Herramientas de virtualización libres para sistemas GNU/Linux

Congreso Internet del Mediterráneo

Sergio Talens-Oliag
sto@iti.upv.es
Instituto Tecnológico de Informática (ITI)

25 de septiembre de 2010

Resumen

En este artículo hablaremos de qué son y cómo funcionan los sistemas de virtualización y las herramientas relacionadas. Una vez introducidos los conceptos básicos comentaremos con más detalle algunas de las soluciones de virtualización disponibles en las distribuciones de GNU/Linux actuales (fundamentalmente para **Debian GNU/Linux** y derivadas, a pesar de que también funcionan en otras distribuciones), con una pequeña explicación de sus características, casos de uso y modo de funcionamiento.

Contenido

¿Qué es la virtualización?

Antes de empezar a hablar de los diferentes tipos de virtualización haría falta concretar a que nos referimos cuando hablamos de *virtualización*, puesto que se trata de un término muy genérico que se puede emplear para referirse a cosas diferentes (para ver algunas acepciones se puede consultar el término en la Wikipedia: `http://en.wikipedia.org/wiki/Virtualization`).

En este documento hablaremos de la *virtualización de plataforma* o *virtualización de servidores*, es decir, la capacidad de ejecutar en un único equipo físico, el *anfitrión* (*host* en inglés), múltiples *sistemas operativos invitados* (*guests* en inglés).

La idea básica es tener la posibilidad de ejecutar *programas de usuario* dentro de un entorno virtual sin tener que modificarlos.

Hablamos de *software* o *programas de usuario* porque en algunos de los modelos de virtualización que vamos a comentar estos funcionan sin hacer ningún cambio pero si que puede ser necesario modificar el sistema operativo *invitado* para que todo funcione correctamente.

Tipos de virtualización

En los puntos siguientes enumeraremos los diferentes tipos de virtualización existentes explicando como funcionan y dando ejemplos de sistemas reales que los implementan.

Emulación o simulación del hardware a nivel de aplicación

Una aplicación simula el *hardware* completo, permitiendo la ejecución de sistemas operativos sin modificar.

La ejecución se hace bajo el control del emulador que simula el sistema completo, incluyendo la ejecución de las instrucciones a nivel de CPU. El emulador simula la ejecución de código binario para una CPU concreta en un sistema real que usa un procesador y un juego de instrucciones diferente al del sistema emulado.

El inconveniente de este modelo de virtualización es que la simulación es muy lenta (para cada instrucción del sistema emulado puede ser necesario ejecutar entre 100 y 1000 instrucciones a la CPU real), a pesar de que en algunos casos no es un problema grande (por ejemplo la emulación de sistemas de los años 80 en hardware actual funciona mucho más rápida que en los equipos originales).

Ejemplos:

- **Bochs**: http://bochs.sourceforge.net/
- **MAME**: http://mamedev.org/
- **QEMU**: http://bellard.org/qemu/

Virtualización completa o nativa sin apoyo hardware

Este tipo de sistemas usan una máquina virtual que hace de intermediaria entre el sistema *invitado* y el hardware real.

El *software* de virtualización es conocido generalmente como *monitor de máquina virtual* (VMM, *Virtual Machine Monitor*) o *hipervisor* (*hypervisor*).

En este tipo de sistemas el *hipervisor* se encarga de emular un sistema completo y analiza dinámicamente el código que quiere ejecutar el sistema *invitado*, reemplazando las instrucciones críticas (las que hace falta virtualizar) por nuevas secuencias de instrucciones que tienen el efecto deseado en el hardware virtual, mientras que las instrucciones no críticas se ejecutan tal cual en la CPU real.

Este tipo de sistemas permiten la ejecución de sistemas operativos sin modificar.

Ejemplos:

- **VirtualBox**: http://www.virtualbox.org/
- **VMWare**: http://www.vmware.com/

Virtualización completa o nativa con apoyo hardware

Este tipo de sistemas funcionan de manera similar a los *sistemas de virtualización completa sin apoyo hardware*, pero aprovechan tecnologías incorporadas a las nuevas generaciones de microprocesadores como las de *Intel* (Intel-VT, VT-x para 32 bit y VT-y para 64 bit) y *AMD* (AMD-V), de forma que es posible ejecutar el código del sistema operativo *invitado* sin modificarlo.

En estos sistemas lo que se hace es ejecutar el *hipervisor* o VMM con el máximo nivel de acceso a la CPU (*Anillo -1* en procesadores AMD e Intel) y los sistemas *invitados* se ejecutan a un nivel inferior (*Anillo 0* en procesadores AMD e Intel, que era el máximo nivel de ejecución cuando los procesadores no incorporaban apoyo para la virtualización).

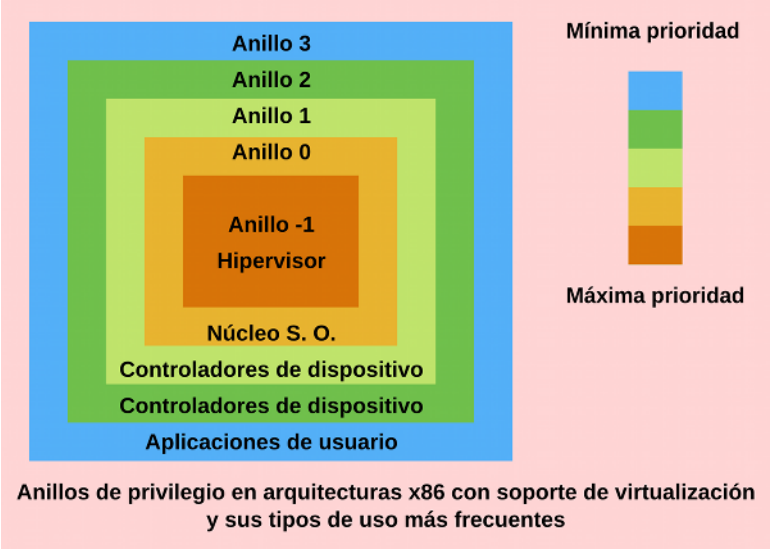

Figura 1: Anillos de privilegio en x86

Con la introducción de un nivel superior al que ya usaban los sistemas reales conseguimos que no sea necesario hacer ningún cambio a los sistemas *invitados*, pero ahora esos sistemas no tienen acceso a los dispositivos reales y es la CPU quién avisa al VMM cuando se quieren ejecutar instrucciones para acceder a los dispositivos desde los sistemas *invitados* y es el *hipervisor* quién se encarga de dar el acceso a los dispositivos virtuales o reales que correspondan.

Ejemplos:

- **KVM**: `http://kvm.qumranet.com/`

Paravirtualización

Son sistemas similares a los de virtualización *completa*, que ejecutan el sistema *invitado* con un *hipervisor* que se ejecuta sobre el sistema real.

La diferencia con el modelo anterior es que en este tipo de virtualización se modifica el sistema operativo *invitado* para incluir instrucciones relacionadas con la virtualización, de forma que en lugar que el *hipervisor* capture las instrucciones problemáticas, es el sistema *invitado* quién llama directamente al *hipervisor* cuando es necesario.

Evidentemente, con independencia de las modificaciones del *núcleo del sistema invitado*, los programas de usuario se pueden ejecutar sin ningún cambio.

Un caso extremo de este modelo de virtualización es el del UML (*User-mode Linux*), en el cual el núcleo del sistema *invitado* se transforma en una aplicación a nivel de usuario que hace la función de *hipervisor* y se encarga de emular el hardware, a pesar de que lo hace a nivel de la interfaz del sistema operativo y no a nivel de interfaz física (como el acceso a los dispositivos dentro del sistema *invitado* siempre se hace a través del *núcleo* no hace falta emular el hardware, sólo la visión que tienen de él los programas de usuario).

El gran problema de este modelo es que hace falta modificar el sistema operativo *invitado*, cosa que no es posible en algunos casos (por ejemplo, ¿cómo modificamos el código de **Windows XP** para que funcione con *para-virtualización*?).

Ejemplos:

- **User-mode Linux**: http://user-mode-linux.sourceforge.net/
- **Xen**: http://www.xen.org/

Virtualización a nivel de sistema operativo

En este tipo de sistemas sólo ejecutamos un núcleo (el del *anfitrión*) y este núcleo crea *entornos de ejecución* que las aplicaciones ven como *máquinas virtuales*.

En principio en este tipo de sistemas no hace falta emular el hardware a bajo nivel, puesto que en realidad es el mismo sistema operativo quién controla los dispositivos físicos. Lo que sí se suele hacer es incluir apoyo para tener dispositivos virtuales como discos o tarjetas de red dentro de cada entorno de ejecución.

La idea es que los programas se ejecutan en un entorno que hace creer a las aplicaciones que se encuentran en un sistema independiente cuando en realidad comparten recursos con otras máquinas virtuales, a pesar de que el sistema organiza las cosas para evitar que los entornos se interfieran entre ellos.

Este es uno de los modelos de virtualización más económico, puesto que no necesita apoyo del *hardware* ni hace falta supervisar el código abajo nivel, pero tiene el inconveniente que sólo permite ejecutar entornos virtuales para la misma CPU y sistema operativo y en realidad sólo hay un núcleo, de forma que si ese núcleo tiene un problema todas las máquinas virtuales se ven afectadas.

Ejemplos:

- **Linux-VServer**: http://www.linux-vserver.org/
- **LXC**: http://lxc.sourceforge.net/
- **OpenVZ**: http://www.openvz.org/

Sistemas de virtualización libres para GNU/Linux

En este apartado comentaremos las características más interesantes de algunos de los sistemas de virtualización disponibles para GNU/Linux.

Bochs (<http://bochs.sourceforge.net/>)

Es un emulador de arquitecturas basadas en `x86` que funciona en múltiples plataformas; el mayor interés de **Bochs** es que es capaz de emular un PC completo incluyendo los periféricos y funciona en prácticamente cualquier sistema anfitrión (por ejemplo se puede usar para emular un PC en un Linux que se ejecuta en una arquitectura *PowerPC*, *Alpha*, *SPARC* o *MIPS*).

El problema de este sistema es que es muy lento. De todos modos las últimas versiones van mejorando la velocidad de emulación empleando técnicas de optimización como las descritas el artículo *Virtualization without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure* de Darek Mihocka y Stanislav Shwartsman, disponible en:

http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf

QEMU (<http://wiki.qemu.org/>)

QEMU es un emulador similar a **Bochs** que tiene dos modos de funcionamiento, uno de *emulación de sistema completo* y otro de *emulación en modo usuario*.

En el modo de *sistema completo* el programa emula un equipo entero (por ejemplo un PC basado en microprocesadores `x86` o `x86_64`) incluyendo múltiples procesadores y periféricos. Este modo se usa para ejecutar sistemas operativos completos. En las últimas versiones del programa se soportan más de 15 arquitecturas diferentes.

En la emulación en modo usuario el programa puede ejecutar programas compilados para una CPU concreta en un sistema que funciona sobre una CPU diferente; esto se puede emplear, por ejemplo, para ejecutar el **Wine** en una arquitectura no *Intel*.

Para las arquitecturas `x86` QEMU soporta el uso de un módulo de aceleración para sistemas anfitriones *Linux* y *Windows* que permite que parte del código que se ejecuta en los sistemas *invitados* sea ejecutado directamente por la CPU real, haciendo que el *QEMU* funcione como un sistema de virtualización *nativa* en lugar de como un emulador.

VirtualBox (<http://www.virtualbox.org/>)

Oracle VM VirtualBox es un software de virtualización para arquitecturas x86 que funciona sobre múltiples sistemas operativos (la versión actual tiene binarios para **GNU/Linux**, **Mac OS X**, **Solaris/OpenSolaris** y **Windows**).

El sistema permite crear múltiples máquinas virtuales de 32 y 64 bits sobre las que se puede instalar casi cualquier sistema operativo que funcione en la arquitectura virtualizada; cada máquina virtual puede arrancarse, suspenderse y pararse de modo independiente y como el funcionamiento del software y los ficheros que definen la máquina virtual son los mismos en todos los sistemas anfitriones es fácil mover las máquinas virtuales entre distintos equipos; por otro lado las últimas versiones del producto permiten importar y exportar máquinas virtuales en el formato de virtualización abierto (**OVF**, por *Open Virtualization Format*).

Otra característica importante del virtualizador es que puede funcionar sin soporte de virtualización hardware aunque si trabajamos en máquinas con instrucciones VT-x o AMD-V el sistema las aprovecha para mejorar el rendimiento.

Lo que hace el emulador *software* es parchear el código del anfitrión para que el código que normalmente se ejecuta en el anillo 0 lo haga en el 1, dejándo el hipervisor en el nivel 0; si se detecta un problema VirtualBox emplea un recompilador dinámico basado en el de QEMU para resolverlo, en muchos casos desensamblando y parcheando el código del invitado para evitar futuras *recompilaciones*.

El sistema también incluye *drivers* para gestionar el acceso al hardware virtualizado en distintos sistemas operativos; la ventaja de usar estos *controladores* es que permite la interacción entre el *anfitrión* y sus *invitados* y que se simplifica la emulación de los dispositivos, lo que mejora la velocidad de ejecución del sistema emulado.

Aunque no lo habíamos mencionado, existen dos versiones del producto, una libre y otra con licencia privativa, aunque las diferencias entre ellas son pocas.

En cuanto al hardware **OpenBox** emula procesadores multi núcleo (SMP), controladoras USB (sólo en la versión con licencia privativa), discos IDE, SATA y SCSI, tarjetas de sonido, tarjetas de red, puertos serie y paralelo y da acceso a ACPI, permite usar múltiples resoluciones de pantalla, etc.

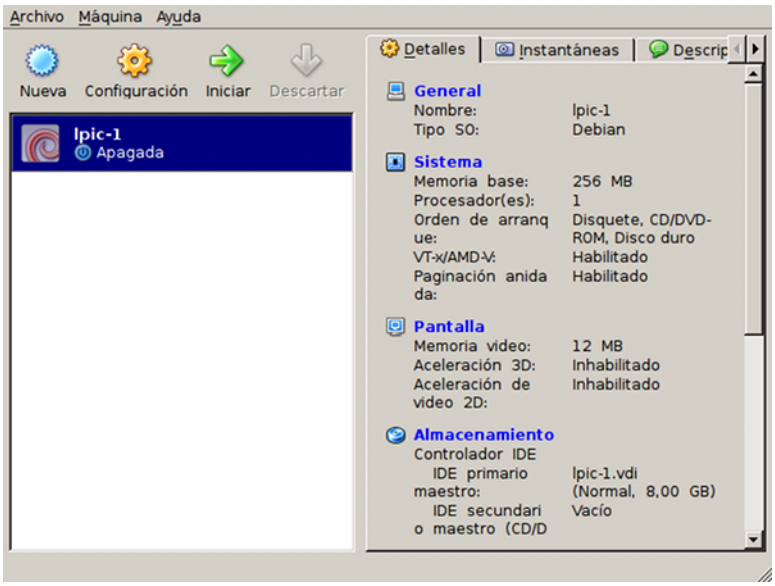

Figura 2: Configuración de una Máquina virtual con VirtualBox

Por último mencionar que el producto incluye varias funcionalidades interesantes desde el punto de vista de la usabilidad de la herramienta, como:

- Múltiples frontales para controlar la aplicación: `VirtualBox` (interfaz basada en Qt, es la empleada por los usuarios de escritorio), `VBoxManage` (interfaz de línea de órdenes), `VBoxSDL` (interfaz gráfico simplificado, empleado para dar acceso a la máquina virtual sin permitir la configuración de la misma) y `VBoxHeadless` (lanza la máquina virtual sin entorno gráfico local, aunque podemos habilitar el servidor RDP en la versión comercial y el VNC en la edición libre).
- Permite el acceso a maquinas virtuales de forma remota, usando VNC en la edición libre y por medio del Remote Desktop Protocol (RDP) en la versión completa (el servidor RDP permite hasta compartir dispositivos USB remotos).
- Soporte de múltiples ramas de *snapshots* (podemos almacenar el estado de la máquina virtual en un momento cualquiera y recuperarlo cuando nos convenga).

KVM (<http://kvm.qumranet.com/>)

KVM (*Kernel Virtual Machine*) es una solución de virtualización completa en la que se utiliza el núcleo de Linux como *hipervisor*, de manera que tanto el control de los dispositivos reales como la planificación de tareas y la gestión de memoria del sistema *anfitrión* las hace el núcleo de Linux.

En este modelo las máquinas virtuales son procesos normales del sistema (por esto la gestión de memoria y la planificación de procesos son las estándar del sistema) a los que añadimos un modo de ejecución adicional (*invitado*) a parte de los modos de ejecución estándar de Linux (*usuario* y *núcleo*).

Así, una máquina virtual tendrá tres modos de ejecución:

- Modo *invitado*: será el modo de ejecución normal para el código del sistema *invitado* siempre que no tenga operaciones de entrada/salida.

- Modo *usuario*: sólo lo usaremos para ejecutar las operaciones de entrada/salida del sistema *invitado*, nos permitirá gestionar dispositivos virtuales a nivel de usuario.
- Modo *núcleo*: se usará para entrar a trabajar en modo *invitado* y para gestionar las salidas desde modo *usuario* causadas por operaciones especiales o de entrada/salida.

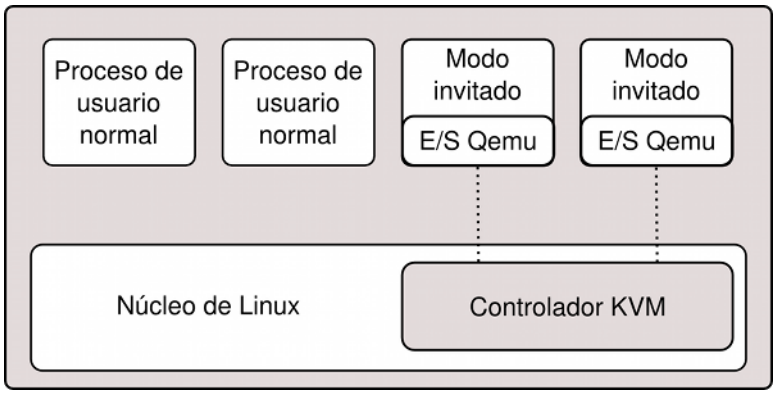

Figura 3: Arquitectura de KVM

En cuanto a la implementación del sistema, el KVM está formado por dos componentes:

- Un *controlador de dispositivos* para gestionar el hardware de virtualización, accesible desde el dispositivo `/dev/kvm` (incluido en el núcleo de Linux desde la versión 2.6.20, con soporte para microprocesadores Intel y AMD).
- Un *programa de usuario* que emula el hardware del PC (actualmente se usa una versión modificada de `qemu`) que se encarga de reservar la memoria de la máquina virtual y llamamiento al controlador anterior para ejecutar código en modo *invitado*.

Una de las ventajas de haber usado el **QEMU** como componente de espacio de usuario es que la gestión de la entrada/salida es la misma que a la emulador y por lo tanto podemos usar los mismos dispositivos virtuales que funcionan con **QEMU**.

El hecho de que el apoyo para el KVM esté integrado en las versiones oficiales del núcleo y que sea el sistema de virtualización preferido de distribuciones como **RedHat** o **Ubuntu** hacen que KVM sea una tecnología a considerar a corto y medio plazo para hacer *virtualización nativa* con Linux.

User-mode Linux (<http://user-mode-linux.sourceforge.net/>)

Podríamos decir que **User-mode Linux** (UML) es una aplicación que sólo se puede ejecutar sobre sistemas GNU/Linux y que nos proporciona un sistema operativo Linux virtual.

Técnicamente el UML es una adaptación del núcleo de Linux como las que se hacen para poder ejecutarlo en diferentes procesadores, con la diferencia que en este caso es una adaptación a la interfaz software definida por el núcleo y no a la interfaz hardware definida por la arquitectura física.

En realidad en UML lo que se hace es transformar un núcleo pensado para ejecutarse sobre un sistema físico en una aplicación de nivel de usuario en la que todos los dispositivos son virtuales, con sus ventajas e inconvenientes.

Un sistema virtualizado con UML es más lento que un sistema de virtualización a nivel del sistema operativo, puesto que estamos ejecutando el núcleo como proceso, pero por otro lado tenemos la ventaja de que estamos seguros de que la máquina virtual está claramente aislada del sistema real y de otras máquinas virtuales como ella, lo cual nos da muchas garantías respecto a las consecuencias de los problemas generados por el código que se ejecuta dentro de cada una de las máquinas virtuales.

Xen (<http://www.xen.org/>)

Xen es una solución de *paravitualización* que implementa un *hipervisor* que se ejecuta en el nivel más privilegiado de la máquina y que básicamente se hace cargo de la planificación de tareas y de la gestión de memoria, delegando la gestión de la Entrada/Salida en un *invitado privilegiado* (llamado *domain 0* o *dom0* en **Xen**) que arranca siempre que lanzamos el *hipervisor* y que en las distribuciones de GNU/Linux que incluyen **Xen** es normalmente una versión modificada del núcleo de Linux.

A pesar de que en este artículo hablamos de sistemas basados en núcleos GNU/Linux, es importante indicar que también se posible usar versiones modificadas de **NetBSD** y **Solaris** como núcleo para el *dom 0*, es decir, que en la práctica **Xen** no es un sistema de virtualización ligado al núcleo de Linux.

La idea básica detrás de este modelo de funcionamiento es que así el código del *hipervisor* es más sencillo y ligero, a pesar de que actualmente y dada la complejidad de las CPU (*multithreading*, *multicore*, etc.) y de la gestión de memoria, cada vez el tema de la simplicidad es menos evidente.

Cuando **Xen** se emplea en una CPU que no soporta virtualización a nivel hardware es necesario modificar el código del sistema operativo que se vaya a ejecutar sobre él, por lo que no es posible ejecutar sistemas como **Windows XP** en una CPU que no proporcione soporte hardware a la virtualización.

Si la CPU soporta virtualización el *hipervisor* de **Xen** se ejecuta en el anillo de máxima prioridad (*anillo -1* en Intel/AMD) y en ese caso sí que podemos ejecutar sistemas operativos sin ninguna modificación.

Es importante indicar que **Xen** o más concretamente su *hipervisor* nunca será integrado al núcleo de Linux, puesto que en realidad se trata de un programa independiente que no está integrado de ninguna manera con el código del núcleo de Linux.

Lo que si podría incorporarse al núcleo de Linux serían los parches que hacen Linux se pueda emplear como *invitado privilegiado* (*dom0*) o como *invitado* a secas (*domU* en terminología **Xen**).

Linux-VServer (http://www.linux-vserver.org/)

El **Linux-VServer** es un sistema de virtualización a nivel de sistema operativo que se implementa como una serie de parches sobre el núcleo de Linux.

Lo que hace este sistema es incluir apoyo en el núcleo para crear y mantener múltiples entornos de usuario independientes (conocidos como VPS o *Virtual Private Servers*) sin que tengan ninguna interferencia entre ellos.

Para independizar los espacios de usuario se define el concepto de *contexto*, que no es más que un *contenedor* (*container*) de procesos relacionados con un único VPS. Cuando el sistema arranca define un contexto por defecto que es el que emplean todos los procesos que pertenecen al sistema *anfitrión*.

A parte de los *contextos*, el **Linux-VServer** también emplea una llamada similar a `chroot` para redefinir el directorio raíz de los procesos que se ejecutan dentro de un contexto determinado y evita que puedan acceder a los directorios que hay por debajo de la raíz.

Como esta tecnología no tiene ninguna dependencia relacionada con la CPU del anfitrión, el sistema está disponible para múltiples familias de microprocesadores (`x86`, `x86-64`, `PowerPC`, `ARM`, etc.).

El problema de este sistema es que no gestiona adecuadamente la utilización compartida de recursos virtuales como las tarjetas de red (o en este caso sería más correcto hablar de dispositivos virtuales de red), puesto que lo que hace es usar los recursos del *anfitrión* sin aislarlos de los que usa la máquina virtual (por ejemplo si lanzamos una operación de *bind()* contra un puerto dentro de una máquina virtual y el puerto está ocupado por un proceso que se ejecuta en el *anfitrión* y que no especificó una dirección IP el *bind()* falla, cosa que no pasaría si el aislamiento entre dispositivos virtuales de red fuera total).

OpenVZ (http://www.openvz.org/)

El **OpenVZ** es un sistema de virtualización similar al **LinuxVServer** que incluye capacidades y herramientas de administración más adelantadas que las de este último.

En concreto el **OpenVZ** añade virtualización (permite la existencia de múltiples entornos virtuales aislados dentro del mismo núcleo), gestión de recursos (proporciona mecanismos para limitar y en ocasiones garantizar la disponibilidad de recursos como la CPU, la memoria o el espacio de disco para cada entorno virtual) y capacidad de *checkpointing* (posibilidad de *congelar* un entorno virtual, almacenar su estado completo en un fichero que podemos usar más tarde para *descongelar* el entorno virtual en la misma o en otra máquina real dejándolo en el mismo estado que tenía antes de la congelación).

Además **OpenVZ** se distribuye con un conjunto de utilidades que simplifican mucho la creación y mantenimiento de los entornos virtuales (VE, *Virtual Environments* en la documentación de **OpenVZ**), incluyendo la posibilidad de trabajar en plantillas de entornos virtuales pre instaladas (básicamente las plantillas son archivos `tar.gz` que contienen una imagen del sistema de archivos raíz de un VE).

Cómo en el caso de **Linux-VServer** este sistema de virtualización funciona sobre múltiples arquitecturas: x86, x86-64, PowerPC, etc.

LXC (`http://lxc.sourceforge.net/`)

Los **Linux Containers** (`lxc`) son un método de virtualización a nivel de sistema operativo que permite ejecutar múltiples instalaciones de servidores aisladas (contenedores) en un único anfitrión. Los **Linux Containers** no proporcionan una máquina virtual, sino que proporcionan un entorno que tiene su propio espacio para procesos y conexiones de red; en la práctica su uso es similar a un **chroot** pero ofrece mucha más aislación.

En LXC la gestión de recursos se hace a través de los `cgroups` (grupos de control) también conocidos como contenedores de procesos (*process containers*) y la independencia de recursos se consigue usando los `namespaces` (espacios de nombres) de **Linux**.

LXC forma parte del núcleo oficial de linux desde la versión 2.6.29, lo que hace que sepamos que siempre lo vamos a tener disponible con los nuevos núcleos; su desventaja respecto a **Linux-Vserver** y **OpenVZ** es que tiene menos funcionalidades que los sistemas anteriores, aunque es una buena solución si no necesitamos esas funcionalidades adicionales.

Herramientas relacionadas con los sistemas de virtualización

Todos los sistemas de virtualización que hemos comentado antes incluyen herramientas de administración para crear y configurar máquinas virtuales y para ejecutar y parar esas máquinas.

De cualquier modo esas herramientas no suelen ser demasiado cómodas de utilizar (de los sistemas comentados, salvo **VirtualBox** ninguno viene con un interfaz gráfico de administración *por defecto*, en general sólo se incluyen herramientas de línea de órdenes), razón por la cual es habitual que se desarrollen aplicaciones que hacen de *frontend* de esos programas que simplifican y hacen más cómodo el trabajo con los sistemas de virtualización desde la línea de órdenes, en el escritorio o a través de la web.

De hecho las operaciones e interfaces son tan similares que incluso existe una biblioteca denominada **libvirt** (`http://libvirt.org`) que proporciona una caja de herramientas para gestionar de manera unificada múltiples sistemas de virtualización (**libvirt** soporta **Xen**, **QEMU**, **KVM**, **LXC**, **OpenVZ**, **User Mode Linux** y **VirtualBox** entre otras tecnologías) sobre la que se han desarrollado múltiples aplicaciones.

Como ya hemos mencionado, los *frontends* suelen ser aplicaciones de escritorio o aplicaciones web.

En el ámbito del interfaz gráfico encontramos unas cuantas aplicaciones:

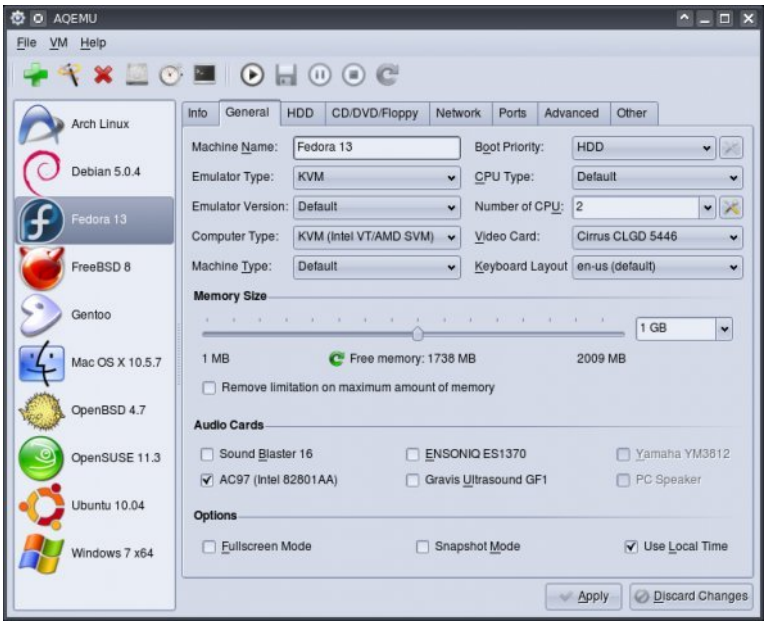

Figura 4: **aqemu** <http://aqemu.sourceforge.net/> para **QEMU** y **KVM**

Figura 5: **kemu** <http://www.rpdev.net/home/kemu> para **QEMU** y **KVM**,

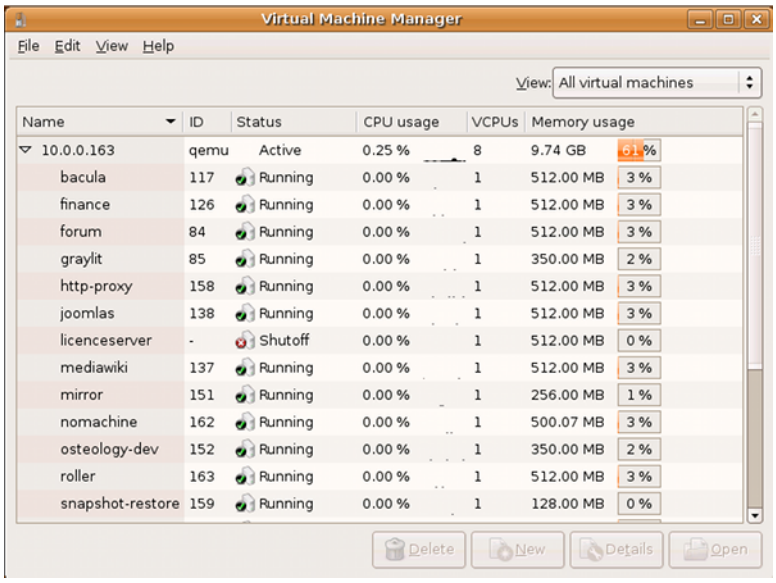

Figura 6: **Virtual Machine Manager** <http://virt-manager.org/> para los sistemas de virtualización soportados por **lib-virt**

Y vía web encontramos aplicaciones como:

Figura 7: **qemudo** (`http://qemudo.sourceforge.net/`) para **QEMU** y **KVM**

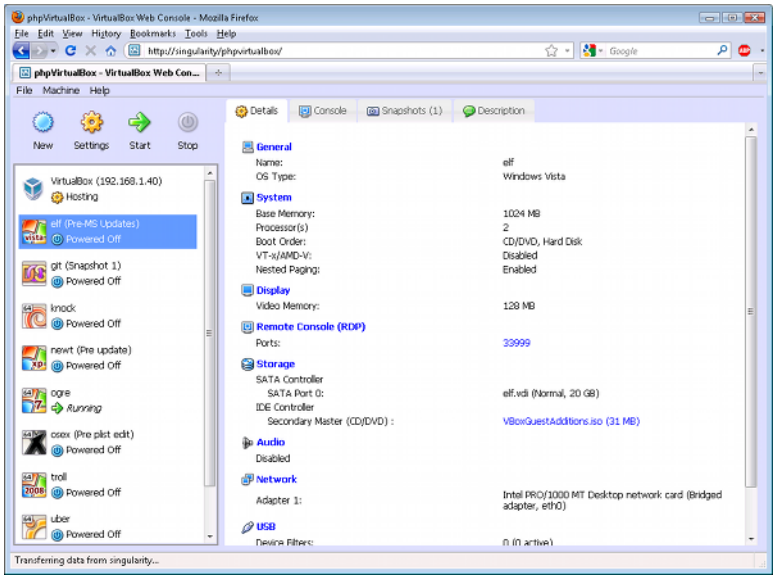

Figura 8: **phpVirtualBox** (`http://code.google.com/p/phpvirtualbox/`) para **VirtualBox**

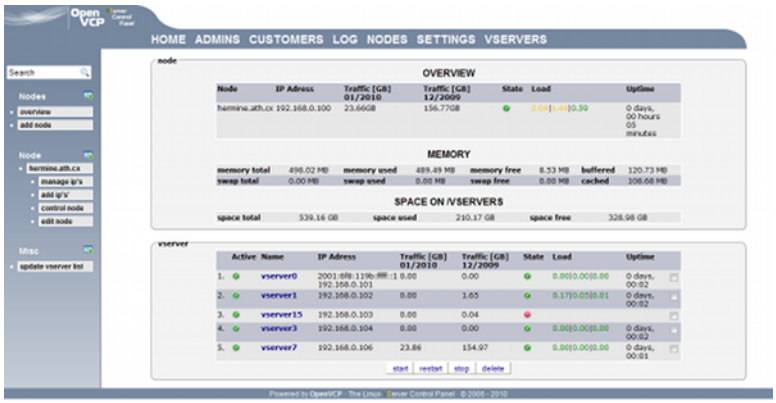

Figura 9: **OpenVCP** (`http://www.openvcp.org/`) para **Linux-Vserver**

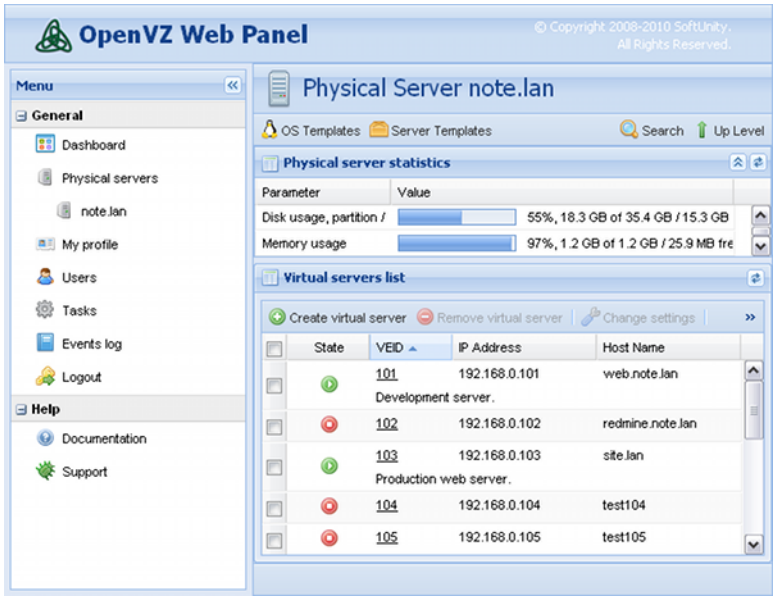

Figura 10: **OpenVZ Web Panel** (`http://code.google.com/p/ovz-web-panel/`) para **OpenVZ**

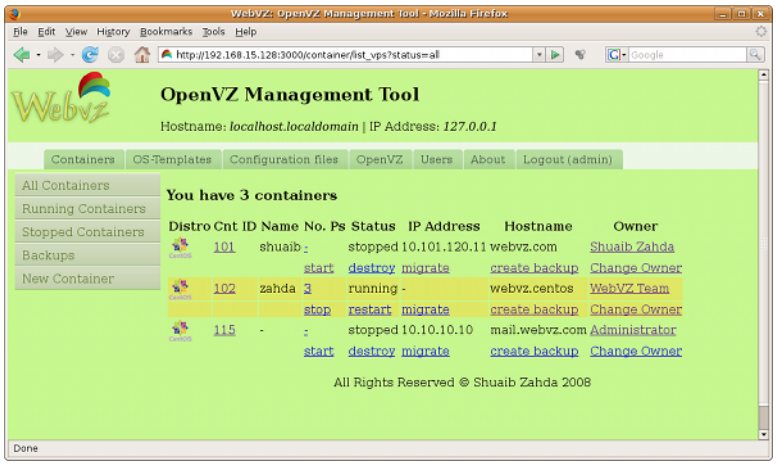

Figura 11: **WebVZ** (`http://webvz.sourceforge.net/`) para **OpenVZ**

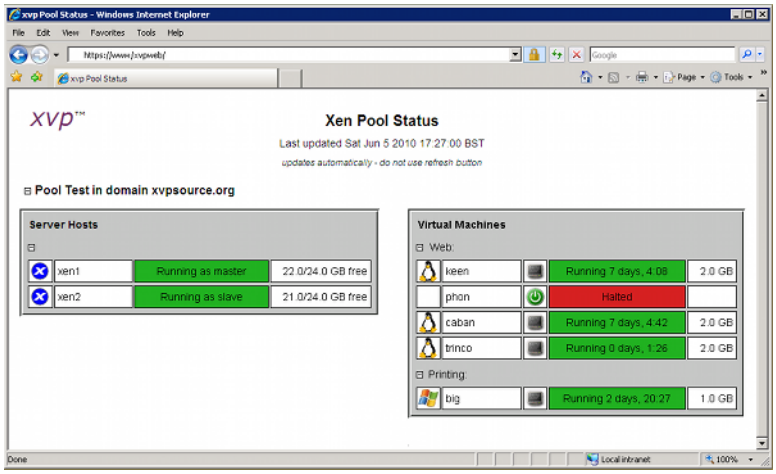

Figura 12: **xvp** (`http://www.xvpsource.org/`) para **XEN**

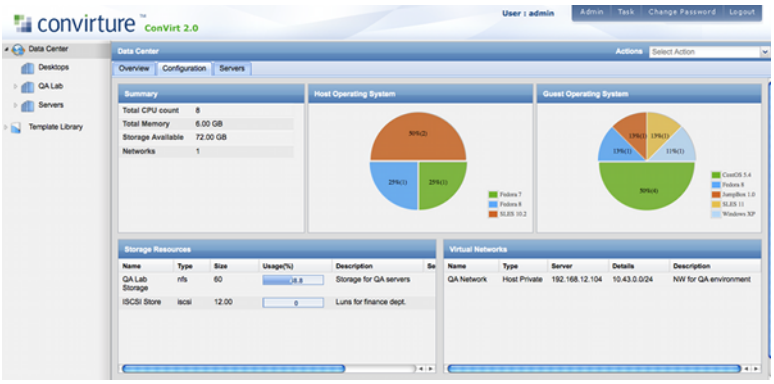

Figura 13: **ConVirt** (http://www.convirture.com/) para **XEN** y **KVM**,

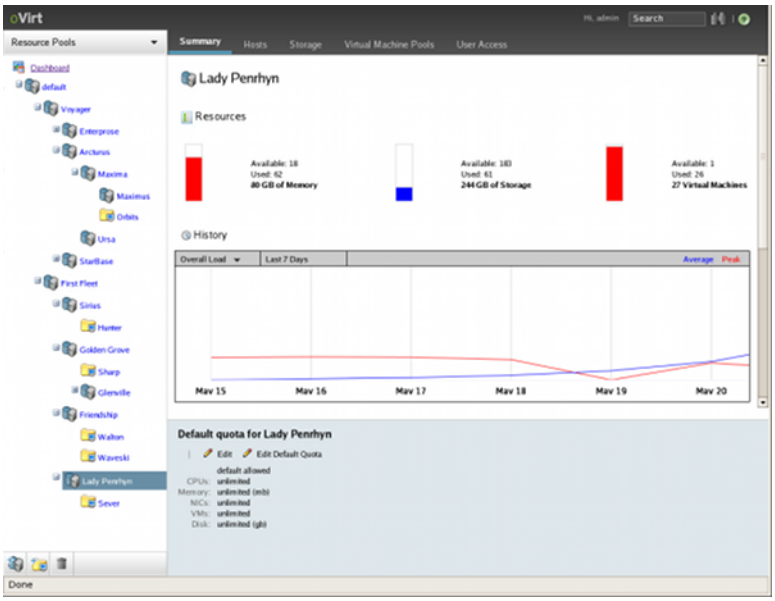

Figura 14: **oVirt** (https://fedorahosted.org/ovirt/) para los sistemas de virtualización soportados por **libvirt**

Además de las aplicaciones independientes también están empezando a aparecer distribuciones como **Proxmox VE** (http://pve.proxmox.com/) que proporcionan el soporte necesario para instalar y gestionar clusters de servidores virtuales (en concreto esta distribución soporta el uso de **KVM** y **OpenVZ**).

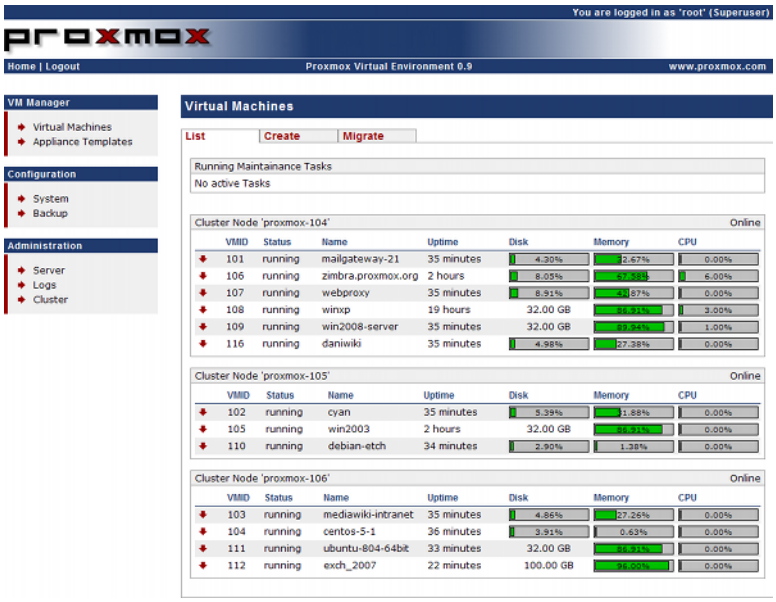

Figura 15: Listado de máquinas virtuales en un cluster ProxmoxVE

Alrededor de la virtualización de sistemas también se desarrollan sistemas para virtualizar otro tipo de recursos, como los discos o los dispositivos de red a nivel de sistema *anfitrión*, como por ejemplo el sistema de ethernet *distribuida virtual* (*Virtual Distributed Ethernet* <`http://vde.sourceforge.net/`>), que se puede usar para crear una red ethernet virtual entre sistemas **UML**, **QEMU** y **KVM** que se ejecuten en el mismo o en diferentes sistemas reales.

Por último es interesante indicar que el hecho de usar un sistema como Linux como *anfitrión* para la virtualización de sistemas nos permite emplear otras muchas tecnologías que son o pueden ser interesantes para montar infraestructuras virtuales como sistemas de ficheros con funcionalidades avanzadas (*RAID*, *LVM2*) o herramientas de gestión de red avanzadas (*iptables*, *iproute2* ó *brctl*).

Ejemplos de uso

A continuación comentaremos unos cuantos ejemplos de uso de tecnologías de virtualización, explicando las razones para elegir una tecnología u otra y como se organiza la gestión y el mantenimiento de los sistemas de virtualización dentro de las infraestructuras en las que se integran.

Prácticas de administración de sistemas con user-mode-linux

Hay situaciones en las que vale la pena emplear tecnologías como el `user-mode-linux` pese a que no tenga las mismas prestaciones que otros sistemas de virtualización.

Un buen ejemplo de esto se da cuando nos interesa que la herramienta de virtualización se ejecute a nivel de usuario, sin necesidad de dar permisos especiales a esos usuarios. Este es el caso, por ejemplo, cuando queremos realizar prácticas de administración de sistemas GNU/Linux i queremos ahorrarnos la reinstalación del sistema después de cada sesión.

Gracias a las capacidades del UML y a la existencia SLIRP (un programa a nivel de usuario que nos permite dar acceso a la red a los servidores UML) resulta rápido y económico dar a cada alumno una o varias máquinas virtuales para poder trabajar con ellas como administrador sin tener que darles ningún permiso especial en el sistema *anfitrión*.

Otra ventaja interesante del UML es que una vez instalado el software necesario en el sistema *anfitrión* no es precisa ninguna intervención de un administrador del equipo para crear, borrar o modificar las máquinas virtuales y resulta muy sencillo compartir datos entre la máquina virtual y el *anfitrión*, ya que el UML también soporta el acceso a directorios del sistema *anfitrión* desde el sistema *invitado*.

Por último comentar que hay una característica muy interesante del UML cuando queremos economizar recursos, en concreto nos referimos a la posibilidad del uso de ficheros COW con capas, que nos permiten generar un fichero base en el que hacer una instalación inicial de la máquina virtual que puede ser compartido por todos los alumnos (p. ej. accediendo al mismo usando un directorio compartido por NFS) de modo que ellos sólo necesiten crear un fichero adicional (p. ej. en su directorio de usuario) en el que se guarden los cambios respecto al sistema de ficheros base compartido, que en una sesión de prácticas suelen ser pocos.

Curso de instalación y configuración de servidores con VirtualBox

Si lo que nos interesa es que las máquinas virtuales puedan funcionar en múltiples sistemas operativos y que dispongan de interfaz gráfico (p. ej. para impartir un curso de instalación y configuración de servidores) **VirtualBox** es una buena opción, ya que tiene versiones libres que funcionan en distintos sistemas operativos (desde la web del proyecto se pueden descargar binarios del para sistemas **GNU/Linux**, **MacOS X**, **Solaris** y **OpenSolaris** o **Windows** ejecutándose sobre arquitecturas Intel de 32 y 64 bit) y es relativamente sencillo mover la máquina virtual de un sitio a otro; si los anfitriones son distintos es probable que tengamos que preparar dos configuraciones equivalentes, pero si son iguales es probable que sea suficiente con copiarlas y una vez tenemos las configuraciones el paso de un sistema a otro puede ser tan simple como copiar los ficheros que contienen las imágenes de los discos duros del sistema virtual.

Es interesante indicar que, al igual que en el caso de otras tecnologías, **VirtualBox** permite suspender las máquinas virtuales y hacer *fotos* (*snapshots*) del estado del sistema en un momento dado, lo que puede resultar muy útil cuando usamos la máquina para aprender a hacer algo, ya que podemos ir guardando el estado del sistema cada vez que completemos una tarea para volver atrás en el caso de que algo no funcione como debe.

Infraestructura de virtualización basada en linux-vserver

Hace más de cuatro años que en el ITI se planteó la necesidad de renovar las infraestructuras informáticas de uso común para el personal del centro y se decidió que lo que más interesaba era consolidar los servidores para poder retirar equipos antiguos y emplear virtualización para facilitar la escalabilidad y movilidad de los sistemas.

En su momento se compraron servidores con microprocesadores Intel y AMD de 64 bits sobre los que se instaló la versión estable del sistema operativo Debian GNU/Linux (en aquel momento la versión 4.0, con el nombre en código *etch*).

Esa versión incluía de serie paquetes binarios de la versión 2.6.18 del núcleo de Linux con soporte para **Xen** y **Linux-VServer**.

Como inicialmente sólo queríamos instalar servidores virtuales Linux optamos por el **Linux-VServer** por ser una tecnología más ligera y sencilla que **Xen**, sabiendo que en un futuro podríamos emplear ambas tecnologías en un mismo sistema (en **etch** había versiones del núcleo con soporte simultáneo de **Linux-VServer** y **XEN**).

Desde que comenzamos a utilizar el **Linux-VServer** definimos procedimientos para automatizar la instalación y configuración de los sistemas *anfitriones* e *invitados* para nuestro entorno.

El modelo empleado asumía que el sistema *anfitrión* ha de dar la misma imagen a los servidores virtuales que se ejecutan sobre él, por lo que definimos una serie de servicios y configuraciones que siempre deben ser iguales:

- En primer lugar hacemos que todos los vservers tengan sus interfaces de red ligadas al dispositivo virtual de red `dummy0` que está configurado en el sistema anfitrión para tener la IP 10.0.0.1 (usaremos esa dirección como IP local del *anfitrión* desde todos los servidores virtuales). Las direcciones de los servidores virtuales están siempre en la red privada `10.0.0.0/8`.
- Para dar acceso al exterior a los vservers empleamos un script de '*iptables* que hace NAT de salida para las direcciones internas de los servidores virtuales.

Para dar acceso a los servicios públicos que se ejecutan en los servidores virtuales usamos *port forwarding* (PAT) de entrada, las reglas las instala el mismo script de *iptables*.

- El *anfitrión* siempre ejecuta un servidor de SMTP local que escucha en la IP 10.0.0.1 y se encarga de modificar las direcciones de los mensajes que salen de los servidores virtuales (usamos la reescritura *canonical* de *postfix*, que cambia las direcciones de entrada y salida) y los envía a los *smarthosts* que correspondan (dentro del ITI se envían a nuestro servidor de correo principal).
- Opcionalmente el *anfitrión* puede ejecutar un servidor proxy de DNS (normalmente usamos `pdnsd` o `pdns-resolver`) que también funciona con la dirección 10.0.0.1 y nos independiza de la red en la que se ejecuta el servidor virtual, ya que no hace falta saber cual es la dirección del servidor de nombres de la red local.
- Como todos los servidores se compraron con dos discos, se montan con RAID-1 (*mirroring*) para garantizar que tenemos redundancia ante problemas físicos en uno de los discos.
- Sobre los RAID montamos sistemas de ficheros con LVM2, dejando siempre espacio para poder emplear *snapshots*, ya que nos resultan muy útiles para hacer copias de seguridad de los sistemas de archivos sin tener que parar los servicios. Lo que hacemos es asegurarnos de los ficheros en disco son correctos, tomamos una foto fija del sistema de ficheros (un *snapshot* de LVM2) y continuamos trabajando normalmente. El proceso de copia de seguridad se hace contra la foto fija (en nuestro caso usando `rdiff-backup`') y cuando se termina eliminamos el *snapshot* y el sistema continua funcionando normalmente sin ninguna interrupción.
- Para monitorizar el buen funcionamiento de los servicios locales del *anfitrión* utilizamos el programa `monit`, que está ajustado para monitorizar distintos parámetros del sistema y asegurar la disponibilidad de los servicios que consideramos necesarios.

En cuanto a los sistemas *invitados* lo que hacemos es aplicar un modelo similar al estándar de **OpenVZ**, es decir, hemos preparado un script para replicar una plantilla de un servidor virtual y modificar los cuatro parámetros básicos para que funcione de modo independiente.

Dentro de la plantilla instalamos siempre unos cuantos servicios ya configurados, incluyendo un servidor de `ssh` para conectar a los servidores virtuales directamente desde la red, un servidor `postfix` preparado para interactuar con el servidor del sistema *anfitrión* y un `monit` preparado para supervisar el funcionamiento de los servicios anteriores.

Para migrar o cambiar un servidor virtual de *anfitrión* usamos un proceso manual en el que copiamos los datos del servidor virtual, lo paramos,volvemos a sincronizarlo, ajustamos los scripts de red de ambos anfitriones para mover las direcciones IP públicas y las reglas de cortafuegos y lo levantamos en su nueva ubicación.

Infraestructura de virtualización basada en Proxmox (OpenVZ y KVM)

Aunque la infraestructura basada en `linux-vserver` funcionaba (y funciona) bien, existían algunos problemas con el sistema que nos hicieron plantearnos la migración a un sistema de virtualización diferente aprovechando todo lo posible el modelo ya definido:

- Por un lado parece que el desarrollo de `linux-vserver` está algo parado desde hace unos años, aunque desde que se tomó la decisión de cambiar las sucesivas versiones de **Debian** han seguido publicado versiones del núcleo con soporte para `linux-vserver`.
- No es posible hacer una suspensión ni una migración *en vivo* de los servidores virtuales. Aunque no es un problema grave, es una funcionalidad que resulta muy útil cuando queremos ajustar la carga de los servidores o poner en marcha sistemas de alta disponibilidad para los servidores o servicios.
- El uso de dispositivos de red con `linux-vserver` deja mucho que desear, sobre todo resulta muy molesto el hecho de que no se puedan crear interfaces de *loopback* dentro de los servidores virtuales.

- El uso de sistemas de cuotas para limitar el uso de CPU, disco, memoria, etc. no es especialmente potente con *linux-vserver*.
- Necesitamos disponer de una tecnología que nos permita ejecutar sistemas distintos a GNU/Linux, cosa que no es posible con **Linux-VServer**.

Después de evaluar las alternativas decidimos migrar los servidores que corren sobre **Linux-VServer** a **OpenVZ**, ya que se trata de una tecnología similar y que no fue la primera opción en su momento simplemente porqué la versión 4.0 de Debian no incluía paquetes con soporte de **OpenVZ**, lo que nos obligaba a aplicar parches y generar nuestros propios núcleos, algo que, sin ser difícil, intentamos evitar siempre que podemos para aprovechar las mejoras y correcciones de los núcleos de la distribución oficial.

Por otro lado, viendo la evolución de los sistemas de virtualización dentro del núcleo de Linux decidimos abandonar la idea de usar `Xen` prácticamente sin probarlo y optamos por emplear `KVM` para la virtualización de sistemas que no usen el núcleo de linux, ya que se trata de una tecnología mucho más simple de instalar y configurar y que sabíamos que si se iba a integrar en los núcleos oficiales de **Linux** y por tanto estaría soportada por los núcleos de las distribuciones de GNU/Linux *de serie*.

Además, habiendo decidido cambiar a estas tecnologías encontramos una distribución basada en **Debian** (**Proxmox**) que incluye de serie soporte para ambas y además nos proporciona un interfaz web para gestionar clusters de máquinas virtuales, por lo que decidimos utilizar su tecnología.

Como en realidad **Proxmox** se basa en **Debian** y el instalador que ellos emplean es menos potente que el de la distribución original lo que hacemos actualmente es hacer una instalación mínima de **Debian** configurándolo a nuestro gusto (RAID, LVM2, soporte de *bridging*, ...) y le añadimos los componentes de **Proxmox** que se emplean para gestionar la virtualización (versiones del núcleo y las herramientas estándar de openvz y kvm, ya que suelen ser más modernas que las de la distribución base y herramientas de administración web).

Aunque el interfaz gráfico de **Proxmox** nos resulta útil para muchas tareas lo cierto es hay cosas que hacemos siguiendo un proceso de más bajo nivel, por ejemplo para crear las máquinas virtuales **OpenVZ** empleamos las herramientas de línea de órdenes y plantillas de las distribuciones generadas por nosotros (normalmente la plantilla es una instalación de la versión estable de Debian GNU/Linux generada con una herramienta denominada **dab**).

Para las máquinas con **KVM** (generalmente **Windows**) usamos instalaciones base y las clonamos de modo similar a lo que se hace en redes locales que no usen virtualización, con la diferencia de que en este caso la *clonación* del sistema se limita a copiar ficheros de configuración y la imagen del disco duro.

En cuanto a la red, lo que hemos hecho es emplear *bridging* para los interfaces de red virtuales de ambos sistemas; la idea es que al usar *puentes* las máquinas virtuales se configuran como si fuesen sistemas reales (en **OpenVZ** los interfaces virtuales se ven como `eth0`, `eth1`, etc.), lo que simplifica enormemente la gestión.

De hecho en casi todos nuestros anfitriones configuramos tres puentes ethernet, uno que da acceso a la red pública del instituto, otro que da acceso a una red privada que interconecta todos los anfitriones y máquinas virtuales entre sí y otro que se monta sobre el `dummy0` y que permite que el *anfitrión* y sus *invitados* tengan un canal de comunicación directo que no pase por ninguna red *real*.

Actualmente estamos usando el montaje anterior para gestionar varios clusters de proxmox instalados en servidores Blade (tenemos un Sun Blade 6048 con 48 nodos instalados).

En algunos de los nodos sólo montamos uno o dos servidores virtuales **openvz** (son nodos de cálculo en los que se usa un sistema de colas para controlar la carga y la distribución de tareas, pero la virtualización nos sigue siendo útil para hacer snapshots, migrar nodos, etc.), en otros montamos varios servidores virtuales **KVM** (los usan para desarrollar y ejecutar programas en windows y si sólo se emplean uno o dos cores vale la pena lanzar varias máquinas a la vez) y en otros montamos múltiples servidores linux con **openvz** (para aplicaciones que no son de cálculo intensivo suele valer la pena tener varios servidores y de hecho con la virtualización es típico que cada proyecto que lo requiera tenga entre una y tres máquinas iguales para hacer desarrollo, preproducción y producción).

En cuanto al almacenamiento, de momento los sistemas virtuales se instalan en los discos locales de los anfitriones, aunque tenemos un RAID iSCSI y otro AoE a los que se da acceso empleando servidores de ficheros de red (NFS para los servidores OpenVZ y Samba para los servidores Windows), aunque en el futuro no descartamos ejecutar los sistemas

virtuales completos desde discos montados directamente de la red (eso resultaría especialmente interesante si tuviésemos servicios en alta disponibilidad).