Clúster, Cloud y DevOps

Remo Suppi Boldrito

PID_00212475

Universitat Oberta de Catalunya

Índice

Introducción

Los avances en la tecnología han llevado, por un lado, al desarrollo de procesadores más rápidos, con más de un elemento de cómputo (núcleos o *cores*) en cada uno de ellos, de bajo coste y con soporte a la virtualización *hardware* y por otro, al desarrollo de redes altamente eficientes. Esto, junto con el desarrollo de sistemas operativos como GNU/linux, ha favorecido un cambio radical en la utilización de sistemas de procesadores de múltiple-cores y altamente interconectados, en lugar de un único sistema de alta velocidad (como los sistemas vectoriales o los sistemas de procesamiento simétrico SMP). Además, estos sistemas han tenido una curva de despliegue muy rápida, ya que la relación precio/prestaciones ha sido, y lo es cada día más, muy favorable tanto para el responsable de los sistemas TIC de una organización como para los usuarios finales en diferentes aspectos: prestaciones, utilidad, fiabilidad, facilidad y eficiencia. Por otro lado, las crecientes necesidades de cómputo (y de almacenamiento) se han convertido en un elemento tractor de esta tecnología y su desarrollo, vinculados a la provisión dinámica de servicios, contratación de infraestructura y utilización por uso -incluso en minutos- (y no por compra o por alquiler), lo cual ha generado una evolución total e importante en la forma de diseñar, gestionar y administrar estos centros de cómputo. No menos importante han sido los desarrollos, además de los sistemas operativos, para soportar todas estas tecnologías, en lenguajes de desarrollo, API y entornos (*frameworks*) para que los desarrolladores puedan utilizar la potencialidad de la arquitectura subyacente para el desarrollo de sus aplicaciones, a los administradores y gestores para que para que puedan desplegar, ofrecer y gestionar servicios contratados y valorados por minutos de uso o bytes de E/S, por ejemplo, y a los usuarios finales para que puedan hacer un uso rápido y eficiente de los sistemas de cómputo sin preocuparse de nada de lo que existe por debajo, dónde está ubicado y cómo se ejecuta.

Es por ello por lo que, en el presente capítulo, se desarrollan tres aspectos básicos que son parte de la misma idea, a diferentes niveles, cuando se desea ofrecer la infraestructura de cómputo de altas prestaciones. O una plataforma como servicio o su utilización por un usuario final en el desarrollo de una aplicación, que permita utilizar todas estas infraestructuras de cómputo distribuidas y de altas prestaciones. En sistemas de cómputo de altas prestaciones (*High Performance Computing -HPC*), podemos distinguir dos grandes configuraciones:

1) Sistemas fuertemente acoplados (*tightly coupled systems*): son sistemas donde la memoria es compartida por todos los procesadores (*shared memory systems*) y la memoria de todos ellos "se ve" (por parte del programador) como una única memoria.

2) Sistemas débilmente acoplados (*loosely coupled systems*): no comparten memoria (cada procesador posee la suya) y se comunican mediante mensajes pasados a través de una red (*message passing systems*).

En el primer caso, son conocidos como *sistemas paralelos de cómputo* (*parallel processing system*) y en el segundo, como *sistemas distribuidos de cómputo* (*distributed computing systems*).

En la actualidad la gran mayoría de los sistemas (básicamente por su relación precio-prestaciones) son del segundo tipo y se conocen como clústers donde los sistemas de cómputo están interconectados por una red de gran ancho de banda y todos ellos trabajan en estrecha colaboración, viéndose para el usuario final como un único equipo. Es importante indicar que cada uno de los sistemas que integra el clúster a su vez puede tener más de un procesador con más de un *core* en cada uno de ellos y por lo cual, el programador deberá tener en cuenta estas características cuando desarrolle sus aplicaciones (lenguaje, memoria compartida|distribuida, niveles de caché, etc) y así poder aprovechar toda la potencialidad de la arquitectura agregada. El tipo más común de clúster es el llamado Beowulf, que es un clúster implementado con múltiples sistemas de cómputo (generalmente similares pero pueden ser heterogéneos) e interconectados por una red de área local (generalmente Ethernet, pero existen redes más eficientes como Infiniband o Myrinet). Algunos autores denominan *MPP (massively parallel processing)* cuando es un clúster, pero cuentan con redes especializadas de interconexión (mientras que los clústers utilizan hardware estándar en sus redes) y están formados por un alto número de recursos de cómputo (1.000 procesadores, no más). Hoy en día la mayoría de los sistemas publicados en el TOP500 (http://www.top500.org/system/177999) son clústers y en la última lista publicada (2014) ocupaba el primer puesto el ordenador Tianhe-2 (China) con 3,1 millones de cores, 1 Petabyte de memoria RAM (1000 Tbytes) y un consumo de 17 MW.

El otro concepto vinculado a los desarrollos tecnológicos mencionados a las nuevas formas de entender el mundo de las TIC en la actualidad es el *Cloud Computing* (o cómputo|servicios en la nube) que surge como concepto de ofrecer servicios de cómputo a través de Internet y donde el usuario final no tiene conocimiento de dónde se están ejecutando sus aplicaciones y tampoco necesita ser un experto para contactar, subir y ejecutar sus aplicaciones en minutos. Los proveedores de este tipo de servicio tienen recursos (generalmente distribuidos en todo el mundo) y permiten que sus usuarios contraten y gestionen estos recursos en forma *on-line* y sin la mayor intervención y en forma casi automática, lo cual permite reducir los costes teniendo ventajas (además de que no se tiene que instalar-mantener la infraestructura física -ni la obra civil-) como la fiabilidad, flexibilidad, rapidez en el aprovisionamiento, facilidad de uso, pago por uso, contratación de lo que se necesita, etc.. Obviamente, también cuenta con sus desventajas, como lo son la dependencia de un proveedor y la centralización de las aplicaciones|almacenamiento de datos, servicio vinculado a la disponibilidad de acceso a Internet, cuestiones de seguridad, ya

que los datos 'sensibles' del negocio no residen en las instalaciones de las empresas y pueden generar riesgos de sustracción/robo de información, confiabilidad de los servicios prestados por el proveedor (siempre es necesarios firmar una SLA -contrato de calidad de servicio-), tecnología susceptible al monopolio, servicios estándar (solo los ofrecidos por el proveedor), escalabilidad o privacidad.

Un tercer concepto vinculado a estas tecnologías, pero probablemente más del lado de los desarrolladores de aplicaciones y *frameworks*, es el de DevOps. DevOps surge de la unión de las palabras *Development* (desarrollo) y *Operations* (operaciones), y se refiere a una metodología de desarrollo de software que se centra en la comunicación, colaboración e integración entre desarrolladores de software y los profesionales de operaciones|administradores en las tecnologías de la información (IT). DevOps se presenta como una metodología que da respuesta a la relación existente entre el desarrollo de software y las operaciones IT, teniendo como objetivo que los productos y servicios software desarrollados por una entidad se puedan hacer más eficientemente, tengan una alta calidad y además sean seguros y fáciles de mantener. El término DevOps es relativamente nuevo y fue popularizado a través de *DevOps Open Days* (Bélgica, 2009) y como metodología de desarrollo ha ido ganando adeptos desde grande a pequeñas compañías, para hacer más y mejor sus productos y servicios y, sobre todo, debido a diferentes factores con una fuerte presencia hoy en día como: el uso de los procesos y metodologías de desarrollo ágil, necesidad de una mayor tasa de versiones, amplia disponibilidad de entornos virtualizados|*cloud*, mayor automatización de centros de datos y aumento de las herramientas de gestión de configuración.

En este módulo se verán diferentes formas de crear y programar un sistema de cómputo distribuido (clúster|*cloud*), las herramientas y librerías más importantes para cumplir este objetivo, y los conceptos y herramientas vinculadas a las metodológicas DevOps.

Objetivos

En los materiales didácticos de este módulo encontraréis los contenidos y las herramientas procedimentales para conseguir los objetivos siguientes:

1. Analizar las diferentes infraestructuras y herramientas para el cómputo de altas prestaciones (incluida la virtualización) (HPC).
2. Configurar e instalar un clúster de HPC y las herramientas de monitorización correspondientes.
3. Instalar y desarrollar programas de ejemplos en las principales API de programación: Posix Threads, OpenMPI, y OpenMP.
4. Instalar un clúster específico basado en una distribución ad hoc (Rocks).
5. Instalar una infraestructura para prestar servicios en *cloud* (IasS).
6. Herramientas DevOps para automatizar un centros de datos y gestiones de la configuración.

1. Clusterización

La historia de los sistemas informáticos es muy reciente (se puede decir que comienza en la década de 1960). En un principio, eran sistemas grandes, pesados, caros, de pocos usuarios expertos, no accesibles y lentos. En la década de 1970, la evolución permitió mejoras sustanciales llevadas a cabo por tareas interactivas (*interactive jobs*), tiempo compartido (*time sharing*), terminales y con una considerable reducción del tamaño. La década de 1980 se caracteriza por un aumento notable de las prestaciones (hasta hoy en día) y una reducción del tamaño en los llamados microordenadores. Su evolución ha sido a través de las estaciones de trabajo (*workstations*) y los avances en redes (LAN de 10 Mb/s y WAN de 56 kB/s en 1973 a LAN de 1/10 Gb/s y WAN con ATM, *asynchronous transfer mode* de 1,2 Gb/s en la actualidad o redes de alto rendimiento como Infiniband -96Gb/s- o Myrinet -10Gb/s-), que es un factor fundamental en las aplicaciones multimedia actuales y de un futuro próximo. Los sistemas distribuidos, por su parte, comenzaron su historia en la década de 1970 (sistemas de 4 u 8 ordenadores) y su salto a la popularidad lo hicieron en la década de 1990. Si bien su administración, instalación y mantenimiento pueden tener una cierta complejidad (cada vez menos) porque continúan creciendo en tamaño, las razones básicas de su popularidad son el incremento de prestaciones que presentan en aplicaciones intrínsecamente distribuidas (aplicaciones que por su naturaleza son distribuidas), la información compartida por un conjunto de usuarios, la compartición de recursos, la alta tolerancia a los fallos y la posibilidad de expansión incremental (capacidad de agregar más nodos para aumentar las prestaciones y de forma incremental). Otro aspecto muy importante en la actualidad es la posibilidad, en esta evolución, de la virtualización. Las arquitecturas cada vez más eficientes, con sistemas *multicores*, han permitido que la virtualización de sistemas se transforme en una realidad con todas las ventajas (y posibles desventajas) que ello comporta.

1.1. Virtualización

La virtualización es una técnica que está basada en la abstracción de los recursos de un ordenador, llamada *Hypervisor* o VMM (*Virtual Machine Monitor*) que crea una capa de separación entre el hardware de la máquina física (*host*) y el sistema operativo de la máquina virtual (*virtual machine*, *guest*), y es un medio para crear una "versión virtual" de un dispositivo o recurso, como un servidor, un dispositivo de almacenamiento, una red o incluso un sistema operativo, donde se divide el recurso en uno o más entornos de ejecución. Esta capa de software (VMM) maneja, gestiona y administra los cuatro recur-

sos principales de un ordenador (CPU, memoria, red y almacenamiento) y los reparte de forma dinámica entre todas las máquinas virtuales definidas en el computador central. De este modo nos permite tener varios ordenadores virtuales ejecutándose sobre el mismo ordenador físico.

La máquina virtual, en general, es un sistema operativo completo que se ejecuta como si estuviera instalado en una plataforma de hardware autónoma.

Enlace de interés

Para saber más sobre virtualización podéis visitar: http://en.wikipedia.org/wiki/Virtualization.
Se puede consultar una lista completa de programas de virtualización en: http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines.

1.1.1. Plataformas de virtualización

Los ejemplos más comunes de plataforma de virtualización con licencias GPLs o similiares son Xen, KVM, Qemu (emulación), OpenVz, VirtualBox, Oracle VM (solo el servidor), y entre las plataforma propietarias (algunas como *free to use*) VMware ESX/i, Virtual PC/Hyper-V, Parallels, Virtuozzo. Existen diferentes taxonomías para clasificar la virtualización (y algunas plataformas pueden ejecutarse en más de una categorías) siendo la más conocida la siguiente:

1) Virtualización por HW: considerando como *Full Virtualization* cuando la máquina virtual simula todo el HW para permitir a un sistema *guest* ejecutarse sin modificaciones (siempre y cuando esté diseñado para el mismo set de instrucciones). Fue la 1.ª generación de VM para procesadores x86 y lo que hace es una captura de instrucciones que acceden en modo prioritario a la CPU y las transforma en una llamada a la VM para que sean emuladas por software. En este tipo de virtualización pueden ejecutarse Parallels, VirtualBox, Virtual Iron, Oracle VM, Virtual PC, Hyper-V, VMware por ejemplo.

2) Virtualización asistida por hardware, donde el hardware provee soporte que facilita la construcción y trabajo del VM y mejora notablemente las prestaciones (a partir de a partir de 2005/6 Intel y AMD proveen este soporte como VT-x y AMD-V respectivamente). Las principales ventajas, respecto a otras formas de virtualización, es que no se debe tocar el sistema operativo *guest* (como en paravirtualization) y se obtienen mejores prestaciones, pero como contrapartida, se necesita soporte explícito en la CPU, lo cual no está disponible en todos los procesadores x86/86_64.En este tipo de virtualización pueden ejecutarse KVM, VMware Fusion, Hyper-V, Virtual PC, Xen, Parallels, Oracle VM, VirtualBox.

3) Paravirtualization: es una técnica de virtualización donde la VM no necesariamente simula el HW, sino que presenta una API que puede ser utilizada por el sistema *guest* (por lo cual, se debe tener acceso al código fuente para reemplazar las llamadas de un tipo por otro). Este tipo de llamadas a esta API se denominan *hypercall* y Xen puede ejecutarse en esta forma.

4) Virtualización parcial: es lo que se denomina *Address Space Virtualization*. La máquina virtual simula múltiples instancias del entorno subyacente del hardware (pero no de todo), particularmente el *address space*. Este tipo de virtualización acepta compartir recursos y alojar procesos, pero no permite instancias separadas de sistemas operativos *guest* y se encuentra en desuso actualmente.

5) Virtualización a nivel del sistema operativo: en este caso la virtualización permite tener múltiples instancias aisladas y seguras del servidor, todas ejecutándose sobre el mismo servidor físico y donde el sistema operativo *guest* coincidirá con el sistema operativo base del servidor (se utiliza el mismo kernel). Plataformas dentro de este tipo de virtualización son FreeBSD jails (el pionero), OpenVZ, Linux-VServer, LXC, Virtuozzo.

La diferencia entre instalar dos sistemas operativos y virtualizar dos sistemas operativos es que en el primer caso todos los sistemas operativos que tengamos instalados funcionarán de la misma manera que si estuvieran instalados en distintos ordenadores, y necesitaremos un gestor de arranque que al encender el ordenador nos permita elegir qué sistema operativo queremos utilizar, pero solo podremos tener funcionando simultáneamente uno de ellos. En cambio, la virtualización permite ejecutar muchas máquinas virtuales con sus sistemas operativos y cambiar de sistema operativo como si se tratase de cualquier otro programa; sin embargo, se deben valorar muy bien las cuestiones relacionadas con las prestaciones, ya que si el HW subyacente no es el adecuado, podremos notar muchas diferencias en las prestaciones entre el sistema operativo instalado en base o el virtualizado.

Entre las principales ventajas de la virtualización podemos contemplar:

1) Consolidación de servidores y mejora de la eficiencia de la inversión en HW con reutilización de la infraestructura existente.

2) Rápido despliegue de nuevos servicios con balanceo dinámico de carga y reducción de los sobredimensionamientos de la infraestructura.

3) Incremento de *Uptime* (tiempo que el sistema está al 100 % en la prestación de servicios), incremento de la tolerancia a fallos (siempre y cuando exista redundancia física) y eliminación del tiempo de parada por mantenimiento del sistema físico (migración de las máquinas virtuales).

4) Mantenimiento a coste aceptables de entornos software obsoletos pero necesarios para el negocio.

5) Facilidad de diseño y test de nuevas infraestructuras y entornos de desarrollo con un bajo impacto en los sistemas de producción y rápida puesta en marcha.

6) Mejora de TCO (*Total Cost of Ownership*) y ROI (*Return on Investment*).

7) Menor consumo de energía que en servidores físicos equivalentes.

Como desventajas podemos mencionar:

1) Aumenta la probabilidad de fallos si no se considera redundancia/alta disponibilidad (si se consolidan 10 servidores físicos en uno potente equivalente, con servidores virtualizados, y dejan de funcionar todos los servidores en él, dejarán de prestar servicio).

2) Rendimiento inferior (posible) en función de la técnica de virtualización utilizada y recursos disponibles.

3) Proliferación de servicios y máquinas que incrementan los gastos de administración/gestión (básicamente por el efecto derivado de la 'facilidad de despliegue' se tiende a tener más de lo necesarios).

4) Infraestructura desaprovechada (posible) ya que es habitual comprar una infraestructura mayor que la necesaria en ese momento para el posible crecimiento futuro inmediato.

5) Pueden existir problemas de portabilidad, hardware específico no soportado, y compromiso a largo término con la infraestructura adquirida.

6) Tomas de decisiones en la selección del sistema anfitrión puede ser complicada o condicionante.

Como es posible observar, las desventajas se pueden resolver con una planificación y toma de decisiones adecuada, y esta tecnología es habitual en la prestación de servicios y totalmente imprescindible en entornos de *Cloud Computing* (que veremos en este capítulo también).

1.2. Beowulf

Beowulf [3, 1, 2, 4] es una arquitectura multiordenador que puede ser utilizada para aplicaciones paralelas/distribuidas. El sistema consiste básicamente en un servidor y uno o más clientes conectados (generalmente) a través de Ethernet y sin la utilización de ningún hardware específico. Para explotar esta capacidad de cómputo, es necesario que los programadores tengan un modelo de programación distribuido que, si bien es posible mediante UNIX (Sockets, RPC), puede implicar un esfuerzo considerable, ya que son modelos de programación a nivel de *systems calls* y lenguaje C, por ejemplo; pero este modo de trabajo puede ser considerado de bajo nivel. Un modelo más avanzado en

esta línea de trabajo son los **Posix Threads**, que permiten explotar sistemas de memoria compartida y *multicores* de forma simple y fácil. La capa de software (interfaz de programación de aplicaciones, API) aportada por sistemas tales como ***Parallel Virtual Machine*** (PVM) y ***Message Passing Interface*** (MPI) facilita notablemente la abstracción del sistema y permite programar aplicaciones paralelas/distribuidas de modo sencillo y simple. Una de las formas básicas de trabajo es la de maestro–trabajadores (*master-workers*), en que existe un servidor (maestro) que distribuye la tarea que realizarán los trabajadores. En grandes sistemas (por ejemplo, de 1.024 nodos) existe más de un maestro y nodos dedicados a tareas especiales, como por ejemplo entrada/salida o monitorización. Otra opción no menos interesante, sobre todo con el auge de procesadores *multicores*, es **OpenMP** que es una API para la programación multiproceso de memoria compartida. Esta capa de software permite añadir concurrencia a los programas sobre la base del modelo de ejecución *fork-join* y se compone de un conjunto de directivas de compilador, rutinas de biblioteca y variables de entorno, que influencian el comportamiento en tiempo de ejecución y proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas y arquitecturas de CPU que puedan ejecutar más de un hilo de ejecución simultáneo (hyperthreading) o que dispongan de más de un núcleo por procesador accediendo a la misma memoria compartida.

Existen dos conceptos que pueden dar lugar a dudas y que son Cluster Beowulf y COW (*Cluster of Workstations*). Una de las principales diferencias es que Beowulf "se ve" como una única máquina donde se accede a los nodos remotamente, ya que no disponen de terminal (ni de teclado), mientras que un COW es una agrupación de ordenadores que pueden ser utilizados tanto por los usuarios de la COW como por otros usuarios en forma interactiva, a través de su pantalla y teclado. Hay que considerar que Beowulf no es un software que transforma el código del usuario en distribuido ni afecta al núcleo del sistema operativo (como por ejemplo, Mosix). Simplemente, es una forma de agrupación (un clúster) de máquinas que ejecutan GNU/Linux y actúan como un super-ordenador. Obviamente, existe una gran cantidad de herramientas que permiten obtener una configuración más fácil, bibliotecas o modificaciones al núcleo para obtener mejores prestaciones, pero es posible construir un clúster Beowulf a partir de un GNU/Linux estándar y de software convencional. La construcción de un clúster Beowulf de dos nodos, por ejemplo, se puede llevar a cabo simplemente con las dos máquinas conectadas por Ethernet mediante un concentrador (*hub*), una distribución de GNU/ Linux estándar (Debian), el sistema de archivos compartido (NFS) y tener habilitados los servicios de red, como por ejemplo `ssh`. En estas condiciones, se puede argumentar que se dispone de un clúster simple de dos nodos. En cambio, en un COW no necesitamos tener conocimientos sobre la arquitectura subyacente (CPU, red) necesaria para hacer una aplicación distribuida en Beowulf pero sí que es necesario tener un sistema operativo (por ejemplo, Mosix) que permita este tipo de compartición de recursos y distribución de tareas (además de un API específica para programar la aplicación que es necesaria en los dos

sistemas). El primer Beowulf se construyó en 1994 y en 1998 comienzan a aparecer sistemas Beowulf/linux en las listas del Top500 (Avalon en la posición 314 con 68 cores y 19,3 Gflops, junio'98).

1.2.1. ¿Cómo configurar los nodos?

Primero se debe modificar el archivo `/etc/hosts` para que contenga la línea de `localhost` (con 127.0.0.1) y el resto de los nombres a las IP internas de los restantes nodos (este archivo deberá ser igual en todos los nodos). Por ejemplo:

```
    127.0.0.1 localhost
    192.168.0.254   lucix-server
Y añadir las IP de los nodos (y para todos los nodos), por ejemplo:
    192.168.0.1 lucix1
    192.168.0.2 lucix2
...
Se debe crear un usuario (nteum) en todos los nodos, crear un grupo y añadir este usuario al grupo:
    groupadd beowulf
    adduser nteum beowulf
    echo umask 007 >> /home/nteum/.bash_profile
```

Así, cualquier archivo creado por el usuario nteum o cualquiera dentro del grupo será modificable por el grupo beowulf. Se debe crear un servidor de NFS (y los demás nodos serán clientes de este NFS) y exportar el directorio */home* así todos los clientes verán el directorio *$HOME* de los usuarios. Para ello sobre el servidor se edita el */etc/exports* y se agrega una línea como `/home lucix*(rw,sync,no_root_squash)`. Sobre cada nodo cliente se monta agregando en el */etc/fstab* para que en el arranque el nodo monte el directorio como `192.168.0.254:/home /home nfs defaults 0 0`, también es aconsejable tener en el servidor un directorio */soft* (donde se instalará todo el software para que lo vean todos los nodos así nos evitamos de tener que instalar este en cada nodo), y agregarlos al *export* como `/soft lucix*(rw,async,no_root_squash)` y en cada nodo agregamos en el *fstab* `192.168.0.254:/soft /soft nfs defaults 0 0`. También deberemos hacer la configuración del */etc/resolv.conf* y el `iptables` (para hacer un NAT) sobre el servidor si este tiene acceso a Internet y queremos que los nodos también tengan acceso (es muy útil sobre todo a la hora de actualizar el sistema operativo).

A continuación verificamos que los servicios están funcionando (tened en cuenta que el comando `chkconfig` puede no estar instalado en todas las distribuciones) (en Debian hacer `apt-get install chkconfig`):

```
chkconfig --list sshd
chkconfig --list nfs
```

Que deben estar a "on" en el nivel de trabajo que estemos (generalmente el nivel 3 pero se puede averiguar con el comando `runlevel`) y sino se deberán hacer las modificaciones necesarias para que estos *daemons* se inicien en este nivel (por ejemplo, `chkconfig name_service on; service`

`name_service start`). Si bien estaremos en un red privada, para trabajar de forma segura es importante trabajar con `ssh` y nunca con `rsh` o `rlogin` por lo cual deberíamos generar las claves para interconectar en modo seguro las máquinas-usuario nteum sin `passwd`. Para eso modificamos (quitamos el comentario `#`), de `/etc/ssh/sshd_config` en las siguientes líneas:

```
RSAAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys
```

Reiniciamos el servicio (`service sshd restart`) y nos conectamos con el usuario que deseamos y generamos las llaves:

```
ssh-keygen -t rsa
```

En el directorio *$HOME/.ssh* se habrán creado los siguientes archivos: `id_rsa` y `id_rsa.pub` en referencia a la llave privada y pública respectivamente. Podemos de forma manual copiar `id_rsa.pub` en un archivo con nombre `authorized_keys` en el mismo directorio (ya que al estar compartido por NFS será el mismo directorio que verán todas los nodos y verificamos los permisos: 600 para el directorio *.ssh*, *authorized_keys* e *id_rsa* y 644 para *id_rsa.pub*. Es conveniente también instalar NIS sobre el clúster así evitamos tener que definir cada usuario en los nodos y un servidor DHCP para definir todos los parámetros de red de cada nodo que que cada nodo solicite estos durante la etapa de *boot*, para ello consultad en el capítulo de servidores y el de red de la asignatura de Administración GNU/Linux donde se explica con detalle estas configuraciones.

A partir de aquí ya tenemos un clúster Beowulf para ejecutar aplicaciones con interfaz a MPI para aplicaciones distribuidas (también puede ser aplicaciones con interfaz a PVM, pero es recomendable MPI por cuestiones de eficiencia y prestaciones). Existen sobre diferentes distribuciones (Debian, FC incluidas) la aplicación `system-config-cluster`, que permite configurar un clúster en base a una herramienta gráfica o `clusterssh` o `dish`, que permiten gestionar y ejecutar comandos en un conjunto de máquinas de forma simultánea (comandos útiles cuando necesitamos hacer operaciones sobre todos los nodos del clúster, por ejemplo, apagarlos, reiniciarlos o hacer copias de un archivo a todos ellos).

Enlace de interés

Información Adicional:
https://wiki.debian.org
/HighPerformanceComputing

1.3. Beneficios del cómputo distribuido

¿Cuáles son los beneficios del cómputo en paralelo? Veremos esto con un ejemplo [3]. Consideremos un programa para sumar números (por ejemplo, $4 + 5 + 6 + \ldots$) llamado `sumdis.c`:

```
#include <stdio.h>

int main (int argc, char** argv){
long inicial, final, resultado, tmp;
    if (argc < 2) {
   printf (" Uso: %s N° inicial N° final\n",argv[0]);
     return (4); }
    else {
   inicial = atoll(argv[1]);
   final = atoll(argv[2]);
   resultado = 0;}
for (tmp = inicial; tmp <= final; tmp++){resultado += tmp; };
    printf("%lu\n", resultado);
    return 0;
}
```

Lo compilamos con `gcc -o sumdis sumdis.c` y si miramos la ejecución de este programa con, por ejemplo,

```
 time ./sumdis 1 1000000
500000500000

    real 0m0.005s
    user 0m0.000s
    sys     0m0.004s
```

se podrá observar que el tiempo en una máquina Debian 7.5 sobre una máquina virtual (Virtualbox) con procesador i7 es 5 milésimas de segundo. Si, en cambio, hacemos desde 1 a 16 millones, el tiempo real sube hasta 0,050s, es decir, 10 veces más, lo cual, si se considera 1600 millones, el tiempo será del orden de unos 4 segundos.

La idea básica del cómputo distribuido es repartir el trabajo. Así, si disponemos de un clúster de 4 máquinas (lucix1–lucix4) con un servidor y donde el archivo ejecutable se comparte por NFS, sería interesante dividir la ejecución mediante ssh de modo que el primero sume de 1 a 400.000.000, el segundo, de 400.000.001 a 800.000.000, el tercero, de 800.000.001 a 1.200.000.000 y el cuarto, de 1.200.000.001 a 1.600.000.000. Los siguientes comandos muestran una posibilidad. Consideramos que el sistema tiene el directorio `/home` compartido por NFS y el usuario **adminp**, que tiene adecuadamente configurado las llaves para ejecutar el código sin *password* sobre los nodos, ejecutará:

```
    mkfifo out1     Crea una cola fifo en /home/adminp
    ./distr.sh & time cat out1 | awk '{total + = $1 } END {printf "%lf", total}'
```

Se ejecuta el comando `distr.sh`; se recolectan los resultados y se suman mientras se mide el tiempo de ejecución. El *shell script* `distr.sh` puede ser algo como:

```
ssh lucix1 /home/nteum/sumdis 1 400000000 > /home/nteum/out1 < /dev/null &
ssh lucix2 /home/nteum/sumdis 400000001 800000000 > /home/nteum/out1 < /dev/null &
ssh lucix3 /home/nteum/sumdis 800000001 1200000000 > /home/nteum/out1 < /dev/null &
ssh lucix4 /home/nteum/sumdis 1200000001 1600000000 > /home/nteum/out1 < /dev/null &
```

Podremos observar que el tiempo se reduce notablemente (aproximadamente en un valor cercano a 4) y no exactamente de forma lineal, pero muy próxima. Obviamente, este ejemplo es muy simple y solo válido para fines demostrativos. Los programadores utilizan bibliotecas que les permiten realizar el tiempo de ejecución, la creación y comunicación de procesos en un sistema distribuido (por ejemplo MPI u OpenMP).

1.3.1. ¿Cómo hay que programar para aprovechar la concurrencia?

Existen diversas maneras de expresar la concurrencia en un programa. Las tres más comunes son:

1) Utilizando hilos (o procesos) en el mismo procesador (multiprogramación con solapamiento del cómputo y la E/S).

2) Utilizando hilos (o procesos) en sistemas *multicore*.

3) Utilizando procesos en diferentes procesadores que se comunican por medio de mensajes (MPS, *Message Passing System*).

Estos métodos pueden ser implementados sobre diferentes configuraciones de hardware (memoria compartida o mensajes) y, si bien ambos métodos tienen sus ventajas y desventajas, los principales problemas de la memoria compartida son las limitaciones en la escalabilidad (ya que todos los *cores*/procesadores utilizan la misma memoria y el número de estos en el sistema está limitado por el ancho de banda de la memoria) y, en los los sistemas de paso de mensajes, la latencia y velocidad de los mensajes en la red. El programador deberá evaluar qué tipo de prestaciones necesita, las características de la aplicación subyacente y el problema que se desea solucionar. No obstante, con los avances de las tecnologías de *multicores* y de red, estos sistemas han crecido en popularidad (y en cantidad). Las API más comunes hoy en día son Posix Threads y OpenMP para memoria compartida y MPI (en sus versiones OpenMPI o Mpich) para paso de mensajes. Como hemos mencionado anteriormente, existe otra biblioteca muy difundida para pasos de mensajes, llamada PVM, pero que la versatilidad y prestaciones que se obtienen con MPI ha dejado relegada a aplicaciones pequeñas o para aprender a programar en sistemas distribuidos. Estas bibliotecas, además, no limitan la posibilidad de utilizar hilos (aunque a nivel local) y tener concurrencia entre procesamiento y entrada/salida.

Para realizar una aplicación paralela/distribuida, se puede partir de la versión serie o mirando la estructura física del problema y determinar qué partes pueden ser concurrentes (independientes). Las partes concurrentes serán candidatas a re-escribirse como código paralelo. Además, se debe considerar si es posible reemplazar las funciones algebraicas por sus versiones paralelizadas (por ejemplo, ScaLapack *Scalable Linear Algebra Package* (se puede probar en Debian

los diferentes programas de prueba que se encuentran en el paquete scalapack-mpi-test, por ejemplo y directamente, instalar las librerías libscalapack-mpi-dev). También es conveniente averiguar si hay alguna aplicación similar paralela que pueda orientarnos sobre el modo de construcción de la aplicación paralela*.

*http://www.mpich.org/

Paralelizar un programa no es una tarea fácil, ya que se debe tener en cuenta la **ley de Amdahl**, que afirma que el incremento de velocidad (*speedup*) está limitado por la fracción de código (f), que puede ser paralelizado de la siguiente manera:

$$\text{speedup} = \frac{1}{1-f}$$

Esta ley implica que con una aplicación secuencial $f = 0$ y el speedup = 1, mientras que con todo el código paralelo $f = 1$ y el speedup se hace infinito. Si consideramos valores posibles, un 90 % ($f = 0{,}9$) del código paralelo significa un speedup igual a 10, pero con $f = 0{,}99$ el speedup es igual a 100. Esta limitación se puede evitar con algoritmos escalables y diferentes modelos de programación de aplicación (paradigmas):

1) Maestro-trabajador: el maestro inicia a todos los trabajadores y coordina su trabajo y el de entrada/salida.

2) *Single Process Multiple Data* (SPMD): mismo programa que se ejecuta con diferentes conjuntos de datos.

3) Funcional: varios programas que realizan una función diferente en la aplicación.

En resumen, podemos concluir:

1) Proliferación de máquinas multitarea (multiusuario) conectadas por red con servicios distribuidos (NFS y NIS).

2) Son sistemas heterogéneos con sistemas operativos de tipo NOS (*Networked Operating System*), que ofrecen una serie de servicios distribuidos y remotos.

3) La programación de aplicaciones distribuidas se puede efectuar a diferentes niveles:

a) Utilizando un modelo cliente-servidor y programando a bajo nivel (*sockets*) o utilizando memoria compartida a bajo nivel (Posix Threads).

b) El mismo modelo, pero con API de "alto" nivel (OpenMP, MPI).

c) Utilizando otros modelos de programación como, por ejemplo, programación orientada a objetos distribuidos (RMI, CORBA, Agents, etc.).

1.4. Memoria compartida. Modelos de hilos (*threading*)

Normalmente, en una arquitectura cliente-servidor, los clientes solicitan a los servidores determinados servicios y esperan que estos les contesten con la mayor eficacia posible. Para sistema distribuidos con servidores con una carga muy alta (por ejemplo, sistemas de archivos de red, bases de datos centralizadas o distribuidas), el diseño del servidor se convierte en una cuestión crítica para determinar el rendimiento general del sistema distribuido. Un aspecto crucial en este sentido es encontrar la manera óptima de manejar la E/S, teniendo en cuenta el tipo de servicio que ofrece, el tiempo de respuesta esperado y la carga de clientes. No existe un diseño predeterminado para cada servicio, y escoger el correcto dependerá de los objetivos y restricciones del servicio y de las necesidades de los clientes.

Las preguntas que debemos contestar antes de elegir un determinado diseño son: ¿Cuánto tiempo se tarda en un proceso de solicitud del cliente? ¿Cuántas de esas solicitudes es probable que lleguen durante ese tiempo? ¿Cuánto tiempo puede espera el cliente? ¿Cuánto afecta esta carga del servidor a las prestaciones del sistema distribuido? Además, con el avance de la tecnología de procesadores nos encontramos con que disponemos de sistemas *multicore* (múltiples núcleos de ejecución) que pueden ejecutar secciones de código independientes. Si se diseñan los programas en forma de múltiples secuencias de ejecución y el sistema operativo lo soporta (y GNU/Linux es uno de ellos), la ejecución de los programas se reducirá notablemente y se incrementarán en forma (casi) lineal las prestaciones en función de los *cores* de la arquitectura.

1.4.1. Multihilos (*multithreading*)

Las últimas tecnologías en programación para este tipo aplicaciones (y así lo demuestra la experiencia) es que los diseños más adecuados son aquellos que utilizan modelos de multi-hilos (*multithreading models*), en los cuales el servidor tiene una organización interna de procesos paralelos o hilos cooperarantes y concurrentes.

Un hilo (*thread*) es una secuencia de ejecución (hilo de ejecución) de un programa, es decir, diferentes partes o rutinas de un programa que se ejecutan concurrentemente en un único procesador y accederán a los datos compartidos al mismo tiempo.

¿Qué ventajas aporta esto respecto a un programa secuencial? Consideremos que un programa tiene tres rutinas A, B y C. En un programa secuencial, la rutina C no se ejecutará hasta que se hayan ejecutado A y B. Si, en cambio, A, B y C son hilos, las tres rutinas se ejecutarán concurrentemente y, si en ellas hay E/S, tendremos concurrencia de ejecución con E/S del mismo programa (proceso), cosa que mejorará notablemente las prestaciones de dicho programa. Generalmente, los hilos están contenidos dentro de un proceso y diferentes hilos de un mismo proceso pueden compartir algunos recursos, mientras que diferentes procesos no. La ejecución de múltiples hilos en paralelo necesita el soporte del sistema operativo y en los procesadores modernos existen optimizaciones del procesador para soportar modelos multihilo (*multithreading*) además de las arquitectura multicores donde existe múltiples núcleo y donde cada uno de ellos puede ejecutar un thread.

Generalmente, existen cuatro modelos de diseño por hilos (en orden de complejidad creciente):

1) Un hilo y un cliente: en este caso el servidor entra en un bucle sin fin escuchando por un puerto y ante la petición de un cliente se ejecutan los servicios en el mismo hilo. Otros clientes deberán esperar a que termine el primero. Es fácil de implementar pero solo atiende a un cliente a la vez.

2) Un hilo y varios clientes con selección: en este caso el servidor utiliza un solo hilo, pero puede aceptar múltiples clientes y multiplexar el tiempo de CPU entre ellos. Se necesita una gestión más compleja de los puntos de comunicación (*sockets*), pero permite crear servicios más eficientes, aunque presenta problemas cuando los servicios necesitan una alta carga de CPU.

3) Un hilo por cliente: es, probablemente, el modelo más popular. El servidor espera por peticiones y crea un hilo de servicio para atender a cada nueva petición de los clientes. Esto genera simplicidad en el servicio y una alta disponibilidad, pero el sistema no escala con el número de clientes y puede saturar el sistema muy rápidamente, ya que el tiempo de CPU dedicado ante una gran carga de clientes se reduce notablemente y la gestión del sistema operativo puede ser muy compleja.

4) Servidor con hilos en granja (*worker threads*): este método es más complejo pero mejora la escalabilidad de los anteriores. Existe un número fijo de hilos trabajadores (*workers*) a los cuales el hilo principal distribuye el trabajo de los clientes. El problema de este método es la elección del número de trabajadores: con un número elevado, caerán las prestaciones del sistema por saturación; con un número demasiado bajo, el servicio será deficiente (los clientes deberán esperar). Normalmente, será necesario sintonizar la aplicación para trabajar con un determinado entorno distribuido.

Existen diferentes formas de expresar a nivel de programación con hilos: paralelismo a nivel de tareas o paralelismo a través de los datos. Elegir el modelo

adecuado minimiza el tiempo necesario para modificar, depurar y sintonizar el código. La solución a esta disyuntiva es describir la aplicación en términos de dos modelos basados en un trabajo en concreto:

- Tareas paralelas con hilos independientes que pueden atender tareas independientes de la aplicación. Estas tareas independientes serán encapsuladas en hilos que se ejecutarán asincrónicamente y se deberán utilizar bibliotecas como Posix Threads (Linux/Unix) o Win32 Thread API (Windows), que han sido diseñadas para soportar concurrencia a nivel de tarea.

- Modelo de datos paralelos para calcular lazos intensivos; es decir, la misma operación debe repetirse un número elevado de veces (por ejemplo comparar una palabra frente a las palabras de un diccionario). Para este caso es posible encargar la tarea al compilador de la aplicación o, si no es posible, que el programador describa el paralelismo utilizando el entorno OpenMP, que es una API que permite escribir aplicaciones eficientes bajo este tipo de modelos.

Una aplicación de información personal (*Personal Information Manager*) es un buen ejemplo de una aplicación que contiene concurrencia a nivel de tareas (por ejemplo, acceso a la base de datos, libreta de direcciones, calendario, etc.). Esto podría ser en pseudocódigo:

```
Function addressBook;
Function inBox;
Function calendar;
Program PIM     {
    CreateThread (addressBook);
    CreateThread (inBox);
    CreateThread (calendar);   }
```

Podemos observar que existen tres ejecuciones concurrentes sin relación entre ellas. Otro ejemplo de operaciones con paralelismo de datos podría ser un corrector de ortografía, que en pseudocódigo sería: `Function SpellCheck {loop (word = 1, words_in_file) compareToDictionary (word);}`

Se debe tener en cuenta que ambos modelos (hilos paralelos y datos paralelos) pueden existir en una misma aplicación. A continuación se mostrará el código de un productor de datos y un consumidor de datos basado en Posix Threads. Para compilar sobre Linux, por ejemplo, se debe utilizar `gcc -o pc pc.c -lptread`.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define QUEUESIZE 10
#define LOOP 20

void *producer (void *args);
```

```
void *consumer (void *args);
typedef struct { /* Estructura del buffer compartido y descriptores de threads */
int buf[QUEUESIZE]; long head, tail; int full, empty;
pthread_mutex_t *mut; pthread_cond_t *notFull, *notEmpty;
    } queue;
queue *queueInit (void);   /* Prototipo de función: inicialización del buffer */
void queueDelete (queue *q); /* Prototipo de función: Borrado del buffer*/
void queueAdd (queue *q, int in); /* Prototipo de función: insertar elemento en el buffer */
void queueDel (queue *q, int *out); /* Prototipo de función: quitar elemento del buffer */

int main () {
queue *fifo; pthread_t pro, con; fifo = queueInit ();
if (fifo ==  NULL) { fprintf (stderr, " Error al crear buffer.\n");  exit (1);   }
pthread_create (&pro, NULL, producer, fifo);   /* Creación del thread productor */
pthread_create (&con, NULL, consumer, fifo); /* Creación del thread consumidor*/
pthread_join (pro, NULL);   /* main () espera hasta que terminen ambos threads */
pthread_join (con, NULL);
queueDelete (fifo);       /* Eliminación del buffer compartido */
return 0; }               /* Fin */

void *producer (void *q) { /*Función del productor */
queue *fifo; int i;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) {    /* Inserto en el buffer elementos=LOOP*/
pthread_mutex_lock (fifo->mut);   /* Semáforo para entrar a insertar */
while (fifo->full) {
printf ("Productor: queue FULL.\n");
pthread_cond_wait (fifo->notFull, fifo->mut);  }
/* Bloqueo del productor si el buffer está lleno, liberando el semáforo mut
    para que pueda entrar el consumidor. Continuará cuando el consumidor ejecute
    pthread_cond_signal (fifo->notFull);*/
         queueAdd (fifo, i); /* Inserto elemento en el buffer */
                   pthread_mutex_unlock (fifo->mut); /* Libero el semáforo */
         pthread_cond_signal (fifo->notEmpty);/*Desbloqueo consumidor si está bloqueado*/
         usleep (100000);  /* Duermo 100 mseg para permitir que el consumidor se active */
}
return (NULL); }

void *consumer (void *q) { /*Función del consumidor */
queue *fifo; int i, d;
fifo = (queue *)q;
for (i = 0; i < LOOP; i++) { /* Quito del buffer elementos=LOOP*/
pthread_mutex_lock (fifo->mut); /* Semáforo para entrar a quitar */
while (fifo->empty) {
printf (" Consumidor: queue EMPTY.\n");
pthread_cond_wait (fifo->notEmpty, fifo->mut); }
    /* Bloqueo del consumidor si el buffer está vacío, liberando el semáforo mut
    para que pueda entrar el productor. Continuará cuando el consumidor ejecute
    pthread_cond_signal (fifo->notEmpty);*/
queueDel (fifo, &d); /* Quito elemento del buffer */
pthread_mutex_unlock (fifo->mut); /* Libero el semáforo */
pthread_cond_signal (fifo->notFull); /*Desbloqueo productor si está bloqueado*/
printf (" Consumidor: Recibido %d.\n", d);
usleep(200000);/* Duermo 200 mseg para permitir que el productor se active */
}
return (NULL); }

queue *queueInit (void) {
queue *q;
q = (queue *)malloc (sizeof (queue)); /* Creación del buffer */
if (q == NULL) return (NULL);
q->empty = 1;  q->full = 0;  q->head = 0;  q->tail = 0;
q->mut = (pthread_mutex_t *) malloc (sizeof (pthread_mutex_t));
pthread_mutex_init (q->mut, NULL); /* Creación del semáforo */
q->notFull = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notFull, NULL); /* Creación de la variable condicional notFull*/
q->notEmpty = (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
pthread_cond_init (q->notEmpty, NULL);   /* Creación de la variable condicional notEmpty*/
return (q); }

void queueDelete (queue *q) {
pthread_mutex_destroy (q->mut);  free (q->mut);
```

```
pthread_cond_destroy (q->notFull); free (q->notFull);
pthread_cond_destroy (q->notEmpty); free (q->notEmpty);
free (q); }

void queueAdd (queue *q, int in) {
q->buf[q->tail] = in; q->tail++;
if (q->tail == QUEUESIZE) q->tail = 0;
if (q->tail == q->head) q->full = 1;
q->empty = 0;
return; }

void queueDel (queue *q, int *out){
*out = q->buf[q->head]; q->head++;
if (q->head == QUEUESIZE) q->head = 0;
if (q->head == q->tail) q->empty = 1;
q->full = 0;
return; }
```

1.5. OpenMP

El OpenMP (*Open-Multi Processing*) es una interfaz de programación de aplicaciones (API) con soporte multiplataforma para la programación en C/C++ y Fortran de procesos con uso de memoria compartida sobre plataformas Linux/Unix (y también Windows). Esta infraestructura se compone de un conjunto de directivas del compilador, rutinas de la biblioteca y variables de entorno que permiten aprovechar recursos compartidos en memoria y en tiempo de ejecución. Definido conjuntamente por un grupo de los principales fabricantes de hardware y software, OpenMP permite utilizar un modelo escalable y portátil de programación, que proporciona a los usuarios un interfaz simple y flexible para el desarrollo, sobre plataformas paralelas, de aplicaciones de escritorio hasta aplicaciones de altas prestaciones sobre superordenadores. Una aplicación construida con el modelo híbrido de la programación paralela puede ejecutarse en un ordenador utilizando tanto OpenMP como Message Passing Interface (MPI) [5].

OpenMP es una implementación multihilo, mediante la cual un hilo maestro divide la tareas sobre un conjunto de hilos trabajadores. Estos hilos se ejecutan simultáneamente y el entorno de ejecución realiza la asignación de estos a los diferentes procesadores de la arquitectura. La sección del código que está diseñada para funcionar en paralelo está marcada con una directiva de preprocesamiento que creará los hilos antes que la sección se ejecute. Cada hilo tendrá un identificador (`id`) que se obtendrá a partir de una función (`omp_get_thread_num()` en C/C++) y, después de la ejecución paralela, los hilos se unirán de nuevo en su ejecución sobre el hilo maestro, que continuará con la ejecución del programa. Por defecto, cada hilo ejecutará una sección paralela de código independiente pero se pueden declarar secciones de "trabajo compartido" para dividir una tarea entre los hilos, de manera que cada hilo ejecute parte del código asignado. De esta forma, es posible tener en un programa OpenMP paralelismo de datos y paralelismo de tareas conviviendo conjuntamente.

Los principales elementos de OpenMP son las sentencias para la creación de hilos, la distribución de carga de trabajo, la gestión de datos de entorno, la sincronización de hilos y las rutinas a nivel de usuario. OpenMP utiliza en C/C++ las directivas de preprocesamiento conocidas como *pragma* (`#pragma omp <resto del pragma>`) para diferentes construcciones. Así por ejemplo, `omp parallel` se utiliza para crear hilos adicionales para ejecutar el trabajo indicado en la sentencia paralela donde el proceso original es el hilo maestro (`id=0`). El conocido programa que imprime "Hello, world" utilizando OpenMP y multihilos es*:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]){
   #pragma omp parallel
   printf("Hello, world.\n");
   return 0;}
```

***Compilad con** `gcc -fopenmp -o hello hello.c` **(en algunas distribuciones -debian está instalada por defecto- se debe tener instalada la biblioteca GCC OpenMP (GOMP)** `apt-get install libgomp1`

Donde ejecutará un thread por cada core disponible en la arquitectura. Para especificar *work-sharing constructs* se utiliza:

- **omp for o omp do**: reparte las iteraciones de un lazo en múltiples hilos.
- **sections**: asigna bloques de código independientes consecutivos a diferentes hilos.
- **single**: especifica que el bloque de código será ejecutado por un solo hilo con una sincronización (*barrier*) al final del mismo.
- **master**: similar a `single`, pero el código del bloque será ejecutado por el hilo maestro y no hay *barrier* implicado al final.

Por ejemplo, para inicializar los valores de un *array* en paralelo utilizando hilos para hacer una porción del trabajo (compilad, por ejemplo, con `gcc -fopenmp -o init2 init2.c`):

```
#include <stdio.h>
#include <omp.h>
#define N 1000000
int main(int argc, char *argv[]) {
   float a[N]; long i;
    #pragma omp parallel for
   for (i=0;i<N;i++)  a[i]= 2*i;
return 0;
}
```

Si ejecutamos con el *pragma* y después lo comentamos y calculamos el tiempo de ejecución (`time ./init2`), vemos que el tiempo de ejecución pasa de 0.003 s a 0.007 s, lo que muestra la utilización del *dualcore* del procesador.

Ya que OpenMP es un modelo de memoria compartida, muchas variables en el código son visibles para todos los hilos por defecto. Pero a veces es necesario tener variables privadas y pasar valores entre bloques secuenciales del código y bloques paralelos, por lo cual es necesario definir atributos a los datos (*data clauses*) que permitan diferentes situaciones:

- **shared**: los datos en la región paralela son compartidos y accesibles por todos los hilos simultáneamente.

- **private**: los datos en la región paralela son privados para cada hilo, y cada uno tiene una copia de ellos sobre una variable temporal.

- **default**: permite al programador definir cómo serán los datos dentro de la región paralela (`shared`, `private` o `none`).

Otro aspecto interesante de OpenMP son las directivas de sincronización:

- **critical section**: el código enmarcado será ejecutado por hilos, pero solo uno por vez (no habrá ejecución simultánea) y se mantiene la exclusión mutua.
- **atomic**: similar a `critical section`, pero avisa al compilador para que use instrucciones de hardware especiales de sincronización y así obtener mejores prestaciones.
- **ordered**: el bloque es ejecutado en el orden como si de un programa secuencial se tratara.
- **barrier**: cada hilo espera que los restantes hayan acabado su ejecución (implica sincronización de todos los hilos al final del código).
- **nowait**: especifica que los hilos que terminen el trabajo asignado pueden continuar.

Además, OpenMP provee de sentencias para la planificación (*scheduling*) del tipo `schedule(type, chunk)` (donde el tipo puede ser `static`, `dynamic` o `guided`) o proporciona control sobre la sentencias `if`, lo que permitirá definir si se paraleliza o no en función de si la expresión es verdadera o no. OpenMP también proporciona un conjunto de funciones de biblioteca, como por ejemplo:

- **omp_set_num_threads**: define el número de hilos a usar en la siguiente región paralela.
- **omp_get_num_threads**: obtiene el número de hilos que se están usando en una región paralela.
- **omp_get_max_threads**: obtiene la máxima cantidad posible de hilos.
- **omp_get_thread_num**: devuelve el número del hilo.
- **omp_get_num_procs**: devuelve el máximo número de procesadores que se pueden asignar al programa.
- **omp_in_parallel**: devuelve un valor distinto de cero si se ejecuta dentro de una región paralela.

Veremos a continuación algunos ejemplos simples (compilad con la instrucción `gcc -fopenmp -o out_file input_file.c`):

```
/* Programa simple multihreading con OpenMP */
#include <omp.h>
int main()  {
     int iam =0, np = 1;

     #pragma omp parallel private(iam, np)
     #if defined (_OPENMP)
         np = omp_get_num_threads();
         iam = omp_get_thread_num();
     #endif

     printf("Hello from thread %d out of %d \n", iam, np);
     }
 }

 /* Programa simple con Threads anidados con OpenMP */
 #include <omp.h>
 #include <stdio.h>
     main(){
         int x=0,nt,tid,ris;

         omp_set_nested(2);
         ris=omp_get_nested();
         if (ris) printf("Paralelismo anidado activo %d\n", ris);
         omp_set_num_threads(25);
         #pragma omp parallel private (nt,tid) {
             tid = omp_get_thread_num();
             printf("Thread %d\n",tid);
             nt = omp_get_num_threads();
             if (omp_get_thread_num()==1)
             printf("Número de Threads: %d\n",nt);
             }
}

 /* Programa simple de integración con OpenMP */
 #include <omp.h>
 #include <stdio.h>
 #define N 100
     main() {
    double local, pi=0.0, w; long i;
    w = 1.0 / N;
    #pragma omp parallel private(i, local)
        {
        #pragma omp single
        pi = 0.0;
        #pragma omp for reduction(+: pi)
        for (i = 0; i < N; i++) {
          local = (i + 0.5)*w;
          pi = pi + 4.0/(1.0 + local*local);
          printf ("Pi: %f\n",pi);
            }
        }
     }

 /* Programa simple de reducción con OpenMP */
 #include <omp.h>
 #include <stdio.h>
 #define NUM_THREADS 2
     main () {
    int i; double ZZ, res=0.0;
         omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel for reduction(+:res) private(ZZ)
        for (i=0; i< 1000; i++) {
       ZZ = i*i;
       res = res + ZZ;
       printf("ZZ: %f, res: %f\n", ZZ, res);}
               }
         }
```

Nota

Se recomienda ver también los ejemplos que se pueden encontrar en la referencia [6].

1.6. MPI, *Message Passing Interface*

La definición de la API de MPI [9, 10] ha sido el trabajo resultante del MPI Forum (MPIF), que es un consorcio de más de 40 organizaciones. MPI tiene influencias de diferentes arquitecturas, lenguajes y trabajos en el mundo del paralelismo, como por ejemplo: WRC (Ibm), Intel NX/2, Express, nCUBE, Vertex, p4, Parmac y contribuciones de ZipCode, Chimp, PVM, Chamaleon, PICL.

El principal objetivo de MPIF fue diseñar una API, sin relación particular con ningún compilador ni biblioteca, y que permitiera la comunicación eficiente (*memory-to-memory copy*), cómputo y comunicación concurrente y descarga de comunicación, siempre y cuando exista un coprocesador de comunicaciones. Además, se pedía que soportara el desarrollo en ambientes heterogéneos, con interfaz C y F77 (incluyendo C++, F90), donde la comunicación fuera fiable y los fallos, resueltos por el sistema. La API también debía tener interfaz para diferentes entornos, disponer una implementación adaptable a diferentes plataformas con cambios insignificantes y que no interfiera con el sistema operativo (*thread-safety*). Esta API fue diseñada especialmente para programadores que utilizaran el *Message Passing Paradigm* (MPP) en C y F77, para aprovechar su característica más relevante: la portabilidad. El MPP se puede ejecutar sobre máquinas multiprocesador, redes de estaciones de trabajo e incluso sobre máquinas de memoria compartida. La primera versión del estándar fue MPI-1 (que si bien hay muchos desarrollos sobre esta versión, se considera en EOL), la versión MPI-2 incorporó un conjunto de mejoras, como creación de procesos dinámicos, *one-sided communication,* entrada/salida paralela entre otras, y finalmente la última version, MPI-3 (considerada como una revisión mayor), incluye *nonblocking collective operations, one-sided operations* y soporte para Fortran 2008.[7]

Muchos aspectos han sido diseñados para aprovechar las ventajas del hardware de comunicaciones sobre SPC (*scalable parallel computers*) y el estándar ha sido aceptado en forma mayoritaria por los fabricantes de hardware en paralelo y distribuido (SGI, SUN, Cray, HPConvex, IBM, etc.). Existen versiones libres (por ejemplo, Mpich, LAM/MPI y openMPI) que son totalmente compatibles con las implementaciones comerciales realizadas por los fabricantes de hardware e incluyen comunicaciones punto a punto, operaciones colectivas y grupos de procesos, contexto de comunicaciones y topología, soporte para F77 y C y un entorno de control, administración y *profiling*. [11, 12, 10]

Pero existen también algunos puntos que pueden presentar algunos problemas en determinadas arquitecturas, como son la memoria compartida, la ejecución remota, las herramientas de construcción de programas, la depuración, el control de hilos, la administración de tareas y las funciones de entrada/salida concurrentes (la mayor parte de estos problemas de falta de herramientas están resueltos a partir de la versión 2 de la API -MPI2-). Una de los

problemas de MPI1, al no tener creación dinámica de procesos, es que solo soporta modelos de programación MIMD (*Multiple Instruction Multiple Data*) y comunicándose vía llamadas MPI. A partir de MPI-2 y con la ventajas de la creación dinámica de procesos, ya se pueden implementar diferentes paradigmas de programación como *master-worker/farmer-tasks*, *divide & conquer*, paralelismo especulativo, etc. (o al menos sin tanta complejidad y mayor eficiencia en la utilización de los recursos).

Para la instalación de MPI se recomienda utilizar la distribución (en algunos casos la compilación puede ser compleja debido a las dependencias de otros paquetes que puede necesitar. Debian incluye la versión OpenMPI (sobre Debian 7.5 es version 2, pero se pueden bajar los fuentes y compilarlos) y Mpich2 (Mpich3 disponible en http://www.mpich.org/downloads/). La mejor elección será OpenMPI, ya que combina las tecnologías y los recursos de varios otros proyectos (FT-MPI, LA-MPI, LAM/MPI y PACX-MPI) y soporta totalmente el estándar MPI-2 (y desde la versión 1.75 soporta la versión MPI3). Entre otras características de OpenMPI tenemos: es conforme a MPI-2/3, *thread safety & concurrency*, creación dinámica de procesos, alto rendimiento y gestión de trabajos tolerantes a fallos, instrumentación en tiempo de ejecución, *job schedulers*, etc. Para ello se deben instalar los paquetes `openmpi-dev`, `openmpi-bin`, `openmpi-common` y `openmpi-doc`. Además, Debian 7.5 incluye otra implementación de MPI llamada LAM (paquetes `lam*`). Se debe considerar que si bien las implementaciones son equivalentes desde el punto de vista de MPI, tienen diferencias en cuanto a la gestión y procedimientos de compilación/ejecución/gestión.

1.6.1. Configuración de un conjunto de máquinas para hacer un clúster adaptado a OpenMPI

Para la configuración de un conjunto de máquinas para hacer un clúster adaptado a OpenMPI [14], se han de seguir los siguientes pasos:

1) Hay que tener las máquinas "visibles" (por ejemplo a través de un `ping`) a través de TCP/IP (IP pública/privada).

2) Es recomendable que todas las máquinas tengan la misma arquitectura de procesador, así es más fácil distribuir el código, con versiones similares de Linux (a ser posible con el mismo tipo de distribución).

3) Se recomienda tener NIS o si no se debe generar un mismo usuario (por ejemplo, mpiuser) en todas las máquinas y el mismo directorio `$HOME` montado por NFS.

4) Nosotros llamaremos a las máquinas como `slave1`, `slave2`, etc., (ya que luego resultará más fácil hacer las configuraciones) pero se puede llamar a las máquinas como cada uno prefiera.

5) Una de las máquinas será el maestro y las restantes, *slaveX*.

6) Se debe instalar en todos los nodos (supongamos que tenemos Debian): `openmpi-bin`, `openmpi-common`, `openmpi-dev`. Hay que verificar que en todas las distribuciones se trabaja con la misma versión de OpenMPI.

7) En Debian los ejecutables están en `/usr/bin` pero si están en un *path* diferente, deberá agregarse a mpiuser y también verificar que `LD_LIBRARY_PATH` apunta a `/usr/lib`.

8) En cada nodo esclavo debe instalarse el SSH server (instalad el paquete `openssh-server`) y sobre el maestro, el cliente (paquete `openssh-client`).

9) Se deben crear las claves públicas y privadas haciendo `ssh-keygen -t dsa` y copiar a cada nodo con `ssh-copy-id` para este usuario (solo se debe hacer en un nodo, ya que, como tendremos el directorio $HOME compartido por NFS para todos los nodos, con una copia basta).

10) Si no se comparte el directorio hay que asegurar que cada esclavo conoce que el usuario mpiuser se puede conectar sin passwd, por ejemplo haciendo: `ssh slave1`.

11) Se debe configurar la lista de las máquinas sobre las cuales se ejecutará el programa, por ejemplo `/home/mpiuser/.mpi_hostfile` y con el siguiente contenido:

```
# The Hostfile for Open MPI
# The master node, slots=2 is used because it is a dual-processor machine.
    localhost slots=2
# The following slave nodes are single processor machines:
    slave1
    slave2
    slave3
```

12) OpenMPI permite utilizar diferentes lenguajes, pero aquí utilizaremos C. Para ello hay que ejecutar sobre el maestro `mpicc testprogram.c`. Si se desea ver que incorpora mpicc, se puede hacer `mpicc -showme`.

13) Para ejecutar en local podríamos hacer `mpirun -np 2 ./myprogram` y para ejecutar sobre los nodos remotos (por ejemplo 5 procesos) `mpirun -np 2 -hostfile ./mpi_hostfile ./myprogram`.

Es importante notar que `np` es el número de procesos o procesadores en que se ejecutará el programa y se puede poner el número que se desee, ya que OpenMPI intentará distribuir los procesos de forma equilibrada entre todas las máquinas. Si hay más procesos que procesadores, OpenMPI/Mpich utilizará las características de intercambio de tareas de GNU/Linux para simular la ejecución paralela. A continuación se verán dos ejemplos: Srtest es un programa simple para establecer comunicaciones entre procesos punto a punto, y cpi calcula el valor del número π de forma distribuida (por integración).

```
/* Srtest Progam */
#include "mpi.h"
#include <stdio.h>
```

```
#include <string.h>
#define BUFLEN 512

int main(int argc, char *argv[]){
    int myid, numprocs, next, namelen;
    char buffer[BUFLEN], processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
    MPI_Init(&argc,&argv);  /* Debe ponerse antes de otras llamadas MPI, siempre */
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);    /*Integra el proceso en un grupo de comunicaciones*/
    MPI_Get_processor_name(processor_name,&namelen); /*Obtiene el nombre del procesador*/

    fprintf(stderr,"Proceso %d sobre %s\n", myid, processor_name);
    fprintf(stderr,"Proceso %d de %d\n", myid, numprocs);
    strcpy(buffer,"hello there");
    if (myid == numprocs-1) next = 0;
    else next = myid+1;

    if (myid == 0) { /*Si es el inicial, envía string de buffer*/
        printf("%d sending '%s' \n",myid,buffer);fflush(stdout);
    MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
        /*Blocking Send, 1:buffer, 2:size, 3:tipo, 4:destino, 5:tag, 6:contexto*/
    printf("%d receiving \n",myid);fflush(stdout);
    MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
    printf("%d received '%s' \n",myid,buffer);fflush(stdout);
       /* mpdprintf(001,"%d receiving \n",myid); */
        }
    else {
       printf("%d receiving  \n",myid);fflush(stdout);
   MPI_Recv(buffer, BUFLEN, MPI_CHAR, MPI_ANY_SOURCE, 99, MPI_COMM_WORLD,&status);
       /* Blocking Recv, 1:buffer, 2:size, 3:tipo, 4:fuente, 5:tag, 6:contexto, 7:status*/
   printf("%d received '%s' \n",myid,buffer);fflush(stdout);
   /* mpdprintf(001,"%d receiving \n",myid); */
   MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, next, 99, MPI_COMM_WORLD);
   printf("%d sent '%s' \n",myid,buffer);fflush(stdout);
       }
    MPI_Barrier(MPI_COMM_WORLD); /*Sincroniza todos los procesos*/
    MPI_Finalize(); /*Libera los recursos y termina*/
    return (0);
    }



/* CPI Program */
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a) { return (4.0 / (1.0 + a*a)); }
int main( int argc, char *argv[] ) {
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264338327950288419716939937510;
    double mypi, pi, h, sum, x; double startwtime = 0.0, endwtime;
    int namelen; char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /*Indica el número de procesos en el grupo*/
    MPI_Comm_rank(MPI_COMM_WORLD,&myid); /*Id del proceso*/
    MPI_Get_processor_name(processor_name,&namelen); /*Nombre del proceso*/
    fprintf(stderr, "Proceso %d sobre %s\n", myid, processor_name);
    n = 0;
    while (!done) {
        if (myid ==0) { /*Si es el primero...*/
            if (n ==0) n = 100; else n = 0;
            startwtime = MPI_Wtime();} /* Time Clock */
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD); /*Broadcast al resto*/
            /*Envía desde el 4 arg. a todos los procesos del grupo Los restantes que no son 0
            copiarán el buffer desde 4 o arg -proceso 0-*/
            /*1:buffer, 2:size, 3:tipo, 5:grupo */
        if (n == 0) done = 1;
        else {h = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
```

```
                x = h * ((double)i - 0.5); sum += f(x); }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
                /* Combina los elementos del Send Buffer de cada proceso del grupo usando la
                operación MPI_SUM y retorna el resultado en el Recv Buffer. Debe ser llamada
                por todos los procesos del grupo usando los mismos argumentos*/
                /*1:sendbuffer, 2:recvbuffer, 3:size, 4:tipo, 5:oper, 6:root, 7:contexto*/
            if (myid == 0){ /*solo el P0 imprime el resultado*/
                printf("Pi es aproximadamente %.16f, el error es %.16f\n", pi,fabs(pi - PI25DT));
                endwtime = MPI_Wtime();
                printf("Tiempo de ejecución = %f\n", endwtime-startwtime); }
            }
        }
        MPI_Finalize(); /*Libera recursos y termina*/
        return 0;
    }
```

Para visualizar la ejecución de un código paralelo/distribuido en MPI existe una aplicación llamada XMPI (en Debian xmpi) que permite 'ver', durante la ejecución, el estado de la ejecución de las aplicaciones sobre MPI, pero está vinculada al paquete LAM/MPI. Para visualizar y analizar código sobre OpenMPI, se debería (y es recomendable) bajar y compilar el código de MPE (http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/) o TAU (http://www.cs.uoregon.edu/research/tau/home.php) que son herramientas de *profiling* muy potentes y que no requieren gran trabajo ni dedicación para ponerlas en marcha.

1.7. Rocks Cluster

Rocks Cluster es una distribución de Linux para clústers de computadores de alto rendimiento. Las versiones actuales de Rocks Cluster están basadas en CentOS (CentOS 6.5 a julio 2014) y, como instalador, Anaconda con ciertas modificaciones, que simplifica la instalación 'en masa' en muchos ordenadores. Rocks Cluster incluye muchas herramientas (tales como MPI) que no forman parte de CentOS pero son los componentes que transforman un grupo de ordenadores en un clúster. Las instalaciones pueden personalizarse con paquetes de software adicionales llamados *rolls*. Los *rolls* extienden el sistema integrando automáticamente los mecanismos de gestión y empaquetamiento usados por el software básico, y simplifican ampliamente la instalación y configuración de un gran número de computadores. Se han creado una gran cantidad de *rolls*, como por ejemplo SGE *roll*, Cóndor *roll*, Xen *roll*, el Java *roll*, Ganglia *roll*, etc. (http://www.rocksclusters.org/wordpress/?page_id=4). Rocks Cluster es una distribución altamente empleada en el ámbito de clústers, por su facilidad de instalación e incorporación de nuevos nodos y por la gran cantidad de programas para el mantenimiento y monitorización del clúster.

Enlaces de interés

Para una lista detallada de las herramientas incluidas en Rocks Cluster, consultad: http://www.rocksclusters.org/roll-documentation/base/5.5/.

Las principales características de Rocks Cluster son:

1) Facilidad de instalación, ya que solo es necesario completar la instalación de un nodo llamado *nodo maestro* (*frontend*), el resto se instala con Avalache,

que es un programa P2P que lo hace de forma automática y evita tener que instalar los nodos uno a uno.

2) Disponibilidad de programas (conjunto muy amplio de programas que no es necesario compilar y transportar) y facilidad de mantenimiento (solo se debe mantener el nodo maestro).

3) Diseño modular y eficiente, pensado para minimizar el tráfico de red y utilizar el disco duro propio de cada nodo para solo compartir la información mínima e imprescindible.

La instalación se puede seguir paso a paso desde el sitio web de la distribución [15] y los autores garantizan que no se tarda más de una hora para una instalación básica. Rocks Cluster permite, en la etapa de instalación, diferentes módulos de software, los *rolls*, que contienen todo lo necesario para realizar la instalación y la configuración de sistema con estas nuevas 'adiciones' de forma automática y, además, decidir en el *frontend* cómo será la instalación en los nodos esclavos, qué *rolls* estarán activos y qué arquitectura se usará. Para el mantenimiento, incluye un sistema de copia de respaldo del estado, llamado *Roll Restore*. Este *roll* guarda los archivos de configuración y *scripts* (e incluso se pueden añadir los archivos que se desee).

1.7.1. Guía rápida de instalación

Este es un resumen breve de la instalación propuesta en [15] para la versión 6.1 y se parte de la base que el *frontend* tiene (mínimo) 30 GB de disco duro, 1 GB de RAM, arquitectura x86-64 y 2 interfaces de red (eth0 para la comunicación con internet y eth1 para la red interna); para los nodos 30 GB de disco duro, 512 MB de RAM y 1 interfaz de red (red interna). Después de obtener los discos *Kernel/Boot Roll, Base Roll, OS Roll CD1/2* (o en su defecto DVD equivalente), insertamos *kernel boot*, seguimos los siguientes pasos:

Enlaces de interés

Se puede descargar los discos desde: http://www.rocksclusters.org/wordpress/?page_id=80

1) Arrancamos el *frontend* y veremos una pantalla en la cual introducimos `build` y nos preguntará la configuración de la red (IPV4 o IPV6).

2) El siguiente paso es seleccionar los `rolls` (por ejemplo seleccionando los "CD/DVD-based Roll" y vamos introduciendo los siguientes *rolls*) y marcar en las sucesivas pantallas cuáles son los que deseamos.

3) La siguiente pantalla nos pedirá información sobre el clúster (es importante definir bien el nombre de la máquina, *Fully-Qualified Host Name*, ya que en caso contrario fallará la conexión con diferentes servicios) y también información para la red privada que conectará el *frontend* con los nodos y la red pública (por ejemplo, la que conectará el *frontend* con Internet) así como DNS y pasarelas.

4) A continuación se solicitará la contraseña para el root, la configuración del servicio de tiempo y el particionado del disco (se recomienda escoger "auto").

5) Después de formatear los discos, solicitará los CD de *rolls* indicados e instalará los paquetes y hará un *reboot* del *frontend*.

6) Para instalar los nodos se debe entrar como root en el *frontend* y ejecutar `insert-ethers`, que capturará las peticiones de DHCP de los nodos y los agregará a la base de datos del *frontend*, y seleccionar la opción *Compute* (por defecto, consultad la documentación para las otras opciones).

7) Encendemos el primer nodo y en el *boot order* de la BIOS generalmente se tendrá CD, PXE (Network Boot), Hard Disk, (si el ordenador no soporta PXE, entonces haced el *boot* del nodo con el *Kernel Roll CD*). En el *frontend* se verá la petición y el sistema lo agregará como `compute-0-0` y comenzará la descarga e instalación de los archivos. Si la instalación falla, se deberán reiniciar los servicios httpd, mysqld y autofs en el *frontend*.

8) A partir de este punto se puede monitorizar la instalación ejecutando la instrucción `rocks-console`, por ejemplo con `rocks-console compute-0-0`. Después de que se haya instalado todo, se puede salir de `insert-ethers` pulsando la tecla F8. Si se dispone de dos *racks*, después de instalado el primero se puede comenzar el segundo haciendo `insert-ethers -cabinet=1` los cuales recibirán los nombres `compute-1-0`, `compute-1-1`, etc.

9) A partir de la información generada en consola por `rocks list host`, generaremos la información para el archivo `machines.conf`, que tendrá un aspecto como:

```
nteum slot=2
compute-0-0 slots=2
compute-0-1 slots=2
compute-0-2 slots=2
compute-0-3 slots=2
```

y que luego deberemos ejecutar con

```
mpirun -np 10 -hostfile ./machines.conf ./mpi_program_to_execute.
```

1.8. FAI

FAI es una herramienta de instalación automatizada para desplegar Linux en un clúster o en un conjunto de máquinas en forma desatendida. Es equivalente a *kickstart* de RH, o Alice de SuSE. FAI puede instalar Debian, Ubuntu y RPMs de distribuciones Linux. Sus ventajas radican en que mediante esta herramienta se puede desplegar Linux sobre un nodo (físico o virtual) simplemente haciendo que arranque por PXE y quede preparado para trabajar y totalmente configurado (cuando se dice uno, se podría decir 100 con el mismo esfuerzo por parte del administrador) y sin ninguna interacción por medio. Por lo tanto, es un método escalable para la instalación y actualización de un clúster Beowulf o una red de estaciones de trabajo sin supervisión y con poco

esfuerzo. FAI utiliza la distribución Debian, una colección de scripts (su mayoría en Perl) y *cfengine* y *Preseeding d-i* para el proceso de instalación y cambios en los archivos de configuración.

Es una herramienta pensada para administradores de sistemas que deben instalar Debian en decenas o cientos de ordenadores y puede utilizarse además como herramienta de instalación de propósito general para instalar (o actualizar) un clúster de cómputo HPC, de servidores web, o un pool de bases de datos y configurar la gestión de la red y los recursos en forma desatendida. Esta herramienta permite tratar (a través de un concepto que incorpora llamado clases) con un hardware diferente y diferentes requisitos de instalación en función de la máquina y características que se disponga en su preconfiguración, permitiendo hacer despliegues masivos eficientes y sin intervención de administrador (una vez que la herramienta haya sido configurada adecuadamente). [29, 30, 31]

1.8.1. Guía rápida de instalación

En este apartado veremos una guía de la instalación básica sobre máquinas virtuales (en Virtualbox) para desplegar otras máquinas virtuales, pero se puede adaptar/ampliar a las necesidades del entorno con mínimas intervenciones y es una herramienta que no cuenta con *daemons*/DB y solo consiste en scripts (consultar las referencias indicadas).

Instalación del servidor. Si bien existe un paquete llamado `fai-quickstart` es mejor hacerlo por partes ya que muchos de los paquetes ya están instalados y sino se pueden instalar cuando es necesario (ver más info en http://fai-project.org/fai-guide/_anchor_id_inst_xreflabel_inst_installing_fai.html).

Descarga de los paquetes: en Debian Wheezy están todos los paquetes. Es mejor bajarse la última versión de http://fai-project.org/download/wheezy/ y aunque se deben descargar todos los paquetes `fai-alguna-cosa-.deb` solo utilizaremos `fai-server` y `fai-doc` en este ejemplo.

Instalación de la llave (*key*):

```
wget -O - http://fai-project.org/download/074BCDE4.asc | sudo
apt-key add -
```

Instalación de los paquetes: dentro del directorio donde se encuentren y como root ejecutar

```
dkpg -i fai-server_x.y_all.deb; dkpg -i fai-doc_x.y_all.deb
```

reemplazando la 'x.y' por la versión descargada.

Configurar el DHCP del servidor: el *boot* de los nodos y la transferencia del sistema operativo la realizaremos por una petición PXE y quien primero debe actuar es el `dhcp` para asignarle al nodo los parámetros de red y el archivo de boot, por lo que deberemos modificar */etc/dhcp/dhcpd.conf* agregando las primitivas **next-server** y **filename** en las zonas de nuestra red:

```
...
authoritative;
...
subnet 192.168.168.0 netmask 255.255.255.0 {
  ...
  next-server 192.168.168.254;
  filename "fai/pxelinux.0";
  ...
}
```

Configuración del servicio TFTP: la transferencia de los archivos del SO se realizará por el servicio TFTP (*Trivial File Transfer Protocol*) por lo que deberemos comprobar si el `tftp-hpa` está instalado y bien configurado con `netstat -anp|grep :69`. Deberá dar algo similar a :

```
udp 0 0 0.0.0.0:69 0.0.0.0:*
```

Si no, habrá que verificar si está instalado (`dpkg -l | grep tftp`) y en caso de que no esté instalado, ejecutar `apt-get install tftpd-hpa`. A continuación verificar que la configuración es la adecuada */etc/default/tftpd-hpa* (y recordar de reiniciar el servicio si se modifica):

```
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="--secure"
```

Crear la configuración de FAI: ejecutando como root (Los archivos de configuración se encontrarán en */srv/fai/config* y hay reglas para modificarlos -ver la documentación-):

```
fai-setup
echo '/srv/fai/config 192.168.168.0/24(async,ro,no_subtree_check,no_root_squash)' >> /etc/exports
/etc/init.d/nfs-kernel-server restart

Con un exportfs veremos algo como:
    /home          192.168.168.0/24
    ...
    /srv/fai/nfsroot 192.168.168.0/24
    /srv/fai/config 192.168.168.0/24
```

Generar una configuración FAI-client: para todos los clientes (luego podremos hacer configuraciones diferentes):

```
fai-chboot -IBv -u nfs://192.168.168.254/srv/fai/config default
```

Deberemos modificar el archivo generado para que trabaje con NFSv3; con la versión del kernel utilizado en Wheezy (3.2.x) solo podemos utilizar V3 (si se desea utilizar NFSv4, deberemos cambiar a un kernel 3.12.x que están disponibles en repositorios como BackPorts http://backports.debian.org/). El archivo */srv/tftp/fai/pxelinux.cfg/default* debe quedar como (se ha modificado la variable root):

```
default fai-generated

label fai-generated
kernel vmlinuz-3.2.0-4-amd64
append initrd=initrd.img-3.2.0-4-amd64 ip=dhcp  root=192.168.168.254:/srv/fai/nfsroot:vers=3
aufs FAI_FLAGS=verbose,sshd,reboot FAI_CONFIG_SRC=nfs://192.168.168.254/srv/fai/config
FAI_ACTION=install
```

Copiar los mínimos ficheros de configuración: `cp -a /usr/share/doc/fai-doc/examples/simple/* /srv/fai/config/`

Configuración del nodo: nuestra prueba de concepto la haremos para un nodo en Virtualbox y primero es necesario instalar las expansiones de la versión que tenemos instalada. Por ejemplo, si tenemos VirtualBox 4.3.10, deberemos ir a *https://www.virtualbox.org/wiki/Downloads* e instalar *VirtualBox 4.3.10 Oracle VM VirtualBox Extension Pack All supported platforms* (esto nos permitirá tener un dispositivo que haga *boot* por PXE) y podremos verificar que están instaladas en el menu de VirtualBox – >*File* – >*Preferences* – >*Extensions*. Luego deberemos crear una nueva máquina virtual con un disco vacío y seleccionarla en *System* la opción de *boot order network* en primer lugar. Arrancar la máquina y ver que recibe IP (verificar los posibles errores en */var/log/messages|syslog*) y que comienza bajándose el **initrd-img** y luego el **kernel**.

Los detalles de la configuración se pueden consultar en http://fai-project.org/faiguide/_anchor_id_config_xreflabel_config_installation_details.html

1.9. Logs

Linux mantiene un conjunto de registros llamados *system logs* o *logs* simplemente, que permiten analizar qué ha pasado y cuándo en el sistema, a través de los eventos que recoge del propio *kernel* o de las diferentes aplicaciones y casi todas las distribuciones se encuentran en */var/log*. Probablemente los dos archivos más importantes (y que en mayor o menor presencia están en todas las distribuciones) son */var/log/messages* y */var/log/syslog* los que tienen registros (Ascii) de eventos tales como errores del sistema, (re)inicios/apagados, errores de aplicaciones, advertencias, etc. Existe un comando, `dmesg`, que permite además ver los mensajes de inicio (en algunas distribuciones son los que aparecen en la consola o con la tecla Esc en el modo gráfico de arranque, o Crtl+F8 en otras distribuciones) para visualizar los pasos seguidos durante

el arranque (teniendo en cuenta que puede ser muy extenso, se recomienda ejecutar `dmesg | more`).

Por lo cual, el sistema nos permitirá analizar qué ha pasado y obtener las causas que han producido este registro y dónde/cuándo. El sistema incluido en Debian es `rsyslog` (http://www.rsyslog.com/) con un servicio (a través del *daemon* **rsyslogd** y que reemplaza al original *syslog*. Es un sistema muy potente y versátil y su configuración es través del archivo */etc/rsyslog.conf* o archivos en el directorio */etc/rsyslog.d*. Mirar la documentación sobre su configuración (`man rsyslog.conf` o en la página indicada) pero en resumen incluye una serie de directivas, filtros, templates y reglas que permiten cambiar el comportamiento de los logs del sistema. Por ejemplo, las reglas son de tipo **recurso.nivel acción** pero se puede combinar más de un recurso separado por ',', o diferentes niveles, o todos '*' o negarlo '!', y si ponemos '=' delante del nivel, indica solo ese nivel y no todos los superiores, que es lo que ocurre cuando solo se indica un nivel (los niveles pueden ser de menor a mayor importancia *debug, info, notice, warning, err, crit, alert, emerg*). Por ejemplo `*.warning /var/log/messages` indicará que todos los mensajes de *warning,err,crit,alert,emerg* de cualquier recurso vayan a parar a este archivo. Se puede incluir también un '-' delante del nombre del fichero, que indica que no se sincronice el fichero después de cada escritura para mejorar las prestaciones del sistema y reducir la carga.

Un aspecto interesante de los logs es que los podemos centralizar desde las diferentes máquinas de nuestra infraestructura en un determinado servidor para no tener que conectarnos si queremos 'ver' qué está ocurriendo a nivel de logs en esas máquinas (sobre todo si el número de máquinas es elevado). Para ello, debemos en cada cliente modificar el archivo */etc/rsyslog.conf* (donde la IP será la del servidor que recogerá los logs):

```
# Provides TCP forwarding.
*.*                             @@192.168.168.254:514
```

En el servidor deberemos quitar el comentario (#) de los módulos de recepción por TCP en */etc/rsyslog.conf*:

```
# Provides TCP syslog reception
$ModLoad imtcp
$InputTCPServerRun 514
```

Después de hacer las modificaciones de cada archivo no hay que olvidar hacer un `/etc/init.d/rsyslog restart`. A partir de ese momento, podemos mirar el */var/log/syslog* del servidor, por ejemplo, y veremos los registros marcados con el nombre (o IP) de la máquina donde se ha generado el log.

Existe un paquete llamado `syslog-ng` que presenta características avanzadas (por ejemplo, cifrado o filtros basados en contenidos) u otras equiva-

lentes con `rsyslog`. Más información en http://www.balabit.com/network-security/syslog-ng.

1.9.1. Octopussy

Octopussy es una herramienta gráfica que permite analizar los *logs* de diferentes máquinas y se integra con *syslog* o equivalentes reemplazándolos. Entre sus principales características soporta LDAP, alertas por mail, mensajes IM (Jabber), se integra con Nagios y Zabbix, permite generar reportes y enviarlos por mail, FTP y scp, hacer mapas de la arquitectura para mostrar estados e incorpora una gran cantidad de servicios predefinidos para registrar los *logs*.[32]

Su instalación (que puede resultar un poco complicada) comienza por añadir el repositorio *non-free /etc/apt/sources.list* para descargar paquetes adicionales (libmail-sender-perl) que necesitará este software:

```
deb http://ftp.es.debian.org/debian/ wheezy non-free
deb-src http://ftp.es.debian.org/debian/ wheezy non-free
```

Luego deberemos ejecutar `apt-get update`.

Después deberemos descargar el paquete debian desde la dirección web siguiente http://sourceforge.net/projects/syslog-analyzer/files/ (por ejemplo ctopussy_x.y.x_all.deb donde la x.y.z es la version -10.0.14 en julio de 2014-) e instalarlo.

```
dpkg -i octopussy_1.0.x_all.deb
apt-get -f install
```

Luego deberemos modificar el archivo *etc/rsyslog.conf*:

```
$ModLoad imuxsock # provides support for local system logging
$ModLoad imklog   # provides kernel logging support (previously done by rklogd)
#$ModLoad immark  # provides --MARK-- message capability
# provides UDP syslog reception
$ModLoad imudp
$UDPServerRun 514
# provides TCP syslog reception
$ModLoad imtcp
$InputTCPServerRun 514
```

Y reinicia el rsyslog `/etc/init.d/rsyslog restart`.

Luego se deberán generar los certificados para *Octopussy Web Server* (también se puede utilizar la OpenCA para generar los certificados web propios tal y como se explicó en el capítulo de servidores):

```
openssl genrsa > /etc/octopussy/server.key
openssl req -new -x509 -nodes -sha1 -days 365 -key /etc/octopussy/server.key >
   /etc/octopussy/server.crt
```

En el último no introducir *passwd* y recordar en *Common Name* incluir el nombre.dominio de nuestra máquina.

Configurar el archivo de la configuración de Apache */etc/octopussy/apache2.conf* por el valor correcto:

```
SSLCertificateFile    /etc/octopussy/server.crt
SSLCertificateKeyFile /etc/octopussy/server.key
```

Reiniciar el servidor (propio) de web que luego se interconectará con nuestro Apache por SSL y puerto 8888.

```
/etc/init.d/octopussy web-stop
/etc/init.d/octopussy web-start
```

Y comprobar que funciona en la URL https://sysdw.nteum.org:8888/index.asp, aceptar el certificado e introducir como Usuario/Passwd admin/admin. A partir de la interfaz general se deberá configurar los *Devices, Service, Alerts*, etc. para definir el comportamiento que queremos que tenga la aplicación.

Si se desea deshabilitar y volver a `rsyslog` original (y dado que tiene servicios propios que se ponen en marcha durante el arranque) deberemos ejecutar la instrucción `update-rc.d octopussy remove` y renombrar los archivos de configuración en */etc/rsyslog.d* de la aplicación (por ejemplo,

```
mv /etc/rsyslog.d/xyz-ocotopussy.conf /etc/rsyslog.d/xyz-octopussy.conf.org
```

reemplazando xyz por lo que corresponda) y reiniciando *rsyslog* con

```
/etc/init.d/rsylog restart
```

1.9.2. Herramientas de monitorización adicionales

Además de las herramienta que se vieron en el capítulo de monitorización (como Ganglia, Nagios|Icinga, MRTG o Zabbix) existen otras herramientas que pueden ayudar en la gestión y administración de un clúster como son Cacti, XyMon, Collectd (también disponibles en Debian). En este apartado dedicaremos una pequeña referencia a Cacti y XYMon, ya que por su visión o capacidad de integración permiten tener información al instante sobre qué y cómo están ocurriendo las ejecuciones en el clúster.

Cacti Cacti [16] es una solución para la visualización de estadísticas de red y fue diseñada para aprovechar el poder de almacenamiento y la funcionalidad de generar gráficas que posee RRDtool (similar a Ganglia). Esta herramienta, desarrollada en PHP, provee diferentes formas de visualización, gráficos avanzados y dispone de una una interfaz de usuario fácil de usar, que la hacen interesante tanto para redes LAN como para redes complejas con cientos de dispositivos.

Su instalación es simple y requiere previamente tener instalado MySQL. Luego, haciendo `apt-get install cacti` se instalará el paquete y al final nos pedirá si queremos instalar la interfaz con la base de datos, solicitándonos el nombre de usuario y contraseña de la misma y la contraseña del usuario admin de la interfaz de Cacti (si no le hemos dado, contendrá la que define por defecto, que es usuario admin y *passwd* admin). Después instalará la configuración en el sitio de Apache2 (*/etc/apache2/conf.d/cacti.conf*) y re-arrancará los servidores y podremos conectarnos a la URL *http://sysdw.nteum.org/cacti/*. Las primeras tareas a realizar es hacer los ajustes pertinentes de la instalación (nos mostrará un *checklist*), cambiar la contraseña y nos mostrará a continuación una pantalla con las pestañas de gráficos y la de Consola, en la cual encontraremos las diferentes opciones para configurar y poner en marcha la monitorización más adecuada para nuestro sitio. Siguiendo el procedimiento indicado en *http://docs.cacti.net/manual:088:2_basics.1_first_graph*, será muy fácil incorporar los gráficos más adecuados para monitorizar nuestra instalación. Es interesante incorporar a Cacti un plugin llamado *weathermap* (http://www.network-weathermap.com/) que nos permitirá ver un diagrama de la infraestructura en tiempo real y los elementos monitorizados con datos dinámicos sobre su estado*.

***http://www.network-weathermap.com/manual/0.97b/pages/cacti-plugin.html**

Xymon Xymon [17], es un monitor de red/servicios (bajo licencia GPL) que se ejecuta sobre máquinas GNU/Linux. La aplicación fue inspirada en la version *open-source* de la herramienta Big Brother y se llamó Hobbit pero, como esta era una marca registrada, finalmente cambió a Xymon. Xymon ofrece una monitorización gráfica de las máquinas y servicios con una visualización adecuada y jerárquica óptima para cuando se necesita tener en una sola visualización el estado de la infraestructura (y sobre todo cuando se deben monitorizar centenas de nodos). También se puede entrar en detalles y mirar cuestiones específicas de los nodos, gráficos y estadísticas, pero entrando en niveles interiores de detalles. La monitorización de los nodos requiere de un cliente que será el encargado de enviar la información al *host* (equivalente a NRPE en Nagios). Su instalación en Debian es simple `apt-get install xymon`. Después podremos modificar el archivo */etc/hobbit/bb-host* para agregar una línea como group Server `158.109.65.67 sysdw.nteum.org ssh http://sysdw.nteum.org` y modificar */etc/apache2/conf.d/hobbit* para cambiar la línea `Allow from localhost` por `Allow from all`, reiniciar los dos servicios: `service hobbit restart` y `service apache2 restart` y lo tendremos disponible en la URL: *http://sysdw.nteum.org/hobbit/* (no descuidar la barra final). Para agregar nuevos *hosts* es muy fácil (se declaran en el archivo anterior), e instalar el cliente en los nodos tampoco tiene ninguna dificultad.

2. *Cloud*

Las infraestructuras *cloud* se pueden clasificar en 3+1 grandes grupos en función de los servicios que prestan y a quién:

1) Públicas: los servicios se encuentran en servidores externos y las aplicaciones de los clientes se mezclan en los servidores y con otras infraestructuras. La ventaja más clara es la capacidad de procesamiento y almacenamiento sin instalar máquinas localmente (no hay inversión inicial ni mantenimiento) y se paga por uso. Tiene un retorno de la inversión rápido y puede resultar difícil integrar estos servicios con otros propios.

2) Privadas: las plataformas se encuentran dentro de las instalaciones de la empresa/institución y son una buena opción para quienes necesitan alta protección de datos (estos continúan dentro de la propia empresa), es más fácil integrar estos servicios con otros propios, pero existe inversión inicial en infraestructura física, sistemas de virtualización, ancho de banda, seguridad y gasto de mantenimiento, lo cual supone un retorno más lento de la inversión. No obstante de tener infraestructura a tener infraestructura *cloud* y virtualizada, esta última es mejor por las ventajas de mayor eficiencia, mejor gestión y control, y mayor aislamiento entre proyectos y rapidez en la provisión.

3) Híbridas: combinan los modelos de nubes públicas y privadas y permite mantener el control de las aplicaciones principales al tiempo de aprovechar el *Cloud Computing* en los lugares donde tenga sentido con una inversión inicial moderada y a la vez contar los servicios que se necesiten bajo demanda.

4) Comunidad: Canalizan necesidades agrupadas y las sinergias de un conjunto o sector de empresas para ofrecer servicios de *cloud* a este grupos de usuarios.

Existen además diferentes capas bajo las cuales se pueden contratar estos servicios:

1) *Infraestructure as a Service* (IaaS): se contrata capacidad de proceso y almacenamiento que permiten desplegar aplicaciones propias que por motivos de inversión, infraestructura, coste o falta de conocimientos no se quiere instalar en la propia empresa (ejemplos de este tipo de EC2/S3 de Amazon y Azure de Microsoft).

2) *Platform as a Service* (PaaS): se proporciona además un servidor de aplicaciones (donde se ejecutarán nuestras aplicaciones) y una base de datos, donde

se podrán instalar las aplicaciones y ejecutarlas -las cuales se deberán desarrollar de acuerdo a unas indicaciones del proveedor- (por ejemplo Google App Engine).

3) *Software as a Service* (SaaS): comúnmente se identifica con *'cloud'*, donde el usuario final paga un alquiler por el uso de software sin adquirirlo en propiedad, instalarlo, configurarlo y mantenerlo (ejemplos Adobe Creative Cloud, Google Docs, o Office365).

4) *Business Process as a Service* (BPaaS): es la capa más nueva (y se sustenta encima de las otras 3), donde el modelo vertical (u horizontal) de un proceso de negocio puede ser ofrecido sobre una infraestructura *cloud*.

Si bien las ventajas son evidentes y muchos actores de esta tecnología la basan principalmente en el abaratamiento de los costes de servicio, comienzan a existir opiniones en contra (http://deepvalue.net/ec2-is-380-more-expensive-than-internal-cluster/) que consideran que un *cloud* público puede no ser la opción más adecuada para determinado tipo de servicios/infraestructura. Entre otros aspectos, los negativos pueden ser la fiabilidad en la prestación de servicio, seguridad y privacidad de los datos/información en los servidores externos, *lock-in* de datos en la infraestructura sin posibilidad de extraerlos, estabilidad del proveedor (como empresa/negocio) y posición de fuerza en el mercado no sujeto a la variabilidad del mercado, cuellos de botellas en la transferencias (empresa/proveedor), rendimiento no predecible, tiempo de respuesta a incidentes/accidentes, problemas derivados de la falta de madurez de la tecnología/infraestructura/gestión, acuerdos de servicios (SLA) pensados más para el proveedor que para el usuario, etc. No obstante es una tecnología que tiene sus ventajas y que con la adecuada planificación y toma de decisiones, valorando todos los factores que influyen en el negocio y escapando a conceptos superficiales (todos lo tienen, todos lo utilizan, bajo costo, expansión ilimitada, etc.), puede ser una elección adecuada para los objetivos de la empresa, su negocio y la prestación de servicios en IT que necesita o que forma parte de sus fines empresariales.[18]

Es muy amplia la lista de proveedores de servicios *cloud* en las diferentes capas, pero de acuerdo a la información de Synergy Research Group en 2014 (https://www.srgresearch.com/articles/amazon-salesforce-and-ibm-lead-cloud-infrastructure-service-segments), los líderes en el mercado de infraestructura *cloud* son:

1) *IasS*: Amazon con casi el 50 % e IBM y Rackspace con valores inferiores al 10 % cada uno y Google en valores inferiores.

2) *PaaS*: Salesforce (20 %) seguidas muy de cerca por Amazon, Google y Microsoft.

3) *Private/Hibrid*: IBM (15 %), Orange y Fujitsu con valores cercanos al 5 % cada uno.

4) *Business Process as a Service* (BPaaS): Existen varios operadores por ejemplo IBM, Citrix, VMware entre otros pero no hay datos definidos de cuota de mercado.

Esto significa un cambio apreciable con relación a los datos de 2012, donde se puede observar una variabilidad de la oferta muy alta*.

***https://www.srgresearch.com/articles/amazons-cloud-iaas-and-paas-investments-pay**

En cuanto a plataformas para desplegar *clouds* con licencias GPL, Apache, BSD (o similares), podemos contar entre las más referenciadas (y por orden alfabético):

1) AppScale: es una plataforma que permite a los usuarios desarrollar y ejecutar/almacenar sus propias aplicaciones basadas en Google AppEngine y puede funcionar como servicio o en local. Se puede ejecutar sobre AWS EC2, Rackspace, Google Compute Engine, Eucalyptus, Openstack, CloudStack, así como sobre KVM y VirtualBox y soporta Python, Java, Go, y plataformas PHP Google AppEngine. http://www.appscale.com/

2) Cloud Foundry: es plataforma *open source* PaaS desarrollada por VMware escrita básicamente en Ruby and Go. Puede funcionar como servicio y también en local. http://cloudfoundry.org/

3) Apache CloudStack: diseñada para gestionar grandes redes de máquinas virtuales como IaaS. Esta plataforma incluye todas las características necesarias para el despliegue de un IaaS: CO (*compute orchestration*), *Network-as-a-Service*, gestión de usuarios/cuentas, API nativa, *resource accounting*, y UI mejorada (*User Interface*). Soporta los *hypervisores* más comunes, gestión del *cloud* vía web o CLI y una API compatible con AWS EC2/S3 que permiten desarrollar *clouds* híbridos. http://cloudstack.apache.org/

4) Eucalyptus: plataforma que permite construir *clouds* privados compatibles con AWS. Este software permite aprovechar los recursos de cómputo, red y almacenamiento para ofrecer autoservicio y despliegue de recursos de *cloud* privado. Se caracteriza por la simplicidad en su instalación y estabilidad en el entorno con una alta eficiencia en la utilización de los recursos. https://www.eucalyptus.com/

5) Nimbus: es una plataforma que una vez instalada sobre un clúster, provee IasS para la construcción de *clouds* privados o de comunidad y puede ser configurada para soportar diferentes virtualizaciones, sistemas de colas, o Amazon EC2. http://www.nimbusproject.org/

6) OpenNebula: es una plataforma para gestionar todos los recursos de un centro de datos permitiendo la construcción de IasS privados, públicos e híbridos. Provee una gran cantidad de servicios, prestaciones y adaptaciones que han permitido que sea una de las plataformas más difundidas en la actualidad. http://opennebula.org/

7) OpenQRM: es una plataforma para el despliegue de *clouds* sobre un centro de datos heterogéneos. Permite la construcción de *clouds* privados, públicos e híbridos con IaaS. Combina la gestión del la CPU/almacenamiento/red para ofrecer servicios sobre máquinas virtualizadas y permite la integración con recursos remotos o otros *clouds*.
http://www.openqrm-enterprise.com/community.html

8) OpenShift: apuesta importante de la compañía RH y es una plataforma (version Origin) que permite prestar servicio *cloud* en modalidad PasS. OpenShift soporta la ejecución de binarios que son aplicaciones web tal y como se ejecutan en RHEL por lo cual permite un amplio número de lenguajes y *frameworks*. https://www.openshift.com/products/origin

9) OpenStack es una arquitectura software que permite el despliegue de *cloud* en la modalidad de IaaS. Se gestiona a través de una consola de control vía web que permite la provisión y control de todos los subsistemas y el aprovisionamiento de los recursos. El proyecto iniciado (2010) por Rackspace y NASA es actualmente gestionado por OpenStack Foundation y hay más de 200 compañías adheridas al proyecto, entre las cuales se encuentran las grandes proveedoras de servicios *clouds* públicos y desarrolladoras de SW/HW (ATT, AMD, Canonical, Cisco, Dell, EMC, Ericsson, HP, IBM, Intel, NEC, Oracle, RH, SUSE Linux, VMware, Yahoo entre otras). Es otra de las plataformas más referenciadas. http://www.openstack.org/

10) PetiteCloud: es una plataforma software que permite el despliegue de *clouds* privados (pequeños) y no orientado a datos. Esta plataforma puede ser utilizada sola o en unión a otras plataformas *clouds* y se caracteriza por su estabilidad/fiabilidad y facilidad de instalación. http://www.petitecloud.org

11) oVirt: si bien no puede considerarse una plataforma *cloud*, oVirt es una aplicación de gestión de entornos virtualizados. Esto significa que se puede utilizar para gestionar los nodos HW, el almacenamiento o la red y desplegar y monitorizar las máquinas virtuales que se están ejecutando en el centro de datos. Forma parte de RH Enterprise Virtualization y es desarrollado por esta compañía con licencia Apache. http://www.ovirt.org/

2.1. Opennebula

Considerando las opiniones en [19],[20], nuestra prueba de concepto la realizaremos sobre OpenNebula. Dado que la versión en Debian Wheezy es la 3.4 y la web de los desarrolladores es la 4.6.2 (http://opennebula.org/) hemos decidido instalar los paquetes Debian con KVM como *hypervisor* de la nueva versión*. Esta instalación es una prueba de concepto (funcional pero mínima), pero útil para después hacer un despliegue sobre una arquitectura real, donde por un lado ejecutaremos los servicios de OpenNebula y su interfaz gráfica (llamada Sunstone) y por otro, un hypervisor (*host*) que ejecutará las máquinas virtuales. OpenNebula asume dos roles separados: *Frontend* y Nodos. El ***Frontend*** es quien ejecuta los servicios/gestión web y los **Nodos** ejecutan la

***http://docs.opennebula.org/4.6/design_and_installation/quick_starts/qs_ubuntu_kvm.html**

máquina virtuales, y si bien en nuestra instalación de pruebas ejecutaremos el Frontend y Nodos en la misma máquina, se recomienda ejecutar las máquinas virtuales en otros *hosts* que tengan las extensiones de virtualización (en nuestro caso KVM). Para verificar si disponemos estas extensiones, HW podemos ejecutar `grep -E 'svm|vmx'/proc/cpuinfo` y si nos da salida es casi seguro que el sistema soporta estas extensiones. Tened en cuenta que esto lo deberemos hacer sobre un sistema operativo base (*Bare-Metal*) y no sobre uno ya virtualizado, es decir, no podemos instalar OpenNebula sobre una máquina virtualizada por VirtualBox por ejemplo (hay hypervisores que permiten esta instalación como VMWare ESXi pero Virtualbox no). Los paquetes que conforma la instalación son:

1) opennebula-common: archivos comunes.

2) libopennebula-ruby: librerías de ruby

3) opennebula-node: paquete que prepara un nodo donde estarán la VM.

4) opennebula-sunstone: OpenNebula Sunstone *Web Interface*

5) opennebula-tools: *Command Line interface*

6) opennebula-gate: permite la comunicación entre VMs y OpenNebula

7) opennebula-flow: gestiona los servicios y la "elasticidad"

8) opennebula: OpenNebula *Daemon*

Instalación del Frontend (como root): Instalar el repositorio: `wget -q -O-http://downloads.opennebula.org/repo/Debian/repo.key | apt-key add -` y luego los agregamos a la lista

```
echo "deb http://downloads.opennebula.org/repo/Debian/7 stable
opennebula"> /etc/apt/sources.list.d/opennebula.list
```

Instalar los paquetes: primero `apt-get update` y luego `apt-get install opennebula opennebula-sunstone` (si el nodo está en un *host* separado deberán compartir un directorio por NFS, por lo cual es necesario también instalar el `nfs-kernel-server`).

Servicios: deberemos tener dos servicios en marcha que son OpenNebula *daemon* (oned) y la interfaz gráfica (sunstone) la cual solo está configurada por seguridad para atender el *localhost* (si se desea cambiar, editar */etc/one/sunstone-server.conf* y cambiar **:host: 127.0.0.1** por **:host: 0.0.0.0** y reiniciar el servidor `/etc/init.d/opennebula-sunstone restart`.

Configurar el NFS (si estamos en un único servidor con *Frontend*+Nodo esta parte no es necesario): Agregar a */etc/exports* del *Frontend* `/var/lib/one/*(rw,sync,no_subtree_check,root_squash)` y después reiniciar el servicio (`service nfs-kernel-server restart`).

Configurar la llave pública de SSH: OpenNebula necesita acceder por SSH a los nodos como el usuario **oneadmin** y sin *passwd* (desde cada nodo a otro, incluido el *frontend*) por lo cual, nos cambiamos como usuario oneadmin (`su - oneadmin`) y ejecutamos

```
cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
```

y agregamos el siguiente texto a *~/.ssh/config* (para que no pida confirmación de known_hosts):

```
cat << EOT > ~/.ssh/config
Host *
    StrictHostKeyChecking no
    UserKnownHostsFile /dev/null
EOT
chmod 600 ~/.ssh/config
```

Instalación de los nodos: repetimos el paso de configurar el repositorio y ejecutamos la instrucción `apt-get install opennebula-node nfs-common bridge-utils` (en nuestro caso no es necesario, ya que es la misma máquina y solo debemos instalar `opennebula-node` y `bridge-utils`). Verificar que podemos acceder como oneadmin a cada nodo y copiamos las llaves de `ssh`.

Configuración de la red: (hacer un *backup* de los archivos que modificaremos previamente) generalmente tendremos eth0 y la conectaremos a un *bridge* (el nombre del *bridge* deberá ser el mismo en todos los nodos) y en */etc/network/interfaces* agregamos (poner las IP que correspondan):

```
auto lo
iface lo inet loopback
auto br0
iface br0 inet static
        address 192.168.0.10
        network 192.168.0.0
        netmask 255.255.255.0
        broadcast 192.168.0.255
        gateway 192.168.0.1
        bridge_ports eth0
        bridge_fd 9
        bridge_hello 2
        bridge_maxage 12
        bridge_stp off
```

Si tenemos servicio de DHCP reemplazar por:

```
auto lo
iface lo inet loopback
auto br0
iface br0 inet dhcp
        bridge_ports eth0
        bridge_fd 9
        bridge_hello 2
        bridge_maxage 12
        bridge_stp off
```

Y reiniciamos la red `/etc/init.d/networking restart`

Configuramos el NFS sobre los nodos (no necesario en nuestro caso): agregamos en */etc/fstab*

```
192.168.1.1:/var/lib/one/ /var/lib/one/ nfs soft,intr,rsize=8192,wsize=8192,noauto
```

donde 192.168.1.1 es la IP del *frontend*; luego montamos el directorio `mount /var/lib/one/`.

Configuramos Qemu: el usuario **oneadmin** debe poder manejar libvirt como root por lo cual ejecutamos:

```
cat << EOT > /etc/libvirt/qemu.conf
user  = "oneadmin"
group = "oneadmin"
dynamic_ownership = 0
EOT
```

Reiniciamos libvirt con `service libvirt-bin restart`

Uso básico: En el Frontend podemos conectarnos a la interfaz web en la URL *http://frontend:9869*. El usuario es **oneadmin** y el *passwd* está su HOME en */.one/one_auth* que se genera en forma aleatoria (*/var/lib/one/.one/one_auth*). Para interactuar con OpenNebula se debe hacer desde el usuario **oneadmin** y desde el frontend (para conectarse hacer simplemente `su - oneadmin`).

Agregar un host: es lo primero que hay que hacer para luego ejecutar VM, se puede hacer desde la interfaz gráfica o desde CLI haciendo como oneadmin `onehost create localhost -i kvm -v kvm -n dummy` (poner el nombre del *host* correcto en lugar de *localhost*).

Si hay errores probablemente son de `ssh` (verificar que igual que en el caso de oneadmin se puede conectar a los otros *hosts* sin *passwd*) y los errores los veremos en */var/log/one/oned.log*.

El segundo paso es agregar la red, una imagen y un template antes de lanzar una VM:

Red: (se puede hacer desde la interfaz gráfica también), creo el archivo *mynetwork.one* con el siguiente contenido:

```
NAME = "private"
TYPE = FIXED
BRIDGE = br0
LEASES = [ IP=192.168.0.100 ]
LEASES = [ IP=192.168.0.101 ]
LEASES = [ IP=192.168.0.102 ]
```

Donde las IP deberán estar libres en la red que estamos haciendo el despliegue y ejecutamos `onevnet create mynetwork.one`

En el caso de la imagen, esta se puede hacer desde CLI pero es más simple hacerla desde la interfaz web. Seleccionamos **Marketplace** y luego **ttylinux-kvm** (es una imagen pequeña de prueba) y luego le decimos **Import**. Podremos verificar que la imagen está en el apartado correspondiente, pero el estado es LOCK hasta que se la haya bajado y cuando finalice, veremos que cambia a READY. Luego podremos crear un template seleccionando esta imagen, la red, y allí podremos poner la llave pública del *∼/.ssh/id_dsa.pub* en el apartado **Context**, y desde este apartado (**Templates**) podremos decirle que cree una instancia de este template (o en su defecto en **VirtualMachines** crear una utilizando este template). Veremos que la VM pasa de PENDING – > PROLOG – > RUNNING (si falla podremos ver en los *logs* la causa) y luego nos podremos conectar o bien por `ssh` a la IP de la VM o a través de **VNC** en la interfaz web.

Como se ha podido comprobar, la instalación está bastante automatizada pero se debe tener en cuenta que es una infraestructura compleja y que se debe dedicar tiempo y análisis a determinar las causas por las cuales se producen los errores y solucionarlos. Existe una gran cantidad de documentación y sitios en Internet pero recomiendo comenzar por las fuentes [21] y [22].

En este apartado se ha visto una prueba de concepto funcional pero OpenNebula es muy extenso y flexible, y permite gran cantidad de opciones/extensiones. A través de OpenNebula C12G Labs (http://c12g.com/) se podrán encontrar y descargar imágenes de máquinas virtuales creadas para OpenNebula* y listas para ponerlas en funcionamiento (son las que se ven desde el MarketPlace de la interfaz web), pero al descargarlas sobre nuestro servidor la podremos utilizar siempre cuando la necesitemos y no se deberá descargar cada vez (no se debe descomprimir ni hacer nada, utilizar tal y como está). Una de las extensiones interesantes es OpenNebula Zones (llamada **ozones**) que nos permite una administración centralizada de múltiples instancias de OpenNebula (zones), gestionando diferentes dominios administrativos. El módulo es gestionado por el *oZones administrator*, que es que el administra los permisos a las diferentes zonas de los usuarios individuales**. Su configuración puede encontrarse en http://docs.opennebula.org/4.4/advanced_administration/multiple_zone_and_virtual_data_centers/zonesmngt.html.

***http://marketplace.c12g.com/appliance**

****http://archives.opennebula.org/documentation:archives:rel3.0:ozones**

3. DevOps

Tal y como hemos comentado al inicio de este capítulo, Devops se puede considerar, en palabras de los expertos, un movimiento tanto en lo profesional como en lo cultural del mundo IT. Si bien no existen todas las respuestas todavía, un administrador se enfrentará a diferentes 'comunidades' (y él mismo formará parte de una de ellas), que tendrán nuevas necesidades de desarrollo y producción de servicios/productos dentro del mundo IT y con las premisas de 'más rápido', 'más eficiente', 'de mayor calidad' y totalmente adaptable a los 'diferentes entornos'. Es decir, el grupo de trabajo deberá romper las barreras existentes entre los departamentos de una empresa permitiendo que un producto pase rápidamente desde el departamento de investigación al de diseño, luego al de desarrollo + producción, luego al de test + calidad y por último, a ventas, y con las necesidades de herramientas que permitan desplegar todas estas actividades en cada una de las fases y llevar el control de todos por los responsables de cada uno de los ámbitos, incluidos los funcionales y los de la organización/directivos. Es por ello por lo que un administrador necesitará herramientas de gestión, control, despliegue, automatización y configuración. Una lista (corta) de las herramientas que podríamos considerar de acuerdo a nuestros objetivos de licencia GPL-BSD-Apache o similares (es interesante el sitio http://devs.info/, ya que permite acceder a la mayor parte de las herramientas/entornos/documentación para programadores y una lista más detallada -que incluye SW propietario- se puede encontrar en [27]):

1) **Linux**: Ubuntu|Debian^{v}, Fedora^{v}|CentOS|SL

2) **IasS**: Cloud Foundry, OpenNebula^{v}, OpenStack

3) **Virtualización**: KVM^{v}, Xen, VirtualBox^{v}, Vagrant^{v}

4) Contenedores: LXC^{v}, Docker^{v}

5) Instalación SO: Kickstart y Cobbler (rh), Preseed|Fai^{v} (deb), Rocks^{v}

6) Gestión de la configuración: Puppet^{v}, Chef^{v}, CFEngine, SaltStack, Juju, bcfg2, mcollective, fpm (effing)

7) Servidores Web y aceleradores: Apache^{v}, nginx^{v}, varnish, squid^{v}

8) BD: MySQL^{v}|MariaDB^{v}, PostgreSQL^{v}, OpenLDAP^{v}, MongoDB, Redis

9) Entornos: Lamp, Lamr, AppServ, Xampp, Mamp

10) Gestión de versiones: Git^{v}, Subversion^{v}, Mercurial^{v}

11) Monitorización/supervisión: Nagios^{v}, Icinga, Ganglia^{v}, Cacti^{v}, Monin^{v}, MRTG^{v}, XYmon^{v}

12) Misc: pdsh, ppsh, pssh, GNUparallel, nfsroot, Multihost SSH Wrapper, lldpd, Benchmarks^{v}, Librerías^{v}

13) Supervisión: Monit^{v}, runit, Supervisor, Godrb, BluePill-rb, Upstart, Systemd^{v}

14) Security: OpenVas^{v}, Tripwire^{v}, Snort^{v}

15) Desarrollo y Test: Jenkins, Maven, Ant, Gradle, CruiseControl, Hudson

16) Despliegue y Workflow: Capistrano

17) Servidores de aplicaciones: JBoss, Tomcat, Jetty, Glassfish,

18) Gestión de Logs: Rsyslog^{v}, Octopussy^{v}, Logstash

Excepto las que son muy orientadas a desarrollo de aplicaciones y servicios, en las dos asignaturas se han visto (o se verán dentro de este capítulo) gran parte de ellas (marcadas con ^{v}) y sin ninguna duda, con los conocimientos obtenidos, el alumno podrá rápidamente desplegar todas aquellas que necesite y que no se han tratado en estos cursos.

A continuación veremos algunas herramientas (muy útiles en entornos DevOps) más orientadas a generar automatizaciones en las instalaciones o generar entornos aislados de desarrollos/test/ejecución que permiten de forma simple y fácil (y sin pérdidas de prestaciones/rendimiento) disponer de herramientas y entornos adecuados a nuestras necesidades.

3.1. Linux Containers, LXC

LXC (LinuX Containers) es un método de virtualización a nivel del sistema operativo para ejecutar múltiples sistemas Linux aislados (llamados contenedores) sobre un único *host*. El *kernel* de Linux utiliza `cgroups` para poder aislar los recursos (CPU, memoria, E/S, network, etc.) lo cual no requiere iniciar ninguna máquina virtual. `cgroups` también provee aislamiento de los espacios de nombres para aislar por completo la aplicación del sistema operativo, incluido árbol de procesos, red, id de usuarios y sistemas de archivos montados. A través de una API muy potente y herramientas simples, permite crear y gestionar contenedores de sistema o aplicaciones. LXC utiliza diferentes módulos del *kernel* (ipc, uts, mount, pid, network, user) y de aplicaciones (Apparmor, SELinux profiles, Seccomp policies, Chroots -pivot_root- y Control groups -cgroups-) para crear y gestionar los contenedores. Se puede considerar que LXC está a medio camino de un 'potente' `chroot` y una máquina virtual, ofreciendo un entorno muy cercano a un Linux estándar pero sin necesidad de tener un kernel separado. Esto es más eficiente que utilizar virtualización con un `hypervisor` (KVM, Virtualbox) y más rápido de (re)iniciar, sobre todo si se están haciendo desarrollos y es necesario hacerlo frecuentemente, y su impacto en el rendimiento es muy bajo (el contenedor no ocupa recursos)

y todos se dedicarán a los procesos que se estén ejecutando. El único inconveniente de la utilización de LXC es que solo se pueden ejecutar sistemas que soportan el mismo *kernel* que su anfitrión, es decir, no podremos ejecutar un contenedor BSD en un sistema Debian.

La instalación se realiza a través de `apt-get install lxc lxctl` y se pueden instalar otros paquetes adicionales que son opcionales (`bridge-utils libvirt-bin debootstrap`), no obstante, si queremos que los contenedores tengan acceso a la red, con Ip propia es conveniente instalar el paquete `bridge-utils`: `apt-get install bridge-utils`. Para preparar el *host* primero debemos montar el directorio `cgroup` añadiendo a */etc/fstab* la siguiente línea `cgroup /sys/fs/cgroup cgroup defaults 0 0` y verificando que podemos hacer `mount -a` y el sistema *cgroups* aparecerá montado cuando ejecutemos el comando `mount`. Ver detalles y posibles soluciones a errores en [23, 24, 26]. A partir de este momento podemos verificar la instalación con `lxc-checkconfig` que dará una salida similar a:

```
Found kernel config file /boot/config-3.2.0-4-amd64
--- Namespaces ---
 Namespaces: enabled
 Utsname namespace: enabled
 Ipc namespace: enabled
 Pid namespace: enabled
 User namespace: enabled
 Network namespace: enabled
 Multiple /dev/pts instances: enabled

--- Control groups ---
 Cgroup: enabled
 Cgroup clone_children flag: enabled
 Cgroup device: enabled
 Cgroup sched: enabled
 Cgroup cpu account: enabled
 Cgroup memory controller: enabled
 Cgroup cpuset: enabled

--- Misc ---
 Veth pair device: enabled
 Macvlan: enabled
 Vlan: enabled
 File capabilities: enabled
```

Si surgen opciones deshabilitadas, se debe mirar la causa y solucionarlo (aunque algunas pueden estarlo). Si bien la distribución incluye un contenedor 'debian' (*/usr/share/lxc/templates*) es recomendable obtener uno desde la dirección https://github.com/simonvanderveldt/lxc-debian-wheezy-template que da solución a algunos problemas que tiene el original cuando se instala sobre Debian Wheezy. Para ello podemos hacer:

```
wget https://github.com/simonvanderveldt/lxc-debian-wheezy-template/raw/master/lxc-debian-wheezy-robvdhoeven
-O /usr/share/lxc/templates/lxc-debianW
host# chown root:root /usr/share/lxc/templates/lxc-debianW
host# chmod +x /usr/share/lxc/templates/lxc-debianW
```

A continuación se crear el primer contenedor como `lxc-create -n mycont -t debianW` (en este caso lo llamo **mycont** y utilizo el template de **debianW**

pero se pueden también debian que es el que incluye la distribución) Si deseamos configurar la red debemos primero configurar el *bridge* modificando (en el host) */etc/network/interfaces* con lo siguiente:

```
# The loopback network interface
auto lo br0
iface lo inet loopback

# The primary network interface
iface eth0 inet manual          # la ponemos en manual para evitar problemas
iface br0 inet static           # inicializamos el bridge
address 192.168.1.60
netmask 255.255.255.0
gateway 192.168.1.1
bridge_ports eth0
bridge_stp off       # disable Spanning Tree Protocol
    bridge_waitport 0    # no delay before a port becomes available
    bridge_fd 0          # no forwarding delay
```

Hacemos un `ifdown eth0` y luego un `ifup br0` y verificamos con `ifconfig` y por ejemplo `ping google.com` que tenemos conectividad. Luego modificamos el archivo de configuración */var/lib/lxc/mycont/config* para insertar el bridge:

```
lxc.utsname = myvm
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0          #debe existir en el host
lxc.network.ipv4 = 192.168.1.100/24     # Ip para el contenedor 0.0.0.0 indica dhcp
lxc.network.hwaddr = 00:1E:1E:1a:00:00
```

Los comandos más útiles para gestionar los contenedores son:

1) Para iniciar la ejecución como *daemon* -en *background*- (el login/*passwd* por defecto es root/root): `lxc-start -n mycont -d`

2) Para conectarnos al contenedor: `lxc-console -n mycont` "Ctrl+a q" para salir de la consola.

3) Para iniciar el contenedor anexado a la consola: `lxc-start -n mycont` (en *foreground*)

4) Para parar la ejecución del contenedor: `lxc-stop -n myvm`

5) Para iniciar los contenedores al *boot* en forma automática se deberá hacer un enlace del archivo de configuración en el directorio */etc/lxc/auto/*, por ejemplo `ln -s /var/lib/lxc/mycont/config /etc/lxc/auto/mycont`

6) Para montar sistemas de archivos externos dentro del contenedor agregar a */var/lib/lxc/mycont/config* la línea

```
lxc.mount.entry=/path/in/host/mount_point /var/lib/lxc/mycont/rootfs/mount_moint
none bind 0 0
```

y reiniciar el contenedor.

Se debe tener cuidado cuando se inicia el contenedor sin '-d', ya que no hay forma de salir (en la versión actual el 'Ctrl+a q' no funciona). Iniciar siempre los contenedores en *background* (con '-d') a no ser que necesite depurar porque el contenedor no arranca. Otra consideración a tener es que el comando `lxc-halt` ejecutará el `telinit` sobre el archivo */run/initctl* si existe y apagará el *host*, por lo cual hay que apagarlo con `lxc-stop` y tampoco hacer un `shutdown -h now` dentro del contenedor, ya que también apagará el *host*. También existe un panel gráfico para gestionar los contenedores vía web http://lxc-webpanel.github.io/install.html (en Debian hay problemas con el apartado de red; algunos de ellos los soluciona https://github.com/vaytess/LXC-Web-Panel/tree/lwp-backup. Para ello clonar el sitio

```
git clone https://github.com/vaytess/LXC-Web-Panel.git
```

y luego reemplazar el directorio obtenido por */srv/lwp* -renombrar este antes- y hacer un `/etc/init.d/lwp restart`). También es importante notar que la versión disponible en Debian es la 0.8 y en la web del desarrollador* es la 1.04, que tiene muchos de los errores corregidos y su compilación en muy simple (ver archivo INSTALL dentro de paquete) por lo cual, si se va a trabajar con ella, se recomienda esta versión.

***https://linuxcontainers.org /downloads/**

3.2. Docker

Docker es una plataforma abierta para el desarrollo, empaquetado y ejecución de aplicaciones de forma que se puedan poner en producción o compartir más rápido separando estas de la infraestructura, de forma que sea menos costoso en recursos (básicamente espacio de disco, CPU y puesta en marcha) y que esté todo preparado para el siguiente desarrollador con todo lo que Ud. puso pero nada de la parte de infraestructura. Esto significará menor tiempo para para probar y acelerará el despliegue acortando en forma significativa el ciclo entre que se escribe el código y pasa a producción. Docker emplea una plataforma (contenedor) de virtualización ligera con flujos de trabajo y herramientas que le ayudan a administrar e implementar las aplicaciones proporcionando una forma de ejecutar casi cualquier aplicación en forma segura un contenedor aislado. Este aislamiento y seguridad permiten ejecutar muchos contenedores de forma simultánea en el *host* y dada la naturaleza (ligera) de los contenedores todo ello se ejecuta sin la carga adicional de un *hypervisor* (que sería la otra forma de gestionar estas necesidades compartiendo la VM) lo cual significa que podemos obtener mejores prestaciones y utilización de los recursos. Es decir, con un VM cada aplicación virtualizada incluye no solo la aplicación, que pueden ser decenas de MBytes -binarios + librerías- sino también el sistema operativo 'guest', que pueden ser varios Gbytes; en cambio, en Docker lo que se denomina DE (Docker Engine) solo comprende la aplicación y sus dependencias, que se ejecuta como un proceso aislado en el espacio de usuario del SO 'host', compartiendo el *kernel* con otros contenedores. Por lo tanto,

tiene el beneficio del aislamiento y la asignación de recursos de las máquinas virtuales, pero es mucho más portátil y eficiente transformándose en el entorno perfecto para dar soporte al ciclo de vida del desarrollo de software, test de plataformas, entornos, etc.[33]

El entorno está formado por dos grandes componentes: **Docker** [34] (es la plataforma de virtualización -contenedor-) y **Docker Hub** [35] (una plataforma SasS que permite obtener y publicar/gestionar contenedores ya configurados). En su arquitectura, Docker utiliza una estructura cliente-servidor donde el cliente (CLI **docker**) interactúa con el *daemon* Docker, que hace el trabajo de la construcción, ejecución y distribución de los contenedores de Docker. Tanto el cliente como el *daemon* se pueden ejecutar en el mismo sistema, o se puede conectar un cliente a un *daemon* de Docker remoto. El cliente Docker y el servicio se comunican a través de sockets o una API RESTful.

Para la instalación no están disponibles los paquetes sobre Debian Wheezy (sí están sobre Jessie) y debemos actualizar el kernel, ya que Docker necesita una versión 3.8 (o superior). Para ello cargaremos el nuevo kernel 3.14 (julio 2014) desde Debian Backports ejecutando:

```
echo "deb http://ftp.debian.org/debian/ wheezy-backports main non-free contrib" >
/etc/apt/sources.list.d/backport.list
apt-get update
apt-get -t wheezy-backports install linux-image-amd64 linux-headers-amd64
reboot
```

Una vez que hemos seleccionado el nuevo *kernel* durante el arranque podemos hacer:

```
Instalamos unas dependencias:
    apt-get install apt-transport-https
Importamos la key de Ubuntu:
    apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
    36A1D7869245C8950F966E92D8576A8BA88D21E9
Agregamos el repositorio de Docker:
    echo "deb http://get.docker.io/ubuntu docker main" > /etc/apt/sources.list.d/docker.list"
Actualizamos e instalamos:
    apt-get update
    apt-get install lxc-docker
Ahora podemos verificar si la instalación funciona ejecutando una imagen de Ubuntu (local)
dentro de su contenedor:
    docker run -i -t ubuntu /bin/bash           (Crtl+D para salir del contenedor)
Desde dentro podremos ejecutar \texttt{more lsb-release} y nos indicará:
        DISTRIB_ID=Ubuntu
        DISTRIB_RELEASE=14.04
        DISTRIB_CODENAME=trusty
        DISTRIB_DESCRIPTION="Ubuntu 14.04 LTS"
Pero también podemos hacer \texttt{docker run -i -t centos /bin/bash} como que la imagen no la tiene en
local se la descargará del Docker Hub y la ejecutará.
```

Haciendo more */etc/os-release* nos indicará que es un CentOS Linux, 7 (Core) e información complementaria.

Antes de cargar el primer contenedor nos puede dar un error como *Cannot start container ... mkdir /sys/fs/devices: operation not permitted*. Esto es debido a una instalación previa de LXC anterior y solo debemos comentar en el archivo */etc/fstab* la línea `#cgroup /sys/fs/cgroup cgroup defaults 0 0`, reiniciar la máquina y ejecutar nuevamente el contenedor. Los comandos para inciarse (además de los que hemos visto) son:

1) docker: muestra todas las opciones.

2) docker images: lista las imágenes localmente.

3) docker search patrón: busca contenedores/imágenes que tengan ese patrón.

4) docker pull nombre: obtiene imágenes del Hub. nombre puede ser <username>/<repository>para la particulares.

5) docker run name cmd: ejecutará el contenedor y dentro el comando indicado por cmd

6) docker ps -l: permite obtener el ID e información de una imagen.

7) docker commit ID name: actualiza la imagen con lo que se ha instalado hasta ese momento (con 3-4 números del ID es suficiente).

8) docker inspect: muestra las imágenes corriendo -similar a docker ps-.

9) docker push: salva la imagen el el Hub (debemos registrarnos primero) y está disponible para otros usarios/host que se la quieran instalar con todo lo que hemos configurado dentro.

10) docker cp: copia archivos/folders desde el contenedor al host.

11) docker export/import: exporta/importa el contenedor en un archivo tar.

12) docker history: muestra la historia de una imagen.

13) docker info: muestra información general (imagenes, directorios, ...).

14) docker kill: para la ejecución de un contenedor.

15) docker restart: reinicia un contenedor.

16) docker rm: elimina un contendor.

17) docker rmi: elimina imagenes.

18) docker start/stop: inicia/para un contenedor.

3.3. Puppet

Puppet es una herramienta de gestión de configuración y automatización en sistemas IT (licencia Apache, -antes GPL-). Su funcionamiento es gestionado a través de archivos (llamados *manifiests*) de descripciones de los recursos del sistema y sus estados, utilizando un lenguaje declarativo propio de la herramienta. El lenguaje Puppet puede ser aplicado directamente al sistema, o compilado en un catálogo y distribuido al sistema de destino utilizando un modelo

cliente-servidor (mediante una API REST), donde un agente interpela a los proveedores específicos del sistema para aplicar el recurso especificado en los *manifiests*. Este lenguaje permite una gran abstracción que habilita a los administradores describir la configuración en términos de alto nivel, tales como usuarios, servicios y paquetes sin necesidad de especificar los comandos específicos del sistema operativo (apt, dpkg, etc.).[36, 37, 38] El entorno Puppet tiene dos versiones Puppet (*Open Source*) y Puppet Enterprise (producto comercial) cuyas diferencias se pueden ver en http://puppetlabs.com/puppet/enterprise-vs-open-source y además se integran con estos entornos otras herramientas como MCollective (*orchestration framework*), Puppet Dashboard (consola web pero abandonada en su desarrollo), PuppetDB *(datawarehouse* para Puppet), Hiera (herramienta de búsqueda de datos de configuración), Facter (herramienta para la creación de catálogos) y Geppetto (IDE -integrated development environment- para Puppet).

3.3.1. Instalación

Es importante comenzar con dos máquinas que tengan sus *hostname* y definición en */etc/hosts* correctamente y que sean accesibles a través de la red con los datos obtenidos del */etc/hosts* (es importante que el nombre de la máquina -sin dominio- de este archivo coincida con el nombre especificado en *hostname* ya que si no habrá problemas cuando se generen los certificados).

```
Obtenemos los repositorios e instalamos puppetmaster en el servidor:
    wget http://apt.puppetlabs.com/puppetlabs-release-wheezy.deb
    dpkg -i puppetlabs-release-wheezy.deb
    apt-get update
    apt-get install puppetmaster
Puppetlabs recomienda ejecutar el siguiente comando antes de correr puppetmaster:
    puppet resource service puppetmaster ensure=running enable=true
    service puppetmaster restart

Sobre el cliente, instalar los repositorios (tres primeras instrucciones) e instalar puppet
(si el cliente es Ubuntu, no es necesario instalar el repositorio):
    apt-get install puppet
Sobre el cliente modificar /etc/puppet/puppet.conf para agregar en la sección [main] el
servidor (sysdw.nteum.org en nuestro caso) y reiniciar:
    [main]
    server=sysdw.nteum.org
    /etc/init.d/puppet restart
Luego ejecutar sobre el cliente:
    puppet agent  --waitforcert 60 --test
Este comando enviará una petición de firma del certificado al servidor y esperará a que este se lo
firme, por lo cual, rápidamente sobre el servidor se debe hacer:
    puppetca --list
Nos responderá con algo como "pucli.nteum.org" (...) y ejecutamos:
    puppetca --sign pucli.nteum.org
Veremos información por parte del servidor y también por parte del cliente y la ejecución de los
catálogos que haya pendientes.
```

De esta forma tenemos instalado el cliente y el servidor y ahora deberemos hacer nuestro primer 'manifiesto'. Para ello organizaremos la estructura de acuerdo a las recomendaciones de PuppetLabs haciendo un árbol que tendrá la siguiente estructura a partir de */etc/puppet*:

```
+-- auth.conf
+-- autosign.conf
+-- environments
|   +-- common
|   +-- development
|   |   +-- modules
|   +-- production
|       +-- modules
|           +-- ntp
+-- etckeeper-commit-post
+-- etckeeper-commit-pre
+-- files
|   +-- shadow.sh
+-- fileserver.conf
+-- manifests
|   +-- nodes
|   |   +-- client1.pp
|   |   +-- server.pp
|   +-- site.pp
+-- modules
|   +-- accounts
|   |   +-- manifests
|   |       +-- init.pp
|   |       +-- system.pp
|   +-- elinks
|   |   +-- manifests
|   |       +-- init.pp
|   +-- nmap
|       +-- manifests
|           +-- init.pp
+-- node.rb
+-- puppet.conf
+-- rack
+-- templates
```

Comenzaremos por el archivo */etc/puppet/manifests/site.pp*, que como contenido tendrá `import 'nodes/*.pp'`. En el directorio */etc/puppet/manifests/nodes* tendremos dos archivos (server.pp y client1.pp) con el siguiente contenido:

```
Archivo server.pp:
    node 'sysdw.nteum.org' {
    }
Archivo client1.pp:
   node 'pucli.nteum.org' {
        include nmap
        include elinks
}
```

Donde definimos dos servicios a instalar en el cliente pucli.nteum.org (nmap y elinks). Después definimos los directorios */etc/puppet/modules/elinks/manifests* y */etc/puppet/modules/nmap/manifests* donde habrá un archivo *init.pp* con la tarea a realizar:

```
Archivo /etc/puppet/modules/elinks/manifests/init.pp:
    class elinks {
        case $operatingsystem {
            centos, redhat: {
                package { " elinks":
                ensure => installed,}}
            debian, ubuntu: {
                package { " elinks":
                ensure => installed,}}
```

```
            }
    }
Archivo /etc/puppet/modules/nmap/manifests/init.pp:
   class nmap {
    case \$operatingsystem {
        centos, redhat: {
            package { "nmap":
            ensure => installed,}}
        debian, ubuntu: {
            package { "nmap":
            ensure => installed,}}
        }
    }
```

En estos podemos ver la selección del SO y luego las indicaciones del paquete a instalar. Todo esto se podría haber puesto en el archivo inicial (site.pp), pero se recomienda hacerlo así para mejorar la visibilidad/estructura y que puedan aprovecharse los datos para diferentes instalaciones. Sobre el servidor deberíamos ejecutar `puppet apply -v /etc/puppet/manifests/site.pp` y sobre el cliente para actualizar el catálogo (e instalar las aplicaciones) `puppet agent -v -test` (en caso de no hacerlo, el cliente se actualizará a los 30 minutos por defecto). Se podrá verificar sobre el cliente que los paquetes se han instalado e incluso se puede desinstalar desde la línea de comandos y ejecutar el agente que los volverá a instalar.[37]

Como siguiente acción procuraremos crear un usuario y ponerle un *password*. Comenzamos creando un módulo */etc/puppet/modules/accounts/manifests* y dentro de él, dos archivos llamados *init.pp* y *system.pp* con el siguiente contenido:

```
Archivo /etc/puppet/modules/accounts/manifests/system.pp:
    define accounts::system ($comment,$password) {
    user { $title:
       ensure => 'present',
       shell => '/bin/bash',
       managehome => true,}
    }
Archivo /etc/puppet/modules/accounts/manifests/init.pp:
    class accounts {
    file { '/etc/puppet/templates/shadow.sh':
        ensure => file,
        recurse => true,
        mode => "0777",
        source => "puppet:///files/shadow.sh",}
    @accounts::system { 'demo':
       comment      => 'demo users',
       password     => '*',}
    exec { "demo":
          command => 'echo "demo:123456" | chpasswd',
          provider => 'shell',
          onlyif => " /etc/puppet/templates/shadow.sh demo",}
}
```

En el archivo system hemos definido el tipo de accounts::system para asegurar que cada usuario tiene directorio HOME y *shell* (en lugar de los valores predeterminados del comando `useradd`), contraseña y el campo Gecos. Luego en el *init.pp* le indicamos el nombre del usuario a crear y el campo Geco. Para crear este usuario sin *passwd* debemos modificar *client1.pp* para que quede:

```
Archivo client1.pp:
   node 'pucli.nteum.org' {
     #include nmap
     #include elinks
include accounts
    realize (Accounts::System['demo'])
}
```

Como se puede observar, hemos comentado lo que no quiero que se ejecute, ya que el nmap y el elinks se instalaron en la ejecución anterior. Aquí indicamos que se incluya el módulo *accounts* y que se realice la acción. Con esto solo crearía usuarios pero no les modificaría el *password*. Hay diferentes formas de inicializar el *password*, ya que solo se debe ejecutar una vez y cuando no esté inicializado y aquí seguiremos una de ellas, tratada en [37]. Para ello utilizaremos un *script* que deberemos enviar desde el servidor al cliente y por lo cual debemos:

```
Archivo /etc/puppet/fileserver.conf modificar:
    [files]
        path /etc/puppet/files
        allow *
Archivo /etc/puppet/auth.conf incluir:
    path /files
        auth *
Archivo /etc/puppet/puppet.conf en la sección [main] incluir:
        pluginsync = true
Rearrancar el servidor: /etc/init.d/puppetmaster restart
```

Ahora crearemos un script */etc/puppet/files/shadow.sh* que nos permitirá saber si debemos modificar el passwd del */etc/shadow* o no:

```
    #!/bin/bash
    rc=` /bin/grep $1 /etc/shadow | awk -F":" '($2 == " ! ")' | wc -l `
    if [ $rc -eq 0 ]
        then
            exit 1
        else
            exit 0
        fi
```

Ya tenemos las modificaciones en el archivo */etc/puppet/modules/accounts/init.pp* donde le decimos qué archivo hay que transferir y dónde (file) y luego el exec, que permite cambiar el *passwd* si el *script* nos devuelve un 0 o un 1. Luego deberemos ejecutar nuevamente sobre el servidor `puppet apply -v /etc/puppet/manifests/site.pp` y sobre el cliente para actualizar el catálogo `puppet agent -v -test` y verificar si el usuario se ha creado y podemos acceder. Como complemento a Puppet existe el Puppet Dashboard, pero este software está sin mantenimiento, por lo cual no se recomienda instalarlo (a no ser que sea estrictamente necesario -indicaciones en Wiki de Debian*-). En su lugar se puede instalar **Foreman** [39, 40], que es una aplicación que se integra muy bien con Puppet, permite ver en una consola toda la información y seguimiento, así como hacer el despliegue y automatización en forma gráfica de una instalación. La instlación es muy simple [40, 41]:

***https://wiki.debian.org/PuppetDashboard**

```
Inicializamos el repositorio e instalamos el paquete foreman-installer:
        echo "deb http://deb.theforeman.org/ wheezy stable" > \
            /etc/apt/sources.list.d/foreman.list
        wget -q http://deb.theforeman.org/foreman.asc -O- | apt-key add -
        apt-get update
        apt-get install -y foreman-installer
Ejecutamos el instalador con -i (interactive) para verificar los setting:
        foreman-installer -i

Contestamos (y) a la pregunta y seleccionamos las opciones
1. Configure foreman, 2. Configure foreman_proxy, 3. Configure puppet.
Veremos que el proceso puede acabar con unos errores (apache), pero no debemos preocuparnos
y tenemos que reiniciar el servicio con service apache2 restart.

Nos conectamos a la URL https://sysdw.nteum.org/, aceptamos el certificado y entramos
con usuario admin y passwd changeme.
Sobre Infrastructure > Smart Proxy hacemos clic en New Proxy y seleccionamos el nombre
y la URL: https://sysdw.nteum.org:8443.

Lo siguiente es hacer que se ejecute el puppet sobre el servidor (si no está puesto
en marcha se debe poner en marcha service puppet start y puede ser necesario modificar
el archivo /etc/default/puppet y ejecutamos puppet agent -t.

Sobre Foreman veremos que el DashBoard cambia y actualiza los valores para la gestión
integrada con puppet.
```

3.4. Chef

Chef es otra de la grandes herramientas de configuración con funcionalidades similares a Puppet (existe un buen artículo donde realiza una comparación sobre el dilema de Puppet o Chef [42]). Esta herramienta utiliza un lenguaje (basado en Ruby) DSL (*domain-specific language*) para escribir las 'recetas' de configuración que serán utilizadas para configurar y administrar los servidores de la compañía/institución y que además puede integrarse con diferentes plataformas (como Rackspace, Amazon EC2, Google y Microsoft Azure) para, automáticamente, gestionar la provisión de recursos de nuevas máquinas. El administrador comienza escribiendo las 'recetas' que describen cómo maneja Chef las aplicaciones del servidor (tales como Apache, MySQL, or Hadoop) y cómo serán configuradas indicando qué paquetes deberán ser instalados, los servicios que deberán ejecutarse y los archivos que deberán modificarse informando además de todo ello en el servidor para que el administrador tenga el control del despliegue. Chef puede ejecutarse en modo cliente/servidor o en modo *standalone* (llamado 'chef-solo'). En el modo C/S, el cliente envía diversos atributos al servidor y el servidor utiliza la plataforma `Solr` para indexar estos y proveer la API correspondiente, y utiliza estos atributos para configurar el nodo. 'chef-solo' permite utilizar las recetas en nodos que no tengan acceso al servidor y solo necesita 'su' receta (y sus dependencias) que deben estar en el disco físico del nodo (chef-solo es una versión con funcionalidad limitada de chef-client). Chef junto con Puppet, CFEngine, Bcfg2 es una de las herramientas más utilizadas para este tipo de funcionalidad y que ya forma parte de las media-grandes instalaciones actuales de GNU/Linux (es interesante ver los detalles del despliegue de la infraestructura Wikipedia que, excepto *passwords* y certificados, todo está documentado en http://blog.wikimedia.org/2011/09/19/ever-wondered-how-the-wikimedia-servers-are-configured/).

La instalación básica de Chef-server y Chef-client puede ser un poco más complicada, pero comenzamos igualmente con máquinas que tengan el */etc/hosts* y *hostname* bien configuradas. En este caso sugerimos trabajar con máquinas virtuales Ubuntu (14.04 a ser posible), ya que existen dependencias de la librería Libc (necesita la versión 2.15) y ruby (1.9.1) que en Debian Wheezy generan problemas (incluso utilizando el repositorio experimental). El primer paso es bajarnos los dos paquetes de *http://www.getchef.com/chef/install/* y ejecutar en cada uno de ellos `dpkg -i paquete-correspondiente.deb` (es decir, el servidor en la máquina servidora y el cliente en la máquina cliente). Después de cierto tiempo en el servidor podemos ejecutar `chef-server-ctl reconfigure`, que reconfigurará toda la herramienta y creará los certificados. Cuando termine podremos conectarnos a la URL *https://sysdw.nteum.org/* con usuario y *password* admin/pssw0rd1.

Sobre la interfaz gráfica, ir a *Client* > *Create* y crear un cliente con el nombre deseado (pucli en nuestro caso) y marcando en la casilla Admin y hacer *Create Client*. Sobre la siguiente pantalla se generarán las llaves privadas y pública para este cliente y deberemos copiar y salvar la llave privada en un archivo (por ejemplo */root/pucli.pem*). Sobre el cliente deberemos crear un directorio */root/.chef* y desde el servidor le copiamos la llave privada `scp /root/pucli.pem pucli:/root/.chef/pucli.pem`. Sobre el cliente hacemos `knife configure` y los datos deberán quedar como:

```
log_level                :info
log_location             STDOUT
node_name                'pucli'
client_key               '/root/.chef/pucli.pem'
validation_client_name   'chef-validator'
validation_key           '/etc/chef/validation.pem'
chef_server_url          'https://sysdw.nteum.org'
cache_type               'BasicFile'
cache_options( :path => '/root/.chef/checksums' )
cookbook_path ['/root/chef-repo/cookbooks' ]
```

Luego podremos ejecutar `knife client list` y nos deberá mostrar los clientes, algo como:

```
chef-validator
chef-webui
pucli
```

A partir de este punto estamos en condiciones de crear nuestra primera receta (*cookbook*) pero dada la complejidad de trabajar con este paquete y los conocimientos necesarios, recomendamos comenzar trabajando en el cliente como 'chef-solo' para automatizar la ejecución de tareas y pasar luego a describir las tareas del servidor y ejecutarlas remotamente. Para ello recomendamos seguir la guía en http://gettingstartedwithchef.com/.[44, 43, 45], que si bien se deben hacer algunos cambios en cuanto a los *cookbook* que se deben instalar, la secuencia es correcta y permite aprender cómo trabajar y repetir los

resultados en tantos servidores como sea necesario (se recomienda hacer el procedimiento en uno y luego sobre otra máquina sin ningún paquete, instalar el chef-solo, copiar el repositorio del primero y ejecutar el chef-solo para tener otro servidor instalado exactamente igual que el primero.

3.5. Vagrant

Esta herramienta es útil en entornos DevOps y juega un papel diferente al de Docker pero orientado hacia los mismos objetivos: proporcionar entornos fáciles de configurar, reproducibles y portátiles con un único flujo de trabajo que ayudará a maximizar la productividad y la flexibilidad en el desarrollo de aplicaciones/servicios. Puede aprovisionar máquinas de diferentes proveedores (VirtualBox, VMware, AWS, u otros) utilizando *scripts*, Chef o Puppet para instalar y configurar automáticamente el software de la VM. Para los desarrolladores Vagrant aislará las dependencias y configuraciones dentro de un único entorno disponible, consistente, sin sacrificar ninguna de las herramientas que el desarrollador utiliza habitualmente y teniendo en cuenta un archivo, llamado *Vagrantfile*, el resto de desarrolladores tendrá el mismo entorno aun cuando trabajen desde otros SO o entornos, logrando que todos los miembros de un equipo estén ejecutando código en el mismo entorno y con las mismas dependencias. Además, en el ámbito IT permite tener entornos desechables con un flujo de trabajo coherente para desarrollar y probar scripts de administración de la infraestructura ya que rápidamente se pueden hacer pruebas de scripts, 'cookbooks' de Chef, módulos de Puppets y otros que utilizan la virtualización local, tal como VirtualBox o VMware. Luego, con la misma configuración, puede probar estos *scripts* en el *cloud* (p. ej., AWS o Rackspace) con el mismo flujo de trabajo.

En primer lugar es necesario indicar que deberemos trabajar sobre un sistema Gnu/Linux base (lo que se define como *Bare-Metal*, es decir, sin virtualizar, ya que necesitaremos las extensiones de HW visibles y si hemos virtualizado con Virtualbox, por ejemplo, esto no es posible). Es recomendable instalar la versión de Virtualbox (puede ser la del repositorio Debian), verificar que todo funciona, y luego descargar la última versión de Vagrant desde *http://www.vagrantup.com/downloads.html* e instalarla con la instrucción `dpkg -i vagrant_x.y.z_x86_64.deb` (donde x.y.z será la versión que hemos descargado).

A partir de este punto es muy simple ejecutando [46] la instrucción `vagrant init hashicorp/precise32`, que inicializará/generará un archivo llamado **Vagrantfile** con las definiciones de la VM y cuando ejecutemos **vagrant up** descargará del repositorio *cloud* de Vagrant una imagen de Ubuntu 12.04-32b y la pondrá en marcha. Para acceder a ella simplemente debemos hacer `vagrant ssh` y si queremos desecharla, `vagrant destroy`. En el *cloud* de Vagrant [47] podemos acceder a diferentes imágenes preconfiguradas* que po-

***https://vagrantcloud.com /discover/featured**

dremos cargar simplemente haciendo `vagrant box add nombre`, por ejemplo `vagrant box add chef/centos-6.5`. Cuando se ejecute el comando 'up' veremos que nos da una serie de mensajes, entre los cuales nos indicará el puerto para conectarse a esa máquina virtual directamente (p.ej., `ssh vagrant@localhost -p 2222` y con *passwd* **vagrant**). Para utilizar una VM como base, podemos modificar el archivo Vagrantfile y cambiar el contenido:

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise32"
end
```

Si deseamos trabajar con dos máquinas en forma simultánea deberemos modificar el archivo Vagrantfile con lo siguiente:

```
Vagrant.configure("2") do |config|
  config.vm.define "centos" do |centos|
    centos.vm.box = "chef/centos-6.5"
  end
  config.vm.define "ubu" do |ubu|
    ubu.vm.box = "hashicorp/precise32"
  end
end
```

Podremos ver que les asigna un puerto SSH a cada una para evitar colisiones cuando hacemos el `vagrant up`. Desde la VM podríamos instalar el software como se hace habitualmente, pero para evitar que cada persona haga lo mismo existe una forma de aprovisionamiento que se ejecutará cuando se haga el 'up' de la VM. Para ello, escribimos un *script init.sh* con el siguiente contenido:

```
#!/usr/bin/env bash
apt-get update
apt-get install -y apache2
rm -rf /var/www
ln -fs /tmp /var/www
```

En este script (para no tener problemas de permisos para esta prueba) hemos apuntando el directorio */var/www* a */tmp*. A continuación modificamos el Vagrantfile (solo hemos dejado una VM para acelerar los procesos de carga):

```
Vagrant.configure("2") do |config|
  config.vm.define " ubu" do |ubu|
    ubu.vm.box = "hashicorp/precise32"
    ubu.vm.provision :shell, path: " init.sh"
  end
end
```

Luego deberemos hacer `vagrant reload -provision`. Veremos cómo se aprovisiona y carga Apache y luego, si en entramos en la máquina y creamos un */tmp/index.html* y hacemos `wget 127.0.0.1` (o la ip interna de la

máquina),veremos que accedemos al archivo y lo bajamos (igualmente si hacemos `ps -edaf | grep apache2` veremos que está funcionando). Por último y para verla desde el *host* debemos redireccionar el puerto 80 por ejemplo al 4444. Para ello debemos agregar en el mismo archivo (debajo de donde pone ubu.vm.provision) la línea: `config.vm.network :forwarded_port, host: 4444, guest: 80`. Volvemos a hacer `vagrant reload -provision` y desde el host nos conectamos a la url: *127.0.0.1:4444* y veremos el contenido del index.html.

En estas pruebas de concepto hemos mostrado algunos aspectos interesantes pero es una herramienta muy potente que se debe analizar con cuidado para configurar las opciones necesarias para nuestro entorno, como por ejemplo, aspectos del *Vagrant Share* o cuestiones avanzadas del *Provisioning* que aquí solo hemos tratado superficialmente.[46]

Actividades

1. Instalad y configurad OpenMPI sobre un nodo; compilad y ejecutad el programa `cpi.c` y observad su comportamiento.

2. Instalad y configurad OpenMP; compilad y ejecutad el programa de multiplicación de matrices (https://computing.llnl.gov/tutorials/openMP/exercise.html) en 2 *cores* y obtened pruebas de la mejora en la ejecución.

3. Utilizando dos nodos (puede ser con VirtualBox), Cacti y XYmon y monitorizad su uso.

4. Utilizando Rocks y VirtualBox, instalad dos máquinas para simular un clúster.

5. Instalad Docker y cread 4 entornos diferentes.

6. Instalad Puppet y configurar un máquina cliente.

7. Ídem punto anterior con Chef.

8. Con Vagrant cread dos VM, una con Centos y otra con Ubuntu y aprovisionar la primera con Apache y la segunda con Mysql. Las máquinas se deben poder acceder desde el host y comunicar entre ellas.

Bibliografía

[1] *Beowulf cluster*.
<http://en.wikipedia.org/wiki/Beowulf_cluster>

[2] **S. Pereira** *Building a simple Beowulf cluster with Ubuntu*.
<http://byobu.info/article/Building_a_simple_Beowulf_cluster_with_Ubuntu/>

[3] **Radajewski, J.; Eadline, D.** *Beowulf: Installation and Administration*. TLDP.
<http://www2.ic.uff.br/ vefr/research/clcomp/Beowulf-Installation-and-Administration-HOWTO.html>

[4] **Swendson, K**. *Beowulf HOWTO* (tlpd).
<http://www.tldp.org/HOWTO/Beowulf-HOWTO/>

[5] **Barney, B.** *OpenMP*. Lawrence Livermore National Laboratory.
<https://computing.llnl.gov/tutorials/openMP/>

[6] *OpenMP Exercise*.
<https://computing.llnl.gov/tutorials/openMP/exercise.html>

[7] *MPI 3*.
<http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf>

[8] *MPI Examples*.
<http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/main.htm>(Download Section)

[9] *Mpich1 Project*.
<http://www.mcs.anl.gov/research/projects/mpi/>

[10] *MPICH, High-Performance Portable MPI* .
<http://www.mpich.org/>

[11] *LAM/MPI*.
<http://www.lam-mpi.org/ >

[12] *OpenMPI*.
<http://www.open-mpi.org/>

[13] *FAQ OpenMPI*.
<http://www.open-mpi.org/faq/>

[14] **Woodman, L.** *Setting up a Beowulf cluster Using Open MPI on Linux*.
<http://techtinkering.com/articles/?id=32>

[15] *Rocks Cluster. Base Users Guide*.
<http://central6.rocksclusters.org/roll-documentation/base/6.1.1/>

[16] *Cacti*.
<http://www.cacti.net/>

[17] *Xymon*.
<http://xymon.sourceforge.net/>

[18] *Cloud Computing. Retos y Oportunidades*.
<http://www.ontsi.red.es/ontsi/sites/default/files/2-_resumen_ejecutivo_cloud_computing_vf.pdf>

[19] *Eucalyptus, CloudStack, OpenStack and OpenNebula: A Tale of Two Cloud Models*.
<http://opennebula.org/eucalyptus-cloudstack-openstack-and-opennebula-a-tale-of-two-cloud-models/>

[20] *OpenNebula vs. OpenStack: User Needs vs. Vendor Driven*.
<http://opennebula.org/opennebula-vs-openstack-user-needs-vs-vendor-driven/>

[21] *OpenNebula Documentation*.
<http://opennebula.org/documentation/>

[22] *Quickstart: OpenNebula on Ubuntu 14.04 and KVM*.
<http://docs.opennebula.org/4.6/design_and_installation/quick_starts/qs_ubuntu_kvm.html>

[23] *LXC*.
<https://wiki.debian.org/LXC>

[24] *LXC en Ubuntu*.
<https://help.ubuntu.com/lts/serverguide/lxc.html>

[25] **S. Graber***LXC 1.0*.
<https://wiki.debian.org/BridgeNetworkConnections>

[26] *LXC: Step-by-Step Guide*.
<https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/>

[27] *A Short List of DevOps Tools*.
<http://newrelic.com/devops/toolset>

[28] *Preseeding d-i en Debian*.
<https://wiki.debian.org/DebianInstaller/Preseed>

[29] *FAI (Fully Automatic Installation) for Debian GNU/Linux*.
<https://wiki.debian.org/FAI>

[30] *FAI (Fully Automatic Installation) project and documentation*.
<http://fai-project.org/>

[31] *El libro del administrador de Debian. Instalación automatizada*.
<http://debian-handbook.info/browse/es-ES/stable/sect.automated-installation.html>

[32] *Octopussy. Open Source Log Management Solution*.
<http://8pussy.org/>

[33] *Understanding Docker*.
<https://docs.docker.com/introduction/understanding-docker/>

[34] *Docker Docs*.
<https://docs.docker.com/>

[35] *Docker HUB*.
<https://registry.hub.docker.com/>

[36] *Puppet Labs Documentation*.
<http://docs.puppetlabs.com/>

[37] *Puppet - Configuration Management Tool*.
<http://puppet-cmt.blogspot.com.es/>

[38] *Learning Puppet*.
<http://docs.puppetlabs.com/learning/ral.html>

[39] *Foreman - A complete lifecycle management tool*.
<http://theforeman.org/>

[40] *Foreman - Quick Start Guide*.
<http://theforeman.org/manuals/1.5/quickstart_guide.html>

[41] *How to Install The Foreman and a Puppet Master on Debian Wheezy*.
<http://midactstech.blogspot.com.es/2014/02/PuppetMasterForeman-Wheezy.html>

[42] *Puppet or Chef: The configuration management dilemma*.
<http://www.infoworld.com/d/data-center/puppet-or-chef-the-configuration-management-dilemma-215279?page=0,0>

[43] *Download Chef: Server and Client*.
<http://www.getchef.com/chef/install/>

[44] **A. Gale***Getting started with Chef*.
<http://gettingstartedwithchef.com/>

[45] *Chef Cookbooks*.
<https://supermarket.getchef.com/cookbooks-directory>

[46] *Vagrant: Getting Started*.
<http://docs.vagrantup.com/v2/getting-started/index.html >

[47] *Vagrant Cloud*.
<https://vagrantcloud.com/>

[48] *KVM*.
<https://wiki.debian.org/es/KVM>

[49] *VirtualBox*.
<https://wiki.debian.org/VirtualBox>

[50] *Guía rápida de instalación de Xen*.
<http://wiki.debian.org/Xen>

[51] *The Xen hypervisor*.
<http://www.xen.org/>

[52] *Qemu*.
<http://wiki.qemu.org/Main_Page>

[53] *QEMU. Guía rápida de instalación e integración con KVM y KQemu*.
<http://wiki.debian.org/QEMU>