Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Emulation of Computer Networks with Netkit

Massimo Rimondini[1]

RT-DIA-113-2007 **January 2007**

(1) Dipartimento di Informatica e Automazione,
Università di Roma Tre,
Rome, Italy.
`rimondin@dia.uniroma3.it`

Work partially supported by European Commission – Fet Open project DELIS – Dynamically Evolving Large Scale Information Systems – Contract no 001907 and by MIUR under Project "MAINSTREAM: Algoritmi per strutture informative di grandi dimensioni e data streams", MIUR Programmi di ricerca di Rilevante Interesse Nazionale.

ABSTRACT

An emulator is a software or hardware environment that is capable of closely reproducing the functionalities of a real world system. Emulators, especially if implemented in software, are very useful for performing experiments that might compromise the operation of the target system or simply when the real system itself is not available. This is true in particular for computer networks, where configurations of network devices often need to be tested before being deployed. In this paper we present a survey of network emulation systems and we describe in detail Netkit, a lightweight emulator based on User-Mode Linux. Besides being an effective instrument to support teaching of computer networks, Netkit has also proven itself to be helpful in testing the configuration of large scale real world networks. Netkit provides tools for a straightforward setup of complex network scenarios that can be easily distributed and comes with a set of ready to use experiences that permit to immediately experiment with specific case studies. Netkit also fully installs and runs in user space and provides users with a familiar environment consisting of a Debian based Linux box and well known routing software and networking tools.

1 Introduction

Testing configurations is a common need both for network administrators and for computer scientists interested in networking. The first can take advantage of a testing phase for checking that a particular configuration works as expected before deploying it, while the latter can exploit test results in order to validate theoretical models with practical experimentation. Ideally, testing should take place under the very same conditions in which the configuration is to be eventually deployed.

However, this often means injecting artificially generated, potentially harmful traffic into a live network, which may cause damage to it. An effective alternative to live testing consists in implementing the network configuration of interest inside a safe, isolated software environment which closely reproduces the real target setting. Such environments are usually available in two flavours:

- **Simulation environments** allow the user to predict the outcome of running a set of network devices on a complex network by using an internal model that is specific to the simulator. With the network as an input and the outcome (possibly a network state) as an output, simulators do not necessarily reproduce the same sequence of events that would take place in the real system, but rather apply an internal set of transformation routines that brings the network to a final state that is as close as possible to the one the real system would evolve to.

  As this approach can be optimized in performance, the simulated network can typically scale well in size. The drawback is that the simulated devices may have limited functionalities and their behaviour may not closely resemble that of real world devices.

  Some of the most representative network simulators are C-BGP [8, 10, 9], a routing solver that computes the outcome of the decision process of the BGP protocol, NS-2 [89], a very comprehensive object oriented network simulator, and SSFNet [95], which provides open source implementations of several network protocols and network devices.

- **Emulation environments** aim at closely reproducing the features and behaviour of real world devices. For this reason, they often consist of a software or hardware platform that allows to run the same pieces of software that would be used on real devices. Differently from simulation systems, in an emulator the network being tested undergoes the same packet exchanges and state changes that would occur in real world.

Since emulation environments are the focus of this paper, examples of them are presented in the following Sections.

The real advantage of the emulation approach comes out when the emulator itself is a software piece, as this allows much higher flexibility in carrying out network tests. Since emulation makes use of real routing software, every aspect of the network can be influenced and monitored like it could be in a real network. While this ensures very high accuracy, the computational resources needed to run an emulated device are typically higher than those available in the device itself. Hence, the performance of an emulated device is, in general, lower than that of the real one, and this often poses limits on the scalability of the size of the emulated network.

This paper introduces to the fundamental concepts of emulation environments and presents Netkit [11], a lightweight product based on open source software that enables experimenting networking on a personal computer. After describing the architecture and some internals of Netkit, we show how it is possible to easily prepare and run complex network experiences, which can also be easily packaged and redistributed. An example case study is also presented to show the capabilities of Netkit.

2 An Overview of Emulation Environments

An *emulator* is a software or hardware environment that aims at closely reproducing the features and behaviour of a real device or system, usually making it possible to use software products unaltered inside a system they were not designed for. A *virtual machine* is a running software that creates an abstraction layer between a hardware/software platform and other software (possibly an operating system). Applications running inside a virtual machine interface with this abstraction layer instead of the physical hardware. Therefore, a virtual machine may also implement virtual devices (disks, network interfaces, etc.) that are different from those available on the platform the emulator runs on.

	Scale	Emulation type	Emulated device	License	Link
Bochs	Small	Full emulation	IA-32 x86	GPL	[75]
Cooperative Linux	Medium	Kernel port	Linux box	Free	[76]
CrossOver	Medium	Compatibility layer	Windows APIs	Commercial	[104]
DosBox	Small	Full emulation	x86 DOS box	GPL	[78]
DosEMU	Small	Compatibility layer	DOS box	GPL	[79]
Einar	Large	Router emulation	Quagga based router	GPL	[80]
Emulab	Large	Testbed	—	Project based	[74]
FAUmachine	Medium	User-mode kernel	x86 box	GPL	[15]
IMUNES	Medium	Virtual image	Linux box	Free	[18]
KVM	Medium	Native virtualization	x86 box	GPL	[85]
MLN	Medium	Paravirtualization/User-mode kernel	Linux box	Free	[86]
Modelnet	Large	Testbed	Linux box	GPL/BSD	[16]
NCTUns	Medium	Simulation	Host/Router	Free	[70]
Netkit	Medium	User-mode kernel	Linux box	GPL	[11]
Parallels	Medium	Full virtualization	x86 box	Commercial	[58]
PearPC	Small	Full emulation	PowerPC box	GPL	[90]
Planetlab	Large	Overlay network	—	Membership	[60]
Plex86	Medium	User-mode kernel	Linux box	Free	[92]
Q	Small	Full virtualization	x86 box	Free	[93]
QEMU	Small	Full virtualization	x86 box	GPL	[19]
SVISTA	Small	Full virtualization	x86 box	Commercial	[96]
UML	Medium	User-mode kernel	Linux box	GPL	[97]
UMLMON	Medium	User-mode kernel	Linux box	GPL	[24]
vBET	Medium	User-mode kernel	Linux box	N/A	[17]
VDE	Large	Overlay network	—	GPL	[62]
VINI	Large	User-mode kernel	Linux box	Membership	[100]
VirtualBox	Small	Full virtualization	x86 box	GPL/Commercial	[28]
Virtual PC	Small	Full virtualization	x86 box	Free	[54]
Virtuozzo	Small	Full virtualization	x86 box	Commercial	[101]
VMware	Small	Full virtualization	x86 box	Commercial	[102]
VNUML	Medium	User-mode kernel	Linux box	GPL	[72]
Win4Lin	Medium	Full virtualization	x86 box	Commercial	[103]
Wine	Medium	Compatibility layer	Windows APIs	GLPL	[104]
Xen	Medium	Paravirtualization	x86 box	GPL/Commercial	[55]

Table 1: A taxonomy of emulation-related products.

There are a lot of emulation products available, which can be distinguished on the basis of the emulation technique adopted, of the type of device they emulate, and of the license with which they are distributed. This Section attempts to provide a taxonomy of a number of emulators, with the twofold purpose of outlining the landscape of available alternatives and of pointing out those products that are more network oriented.

Table 1 shows a fairly comprehensive list of the existing emulation products. The proposed classification coordinates have the following meaning:

- **License** specifies the license agreement under which the emulator is being distributed.
- The **emulated device** specifies the machine whose features are reproduced by the emulator. In most cases it is a standard PC which, by running suitable pieces of software, can be turned to a router, switch, or other network device.
- The **scale** ranges from small to large, and describes the number of virtual entities (hosts, routers, switches, or whatever else) each emulator allows to start on the platform it is intended to be used on (typically a standard workstation). A *small* scale emulator is usually conceived for running very few instances of virtual machines, as their resource requirements may be rather high. A *large* scale emulator is usually designed to run on a distributed architecture (possibly a cluster of geographically distributed workstations connected by an overlay network), which allows to perform arbitrarily wide-ranging experiments. *Medium* scale emulators typically allow to run around tens of virtual machines on a single workstation.
- **Emulation type** specifies the technique used for virtualizing resources.
  Full virtualization indicates that the emulated entity is a full-fledged system consisting of system

buses, CPU, memory, disk, and other devices, and that optimization techniques are used to improve the performance of the emulation. Among these techniques *dynamic translation* is often used, which consists in translating blocks of binary code being executed in the emulated machine into instructions for the real host, and in caching the translated pieces of code for future execution.

Full emulation adopts a complementary approach, in which every instruction of the emulated CPU is implemented as an entire function or procedure in the emulator. While this ensures compatibility and makes it easier to debug the code of the emulator, performance is severely impacted by this technique, and a high-end workstation is usually needed to achieve nearly native speed emulation.

Native virtualization takes place whenever the emulator takes advantage of extensions available on recent families of processors (Intel®VT [29], AMD-V™ [3]), which allow a more effective distribution of resources between the emulated machine and the host it runs on, thus achieving much better performance.

In a *paravirtualization* environment each virtual entity is presented a special hardware abstraction layer. Virtual machines must run slightly modified versions of the standard operating systems, so that system calls are submitted to this abstraction layer (called *hypervisor* or *virtual machine monitor*) instead of the host operating system.

Emulation environments using a *virtual image* are based on virtualization extensions to the FreeBSD kernel [81] which allow to maintain multiple independent network stack instances within a single operating system kernel.

Some products exploit a *user-mode kernel*, often called *User-Mode Linux* (UML) [97, 39, 38], which is a slightly modified version of a standard Linux kernel that is compiled to run as a userspace process. An instance of User-Mode Linux uses its own filesystem image and allocates a subset of the memory available on the hosting machine. A UML kernel can start and schedule processes on its own and has its own virtual memory manager as well as every other kernel subsystem. Device drivers are suitably rewritten so that UML can provide some support for virtualized hardware (disks, network interfaces, consoles). Differently from other emulation systems, User-Mode Linux does not directly interface with the hardware but achieves virtualization based on the system calls interface provided by the standard kernel.

A *compatibility layer* is not a real emulator, but rather a porting of a system call interface and a set of libraries on a different platform. In this case there is no hardware being emulated. However, programs that solely perform standard system calls and invoke library functions can run unmodified because they are presented the same software interface as that of their native system.

Testbeds and *overlay networks* are large scale environments consisting of tens or hundreds of servers that can be geographically distributed (in which case they are often connected so as to form an overlay network). Such large scale architectures can be typically accessed by research institutions to perform controlled experiments on a realistic setting.

Not all the products listed in Table 1 are tailored for performing networking experiments. Some of them are general purpose, often small scale emulators that allow to run entire operating systems. We are more interested in the environments that provide configuration capabilities and tools to ease building and running emulated networks.

VDE and Xen are components that are often used in emulation environments. Virtual Distributed Ethernet (VDE) [62, 64, 63] is a set of tools to create and manage a virtual network that can be spawned over a set of arbitrarily distributed physical computers. VDE can be used to handle tunnels that separate actual connectivity from the topology established by VDE virtual cables, thus providing with the ability to transparently distribute local network experiences on different nodes. Xen [55, 27, 26, 25, 43, 59] is a *virtual machine monitor* or *hypervisor*, that is, a software that provides an abstraction of hardware resources. It consists of a kernel patch and some userspace tools. Because it directly interfaces with hardware resources, using a Xen enabled kernel allows to run virtual machines (also called *domains*) with very high performance levels. Operating systems cannot run unmodified inside Xen domains, as the system calls they make must be mapped to software traps to the hypervisor. Yet, there are plans to extend Xen to support virtualization technologies found on recent processors (Intel®VT [29] and, in the future, AMD-V™ [3]), which would provide with the ability to run unmodified software.

Einar [80] is a live CD router emulator based on the Xen hypervisor [55] and developed by a team of 5 students at the Stockholm KTH Royal Institute of Technology. It provides a wizard interface to quickly set up and run experiences consisting of several switches, routers, and computers, possibly running some services. Each virtual machine is implemented as a Xen domain, and it possible to configure bandwidth and delay limits on network interfaces. The traffic exchanged among the virtual machines can be investigated by using the Ethereal network sniffer. As it runs from a live CD, the product requires no installation.

Emulab, Modelnet, PlanetLab, and VINI share similar purposes and architecture. They all are large scale testbeds which allow to run experiments involving a set of geographically distributed nodes. The University of Utah makes available to researchers Emulab [74, 34, 34], a cluster of networked high-end workstations that can be configured to run fully customizable tests. Faculty or senior staff members coordinating an experiment have fine grained control on the software running on workstation nodes, from the operating system to user level applications, and get exclusive administrator level access during the time interval they are assigned. This allows to configure each node to act as an arbitrary network device. Modelnet [16, 61, 33, 4] is a software emulator developed at the University of California, San Diego, that allows to build distributed network experiences running on an Internet-like environment. PlanetLab [60, 2, 48, 47, 71] is an overlay testbed conceived to allow researchers to perform controlled experiments that would benefit from running on geographically distributed nodes. The PlanetLab network consists of more than 350 nodes at the time this paper is being written, and is managed by a consortium of academic, industrial, and government institutions. Experiments are initiated on the basis of an Acceptable Use Policy [91] that defines the rules for determining whether they are considered as appropriate. VINI [100, 6] is a virtual network infrastructure running on about 15 PlanetLab nodes. It exploits User-Mode Linux [97] to provide a realistic test environment while keeping the network conditions under control.

IMUNES [18] is a software emulator that takes advantage of an extension of the FreeBSD kernel providing with the ability of running multiple independent network stack instances on a single operating system kernel [53, 81]. A graphical topology editor allows to quickly prepare experiments consisting of hubs, switches, routers, and hosts. Intermediate systems can arbitrarily use Quagga [94] or XORP [30] as routing software. The product is available in the form of a live CD, which takes off the burden of having to deploy a modified kernel on the host machine.

MLN [86] is a Perl script that can be used to quickly create a network consisting of Xen [55] and User-Mode Linux [97] based virtual machines. By means of an ad-hoc language (MLN), it is possible to specify the configuration of each device participating in the virtual network, including the emulation type (Xen or UML), the filesystem being used, the amount of available memory, the startup commands to be executed, and, of course, the topology of the network.

NCTUns [70, 65, 69, 66, 1, 67] is rather a simulator than an emulator. After defining a topology with the integrated graphical editor, it is possible to launch traffic generators and sinks at a given time during the simulation. The resulting traffic flows can then be investigated either by looking at sniffer traces or by displaying an easy to read animation of the edges on the graphical topological map.

Netkit, UMLMON, and VNUML are all medium scale software emulators that utilize a User-Mode Linux [97] kernel to run the emulated network experiences. Netkit is extensively described in the following of this paper. UMLMON [24] is a solution for managing a set of User-Mode Linux virtual machines. It comes in the form of a daemon written in Objective Caml and acts as a supervisor that allows to configure and run UML instances by using a Remote Procedure Call interface. Interaction between the end user and the daemon is possible by means of a command line or a web based interface. VNUML [72, 22, 21] consists of an XML based language and an interpreter that can be used to describe and run an emulated network of User-Mode Linux virtual machines. VNUML developers also propose an interesting set of ready to use examples that implement some typical networking case studies.

3 The Architecture of Netkit

In the rest of this paper we describe in detail Netkit [11], a lightweight network emulator developed by people at the Roma Tre University [99]. Netkit consists of two main components: a set of scripts and tools that ease setting up complex network scenarios consisting of several virtual devices, and a set of ready-to-use virtual experiences that can be used to analyze typical case studies. Netkit is conceived for easy installation and usage and does not require administrative privileges for either one of these operations.

As time goes on, old bugs are fixed and new features are introduced into Netkit. This paper refers to the version of Netkit available at the time of writing, which is 2.4, filesystem F2.2, kernel K2.2 (User-Mode Linux kernel 2.6.11.7).

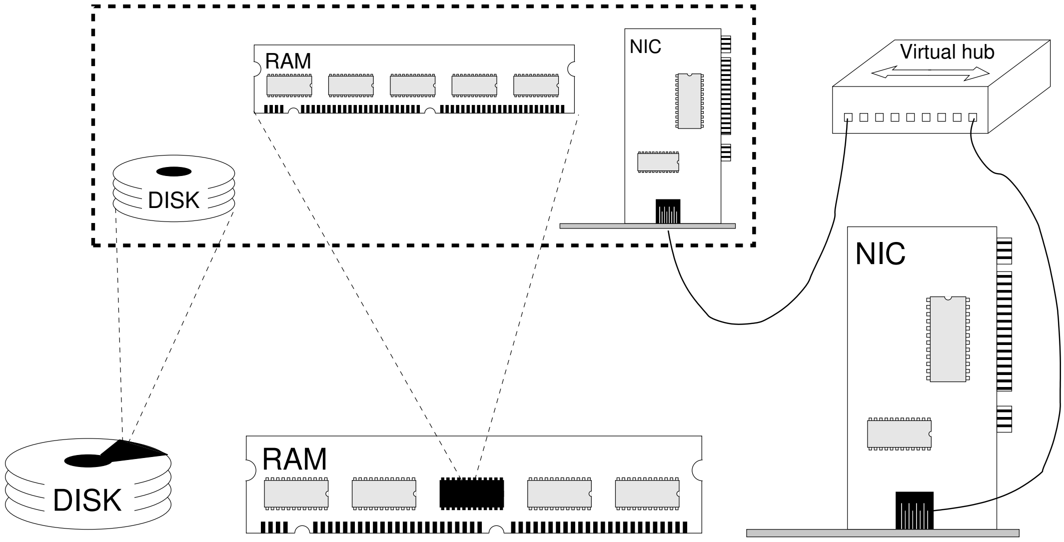

Figure 1: Resources of a Netkit virtual machine. Entities enclosed in the dashed box are virtualized, while all the others correspond to devices or processes on the host machine.

The emulation approach adopted in Netkit is simple. Basically, every device that makes up a network is implemented inside Netkit as a virtual machine. Each virtual machine owns a set of virtual resources that are mapped to portions of the corresponding resources on the host. Figure 1 shows how this mapping takes place. Virtual machines are equipped with a disk, whose raw image is a file in the host machine; they have their own memory region, whose size can be established upon startup; and they can be configured with an arbitrary number of virtual network interfaces which are connected to a virtual hub. The dashed box in Figure 1 encloses virtualized resources, while everything outside of that box is a device or process on the host. It is possible to observe that the virtual hub lives on the real host: indeed, it is a special process that replicates packets on all the connected interfaces. If requested, the virtual hub can be connected to a network interface of the host machine, so that a virtual machine can reach an external network such as the Internet.

To make a clear distinction between an emulated device and the real machine it is running on, in the following we label Netkit virtual machines and the software they run as *guest*, and we refer to the real machine and the software it runs as *host*.

Netkit virtual machines can be networked by means of the virtual hub. In practice, a hub works as a sort of cable that connects multiple guests. Notice that a guest must always connect to a virtual hub, and cannot be directly connected to another guest. Figure 2 shows an example of how a typical Netkit network looks like. `VM1`, `VM2`, and `VM3` are virtual machines and they make up a simple topology consisting of two collision domains: one including `VM1` and `VM2` and the other including `VM2` and `VM3`. In this topology `VM2` can operate as a virtual switch that forwards traffic only on the ports connected to the LAN segment containing the destination host. For this purpose, `VM2` has been equipped with two network interfaces.

The example in Figure 2 also introduces another principle of the Netkit emulation approach: the functionalities of an emulated device depend on the software installed in the virtual machine that implements it. For instance, `VM2` can be turned to a switch by running standard ethernet bridge administration software such as `brctl` or to a router by properly setting up the IP routing tables.

Netkit virtual machines are based on the User-Mode Linux kernel [97], which is described in more detail in Section 3.1. Starting a virtual machine means starting a UML instance, which often requires dealing with somewhat complex command line arguments. For this reason Netkit supports straightforward configuration and management of virtual machines by means of an intuitive interface consisting of several scripts.

Figure 3 synthetically describes the architecture of Netkit, consisting of the blocks inside the dashed box. Each block represents a piece of software that runs on top of the ones beneath and is controlled by the tools on its left. Virtual machines are UML instances that directly run on the host kernel and are managed by a set of commands whose names are `l`-prefixed (*ltools*) and `v`-prefixed (*vtools*) utilities.

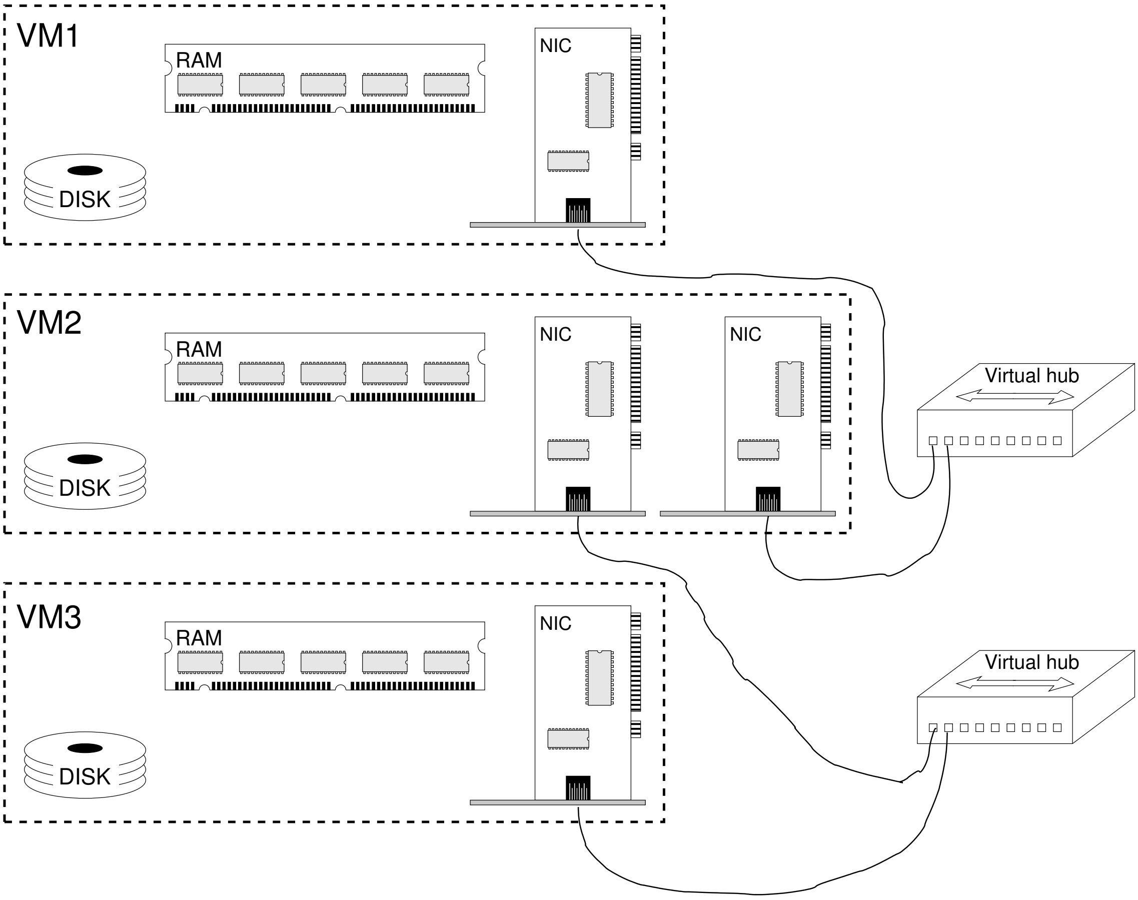

Figure 2: Sample Netkit emulated network.

Virtual machines can run routing software, as well as other tools. Virtual hubs are implemented as processes running on the host kernel.

Blocks with boldface text represent the only components that are presented to the end user. It is possible to observe that there is no need for him to directly interact with UML kernels or virtual hubs. On the other hand, both the `ltools` and the `vtools` are accessible to the user. The difference between the two is that the ltools provide a higher level interface to virtual machines and therefore exploit the `vtools` in order to implement their functionalities.

Because of its architecture, taking advantage of Netkit can be beneficial in several contexts. The most common application is within didactics: as it gives students the feeling of what is actually going on inside a network, Netkit has been successfully used to teach networking protocols within University level degree courses. This is further supported by the fact that Netkit comes with a set of ready to use network experiences that implement case studies spanning from routing protocols (TCP, RIP, BGP, etc.) to application level services (DNS, e-mail, etc.) [12]. Another valuable application consists in preparing virtual networks that act as sandboxes for safe debugging purposes: this spares the need to perform potentially harmful tests on a live network. The ability to carry out experiments in a safe environment, combined with the easiness of maintaining a one-to-one reproduction of a real network, also comes handy for testing a configuration before deploying it. For example, Netkit has been used for helping in determining the OSPF weights to be assigned within portions of the GARR Italian Academic Research Network [82].

With respect to the other available network emulators, Netkit has the advantage of being lightweight and easy to install and run. It is possible to launch a complex network experience consisting of 200 virtual machines in about 30 minutes on a typical workstation (Pentium IV 3.2GHz 2MB cache, 2GB RAM)[1]. Netkit fully runs in userspace and has no dependencies on other software pieces. It natively uses state of the art routing software [94, 44] and comes with most of the commonly used networking

[1]This test has been performed on a SKAS enabled host kernel. For more details about this, see Section 3.1

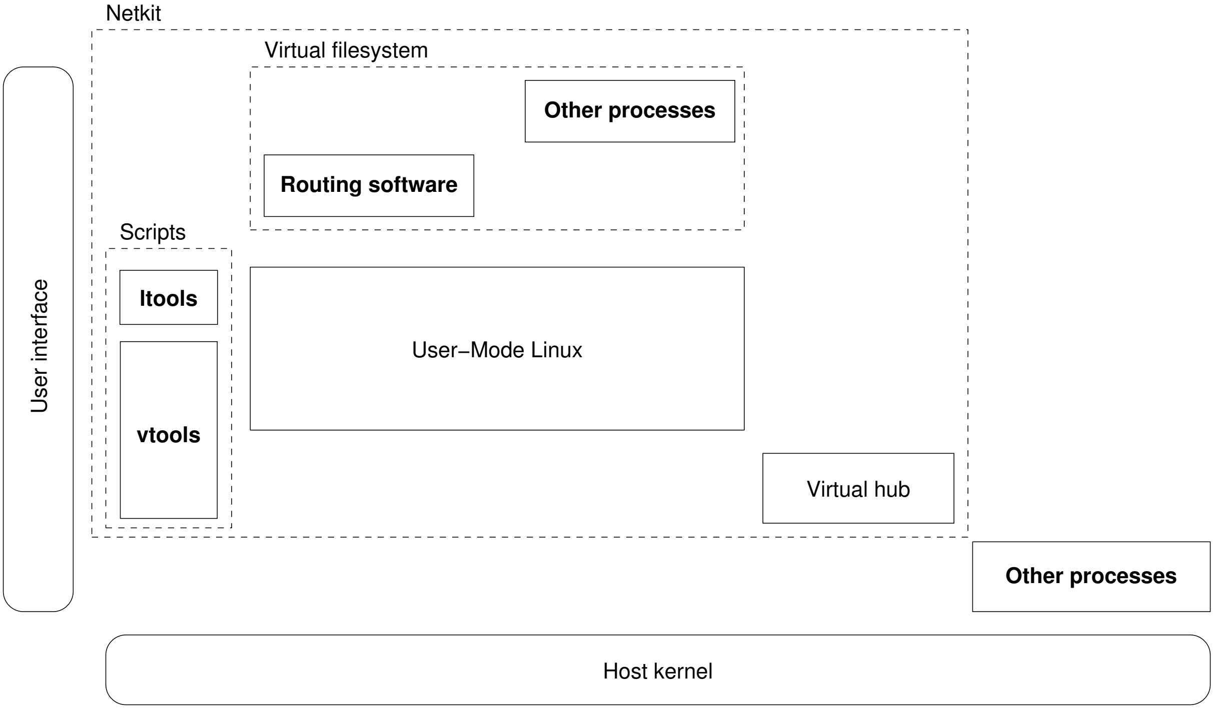

Figure 3: Architecture of Netkit. Each software runs on top of the ones beneath and is controlled by the tools on its left.

tools (sniffers, servers, etc.). Should it be needed, it is possible to also install additional packets inside virtual machines. As detailed in Section 4, emulated network experiences can be easily redistributed in a preconfigured, ready to use, and automatically starting up package without the need to carry any heavy filesystem image.

On the other side, Netkit only works on top of Linux hosts and provides emulated Linux virtual machines. While preliminary versions for the Windows OS are being released, and a live CD [68] is also available to make it easier to play around with Netkit, the second restriction is inherent in Netkit and cannot therefore be overcome. Also, at present Netkit provides a somewhat fixed implementation of the link-level network layer which does not provide support for the emulation of physical link properties (latency, packet loss, reordering, etc.) or the behaviour of wireless mobile stations. However, there are plans to implement some of these features in future releases. Moreover, being an emulator, Netkit is not suited for reproducing real time performance of network protocols and services.

There are a couple of projects that propose XML based languages for describing networks and can therefore be exploited to describe and implement emulated scenarios. NetML [13, 31] is a project carried on by the same group that maintains Netkit, and for this reason it is strongly integrated with it. Besides an XML Schema, NetML consists of a set of tools that allow to turn a vendor independent description of a network into configuration statements for specific routing software (Cisco, Juniper, Zebra) or, optionally, a package implementing a Netkit network. NDL [73, 42, 41, 40] is a language based on the Resource Description Framework (RDF) and a set of tools that ease producing network descriptions. While NetML is conceived to describe small as well as broad networks, NDL is more suitable for large scenarios, as it also takes into account the geographical location of network nodes. NDL tools also allow to automatically sketch visual representations of a network.

3.1 User-Mode Linux: a Kernel in the Kernel

In an emulation environment virtual machines have nearly the same characteristics of a real host, including their own kernel. Netkit exploits User-Mode Linux as kernel for the virtual machines. User-Mode Linux is widely used by kernel hackers, who are doing filesystem and memory management development and debugging, as well as by hardware developers, who are prototyping new types of device in software. It also meets the interests of the security community, as it fits well the creation of jails and honeypots, and is often employed by the hosting industry to run virtual servers [49, 46]. The fundamentals of UML are illustrated by its designer Jeff Dike in some publications [39, 35] which the following description is based on.

User-Mode Linux [97, 39, 20, 45, 38, 7, 37, 36] is a port of the standard Linux kernel [84] which is designed to run as a userspace process. Being a kernel in itself, UML comes with its own kernel subsystems, including scheduler, memory manager, filesystem, network, and devices. In this sense an instance of UML provides a virtualized environment in which everything (processes, memory, filesystem, etc.) is controlled by itself instead of the host kernel. In practice, UML appears as a userspace process on the hosting machine and acts as a kernel for its own processes.

Virtual machine settings can be passed to UML via a command line interface. While this is an effective way of specifying configuration parameters, it is often the case that users interested in just emulating networks are not willing to deal with complex kernel invocation commands. For this reason, Netkit provides a higher level user interface to UML which is described in more detail in Section 3.4.

There is an important difference in the approach adopted by full emulation/virtualization products and that adopted in Netkit. Full emulation products usually attain virtualization by directly interfacing with the host hardware, and provide an abstraction layer implementing an architecture that may also be different from the one of the host they are running on. In the case of UML, virtualization takes place within the host kernel rather than at the hardware layer. In other words, UML provides simulated hardware constructed on the basis of services provided by the host kernel. Essentially, UML is a port of the Linux kernel to the Linux system call interface rather than to a specific hardware interface. User space code simply runs natively (no emulation), while processes in kernel mode see a special environment which limits access to host resources. This makes the emulation faster and more responsive, and is the reason why Netkit is considered a lightweight emulator. The drawback of this approach is that it only allows to run emulated Linux boxes.

Basically, what UML does is to provide virtualization for system calls. Normally, a process in the virtual machine doing a system call is trapped directly into the host kernel. Instead, UML intercepts system calls so that they are run in virtual kernel mode. This is implemented by using threads and the `ptrace` system call. `ptrace` allows one process to control the execution of another, as well as change its core image and be notified when it receives a signal. In UML a special thread called *tracing thread* makes use of `ptrace` to intercept and dispatch to the virtual kernel the system calls issued by processes in the virtual machine. Other traps caused by pieces of hardware (clock, I/O devices, etc.) are implemented in UML by using signals.

For each process or thread running inside the UML virtual machine, the tracing thread creates a new process on the host. Figure 4 shows a process listing on the host while `tcpdump` is running inside a virtual machine. It can be easily seen that `tcpdump` appears as a process entry (pid 13868) on the host.

After performing preliminary initializations, UML mounts a virtual disk device provided by the user and boots the Linux distribution it finds inside it. Virtual disks are managed by a User-mode Block Device (UBD) driver, which uses a standard file (called *backing file*) on the host filesystem as storage area. The backing file can be filled with Linux software in a way that is very similar to what would happen on a real host. Indeed, the backing file can be made available as a loopback device on the host machine by using the `losetup` command. Once done, it can be initialized by using `mkfs` and populated by using tools such as `debootstrap`. In order to support hassle free usage, Netkit comes with a ready to use filesystem containing most state of the art networking tools. For more details, see Section 3.3.

UML supports the emulation of an arbitrary number of network interfaces. This aspect is discussed in more detail in Section 3.2.

Being a lightweight environment, it is possible to implement complex setups consisting of several instances of UML based virtual machines. Since each virtual machine writes on its own filesystem, this potentially implies using several backing files, which size is often not negligible (hundreds of megabytes). However, the ubd block driver is able to support sharing of a filesystem among different virtual machines. This is achieved by writing changes to the backing file into a different file, a technique that is also known as *Copy-On-Write* (COW). Therefore, a typical setup of a complex emulated scenario consists of a single large backing file containing a model filesystem and several small (typically less than 10 MB) COW files that store the changes to the model filesystem. Thus, filesystem information for a virtual machine can only be reconstructed based on both its own COW file and the backing file. Each COW file can only be used together with the backing file it was created from. However, by using the UML utility `uml_moo` [98, 57], it is also possible to merge the two and get a standalone backing file that contains all the filesystem information for that virtual machine. COW files are implemented as *sparse files*. A sparse

Host console

```
max@foo:~$ ps -o pid,command
  PID COMMAND
12959 /bin/bash
12987 /bin/sh /usr/local/netkit/netkit2/bin/vstart pc1
12990 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(tracing thread)]
12991 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
12998 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13000 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13002 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13004 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13006 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13007 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13009 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13011 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13013 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13015 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13016 xterm -T Virtual Console #0 (pc1) -e port-helper -uml-socket
                                                     /tmp/xterm-pipexvvhYG
13018 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [(kernel thread)]
13021 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [/sbin/init]
13696 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [/sbin/klogd]
13792 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [/sbin/syslogd]
13834 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [-bash]
13836 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [-bash]
13868 /usr/local/netkit/netkit2/kernel/netkit-kernel (pc1) [tcpdump]
13895 ps -o pid,command
```

Figure 4: Processes and threads in a UML virtual machine are implemented as processes on the host.

Host console

```
max@foo:~$ ls -ls --block-size=1 pc1.disk
634880 -rw-r--r-- 1 max max 630358016 2007-01-19 18:24 pc1.disk
max@foo:~$ du --block-size=1 pc1.disk
634880  pc1.disk
```

Figure 5: Size and actual disk space consumption of sparse files. |pc1.disk— takes about 620 KB of disk space, but is apparently as large as 602 MB.

file is one which efficiently uses the filesystem by allocating space only when data is actually written to the file. Figure 5 shows an example of the space allocation strategy for a sparse file. Apparently, the size of `pc1.disk` is 602 MB (output of `ls`), but the actual disk space consumption for this file is of 620 KB (first column of the output of `ls` and output of `du`). This means that `pc1.disk` has a lot of contiguous empty regions which are not actually written to disk: this is often the case for COW files, as the filesystem of an only just started virtual machine only slightly strays from the initial status.

Once a UML virtual machine has started, it is possible to interact with it by means of a terminal interface. The interface can be attached to arbitrary file descriptors, `pty` pseudo terminal devices, or to a user space application like `xterm` or a server like `telnetd`. This is possible thanks to the `port-helper` tool, that is part of the UML utilities [98, 57]. Essentially, `port-helper` uses a UNIX socket to pass to the UML kernel a file descriptor obtained from an application (like `xterm`) and used to perform input/output. This allows UML to directly interface with the application.

UML instances can be managed from the host machine by means of the `uml_mconsole` utility [98, 57], which allows to halt or reboot a virtual machine, to send it magic SysRq sequences, to configure emulated devices on the fly, and to pause or continue its execution. `uml_mconsole` is used in the Netkit scripts to manage running virtual machines.

In the past UML used to be available in the form of a patch to a standard Linux kernel [97, 84]. Most recent kernels already include user-mode code, thus a UML kernel can be simply built by specifying `um` as target architecture while compiling a vanilla kernel. Starting from 2002, Paolo Giarrusso [56] maintains a set of patches called SKAS that can be optionally applied to the host kernel to change the way UML behaves. The patches have beneficial effects in terms of both security and performance. Essentially, when using the standard technique with tracing thread, the address space of each virtual machine contains the image of the UML kernel, and can have write access to it. Since UML runs on the host, this also means

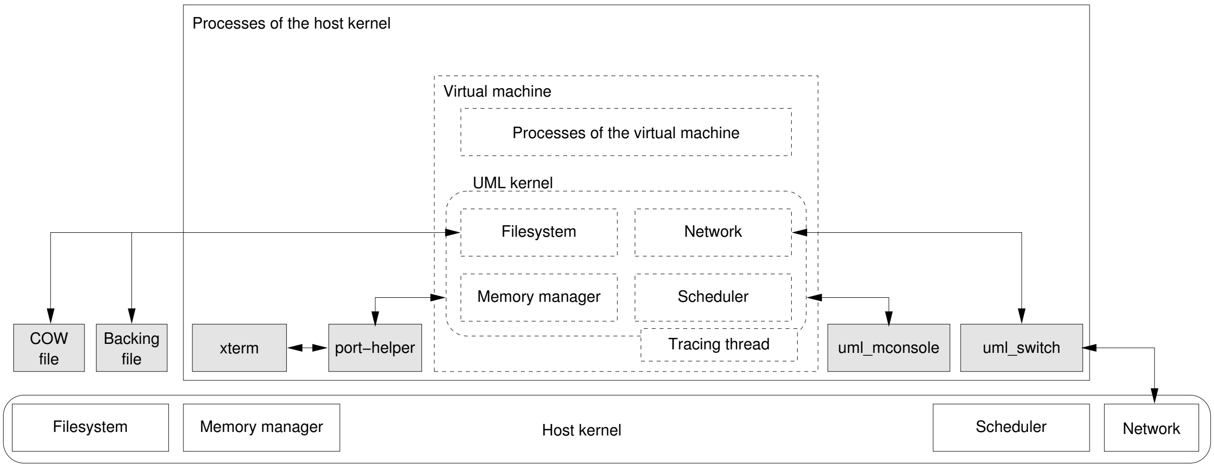

Figure 6: Relationships between User-Mode Linux and the other components of the host system. Boxes represent kernel entities (subsystems, interfaces, or processes). Dashed boxes represent virtualized resources, while gray filled ones are instantiations of kernel entities (processes or files).

gaining access to the host machine, which affects security. On the other hand, UML takes advantage of signals in order to implement system call dispatching, and this affects performance. A SKAS (*Separate Kernel Address Space*) enabled host kernel allows to overcome both these issues. As for security, SKAS makes UML run in a different address space from the processes it controls. In turn, this limits the number of signals to deliver, as fewer context switches are needed within this rearranged setting, and this improves performance. Benchmarks have shown that starting 100 virtual machines with the default configuration (8 MB, minimal services running) on a Pentium IV 3.2GHz with 2MB of cache and 2GB of RAM takes about 30 minutes with a plain 2.6.16.16 host kernel. The time required to start the same set of virtual machines on the same workstation running a 2.6.16.16 SKAS enabled kernel dramatically reduces to about 10 minutes, and starting 100 more virtual machines in this setting takes less than 30 more minutes, thus having a scenario consisting of 200 devices running on a single workstation within less than 40 minutes.[2] Even though the SKAS patch improves security and boosts performance, Netkit still scales rather well without the need to replace the host kernel.

Figure 6 shows the relationships between the user-mode kernel and the other components of the host system. Boxes represent kernel entities (subsystems, interfaces, or processes). Dashed boxes represent virtualized resources, while gray filled ones are instantiations of kernel entities (processes or files). A virtual machine is a set of processes on the host. The virtual machine kernel has its own subsystems that are independent from those of the host kernel. The virtual machine filesystem is implemented in terms of files on the host. Other processes running on the host provide a user interface to the virtual machine (`xterm`) or are used for its management (`uml_mconsole`). The `uml_switch` is a user space utility to support networking whose functionalities are explained in Section 3.2.

3.2 Networking Support in Netkit

Communication between different virtual machines is made possible in Netkit by emulated networking. Figure 7 describes graphically how this emulation takes place.

UML allows to configure virtual machines with an arbitrary number of network interfaces. By using appropriate UML command line arguments, these interfaces can be attached to a `uml_switch` process [98, 57] running on the host, which simulates the behaviour of a network switch or hub. In this way, different virtual machines attached to the same switch can exchange data with each other.

More specifically, UML virtual network interfaces can be attached to a UNIX socket. In turn, a `uml_switch` can be attached to the same socket and forward packets among the virtual machines that

[2]In order not to overload the host with several virtual machines booting up at the same time, Netkit adopts an optimization technique that ensures that no more than a fixed number of virtual machines are booting simultaneously at any given time. For the case of these performance tests the number has been always set to 3. For more details about this feature, see Section 3.4.

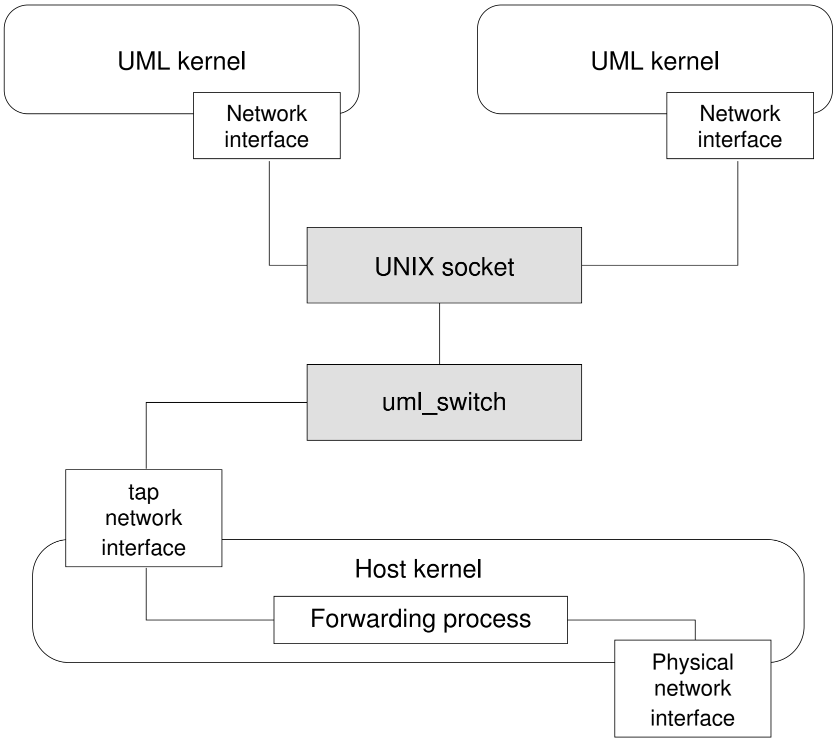

Figure 7: This diagram shows how Netkit virtual machines are networked, possibly with a connection to an external network.

are connected to that socket. Netkit scripts take care of automatically starting `uml_switch` processes according to user's needs.

From the point of view of the network stack, Netkit provides implementations of the ISO-OSI layers as described in Figure 8.

- The physical layer is implemented by a set if `uml_switch` processes running on the host. They are configured to behave as hubs, and packets are forwarded to interfaces of other virtual machines by using UNIX sockets. For this reason, in this paper we also refer to the `uml_switch` as *virtual hub*. At present this mechanism does not provide support for simulating delay, packet loss, and reordering.
- The data link layer supports the Ethernet protocol, but collisions cannot happen because the `uml_switch` avoids them. Unless differently specified, emulated network interfaces are assigned an automatically generated MAC address.
- The network layer supports both IPv4 and IPv6 by means of kernel code and user space utilities.
- What happens on upper layers is up to the specific software being run inside virtual machines. For example, running a web server would introduce support to HTTP.

Notice that layers from data-link through transport are (at least partly) implemented inside the UML kernel. Therefore, changing the kernel results in making new implementations and features available.

The configuration of network interfaces in Netkit is straightforward thanks to the existence of scripts set up for the purpose. In order to slightly abstract from the technicalities of how networking is implemented, Netkit presents `uml_switch`es as virtual collision domains. Each virtual network interface must be attached to a collision domain that is identified by an arbitrary name. Therefore, connecting virtual machines is simply a matter of attaching their interfaces on the same collision domain.

For example, the following command line starts up a virtual machine named `foo` with a single network interface attached to collision domain `COLL-DOM-A`.

```
vstart foo --eth0=COLL-DOM-A
```

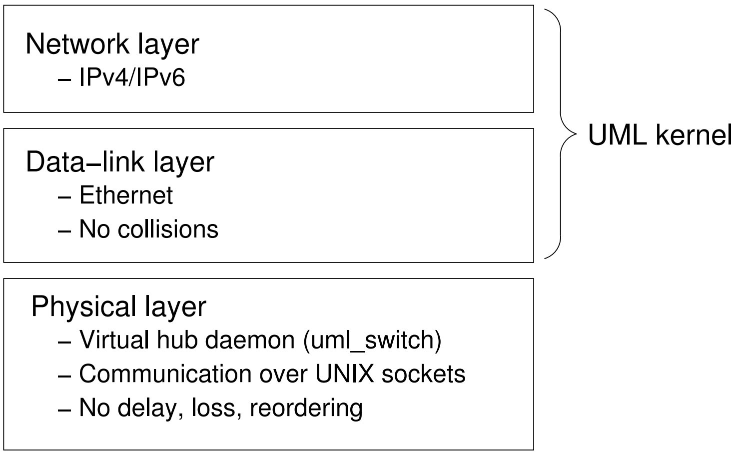

Figure 8: The emulated network stack in Netkit.

IP addresses for the interfaces can then be configured in the usual way by using `ifconfig` inside virtual machines.

Configuring a virtual machine to reach an external network requires setting up a `TAP` device. `TUN/TAP` is a device driver inside the Linux kernel that sets up a special network interface connected to a user space application instead of a physical medium. Packets sent to a `TUN/TAP` interface are written to the underlying application, while information generated by the application appears as it were received from the interface. The only difference between `TUN` and `TAP` is that the former expects the application to handle IP packets, while the latter allows to directly transfer Ethernet frames.

As shown in Figure 7, the setup for an external connection involves configuring a `TAP` device on the host and using an `uml_switch` as application that sends and receives data from it. Since the `uml_switch` can still be attached to virtual machines via UNIX sockets, this allows to connect a virtual network segment to a real network one. Depending on the specific configuration set up on the host, the two segments can be bridged or routed. In order to prevent IP addresses used for experiments from leaking on a real network, Netkit enables routing instead of bridging, and uses Masquerading to hide addresses assigned to emulated network interfaces. Masquerading is a variant of port-based NAT in which outgoing packets are mangled to appear as originated from the interface they are sent through; incoming response packets are forwarded to the interface that had actually initiated the network transfer. Masquerading can be enabled in Linux by taking advantage of the netfilter framework [87] inside the kernel. This can be achieved by using `iptables` to configure an appropriate rule with target `MASQUERADE` in the `POSTROUTING` chain of the `nat` table.

Netkit scripts take care of automatically setting interfaces connected to an external network, so that configuring them is as easy as configuring standard interfaces. The following is a command line that configures virtual machine `foo` to have a `TAP` (LAN or Internet connected) interface:

```
vstart foo --eth0=tap,10.0.0.1,10.0.0.2
```

The two IP addresses are automatically assigned to the `TAP` interface on the host and to the emulated interface inside the guest, respectively. These addresses are also used to set a proper routing table inside the guest. Both addresses must be provided so that, once the virtual machine has started up, it is already able to reach the external network. Notice that the configuration of a `TAP` interface requires administrative privileges, therefore Netkit will ask for the root password when the above command is executed.

3.3 A Filesystem of Networking Tools

Netkit is meant to be an environment that makes it easy and quick to set up complex network experiences. For this reason, virtual machines are equipped with a filesystem that contains most well known servers and tools for network analysis.

The Netkit filesystem contains a full-fledged Debian GNU/Linux distribution [77] which has been suitably tuned to run inside UML. This provides users with a familiar environment and allows to quickly grab new software pieces that might be needed for specific experimentation purposes, or to flexibly upgrade currently installed tools. Actually, installing or upgrading software pieces is simply a matter of starting a virtual machine that is connected to the Internet (see Section 3.2) and running the `apt` tools [14].

Among the network services available in Netkit there are the `apache` web server, the `bind` Domain Name Server, a DHCP server, the `exim4` Mail Transport Agent, an FTP server, the netfilter configuration tool `iptables`, an NFS server, and a Samba server. Utilities include `traceroute`, `ping`, `arping`, `netcat`, `tcpdump`. For a complete list of installed packages, see [88].

Netkit makes use of the Copy-On-Write technique described in Section 3.1 to save disk space when multiple virtual machines are run. In this way there is only one shared backing file containing the model filesystem downloaded with Netkit, and each virtual machine stores changes to that filesystem inside its own COW file. In this way, not only the filesystem is shared among all the virtual machines, so that they see completely aligned tools and services, but it is also possible to easily revert to the original filesystem contents by simply deleting a COW file in case things mess up. There is also an option of the Netkit commands that allows to write changes to the model filesystem permanently, which comes handy for example when installing new packages which are supposed to be available inside all virtual machines.

In order to facilitate the transfer of files between the host and a virtual machine, the special directory `/hosthome` inside a virtual machine makes the user's home directory on the host always accessible. The same technique of providing special directories pointing to the host filesystem is also used by Netkit to automatically transfer settings for preconfigured network experiences. For more details about this, see Section 4.

3.3.1 The Zebra Routing Software Suite

A special mention is due to the routing software installed in the Netkit filesystem. In order to experiment with routing protocols, Netkit comes with an installed release of the Zebra routing software [44]. Zebra is a suite of daemons that provide support for several routing protocols, including RIP [23], OSPF [32], and BGP [105]. Routing protocols take care of spreading information about available destinations on a network in order to automatically update the routing tables of each device.

Figure 9 describes an abstraction of the architecture of Zebra and of the way in which it injects information in the kernel routing table. Each Zebra routing daemon manages a specific routing protocol, has its own configuration file, and writes to its own log. They listen on different TCP ports, so that messages of a particular routing protocol can be sent to the appropriate daemon. For each routing protocol, a Routing Information Base (RIB) and a Forwarding Information Base (FIB) are maintained. The RIB is the set of all destinations known to that protocol, together with the path to reach them and some additional reachability information. The FIB contains, for each possible destination on the network, only the alternative that is considered the best one to reach it. Zebra in itself is a routing daemon: it receives information from the FIBs of the other daemons and, for each destination, selects the best alternative among those made available by different routing protocols. Zebra's best routes are finally injected into the routing table of the kernel, which is used to actually forward packets.

All the routing daemons, including Zebra, can be contacted via `telnet` on a dedicated TCP port to check the status of routing protocols and perform "on the fly" configuration. The daemons provide a command line interface which closely resembles that of Cisco routers. Most of the commands available on real devices can be used, but each daemon only provides those commands that are specifically oriented to the routing protocol it manages. For example, `ripd` does not provide the `show ip bgp` command, and `bgpd` must be contacted in order to be able to issue it. However, the Zebra suite also comes with `vtysh`, an integrated shell that provides a unique interface to all the daemons. Figure 10 shows a sample session of usage of the `bgpd` prompt inside a virtual machine.

Unfortunately, Zebra development is somewhat slow. For this reason, and also in order to create a community that does not rely on a centralized model, the Quagga project has been started [94]. Quagga is essentially a fork of Zebra in which proposals from a community of users are usually discussed and more quickly acknowledged. As a result, Quagga provides bug fixes and functionalities that are missing in Zebra, sometimes at the expense of stability (both stable and unstable releases of Quagga are available). At present, Netkit does not provide the Quagga routing suite. However, it can be easily installed in case it is needed, and there are plans to include it in future releases.

An alternative routing software suite is XORP [30, 51, 52, 50], an extensible routing platform that overcomes some of the limitations of Zebra/Quagga. XORP leverages on a framework from abstracting

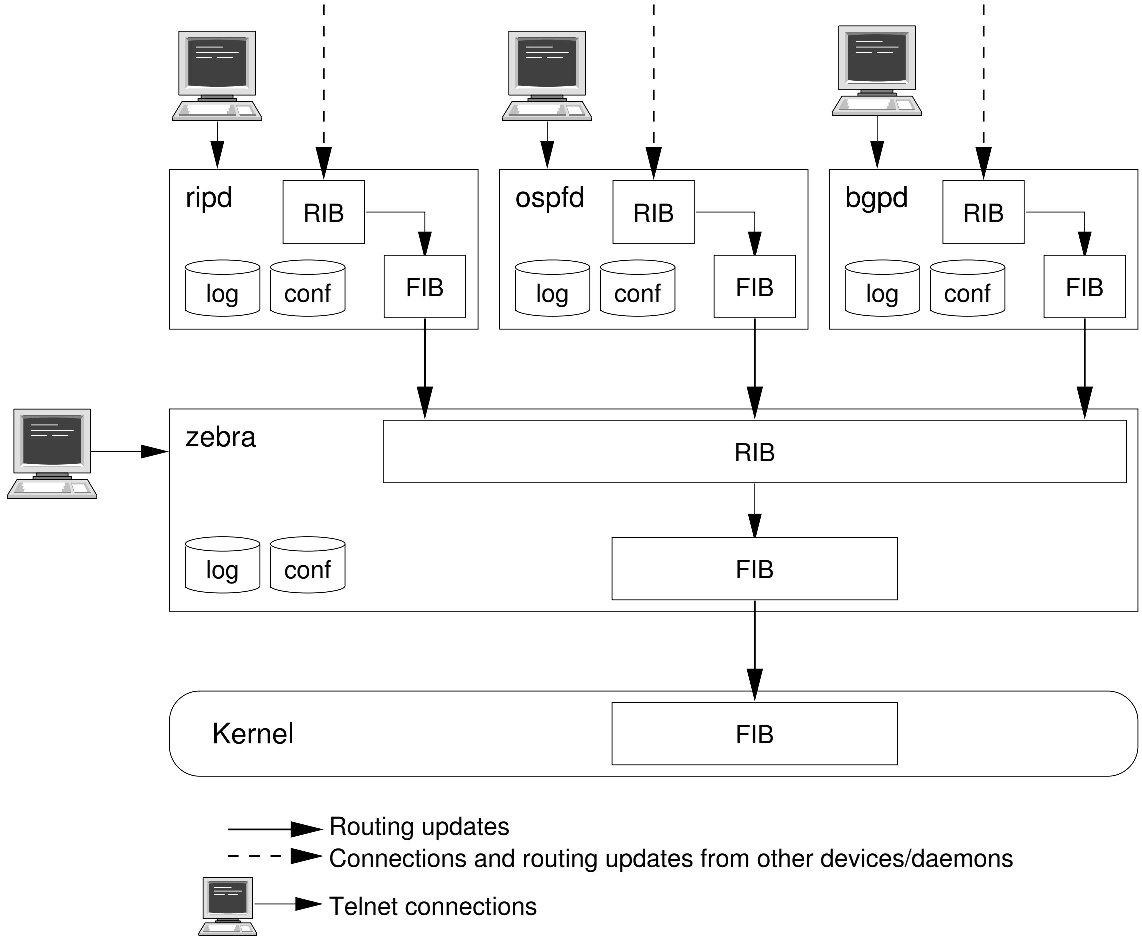

Figure 9: An abstraction of the architecture of the Zebra routing software.

and decoupling routing policies from protocols [5]. This allows a greater degree of flexibility than in products like Zebra/Quagga, where policy constructions are hardwired: for example, there is no way to express a matching condition like $metric < 3$ in Zebra/Quagga. Another difference is that the Zebra/Quagga interface supports Cisco IOS-like commands, while XORP's interface is quite similar to that of Juniper JunOS platforms. XORP intends to be an attractive alternative both for researchers and for hardware vendors, as stability for mission-critical production use is one of the main goals of the project. The current version of the Netkit filesystem does not include XORP. Yet, it can be installed in case it is needed.

3.4 User Interface

One of the most interesting features of Netkit is the user interface, which includes tools for quickly and easily setting up complex and redistributable network experiences. Netkit virtual machines can be managed by means of a set of commands consisting of `v`-prefixed (*ltools*) and `l`-prefixed (*vtools*). All these commands must be run on the host machine and are implemented as shell scripts.

`vtools` are conceived to configure and manage single virtual machines, and they provide the following functionalities:

`vstart` can be used to configure and start a new virtual machine, identified by an arbitrary name. It allows to set parameters such as the amount of available memory, the kernel and filesystem to be used, the type of terminal device to make available, as well as the network interfaces and the collision domains they are attached to. Netkit's default settings usually fit most of the needs, so that starting a virtual network device often simply consists in specifying the network interfaces it should be equipped with.

Virtual machine console

```
router:~# telnet localhost bgpd
Trying 127.0.0.1...
Connected to router.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password: zebra
bgpd> enable
bgpd# configure terminal
bgpd(config)# router bgp 1
bgpd(config-router)# network 10.0.0.0/8
bgpd(config-router)# neighbor 192.168.0.1 remote-as 2
bgpd(config-router)# end
bgpd# disable
bgpd> show ip bgp
BGP table version is 0, local router ID is 0.0.0.0
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 10.0.0.0         0.0.0.0                  0         32768 i

Total number of prefixes 1
bgpd> exit
```

Figure 10: Session of usage of the `bgpd` daemon. Available commands closely resemble those of Cisco routers.

For example, the following command starts a new virtual machine named `pc1`, with no network interfaces.

```
vstart pc1
```

If Netkit is properly installed, the above command prints some informative messages on the host terminal and boots `pc1` inside a new terminal window. Figure 11 shows the booting phase of `pc1`. It is possible to notice that boot time messages are exactly those that would be observed on a standard Debian system, and that at the end the user is presented a shell prompt as root in the virtual machine.

The following commands start two virtual machines, `pc2` and `pc3`, with two network interfaces each. Notice that `pc2`'s `eth0` and `pc3`'s `eth0` are attached to the same collision domain `COLL-DOM-A`, therefore the two virtual machines will be able to communicate with each other.

```
vstart pc2 --eth0=COLL-DOM-A --eth1=COLL-DOM-B
vstart pc3 --eth0=COLL-DOM-A --eth1=COLL-DOM-C
```

Once they have finished booting, `pc2` and `pc3`'s network interfaces can be configured by using `ifconfig` as shown in Figure 12, so that the two machines can reach each other. Also notice that informational messages on the host terminal show that the necessary `uml_switch`es have automatically been started without any need for user intervention.

`vconfig` can be used to attach a network interface "on the fly" to a running virtual machine. This is useful to alter the configuration of an already running scenario or just to avoid having to reboot some machine because one of its interfaces has been forgotten.

The syntax of this command is analogous to that of `vstart`, so that the following command adds to `pc2` an interface `eth2` attached to collision domain `COLL-DOM-D`:

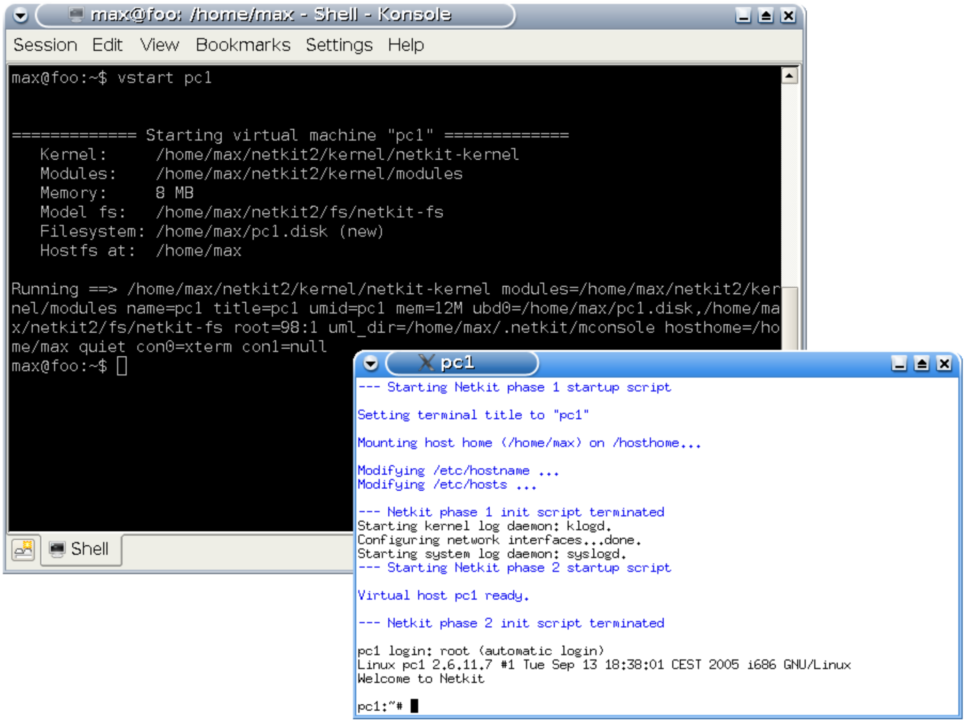

Figure 11: A Netkit virtual machine starting up. The window with black background is a terminal on the host, while the other one is the virtual machine terminal.

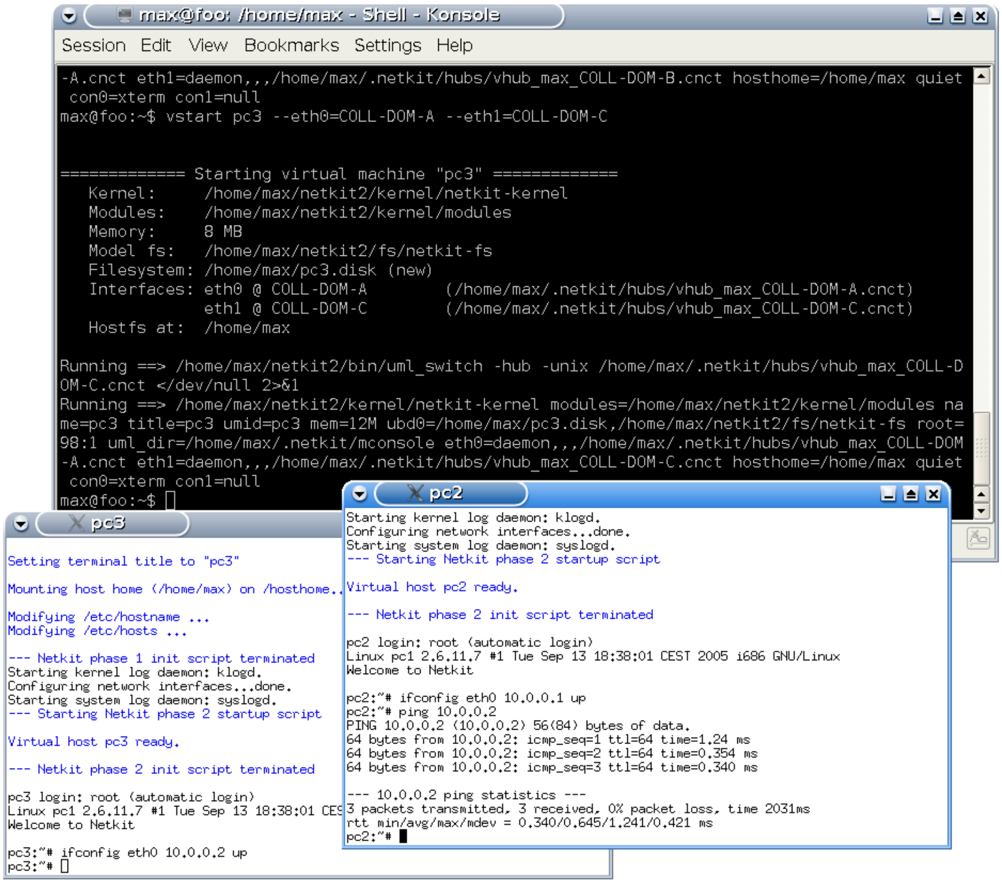

Figure 12: A simple Netkit experiment consisting of two directly connected virtual machines.

```
vconfig pc2 --eth2=COLL-DOM-D
```

`vlist` is a command that provides information about currently running virtual machines. If invoked with no arguments, it produces a list of virtual machines, including the user that has started them, the PID of the tracing thread, the amount of consumed memory, and the list of network interfaces with the collision domains they are attached to (see, e.g., Figure 13). If a virtual machine name is provided as argument, `vlist` provides detailed information about it, as shown in Figure 14.

`vhalt` is exactly the equivalent of running `halt` inside a virtual machine. It gracefully stops the machine by running its shutdown scripts, properly unmounting the filesystem and stopping it. `vhalt` is provided on the host as a convenient tool for performing the shutdown of multiple virtual machines within a script. Figure 15 shows the usage of `vhalt` for stopping `pc3`.

`vcrash` provides a method to immediately stop a running virtual machine. Actually, using `vcrash` is equivalent to abruptly unplugging the virtual machine power cord. This is achieved by first asking the UML kernel to stop itself through a special UNIX management socket. The UML utility [57, 98] `uml_mconsole` is used for the purpose. In case this attempt fails, for example because UML is somewhy frozen, the virtual machine processes are automatically killed by `vcrash`. Upon the subsequent boot, a crashed virtual machine will run a filesystem check to recover inconsistencies. This can be avoided, and the filesystem can be reverted to the original contents, by simply removing the COW file.

`vcrash` is often used in networking experiments consisting of several machines, as it is much faster than `vhalt`.

`vclean` is the Netkit "panic button". Should things mess up, some virtual machine be stuck, or tunnel configurations be left on the host, `vclean` helps in getting rid of all this with a single command. `vclean` can be invoked to perform several operations: it can simply remove unused `uml_switches`, kill all running virtual machines owned by a specific user, and remove tunnels connecting to an external network. Figure 16 shows the usage of `vclean` to kill unused virtual hubs.

`vclean` also comes handy when used in combination with `vconfig`. In fact, due to the mechanism by which they are configured, network interfaces attached by `vconfig` do not show up in the output of `vlist` nor the `uml_switch`es they are attached to can be automatically killed when the virtual machine stops: `vclean` must be used for the purpose.

`vtools` can be profitably used for configuring, starting, and managing few virtual machines. In principle, if used within a suitably written shell script, they could also be exploited to automatically start several virtual machines by invoking a single command. However, the setup of a complex experience usually involves lots of configurations not just of the emulated hardware but also of the services that should be available on the emulated network. Also, it may be difficult to translate the network topology in terms of `vstart` command line options. For this reason, Netkit provides higher level `ltools` which allow to easily set up, launch, or shutdown a complex scenario in a straightforward way. The `l` in `ltools` stands for "laboratory" (in the following abbreviated as "lab"), which is the name that is often associated to preconfigured redistributable Netkit scenarios. `ltools` rely on functionalities provided by `vtools` and offer the following interface:

`lstart` is the command that is used to start virtual machines that make up a lab. Actually, starting all of them is as simple as issuing `lstart` alone in a shell on the host. Details about how to prepare a self running scenario are provided in Section 4. Optionally, `lstart` can be used to start only a subset of the virtual machines that are part of the lab.

`ltest` supports the creation of redistributable self testing labs. Basically, the principle behind `ltest` is that each virtual machine is automatically instructed to perform a set of hardwired and user defined dumps of the significant information that contributes to define its status. For example, the dumps can include the contents of a routing table or the results of a `ping`. These dumps can then be collected and saved as a signature of a correctly running emulated network. Once the lab is moved to a different host, checking that it is still properly running is simply a matter of running it in test mode and verifying that the obtained dumps match the signature.

```
Host console
max@foo:~$ vlist
USER             VHOST               PID      SIZE  INTERFACES
max              pc2                 920     13140  eth0 @ COLL-DOM-A,
                                                    eth1 @ COLL-DOM-B
max              pc3                1124     12380  eth0 @ COLL-DOM-A,
                                                    eth1 @ COLL-DOM-C

Total virtual machines:       2    (you),        2    (all users).
Total consumed memory:    25520 KB (you),    25520 KB (all users).
```

Figure 13: Sample output of `vlist`. The command shows that there are two virtual machines running and reports their owner, the PID of the tracing thread, the amount of consumed memory, and a list of network interfaces, together with the collision domains they are attached to.

```
Host console
max@foo:~$ vlist pc3

============= Information for virtual machine "pc3" =============
 --- Accounting information ---
   PID:        1124
   Owner:      max
   Used mem:   12380 KB
 --- Emulation parameters ---
   Kernel:     /home/max/netkit2/kernel/netkit-kernel
   Modules:    /home/max/netkit2/kernel/modules
   Memory:     8 MB
   Model fs:   /home/max/netkit2/fs/netkit-fs
   Filesystem: /home/max/pc3.disk
   Interfaces: eth0 @ COLL-DOM-A  (/home/max/.netkit/hubs/
                                             vhub_max_COLL-DOM-A.cnct)
               eth1 @ COLL-DOM-C  (/home/max/.netkit/hubs/
                                             vhub_max_COLL-DOM-C.cnct)
   Hostfs at:  /home/max
   Console 1:  terminal emulator
   Console 2:  disabled
   Other args: umid=pc3 root=98:1 uml_dir=/home/max/.netkit/mconsole quiet
   Mconsole:   /home/max/.netkit/mconsole/pc3/mconsole
```

Figure 14: Usage of `vlist` to get detailed information about a running virtual machine.

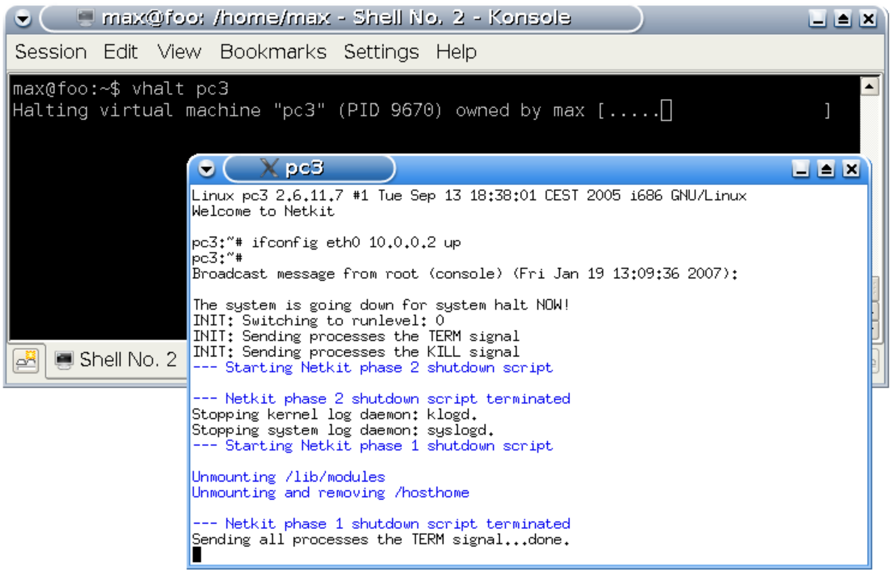

Figure 15: Shutdown of a virtual machine by using `vhalt`.

```
                                    Host console
max@foo:~$ vclean
Killing unusued virtual hubs owned by user max:
   /home/max/.netkit/hubs/vhub_max_COLL-DOM-D.cnct (PID 4168): Killing... done.
```

Figure 16: Usage of `vclean` to kill unused virtual hubs.

```
                                    Host console
max@foo:~/netkit-lab_bgp-6-multi-homed-stub$ linfo

===================== Lab information ==========================
Lab directory: /home/max/netkit-lab_bgp-6-multi-homed-stub
Version:       1.0
Author:        The Netkit Team
Email:         netkit.users@list.dia.uniroma3.it
Web:           <unknown>
Description:
<unknown>

The lab is made up of 4 virtual machines ( as100r1 as200r1 as20r1 as20r2).
=================================================================
```

Figure 17: Sample usage of `linfo` to get basic information about a Netkit lab.

`lhalt` and `lcrash` behave like their counterparts `vhalt` and `vcrash`, but they automatically perform the shutdown or crash operation on all the virtual machines that make up a lab. Optionally, `lhalt` and `lcrash` can affect only a subset of the machines of the lab, for example in case some of them need to be rebooted without having to restart the whole lab.

`linfo` can be used to get basic information about a lab, including descriptive data and a list of the virtual machines that make it up. Figure 17 shows an example of its usage. `linfo` can also be used to get a sketch of the data-link topology of a lab, including hosts, interfaces, and collision domains. This feature requires the Graphviz [83] graph drawing library to be correctly installed. Figure 18 shows an example of topology generated by `linfo`. Ellipses represent hosts and are surrounded by integer values indicating network interfaces. Diamonds represent collision domains (virtual hubs).

`lclean` just performs a cleanup of temporary files left over after running a lab (COW files, logs, etc.). It has nothing to do with its counterpart `vclean`, which instead performs cleanup on running processes.

Both the `vtools` and the `ltools`, as well as the other Netkit components, are fully documented by means of `man` pages that are available with the Netkit distribution.

4 Setting up a Virtual Lab

It has already been pointed out in this paper that setting up an emulated network experiment involves dealing with lots of configuration files and a potentially complex topology. Netkit provides tools to facilitate this task and to support the creation of easily redistributable network scenarios that can be launched with a single command. This Section describes how to prepare such a scenario and manage it with Netkit's `ltools`.

A Netkit *lab* is a set of fully preconfigured virtual machines that can be started or stopped together. Netkit comes with a set of ready to use labs implementing interesting network scenarios involving bridging, transport level congestion control algorithms, routing protocols (with special attention to BGP), and application level services (DNS, E-mail). The labs can be downloaded from [12]. Further labs are periodically added.

Figure 19 summarizes the components of a Netkit lab. Actually, a lab consists of a hierarchy of files and directories having special roles. The following Sections describe these roles in detail and explain how to create a custom Netkit lab.

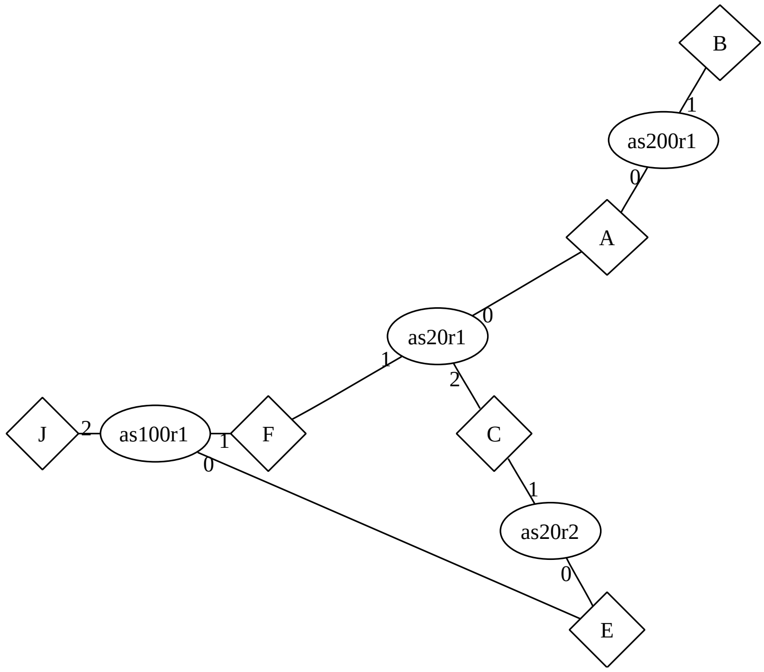

Figure 18: A sketch of the data-link layer topology generated by `linfo`. Ellipses represent hosts; diamonds represent collision domains (virtual hubs); integers around ellipses indicate network interfaces.

4.1 Defining the Topology

Before starting to configure a lab, it is strongly advised to prepare a detailed map of the topology that is about to be implemented. We propose an effective formalism to do this, which is also used in the documentation and examples supplied with Netkit. Figure 20 shows an example of usage of this formalism. As explained in the legend, router symbols represent emulated devices; they do not necessarily have to be routers but, since most of the labs are likely to deal with routing issues, they are represented as such. Circled letters represent collision domains; the letter serves as identifier for the collision domain, and will be used by Netkit to associate a virtual hub to it. Each collision domain is linked to a square box containing the address of the corresponding subnet. Boxes surrounding devices detail information about their network interfaces: the lower half contains the name of the interface and the upper half specifies the last byte of its IP address on the subnet it is attached to. A thick line represents a local network.

In the following we use the topology in Figure 20 as a reference for implementing a sample Netkit lab. Collision domains B and C represent stub LAN segments.

4.2 Implementing the Topology

The topology of a network can be specified in Netkit in two steps.

First, Netkit needs to know the virtual machines that make up the lab. Each directory in a lab represents a virtual machine named as the directory itself. The sole existence of a directory tells Netkit to start a virtual machine with that name. Figure 21 shows an example of lab consisting of two virtual machines, `router1` and `router2`, which is also confirmed by `linfo`.

Second, the link-level topology of the lab must be defined. The file `lab.conf` contains the description of the topology of the lab in terms of connections between different virtual machines. An example of `lab.conf` file is shown in Figure 22. The first part of the file contains optional descriptive information about the lab, which may be useful when the lab is redistributed to other people. The rest of the file contains a mapping between network interfaces and collision domains. For example, the line:

```
router1[0]=A
```

means that interface `eth0` of `router1` is attached to collision domain A.

Directories: each directory specifies the existence of a virtual machine named as the directory itself; files contained in a directory `vm` are automatically copied to the root (`/`) of `vm`'s filesystem; files contained in the `shared` directory are copied to the root (`/`) of every virtual machine.

`lab.conf`: a file describing the link-level topology and other configuration parameters for virtual machines (amount of memory, etc.).

Startup and shutdown scripts: they can be used to apply some settings (e.g., configure IP addresses, start services) during the boot phase; they can be shared or specific to each virtual machine.

`_test`: a directory that contains scripts for dumping the status of virtual machines and that accommodates the dumps themselves.

`lab.dep`: a file describing dependencies in the boot order of virtual machines.

Figure 19: Summary of the components of a Netkit lab.

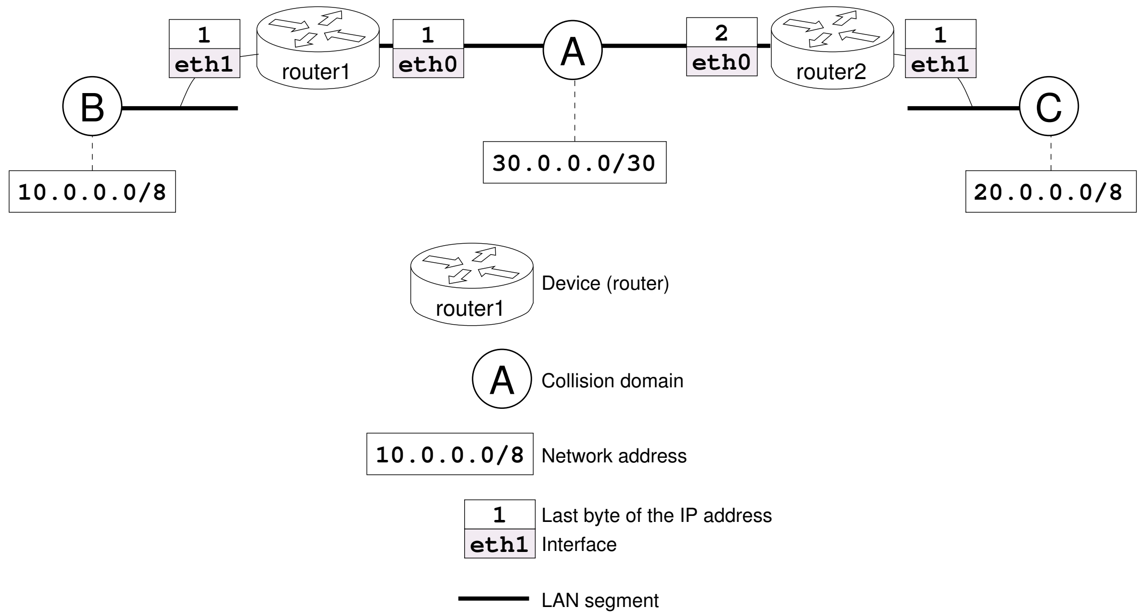

Figure 20: A possible formalism for describing the link-level topology of a lab.

Host console

```
max@foo:~/sample_lab$ ls -l
total 8
drwxr-xr-x 2 max max 4096 2007-01-20 15:49 router1
drwxr-xr-x 2 max max 4096 2007-01-20 15:49 router2
max@foo:~/sample_lab$ linfo

====================== Lab information ==========================
Lab directory: /home/max/sample_lab
Version:       <unknown>
Author:        <unknown>
Email:         <unknown>
Web:           <unknown>
Description:
<unknown>

The lab is made up of 2 virtual machines ( router1 router2).
=================================================================
```

Figure 21: Sample Netkit lab made up of two virtual machines.

```
LAB_DESCRIPTION="Sample lab for testing purposes"
LAB_VERSION="0.1"
LAB_AUTHOR="Massimo Rimondini"
LAB_EMAIL="contact@netkit.org"
LAB_WEB="http://www.netkit.org/"

router1[0]=A
router1[1]=B

router2[0]=A
router2[1]=C
```

Figure 22: A sample `lab.conf` file.

```
ifconfig eth0 30.0.0.1 netmask 255.255.255.252 up
ifconfig eth1 10.0.0.1 netmask 255.0.0.0 up
/etc/init.d/zebra start
```

Figure 23: A startup file for `router1` (`router1.startup`).

Optionally, other configuration parameters can be put inside `lab.conf`. For example, the amount of memory for a virtual machine can be increased to accommodate larger routing tables with a line similar to the following:

```
router[mem]=128
```

Basically, any valid option to `vstart` can be used between square brackets and assigned a value.

4.3 Setting Network Addresses and Startup Time Services

Virtual machines can be instructed to run specific commands on startup and shutdown. During the startup phase, each virtual machine `vm` runs the scripts `shared.startup` and `vm.startup`. These scripts must therefore contain commands that are available in the virtual machine filesystem. Apart from this restriction, almost anything can be put in the startup files. In practice they are generally used to set IP addresses for network interfaces and to start network services. Hence, a typical startup file often looks like the one in Figure 23, which tells `router1` to self configure IP addresses for its network interfaces and to enable the Zebra routing daemon. It comes straightforward that `router2.startup` will look much similar.

In the same way, virtual machines can run user defined shutdown scripts. Upon halting, a virtual machine `vm` first runs `vm.shutdown` and then `shared.shutdown`. Shutdown scripts are only executed if the lab is gracefully stopped with `lhalt`.

4.4 Configuring Services

After specifying which services should be started at boot time, configuration files for them must also be provided. In Netkit this is made possible by a mechanism that automatically makes some of the files that are part of the lab available inside a virtual machine.

In particular, upon starting up a virtual machine Netkit copies all the files inside the directory associated with that machine inside its filesystem. For example, if `vm` is a virtual machine, all the files inside directory `vm/` on the host are copied to the root directory (`/`) of `vm`'s filesystem during its boot phase. If several machines need to access the same files, these can be placed in a directory named `shared`: everything inside `shared` is copied to the root (`/`) of any virtual machine upon its startup. Netkit recognizes `shared` to be a special directory and does not start a virtual machine for it.

This approach of mirroring files from the host has a twofold advantage. It does not add any overhead to the configuration, because the only files to be prepared are those that will be actually fetched by the servers. Moreover, in this way there is no restriction on the number or type of services that can be configured.

```
                                    Host console
max@foo:~/sample_lab$ tree -n
.
|-- lab.conf
|-- router1
|   \-- etc
|       \-- zebra
|           |-- daemons
|           \-- ospfd.conf
|-- router1.startup
|-- router2
|   \-- etc
|       \-- zebra
|           |-- daemons
|           \-- ospfd.conf
\-- router2.startup
```

Figure 24: A possible hierarchy of directories for a lab. Configuration files are placed inside `router1` and `router2`. All the files inside these two directories are copied inside the homonymous virtual machines upon their startup.

```
# This file tells the zebra package
# which daemons to start.
# Entries are in the format: <daemon>=(yes|no|priority)
# where 'yes' is equivalent to infinitely low priority, and
# lower numbers mean higher priority. Read
# /usr/doc/zebra/README.Debian for details.
# Daemons are: bgpd zebra ospfd ospf6d ripd ripngd
zebra=yes
bgpd=no
ospfd=yes
ospf6d=no
ripd=no
ripngd=no
```

Figure 25: A sample `daemons` file, telling Zebra which routing daemons to start.

In the case of our example, we may have a hierarchy of files like the one in Figure 24. It is possible to notice that we have provided two configuration files for each router. `daemons`, shown in Figure 25, tells Zebra which routing daemons should be activated. `ospfd.conf`, shown in Figure 26, enables basic OSPF routing in order to make the internal LAN of `router1` reachable from `router2` and vice versa. Due to the symmetrical nature of the example, the two files `daemons` and `ospfd.conf` are identical on the two routers.

This configuration pattern is common when the Zebra routing daemon is being used: routing protocols to be supported are listed inside `etc/zebra/daemons`, while protocol specific configurations go inside etc/zebra/*`daemon_name`*`.conf` files.

4.5 Tuning Lab Startup

Once the lab has been prepared, it is possible to alter some other settings that affect the way in which it will be started. By default, Netkit starts the virtual machines of a lab one after the other, and waits for the previous one to end the boot phase before starting the next one. This is done in order to prevent overloading of the host machine. An option of `lstart` allows to disable this precautionary measure and to start multiple virtual machines at the same time. A variant of the same option also allows to ensure that at every time instant no more than a fixed number of virtual machines is booting: this is a good compromise between lab startup speed and host side load.

If Netkit is instructed to simultaneously start multiple virtual machines, some of the services may not start up properly. This is usually the case when a virtual machine attempts to initialize a service that depends on another one hosted on a virtual machine that has not been started yet (for example, some mail service relying on DNS). For this reason, the startup order of virtual machines can be influenced in two ways. One possibility is to explicitly tell Netkit the startup order. This can be done by either

```
hostname ospfd
password zebra
!
router ospf
network 30.0.0.0/30 area 0
redistribute connected
```

Figure 26: A sample `ospfd.conf` file.

```
pc2: router pc1
pc3: router pc1
pc4: pc2 pc3
```

Figure 27: A sample `lab.dep` file specifying dependencies on the startup order of virtual machines.

```
#!/bin/sh

# Connectivity tests
ping -c 3 -i 0.3 195.11.14.1 | head -n -3 | sed 's/time=.*//'

sleep 5

# Inspect the arp cache (should be populated)
arp | sort

sleep 5
halt
```

Figure 28: A sample script for performing tests when a lab is launched in test mode.

explicitly listing virtual machines on the command line of `lstart` or by placing an assignment like the following one inside `lab.conf`:

```
machines="first_vm second_vm third_vm fourth_vm fifth_vm"
```

A more flexible way to influence the startup order is to just specify dependencies between virtual machines. This can be done by means of a file `lab.dep` whose syntax is exactly the same used in Makefiles. For example, in Figure 27 line

```
pc2: router pc1
```

means that `router` and `pc1` have no dependencies on each other and can therefore be started simultaneously (provided that the maximum number of booting machines is not exceeded). However, `pc2` can only be started after both `router` and `pc1` have completed their boot phase. Observe that, according to Figure 27, also `pc2` and `pc3` can start up at the same time after `router` and `pc1`. Instead, `pc4` cannot be started until all the other machines are running.

In the example of lab presented in these Sections we do not make use of the `lab.dep` file.

4.6 Testing the Lab

Netkit labs are conceived to facilitate distribution and execution on any other platform equipped with a release of Netkit. If things are configured properly, network experiences should run in the same way and exhibit the same evolution on any platform. However, there may still be particular settings that affect the behaviour of poorly configured labs or cause malfunctions of the emulation environment. For this reason, a lab can be equipped with a special set of scripts that instruct virtual machines to perform a certain number of tests. Once the lab has been found to behave properly, the results of these tests can be collected and constitute a signature of a correctly running lab. When moving it to a different platform,

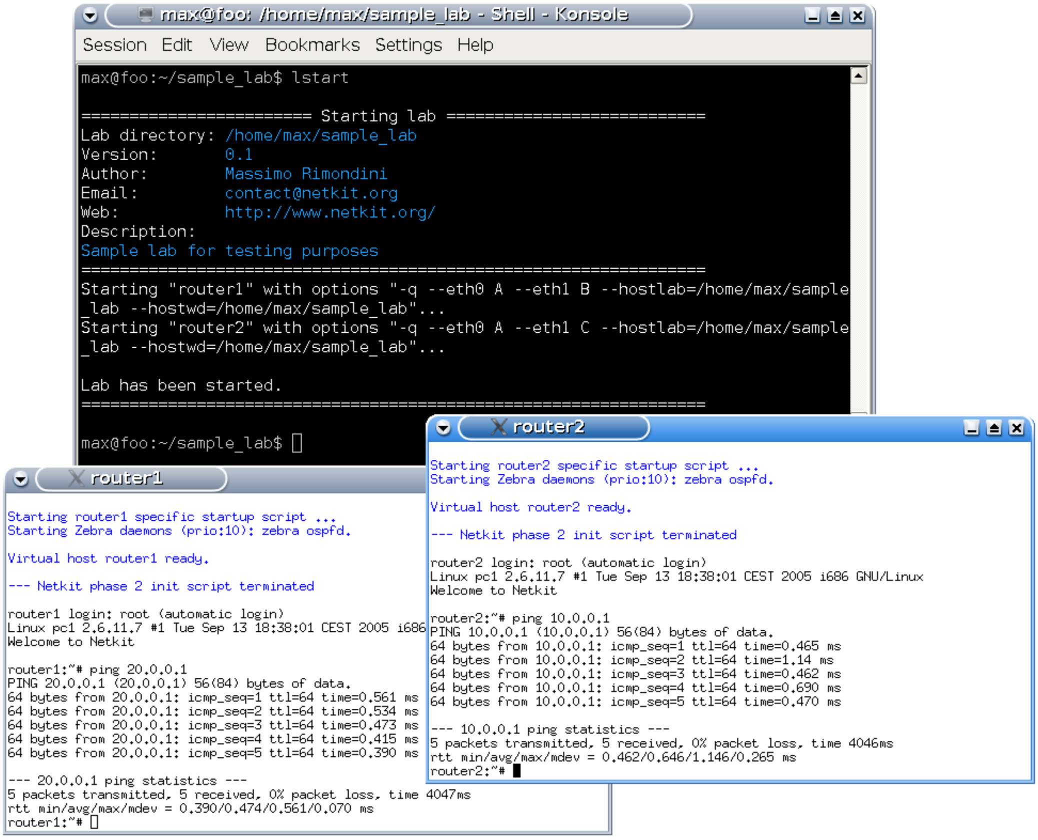

Figure 29: Startup of the Netkit lab described in Section 4

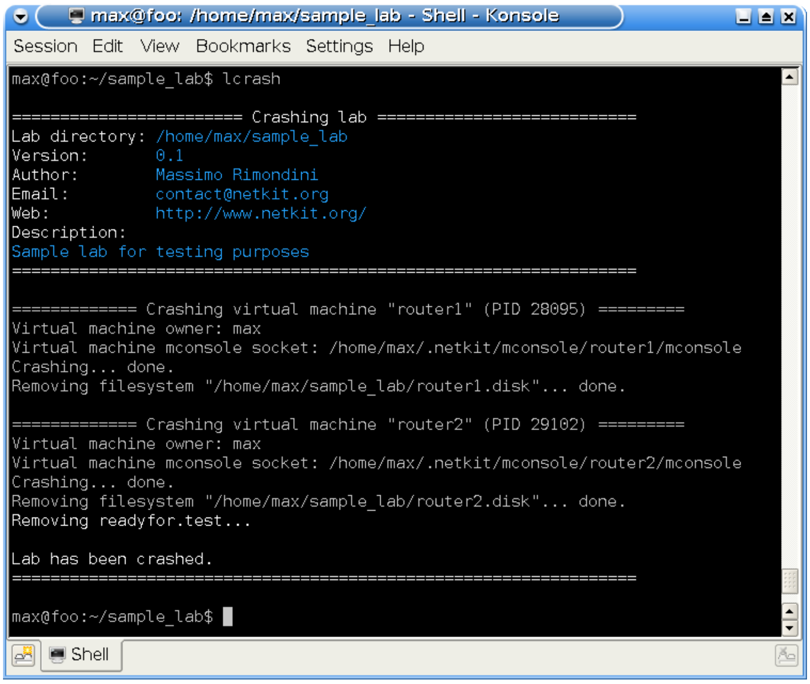

Figure 30: Shutdown of a Netkit lab by using `lcrash`.

router1 console

```
router1:~# telnet localhost ospfd
Trying 127.0.0.1...
Connected to router1.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password: zebra
ospfd> show ip ospf database

       OSPF Router with ID (30.0.0.1)

                Router Link States (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum  Link count
30.0.0.1        30.0.0.1        1193 0x80000004 0x0abd 1
30.0.0.2        30.0.0.2        1194 0x80000003 0x0abb 1

                Net Link States (Area 0.0.0.0)

Link ID         ADV Router      Age  Seq#       CkSum
30.0.0.2        30.0.0.2        1199 0x80000001 0xfce0

                AS External Link States

Link ID         ADV Router      Age  Seq#       CkSum  Route
10.0.0.0        30.0.0.1        1198 0x80000002 0x1282 E2 10.0.0.0/8 [0x0]
20.0.0.0        30.0.0.2        1198 0x80000002 0x89ff E2 20.0.0.0/8 [0x0]
```

Figure 31: A telnet session with `ospfd` showing the information collected by OSPF.

the lab can again be self tested and the results be compared against the signature. If they match, the lab is running properly.

The scripts for self testing must be placed inside the directory `_test`. Netkit understands that this is a special directory and does not start a virtual machine for it. The script `vm.test` inside `_test` is automatically executed by virtual machine `vm` when the lab is in launched in test mode (see Section 3.4). The output of this script is stored in `results/vm.user` inside the `_test` directory. In addition, Netkit also performs some default tests including the status of network interfaces, the contents of the kernel forwarding table, a summary of the ports in listening status, and a listing of running processes. The results of these default tests are stored inside `results/vm.default` inside directory `_test`.

Figure 28 shows a possible example of script for self testing. It reports the success of a connectivity test and the contents of the ARP cache. Observe that information that is subject to unpredictable changes (round-trip time for the `ping`, ordering of the entries in the ARP cache) is filtered out, so that the results of tests performed on different platforms can be immediately compared.

5 Managing a Virtual Lab

Once a Netkit virtual lab has been set up, it can be easily managed by means of the `ltools`. The lab can be simply started by issuing `lstart` on the host, as demonstrated in Figure 29. The Figure also shows that, thanks to OSPF, each router can also `ping` the LAN it is not directly connected to. Other investigations can be performed on the running lab by simply interacting with one of the virtual machine terminal windows. For example, Figure 31 shows the contents of the OSPF database.

Stopping a running lab is as simple as issuing `lcrash` or `lhalt` (slower) on the host (see Figure 30). Optionally, `lclean` can be used to get rid of temporary files (COW files, logs) left over after running the lab.

Netkit labs can be easily moved to another workstation by simply wrapping them into an archive, typically a `tar.gz`. Notice that, as there is no need to carry potentially large filesystem images, a lab

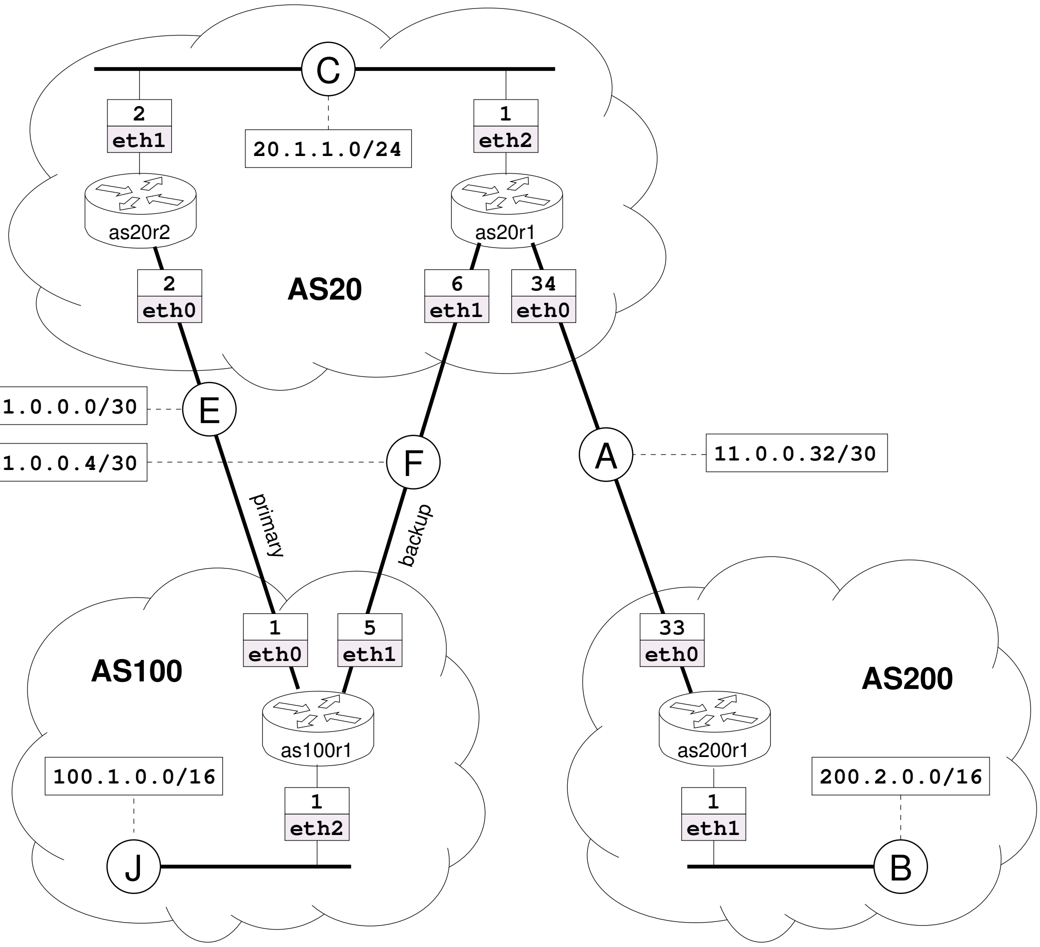

Figure 32: The topology of the lab for studying multihomed stub networks.

can be packed into a very small file and can therefore be also transferred very quickly over the web or by e-mail. Taking advantage of this, lots of ready to use lab experiences are made available on the Netkit web site [12].

Thanks to the architecture described in Section 3, Netkit makes it simple to build even large network scenarios consisting of several virtual devices. In theory there is no limit to the number of virtual machines that can compose a lab. In practice, experiments have shown that it is possible to run experiences consisting of more than 100 virtual machines on a typical workstation (see Section 3.1).

6 A Case Study: Multihoming

This Section provides an example of usage of Netkit to study issues that may arise in the interaction between routing protocols. We take as reference one of the labs made available on the Netkit web site [12], which considers a multihomed stub network as case study for pointing out these issues.

Figure 32 shows the topology of the lab we are considering. It consists of three Autonomous Systems: AS100 and AS200 are customers, while AS20 is their provider. AS100 is a multihomed stub, meaning that it has multiple connections to its ISP but does not provide transit service to its neighbours. Instead, AS200 is single homed. As a provider, AS20 supplies transit service to its customers. The Autonomous Systems in this topology exchange routing information with each other by using BGP. In particular, AS100 exploits its multiple connections to the provider AS20 to enforce a backup policy: the link on collision domain F is never used unless link E fails. Keeping a link completely unused is of course an unrealistic situation. However, it is a reasonable assumption for the purpose of showing an example of the effect of routing policies on the choice of routing paths. The two routers inside AS20 establish an iBGP peering to exchange external routes they have learned via BGP.

Host console

```
max@foo:~/netkit-lab_bgp-6-multi-homed-stub$ tree -n
.
|-- CHANGES
|-- as100r1
|   \-- etc
|       \-- zebra
|           |-- bgpd.conf
|           \-- daemons
|-- as100r1.startup
|-- as200r1
|   \-- etc
|       \-- zebra
|           |-- bgpd.conf
|           \-- daemons
|-- as200r1.startup
|-- as20r1
|   \-- etc
|       \-- zebra
|           |-- bgpd.conf
|           \-- daemons
|-- as20r1.startup
|-- as20r2
|   \-- etc
|       \-- zebra
|           |-- bgpd.conf
|           \-- daemons
|-- as20r2.startup
\-- lab.conf
```

Figure 33: Directory structure for the multihoming lab.

```
as20r1[0]="A"
as20r1[1]="F"
as20r1[2]="C"

as20r2[0]="E"
as20r2[1]="C"

as200r1[0]="A"
as200r1[1]="B"

as100r1[0]="E"
as100r1[1]="F"
as100r1[2]="J"
```

Figure 34: `lab.conf` file describing the topology in Figure 32

Figure 33 shows the hierarchy of files and directories that make up the lab. Apart from the file `CHANGES`, which is only a log of the fixes and changes made to the lab, it can be easily seen from the configuration files that routers in this lab run the BGP routing protocol. The topology of Figure 32 is implemented inside the file `lab.conf` as shown in Figure 34.

It is interesting to investigate in the BGP configuration of the customer `as100r1`. Figure 35 shows how to obtain this configuration through the command line interface of `bgpd` instead of the file `bgpd.conf`. The backup policy is enforced by the customer AS100 by using local-preference and metric. In particular, the route-map `localPrefIn` lowers to 90 the local-preference on announcements coming from neighbour `as20r1` (link F, used as backup); for the announcements received from link E the default value of 100 is fine. The other route-map, `metricOut`, is used to increase to 10 the metric on outgoing announcements made by `as100r1` to its neighbour `as20r1`; as for outgoing announcements directed to `as20r2`, the default metric value of 0 is retained. In this way, both the outgoing and the incoming traffic are discouraged from taking a path through link F. The other prefix-lists prevent transit traffic from traversing the customer `as100r1` by discarding announcements of extraneous prefixes.

By looking at the BGP routing table of `as100r1` (Figure 36), it can be easily seen that the chosen as100r1 console

```
as100r1:~# telnet localhost bgpd
Trying 127.0.0.1...
Connected to as100r1.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password: zebra
bgpd> enable
Password: zebra
bgpd# show running-config

Current configuration:
!
hostname bgpd
password zebra
enable password zebra
log file /var/log/zebra/bgpd.log
!
debug bgp
debug bgp events
debug bgp keepalives
debug bgp updates
debug bgp fsm
debug bgp filters
!
router bgp 100
 network 100.1.0.0/16
 neighbor 11.0.0.2 remote-as 20
 neighbor 11.0.0.2 description Router as20r2 (primary)
 neighbor 11.0.0.2 prefix-list defaultIn in
 neighbor 11.0.0.2 prefix-list mineOutOnly out
 neighbor 11.0.0.6 remote-as 20
 neighbor 11.0.0.6 description Router as20r1 (backup)
 neighbor 11.0.0.6 prefix-list defaultIn in
 neighbor 11.0.0.6 prefix-list mineOutOnly out
 neighbor 11.0.0.6 route-map localPrefIn in
 neighbor 11.0.0.6 route-map metricOut out
!
access-list myAggregate permit 100.1.0.0/16
!
ip prefix-list defaultIn seq 5 permit 0.0.0.0/0
ip prefix-list mineOutOnly seq 5 permit 100.1.0.0/16
!
route-map metricOut permit 10
 match ip address myAggregate
 set metric 10
!
route-map localPrefIn permit 10
 set local-preference 90
!
line vty
!
end
```

Figure 35: Configuration of BGP on router as100r1.

```
                                  as100r1 console
as100r1:~# telnet localhost bgpd
Trying 127.0.0.1...
Connected to as100r1.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password: zebra
bgpd> show ip bgp
BGP table version is 0, local router ID is 100.1.0.1
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 0.0.0.0          11.0.0.2                               0 20 i
*                   11.0.0.6                      90       0 20 i
*> 100.1.0.0/16     0.0.0.0                  0         32768 i

Total number of prefixes 2
```

Figure 36: The BGP routing table of as100r1.

```
                                  as100r1 console
as100r1:~# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
11.0.0.4        *               255.255.255.252 U     0      0        0 eth1
11.0.0.0        *               255.255.255.252 U     0      0        0 eth0
100.1.0.0       *               255.255.0.0     U     0      0        0 eth2
default         11.0.0.2        0.0.0.0         UG    0      0        0 eth0
```

Figure 37: Kernel forwarding table of as100r1.

backup policy is actually being enforced. In fact, due to the higher value of the local-preference, the instance of BGP running on `as100r1` has chosen the path through neighbour `11.0.0.2` (`as20r2`) as best alternative to reach the default route announced by the provider. The "greater than" symbol (`>`) indicates the currently selected route. This information is further confirmed by the entry injected in the kernel forwarding table (see Figure 37).

Figures 38 and 41 show the status (`show ip bgp summary`) and routing table (`show ip bgp`) of BGP on `as20r2` and `as20r1`, respectively. The status confirms that the iBGP peering between `20.1.1.1` and `20.1.1.2` is active. However, while `as20r2` chooses `11.0.0.1` (link E, used as primary) as best alternative to reach `as100r1`, the same does not happen on `as20r1` despite the fact that the entry with next hop `11.0.0.1` is present in the routing table of BGP, has lower metric and higher local-preference. Even more strangely, Figure 38 shows that the only available alternative on `as20r2` to reach prefix `200.2.0.0/16` has not been selected as best, and has therefore not been injected in the kernel forwarding table. As a consequence, a `ping` from `as20r2` to `200.2.0.1` unavoidably fails (see Figure 39). To complete the picture of oddities, Figure 40 shows that a `traceroute` from `as200r1` to `100.1.0.1` reveals that the backup link F is being used for traffic directed to `as100r1`, which is undesired.

Obviously, something is going wrong with routing, even if the deployed configuration is correct. The point here is that BGP only considers an entry of the routing table as usable if the path to reach its next hop has been learned by some IGP or is statically configured. In our case neither of the conditions occurs, as `as20r2` learns how to reach `11.0.0.33` only via iBGP and `as20r1` learns how to reach `11.0.0.1` again only via iBGP. The origin of learned routes can be checked by querying the Zebra daemon. For example, Figure 42 shows that the route to `11.0.0.33` has been learned by BGP.

There are two possible fixes to this situation. The first one is to enable some IGP inside AS20, so that information about directly connected networks is propagated inside the AS and can be used by BGP to reach all the next hops. The second one, which we apply here, is to configure two static entries in

```
                                    as20r2 console
as20r2:~# telnet localhost bgpd
Trying 127.0.0.1...
Connected to as20r2.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password: zebra
bgpd> show ip bgp summary
BGP router identifier 20.1.1.2, local AS number 20
3 BGP AS-PATH entries
0 BGP community entries

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
11.0.0.1        4   100     116     117        0    0    0 01:53:34        1
20.1.1.1        4    20     118     118        0    0    0 01:53:39        6

Total number of neighbors 2
bgpd> show ip bgp
BGP table version is 0, local router ID is 20.1.1.2
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
* i0.0.0.0          20.1.1.1                 0    100      0 i
*>                  0.0.0.0                  0         32768 i
*> 11.0.0.0/30      0.0.0.0                  0         32768 i
*>i11.0.0.4/30      20.1.1.1                 0    100      0 i
*>i11.0.0.32/30     20.1.1.1                 0    100      0 i
* i20.1.1.0/24      20.1.1.1                 0    100      0 i
*>                  0.0.0.0                  0         32768 i
*> 100.1.0.0/16     11.0.0.1                 0             0 100 i
* i                 11.0.0.5                10    100      0 100 i
* i200.2.0.0/16     11.0.0.33                0    100      0 200 i

Total number of prefixes 7
```

Figure 38: Peering status and BGP routing table of as20r2.

```
                                    as20r2 console
as20r2:~# ping 200.2.0.1
connect: Network is unreachable
```

Figure 39: A failed ping from as20r2.

```
                                    as200r1 console
as200r1:~# traceroute 100.1.0.1
traceroute to 100.1.0.1 (100.1.0.1), 64 hops max, 40 byte packets
 1  11.0.0.34 (11.0.0.34)  1 ms  2 ms  1 ms
 2  100.1.0.1 (100.1.0.1)  2 ms  2 ms  2 ms
```

Figure 40: Traffic from as200r1 to 100.1.0.1 takes the path through the backup link F, which is undesired.

as20r1 console

```
as20r1:~# telnet localhost bgpd
Trying 127.0.0.1...
Connected to as20r1.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.


User Access Verification

Password: zebra
bgpd> show ip bgp summary
BGP router identifier 20.1.1.1, local AS number 20
3 BGP AS-PATH entries
0 BGP community entries

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
11.0.0.5        4   100     118     121        0    0    0 01:56:49        1
11.0.0.33       4   200     118     120        0    0    0 01:56:50        1
20.1.1.2        4    20     119     122        0    0    0 01:56:41        4

Total number of neighbors 3
bgpd> show ip bgp
BGP table version is 0, local router ID is 20.1.1.1
Status codes: s suppressed, d damped, h history, * valid, > best, i - internal
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
* i0.0.0.0          20.1.1.2                 0    100      0 i
*>                  0.0.0.0                  0         32768 i
*>i11.0.0.0/30      20.1.1.2                 0    100      0 i
*> 11.0.0.4/30      0.0.0.0                  0         32768 i
*> 11.0.0.32/30     0.0.0.0                  0         32768 i
* i20.1.1.0/24      20.1.1.2                 0    100      0 i
*>                  0.0.0.0                  0         32768 i
* i100.1.0.0/16     11.0.0.1                 0    100      0 100 i
*>                  11.0.0.5                10             0 100 i
*> 200.2.0.0/16     11.0.0.33                0             0 200 i

Total number of prefixes 7
```

Figure 41: Peering status and BGP routing table of `as20r1`.

as20r2 console

```
as20r2:~# telnet localhost zebra
Trying 127.0.0.1...
Connected to as20r2.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.

User Access Verification

Password: zebra
Router> show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       B - BGP, > - selected route, * - FIB route

C>* 11.0.0.0/30 is directly connected, eth0
B>* 11.0.0.4/30 [200/0] via 20.1.1.1, eth1, 03:19:58
B>* 11.0.0.32/30 [200/0] via 20.1.1.1, eth1, 03:19:58
C>* 20.1.1.0/24 is directly connected, eth1
B>* 100.1.0.0/16 [20/0] via 11.0.0.1, eth0, 03:19:53
C>* 127.0.0.0/8 is directly connected, lo
```

Figure 42: The protocol by which routes in the forwarding table have been learned can be checked by querying the `zebra` routing daemon.

```
as20r2 console
as20r2:~# route add -net 11.0.0.32/30 gw 20.1.1.1 dev eth1
```

Figure 43: Configuration of a static entry to reach `11.0.0.33` in the forwarding table of `as20r2`.

```
as20r1 console
as20r1:~# route add -net 11.0.0.0/30 gw 20.1.1.2 dev eth2
```

Figure 44: Configuration of a static entry to reach `11.0.0.1` in the forwarding table of `as20r1`.

```
as20r2 console
as20r2:~# telnet localhost zebra
Trying 127.0.0.1...
Connected to as20r2.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.

User Access Verification

Password: zebra
Router> show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP, O - OSPF,
       B - BGP, > - selected route, * - FIB route

C>* 11.0.0.0/30 is directly connected, eth0
B>* 11.0.0.4/30 [200/0] via 20.1.1.1, eth1, 00:00:43
K>* 11.0.0.32/30 via 20.1.1.1, eth1
B   11.0.0.32/30 [200/0] via 20.1.1.1, eth1, 00:00:43
C>* 20.1.1.0/24 is directly connected, eth1
B>* 100.1.0.0/16 [20/0] via 11.0.0.1, eth0, 00:00:41
C>* 127.0.0.0/8 is directly connected, lo
```

Figure 45: Statically configured routes are marked as `kernel` by Zebra and are preferred over those learned via other routing protocols.

the forwarding tables of `as20r2` and `as20r1` telling them how to reach the next hops `11.0.0.33` and `11.0.0.1`, respectively. The commands for doing this are shown in Figures 43 and 44. The statically configured routes are marked by Zebra as being learned from the kernel (see Figure 45) and are preferred over routes learned via other routing protocols.

After the fix has been applied, everything starts to work as desired. Figures 46 and 47 show that `traceroute`s actually use the primary link E.

Now that everything is working as expected, we can check that the backup policy is operational: we intentionally break link E and check that the traffic shifts to link F. Provided that we are inside an emulation environment, a link can be broken in at least two ways. One is to simply bring down one of the network interfaces attached to that link, by using the following command on either `as20r2` or `as100r1`:

```
ifconfig eth0 down
```

The other one is to administratively shutdown the BGP peering between `as20r2` and `as100r1`. We choose the latter alternative, which can be carried into effect by issuing the commands shown in Figure 48. Notice that, since a BGP peering must be explicitly configured by both participants, shutting it down on just one side is enough to divert routing.

After waiting some time for the routing updates to propagate, we fall into the expected state in which all the traffic uses the backup link F (see Figure 49).

We do not show, but it is easy to imagine, that restoring the peering also results in traffic shifting back to the primary link E.

7 Conclusions

We provide a survey of environments for the emulation of computer networks. We describe in detail Netkit, a lightweight network emulator based on User-Mode Linux. We show how Netkit can be effectively

```
as100r1 console
as100r1:~# traceroute 200.2.0.1
traceroute to 200.2.0.1 (200.2.0.1), 64 hops max, 40 byte packets
 1  11.0.0.2 (11.0.0.2)  1 ms  1 ms  1 ms
 2  20.1.1.1 (20.1.1.1)  1 ms  2 ms  1 ms
 3  200.2.0.1 (200.2.0.1)  3 ms  2 ms  2 ms
```

Figure 46: After fixing the forwarding tables of routers in AS20, a `traceroute` from `as100r1` to `200.2.0.1` correctly uses `as100r1`'s primary upstream link E.

```
as200r1 console
as200r1:~# traceroute -n 100.1.0.1
traceroute to 100.1.0.1 (100.1.0.1), 64 hops max, 40 byte packets
 1  11.0.0.34  1 ms  1 ms  1 ms
 2  20.1.1.2  1 ms  1 ms  1 ms
 3  100.1.0.1  2 ms  2 ms  2 ms
```

Figure 47: After fixing the forwarding tables of routers in AS20, a `traceroute` from `as200r1` to `100.1.0.1` correctly uses `as100r1`'s primary upstream link E.

```
as20r2 console
as20r2:~# telnet localhost bgpd
Trying 127.0.0.1...
Connected to as20r2.
Escape character is '^]'.

Hello, this is zebra (version 0.94).
Copyright 1996-2002 Kunihiro Ishiguro.

User Access Verification

Password: zebra
bgpd> enable
Password: zebra
bgpd# configure terminal
bgpd(config)# router bgp 20
bgpd(config-router)# neighbor 11.0.0.1 shutdown
bgpd(config-router)# end
bgpd# show ip bgp summary
BGP router identifier 20.1.1.2, local AS number 20
3 BGP AS-PATH entries
0 BGP community entries

Neighbor        V    AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
11.0.0.1        4   100     220     222        0    0    0 00:00:05 Idle (Admin)
20.1.1.1        4    20     226     223        0    0    0 03:37:37        6

Total number of neighbors 2
```

Figure 48: A telnet session showing the commands to administratively shutdown a BGP peering.

```
as100r1 console
as100r1:~# traceroute 200.2.0.1
traceroute to 200.2.0.1 (200.2.0.1), 64 hops max, 40 byte packets
 1  11.0.0.6 (11.0.0.6)  1 ms  1 ms  1 ms
 2  200.2.0.1 (200.2.0.1)  1 ms  2 ms  1 ms
```

```
as200r1 console
as200r1:~# traceroute -n 100.1.0.1
traceroute to 100.1.0.1 (100.1.0.1), 64 hops max, 40 byte packets
 1  11.0.0.34  2 ms  1 ms  1 ms
 2  100.1.0.1  1 ms  1 ms  1 ms
```

Figure 49: After administratively shutting down the peering between `as20r2` and `as100r1`, traffic goes through the backup link F.

exploited to ease experimenting with complex scenarios consisting of several virtual devices. We also explain how these scenarios can be easily packed and redistributed for sharing them with others. Last, we show a sample application of Netkit to the study of a multihoming configuration, pointing out issues related to the interplay between BGP and IGP protocols. Among the other network emulators, Netkit has the advantages of being lightweight and to support installation and execution fully in user space. It also facilitates the exchange and redistribution of emulated scenarios and supports reasonably good scalability by allowing to run more than one hundred of virtual devices on a typical workstation. Moreover, Netkit provides users with a familiar environment consisting of a standard Linux distribution and routing daemons using a Cisco-like command syntax.

One of the most natural contexts of application of Netkit is probably didactics: it can be effectively exploited in teaching networking by giving the students the opportunity to experiment with the protocols and services they are learning. However, we believe that Netkit may prove itself useful for both operators and researchers in several other contexts, ranging from testing of configurations before deployment to debugging and development of new services and protocols, from studying abnormal routing behaviours to validating theoretical model by experimentation.

8 Team and History

Netkit is the result of the joint work of several people over time. Giuseppe Di Battista and Maurizio Pizzonia conceived the first idea of Netkit in 1999. At that time, Andrea Cecchetti suggested to use User-Mode Linux. Maurizio Pizzonia implemented the first releases of the vtools. A significant improvement has then been introduced by Stefano Pettini who devised the idea of the ltools to support complex scenarios and wrote the first version. Most of the ready to use labs available on the Netkit site have been devised and implemented by Giuseppe Di Battista and Maurizio Patrignani, who currently use them within their courses (basic and advanced) at the Roma Tre University, where hundreds of students every year use Netkit to learn networking. Netkit has also been successfully used in several other contexts and different people have contributed to it. In particular, Sandro Doro is maintaining a live CD version of Netkit and is using it within his networking courses. Federico Mariani helped us in emulating the GARR network. Lorenzo Colitti performed extensive experimentations with IPv6 inside Netkit. Fabio Ricci implemented the self testing system for Netkit labs.

We would also like to thank the community of Netkit users for the useful hints and discussions in the Netkit mailing list, as well as all the people who put their effort in improving and making Netkit what it currently is and who are not explicitly mentioned here.

References

[1] NCTUns 1.0. *IEEE Network Magazine*, 17(4), Jul 2003. Appeared in the column "Software Tools for Networking".

[2] Aaron Klingaman, Mark Huang, Steve Muir, and Larry Peterson. PlanetLab Core Specification 4.0. Technical Report PDN–06–032, PlanetLab Consortium, Jun 2006.

[3] Advanced Micro Devices. Introducing AMD Virtualization™. `http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8826_14287,00.html`.

[4] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[5] Andrea Bittau and Mark Handley. Decoupling Policy from Protocols. Draft paper, Feb 2006.

[6] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. *ACM SIGCOMM Computer Communication Review*, 36(4):3–14, Sep 2006.

[7] Brad Marshall. User Mode Linux, VMWare and Wine – Virtual Machines under Linux, Apr 2003. `http://quark.humbug.org.au/publications/linux/uml.pdf`.

[8] Bruno Quoitin. C-BGP. http://cbgp.info.ucl.ac.be/.

[9] Bruno Quoitin. CBGP – A new approach to BGP simulation. E-Next Advanced two-day course on Interdomain Routing with BGP4, Nov 2004. http://cbgp.info.ucl.ac.be/downloads/cbgp-1.pdf, http://cbgp.info.ucl.ac.be/downloads/cbgp-2.pdf.

[10] Bruno Quoitin and Steve Uhlig. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network*, 19(6), Nov 2005.

[11] University of Roma Tre Computer Networks Research Group. Netkit. http://www.netkit.org.

[12] University of Roma Tre Computer Networks Research Group. Netkit ready to use Labs. http://www.netkit.org/labs.html.

[13] University of Roma Tre Computer Networks Research Group. NetML. http://www.dia.uniroma3.it/~compunet/netml/.

[14] Debian. APT Howto. http://www.debian.org/doc/manuals/apt-howto/.

[15] Friedrich Alexander Universität Erlangen-Nürnberg Department of Computer Science. FAUmachine Project (formerly UMLinux). http://www3.informatik.uni-erlangen.de/Research/FAUmachine/.

[16] University of California, San Diego Department of Computer Science. ModelNet. http://modelnet.ucsd.edu/.

[17] Purdue University Department of Computer Sciences. vBET: a VM-Based Education Testbed. http://www.cs.purdue.edu/homes/jiangx/vBET/.

[18] University of Zagreb Department of Telecommunications. IMUNES – An Integrated Multiprotocol Network Emulator/Simulator. http://www.tel.fer.hr/imunes/.

[19] Fabrice Bellard. QEMU Open Source Processor Emulator. http://www.qemu.org/, http://fabrice.bellard.free.fr/qemu/.

[20] Fabrizio Ciacchi. Linux in Linux. Linux Magazine, May 2005.

[21] Fermín Galán and David Fernández. VNUML: Una Herramienta de Virtualización de Redes Basada en Software Libre. In *Proc. Open Source International Conference 2004*, pages 35–41, Feb 2004. In Spanish.

[22] Fermín Galán, David Fernández, Javier Ruiz, Omar Walid, and Tomás de Miguel. Use of Virtualization Tools in Computer Network Laboratories. In *Proc. 5th International Conference on Information Technology Based Higher Education and Training (ITHET 2004)*, pages 209–214, Jun 2004.

[23] G. Malkin. RIP Version 2. RFC 2453, Nov 1998.

[24] Gerd Stolpmann. UMLMON. http://www.gerd-stolpmann.de/buero/umlmon.html.en.

[25] Ian Pratt. Xen and the Art of Virtualization. Ottawa Linux Symposium 2004, 2004.

[26] Ian Pratt. Xen 3.0 and the Art of Virtualization. Ottawa Linux Symposium 2005, 2005.

[27] Ian Pratt. Xen and the Art of Virtualization. Ottawa Linux Symposium 2006, 2006.

[28] Innotek. VirtualBox. http://www.virtualbox.org/.

[29] Intel. Intel®Virtualization Technology. http://www.intel.com/technology/virtualization/index.htm.

[30] International Computer Science Institute, Berkeley, California. XORP Open Source IP Router. http://www.xorp.org/.

[31] Ivan Santarelli and Alexandra Bellogini. NetML – Network Markup Language. RIPE Meeting 47, Feb 2004.

[32] J. Moy. OSPF Version 2. RFC 2328, Apr 1998.

[33] Jay Chen, Diwaker Gupta, Kashi V. Vishwanath, Alex C. Snoeren, and Amin Vahdat. Routing in an Internet-Scale Network Emulator. In *Proc. 12th IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, pages 275–283. IEEE Computer Society, 2004.

[34] Jay Lepreau. Emulab: Recent Work, Ongoing Work. Talk at DETER Lab Community Meeting, Jan 2006.

[35] Jeff Dike. A User-Mode Port of the Linux Kernel. `http://user-mode-linux.sourceforge.net/als2000/index.html`.

[36] Jeff Dike. A User-Mode Port of the Linux Kernel. In *Proc. 4th Annual Linux Showcase & Conference*, pages 63–72, Oct 2000.

[37] Jeff Dike. User-Mode Linux. Talk at the 2001 Ottawa Linux Symposium, Jul 2001.

[38] Jeff Dike. Double your Fun with User-Mode Linux, Nov 2004.

[39] Jeff Dike. *User Mode Linux*. Prentice Hall, Apr 2006.

[40] Jeroen van der Ham, Freek Dijkstra, Franco Travostino, Hubertus Andree, and Cees de Laat. Using RDF to Describe Networks. *Future Generation Computer Systems, Feature topic iGrid 2005*, 2006. `http://staff.science.uva.nl/~vdham/research/publications/0510-NetworkDescriptionLanguage.pdf`.

[41] Jeroen van der Ham, Paola Grosso, and Cees de Laat. Semantics for Hybrid Networks Using the Network Description Language. Poster at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2006), Mar 2006. `http://staff.science.uva.nl/~vdham/research/publications/0603-NetworkDescriptionLanguage.pdf`.

[42] Jeroen van der Ham, Paola Grosso, Ronald van der Pol, Andree Toonk, and Cees de Laat. Using the Network Description Language in Optical Networks. May 2006. Accepted at the IEEE Integrated Management Conference 2007.

[43] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proc. 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004)*, Oct 2004.

[44] Kunihiro Ishiguro. GNU Zebra Routing Software. `http://www.zebra.org/`.

[45] Kuthonuzo Luruo and Shashank Khanvilkar. Virtual Networking with User-Mode Linux. Linux for You – Pro, Mar 2005.

[46] Kyron. Linux Virtual Server. `http://www.kyron.it/virtual_server.asp`.

[47] Larry Peterson, Steve Muir, Timothy Roscoe, and Aaron Klingaman. PlanetLab Architecture: An Overview. Technical Report PDN–06–031, PlanetLab Consortium, May 2006.

[48] Larry Peterson and Timothy Roscoe. The Design Principles of PlanetLab. *ACM SIGOPS Operating Systems Review*, 40(1):11–16, 2006.

[49] Linode.com. Virtual Private Server. `http://linode.com/`.

[50] Mark Handley. Proposal to Develop an Extensible Open Router Platform. Initial project proposal, Nov 2000.

[51] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing Extensible IP Router Software. In *Proc. 2ns USENIX Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.

[52] Mark Handley, Orion Hodson, and Eddie Kohler. XORP: An Open Platform for Network Research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.

[53] Marko Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proc. 2003 USENIX Annual Technical Conference*, Jun 2003.

[54] Microsoft. Virtual PC 2004. `http://www.microsoft.com/italy/windows/virtualpc/default.mspx`.

[55] University of Cambridge Networks and Operating Systems Group. XEN. `http://www.cl.cam.ac.uk/research/srg/netos/xen/`.

[56] Paolo Giarrusso. SKAS patches and UML updates. `http://www.user-mode-linux.org/~blaisorblade/`.

[57] Paolo Giarrusso. UML Utilities. `http://www.user-mode-linux.org/~blaisorblade/uml-utilities/`.

[58] Parallels. Parallels Workstation. `http://www.parallels.com/`.

[59] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symposium on Operating System Principles (SOSP 2003)*, Oct 2003.

[60] PlanetLab Consortium. PlanetLab. `http://www.planet-lab.org/`.

[61] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad hoc and Wireless Networks. In *Proc. 2005 Workshop on Wireless Traffic Measurements and Modeling (WiTMeMo 2005)*, pages 7–12. USENIX Association, 2005.

[62] Renzo Davoli. VDE: Virtual Distributed Ethernet. `http://sourceforge.net/projects/vde/`.

[63] Renzo Davoli. VDE: Virtual Distributed Ethernet. Technical Report UBLCS-2004-12, University of Bologna, Jun 2004.

[64] Renzo Davoli. VDE: Virtual Distributed Ethernet. In *Proc. 1st International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2005)*, pages 213–220. IEEE Computer Society, 2005.

[65] S. Y. Wang. Using the NCTUns 2.0 Network Simulator and Emulator to Facilitate Network Researches. In *Proc. 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM 2006)*, Mar 2006.

[66] S. Y. Wang, C. L. Chou, C. H. Huang, C. C. Hwang, Z. M. Yang, C. C. Chiou, and C. C. Lin. The Design and Implementation of the NCTUns 1.0 Network Simulator. *Computer Networks*, 42(2):175–197, 2003.

[67] S. Y. Wang and H. T. Kung. A New Methodology for Easily Constructing Extensible and High-fidelity TCP/IP Network Simulators. *Computer Networks*, 40(2):257–278, 2002.

[68] Sandro Doro. Netkit4TIC Virtual Laboratory. `http://www.tic.fdns.net/tic/html/lab.html`.

[69] Shie-Yuan Wang and Kuo-Chiang Liao. Innovative Network Emulations Using the NCTUns Tool. *Computer Networking and Networks*, pages 157–187, 2006.

[70] SimReal Inc. NCTUns Network Simulator and Emulator. `http://nsl10.csie.nctu.edu.tw/`.

[71] Suman Banerjee, Timothy G. Griffin, and Marcelo Pias. The Interdoman Connectivity of PlanetLab Nodes. In *Proc. Passive & Active Measurement Workshop (PAM 2004)*, Apr 2004.

[72] Technical University of Madrid (UPM) Telematics Engineering Department. VNUML. `http://jungla.dit.upm.es/~vnuml/`.

[73] Universiteit van Amsterdam. NDL – Network Description Language. `http://www.science.uva.nl/research/sne/ndl/`.

[74] University of Utah. Emulab Network Emulation Testbed. http://www.emulab.net/.

[75] Bochs. http://bochs.sourceforge.net/.

[76] Cooperative Linux. http://www.colinux.org/.

[77] Debian GNU/Linux. http://www.debian.org/.

[78] DOSBox. http://dosbox.sourceforge.net/.

[79] DOSEMU. http://www.dosemu.org/.

[80] EINAR (Einar Is Not a Router) Router Simulator. http://www.isk.kth.se/proj/einar/.

[81] Network stack cloning/virtualization extensions to the FreeBSD kernel. http://www.tel.fer.hr/zec/BSD/vimage/.

[82] GARR - The Italian Academic and Research Network. http://www.garr.it/.

[83] Graphviz Graph Visualization Software. http://www.graphviz.org/.

[84] The Linux Kernel Archives. http://www.kernel.org/.

[85] KVM Kernel-based Virtual Machine. http://kvm.sourceforge.net/.

[86] The MLN Project. http://mln.sourceforge.net/.

[87] netfilter – Firewalling, NAT, and Packet Mangling for Linux. http://www.netfilter.org/.

[88] Packages installed in Netkit filesystem version F2.2. http://www.netkit.org/download/netkit-filesystem/installed-packages-F2.2.

[89] The Network Simulator – NS-2. http://www.isi.edu/nsnam/ns/.

[90] PearPC PowerPC Architecture Emulator. http://pearpc.sourceforge.net/.

[91] PlanetLab Acceptable Use Policy. http://www.planet-lab.org/php/aup/.

[92] Plex86 x86 Virtual Machine Project. http://plex86.sourceforge.net/, http://www.plex86.org/.

[93] Q – Mac port of QEMU. http://www.kju-app.org/kju/.

[94] Quagga Routing Suite. http://www.quagga.net/.

[95] Scalable Simulation Framework (SSFNet). http://www.ssfnet.org/homePage.html.

[96] Serenity Virtual Station (formerly TwoOSTwo). http://www.serenityvirtual.com/.

[97] User-mode Linux Kernel. http://user-mode-linux.sourceforge.net/.

[98] UML Utilities. http://user-mode-linux.sourceforge.net/dl-sf.html.

[99] Università degli Studi Roma Tre. http://www.uniroma3.it/.

[100] VINI – A Virtual Network Infrastructure. http://vini-veritas.net/.

[101] Virtuozzo. http://www.swsoft.com/en/products/virtuozzo.

[102] VMware. http://www.vmware.com/.

[103] Win4Lin. http://www.win4lin.com/.

[104] Wine. http://www.winehq.com/.

[105] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Jan 2006.