building a virtual IPv6 lab using user-mode Linux

by Salah M.S. Al-Buraiky

Salah M.S. Al-Buraiky is a data network engineer working for the Communication Solutions Engineering Group of Saudi Aramco. He is also an electrical engineering graduate student at King Fahd University of Petroleum and Minerals. He is specializing in data communication and machine learning.

salah.buraiky@aramco.com

The function of an operating system is to provide users and applications with a high-level abstraction of the underlying hardware architecture, isolating them from the complexity of such architecture and providing them with a simple and consistent view of system resources. Applications running within a certain operating system actually deal with a "virtual machine" composed of the physical hardware and the layer of abstraction superimposed on it by the operating system. User-mode Linux (UML) is a port of the Linux kernel to the virtual machine composed of the physical hardware and the Linux kernel. In simple terms, UML is a kernel patch that allows users (even unprivileged users) to run an instance of the Linux kernel as a user-land process. UML was introduced by Jeff Dike and is available for the 2.4 kernel series as a patch, while being a standard part of the 2.6 kernel series.

In this article, the kernel running as a user process will be called the UML kernel, while the "real" kernel will be called the Linux kernel. Similarly, the virtual machine consisting of the UML kernel, its root file system and the processes created by it will be called the UML machine, while the "real" machine will be called the host machine.

Having the ability to run a Linux kernel as a user-space process has many practical applications. Some of the uses of UML are:

- Kernel development: With UML, familiar user-space debugging and performance profiling tools can be used for kernel development. A misbehaving UML kernel can be just killed like a normal process – no need to reboot if your experimental kernel crashes.
- Virtual hosting: With UML, a single physical machine can host a number of UML machines, depending on the available processing power and memory. Each UML machine can be dedicated to a user to run whatever services he or she needs.
- Honeypot building: A UML machine can be used as a honeypot for hackers, where it offers a sandbox for them to play around in without causing any harm, while providing security experts with the opportunity to study their techniques.
- Virtual networking: UML machines running on the same host can be networked together and with the host machine. They can also be connected to the rest of the world, using the host machine as a gateway.

Being a data network engineer, I am mostly interested in virtual networking; in fact, the original motivation for me to explore UML was my need for IPv6 routers and servers to experiment with as part of an IPv6 migration study I am working on. In the lab, UML provides me with a cost-effective and space-conserving method of constructing an IPv6 network to test routing and serving with the new network layer protocol. Outside the lab, UML provides me with a "virtual portable lab" in my laptop, allowing me to carry my experimental IPv6 network with me while I move around.

In this article, I'll present a method for creating UML machines for experimenting with the IPv6 protocol. The procedure presented, however, can be used for other data-networking experiments. The article assumes that the reader is familiar with IPv6 and will show how to create an IPv6 network consisting of three IPv6 routers connected with point-to-point links. One of the routers is the host machine (called alpha) and the other two are UML machines, ghost and shadow (see Figure 1). Although the standard Linux kernel contains an IPv6 stack, it is recommended to use the USAGI Linux stack. The USAGI (UniverSAl playGround for IPv6) implementation has better performance, better standard conformance, and fewer bugs compared to the IPv6 stack of the standard kernel. The home page of the USAGI project is *http://www.linux-ipv6.org*. In our project, we'll use the standard Linux kernel for ghost and the USAGI kernel for shadow, just to illustrate the procedure for both kernels. The routing software we'll be using is GNU Zebra (*http://www.zebra.org*).

The article will show the reader what software components are needed, how to customize and create a UML kernel, how to create a root file system for the UML machine, and how to configure networking. Using pre-built components all the way makes the process much easier but leads to a rigid configuration, while building all components from scratch may be difficult and time-consuming. Therefore, the procedure presented tries to adopt a hybrid approach to achieve a balance between customizability and ease of implementation.

The Building Software Blocks

The needed software components are:

- A fresh source tree for a recent kernel obtained from any kernel source repository mirror. The kernel source I'll be using for this article is *http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.20.tar.bz2*.
- A USAGI-patched kernel. This can be obtained from *ftp://ftp.linux-ipv6.org/pub/usagi/stable/kit/* or any other USAGI mirror. I had some trouble compiling the USAGI kernel with the UML patch, but the following USAGI package worked for me:

```
usagi-linux24-stable-20021007.tar
with patch: uml-patch-2.4.19-51.bz2
```

- The UML patch to be applied to the kernel. Choose a patch that matches the version of the kernel source tree you have downloaded. The patches I'll be using here are *http://prdownloads.sourceforge.net/user-mode-linux/uml-patch-2.4.20-6.bz2*, for the standard kernel, and *http://prdownloads.sourceforge.net/user-mode-linux/uml-patch-2.4.19-51.bz2* for the USAGI kernel.
- The user_mode_linux package. This package contains a pre-built UML kernel and a number of utilities to be used with UML. The pre-built UML kernel contained in the package will only be used initially and will be replaced by a customized kernel once the root file system is built. The version used for this article is *http://prdownloads.sourceforge.net/user-mode-linux/user_mode_linux-2.4.19.5um-0.i386.rpm*.
- The UMLBuilder package. The root file system of a UML kernel is created within an ordinary file structured like a file system rather than on a disk partition, as is usually the case with "real" kernels. We'll call that file the rootfs file. A relatively easy way to create a customized rootfs file is to use UMLBuilder, a graphical application for creating rootfs files from RPM-based distributions (e.g., RedHat and SuSE). UMLBuilder requires the user_mode_linux package in order to work. The version used for this article is *http://prdownloads.sourceforge.net/umlbuilder/umlbuilder-1.40-5.i386.rpm*.
- The GNU Zebra Routing Package: There isn't a need to worry about downloading GNU Zebra, since it is part of the RedHat distribution.

After getting the needed software, perform the following preparatory steps.

First, install the RPM packages:

```
rpm -i user_mode_linux-2.4.19.5um-0.i386.rpm
rpm -i umlbuilder-1.40-5.i386.rpm
```

Second, copy all RPM files in the distribution CDs to a directory on the host machine (alpha). In my case, that directory is /tmp/redhat/rpm. On the CDs, the binary RPM files are kept in /mnt/cdrom/RedHat/RPMS (assuming the CD is mounted on /mnt/cdrom). Those RPM files will be used by UMLBuilder.

Now create a directory for the project (/home/usenix in my case) and copy the kernel tarballs and the UML patches to it.

Building the Root File System

The root file system for a Linux system is usually hosted by a hard disk or a hard disk partition. It is more convenient, however, to use an ordinary file as the root file system for a UML machine. Linux uses a special device called the loopback device (not to be confused with the network loopback interface) to enable dealing with an ordinary file as if it were a block device, allowing a file to host a file system and a directory tree. Obtaining a root file system for our UML machine can be done by downloading a pre-built one from the UML home page, by creating it from scratch using basic Linux tools, or by using UMLBuilder. The approach chosen here is to rely on UMLBuilder.

Start by launching the UMLBuilder GUI from within the X window system launching and xterm and typing: UMLBuilder_gui. Press "next" and you'll be presented with a number of distributions to choose from. Choose RedHat 8.0. When asked about the location of the RPMs, enter /tmp/redhat/rpm.

Now, you'll be presented with a selection of packages to choose from. Choose "Various Network Server Daemons (network-server)" and press "next." In the file system settings window, specify the mount point for device udb0 as "/", the size of the file system as 400MB, and the file system type as ext2. Leave the filename as rootfs. In the Miscellaneous Settings window, set the hostname as ghost and the IP address as 10.20.0.1. Set root's password and press "next." You'll be asked about the location where the files pertaining to the UML instance should be stored. Enter /home/usenix/ghost and press "next." All settings will be displayed so that you can review them. If all are correct, proceed by pressing the relevant button. The creation of rootfs will start and might take a quite long time to finish.

After the building completes, the directory /home/usenix/ghost should contain, among other files, the rootfs file and a shell script called "control" that facilitates launching the UML machine. Edit the control script so that the variables net and hostiface are set to the values indicated below:

```
net="eth0=tuntap,,,10.20.0.254"
hostiface="tap0"
```

The second line sets the name of the host machine's interface linking it to the UML machine to tap0 (creates an interface on

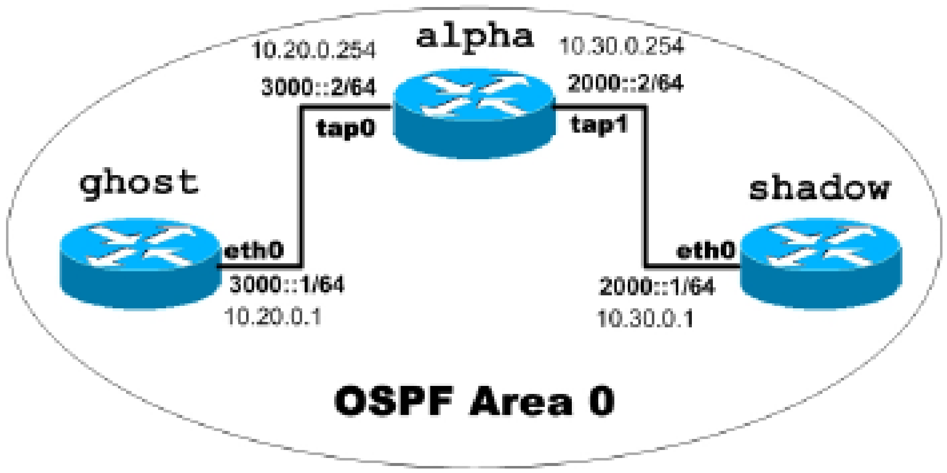


Figure 1

alpha called tap0). The first line sets the IP address of that interface to 10.20.0.254. Although our interest is IPv6 configuration, we'll configure IPv4 addresses for the time being so that we can use them to test connectivity even before we start our IPv6 configuration (see Figure 1 for the network's topology).

At this stage you can launch the UML machine by typing (from an xterm): /home/usenix/ghost/control start.

To shut down the UML machine, just type, as root: shutdown -h now.

Keep in mind that the UML kernel used to boot the machine is actually the binary /usr/bin/linux installed as part of the user_mode_linux package (remember that the UML kernel is just another user-land program). In the following section, we'll create our own UML kernels.

Patching and Compiling a Standard Kernel

Start by uncompressing the kernel source tree and extracting the files in the archive:

```
bzip2 -d linux-2.4.20.tar.bz2
tar -xvf linux-2.4.20.tar
```

The directory /home/usenix/linux-2.4.20 contains the kernel source tree.

Next, copy the patch to the kernel source tree uppermost directory and apply the patch:

```
cp uml-patch-2.4.20-6.bz2 /home/usenix/linux-2.4.20
cd /home/usenix/linux-2.4.20
patch -p1 < uml-patch-2.4.20-6
```

Now we have a UML-patched kernel, and we can start the kernel configuration and compilation.

Launch the kernel configuration GUI:

```
make xconfig ARCH=um
```

(ARCH=um sets the architecture to UML instead of the default x86 architecture.)

Under Network Options, enable IPv6 support as a module (it is, of course equally possible to enable IPv6 support as an integral part of the kernel rather than a module). Disable all unneeded drivers, protocols, and features.

Now, create the prerequisite object files:

```
make dep ARCH=um
```

Now, create the UML kernel itself:

```
make linux ARCH=um
```

If the compilation succeeds, you'll find an executable called linux in the directory /home/usenix/linux-2.4.20/. This is our newly created UML kernel.

If you choose to configure some parts of the kernel as modules, then you need to compile the modules and install them in rootfs.

To compile the modules for the UML kernel, type:

```
make modules ARCH=um
```

To install the modules, start by making sure that the UML machine is shut down and then mount the rootfs file:

```
mkdir /mnt/rootfs
mount -o loop /home/usenix/ghost/rootfs /mnt/rootfs
```

The commands typed above mount the rootfs file system on /mnt/rootfs.

To install the modules in rootfs, type:

```
make modules_install INSTALL_MOD_PATH=/mnt/rootfs/
```

The name of the directory containing the modules must match the kernel's version, therefore:

```
mv /mnt/rootfs/lib/modules/2.4.20
/mnt/rootfs/lib/modules/2.4.20-6um
umount /mnt/rootfs
```

At this stage we have the root file system built, an IPv6-enabled kernel built, and the associated kernel modules built and installed.

Place your newly created UML kernel in /usr/bin:

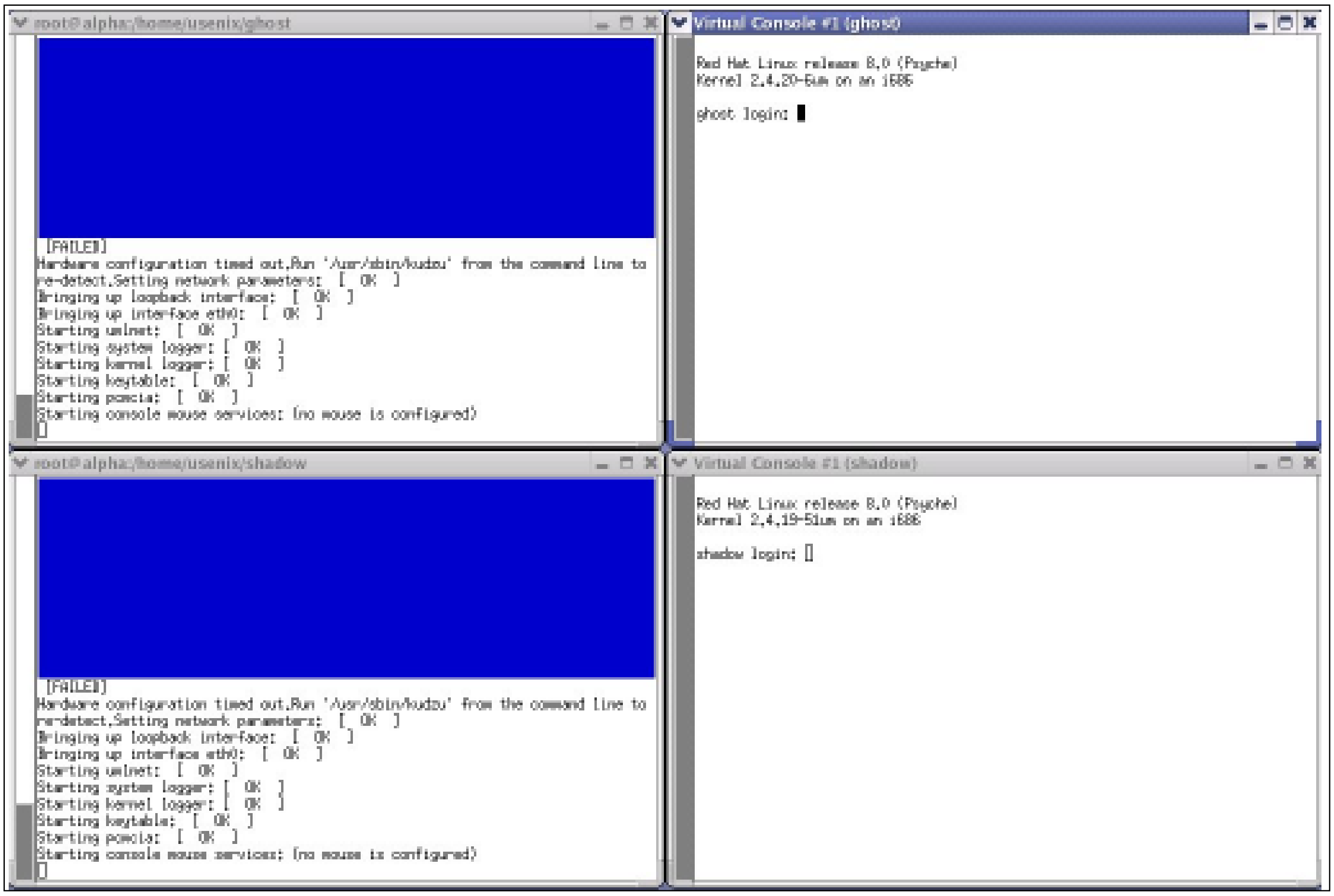
```
mv /home/usenix/linux-2.4.20/linux /usr/bin/ghost
```

Edit the control script to launch the new kernel by changing the line (line 126):

```
exec $linux $initrd umid="$name" $fs $swap
              mem=$memsize $net $ux $args "$@"
```

to{

```
exec ghost $initrd umid="$name" $fs $swap
              mem=$memsize $net $ux $args "$@"
```


root@alpha:/home/usenix/ghost
Virtual Console #1 (ghost)
root@alpha:/home/usenix/shadow
Virtual Console #1 (shadow)

```
 [FAILED]
Hardware configuration timed out.Run '/usr/sbin/kudzu' from the command line to
re-detect.Setting network parameters:  [  OK  ]
Bringing up loopback interface:  [  OK  ]
Bringing up interface eth0:  [  OK  ]
Starting umlnet:  [  OK  ]
Starting system logger: [  OK  ]
Starting kernel logger: [  OK  ]
Starting keytable:  [  OK  ]
Starting pcmcia:  [  OK  ]
Starting console mouse services: (no mouse is configured)
```

```
Red Hat Linux release 8.0 (Psyche)
Kernel 2.4.20-6um on an i686

ghost login: 
```

```
 [FAILED]
Hardware configuration timed out.Run '/usr/sbin/kudzu' from the command line to
re-detect.Setting network parameters:  [  OK  ]
Bringing up loopback interface:  [  OK  ]
Bringing up interface eth0:  [  OK  ]
Starting umlnet:  [  OK  ]
Starting system logger: [  OK  ]
Starting kernel logger: [  OK  ]
Starting keytable:  [  OK  ]
Starting pcmcia:  [  OK  ]
Starting console mouse services: (no mouse is configured)
```

```
Red Hat Linux release 8.0 (Psyche)
Kernel 2.4.19-51um on an i686

shadow login: 
```

Figure 2

Patching and Compiling a USAGI Kernel

To create the USAGI-based UML machine, we'll use the same rootfs. So start by copying the ghost UML directory and editing the control script:

```
cp -R /home/usenix/ghost /home/usenix/shadow
```

In shadow's control script make the following changes:

```
net="eth0=tuntap,,,10.30.0.254"
hostiface="tap1"
```

Change the line (line 126):

```
exec $linux $initrd umid="$name" $fs $swap
                mem=$memsize $net $ux $args "$@"
```

to:

```
exec shadow $initrd umid="$name" $fs $swap
                mem=$memsize $net $ux $args "$@"
```

Now, patch and compile the USAGI kernel:

```
bzip2 -d usagi-linux24-stable-20021007.tar.bz2
tar -xvf usagi-linux24-stable-20021007.tar
cd usagi/
```

Specify the major kernel version USAGI is to be compiled for by typing:

```
make prepare TARGET=linux24
```

Now copy and apply the patch:

```
cp uml-patch-2.4.19-51.bz2
/home/usenix/usagi/kernel/linux24
cd /home/usenix/usagi/kernel/linux24
patch -p1 < cp uml-patch-2.4.19-51
```

Now configure and compile the UML kernel just as we did with the standard kernel:

```
make xconfig ARCH=um
make dep ARCH=um
make linux ARCH=um
make modules ARCH=um
```

With the USAGI kernel we have chosen to compile IPv6 support as part of the kernel rather than as a module.

Place your newly created USAGI UML kernel in /usr/bin:

```
mv /home/usenix/USAGI/usagi/kernel/linux24/linux
                /usr/bin/shadow
```

Install the USAGI kernel modules, just as we did with the standard kernel:

```
mount -o loop /home/usenix/shadow/rootfs /mnt/rootfs
make modules_install INSTALL_MOD_PATH=/mnt/rootfs/
umount /mnt/rootfs
```

At this stage, the creation of our IPv6 UML machines is completed and it is now time to bring life to the machines.

From an xterm:

```
cd /home/usenix/ghost
./control ghost
```

After ghost completes booting, from another xterm boot shadow:

```
cd /home/usenix/shadow
./control shadow
```

The virtual consoles of both UML machines should be appearing on your screen (see Figure 2).

IPv6 Addressing, Routing, and Connectivity Testing

Addressing

We are ready now to enter the world of IPv6 networking. We'll start by assigning our routers their network addresses. On ghost, load the IPv6 module:

```
insmod ipv6
```

then assign the IPv6 address:

```
ifconfig eth0 add 3000::1/64
```

which, on shadow, should be:

```
ifconfig eth0 add 2000::1/64
```

On alpha, load the IPv6 module if you have IPv6 support compiled as a module:

```
insmod ipv6
ifconfig tap0 add 3000::2/64
ifconfig tap1 add 2000::2/64
```

(See Figure 1.)

Routing

Now that we have created two UML hosts and assigned them IPv6 addresses, we are ready to configure Zebra to perform OSPFv3 dynamic routing. GNU Zebra provides an implementation for a number of IPv4 and IPv6 dynamic routing protocols with an interface similar to Cisco's CLI. The Zebra routing system consists of a kernel routing table manager daemon called zebra and a number of routing daemons, each implementing an IPv4 or an IPv6 routing protocol. The manager daemon zebra receives input from the protocol-specific routing daemons and modifies the kernel routing table accordingly. Examples of routing daemons that are part of Zebra are ospfd and ospf6d: ospfd is the OSPFv2 routing daemon, which performs OSPF routing for IPv4, and ospf6d is the OSPFv3 routing daemon, which performs OSPF routing for IPv6. Note that although the OSPF version designed to work with IPv6 is OSPF version 3, the zebra OSPFv3 daemon is called ospf6d. The "6" here indicates the IP version rather than the OSPF version. There can be more than one protocol-specific routing daemon running on the same host. A machine operating in a dual-stack environment (a network in which IPv4 and IPv6 coexist) can, for example, run ospfd and ospf6d simultaneously.

Since configuring and running zebra is a prerequisite for running any protocol-specific daemon, we'll start with the creation of the zebra daemon configuration file:

On ghost:

```
vi /etc/zebra/zebra.conf

!
! Setting the hostname for the zebra daemon
!
hostname ghostz
!
! Setting the password for the zebra daemon
!
password zebra
!
! Setting the enable password for the zebra daemon
!
enable password zebra
```

Note that the bangs (!) are used to add comments to the configuration file.

On shadow:

```
hostname shadowz
password zebra
enable password zebra
```

On alpha (the host machine):

```
hostname alphaz
password zebra
enable password zebra
```

The next step is configuring the OSPFv3 daemon.

Since our virtual network is a rather small one, all our IPv6 routers will be configured in the same area (area 0.0.0.0 or area 0). Just like OSPFv2, OSPFv3 assigns each router a unique 32-bit router ID. In our virtual network, we'll assign ghost the ID 0.0.0.2, shadow the ID 0.0.0.3, and alpha the ID 0.0.0.1.

To configure the OSPFv3 daemon on ghost:

```
vi /etc/zebra/ospf6d.conf

!
hostname ghostz
password zebra
enable password zebra
!
router ospf6
      router-id 0.0.0.2
      redistribute static
      interface eth0 area 0.0.0.0
!
```

On shadow:

```
vi /etc/zebra/ospf6d.conf
```

```
!
hostname shadowz
password zebra
enable password zebra
!
router ospf6
      router-id 0.0.0.3
      redistribute static
      interface eth0 area 0.0.0.0
!
```

On alpha:

```
vi /etc/zebra/ospf6d.conf
```

```
!
hostname alphaz
password zebra
enable password zebra
!
router ospf6
      router-id 0.0.0.1
      redistribute static
      redistribute connected
      interface tap0 area 0.0.0.0
      interface tap1 area 0.0.0.0
!
```

Note that we added the "redistribute connected" statement so that alpha tells ghost about shadow (which is directly connected) and tells ghost about shadow using OSPFv3.

Although we have chosen to perform the configuration through editing the configuration files, we could have established a Telnet session to the daemons and configured routing using the Cisco-like interface. This could be done by typing:

```
telnet localhost zebra
```

or

```
telnet localhost 2601
```

for zebra and

```
telnet localhost ospf6d
```

or

```
telnet localhost 2606
```

for ospf6d. Now to start OSPF routing, type the following in all three machines:

```
/etc/init.d/zebra start
/etc/init.d/ospf6d start
```

Connectivity Testing

To test the connectivity, ping shadow from ghost:

```
ping6 2000::1
```

Successful pinging indicates that routing is working without problems and that we have successfully completed the construction of our IPv6 lab!

To display the IPv6 routing table, type:

```
route -A inet6
```

or:

```
ip -6 route show
```

You'll notice that the routes obtained through dynamic routing have a higher metric than static and directly connected routes.

You can also capture the IPv6 traffic using:

```
tcpdump -qtfn ip6
```

Increasing Topological Complexity

The IPv6 lab we have constructed is a simple one, with only three routers and one OSPF area, but it illustrates the basic procedures needed for virtual UML networking. Creating more complex networks can be done using nested UML machines and the uml_switch daemon. A nested UML machine is a UML kernel launched from within a UML machine. The uml_switch is a daemon that simulates physical switches and can be used to connect a number of UML machines running on the same physical hosts. Information about nesting and the uml_switch can be found in the UML Kernel Home Page (*http://user-mode-linux.sourceforge.net*).

Conclusion

This article shows how to use User-mode Linux (UML) to build a simple IPv6 lab on a laptop (a lab in the lap). OSPFv3 was enabled to perform dynamic routing among the three IPv6 routers in our virtual network. Virtual UML networking is particularly valuable when it comes to studying and experimenting with new technologies like IPv6 when not enough test machines are available. In addition, UML virtual networking is more cost-effective, takes much less space, and allows rapid prototyping and experimentation portability.