Installing and Using Xen

Xen Hypervisor 4.1.1

Giorgos Kappes (geokapp@gmail.com)

Last Update – 18 March 2012

Contents

1. Introduction

Virtualization is a broad term of computer systems that refers to an abstraction mechanism which hides the physical characteristics of certain computational resources in order to simplify the way in which other systems, applications or end users interact with them. Thus, virtualization enables sharing the resources of a computer system in multiple execution environments.

The concept of virtualization is not new. It has its roots in the mid 1960's, when it was used by IBM as a method for logical partitioning of large centralized systems (mainframes) into separate virtual machines. The virtual computers distributed to users of the system, allowing each user to work in an isolated environment without affecting other users. For this sharing to be possible IBM introduced a new feature called Virtual Machine Monitor (VMM).

The Virtual Machine Monitor is a software layer that is placed on top of the hardware layer and has direct access to hardware resources. The main objective of Virtual Machine Monitor is to manage and allocate system resources to one or more virtual machines in order to make virtualization possible.

1.1. An overview of virtualization architectures

Virtualization follows various approaches that are directly related to the architecture of the virtual machine monitor.

In the hosted architecture the VMM runs as an application on the host operating system and relies on it for resource management, system memory devices and drivers. It is also responsible for starting, stopping and managing each virtual machine and also controls access of virtual machines to physical system resources (figure 1, a). A virtualization system that follows this approach is the VMware Workstation.

In the autonomous architecture (figure 1, b), the VMM is placed directly above the material. Thus, it is responsible for managing system resources and their allocation to different virtual machines. This architecture is more efficient because the VMM has direct access to system resources. An example of an autonomous architecture is Xen.

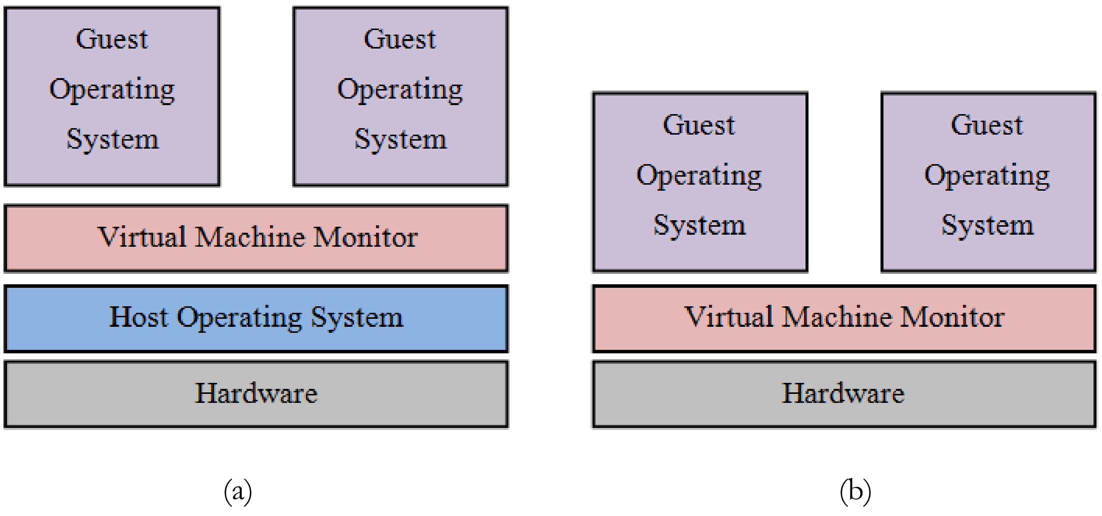


Figure 1 – Virtual Machine Monitor architecture. (a) Hosted, (b) autonomous

The guest operating systems runs with limited privileges and doesn't have direct access to hardware. Thus, it is difficult to virtualize some critical operating system instructions because their implementation requires higher privileges. To make it easier to understand the problem we will describe briefly how the x86 processor architecture separates the operating system from simple applications. The x86 processor architecture includes four privilege levels (rings). The operating system kernel is running at level 0, which has the most privileges. This level provides complete control on system hardware. Simple applications are running on level 3, which has limited privileges. Levels 1 and 2 are not used. Thus, in a virtualization environment the guest operating systems is running on the level of applications. For this reason some critical instructions that require more rights, they cannot be virtualized. Two approaches were followed to solve this problem: full virtualization and paravirtualization.

Full virtualization (figure 2) provides a virtual environment that simulates the real hardware. Specifically, each virtual machine is provided with all the services of the real system, such as full command set of the real processor, virtual appliances and virtual memory management. The major difference from other virtualization techniques is that the operating system does not understand that it runs in a virtual environment. So, any software that is capable to run in the real system can run without changes in the virtualized environment. In order the execution of critical instructions to be possible, a technique known as binary translation is used. According to this approach, the software is patched while it runs, for example the critical instructions that cannot run in the virtual environment are replaced by different instructions that can run safely. However, continuous scanning and emulation of critical instructions reduces performance. Some examples of full virtualization systems are the VMware Workstation and VirtualBox.

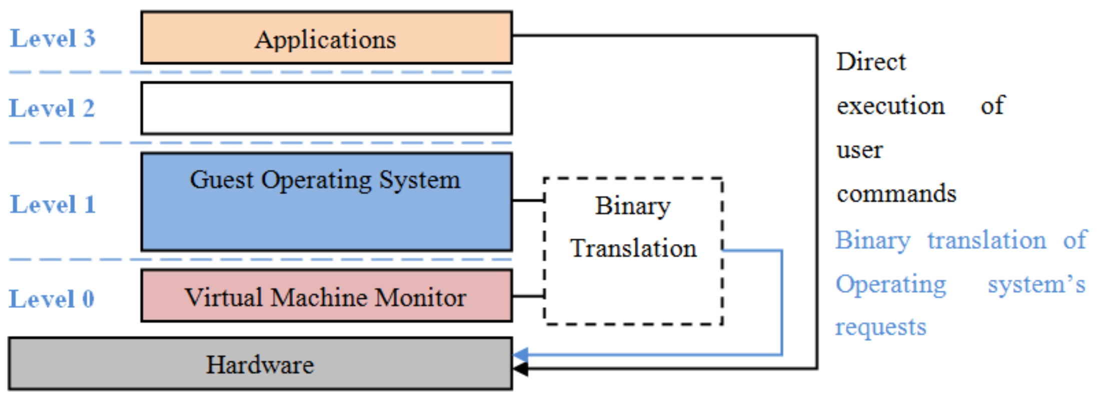


Figure 2 – Full virtualization

On the other hand, paravirtualization (figure 3) provides to the virtual machines a software interface that is similar but not identical to that of the real system. The main purpose of paravirtualization is to reduce the proportion of time spent in performing critical patches on the guest's unsafe instructions. This is achieved by modifying the client software so it can communicate with the VMM, which run at ring 0 and has direct access to hardware. So, when you need to perform a critical instruction, the guest operating system communicates directly with the VMM and executes. As a consequence, the guest operating system must be altered slightly in order to run in a paravirtualized environment. Examples of this technique are the Xen and Denali.

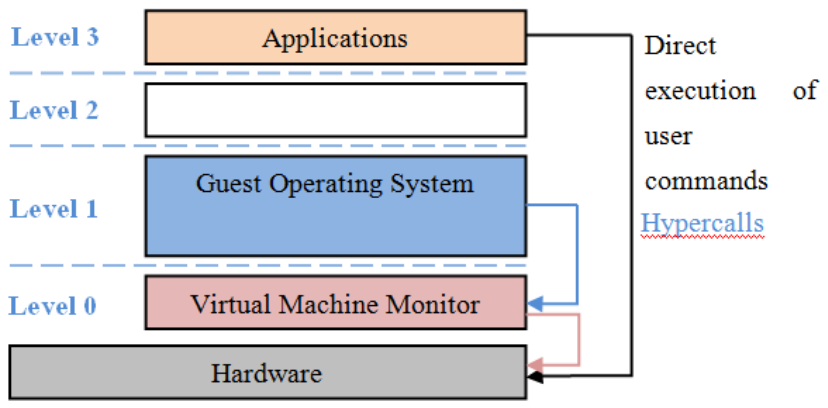


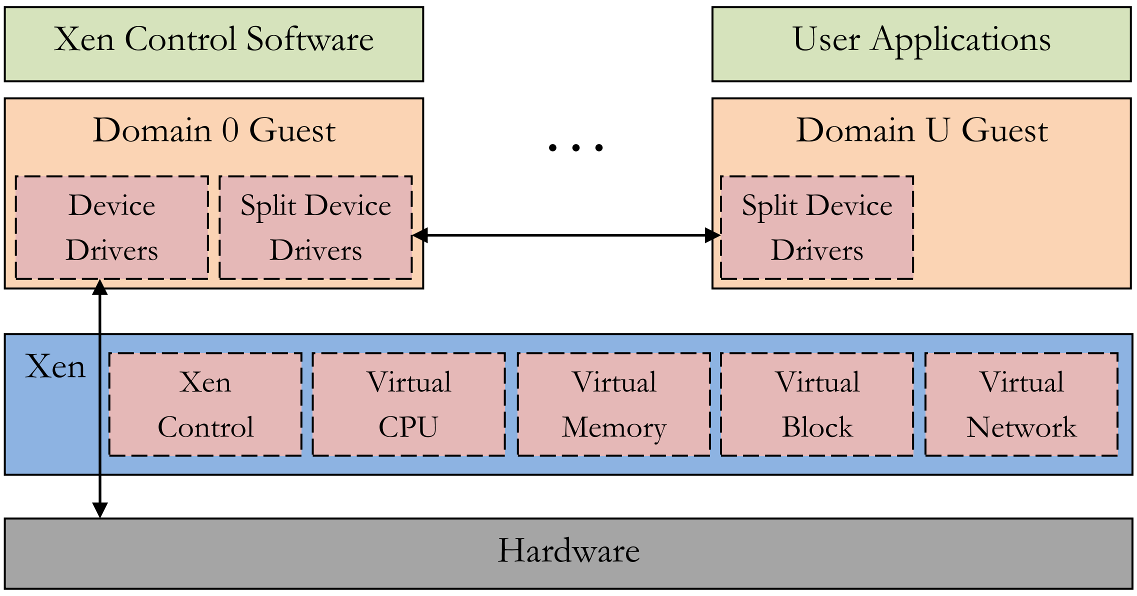
Figure 3 – Paravirtualization

As the benefits of virtualization are tremendous, manufacturers of processors have reviewed the instruction set of their products by making them friendlier to virtualization. Thus, the problem described above can be solved directly using the new instruction set. The main idea behind this is to introduce a new privilege level, called the level -1 below the level 0. The VMM can run on this new level. By introducing this new level, the guest operating systems can run at level 0 and the hardware requests they perform can be captured directly from the system.

1.2. The way of Xen

In 2003 a team of researchers at the University of Cambridge released the world's most advanced hypervisor: Xen. At first, the team developed a multimedia operating system called 'Nemesis'. This introduced the seed of what would become Xen: it had a domain switcher, which was a very low-level, very thin virtual machine layer that provided resource guarantees to the domains above it. Later, the developers needed a Virtual Machine Monitor (VMM) that would provide strong resource isolation between users running virtual machines on the physical machine, and most importantly, would provide isolation without a noticeable drop in performance. Having failed to find anything that came close to their criteria, they built their own.

Xen is an Open-Source hypervisor that creates and manages domains for multiple virtual machines. It uses paravirtualization, where the guest operating systems are aware that they are running in a virtualized environment. With paravirtualization, the guest operating system is modified to make special calls (hypercalls) to the hypervisor for privileged operations, instead of the regular system calls in a traditional unmodified operating system. The applications in the guest operating system's domain, however, remain unmodified.

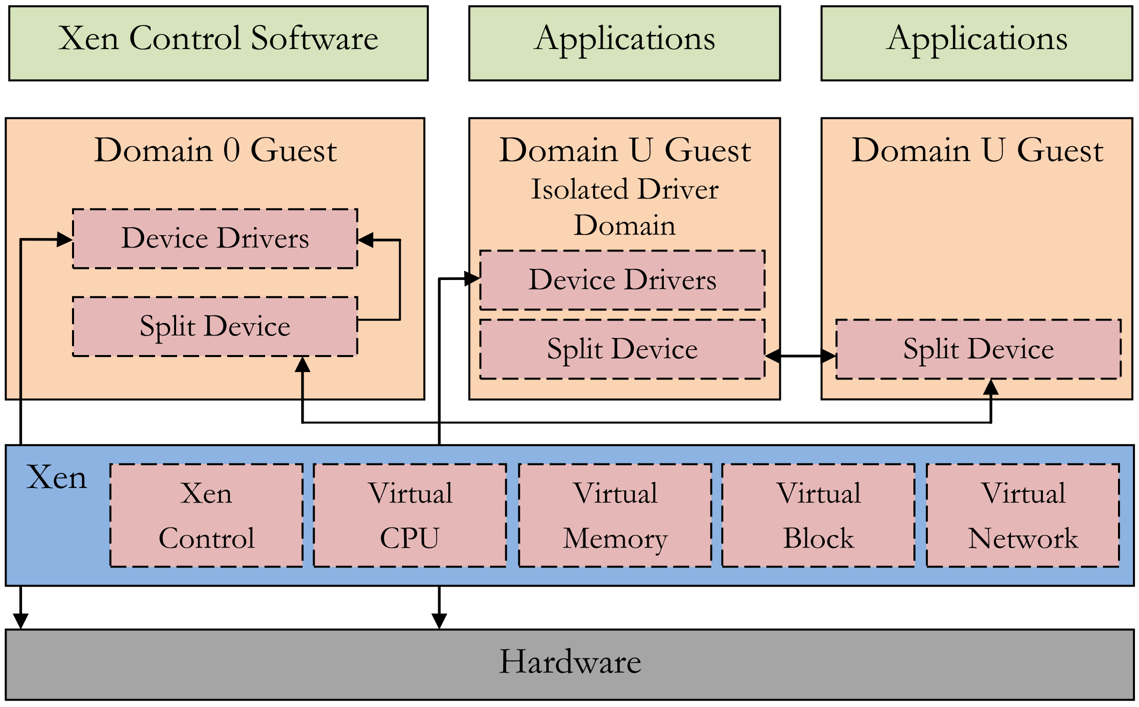


(Figure 4 – A basic Xen configuration)

Xen lies between the hardware and the operating system allowing multiple virtual machines to run simultaneously on a single physical system. Xen does not support any devices natively. Instead, it provides a mechanism by which a guest operating system can be given direct access to a physical device. Thus, the guest operating system can use an existing device driver. By separating the guests from the hardware, the Xen hypervisor is able to run multiple operating systems securely and independently.

As mentioned before, Xen provides an environment where multiple guest operating systems can run. Each gust runs inside its own environment called 'domain'. In Xen not all guests are created equal, one in particular is significantly more equal than the others. This one which is called domain0 (**dom0**) is the first guest to run when the physical system boots and has elevated privileges. On the other hand, the other domains are referred as DomainU (**domU**) (Figure 4).

Domain0 is very important to a Xen system. Xen itself doesn't include any device drivers. All device drivers are provided and handled by Domain0 which runs at a higher privilege level than the other guests and thus can access the hardware. Except from accessing the hardware, Domain0 is also responsible for handling administrative tasks, for example starting and stopping DomainU guests. We can say that Domain0 provides the user interface to the Xen hypervisor. For security reasons, it is a good practice to do as little in Domain0 as possible. An exploit on Domain0 can harm the whole system. Thus most work should be done in DomainU guests.



(Figure 5 – Driver isolation in Xen)

DomainU is an unprivileged domain and typically is not allowed to perform any hypercalls that directly access the hardware. Instead, a domU guest implements a front end of some split device drivers. Split device drivers are not normal drivers. Their job is to put the request from the user application into some shared memory. The other half of the split driver which runs on dom0 reads the request from the shared memory and passes it to the real device driver. Unlike dom0 guests, you can have an arbitrary number of domU guests on a single system.

As mentioned above, all the hardware is controlled in Domain0. However this configuration hides some dangers. Specifically, if a device driver contains bugs, it can crash the Domain0 kernel, which in turn can bring down all the guests. Thus, it's a good approach to isolate a device driver in its own domain (Figure 5).

2. Xen installation

This section describes how to install Xen 4.1.1 from source. Alternatively, there may be pre-built packages available as part of your operating system distribution.

2.1. Preparation

In order to successfully build and run Xen we need to download and install some tools and utilities. To install them in Debian Squeeze we run the following command as root:

```
root@dom0$ apt-get install bcc bin86 gawk bridge-utils iproute libcurl3
libcurl4-openssl-dev bzip2 module-init-tools transfig tgif texinfo texlive-latex-
base texlive-latex-recommended texlive-fonts-extra texlive-fonts-recommended
pciutils-dev mercurial build-essential make gcc libc6-dev zlib1g-dev python
python-dev python-twisted libncurses5-dev patch libvncserver-dev libsdl-dev
libjpeg62-dev iasl libbz2-dev e2fslibs-dev git-core uuid-dev ocaml libx11-dev
bison flex xz-utils ocaml-findlib
```

If you have a 64 bit version of Debian/Ubuntu you also need this additional package:

```
root@dom0$ apt-get install gcc-multilib
```

To be able to use other operating systems than Linux/ FreeBSD as a guest (e.g. Windows) our hardware must has support for VT (Virtualization Technology). Hardware assisted virtualization offers new instructions to support direct calls by a paravirtualized guest/driver into the hypervisor, typically used for I/O or other hypercalls. The main idea behind this is to introduce a new privilege level, called the level -1 below the level 0. The VMM can run on this new level. By introducing this new level, the guest operating systems can run at level 0 without any modifications and the hardware requests they perform can be captured directly from the system. To check if our Intel based system supports this feature we can run:

```
root@dom0$ cat /proc/cpuinfo | grep vmx
```

For AMD based processors, we can run the following command:

```
root@dom0$ cat /proc/cpuinfo | grep svm
```

2.2. Installation

We are now ready to download and install Xen. In this guide we will use the version 4.1.1 of Xen. To get the source code we run:

```
user@dom0$ wget http://bits.xensource.com/oss-xen/release/4.1.1/xen-
4.1.1.tar.gz
```

Now we can extract it and build it:

```
user@dom0$ tar xvf xen-4.1.1.tar.gz
user@dom0$ cd xen-4.1.1
user@dom0$ make xen
user@dom0$ make tools
user@dom0$ make stubdom
```

After the build has completed we should have a top-level directory called 'dist' in which all resulting targets will be placed. To install them on our system we run:

```
root@dom0$ make install-xen
root@dom0$ make install-tools PYTHON_PREFIX_ARG=
root@dom0$ make install-stubdom
```

At this point Xen and its utilities are installed on our system. We should have the following files in '/boot' directory:

```
/boot/xen.gz
/boot/xen-4.gz
/boot/xen-4.1.gz
/boot/xen-4.1.1.gz
```

Now, in the file '/etc/xen/xend-config.sxp' we enable the following option:

```
(xend-unix-server yes)
```

And finally, to enable automatic start of Xen related services on system startup we run:

```
root@dom0$ update-rc.d xencommons defaults 19 18
root@dom0$ update-rc.d xend defaults 20 21
root@dom0$ update-rc.d xendomains defaults 21 20
root@dom0$ update-rc.d xen-watchdog defaults 22 23
```

3. Creating and installing a dom0 kernel

At this point Xen is built and installed on our system but it is not ready for use, as we don't have a dom0 kernel yet. The kernel with which our system booted will not work with Xen. So, we are going to compile a dom0 kernel.

There are two different types of Xen dom0 capable kernels available today:

- **pvops** (paravirt_ops) kernels, featuring new rewritten Xen support based on the upstream (kernel.org) Linux pvops framework. This work has been included in upstream kernel.org kernel since Linux 2.6.37. Pvops is a piece of Linux kernel infrastructure to allow it to run paravirtualized on a hypervisor. This feature allows us to build a single Linux kernel binary which will either boot native on bare hardware, or boot fully paravirtualized as a Xen dom0 or domU.
- **xenlinux** kernels based on the "old" patches originally for Linux 2.6.18. These Xenlinux patches won't be integrated to upstream Linux.

In this guide we will use the latest stable Linux kernel which is 3.0.4 and has build-in support for pvops. The first thing to do is to grab the kernel from kernel.org:

```
user@dom0$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-
3.0.4.tar.gz
user@dom0$ tar xvf linux-3.0.4.tar.gz
```

Now, we can configure our new kernel by running:

```
user@dom0$ make menuconfig
```

We must be sure that we have a correct Processor family set in the kernel configuration. Xen dom0 options won't show up at all if we have too old CPU selected. If we are building a 32 bit version of the kernel we need to enable PAE support:

```
Processor type and features →
        High memory support (64GB)
        PAE (Physical Address Extension) Support - enabled
```

Note that PAE is not needed for 64 bit kernels. Also, if we are building a 32 bit kernel we need to set the option 'CONFIG_HIGHPTE=n':

```
 Processor type and features →
        Allocate 2nd-level pagetables from highmem - disabled
```

Furthermore, Xen Dom0 support depends on ACPI support on both 32 and 64 bits versions of the Linux kernel. Thus, we must enable ACPI support during configuration:

```
 ACPI (Advanced Configuration and Power Interface) Support -
enabled
```

Below, is a list of options needed to compile Linux kernel with dom0 support:

- CONFIG_ACPI_PROCFS=y
- CONFIG_XEN=y
- CONFIG_XEN_MAX_DOMAIN_MEMORY=32
- CONFIG_XEN_SAVE_RESTORE=y
- CONFIG_XEN_DOM0=y
- CONFIG_XEN_PRIVILEGED_GUEST=y
- CONFIG_XEN_PCI=y
- CONFIG_PCI_XEN=y
- CONFIG_XEN_BLKDEV_FRONTEND=y
- CONFIG_XEN_NETDEV_FRONTEND=y
- CONFIG_XEN_KBDDEV_FRONTEND=y
- CONFIG_HVC_XEN=y
- CONFIG_XEN_FBDEV_FRONTEND=y
- CONFIG_XEN_BALLOON=y
- CONFIG_XEN_SCRUB_PAGES=y
- CONFIG_XEN_DEV_EVTCHN=y
- CONFIG_XEN_GNTDEV=y
- CONFIG_XEN_BACKEND=y
- CONFIG_XEN_BLKDEV_BACKEND=y
- CONFIG_XEN_NETDEV_BACKEND=y
- CONFIG_XENFS=y
- CONFIG_XEN_COMPAT_XENFS=y
- CONFIG_XEN_XENBUS_FRONTEND=y
- CONFIG_XEN_PCIDEV_FRONTEND=y

To enable these options we can select the following additional fields during configuration:

```
Processor type and features →
        Paravirtualized guest support [y] →
                Xen guest support – enabled

Bus oprions (PCI etc.)→
```

```
          Xen PCI frontend – enabled

Device Drivers →
          Block Devices [*] →
                    Xen virtual block device support – enabled
                    Block-device backend driver – enabled
          Network device support [*] →
                    Xen network device frontend driver – enabled
                    Xen backend network device – enabled
          Input device support →
                    Miscellaneous devices →
                              Xen virtual keyboard and mouse support –
                              enabled
          Character devices →
                    Xen Hypervisor console support – enabled

          Xen driver support →
                    Xen memory balloon driver – enabled
                    Scrub pages before returning them to system –
                    enabled
                    Xen /dev/xen/evtchn device Backend driver support
                    – enabled
                    Xen filesystem – enabled
                    Create compatibility mount point /proc/xen –
                    enabled
                    Create xen entries under /sys/hypervisor – enabled
                    userspace grant access device driver – enabled
                    User-space grant reference allocator driver – enabled
                    xen platform pci device driver – enabled
```

Now, we can build and install the kernel:

```
user@dom0$ make
root@dom0$ make modules_install
root@dom0$ make install
root@dom0$ cd /boot
root@dom0$ mkinitramfs -o initrd.img-3.0.4 3.0.4
root@dom0$ update-grub
```

We can now reboot our system to the Xen enabled 3.04 Linux kernel. To verify that Xen is running we can do the following:

```
root@dom0$ cat /proc/xen/capabilities
```

The output must be same as the following line:

```
control_d
```

This tells us that we have booted in a Xen control domain (dom0). Also, to verify that our Xen environment is working properly we can run:

```
root@dom0$ xm info
root@dom0$ xm list
```

If the output of these commands is the expected we have successfully installed Xen to our system. For example, 'xm list' command should list our dom0 domain:

```
Name                    ID   Mem    VCPUs     State   Time(s)
Domain-0                0    1895   2         r-----  419.0
```

A final step, before we create a domU guest domain, is to install the blktap driver. Blktap is used to provide a paravirtualized and high performance disk I/O interface to virtual block devices [9]. Its current main application is to replace the common loopback driver for file-based images. Since version 3.0 of the Linux kernel, the bkltap driver is not included in the kernel and it's now implemented completely in user space. In order to install it we have to run the following commands:

```
root@dom0$ apt-get install dkms
# For amd64 CPU architecture:
root@dom0$ wget http://launchpadlibrarian.net/87540571/blktap-
dkms_2.0.91-1_amd64.deb
root@dom0$ dpkg –i blktap-dkms_2.0.91-1_amd64.deb
# For x86 CPU architecture:
root@dom0$ https://launchpad.net/ubuntu/+source/blktap-dkms/2.0.91-
1/+build/3012200/+files/blktap-dkms_2.0.91-1_i386.deb
root@dom0$ dpkg –i blktap-dkms_2.0.91-1_i386.deb
```

After the above steps, we append the '/etc/modules' file with the following lines, in order for the blktap kernel module to be loaded automatically when the kernel boots and finally we reboot our system:

```
# blktap (with FUSE) driver
blktap
```

4. Creating a domU guest

In this section we will explain how to create a new Xen unprivileged guest (domU). A Xen guest can be one of the following two types:

- HVM Guest – Fully virtualized guest by using hardware assisted virtualization. The guest's kernel needs no modifications but our CPU must include virtualization technology to support hardware assisted virtualization.
- PV Guest – Paravirtualized guest. The guest's kernel needs some modifications.

4.1. Creating a PV domU Linux guest

The first thing to do when creating a domU Linux guest is to decide what kernel we will use. On the one hand, we can use the existing dom0 kernel. However, because the dom0 kernel is full of unwanted stuff for a guest, like backend drivers we can build a new lightweight kernel for a domU guest. The next step after deciding what kernel to use is to create and configure the root file system for our domU guest. The guest's root file system can reside in a file which acts as a virtual disk, in a physical partition or even in a remote NFS server. Finally, the last step is to create a configuration file to inform Xen about our new guest.

4.1.1. Building a domU kernel

As we said in section 4, there are two different types of Xen dom0 capable kernels available today: pvops and xenlinux. Pvops support for a domU guest has been in mainline Linux kernel since 2.6.23. In this guide we will use the latest stable Linux kernel which is 3.0.4. The first thing to do is to grab the kernel from kernel.org:

```
user@dom0$ cd /home/giorgos/xen
user@dom0$ wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-
3.0.4.tar.gz
user@dom0$ tar xvf linux-3.0.4.tar.gz
user@dom0$ cd linux-3.0.4
```

Now, we can configure our new kernel by running:

```
user@dom0$ make menuconfig
```

We must be sure that we have a correct processor family set in the kernel configuration. Xen options won't show up at all if we have too old CPU selected. If we are building a 32 bit version of the kernel, we first need to enable PAE support:

```
Processor type and features →
        High memory support (64GB)
        PAE (Physical Address Extension) Support - enabled
```

PAE is not needed for 64 bit kernels. Also, if we are building a 32 bit kernel we need to set the option 'CONFIG_HIGHPTE=n':

```
 Processor type and features →
        Allocate 2nd-level pagetables from highmem - disabled
```

Below, is a list of options needed to compile Linux kernel with domU support. It's important to mention that we prefer to select all the options below as build in, not as modules:

- CONFIG_XEN=y
- CONFIG_PARAVIRT_GUEST=y
- CONFIG_PARAVIRT=y
- CONFIG_XEN_PVHVM=y
- CONFIG_XEN_MAX_DOMAIN_MEMORY=128
- CONFIG_XEN_SAVE_RESTORE=y
- CONFIG_PCI_XEN=y
- CONFIG_XEN_PCIDEV_FRONTEND=y
- CONFIG_XEN_BLKDEV_FRONTEND=y
- CONFIG_XEN_NETDEV_FRONTEND=y
- CONFIG_INPUT_XEN_KBDDEV_FRONTEND=y
- CONFIG_HVC_XEN=y
- CONFIG_XEN_FBDEV_FRONTEND=y
- CONFIG_XEN_DEV_EVTCHN=y
- CONFIG_XEN_XENBUS_FRONTEND=y

Now, we can build the kernel:

```
root@dom0$ make
```

When the build process is complete, we will have a file called 'vmlinux' on the top-level directory. We can copy this file to our working directory:

```
root@dom0$ cp vmlinux /home/giorgos/xen/vmlinux-3.0.4-domU
```

When we will create the guest's file system later, we will come back to install the necessary kernel modules.

4.1.2. Creating guest's file system

In order for our new domU kernel to be functional we have to supply it with a root file system. We will create a disk image which will contain the root file system for our domU guest. This image will act as a hard disk for our guest system. In order to create an empty disk image we will use the 'dd' (man 1 dd) utility:

```
user@dom0$ dd if=/dev/zero of=/home/giorgos/xen/domu1.img bs=1024K
count=1024
```

With the above command we created an empty disk image with 1GB free space. Now, we must initialize the file system inside of the image file. We will use ext4 as our root file system:

```
user@dom0$ /sbin/mkfs.ext4 /home/giorgos/xen/domu1.img
```

If we would like to provide a swap space to our guest we can create a second image and initialize it with the swap file system:

```
user@dom0$ dd if=/dev/zero of=/home/giorgos/xen/swap.img bs=1024K
count=256
user@dom0$ /sbin/mkfs.swap /home/giorgos/xen/swap.img
```

At this point we have two disk images, one formatted with ext4 and one formatted as swap space. The next step is to populate the file system with a copy of the host's root or with a base system of our favorite Linux distribution. Here we choose to install a Debian Squeeze base system. To achieve this, we first mount the disk image to a new directory on the host and then we use the 'debootstrap' (man 8 debootstrap) utility to populate the image with a Debian Squeeze base system:

```
root@dom0$ mount –o loop /home/giorgos/xen/domu1.img /mnt/
root@dom0$ debootstrap --arch i386 squeeze /mnt/
http://ftp.us.debian.org/debian
```

In the above example we chose i386 as the architecture of our new system. We can specify a different architecture by changing the argument after 'arch' parameter. However, it's important to select the same architecture with the kernel we built earlier.

Alternatively, we can populate the guest's root file system by copying from the host's root:

```
root@dom0$ cp –ax /{root,dev,var,etc,usr,bin,sbin,lib} /mnt
root@dom0$ mkdir /mnt/{proc,sys,home,tmp}
```

4.1.3. Guest system configuration

After populating the disk image we need to edit some files in order to have our guest system in a valid configuration:

a) /etc/fstab
The '/etc/fstab' file is a system configuration file that lists all available disks and disk partitions and indicates how they are to be initialized or otherwise integrated into the overall system's file system. Here we mount the virtual disk device '/dev/xvda1' on the root specifying 'ext4' as the file system and the virtual disk device '/dev/xvda2' for swap space. Also, we mount the proc file system on '/proc' directory:

```
/dev/xvda1    /          ext4   defaults     0     1
/dev/xvda2    none       swap   sw           0     0
proc          /proc      proc   defaults     0     0
```

b) /etc/hostname
The hostname file contains the host name of the guest system. We add the name:

```
domu1
```

c) /etc/network/interfaces
This file contains network interface configuration information for the ifup(8) and ifdown(8) commands. This is where you configure how your system is connected to the network. At this point we will add the loopback interface to this file. Later on this guide we will add some more network interfaces to provide a fully functional networking on the domU guest. So, we open this file with an editor and we add the following values:

```
# This file describes the network interfaces available on your
system
# and how to activate them. For more information, see
interfaces(5).

#
# The loopback network interface
#
auto lo
iface lo inet loopback
```

Note that the lines beginning with # are comments.

d) /etc/securetty

The '/etc/securetty' file allows you to specify which TTY devices the root user is allowed to login on. The '/etc/securetty' file is read by the login program usually '/bin/login'. Its format is a list of the tty devices' names allowed, and for all others that are commented out or do not appear in this file, root login is disallowed. We append this file with the following values:

```
hvc0
```

e) /dev/

The '/dev' directory contains the special device files for all the available devices. If the virtual block devices 'xvda1', 'xvda2' and the virtual console 'hvc0' does not exist there we have to create them manually. We run the following commands as root:

```
root@dom0$ chroot /mnt
root@dom0$ mknod /dev/xvda1 b 202 1
root@dom0$ mknod /dev/xvda2 b 202 2
root@dom0$ mknod /dev/hvc0 c 229 0
root@dom0$ chown root:disk /dev/xvda1 /dev/xvda2
root@dom0$ exit
```

f) /etc/inittab

The '/etc/inittab' file describes which processes are started at boot up and during normal operation. To login immediately after the boot messages comment out the following lines:

```
# 2:23:respawn:/sbin/getty 38400 tty2
# 3:23:respawn:/sbin/getty 38400 tty3
# 4:23:respawn:/sbin/getty 38400 tty4
# 5:23:respawn:/sbin/getty 38400 tty5
# 6:23:respawn:/sbin/getty 38400 tty6
```

And modify the line starting with '1:' as follows:

```
hvc0:2345:respawn:/sbin/agetty -L 9600 hvc0
```

Now, it's time to add a new user to the domU guest. To do this we run the following commands as root:

```
root@dom0$ chroot /mnt
root@dom0$ adduser xen
root@dom0$ exit
```

Instead of using the tool 'adduser', we can add a new user by hand, by adding an entry to '/etc/passwd' and '/etc/shadow' files. After adding the new user we can specify a password for the root user. We run the following commands as root:

```
root@dom0$ chroot /mnt
root@dom0$ passwd
root@dom0$ exit
```

Remember the kernel we compiled earlier in 4.1.1? Now it's time to install the kernel modules to the appropriate location. To do this we run the following:

```
root@dom0$ cd /home/giorgos/xen/linux-3.0.4
root@dom0$ make modules_install INSTALL_MOD_PATH=/mnt
```

Now, we can create an initramfs image for the guest kernel. To do this we run the following commands:

```
root@dom0$ cp /home/giorgos/xen/linux-3.0.4/.config /mnt/boot/config-
3.0.4
root@dom0$ chroot /mnt
root@dom0$ mkinitramfs -o initrd.img-3.0.4-domU 3.0.4
root@dom0$ exit
root@dom0$ mv /mnt/initrd.img-3.0.4-domU /home/giorgos/xen/.
```

At this point the root file system for our domU guest is ready. We can unmount it from '/mnt':

```
root@dom0$ umount /mnt
```

4.1.4. Creating a domain configuration file

Before we can boot the new domU guest, we must create a configuration file. Below, we provide an example file which you can use as a starting point:

```
#
#  domU Configuration file
#

#
#  Kernel & ramdisk
#
kernel = '/home/giorgos/xen/vmlinux-3.0.4-domU'
ramdisk = '/home/giorgos/xen/initrd.img-3.0.4-domU'

#
# Number of virtual CPUs
```

```
#
vcpus = '1'

#
# Amount of RAM
#
memory = '512'

#
#  Disk device(s).
#
root = '/dev/xvda1 ro'
disk = [
                'tap:aio:/home/giorgos/xen/domu1.img,xvda1,w',
                'tap:aio:/home/giorgos/xen/swap.img,w',
          ]

#
#  A name for your domain.
#
name = 'domu1'

#
#  Networking
#
#  Examples:
#  vif = [ 'mac=00:16:3e:00:00:11, bridge=xenbr0' ]
#  vif = [ '', 'bridge=xenbr1' ]
#  Active:
vif = [ '' ]

#  Set if you want dhcp to allocate the IP address.
#  dhcp = "dhcp"

#
#  Behaviour
#
on_poweroff = 'destroy'
on_reboot = 'restart'
on_crash = 'restart'

#
#  Extra kernel paramerers
#
extra = 'console=hvc0 xencons=tty'
```

You can find additional templates in '/etc/xen' directory. Let's explain the basic options we use above:

- **kernel:** Set this to the path of the domU kernel you compiled for use with Xen.
- **ramdisk:** Set this to the path of the initramfs you created for the above domU kernel.
- **memory:** Set this to the size of domain's memory in megabytes.

- **vcpus:** Set the number of virtual CPUs.
- **name:** Give a unique name to the new domain.
- **vif:** Define network interfaces. The syntax of this option is:

  vif = ['mac=MAC_ADDRESS', 'bridge=BRIDGE']

  where MAC_ADDRESS is a valid MAC address for the new virtual interface and BRIDGE is a valid BRIDGE on dom0. You can also specify only the bridge:

  vif = ['', 'bridge=BRIDGE']

  or let the system to choose the defaults:

  vif = ['']

- **disk:** Define the disk devices you want the domain to have access to and what you want them accessible as. If you want to export individual partitions from domain 0 to other domains use the following syntax:

  phy:UNAME,DEV,MODE

  UNAME is the device, DEV is the device name the domain will see, and MODE is 'r' for read-only, 'w' for read-write. For example:

  disk = ['phy:sda1,xvda1,w']

  Storage may also be exported from a file system image or a partitioned file system image as a file-based VBD with the following syntax:

  file:PATH,DEV,MODE

  PATH is the full path to the file system image, DEV is the device name the domain will see, and MODE is 'r' for read-only, 'w' for read-write. For example:

  disk = ['file:/home/giorgos/disks/root.img,xvda1,w']

  With the above syntax the support for file-backed VBDs is provided through the loopback driver. If you want to use the blktap driver instead, use the following configuration line:

  tap:aio:PATH,DEV,MODE
- **dhcp:** Uncomment the dhcp variable, so that the domain will receive its IP address from a DHCP server.
- **root:** Set the root device. The guest will use this device to mount its root file system.
- **extra:** Set extra kernel parameters (e.g. runlevels).

We name the above file 'domu1'. If we prefer this guest to start automatically when dom0 boots, we have to copy the guest's configuration file to the directory '/etc/auto/' or just create a link there:

```
root@dom0$ ln –s /etc/xen/domu1 /etc/xen/auto/domu1
```

In order to start the new domU guest we can type:

```
root@dom0$ xm create –c domu1
```

The '-c' switch causes xm to turn into the domain's console after creation. We should see the console boot messages from the new domain appearing in the terminal in which we typed the command, culminating in a login prompt.

4.2. Creating an HVM guest

Xen supports fully virtualized guest domains by using hardware assisted virtualization. Currently processors featuring the Intel Virtualization Technology (Intel-VT) or the AMD Virtualization Technology (AMD-V) are supported. For a way to see if your processor has virtualization technology look at the section 2.1 or run the following command if xend is running:

```
root@dom0$ xm dmesg | grep -i hvm
```

If your system does not include this support you can still use Xen in paravirtualization mode. You will not, however, be able to run unmodified operating systems such as Microsoft Windows as a Xen guest operating system.

In order to install an HVM guest some form of disk storage is needed to contain the guest operating system. We can use as storage a whole physical disk (e.g. '/dev/sda'), a physical disk partition (e.g. '/dev/sda1'), or a disk image file. Here we will follow the last approach. To create the disk image file for the HVM guest we use the 'dd' utility:

```
root@dom0$ dd if=/dev/zero of=/home/giorgos/xen/domuhvm.img
bs=1024K count=0 seek = 8192
```

This example creates an 8GB disk image. The next step is to install the guest operating system onto the disk image by a suitable installation media (such as a DVD or ISO image). This task can be performed with the help of the domain's configuration file. More specifically, the HVM guest configuration file should have an option to provide CD-ROM support to the guest as well as a boot device specification in order to boot from the CD-ROM device and install the operating system.

Xen includes a sample configuration file, '/etc/xen/xmexample.hvm' which can be used as a starting point for an HVM domU guest configuration file. Many of these values can be left unchanged. Some key values will be discussed in the remainder of this section. Let's take a look at the following configuration file:

```
#  -*- mode: python; -*-
#=======================================
# Python configuration setup for 'xm create'.
# This script sets the parameters used when a domain is created
using 'xm create'.
# You use a separate script for each domain you want to create, or
# you can set the parameters for the domain on the xm command
line.
#=========================================

# Kernel image file.
kernel = "hvmloader"

# The domain build function. HVM domain uses 'hvm'.
builder='hvm'

# Initial memory allocation (in megabytes) for the new domain.
# Allocating less than 32MBs is not recommended.
memory = 128

# A name for your domain. All domains must have different names.
name = "domu-hvm"

# The number of cpus guest platform has, default=1
vcpus=1

# Enable/disable HVM guest PAE, default=1 (enabled)
pae=1

# Enable/disable HVM guest ACPI, default=1 (enabled)
acpi=1

# Enable/disable HVM APIC mode, default=1 (enabled)
# Note that this option is ignored if vcpus > 1
apic=1

# Primary network interface
vif = [ 'type=ioemu, bridge=xenbr0' ]

# Define the disk devices you want the domain to have access to
disk = [ 'file:/home/giorgos/xen/domuhvm.img,hda,w',
',hdc:cdrom,r' ]

# Device Model to be used
device_model = 'qemu-dm'

# boot on floppy (a), hard disk (c), Network (n) or CD-ROM (d)
# default: hard disk, cd-rom, floppy
boot="dc"

# Enable SDL library for graphics, default = 0
sdl=0

# Enable OpenGL for texture rendering inside the SDL window,
default = 1
# valid only if sdl is enabled.
# opengl=1
# Enable VNC library for graphics, default = 1
vnc=1

# The address that should be listened on for the VNC server if vnc
is set.
# default is to use 'vnc-listen' setting from
# auxbin.xen_configdir() + /xend-config.sxp
#vnclisten="127.0.0.1"

# Set VNC display number, default = domid
#vncdisplay=1

# Try to find an unused port for the VNC server, default = 1
#vncunused=1

# Set password for domain's VNC console
# default is depents on vncpasswd in xend-config.sxp
vncpasswd=''

# No graphics, use serial port
#nographic=0

# Enable stdvga, default = 0 (use cirrus logic device model)
stdvga=0

#   Serial port re-direct to pty deivce, /dev/pts/n
#   then xm console or minicom can connect
serial='pty'

#   tsc_mode
tsc_mode=0

#   Qemu Monitor, default is disable
#   Use ctrl-alt-2 to connect
#monitor=1

# Enable sound card support, [sb16|es1370|all|..,..], default none
#soundhw='sb16'

# Set the real time clock to local time [default=0 i.e. set to utc]
localtime=1

# Start in full screen
#full-screen=1

# Enable USB support
#usb=1

# Enable USB mouse support
#usbdevice='mouse'
#usbdevice='tablet'

# Set keyboard layout, default is en-us keyboard.
#keymap='en-us'

# Enable/disable xen platform PCI device, default=1 (enabled)
```

```
#xen_platform_pci=1
```

Let's discuss now the key settings of the above configuration file:

- **kernel = "hvmloader":** This option defines the HVM firmware loader, which in our case is '/etc/lib/xen/boot/hvmloader'. This setting must be left unchanged.
- **builder = 'hvm':** This option specifies the domain build function and must be left unchanged.
- **memory = 128:** Initial memory allocation (in megabytes) for the new domain. Note that creating a domain with insufficient memory may cause out of memory errors. The domain needs enough memory to boot kernel.
- **name = "domu-hvm":** This option gives a name to the guest domain. Here we name it as 'domu-hvm'.
- **vcpus = 1:** Defines the number of the domain's virtual CPUs.
- **acpi = 1:** Enables HVM guest ACPI.
- **apic = 1:** Enables HVM guest APIC.
- **pae = 1:** Enables HVM guest PAE.
- **vif = ['type=ioemu, bridge=xenbr0']:** Defines the guest's network interfaces.
- **disk = ['file:/home/giorgos/xen/domuhvm.img,hda,w', ',hdc:cdrom,r']:** Defines the disk devices you want the domain to have access to, and what you want accessible as. If using a physical device as the HVM guest's disk, each disk entry is of the form 'phy:UNAME,ioemu:DEV,MODE,' where UNAME is the host device file, DEV is the device name the domain will see, and MODE is 'r' for read-only, 'w' for read-write. 'ioemu' means the disk will use 'ioemu' to virtualize the HVM disk. If not adding 'ioemu', it uses 'vbd' like paravirtualized guests. On the other hand, if using a disk image file, its form should be like 'file:FILEPATH,ioemu:DEV,MODE'. Optional devices can be emulated by appending 'cdrom' to the device type: ',hdc:cdrom,r'. It is also possible to define an ISO image as the guest's CD-ROM device. To do this we can use this setting: disk=['file:/home/giorgos/xen/domuhvm.img,hda,w','file:/home/giorgos/som edvd.iso,hdc:cdrom,r'].
- **boot = "dc":** This option specifies the boot device priority. boot on floppy (a), hard disk (c), Network (n) or CD-ROM (d). For example with the value "dc" the guest will boot from CD-ROM and failback to hard disk.
- **device_model = 'qemu-dm':** The device emulation tool for HVM guests. This parameter should not be changed.
- **sdl = 0:** This option enables SDL library for graphics. The default is disabled.
- **vnc = 1:** This option enables VNC library for graphics. The default is enabled. Note, that if you have 'vnc' enabled you must disable 'sdl' and vice versa.

- **vnclisten = "127.0.0.1":** Defines the address that should be listened on for the VNC server if 'vnc' is set.
- **nographic = 0:** Disables graphics.
- **usb = 1:** Enable USB support without defining a specific USB device.
- **usbdevice = 'mouse':** Enable USB support and also enable support for the given device. Devices that can be specified are 'mouse', 'tablet' and 'host:id1:id2'.
- **localtime = 1:** set the real time clock to local time [default=0 i.e. set to utc].

Since we have to install the operating system before we can boot from the hard disk drive we need to place the CD-ROM first in the boor order by specifying: boot = 'dc'. It is also required to provide a CD-ROM device or an ISO image to the guest as stated above. Now, we can save the above file as '/etc/xen/domu-hvm.config' and start the guest domain:

```
root@dom0$ xm create –c domu-hvm.config
```

The guest domain will boot and the installation process from the given media will begin. If SDL was chosen for the graphical console then the console should appear when the guest starts up. Otherwise, if VNC was selected and the HVM domainU was not automatically start vncviewer, it is necessary to connect manually. By default the VNC port is the ID of the domain to which we wish to connect (which can be obtained using the 'xm list' command). For example, to connect to domain with ID 1 we run:

```
root@dom0$ vncviewer localhost:1
```

5. Networking configuration

In this section we will take a look at networking in Xen. Each domain network interface is connected to a virtual network interface in dom0 by a point to point link. These devices are named vif<domain id><interface id> (e.g. 'vif1.0' for the first interface 'eth0' in domain 1).

Traffic on these virtual interfaces is handled in dom0 using standard Linux mechanisms for bridging, routing with NAT, or two-way routing. Typically, only one of these mechanisms can be used at once. These mechanisms are defined in shell scripts under the directory '/etc/xen/scripts/' and can be used by xend to perform initial configuration of the network and configuration of new virtual interfaces. The default mechanism that is used by xend is bridging. You can change this by commenting out the bridging scripts and enabling your preferred mechanism's scripts in the file '/etc/xen/xend-config.sxp'. Each mechanism has two related scripts as shown below:

- Bridge Networking:
  (network-script network-bridge)
  (vif-script vif-bridge)

- Routed Networking with NAT:
  (network-script network-nat)
  (vif-script vif-nat)

- Two-way Routed Networking:
  (network-script network-route)
  (vif-script vif-route)

5.1. Bridging

The default Xen configuration uses bridging within Domain0 to allow all domains to appear on the network as individual hosts. When xend starts, the physical interface 'eth0' is brought down. Then, a virtual network bridge 'veth0' is created and renamed 'eth0'. Finally, the physical interface is brought up and renamed to 'peth0' and is connected in the bridge 'eth0'. Afterwards, when a new domain is started its virtual interfaces (e.g. vif1.0) are connected in the 'eth0' bridge (Figure 6).

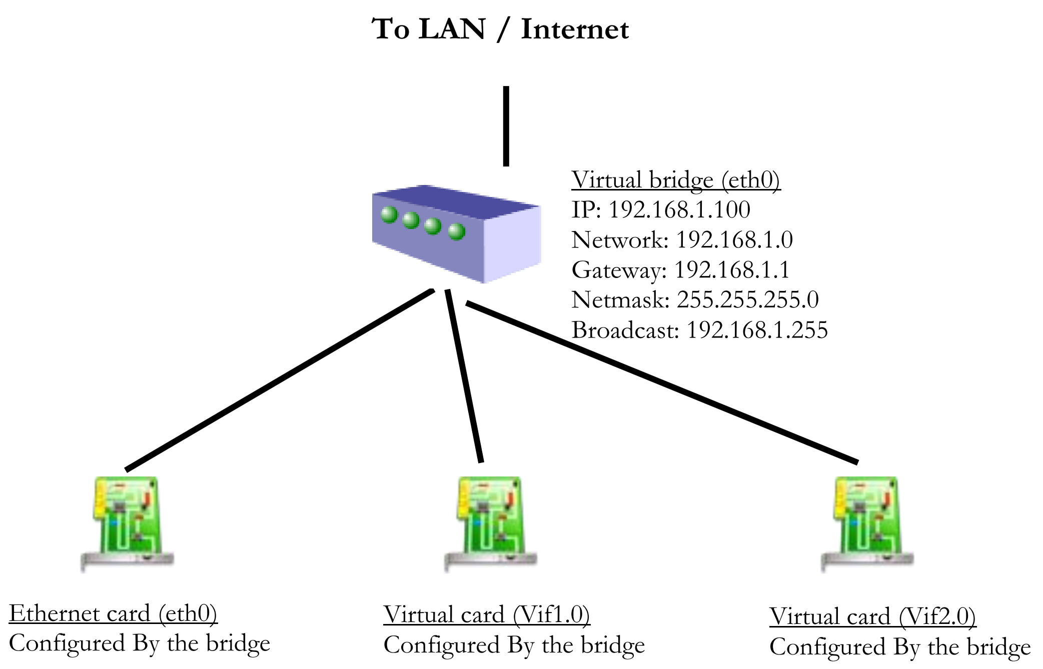


Figure 6 – Connecting all the network interfaces on a virtual network bridge

In this case the '/etc/network/interfaces' file on the dom0 will look like this:

```
eth0    Link encap:Ethernet  HWaddr 00:11:d8:55:4d:74
        inet addr:192.168.1.100  Bcast:192.168.1.255
        Mask:255.255.255.0
        inet6 addr: fe80::211:d8ff:fe55:4d74/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500
        Metric:1
        RX packets:58420 errors:0 dropped:0 overruns:0 frame:0
        TX packets:34142 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:27060374 (25.8 MiB)  TX bytes:4589683 (4.3 MiB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:89 errors:0 dropped:0 overruns:0 frame:0
        TX packets:89 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:8124 (7.9 KiB)  TX bytes:8124 (7.9 KiB)

peth0   Link encap:Ethernet  HWaddr 00:11:d8:55:4d:74
        inet6 addr: fe80::211:d8ff:fe55:4d74/64 Scope:Link
        UP BROADCAST RUNNING PROMISC MULTICAST
        MTU:1500  Metric:1
        RX packets:65661 errors:0 dropped:0 overruns:0 frame:0
        TX packets:34023 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:29744006 (28.3 MiB)  TX bytes:4583617 (4.3 MiB)
        Interrupt:17

Vif1.0 Link encap:Ethernet  HWaddr fe:ff:ff:ff:ff:ff
        inet6 addr: fe80::fcff:ffff:feff:ffff/64 Scope:Link
        UP BROADCAST RUNNING PROMISC MULTICAST
        MTU:1500  Metric:1
        RX packets:6 errors:0 dropped:0 overruns:0 frame:0
        TX packets:24 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:32
        RX bytes:362 (362.0 B)  TX bytes:4739 (4.6 KiB)

Vif2.0 Link encap:Ethernet  HWaddr fe:ff:ff:ff:ff:ff
        inet6 addr: fe80::fcff:ffff:feef:ffff/64 Scope:Link
        UP BROADCAST RUNNING PROMISC MULTICAST
        MTU:1500  Metric:1
        RX packets:8 errors:0 dropped:0 overruns:0 frame:0
        TX packets:14 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:32
        RX bytes:368 (382.0 B)  TX bytes:4934 (4.8 KiB)
```

5.1.1. A basic network configuration

The simplest configuration is to use bridging in order for the domU guests to join our existing network. Let's suppose that our home network is 192.168.1.0/24 with a gateway with IP address 192.168.1.1. The primary network interface in our dom0 system has a static IP address assigned which is 192.168.1.100. We can grab a new IP from this network (e.g. 192.168.1.101) and assign it to our new domU guest. By doing this, our domU guest will be fully visible and available on our existing network, allowing all traffic in both directions.

To achieve this setup we must perform the following steps:

1. Edit the xend configuration file '/etc/xen/xend-config.sxp' and enable these options: (network-script network-bridge) (vif-script vif-bridge) if they are not already enabled.
2. Edit the domU guest's '/etc/network/interface':

```
root@dom0$ mount –o loop /home/giorgos/xen/domu1.img /mnt/
root@dom0$ vi /mnt/etc/network/interfaces
root@dom0$ umount /mnt
```

If we prefer the domU guest's primary network interface to take an IP address from a DHCP server we must add the following lines to the above file:

```
# The loopback network interface
auto lo
```

```
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp
```

Otherwise, if we would like to give a static IP address (e.g. 192.168.1.101) to the domU guest's primary network interface we have to add the following lines:

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet static
        address 192.168.1.101
        netmask 255.255.255.0
        network 192.168.1.0
        broadcast 192.168.1.255
        gateway 192.168.1.1
```

3. In the domain's configuration file (e.g. '/etc/xen/domu1') we must make sure that we have enabled the option vif = ['']. Additionally, if we want to use DHCP we must make sure that we have enabled the option dhcp= "dhcp".

5.1.2. Custom Bridging

If we cannot use IP addresses of the physical network that our host system is connected, the ideal solution is to create a new network for our domU guests. Thus, let's say that our host system is connected on 192.168.1.0/24 network via the interface 'eth0'. The main idea behind this approach is to create a custom virtual bridge and assign it an IP address of a different network. Then all the domU guests will connect to this newly created network having the dom0's bridge as their gateway. However, in order for the domU guests to reach the physical network 192.168.1.0/24 where the dom0 is connected in, some routing must be done on the host between the virtual bridge and its primary network interface. Figure 7 summarizes this setup.

For this setup to take effect we have to disable the option (network-script network-bridge) in the '/etc/xen/xend-config.sxp' configuration file. On the other hand, the option (vif-script vif-bridge) must be left enabled.

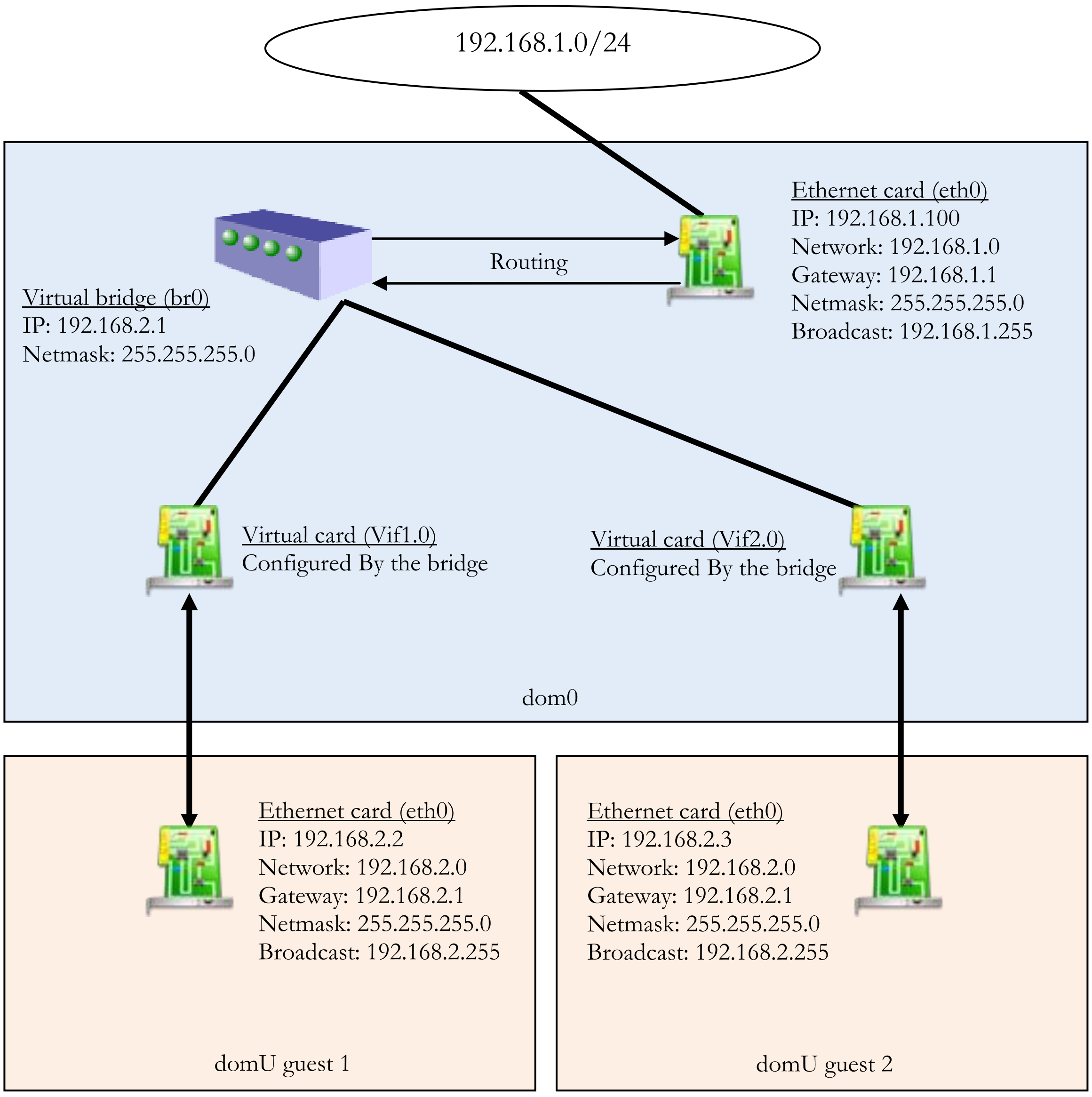


Figure 7 – Connecting the domU guests on a different network

Let's assume that we want to use the network 192.168.2.0/24 to connect our domU guests. We will assign to the bridge the IP address 192.168.2.1 as shown in figure 7. To create the new bridge we modify the '/etc/network/interfaces' file on dom0 as follows:

```
# This file describes the network interfaces available on your
system
# and how to activate them. For more information, see
interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback
```

```
# The primary network interface
auto eth0
iface eth0 inet static
        address 192.168.1.100
        netmask 255.255.255.0
        network 192.168.1.0
        broadcast 192.168.1.255
        gateway 192.168.1.1

# The bridge interface
auto br0
iface br0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        bridge_ports none
        bridge_maxage 12
        bridge_stp off
        bridge_fd 1
        bridge_hello 2
```

Finally, for the above configuration to work, we must make some additional routing configurations on dom0. At this point, the domU guest will be able to communicate properly with dom0 and vice versa. However, in order for the domU guest to communicate with other machines on 192.168.1.0/24 network or with the Internet we must enable routing on dom0. To do this we run the following command as root:

```
root@dom0$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

In order to make this change permanent we can set the option 'net.ipv4.ip_forward=1' in the '/etc/sysctl.conf' file. A final step is to enable IP masquerading with the help of IP tables. To do this, we create a new file: '/etc/network/if-up.d/dom0-routing' with the following content:

```
#!/bin/sh
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
exit 0
```

After that, we must set correct permission to this file by running:

```
root@dom0$ chmod 755 /etc/network/if-up.d/dom0-routing
```

After completing the above steps the network configuration needed to provide network connectivity to the domU guests is ready. Now, we can restart the dom0 system. When the system restarts we can run:

```
root@dom0$ ifconfig
```

The output must be like this:

```
br0     Link encap:Ethernet  HWaddr fe:ff:ff:ff:ff:ff
        inet addr:192.168.2.1  Bcast:192.168.2.255
        Mask:255.255.255.0
        inet6 addr: fe80::fcff:ffff:feff:ffff/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500
        Metric:1
        RX packets:17 errors:0 dropped:0 overruns:0 frame:0
        TX packets:81 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:972 (972.0 B)  TX bytes:8599 (8.3 KiB)

eth0    Link encap:Ethernet  HWaddr 00:11:d8:55:4d:74
        inet addr:192.168.1.100  Bcast:192.168.1.255
        Mask:255.255.255.0
        inet6 addr: fe80::211:d8ff:fe55:4d74/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500
        Metric:1
        RX packets:4703 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2651 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:1044816 (1020.3 KiB)  TX bytes:406619 (397.0
        KiB)
        Interrupt:17

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:89 errors:0 dropped:0 overruns:0 frame:0
        TX packets:89 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:8124 (7.9 KiB)  TX bytes:8124 (7.9 KiB)
```

Also, we can run on dom0:

```
root@dom0$ route –n
```

We will get:

```
Kernel IP routing table
Destination    Gateway       Genmask         Flags Metric Ref
Use Iface
0.0.0.0        192.168.1.1   0.0.0.0         UG    0      0        0 eth0
192.168.1.0    0.0.0.0       255.255.255.0   U     0      0        0 eth0
192.168.2.0    0.0.0.0       255.255.255.0   U     0      0        0 br0
```

Note the final line in the above example. This line specifies the route to the newly created network 192.168.2.0/24. All the packets will be routed to the bridge 'br0' in order to reach the domU guests. Then, the bridge will route each packet to the correct virtual interface which is connected to it.

It's time to do some configuration on our domU guest:

```
root@dom0$ mount –o loop /home/giorgos/xen/domu1.img /mnt
root@dom0$ chroot /mnt
root@dom0$ vi /etc/network/interfaces
root@dom0$ exit
root@dom0$ umount /mnt
```

In the guest's '/etc/network interfaces' we put the following lines:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 192.168.2.2
        netmask 255.255.255.0
        network 192.168.2.0
        broadcast 192.168.2.255
        gateway 192.168.2.1

up route add -net 192.168.2.0/24 gw 192.168.2.1
up route add default gw 192.168.2.1
```

As shown in the above example, we configure the primary network interface of the domU guest 'eth0' with the IP address 192.168.2.2. Also, we specify the netmask, the network, the broadcasting address and finally the gateway 192.168.2.1. The final two lines need some more explanation.

As we said earlier the dom0 guest is connected on the physical network 192.168.1.0/24 whereas the domU guest is connected on the network 192.168.2.0/24. In order for the domU guest to be able to access the machines on the network 192.168.1.0/24 and vice versa we must perform some routing. With the line 'up route add -net 192.168.1.0/24 gw 192.168.2.1' we add a new route to the network 192.168.0/24 specifying that all the packets that are directed for that network must be send on dom0's end which is 192.168.2.1. Then, dom0 is responsible to properly route this packets. The final line 'up route add default gw 192.168.2.1' adds a default route with gateway the dom0's bridge which has the address 192.168.2.1. This helps the domU guest to communicate with other hosts on the Internet.

Finally, for the above configuration to work, we must make some modifications on the domain's configuration file (e.g. '/etc/xen/domu1'). We must make sure that we have enabled the option vif = ['','bridge=br0']. With this option we specify the custom bridge 'br0' to xend. Thus, every time a new domain is started, its virtual interfaces (e.g. vif1.0) are connected in the 'br0' bridge automatically by the 'vif-bridge' script.

It's time to test our configuration. Let's run the domU guest:

```
root@dom0$ xm create –c domu1
```

After the guest boots we can run the following command to see if the primary network interface is correctly configured:

```
root@domu1$ ifconfig
```

The result must look like this:

```
eth0    Link encap:Ethernet  HWaddr ce:15:d1:2d:86:f0
        inet addr:192.168.2.2  Bcast:192.168.2.255
Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500
Metric:1
        RX packets:57 errors:0 dropped:32 overruns:0 frame:0
        TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:3121 (3.0 KiB)  TX bytes:159 (159.0 B)
        Interrupt:5

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Also, in order to see if the routing is properly configured we can run:

```
root@domu1$ route –n
```

The result must look like the lines below:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
Use Iface
0.0.0.0         192.168.2.1     0.0.0.0         UG    0      0       0
eth0
192.168.1.0     192.168.2.1     255.255.255.0   UG    0      0       0
eth0
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0       0
eth0
```

We can try to ping the dom0 and some Internet addresses to see if everything works properly. Also, we can try to ping the domU guest from dom0.

We have now configured a domU guest with network access. We can easily configure more domU guests to have network access by assign them IP addresses from the range of 192.168.2.0/24.

5.2. Routed networking with NAT

In routed networking with NAT (Network Address Translation) a private LAN is created for Xen guests. Traffic coming from the guests is then networked to the outside network via NAT. In this case, dom0 will automatically perform all the NAT'ing required.

Let's suppose that our home network is 192.168.1.0/24 with a gateway with IP address 192.168.1.1. The primary network interface in our dom0 system has a static IP address assigned which is 192.168.1.100. We will create a virtual private LAN for domU guests (e.g. 192.168.2.0/24). The domU guests must NAT via dom0, which will be the gateway of the new virtual network, to reach the home LAN. Thus, traffic on the home LAN appears as if coming from dom0 (192.168.1.100). The domU guests can be directly accessed from 192.168.1.0/24, however a route must be added to the gateway (192.168.1.1) for this to happen. To achieve this setup we must follow these steps:

1. Edit the xend configuration file '/etc/xen/xend-config.sxp' and enable these options (network-script network-nat) (vif-script vif-nat). Also, we must disable the other networking scripts.

2. Edit the domU guest's '/etc/network/interface':

```
root@dom0$ mount –o loop /home/giorgos/xen/domu1.img /mnt/
root@dom0$ vi /mnt/etc/network/interfaces
root@dom0$ umount /mnt
```

   We will give a static IP address (e.g. 192.168.2.1) to the domU guest's primary network interface and will assign its gateway to 192.168.2.254 which will be the host end:

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        network 192.168.2.0
        broadcast 192.168.2.255
        gateway 192.168.2.254
```

3. In the domain's configuration file (e.g. '/etc/xen/domu1') we must make sure that we have enabled the option vif = [''].

5.3. Two-way Routed Networking

In this setup forwarding rules must be put in manually. This setup however allows for the greatest flexibility when it comes to routing and setting up a private network. Routing creates a point-to-point link between dom0 and each domU. Routes to each domU are added to dom0's routing table, so domU must have a static IP. DHCP doesn't work in this setup.

Let's suppose again that our home network is 192.168.1.0/24 with a gateway with IP address 192.168.1.1. The primary network interface in our dom0 guest has a static IP address assigned which is 192.168.1.100. We will create a private LAN for domU guests (e.g. 192.168.2.0/24). The domU traffic is routed to the home network with the help of dom0. The domU guests can be directly accessed from 192.168.1.0/24, however a route must be added to the gateway (192.168.1.1) for this to happen. To achieve this setup we must follow these steps:

1. Edit the xend configuration file '/etc/xen/xend-config.sxp' and enable these options (network-script network-route) (vif-script vif-route). Also we must disable the other networking scripts.

2. Edit the domU guest's '/etc/network/interface':

```
root@dom0$ mount –o loop /home/giorgos/xen/domu1.img /mnt/
root@dom0$ vi /mnt/etc/network/interfaces
root@dom0$ umount /mnt
```

   We will give a static IP address (e.g. 192.168.2.1) to the domU guest's primary network interface and will assign its gateway to 192.168.2.254 which will be the host end:

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet static
        address 192.168.2.1
        netmask 255.255.255.0
        network 192.168.2.0
        broadcast 192.168.2.255
        gateway 192.168.2.254
```

3. In the domain's configuration file (e.g. '/etc/xen/domu1') we must make sure that we have enabled the option vif = [''].

4. We must also configure dom0 for routing. To achieve this we run the following command as root:

```
root@dom0$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

In order to make this change permanent we can set the option 'net.ipv4.ip_forward=1' in the '/etc/sysctl.conf' file. A final step is to enable IP masquerading with the help of IP tables. To do this, we create a new file: '/etc/network/if-up.d/dom0-routing' with the following content:

```
#!/bin/sh
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
exit 0
```

After that, we must set correct permission to this file by running:

```
root@dom0$ chmod 755 /etc/network/if-up.d/dom0-routing
```

In order for the machines on the network 192.168.1.0/24 to be able to see the network 192.168.2.0/24, we need to add a route. This is added into the Default Gateway of the external network (192.168.1.1) so that when any machine queries a 192.168.2.0/24 address, their default gateway is checked for an entry.

6. Advanced topics

In this section we will first summarize the management software and tools that Xen provides. Then we will see some more advanced topics like migration, root on NFS and CPU management.

6.1. Xen management tools

Here we will see the basic management tools that Xen provides. These tools are not part of the hypervisor. Instead they lie in Domain0 and they can communicate with the Xen hypervisor through Xen API which is built atop XML-RPC. The user can manage all the virtual machines with these tools from Domain0 thus the dom0 guest is mentioned as "Xen management domain".

6.1.1. Xend

The Xen daemon is responsible for providing the interface between the rest of the user space tools and the kernel interface. One of the most obvious responsibilities of xend is access control. On the local machine, this is managed by setting the permissions on the socket used to connect to the daemon as you would any other file, allowing access to be granted to specific users and groups. For remote administration, access can be granted to specific SSL client certificates. Furthermore, xend performs system management functions related to virtual machines and must be running in order to start and manage virtual machines (domU guests). To start xend at boot time, an initialization script named '/etc/init.d/xend' is provided. We can manually start xend by running:

```
root@dom0$ /etc/init.d/xend start
```

Also, we can stop xend by running:

```
root@dom0$ /etc/init.d/xend stop
```

Additionally, we can see the status of xend by running:

```
root@dom0$ /etc/init.d/xend status
```

As xend runs, events will be logged to '/var/log/xend.log' and sometimes to '/var/log/xend-debug.log'.

Xend is written in Python. At startup, it reads its configuration information from the file '/etc/xen/xend-config.sxp'. You can see the section 5 man page 'xend-config.sxp' for a full list of parameters and more detailed information about configuring xend.

6.1.2. Xm

The xm (Xen Master) tool is the simplest way of managing Xen. Xm has many subcommands including create, destroy, and list. The point is that xm is a client application that sends its commands to xend which then issues the commands to the Xen hypervisor. The general format of an xm command is:

```
$ xm command [switches] [arguments] [variables]
```

The available switches and arguments are dependent of the command chosen. The variables may be set using declarations of the form 'variable=value'. To get a list of supported commands we can run:

```
root@dom0$ xm help
```

Below, we will summarize some useful xm commands:

- xm list: Lists all the domains running and includes some useful information about each running domain like its name, its assigned memory, the number of virtual CPUs it has and its state.
- xm create <domain-conf>: Creates a new domain based on <domain-conf> configuration file.
- xm console <domain>: Attaches to the <domain>'s console.
- xm destroy <domain>: Terminates the <domain> immediately.
- xm save <domain>: Save the state of <domain> to restore later.
- xm resume <domain>: Resumes the saved state of <domain>.
- xm shutdown/reboot/pause <domain>: Shuts down/reboots/pauses the <domain>.
- xm migrate <domain>: Migrates a domain to another machine. You can append the '--live' switch if you prefer live migration.

6.2. Using NFS Root

The root file system is the file system that is directly mounted by the kernel during the boot phase and that holds the system initialization scripts and the most essential system programs. More specifically, the root file system includes the root directory together with a minimal set of subdirectories and files including '/boot', '/dev', '/etc', '/bin', '/sbin' and sometimes '/tmp'.

Mounting the root file system is a crucial part of system initialization. The Linux kernel allows the root file system to be stored in many different places [4], such as a hard disk partition, a floppy disk, a ramdisk or a remote file system shared via NFS. In any case, the root file system can be specified as a device file in the '/dev' directory either when compiling the kernel or by passing a suitable 'root=' option to the initial bootstrap loader. The root file system is mounted in a two-stage procedure:

1. The kernel mounts the special 'rootfs' file system, which provides an empty directory.
2. The kernel mounts the real root file system over this empty directory.

The root file system of a domU guest can be stored on a disk image in dom0, on a physical disk partition or in a directory hierarchy on dom0 or on a remote system which can be exported to the client via NFS. In this section we follow the last approach. More specifically, the root file system for the domU guest will be located on an exported directory hierarchy stored in a NFS server. This server could be dom0 guest, a remote machine or even another domU guest. Our goal is the domU client to mount this exported directory hierarchy as its root file system via NFS.

6.2.1. Preparation

Our working environment consists of the dom0 guest with IP address 192.168.1.100 and of a domU guest ('domu1') with IP address 192.168.2.2. The 'domu1' can see our home network through a virtual bridge with IP address 192.168.2.1 which resides on dom0. The dom0 guest will act as an NFS server and will provide the root hierarchy to the clients. Thus, our goal is the 'domu1' to mount the server's exported root hierarchy as its root file system via NFS. Note that the root file system can also be stored on a remote machine or even in a domU guest who acts as sever.

6.2.2. Server configuration

At this point we will analyze the necessary steps that need to be done on the server side. Initially, we will focus on the server's kernel configuration and then we will see how to create the root file system for the guest and how to export it.

On the server side we will need to compile NFS server support into the kernel. More specifically, our dom0 kernel must have the following options enabled as build-in:

```
        Network File Systems →
                NFS client support – enabled
                NFS client support for NFS version 3 – enabled
                NFS client support for the NFSv3 ACL protocol
                expansion – enabed
                NFS client support for NFS version 4 – enabled
                Use the new idmapper upcall routine – enabled
                NFS server support – enabled
                NFS server support for NFS version 3 – enabled
                NFS server support for the NFSv3 ACL protocol
                expansion – enabed
                NFS server support for NFS version 4 – enabled
```

Assuming our server's file system is ext4, we should also select the following options if we wish to use NFSv4's extended attributes and ACL features:

```
File Systems →
                The extended 4 (ext4) filesystem
                        Ext4 Extended attributes – enabled
                        Ext4 POSIX Access Control Lists – enabled
```

Now we will create a root file system for our domU guest. Let's assume that we have an NFSv4 server already configured and running. For a guide on how to set up an NFSv4 server look on [7]. Firstly, we create the directory '/export' if it doesn't already exist:

```
root@dom0$ mkdir /export
```

Then, we create a new directory under '/export' to place the guest's root file system:

```
root@host$ mkdir /export/domu1
```

Now, we can populate the guest's root file system either by copying from the dom0's root:

```
root@ dom0$ cp –ax /{root,dev,var,etc,usr,bin,sbin,lib} /export/domu1
root@ dom0$ mkdir /export/domu1/{proc,sys,home,tmp}
```

or by installing a base Linux distribution, for example Debian Squeeze:

```
root@dom0$ debootstrap --arch i386 squeeze /export/domu1
http://ftp.us.debian.org/debian
```

Next, we tailor the file system by editing '/etc/fstab', '/etc/hostname', '/etc/network/interfaces' etc and adding some user accounts. For more information on how to set up a root file system for a domU guest you can look on section 4.1.3. Note that '/etc/fstab' and '/dev/' need some caution. In 'etc/fstab' we have to mount the root file system via the NFS server. Assuming that our NFS-server is on 192.168.2.1, the guest's '/etc/fstab' must look like this:

```
192.168.2.1:/        /          nfs    rw          0      0
proc                 /proc      proc   defaults    0      1
```

Furthermore, we have to create a dummy device on the guest's '/dev' directory:

```
root@dom0$ chroot /export/domu1
root@dom0$ mknod /dev/nfs b 0 255
root@dom0$ chown root:disk /dev/nfs
root@dom0$ exit
```

Also, the guest's '/etc/network/interfaces' file must look like this:

```

# Used by ifup(8) and ifdown(8). See the interfaces(5) manpage or
# /usr/share/doc/ifupdown/examples for more information.

auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 192.168.2.2
        netmask 255.255.255.0
        network 192.168.2.0
        broadcast 192.168.1.255
        gateway 192.168.2.1

up route add -net 192.168.1.0/24 gw 192.168.1.1
up route add default gw 192.168.1.1
```

Of course, you can adjust it properly to suit your needs.

With the root file system populated on the NFS server and NFS services running, the next step is to export it so that it can be mounted on the domU guest. This is achieved by adding an entry to the '/etc/exports' file on the server:

```
/export/domu1
192.168.2.2(rw,fsid=0,async,wdelay,no_root_squash,no_subtree_che
ck)
```

In the above example 192.168.2.2 is the IP address of the domU guest which will use the exported root file system. The option ‘rw’ indicates that the domU guest can make changes to the file system, while the ‘fsid=0’ signals that the current export is the root. Also, ‘async’ enables asynchronous writes and ‘wdelay’ tells to the NFS server to delay writing to the disk if it suspects another write request is imminent. The ‘no_root_squash’ option disables root squashing on the server and finally the ‘no_subtree_check’ disables the sub tree check [7]. Once the ‘/etc/exports’ file has been updated, the ‘exportfs’ command can be run to update the table of exported NFS file systems:

```
root@dom0$ exportfs –a
```

Alternatively you can restart the NFS service.

On the other side, the domU guest’s kernel needs the following settings as minimum in order for the guest to be able to mount its root file system via NFS:

```
Networking support →
        Networking options →
                IP: kernel level autoconfiguration – enabled
                IP: DHCP support – enabled
                IP: BOOTP support – enabled
                IP: RARP support – enabled

File Systems →
        Network File Systems →
                NFS client support – enabled
                NFS client support for NFS version 3 – enabled
                NFS client support for the NFSv3 ACL protocol
                expansion – enabed
                NFS client support for NFS version 4 – enabled
                Root filesystem on NFS – enabled
                Use the new idmapper upcall routine – enabled
```

Be careful to compile the above options as build in and not as modules. Modules only work after the kernel is booted, and these things are needed during boot.

In order to successfully boot the new domU guest we have to pass it some options in order to tell it how to mount its root file system. This task could be done through its Xen configuration file ‘/etc/xen/domu1’:

```
#  -*- mode: python; -*-
#=======================================
# Python configuration setup for 'xm create'.
# This script sets the parameters used when a domain is created
using 'xm create'.
# You use a separate script for each domain you want to create, or
# you can set the parameters for the domain on the xm command
line.
#==========================================

# Kernel image file.
#kernel = "/boot/vmlinuz-3.0.4"
kernel = "/home/giorgos/xen/domains/domu1/vmlinux-3.0.4-
domu"

# Optional ramdisk – In order for the domU guest to mount its
root
# filesystem via NFS correctly, make sure to disable this option.
#ramdisk = "/home/giorgos/xen/domains/domu1/initrd.img-
3.0.4-domu"

# Initial memory allocation (in megabytes) for the new domain
memory = 128

# A name for your domain. All domains must have different names.
name = "domu1 "

# Define network interfaces.
vif = [ '','bridge=br0' ]

# Set ip.
ip='192.168.2.2'

# Set netmask.
netmask='255.255.255.0'

# Set default gateway.
gateway='192.168.2.1'

# Set the hostname.
hostname='domu1'

# Set root device for nfs.
root ='/dev/nfs'

# The nfs server.
nfs_server ='192.168.2.1'

# Root directory on the nfs server.
nfs_root   ='/export/domu1'

# Sets runlevel 4.
extra = "4"
on_poweroff = 'destroy'
on_reboot   = 'restart'
on_crash    = 'restart'
#==========================================
```

Let's explain the basic options we use above:

- **vif = ['','bridge=br0']:** With this option we specify the custom bridge 'br0' to xend. You can use its default bridge if you want to.
- **ip = '192.168.2.2':** With this option we set the IP address of the domU guest's primary network interface.
- **netmask = '255.255.255.0':** With this option we specify the netmask of the domU guest's primary network interface.
- **gateway = '192.168.2.1':** With this option we specify the gateway of the domU guest's primary network interface.
- **hostname = 'domu1':** With this option we specify the hostname of the domU guest.
- **root = '/dev/nfs':** Here comes the '/dev/nfs' dummy device file that we created earlier. This is necessary to enable the pseudo-NFS-device. Note that it's not a real device but just a synonym to tell the kernel to use NFS instead of a real device.
- **nfs_server = '192.168.2.1':** This option specifies the location of the NFS server that exports the domU guest's root file system.
- **nfs_root = '/export/domu1':** This option tells to the domU guest where to find its root file system on the NFS server.

Note that it's important to specify all the necessary information for the domU guest in order to set up its primary network interface correctly, because we need the guest's networking to be correctly configured before it tries to mount its root file system. We achieve this by specifying the parameters 'ip, 'netmask', 'gateway' and 'hostname'. To automatically assign an IP address to the guest using DHCP service we can simply use the option dhcp = "dhcp" and omit the 'ip', 'netmask' and 'gateway' options.

Now, we can start the new guest domain by typing:

```
root@dom0$ xm create –c domu1
```

6.3. Migration

Xen has a wealth of features ready for business use. The most popular is the support for fast migration which enables you to move a virtual machine to a different physical system. There are two variants of migration: regular and live. The first one moves a virtual machine to a different physical host by pausing it, copying its memory contents and then resume it on the destination. The latter performs the same logical functionality but without needing to pause the domain. Thus, we can move a virtual machine to a different physical system while it's still running without having any downtime. To perform a live migration both hosts should be running xend and the destination host must have sufficient resources to accommodate the domain after the move. Furthermore it is required both source and destination machines to be on the same LAN. Additionally, because there is no support for providing automatic remote access to local file systems when a domain is migrated, it is required the domain's root file system to be stored on a remote server. Thus, the domain must use NFS to mount its root file system.

To enable migration in Xen a few changes must be made to the '/etc/xen/xend-config.sxp' configuration file. By default, migration is disabled as it can be a potential security hazard if incorrectly configured. Opening the migration port can allow an unauthorized host to initiate a migration or connect to the migration ports. Thus, special care should be taken to ensure the migration port is not accessible to unauthorized hosts. We enable the following options in '/etc/xen/xend-config.sxp' to enable migration:

- **(xend-relocation-server yes):** This option enables migration server.
- **(xend-relocation-port 8002):** This option specifies the port xend should use for the relocation interface, if xend-relocation-server is set to yes. The port set by the xend-relocation-port parameter must be open on both systems.
- **(xend-relocation-address ''):** This option sets the address the xend listens for migration commands on the relocation-socket connection if xend-relocation-server is set. The default is to listen on all active interfaces. The (xend-relocation-address) parameter restricts the migration server to only listen to a specific interface. For example: (xend-relocation-address '192.168.2.1').
- **(xend-relocation-hosts-allow ''):** The 'xend-relocation-hosts-allow' parameter controls which hostnames can communicate on the relocation port. If the value is empty, then all connections are allowed. Otherwise, the 'xend-relocation-hosts-allow' parameter should be a sequence of regular expressions separated by spaces. Any host with a fully-qualified domain name or an IP address which matches one of these regular expressions will be accepted.

To perform a live migration we can use the xm utility:

```
root@dom0$ xm migrate --live localdomain destination.network.com
```

If we omit the '--live' switch, then xend performs a regular migration by pausing the domain.

6.4. CPU Management

An interesting feature of Xen is the ability to associate a domain's virtual CPU with one or more physical CPUs. Using this feature we can allocate real hardware resources among one or more guests. Furthermore we can make optimal use of processor resources by assigning some of its cores to a domain and others to a different domain.

Let's take a look how Xen enumerates physical CPUs. If we have two quad core, hyper threaded processors, the CPU order would be:

Processor 1								Processor 2							
core0		core1		core2		core3		core0		core1		core2		core3	
ht0	ht1	ht0	ht1	ht0	ht1	ht0	ht1	ht0	ht1	ht0	ht1	ht0	ht1	ht0	ht1
#0	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**	**#11**	**#12**	**#13**	**#14**	**#15**

To associate a domain's virtual CPUs with physical CPUs we can modify its configuration file and use the following options:

- **cpus = "":** This options leaves Xen to make the assignment of virtual CPUs to physical CPUs.
- **cpus = "0":** This options assigns the physical CPU #0 to all the domain's virtual CPUs.
- **cpus = "0-3,5,^1":** # all virtual CPUs are assigned on physical CPUs #0,#2,#3,#5.
- **cpus = ["2", "8"]:** The virtual CPU0 is assigned to the physical CPU #2 and the virtual CPU1 is assigned to the physical CPU #3.
- **vcpus = 2:** Specifies the number of virtual CPUs of the domain.

Note that it is a good practice to avoid having multiple virtual CPUs belonging to the same domain mapped to the same physical CPU, because this assignment can reduce performance.

7. References

1 David Chisnall: The Definitive Guide to the Xen Hypervisor. Prentice Hall, 2007

2 XenParaVirtOps [http://wiki.xensource.com/xenwiki/XenParavirtOps#head-6a64160b847b6fbd780350181436269c5a0a5b11]

3 Xen [http://xen.org/]

4 Xen Networking [http://wiki.xensource.com/xenwiki/XenNetworking]

5 Bridging Network Connections [http://wiki.debian.org/BridgeNetworkConnections]

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

7 NFSv4 Installation guide [http://www.cs.uoi.gr/~gkappes/tutorials/nfs_installation.pdf]

8 Greg Kroah Hartman, Linux Kernel in a Nutshell, O' Reilly, 2006

9 Xen Wiki - blktap Overview [http://wiki.xensource.com/xenwiki/blktap]