.

4.8.4 Filesystem history and defects interfaces

Despite their very different uses, the filesystem history interface shown in Figure 4 and the defects interface (a *notebook* page for editing the defects of all ports and cables) share most of their logic, and at the implementation level they inherit from the same base class, heavily relying on the GTK+ *treeview* widget ([34]).

The functionality of both interfaces consists in displaying and allowing modifications to forest data structures: in one case the tree of filesystem states for each machine or router, in the other one the set of defects for each direction of each port or cable, organized as a set of trees for visual simplicity.

The forest data structure is implemented as a *polymorphic algebraic data type*, as shown in subsection 5.2.

Albeit slightly complicated by to the nature of *treeview*s this kind of implementation is interesting because it relies more on parametric polymorphism than on inheritance for code sharing.

Because performance is not a particular concern for these interfaces due to the very small size of managed data —in the order of few hundreds of nodes for typical cases— we can afford to do translations from forest data structures to GTK+ widgets and vice versa at every structure modification.

[21] Here we do not take into account the subtle distinction between *signals* and *events* made by GTK+. What we say here applies to both signal *and* event handlers

[22] And also made GUI implementation a bit more interesting.

[23] Despite the apparent and indeed partly intended "eye candy" nature of such an interface, experience on the field with students has shown it to be very valuable for debugging virtual network configurations.
LED grids have also been useful for debugging the application itself.

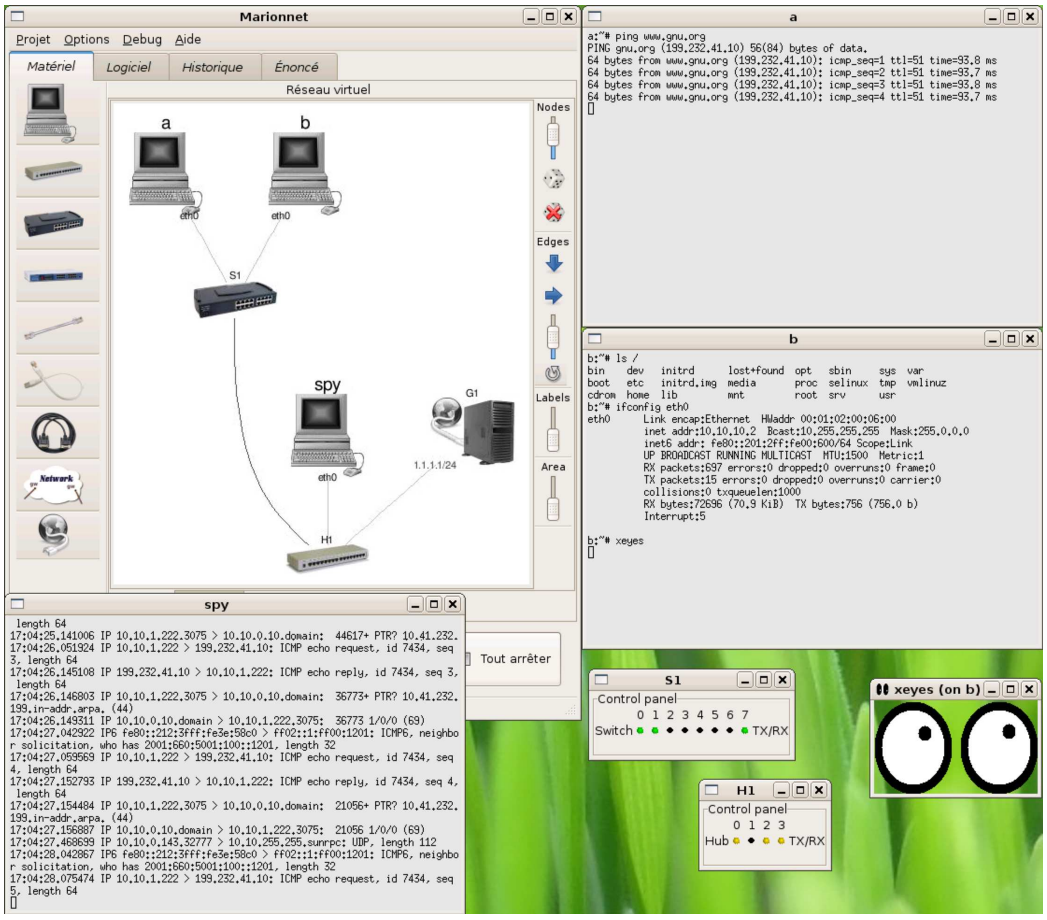

Figure 7. A very typical Marionnet session showing virtual machine terminals and device LED grids. The computer *a* is `ping`ing a remote machine reachable via the gateway *G1*, *spy* is executing `tcpdump` observing *a*'s traffic, and *b* is running the graphical application `xeyes`. The network traffic between *b* and the host X server cannot be spied because of *ghostification*, as explained in subsection 4.9.1.

4.9 Patches to the C code

We modified the Linux kernel (which includes UML[24]) and VDE to implement a couple of functionalities we needed to support emulation.

4.9.1 Ghostification

A *ghostified* interface[25] is a network interface which remains fully functional in receiving and sending frames but can not be in any way detected or configured by userspace processes, including utilities like `ifconfig`, `route` and `netstat`, and sniffers like `tcpdump`.

An interface can be ghostified and unghostified by calling the new *ioctl*s `SIOCGIFGHOSTIFY` and `SIOCGIFUNGHOSTIFY`.

Ghostification works by making some *ioctl*s fail when their parameter is a currently ghostified interface, returning `-ENODEV` as if the device did not exist.

This functionality has a mostly pedagogical purpose: ghostification is used on guest kernels to allow virtual computers to connect to the host X server in a fashion completely invisible to the user. Users might run sniffers with graphical interfaces, hence needing to communicate with the host X server, but such network traffic should be hidden to the sniffer itself.

More in general we are trying to replicate what would happen with a physical network where graphic works only locally, as in most typical cases.

Hiding ghostified interfaces also saves the student from the unnecessary burden of dealing with the complexity of network-transparent graphics.

4.9.2 Defects and blinking

We modified VDE in just two simple ways:

- In virtual cables' defect support we made the random delay follow a normal distribution rather than a uniform distribution.
- For every frame received or sent by a virtual cable we optionally[26] send blink commands (see subsection 4.8.3, dealing with LED grids) to a `PF_LOCAL` namespace *datagram socket* connected to the main Marionnet process.

[24] Our added functionality does not necessarily requires UML, and also works on host kernels. In order to better test the patch (and to better impress coworkers) one of the authors has been running the patched kernel on his main machines for months now, without stability or performance problems.

[25] The video [29] gives a practical demonstration of ghostification.

[26] This functionality is enabled only when appropriate command line parameters are passed.

4.10 Implementation scope

Our OCaml application consists in about 12,000 nonempty source lines, including comments.

It was developed by the two authors in six man-months, working full-time for three months with even distribution of effort.

5. Relevant code samples

We are now going to show some particularly illustrative code snippets as samples of the different coding styles employed in Marionnet.

5.1 Process implementation

This is the base class of the processes hierarchy, showing how external processes are spawn, terminated, suspended and resumed. Note that we do *not* use `Unix.system`, thus saving the overhead of a subshell invocation per started process.

```
class virtual process =
fun program
    (arguments : string list)
    ?stdin:(stdin=Unix.stdin)
    ?stdout:(stdout=Unix.stdout)
    ?stderr:(stderr=Unix.stderr)
    () -> object(self)
  val pid : int option ref = ref None

  (** Get the spawn process pid, or fail if the process
      has not been spawn yet: *)
  method get_pid =
    match !pid with
      (Some p) -> p
    | _ -> raise (ProcessIsntInTheRightState "get_pid")

  (** Startup the process using command_line, and return
      its pid *)
  method spawn =
    match !pid with
      (Some _) ->
        raise (ProcessIsntInTheRightState "spawn")
    | None ->
        let new_pid =
          (Unix.create_process
             program
             (Array.of_list (program :: arguments))
             stdin
             stdout
             stderr) in
        pid := (Some new_pid)

  (** Kill the process with a SIGINT. This forbids any
      interaction, until the process is started again: *)
  method terminate =
    match !pid with
      (Some p) ->
        (try
          Unix.kill p Sys.sigint;
          (* Wait for the subprocess to die, so that no
             zombie processes remain: *)
          ignore (Unix.waitpid [] p);
        with _ ->
          Printf.printf
            "WARNING: termination of %i failed\n"
            p);
        pid := None
    | None ->
        raise (ProcessIsntInTheRightState "terminate")

  (** By default gracefully_terminate is just an alias
      for terminate. *)
  method gracefully_terminate =
    self#terminate

  (** Stop the process with a SIGSTOP. This forbids any
      interaction, until self#continue is called. *)
  method stop =
    match !pid with
      (Some p) -> Unix.kill p Sys.sigstop
    | None -> raise (ProcessIsntInTheRightState "stop")

  (** Make a stopped process continue, with a SIGCONT. *)
  method continue =
    match !pid with
      (Some p) -> Unix.kill p Sys.sigcont
    | None -> raise (ProcessIsntInTheRightState "continue")
end;;
```

5.2 Forest data structure

The forest data structure is an algebraic polymorphic data type, used for implementing the data structures displayed in the filesystem history and defects GUI.

```
type 'a forest =
    Empty
  | NonEmpty of 'a *          (* first tree root *)
                ('a forest) * (* first tree subtrees *)
                ('a forest);; (* other trees *)
```

Forests are easy to manipulate in a purely functional style, and we rely on higher order for many operations:

```
let rec map f forest =
  match forest with
    Empty ->
      Empty
  | NonEmpty(root, subtrees, rest) ->
      NonEmpty(f root, map f subtrees, map f rest)
```

Forests are saved into Marionnet project files using OCaml *marshaling* support.

5.3 Message passing

A queue is a polymorphic data structure used to implement general purpose inter-thread *message passing*.

The linear complexity concatenation in the method `enqueue` has not been a problem in practice, because of the typically small size of queues; however this data structure could of course be modified to rely on circular arrays instead of lists if the need ever arose.

```
class ['a] queue = object(self)
  val elements = ref []
  val mutex = Mutex.create ()
  val empty_condition = Condition.create ()

  method private __empty =
    !elements = []

  method enqueue x =
    Mutex.lock mutex;
    elements := !elements @ [x];
    Condition.signal empty_condition;
    Mutex.unlock mutex

  method dequeue : 'a =
    Mutex.lock mutex;
    while self#__empty do
      Condition.wait empty_condition mutex;
    done;
    let result =
      match !elements with
        x :: rest -> elements := rest; x
      | _ -> assert false in
    Mutex.unlock mutex;
```

```
    result
end;;
```

The *control thread* mentioned in subsection 4.4 heavily relies on this *thunk*-passing (see the type of the `queue` field) facility implemented with queues.

```
class task_runner = object(self)
  val queue : (unit -> unit) queue = new queue

  initializer
    ignore (Thread.create
              (fun () ->
                while true do
                  let task = queue#dequeue in
                  task ();
                done)
              ())

  method schedule task =
    queue#enqueue task
  method terminate =
    self#schedule
      (fun () -> failwith "asked to terminate")
end;;
```

6. Related work

Some projects employing UML as a network emulation platform for didactic purposes already exist; however they are all nontrivial to use for beginners, and for what we know they only provide static control layer:

- VNUML ([21, 20]), written in Python, allows one to define a network as an XML file, describing the network to be emulated *once and for all*, without allowing any change to network elements while the emulation is happening.
- Netkit ([38]), formerly implemented in Python and curiously re-implemented in Bash, is a set of many interdependent scripts implementing single operations such as starting up and terminating a virtual machine. While powerful, this low-level interface requires a considerable learning effort to master the large set of available command line options.
- *MLN* ([5]) is interesting in its support for both UML and Xen with the same configuration, and the possibility of allocating different virtual network components on different hosts. Like VNUML it defines a network configuration language, and provides a static control layer.

Concerning at least VNUML and Netkit, some related projects exist to build high-level graphical interfaces on top of them: NetGUI ([33]), and vnumlgui ([7], written in Perl). This confirms our intuition about the importance of an easy to use GUI layer, particularly if we consider that the control layer they employ is much simpler than ours.

Although it lies at a different level than the other alternatives, VDE has also been directly used for didactics by its author with success ([12]). Despite working at a lower level of abstraction than the other tools, VDE has the advantage of providing a platform for implementing dynamic control.

7. Conclusions and further work

A virtual network can be an excellent teaching tool both for students and for teachers. It can be also useful to system administrators, scientists and developers in designing, implementing, testing, installing and configuring network applications, services and protocols.

In this report we described Marionnet, a system we have written in OCaml to enable users to define and control an emulated network. Despite its large number of features and the complexity of the related GUI, the application has been developed in just six man-months by the authors. On the basis of our long experience in programming using many different styles and tools —"popular" and otherwise— we seriously doubt that this *challenge* could have been won by using more conventional tools like traditional imperative languages (C, C++, Java, Ada) or "fashionable" scripting languages (Perl, Python).

The application has already been tested in the occasion of a real exam and has been publicly demonstrated[27]; the interest shown by teachers and Department directors makes us confident in a wide adoption in French IUTs[28] starting from the next academic year.

Even if most of the code was written in a functional style using immutable structures such as lists, the peculiarities of OCaml in integrating the functional, imperative and object-oriented paradigms have been profitable for our purposes.

Static checks have been extremely helpful to shorten development time. Very few bugs have been found, and practically all of them in the part dealing with external processes interaction, on which the compiler does not have static control.

Our objectives have been almost completely reached. Only a few features remain to be implemented before packaging the application for being installable on major GNU/Linux distributions. This further work will concern mainly GUI internationalization (as it can be seen from screenshots, message localization is still incomplete, but could be easily enhanced by using, for example, a port of *gettext*: [17, 27]) and enhanced support for teaching activities. We plan to enable teachers to broadcast exercises as Marionnet projects to all the students in a classroom, and eventually get the students' work back.

Actually, this is possible thanks to the *copy-on-write* technology supported by UML, allowing to work with very small project files. The teacher receiving back *cow* files containing the students' modifications to filesystems will be able to manually inspect files or to automatically run analysis scripts.

What we regard as the main current limit of our application is directly inherited from the underlying UML technology, consisting in the possibility to emulate *only* the Linux kernel, and not others. In order to enrich the variety of supported operating systems, we are considering the idea of porting Marionnet to the Xen platform as a further interesting, although not prioritary work.

A tempting possible approach for further developing Marionnet would consist in using Marionnet itself to emulate the network-classroom situation, including the teacher and the students' machines. This kind of "bootstrap" would allow one to use a stable version of Marionnet to run a newer testing version of itself *within* itself.

Acknowledgments

First of all we wish to thank Jeff Dike and Renzo Davoli for their great work on which we based our application, the whole free software community, whose first and foremost contributors remain the GNU and Linux projects, and the authors of OCaml.

Université Paris 13 provides us with the practical means of continuing this deeply satisfying work by financing the Marionnet project.

[27] Colloque Pédagogique National des IUTs en Réseaux et Télécommunications, 30 May - 1 June 2007, Saint-Malo, France.

[28] There are about one hundred IUTs in France, of which about half with Networks or Computer Science Departments (Source: `http://www.cefi.org/IUT/AZ_IUT.HTM`).

References

[1] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs*, 2nd ed. The MIT Press, Cambridge, Massachusetts, July 1996.

[2] ADAMS, D. *The Hitch Hiker's Guide to the Galaxy*. Pan Books, London, 1979.

[3] AT&T LABS. Graphviz - open source graph drawing software. URL: `http://www.research.att.com/sw/tools/graphviz/`.

[4] BARHAM, P., ET AL. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM Press, pp. 164–177.

[5] BEGNUM, K., ET AL. The MLN Project Home Page. URL: `http://mln.sourceforge.net/index.php`.

[6] BELLARD, F. QEMU Open Source Processor Emulator. URL: `http://www.qemu.org`.

[7] BLANC, M. The vmumlgui Project Home Page. URL: `http://pagesperso.erasme.org/michel/vnumlgui/`.

[8] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, J. Design and implementation of a routing control platform. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 2–2.

[9] CHAILLOUX, E., MANOURY, P., AND PAGANO, B. *Developing Applications with Objective Caml*. 2000. Developement d'applications avec Objective Caml, O'Reilly, France.

[10] COUSINEAU, G., AND MAUNY, M. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[11] DAVEY, P. The X user: `xterm` tips and tricks. *j-X-RESOURCE 0*, 1 (oct 1991), 24–30.

[12] DAVOLI, R. Teaching Operating Systems Administration with User Mode Linux. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education* (New York, NY, USA, 2004), ACM Press, pp. 112–116.

[13] DAVOLI, R. VDE: Virtual Distributed Ethernet. In *TRIDENTCOM* (2005), IEEE Computer Society, pp. 213–220.

[14] DIKE, J. User Mode Linux Community Site. URL: `http://usermodelinux.org`.

[15] DIKE, J. User Mode Linux Kernel Home Page. URL: `http://user-mode-linux.sourceforge.net`.

[16] DIKE, J. *User Mode Linux*. Prentice-Hall, 2006.

[17] DREPPER, U., MILLER, P., AND HAIBLE, B. `gettext` Home Page. URL: `http://www.gnu.org/software/gettext/`.

[18] FREE SOFTWARE FOUNDATION. GNU Home Page. URL: `http://www.gnu.org`.

[19] FREE SOFTWARE FOUNDATION. GNU General Public License. URL: `http://www.gnu.org/copyleft/gpl.html`, 2007.

[20] GALÁN, F., AND DECCIO, C. T. VNUML Language Reference. URL: `http://jungla.dit.upm.es/vnuml/doc/1.6/reference/index.html`.

[21] GALÁN, F., AND FERNÁNDEZ, D. Virtual Network User Mode Linux. URL: `http://jungla.dit.upm.es/vnuml/`.

[22] GARRIGUE, J., FAUQUE, H., FURUSE, J., AND KAGAWA, K. Lablgtk, a Gtk interface for Objective Label. URL: `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`.

[23] ISHIGURO, K., ET AL. Quagga Home Page. URL: `http://www.quagga.net`.

[24] KRAP, A. Setting up a virtual network laboratory with User-Mode Linux. Tech. rep., 2004. Masters programme on System and Network Administration, University of Amsterdam. URL: `http://www.os3.nl/arjen/snb/asp/asp-report.pdf`.

[25] KRASNYANSKY, M. Universal TUN/TAP device driver. URL: `http://www.kernel.org/pub/linux/kernel/`, FILE: `people/marcelo/linux-2.4/Documentation/networking/tuntap.txt`.

[26] LAWTON, K., ET AL. Bochs Home Page. URL: `http://bochs.sourceforge.net`.

[27] LE GALL, S. `ocaml-gettext` Home Page. URL: `http://sylvain.le-gall.net/ocaml-gettext.html`.

[28] LEROY, X., DOLIGEZ, D., GARRIGUE, J., RÉMY, D., AND VOUILLON), J. *The Objective Caml system, Documentation and user's manual*, release 3.10 ed., 2007. URL: `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[29] LODDO, J.-V., AND SAIU, L. A sample work session with Marionnet. Video. URL: `http://www-lipn.univ-paris13.fr/~loddo/video.ogg`.

[30] MILNER, R. A proposal for standard ml. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming* (New York, NY, USA, 1984), ACM Press, pp. 184–197.

[31] MÜRI, F. `uml_switch2` Home Page. URL: `http://www.uxu.ch/uxu/software/uml_switch2`.

[32] NARBEL, P. *Programmation fonctionnelle, générique et objet. Une introduction avec le langage OCaml*. Vuibert Informatique, 2005.

[33] NEMESIO, S. C., DE LAS HERAS QUIRÒS, P., BARBERO, E. M. C., AND GONZÀLEZ, J. A. C. Early experiences with NetGUI laboratories.

[34] OH, S. Gtk+ 2.0 Tree View Tutorial using OCaml - Adaptation of Tim-Philipp Muller tutorial. URL: `http://plus.kaist.ac.kr/~shoh/ocaml/lablgtk2/treeview-tutorial/`.

[35] PERLMAN, R. J. An algorithm for distributed computation of a spanningtree in an extended LAN. In *SIGCOMM* (1985), pp. 44–53.

[36] PEYTON JONES, S., ET AL. Report on the programming language haskell 98, Feb. 1999.

[37] REMY, D., AND VOUILLON, J. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems 4*, 1 (1998), 27–50.

[38] RIMONDINI, M. Emulation of Computer Networks with Netkit. Tech. rep., 2007. Università degli Studi di Roma Tre. URL: `http://dipartimento.dia.uniroma3.it/` FILE: `ricerca/rapporti/rt/2007-113.pdf`.

[39] SAIU, L. The epsilon project — a functional language implementation *(MD thesis)*. URL: `http://etd.adm.unipi.it/theses/available/etd-02012007-071542` Software available at `http://savannah.gnu.org/projects/epsilon`, Feb. 14 2007.

[40] SCHEIFLER, R. W., AND GETTYS, J. The X window system. *ACM Trans. Graph. 5*, 2 (1986), 79–109.

[41] STEVENS, W. R. *Unix Network Programming*. Prentice Hall, 1990.

[42] TAYLOR, O., ET AL. Gtk+ - GNU toolkit for X windows development. URL: `http://www.gtk.org`.

[43] TORVALDS, L. Linux Home Page. URL: `http://www.linux.org`.

[44] TURNER, D. A. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture* (New York, NY, USA, 1985), Springer-Verlag New York, Inc., pp. 1–16.

[45] VAN BEERS, M., CHAPLIN, D., ET AL. Glade - A User Interface Designer for GTK+ and GNOME. URL: `http://glade.gnome.org`.

[46] VMWARE, INC. VMware Home Page. URL: `http://www.vmware.com`.