

# Virtual environments for networking experiments

*“Analytical Network Project”*

*Masters System and Network Administration,  
University of Amsterdam, the Netherlands\**

Jeroen van der Ham  
jeroen@os3.nl

Gert Jan Verhoog  
gjv@os3.nl

July 1, 2004

## **Abstract**

In this paper, we discuss existing projects for configuring virtual networks consisting of User Mode Linux instances. We explore their strengths and weaknesses and define constraints and requirements for an ideal UML-based virtual network configuration and monitoring tool that is aimed towards experimenting with OSI layer two and three networking protocols in an educational environment. We provide a proof-of-concept implementation of a tool that tries to meet these requirements.

---

\*Supervised by Karst Koymans and Jeroen Scheerder

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Existing configuration tools for UML networks</b>	<b>4</b>
2.1	User Mode Linux . . . . .	4
2.2	My Linux Network . . . . .	4
2.2.1	Installation . . . . .	5
2.2.2	Downloading templates . . . . .	5
2.2.3	Running MLN . . . . .	5
2.2.4	Conclusions . . . . .	5
2.3	Virtual Network User Mode Linux . . . . .	6
2.3.1	Installation . . . . .	6
2.3.2	Configuration . . . . .	6
2.3.3	Running VNUML . . . . .	6
2.3.4	Conclusions . . . . .	7
2.4	SNB UML . . . . .	7
2.4.1	Dedicated filesystem . . . . .	7
2.4.2	Configuration and running . . . . .	8
2.4.3	Conclusions . . . . .	8
2.5	Comparing MLN, VNUML and SNB UML . . . . .	8
<b>3</b>	<b>Requirements for an ideal configuration tool</b>	<b>9</b>
3.1	Avoid the need for super-user privileges . . . . .	9
3.2	Use an intuitive model . . . . .	9
3.3	The software should be clearly understandable . . . . .	9
3.4	Don't restrict users' activities . . . . .	9
3.5	User stories . . . . .	10
<b>4</b>	<b>Proof of concept</b>	<b>10</b>
4.1	Key elements . . . . .	10
4.1.1	Communication with virtual machines . . . . .	10
4.1.2	Configuration . . . . .	11
4.2	Technical requirements . . . . .	12
4.2.1	Controlling UML instances . . . . .	12
4.2.2	Specifying configurations . . . . .	12
<b>5</b>	<b>Project planning</b>	<b>13</b>
5.1	Features and milestones . . . . .	13
5.2	What our prototype will not do . . . . .	14
5.3	Unexpected difficulties . . . . .	14
5.3.1	Communicating with UML instances from the outside world . . . . .	15
5.3.2	Closing active UML instances . . . . .	15
5.3.3	Installing software for UML instances . . . . .	15
5.3.4	Graphical interface toolkits . . . . .	16
<b>6</b>	<b>Description of the prototype</b>	<b>16</b>
6.1	Configuration files . . . . .	16
6.2	Software structure . . . . .	16
6.2.1	VNEConfig . . . . .	16
6.2.2	VNEDevice . . . . .	17
6.2.3	VNEInterface . . . . .	17
6.2.4	VNENetwork . . . . .	18
6.2.5	VNETextInterface . . . . .	19

<b>7</b>	<b>Possible enhancements</b>	<b>19</b>
<b>8</b>	<b>Conclusions</b>	<b>20</b>
<b>A</b>	<b>Extended example</b>	<b>22</b>
<b>B</b>	<b>VNE Document Type Definition</b>	<b>24</b>

## List of Figures

1	MLN's user interface . . . . .	5
2	Simple example of a VNUML configuration. . . . .	6
3	Example of interconnected devices . . . . .	11
4	From Don Libes' article on <code>expect</code> . . . . .	13
5	a minimal configuration file . . . . .	17
6	UML class diagram showing objects used in our software . . . . .	18
7	A network with three broadcast segments. . . . .	18
8	Example text interface session . . . . .	19
9	An extended example with three switches and three hosts. . . . .	22

# 1 Introduction

There are several tools available for experimenting with networks. Because of limited availability, high costs and inflexibility it is usually not practical to work with actual cables, switches or routers. This is why in a learning and research environment it is sometimes preferred to work with virtual machines in a virtual network. The focus of these research environments roughly fall into two categories:

- researching OSI layer 1 (physical layer)
- researching OSI layers 2 (data layer) and 3 (network layer).

The first category concentrates mostly on issues such as timers, delays, noise, properties of different cables, etc. The second category is meant for designing and understanding OSI layer 2 and 3 network protocols for routing and switching in networks with hosts, hubs, switches and routers. It is this last category that our project focuses on.

A lot of work in this category utilizes User Mode Linux[8], or *UML* for short. UML is a modification to the Linux kernel[18] allowing it to run as an ordinary user process inside a running Linux environment. This provides a way of creating and running virtual Linux machines. UML instances can be connected to virtual ethernet networks which are accessible from within UML through interfaces configured with *ifconfig*. All Linux software available for routing, switching, packet sniffing, etcetera also works in these simulated environments.

## 2 Existing configuration tools for UML networks

### 2.1 User Mode Linux

User Mode Linux[7, 8] is a project aimed at using the Linux kernel to create virtual machines within a Linux environment by allowing a Linux kernel to run as a user process. User Mode Linux kernels are used for several purposes such as kernel development, process debugging, sandboxing and by commercial hosting providers providing virtual Linux machines to their customers.

The UML project seeks to provide users with the basic tools to create and configure virtual Linux machines. Besides maintaining a kernel patch tree, it also provides some utilities for managing UML instances and hooking them up in a network. Due to the project's generic nature, it does not provide any means to use UML in a specific way.

Several utilities are available for User Mode Linux, the most important (for our purpose) being `uml_switch`, a daemon that runs on the host to create a virtual shared or switched virtual ethernet network. This daemon creates a socket to which a User Mode Linux kernel connects an ethernet interface.

User Mode Linux itself can be used to create virtual networks, but besides a basic virtual ethernet infrastructure, UML does not ship with tools to configure a number of UML instances in a virtual networking environment. Every UML instance has to be configured and run manually. For this reason several other projects provide tools to configure and run several UML instances more effectively. These projects are discussed in the following sections.

### 2.2 My Linux Network

*My Linux Network*[5], or *MLN* for short, is a small program that generates UML networks based on a specification written in the *MLN-Language*. The basic elements of an MLN configuration are *switch* and *host*. A switch uses the simulated network implemented by the UML utility `uml_switch`. A host is a User Mode Linux instance, using a file system installed on a disk image or in a directory tree.

### 2.2.1 Installation

Installation of MLN is simple and straightforward: just unpack the archive. The archive contains several documentation and example files and an executable perl file named “mln”. For convenience, the executable can be placed inside a directory that is in the “PATH” variable. The User Mode Linux kernel must be installed and configured separately.

### 2.2.2 Downloading templates

One or more ready-made filesystems for use with MLN can be downloaded with the command “mln download.templates”. These file systems are available in two flavors: disk images and directory trees. Using disk images requires super-user<sup>1</sup> privileges, since MLN configures the file system on the disk image by mounting it on the host. Only the super-user can mount and unmount disk images under Linux. Ordinary users need to use MLN’s “.dir” flavored templates. These templates are directory trees that UML can read using the *hostfs* option. MLN specifically mentions that these “.dir” flavored templates are meant for normal users without super-user privileges, as can be seen in figure 1. After replying ‘y’, the template is downloaded. MLN then tries to install this template but fails with a number of “*Cannot mknod: Operation not permitted*” errors. The command *mknod* is used to make devices in Unix /dev directories. It fails because creating devices under Linux requires super-user privileges. A solution to this problem would be using *devfs*[12] on the UML filesystem instead of creating devices manually. *Devfs* dynamically creates the necessary devices when the system boots, thus avoiding the need to use *mknod*.

```
Found this template: sarge-thick.dir Version: 0.1 NEW!
Size: 37MB
Description:
"This filesystem is made from debootstrap and modified a bit. It is
the Debian Linux distribution called sarge (currently testing). A few
additional apps are installed too: tcpdump, bonnie++, vtun, hping2,
gcc etc. This image is 2.6 ready. This is the folder counterpart of
the ext2 version and is meant for user-mode building."
Would you like to download it? (y/n) (Default: n)
```

Figure 1: MLN’s user interface

### 2.2.3 Running MLN

We were not able to run an MLN project using one of the included examples, “simple-network.mln” because MLN needs to modify the template’s file system, which requires mounting of the disk image. Since we do not have super-user privileges on our testing machine, this was not possible. The failure occurred both with disk images and “.dir” flavoured templates. According to the documentation, using directory trees instead of disk images to allow ordinary users to build MLN projects is still in beta, so we will probably be able to use later versions of MLN.

### 2.2.4 Conclusions

Allowing MLN users to download filesystem templates makes it very easy for them to start using virtual networks. The user interface also provides a lot of functionality. However, it is too much for our purpose and this distracts from the goal of experimenting with virtual networks.

We believe MLN provides a versatile and easy method for setting up and operating virtual networks. Unfortunately, we are not able to use MLN as part of our project, since one of our design goals is that super-user privileges should not be required to use it.

---

<sup>1</sup>We use the term *super-user* for any user with a user id of 0 on a Unix system. Usually, this user is named “root”.

## 2.3 Virtual Network User Mode Linux

Another project based on UML is *Virtual Network User Mode Linux (VNUML)*[11]. Like MLN this projects consists of a small program to convert specifications into UML configurations. The VNUML project is a work in progress and still improving.

### 2.3.1 Installation

The installation uses a standard `configure` and `make` approach, which can also fetch some packages it depends on. Apart from these packages, the VNUML parser requires several Perl modules<sup>2</sup> which are not installed by default and may require some effort to install.

### 2.3.2 Configuration

The configuration language for VNUML is XML based, defined in a DTD. A very simple example is given in figure 2. In this example there is only one node, named “host1”. There is a global configuration, defining a simulation name, an ssh identity file, which should be used for remote control of the UML instances, offsets for mac-address and ip-address and what shell should be used for the hosts. The host itself uses the kernel specified to mount a *Copy-On-Write (COW)* image and once booted binds an xterm to “con0”. There are many more options available in the configuration file, for a detailed overview see [10].

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vnuml SYSTEM "/var/vnuml/vnuml.dtd">
<vnuml>

  <global>
    <version>1.3.0</version>
    <simulation_name>simple</simulation_name>
    <ssh_key>/root/.ssh/identity.pub</ssh_key>
    <automac offset="0"/>
    <ip_offset>0</ip_offset>
    <shell>/bin/sh</shell>
  </global>

  <!-- NODES -->
  <vm name="host1">
    <filesystem type="cow">/var/vnuml/root_fs_tutorial</filesystem>
    <kernel>/usr/local/bin/linux-2.4.22-um</kernel>
    <boot>
      <con0>xterm</con0>
    </boot>
  </vm>
</vnuml>
```

Figure 2: Simple example of a VNUML configuration.

### 2.3.3 Running VNUML

Once installed, VNUML provides, among other things, a parser for the XML configuration files. This parser can read the XML configuration files and build a UML environment from it. When the simulation is started, VNUML provides several ways to control the different UML instances, both graphical, console based and command-line based.

The first thing VNUML checks on startup is whether it is executed with super-user privileges. These are needed to create the *tun/tap* devices that are used to configure the hosts and allow the

<sup>2</sup>XML-Parser, XML-RegExp, XML-DOM, XML-Checker, TermReadKey, Math-Base85, Net-IPv4Addr, Net-IPv6Addr and libxml-perl

user to communicate with them. Using super-user privileges, it can also edit `/etc/hosts` to add aliases for the UML instances, as to make life easier for the user.

The command-line interface to VNUML allows the user to start a predefined (in the configuration file) set of commands to be executed. This set of commands can be different for each host, but they are all started at the same time. This allows the user to easily set up an experiment and start or stop the hosts in an easy way.

The UML-instances all use separate disk-images, or share one using *Copy-On-Write*. This last approach has the advantage that all UML instances can run off the same image yet change it in a different (stateful) way. However, the disadvantage of this approach is that any change in the disk image invalidates all changes made by the UML instances.

#### 2.3.4 Conclusions

VNUML is a project that is still in development, but it is a promising project. The developers are working hard on it and are still implementing new features. It also features clear documentation and has a well-defined configuration syntax, which allows for simple and clear configuration of the hosts.

However as with MLN, it is not suitable for our project, since super-user access is a hard requirement in the program. The intended use of VNUML is for lab practices where the teachers start the environments and the students experiment with the simulated machines themselves and not with creating and modifying with the network between the instances.

## 2.4 SNB UML

During our Masters education our fellow student Arjen Krap developed his own approach to virtual networking with UML[4, 13], which we call *SNB UML*<sup>3</sup>. Similar to the MLN and VNUML projects, it is based on UML and uses scripts to set up networking and to allow the user to configure several UML instances.

SNB UML's approach differs in that it does not require super-user privileges to operate. SNB UML uses UML's *hostfs* feature, which allows the virtual machines to directly use part of the host file system as a read-only (root) file system. Configuring UML instances is done by passing environment variables as arguments to the Linux executable. This means that there is no need to mount and modify disk images. Furthermore, Krap's system does not use UML-to-host networking with *tun/tap* devices.

### 2.4.1 Dedicated filesystem

Krap built his filesystem to support one specific task: experimenting with virtual networks. Software that is not necessary to operate the system and irrelevant to networking is not installed. This results in a Linux installation with small memory requirements – 10Mb per instance is enough – and no distracting and irrelevant software components. It consists of the following elements:

- a kernel with routing, bridging and packet filtering support
- *Busybox*[1] which provides core Unix tools
- *Zebra*[9], a routing configuration tool with a command-line interface similar to Cisco's IOS[6]
- *tcpdump* for traffic monitoring

This filesystem uses Linux's *devfs* feature to dynamically build the device entries in the `/dev` directory and a RAM-disk mounted on `/var` to provide a writable filesystem for temporary files.

---

<sup>3</sup>SNB refers to the name of our Masters' course: '*Systeem- en Netwerkkebeheer*'[17]

## 2.4.2 Configuration and running

Configurations are specified in a small Python script responsible for starting, configuring and stopping the UML instances and networking infrastructure. Configuration parameters are passed to the UML instances by using environment variables, specified on the `linux` command-line. These variables are placed in the environment inside the UML instance, where the startup configuration scripts use them to configure network interfaces, networking daemons and other system settings. By default, the Linux kernel supports up to eight command-line variables. Since this approach uses more, this number has to be increased at compile-time.

The script starts the networking infrastructure and the UML instances. For every UML instance, an `xterm` is started with a shell running inside the UML instance. This script is also used to stop every UML instance.

## 2.4.3 Conclusions

Arjen Krap's system has several good features. It uses a highly specialised Linux system that is geared towards networking experiments. This means the UML instances have a small memory footprint and there is no unnecessary and distracting software installed. Furthermore, SNB UML's system does not require super-user privileges to use.

A disadvantage is that the system is not easy to use. Configurations have to be specified by changing data inside a Python script. The configuration itself is very "low-level", requiring a user to specify a network almost in terms of command-line arguments for `uml_switch` commands and the Linux kernel.

The script is not always successful in stopping all UML instances, resulting in lots of zombie `linux` processes on the host system after a while.

Arjen Krap's efforts in creating a small Linux system suitable for networking experiments was extremely valuable in our work. The filesystem we use in our project is largely based on his work.

## 2.5 Comparing MLN, VNUML and SNB UML

Both MLN and VNUML assume that the user setting up the virtual systems has super-user privileges, for different reasons: MLN mounts and modifies the file system on a disk image to configure UML instances and VNUML sets up UML-to-host networking by creating a *tun/tap* device on the host. MLN and VNUML assume that in an educational lab setting, the teacher – who is assumed to have super-user privileges – sets up the UML instances for the students. In our opinion, this is undesirable.

The SNB UML system is specifically designed to support virtual networking experiments, whereas VNUML and MLN are more generic. For VNUML and MLN several ready-made filesystems are available. These filesystems contain typical Linux distributions, which means they are fairly complete in terms of available software. The drawback is that these systems have a larger memory footprint, up to 256Mb. In an educational lab with modest hardware where over twenty students each start over five UML instances, this is not practical. For our purposes, both MLN and VNUML contain too much functionality that distracts from the task of experimenting with networks. We think that a small system that only contains the bare essentials is beneficial to understanding more about the technology and principles of networking. Arjen Krap's filesystem is a good example of such a small and dedicated system.

There is a large difference between the three applications in how the network configurations are defined. MLN uses a custom configuration language. In SNB UML, the script itself is the configuration and if the user wants to change the configuration, some knowledge of Python is needed to be able to change the configuration. Configuring VNUML is easier, because their configuration language is based on XML, which most users already are familiar with.

We think the projects discussed here all see the virtual networks from a technical or "system" point of view: The elements that are specified in configuration closely follow the implementation of User Mode Linux and its utilities. Most notably, configuration of the virtual networking daemon `uml_switch` is required. While UML instances that function as hosts, bridges or routers have

real-world counterparts, UML's switching daemon is not easily classified as one piece of physical networking equipment or another. Several UML instances can be connected to one `uml_switch` socket, making `uml_switch` something between an ethernet cable and a switch or a hub.

Apparently, the current projects do not closely model actual physical networks consisting of cables, hubs, hosts and other equipment. But since they are meant to illustrate the working of real-world situations with various networking topologies and networking protocols that allow users to gain hands-on experience, it is not useful to think of these systems as logical or mathematical models for networking. The question whether the mentioned projects accurately model logical models is not relevant: This kind of software for virtual networking experiments *should* follow and emulate physical networking equipment.

For this reason, we would like to have a system that abstracts from these implementation issues with `uml_switch`. Intuitively, we think of networks as hosts, hubs, switches, routers and UTP cables. We want our virtual equipment to be modeled after this intuition.

### **3 Requirements for an ideal configuration tool**

We propose a number of requirements for an ideal configuration, manipulation and monitoring tool for virtual network experiments. We concentrate on requirements from a user's point of view. Technical requirements are discussed later.

#### **3.1 Avoid the need for super-user privileges**

Users should be able to start UML instances without the need for super-user privileges. Super-user access should be necessary only for administration tasks of the host machine. We expect that system administrators will be less reluctant to allow teachers and students to work with the software if they are confident that security of the host system is not at risk. SNB UML uses part of the host's filesystem as UML's root filesystem using UML's *hostfs* feature. Our software copies this approach.

#### **3.2 Use an intuitive model**

As discussed before, current UML projects for virtual networking closely follow UML's implementation for virtual networking with `uml_switch`. We believe that users experimenting with virtual networks should not need to be concerned about these implementation details. Instead, they should be able to concentrate on building virtual networks that closely mimic physical networks. The virtual equipment that should be available to users consists of connections between endpoints, e.g. UTP cables that connect two ethernet interfaces, hubs, switches, routers and hosts. In a configuration, changing a device from a hub to a switch should be as easy as changing a keyword in the device's definition.

#### **3.3 The software should be clearly understandable**

While it is important to use an intuitive model for our virtual networks, we believe that users should be able to clearly understand the technical details of the software. The boot sequence and configuration of UML instances should be as simple as possible. Furthermore, an ideal software tool for virtual networking experiments should not distract the user with unnecessary software. We need a UML filesystem that contains only a bare-bones Linux and the necessary tools to support networking configuration and monitoring. This means that our software should only contain essential features.

#### **3.4 Don't restrict users' activities**

The software should be easy to control and allow users to configure, operate and monitor virtual networking devices with simple commands. At the same time, users should have full control over the UML instances. Shell access to running UML instances should be available and users should not

be restricted in making non-functional, faulty or unusual configurations in running UML instances or configuration files.

### 3.5 User stories

The user should be able to:

- specify a configuration for an UML network in an XML file
- start and stop UML instances (referenced by name) or the complete configuration
- see information about the running UML configuration or a single instance
- get a command shell or a zebra connection on an UML instance
- monitor traffic on an interface (with tcpdump)
- bring interfaces up or down
- configure parameters of interfaces
- send a command to a running UML instance
- have an indication of the traffic flow in a complete UML network
- see a list of available commands and options

## 4 Proof of concept

To test our ideas and understand the technical difficulties of a software tool that meets the requirements we formulated earlier, we need to build a proof-of-concept implementation. We do not aim to create a full-featured finished product. Instead, we present our prototype as a partial demonstration of our ideas, together with ideas for possible future development. Based on the requirements outlined in section 3, we discuss several concepts and key elements of the proposed tool, followed by a discussion of the technical requirements and components of our software. Furthermore, we constrain the scope of our software by defining features we will *not* implement at this time. After describing the current implementation of the software, we present a roadmap for possible enhancements and future development.

### 4.1 Key elements

Before technical requirements and design decisions can be discussed, the key elements of our software need to be defined.

#### 4.1.1 Communication with virtual machines

There is a strict distinction between the *inside world* of UML instances and the *outside world* of the host computer. The possibilities for communicating between the host and UML instances, for example for collecting status information or configuring network interfaces, are limited. The following methods are available:

**TCP/IP tunnel** User Mode Linux is able to set up TCP/IP networking between an UML instance and the host using *tun/tap* tunneling devices. Unfortunately, creating *tun/tap* devices on the host requires super-user privileges.

**file system** UML instances can mount a directory on the host as a file system. Information between host and virtual machine can be transferred by reading and writing files. This could be useful for configuration files (e.g. for routing software). MLN uses a similar technique, where the MLN software configures UML instances by modifying the file system on a disk image.

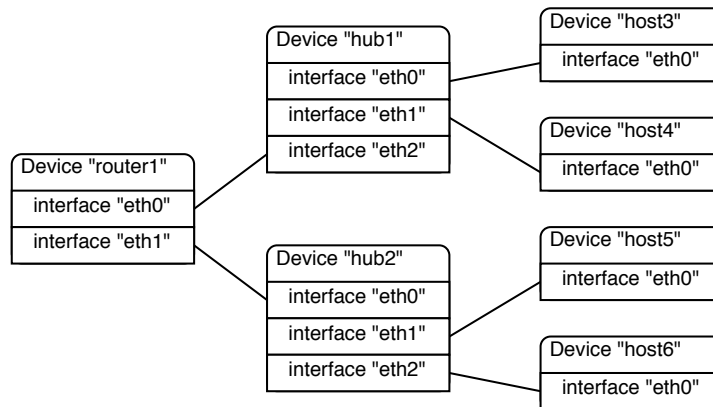


Figure 3: Example of interconnected devices

**uml\_mconsole** UML's *management console* is “a low-level interface to the kernel”<sup>4</sup>. Using the management console, an UML instance can be halted, rebooted, the `/proc` filesystem can be queried for kernel information and some kinds of device entries can be configured, among other things.

**command-line arguments** Yet another possibility for passing information to UML instances is specifying key/value pairs as command-line arguments to the kernel at startup time. These key/value pairs exist as environment variables inside UML instances. This is the approach used by SNB UML, e.g. for configuring network interfaces. Startup scripts in SNB UML's root file system check the values of environment variables specified on the command-line and configure the virtual machine at boot time. It is obvious that this method provides one-way communication only, from the host to an UML instance.

**pseudo terminals** Within an UML instance, several virtual consoles can be defined. A virtual console provides interactive input and output to users. On the startup command-line, these virtual consoles can be attached to pseudo terminal devices on the host. Furthermore, one virtual console uses `stdin/stdout/stderr` which can be captured easily by software that starts the UML instance. This is the method we use in our software.

#### 4.1.2 Configuration

Using virtual networks consists of two tasks: designing a configuration and operating the virtual network. Designing can be done beforehand in an external editor or interactively from within the software. The interactive approach is especially suitable for software with a graphical interface, that allows users to drag and drop devices and connections into place. A text file containing specifications for a network makes sense when designing a configuration beforehand in an editor. Regardless of the configuration method, the same kinds of elements are manipulated.

A virtual network configuration is a graph, where nodes are devices and edges are connections between devices. A device is a hub, router, switch or host. Devices have zero or more ethernet interfaces. A network endpoint is a single interface on a single device. Connections are made between endpoints. An example is shown in figure 3.

Network interfaces optionally have IPv4 or IPv6 parameters such as IP-address and netmask. Switches are devices that allow interfaces to be grouped in a single bridge interface. Routers are configured to run routing software that implement different routing protocols and hosts are generic virtual machines not configured for any specific task, much like an ordinary desktop computer.

<sup>4</sup><http://user-mode-linux.sourceforge.net/mconsole.html>

## 4.2 Technical requirements

To realize the above conceptual model, we make use of a number of existing software tools, which we describe below:

**User Mode Linux Kernel** The base of our system. The UML kernel is used as the basis of each virtual host instance.

**uml\_switch** A separate UML utility which is used to connect UML instances to other UML instances, the host computer or the Internet.

**screen** Average Linux hosts have several (virtual) terminals to allow users to login multiple times and start different things at the same time. To avoid window clutter for users of our experimentation software simulating several devices, we do not automatically create xterms for these devices. Instead we use `screen` to connect the different consoles to sockets, which the user can interact with using `screen`. Interacting with these machines can be done in the text console, using one `screen` instance, or using xterms with one instance per xterm.

**Python** Our programming language of choice. We chose Python because it has a large user base, a very large library of standard modules and is well-suited for rapid prototyping.

**PExpect module**[16] Our software communicates with an UML instance via its main virtual console. Communication is handled by a pure Python module with functionality similar to `expect`[14].

**XML** The configuration files for VNE are written in XML. We have also written a DTD to describe the structure of these configuration files. This also allows us to let the user use `xmlint` to verify the validity of the configuration file.

**Busybox** A very small and bare Linux environment for small and embedded systems. It provides a single executable that acts as a replacement for all essential GNU file and shell utilities. It allows us to make a very small filesystem for the devices, with a limited disk and memory footprint.

**Zebra** An open and free implementation of TCP/IP routing protocols (such as BGP4, RIPv1, RIPv2 and OSPFv2)

### 4.2.1 Controlling UML instances

Our software needs to be able to control and configure UML instances. The software connects to an UML instance's main virtual console where a shell is available. This shell is used for sending configuration commands to the UML instance and retrieving output. To achieve this kind of functionality, a number of steps have to be taken: a Linux kernel must be started as a subprocess; the software needs to connect file handles to the subprocess' stdin and stdout streams; the subprocess' stdout must be periodically polled using *non-blocking IO* to capture important output, e.g. to check if a shell prompt is available; as a response to certain output, commands must be sent to the subprocess' stdin stream; etcetera.

Dealing with pseudo terminals and using non-blocking IO to communicate with subprocesses is not easy to implement. Luckily, a very useful utility called `expect`, created by Don Libes, automates this process to a large extent. The benefits of using `expect` are best illustrated by a picture from [15], shown in figure 4.

### 4.2.2 Specifying configurations

Defining a virtual network configuration is done with an XML-based configuration language, specified in a DTD. Using XML has a number of advantages. First, most users are familiar with XML. This means that users do not have to learn the syntactical details of a custom configuration language. Users still have to learn the names of elements and attributes that can be used in configuration files,

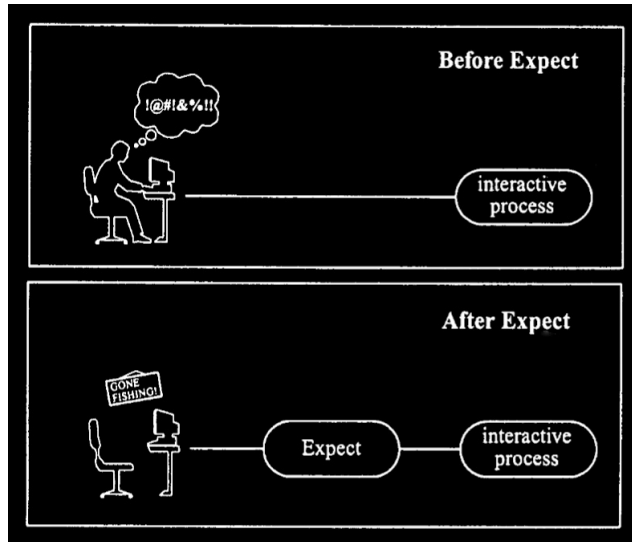


Figure 4: From Don Libes' article on expect

but this would not be different with custom configuration languages. Second, using XML greatly simplifies our software's parsing routines, since XML parsers are available for a large number of software environments.

The configuration files, describing a complete simulation, contains three sections: defaults, devices and connections. The first section contains default settings for the devices, such as the kernel and filesystem that should be used. This can be overridden in each individual device declaration. The device configuration section contains an enumeration of different devices. The last section of the configuration file specifies connections between endpoints (i.e. interfaces of devices), where an endpoint is defined as a single interface on a single device.

## 5 Project planning

This section describes the planning of our prototype development. The goal of developing this software is to understand the technology necessary for a virtual networking experimentation tool. Before we start coding, we must define the requirements for the software, which are outlined in the previous sections. Furthermore, we need some kind of plan defining the order in which features are developed. Another part of planning is constraining the scope of our prototype: we need a list of things our prototype will *not* do. It is important to understand that this planning is not fixed. Due to technological difficulties and new ideas that pop up during the development of the software, the plan can and will change.

### 5.1 Features and milestones

Although creating a detailed schedule was not possible, we needed a plan because "We need to ensure that we are always working on the most important thing we need to do"[3]. Our plan consists of a number of *milestones* for our software. These milestones define what our software should be able to do at a certain moment. These milestones effectively indicate the order in which features will be implemented. The planned milestones are listed below in chronological order:

1. We have a programming environment, with tools such as a Python with TkInter and wxWindows support, and some understanding of Python modules for XML parsing, subprocesses, file IO, etcetera

2. There is a working Linux file system and UML kernel. Our file system is based on Arjen Krap's SNB UML file system and consists of *Busybox* for core Unix tools, *tcpdump* for traffic monitoring and the *Zebra* routing software
3. The software is able to programmatically start and stop an UML instance
4. An UML instance's networking is configured by the software
5. The software constructs objects representing devices and network topology based on a configuration file
6. Two UML instances are able to communicate with each other using TCP/IP. The necessary virtual network infrastructure is set up by the software using `uml_switch`
7. Devices and network connections can be dragged'n'dropped in a graphical interface
8. Working hub devices that integrate multiple network connections in a larger shared network
9. Users can start and stop UML instances and network connections
10. Users can start an interactive shell in an UML instance
11. Router and Switch devices work

This list of milestones was constructed before coding started. After several experiments with graphical interfaces, we decided not to implement item 7, for reasons discussed in section 5.3.4.

## 5.2 What our prototype will not do

There are numerous interesting features that we can not implement due to time constraints. We have to set priorities and only implement the most essential features. Some features that are not implemented would not be helpful in our understanding of the necessary technology or would require too much time.

- fool-proof error checking and exception handling
- an easy to install package
- integration of useful graphical software like *Ethereal*, a traffic monitor
- multiple xterms per hosts (with more virtual consoles)
- passing router configuration files to UML instances
- mounting user defined directories as file systems
- exporting/importing MLN and VNUML configs
- exporting config to dot files for *GraphViz*[2]

## 5.3 Unexpected difficulties

During the creation of our prototype we ran into several problems. These were not insurmountable in that they stopped us from creating a 'proof of concept' implementation, but some are still open problems.

### 5.3.1 Communicating with UML instances from the outside world

Normal Linux machines have several ways of communicating with the outside world. Once started, they have six (or more) virtual consoles, possibly an X Window system, which allows the user to start as many XTerms as he wants and also (possibly) an SSH server, which allows users to connect to the machine and control it from there.

As described in section 4.1.1 there are several options available to communicate with UML instances. However, since we restricted ourselves to not using super-user privileges, this ruled out several options; For example, the SSH server can still be started, but to create a net connection to a UML instance, a *tun/tap* tunneling interface has to be used. Unfortunately creating these interfaces requires super-user privileges. Because no network connection can be made directly to the UML instance, it is not possible to make use of the X window system of the virtual instance.

This leaves us with the option to use the virtual consoles of the virtual Linux instances. UML uses `stdin`, `stdout` and `stderr` as its first virtual console. When our software spawns new instances, we use this tuple to read its output and send commands to it. Reading and parsing the output of the spawned instances is hard, but fortunately, *expect* makes the life of a programmer much easier (see figure 4). Due to the restrictions described above, this is our only life-line to the UML instances. And we do not want the user to run arbitrary commands on it, leaving it in an unknown (possibly blocked) state, since then we have no way to control it.

So we created a separate interface for the user, using the UML command line to specify that we want to bind a virtual console to a pseudo terminal and once we figure out which one this is (by reading from `stdout` using *expect*), we create a `screen` session that allows users to use the virtual console. This method works, but is hard to extend so that the user can start an arbitrary number of terminal sessions on a simulated host.

### 5.3.2 Closing active UML instances

Our experiences with SNB UML was that after a lab session with multiple students working on different simulations, several ‘zombie’ processes remained and these needed to be killed manually. An advantage of having an interactive process, in this context, is that once the interpreter is closed down (by the user or due to an unforeseen error), all the UML instances are immediately killed as well, since they are child processes of the interpreter.

However, when we tried to cleanly stop (`halt`) the simulated UML instances, this resulted in zombie processes as well. At first this was because of lingering output of the shutdown process, which was not read. In itself this is not a problem, but the processes broke their connections with the parent process and so they were not killed once the interpreter exited. When we discovered what was causing these zombie processes, we altered our shutdown methods and allowed for more time to catch output. This provided the result we wanted, but now we still have an open problem with lingering shell sessions which do not seem to quit cleanly.

### 5.3.3 Installing software for UML instances

Building and installing software from source is not very hard. Most packages come with clear instructions on how to configure and build it and most of the time, instructions can be found on the web as well. And most software can be easily installed as ordinary user as well (especially when it uses *autoconf*).

This is different when configuring and installing software on the host machine meant for use on the simulated machine. This means that software is located on different locations when installing it (on the host machine) and when running it (on the simulated machine). It also typically means that the libraries to which the executable was linked are not available on the expected locations.

A solution is that all software that must be installed for the simulated devices must be statically linked after compiling (typically by using “`LDFLAGS=-static`”). Then the software can be installed on any desired location and it will still run fine.

The downside of using static linking is that it requires more disk space, because libraries are not shared (each executable contains all required libraries) and this also requires more memory if they

are run in parallel. Also, it is not possible to upgrade the libraries, without relinking the object files (provided you still have these).

### 5.3.4 Graphical interface toolkits

In our original prototype design we envisioned creating a graphical user interface (GUI) to design networks, dragging and dropping devices on the screen and drawing connections between them. We planned to let the user start and stop instances and to interact with the running instances using the GUI (change network interface configurations, starting `tcpdump`, etc.).

It turned out that this part of our plan was harder than we thought at first. Configuring and installing GUI toolkits, with all their dependencies took more time than expected. And even though there are several GUI builders available, it is not easy to create the graphical interface we had in mind. Another difficulty was that because of other problems we encountered, as described in this section, getting the basics to work also took more time than expected. Furthermore, because creating a prototype was not our first priority and there was limited time available to us to produce a proof-of-concept implementation we decided not to implement the GUI.

However, we implemented our application with a clear object model and also applied the *model-view-controller* design pattern. This results in a strict separation of the user interface and other software components. This way, adding a graphical interface at a later time should not be too difficult.

## 6 Description of the prototype

This section describes the implementation details of our prototype software. The software is written in Python and responsible for parsing configuration files and starting, stopping and configuring UML instances and `uml_switch` processes.

### 6.1 Configuration files

Configurations are specified in an XML file. The XML language for configurations is defined in a DTD, which is also used to validate the configuration before the DOM tree is constructed, if the external program `xmllint` is available.

An XML configuration consists of three sections, `defaults`, `devices` and `connections`. The first section, `defaults` is optional and can specify paths to the Linux kernel and the root file system that should be used by default. The next section, `devices` specifies the devices that make up the virtual network. There are four kinds of devices: `switch`, `router`, `hub` or `host`, where each can have zero or more network interfaces, defined with `interface` elements. In the last section, `connections`, the connections between the different interfaces of the devices are defined. These can be seen as a network cable between the two given interfaces.

A minimal sample configuration is shown in figure 5. For more details, see the extended example in appendix A and the DTD in appendix B.

### 6.2 Software structure

Our software consists of a number of components and objects. An UML<sup>5</sup> class diagram of the most important classes is shown in figure 6. The “VNE” prefix we use in our class definition stands for “Virtual Network Experiments”.

#### 6.2.1 VNEConfig

The software maintains a single `VNEConfig` object, containing a model of the virtual network. The `VNEConfig` object contains a list of devices and a list of connections between endpoints. An

---

<sup>5</sup>in this case, UML means *Unified Modeling Language* instead of *User Mode Linux*

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "dtd-VNEControlCfg.dtd">

<config>
  <devices>
    <host name="host-A">
      <interface name="eth0">
        <ipv4 netmask="255.255.255.0" broadcast="192.0.2.255" addr="192.0.2.1"/>
      </interface>
    </host>
    <host name="host-B">
      <interface name="eth0">
        <ipv4 netmask="255.255.255.0" broadcast="192.0.2.255" addr="192.0.2.2"/>
      </interface>
    </host>
  </devices>
  <connections>
    <connection>
      <device name="host-A" interface="eth0"/>
      <device name="host-B" interface="eth0"/>
    </connection>
  </connections>
</config>

```

Figure 5: a minimal configuration file

endpoint is defined as a single network interface on a single device. `VNEConfig` is able to read an XML configuration file from disk and parse it using a Python module for XML DOM (Document Object Model) parsing. The list of devices contains instances of a subclass of `VNEDevice`, while the connection list contains `VNENetwork` objects.

The `VNEConfig` object examines the DOM of the configuration file and passes DOM fragments to factory functions for devices and connections. These factory functions return an initialized object of the appropriate type, based on the contents of the DOM fragment. For example, if a DOM fragment for a device contains a `switch` element, the factory function for devices creates a `VNESwitch` object.

## 6.2.2 VNEDevice

`VNEDevice` derives from `VNEObject`, a small base class for objects that are able to activate and deactivate external programs as subprocesses. Examples of such external programs are the UML kernel and `uml.switch`. `VNEDevice` implements methods that configure and manipulate UML instances. A device contains a list of `VNEInterface` objects that define network interfaces for an UML instance. A large part of `VNEDevice` consists of methods that communicate with the UML subprocess via a virtual terminal. A `VNEDevice` uses the `pexpect` module to wait for a shell prompt or to send commands to the UML instance. Other functionality includes starting an `XTerm` providing the user with shell access to the UML instance.

Subclasses of `VNEDevice` specialise its behaviour. `VNESwitch`, for example, contains methods that configure the switch's bridge interfaces, while `VNERouter` is meant to be able to activate and configure the *Zebra* routing software on the UML instance. `VNEHub` is special, in that instances of this class do *not* start an UML process. `VNEHub` is described in more detail in section 6.2.4.

## 6.2.3 VNEInterface

`VNEInterface` specifies a network interface in an UML instance. This class contains methods to change the interface parameters and to construct an appropriate `ifconfig` command that is used by `VNEDevice` to configure the network interface in an UML instance. A `VNEInterface` object contains references to both the device it belongs to and the `VNENetwork` object it is connected to.

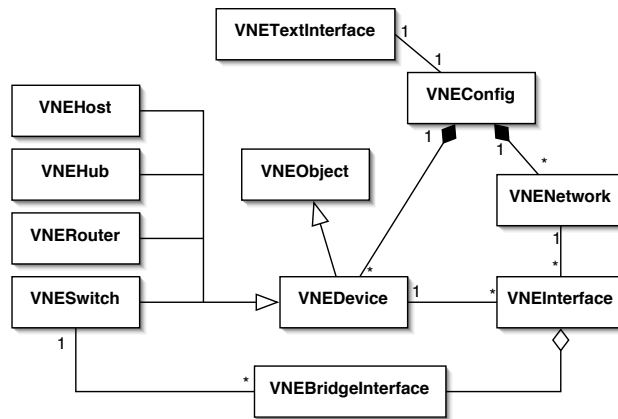


Figure 6: UML class diagram showing objects used in our software

#### 6.2.4 VNENetwork

Instances of `VNENetwork` classes contain references to `VNEInterface` objects. `VNENetwork` objects correspond to connection elements in XML configuration files. There is no limitation on the number of `VNEInterface` objects in a single `VNENetwork`, but the XML configuration DTD limits the number of interfaces in a connection element to two, essentially providing point-to-point links only. Shared networks, consisting of a single broadcast segment, should be constructed using hubs.

`VNENetwork` objects maintain an `uml_switch` subprocess that is responsible for transferring network traffic between endpoints. Usually, `VNENetwork` objects are point-to-point links and for every `VNENetwork`, a separate `uml_switch` is started. However, `VNENetwork` objects connected to a `VNEHub` should all share a `uml_switch` process to provide a single broadcast segment, as illustrated by the connections and hubs in bold in figure 7.

For this reason, `VNENetwork` contains a method that searches for all network elements in its broadcast segment. The `VNENetwork` objects passes its own `uml_switch` process and the path to the socket that `uml_switch` uses to all connected `VNENetwork` objects. During activation of the virtual network, the software starts each network segment in `VNEConfig`'s connection list. When a network segment already contains a reference to another `uml_switch` process and socket, nothing happens. Otherwise, the `uml_switch` process is started and the `assimilateAllConnectedNetworks` method is called.

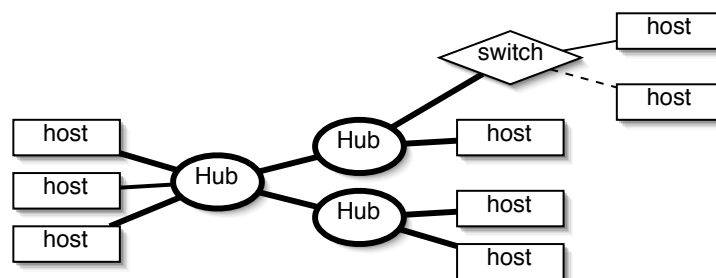


Figure 7: A network with three broadcast segments.

### 6.2.5 VNETextInterface

The interface to our program is at the moment defined by `VNETextInterface`. Thanks to the extensive library of Python it was relatively easy to create a full-fledged command interpreter, with a small set of commands, help functionality and command completion. Our interpreter currently has the following commands:

`start` and `stop` Commands to start/stop the given device names or all devices if no argument is given.

`list` The `list` command can list the different devices, connections, interfaces of a given device or bridge interfaces of a given device, depending on the keyword given after the `list` command.

`config` Command to print the exact commands VNE uses to start and configure the devices and their interfaces.

`xterm` Once devices are started, users must be able to interact with them. Using this command they can start xterms for the devices listed in the arguments.

`shell` While in the interactive command session, it is sometimes useful to be able to execute shell commands, this can be done using the `shell` command, or its shortcut `!`. This command can also be used to start `screen` sessions to the devices.

`help` Online help about the interpreter and its commands is available by using the `help` command.

`quit` When the user is done, he can simply quit by using `CTRL-D` or by typing `quit`.

An example of a session is displayed in figure 8, which uses the configuration as defined earlier in figure 5.

```
$ ./VNE example.xml
-----
Welcome to the text interface of Virtual Network Experiments!
-----
Important commands: help (or ?), shell (or !) and quit
Type 'help' or press <Tab> twice for a complete list of commands.

> list devs
Host    host-A
Host    host-B
> list nets
1. Network (host-A/eth0, host-B/eth0)*
> help start
start <devs> -- start devices <devs>.
> start host-A host-B
Started: host-A (eth0), host-B (eth0).
> list ints host-A
Interface eth0 on host host-A: eth0 (ipv4Address=192.0.2.1, ipv4Netmask=255.255.255.0)
> list nets
1. Network (host-A/eth0, host-B/eth0)
> quit
Thank you, thank you. You were a wonderful audience.
```

Figure 8: Example text interface session

## 7 Possible enhancements

During the four weeks we worked on this project, we were able to implement only the most basic features of our ideal software for virtual network experimentation. However, we tried to design the software with future enhancements in mind.

One element that would really increase the usefulness of our software is a graphical interface that would allow users to design their virtual networks using drag'n'drop to lay out network topologies.

A graphical interface could also show feedback for running networks, such as an indication of traffic flow on network segments and the state of virtual devices.

Another possible enhancement would be to integrate our software with MLN or VNUML in some way. One idea is for the software to be able to import and export configuration files from MLN and VNUML. Another option is using file systems from the MLN and VNUML projects.

Future development should also include items from the list of features we did not plan to implement, discussed in section 5.2.

Furthermore, support for configuring router devices could be enhanced. It would be nice, for example, to be able to include router configurations or references to router configurations on disk in the configuration for our software.

Our software configures UML instances dynamically, by sending shell commands to virtual consoles. It would not be difficult to enhance the software to allow users to change network interface parameters and other configuration options interactively from within the software.

## 8 Conclusions

In this paper we examined several projects for creating virtual networks, based on User Mode Linux. In section 2 we described how UML can be used in this way. In that section we also examined three different projects, MLN, VNUML and SNB UML. The important difference between the three projects are the different ways of configuring the UML instances:

**MLN** modifies configuration files on the disk image containing an UML root file system before it starts UML instances.

**VNUML** sets up TCP/IP networking between host and UML instance with a *tun/tap* tunneling interface and uses a Secure Shell (ssh) connection to configure UML instances.

**SNB UML** uses kernel arguments to set environment variables in the UML instances, which are then used in startup script commands to configure them. This requires a (small) patch to the kernel source, to allow for more kernel arguments.

Because the first two projects require super-user privileges, they are not suitable for our intended purpose; A lab practice where students configure the UML instances and experiment with the network between these instances. SNB UML does not have this requirement and was used successfully in one of our Masters' courses. However, where the first two approaches use separate configuration files, SNB UML's configuration of the hosts is hardcoded in the startup scripts. So, changing the configurations requires changing the script, which is not very user friendly.

In section 3 we explain our idea of an ideal virtual network configuration tool. There we describe the following requirements:

- The tool should not need super-user privileges to install, run the experiment or change the configuration.
- It should have an intuitive configuration model, which follows the abstract model of real-world networks.
- The interface and features of the software should allow the user to easily inspect the internals of all devices involved and also follow the 'KISS'<sup>6</sup> principle, to not distract the user with unnecessary features.
- The user should not be restricted by the software, but should be allowed full access and control over the UML instances and the configuration of the network.

---

<sup>6</sup>“Keep It Simple, Stupid”

In section 3.5 we give a short list of specific tasks the user should be able to do, using an ideal virtual network configuration tool.

Starting with the described requirements we provide a proof of concept design in section 4. The key elements in the design are how the software and the user can interact with the different instances in the simulated network. Another key element is how the simulated hosts are configured and how the user can influence these configurations, both before the simulation is started as well as while the experiment is running. This is followed by a description of the technical requirements for implementing these key elements.

With these key elements in mind, we created a project planning, as described in section 5. There we describe what kind of features and in what order should be implemented in the prototype. We also made a list of specific things we did *not* plan to implement in our prototype. During the implementation phase we also ran into several unexpected difficulties:

- There are several options of communicating with UML instances, however these options were limited by our restriction on using super-user privileges. We chose to use `stdin` and `stdout` of the instances to control them, but even when using *expect* this was more difficult than originally expected.
- Because we have an interactive interpreter, all started processes for the virtual network can be child processes of the interpreter, so that once the interpreter exits, all subprocess are killed automatically. However, if the hosts halt themselves, some process become detached from the parent process and linger after the interpreter exits. It proved very hard to track down where these processes are coming from and we still have not completely tackled this issue.
- Installing software on the file system for use by the UML instances proved to be more difficult than expected. Because of the different location of the software on the host and simulated host file systems, software does not work out of box. To solve this, we linked the software statically, so that it works from any location.
- Creating a graphical user interface proved to be more ambitious than we thought. We moved this part of our project plan to our future plans for the implementation.

The prototype we implemented is described in section 6. There we describe the structure of our configuration file, which uses XML syntax which is defined in a separate DTD. This is followed by a section in which we describe the different elements of the class diagram in figure 6 to explain how our software works and how we structured it. Finally we describe the interactive interpreter and what kind of things the user can do with it.

In conclusion we can say that the main problem with creating virtual networks using UML is the communication and configuration of the simulated devices. MLN, VNUML and SNB UML all use different methods to configure the devices: MLN modifies the filesystem for the simulated host, VNUML uses networking and ssh and SNB UML uses kernel arguments. The approaches used by MLN and SNB UML do not allow for configuration on the fly, VNUML's approach does allow for this, but they do not make use of it.

Our approach uses yet another way of communicating with UML instances, we use the `stdin` and `stdout` of the UML instances. The advantage of this approach is that no super-user privileges are required and it allows for dynamic configuration of the hosts. So our prototype provides the user with an interactive session with our interpreter, from which the user can interact with the simulation. Furthermore, our software is structured in such a way so that the text-based interpreter can easily be replaced by a graphical user interface, as described in our future plans.

Currently our interpreter does not provide for dynamic configuration of the devices, save using a direct shell session with the devices. However, all required technology for providing commands to configure the hosts is already there, but unfortunately we did not have enough time to implement it completely. Once this is implemented, the user can also start configurations, modify them and then save them so he can continue later.

## A Extended example

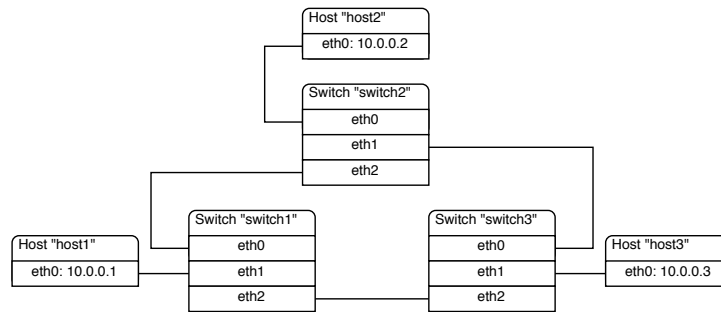


Figure 9: An extended example with three switches and three hosts.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE config SYSTEM "dtd-VNEControlCfg.dtd">

<config>
  <devices>
    <switch name="switch1">
      <interface name="eth0" bridge="br0"/>
      <interface name="eth1" bridge="br0"/>
      <interface name="eth2" bridge="br0"/>
    </switch>

    <switch name="switch2">
      <interface name="eth0" bridge="br0"/>
      <interface name="eth1" bridge="br0"/>
      <interface name="eth2" bridge="br0"/>
    </switch>

    <switch name="switch3">
      <interface name="eth0" bridge="br0"/>
      <interface name="eth1" bridge="br0"/>
      <interface name="eth2" bridge="br0"/>
    </switch>

    <host name="host1">
      <interface name="eth0">
        <ipv4 addr="10.0.0.1" netmask="255.255.255.0"/>
      </interface>
    </host>

    <host name="host2">
      <interface name="eth0">
        <ipv4 addr="10.0.0.2" netmask="255.255.255.0"/>
      </interface>
    </host>

    <host name="host3">
      <interface name="eth0">
        <ipv4 addr="10.0.0.3" netmask="255.255.255.0"/>
      </interface>
    </host>
  </devices>
</config>
```

```
</devices>
<connections>
  <connection>
    <device name="host1" interface="eth0"/>
    <device name="switch1" interface="eth1"/>
  </connection>
  <connection>
    <device name="host2" interface="eth0"/>
    <device name="switch2" interface="eth0"/>
  </connection>
  <connection>
    <device name="host3" interface="eth0"/>
    <device name="switch3" interface="eth1"/>
  </connection>

  <connection>
    <device name="switch1" interface="eth0"/>
    <device name="switch2" interface="eth2"/>
  </connection>
  <connection>
    <device name="switch1" interface="eth2"/>
    <device name="switch3" interface="eth2"/>
  </connection>
  <connection>
    <device name="switch2" interface="eth1"/>
    <device name="switch3" interface="eth0"/>
  </connection>
</connections>
</config>
```

## B VNE Document Type Definition

```
<!ELEMENT config      (defaults?,devices, connections?)>

<!ELEMENT defaults    (kernel?,filesystem?)>
<!ELEMENT kernel      EMPTY>
<!ELEMENT filesystem  EMPTY>

<!ELEMENT devices     (host|hub|switch|router)*>
<!ELEMENT host        (kernel?,filesystem?,gateway?,interface*)>
<!ELEMENT hub         (kernel?,filesystem?,gateway?,interface*)>
<!ELEMENT switch      (kernel?,filesystem?,gateway?,interface*)>
<!ELEMENT router      (kernel?,filesystem?,gateway?,interface*)>
<!ELEMENT gateway     EMPTY>
<!ELEMENT interface   (ipv4?, ipv6?)>
<!ELEMENT ipv4        EMPTY>
<!ELEMENT ipv6        EMPTY>

<!ELEMENT connections (connection*)>
<!ELEMENT connection (device*)>
<!ELEMENT device      EMPTY>

<!ATTLIST kernel      name      CDATA #REQUIRED
                    options    CDATA #IMPLIED>
<!ATTLIST filesystem  name      CDATA #REQUIRED>
<!ATTLIST host        name      ID    #REQUIRED>
<!ATTLIST hub         name      ID    #REQUIRED>
<!ATTLIST switch     name      ID    #REQUIRED>
<!ATTLIST router     name      ID    #REQUIRED>
<!ATTLIST gateway    ipv4      CDATA #IMPLIED
                    ipv6      CDATA #IMPLIED>
<!ATTLIST interface  name      CDATA #REQUIRED
                    macaddr   CDATA #IMPLIED
                    bridge    CDATA #IMPLIED>
<!ATTLIST ipv4       addr      CDATA #REQUIRED
                    netmask   CDATA #REQUIRED
                    broadcast  CDATA #REQUIRED>
<!ATTLIST ipv6       addr      CDATA #REQUIRED>
<!ATTLIST device     name      IDREF #REQUIRED
                    interface CDATA #REQUIRED>
```

## References

- [1] Andersen, Erik. *BusyBox: The Swiss Army Knife of Embedded Linux*  
URL: <http://www.busybox.net/>
- [2] AT&T Labs. *Graphviz - open source graph drawing software*  
URL: <http://www.research.att.com/sw/tools/graphviz/>
- [3] Beck, Kent and Fowler, Martin. *Planning Extreme Programming*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201710919
- [4] Begnum, Kyrre, Koymans, Karst, Krap, Arjen, and Sechrest, John. *Using Virtual Machines in System Administration Education*. In *Proceedings of the 4<sup>th</sup> international SANE conference*. NLUUG, the UNIX User Group - The Netherlands, 2004
- [5] Begnum, Kyrre and Sechrest, John. *My Linux Network*  
URL: <http://mln.sourceforge.net/>
- [6] Cisco Systems. *Cisco IOS Software configuration*  
URL: <http://www.cisco.com/univercd/cc/td/doc/product/software/index.htm>
- [7] Dike, Jeff. *User Mode Linux Community Site*  
URL: <http://usermodelinux.org/>
- [8] Dike, Jeff. *User Mode Linux Kernel Home Page*  
URL: <http://user-mode-linux.sourceforge.net/>
- [9] Free Software Foundation. *GNU Zebra: free software that manages TCP/IP based routing protocols*  
URL: <http://www.zebra.org/>
- [10] Galán, Fermín. *VNUML Language Reference*  
URL: <http://jungla.dit.upm.es/~vnuml/doc/1.3/reference/index.html>
- [11] Galán, Fermín and Fernández, David. *Virtual Network User Mode Linux*  
URL: <http://jungla.dit.upm.es/~vnuml/>
- [12] Gooch, Richard. *Devfs (Device File System) FAQ*  
URL: <http://www.atnf.csiro.au/people/rgooch/linux/docs/devfs.html>
- [13] Krap, Arjen. *Setting up a virtual network Laboratory with User-Mode Linux*. Technical report, Masters programme on System and Network Administration, University of Amsterdam, 2004  
URL: <http://www.os3.nl/~arjen/snb/asp/asp-report.pdf>
- [14] Libes, Don. *Expect Home Page*  
URL: <http://expect.nist.gov/>
- [15] Libes, Don. *expect: Curing Those Uncontrollable Fits of Interaction*. In *Proceedings of the Summer 1990 USENIX Conference*. 1990
- [16] Spurrier, Noah. *Pure Python expect-like module*  
URL: <http://pexpect.sourceforge.net>
- [17] *Systeem- en Netwerkbeheer Homepage*, 2004  
URL: <http://www.os3.nl/>
- [18] Torvalds, Linus. *Linux Home Page*  
URL: <http://www.linux.org/>