UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Máster en Ingeniería de Computadores y Redes

Trabajo Fin de Máster

Configuración, optimización y evaluación de un servidor de alta disponibilidad con equilibrado de carga

Autor:
Abraham Jaramillo Garófalo

Directores:
Pedro Juan López Rodríguez
María Elvira Baydal Cardona

CURSO 2015 - 2016

Resumen

El trabajo realizado contempla la configuración de un sistema servidor de alta disponibilidad con equilibrado de carga. El sistema estará formado por dos tipos de nodos: servidores y directores. Los nodos servidores encargados de ofrecer el servicio, se apoyan sobre la virtualización para una mayor flexibilidad y fácil despliegue. Con el objetivo de determinar la plataforma más adecuada, se ha realizado un análisis comparativo de prestaciones entre las plataformas de virtualización KVM, XEN y VirtualBox.

Los nodos directores se ocupan de repartir las peticiones entrantes utilizando la herramienta HAProxy. La alta disponibilidad se ha conseguido mediante la herramienta *Keepalived*.

En el desarrollo del trabajo se han propuesto diversos escenarios, que permitan evaluar las prestaciones del equilibrador de carga, y así, optimizar el sistema valiéndonos de conceptos de afinidad de memoria y técnicas avanzadas en la gestión de interrupciones de red.

Palabras clave: Servidor de Internet, equilibrado de carga, alta disponibilidad, HAProxy, keepalived, evaluación de prestaciones, optimización.

Índice de abreviaturas

VM (*Virtual Machines*), Máquinas virtuales.

PV (*Para-virtualization*), Para-virtualización.

FV (*Full virtualization*), Virtualización Completa.

HVM (*Hardware-assisted virtualization*), Virtualización asistida por Hardware.

NAT (*Network address translation*), Traducción de direcciones de red.

VRRP (*Virtual Redundancy Routing Protocol*), Protocolo de redundancia.

VIP (*Virtual IP*), IP Virtual

TCP (*Transmission Control Protocol*), Protocolo de Control de Transmisión

UDP (*User Datagram Protocol*), Protocolo de datagramas de usuario

ARP (*Address Resolution Protocol*), Protocolo de resolución de direcciones

NUMA (*Non-Uniform Memory Access*), Acceso no uniforme a memoria

RSS (*Receive-Side Scaling*)

RPS (*Receive Packet Steering*)

RFS (*Receive Flow Steering*)

NIC (*Network Interface Card*), Tarjeta de interfaz de red

ALU (*Arithmetic Logic Unit*), Unidad aritmético lógica

FPU (*Floating Point* Unit), Unidad de coma flotante

MMX (*Matrix Math Extensions*)

SSE (*Streaming SIMD Extensions*)

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

La presencia masiva de dispositivos conectados a la red y la enorme demanda de recursos en línea ha ido aumentado de manera considerable en los últimos años. Se prevé que a finales de 2016 el tráfico IP mundial anual será de 1,1 zettabytes, y que crecerá hasta 2,3 zettabytes en 2020, fecha para la que se estima que el número de dispositivos conectados a Internet será superior al triple de la población mundial [1]. Por tal motivo el uso de sistemas que soporten elevadas cargas de trabajo junto con la alta disponibilidad, se convierten en un gran desafío para todo administrador de sistemas.

La solución adoptada reside en el uso de herramientas que permitan el reparto de carga entre diferentes servidores o nodos de cómputo, con el objetivo de alcanzar potencias de cálculo por encima de la capacidad individual de cada dispositivo. Así mismo la planificación inteligente de los recursos del sistema junto con el uso adecuado de herramientas de optimización, permitirá sacar el máximo provecho de nuestros recursos y mejorar las prestaciones.

Por tal motivo el proyecto propuesto tiene como objetivo configurar un servidor de Internet de alta disponibilidad con equilibrado de carga, haciendo uso de las herramientas HAProxy y *Keepalived*. Además, analizar el comportamiento de cada uno de los componentes del sistema y aplicar técnicas de optimización a nivel de procesos y recursos de red; y así evaluar las prestaciones alcanzadas con cada una de las técnicas aplicadas.

1.1 Estructura del servidor de Internet

El servidor configurado está conformado por dos nodos directores duplicados, encargados de equilibrar la carga a través de la herramienta HAProxy. En un instante dado, solo uno de ellos actuará como repartidor activo. El otro director se utiliza únicamente con propósitos de tolerancia a fallos. Las peticiones serán procesadas por un conjunto de servidores virtualizados, debido a la flexibilidad que ofrece la virtualización en la administración de los recursos del servidor y el rápido despliegue sobre los diferentes nodos del clúster. En principio, los servidores virtuales pueden ser configurados para ofrecer cualquier servicio. Sin embargo, se ha considerado configurarlos como servidores web, encargados de atender las peticiones de los clientes, que derivan del nodo director activo. Es importante resaltar que los repartidores de carga serán instalados sobre el hardware, ya que se desea obtener elevadas prestaciones, que en cierto modo se penalizan con el uso de la virtualización. La Figura 1 muestra la estructura del servidor y el camino que siguen las peticiones HTTP.

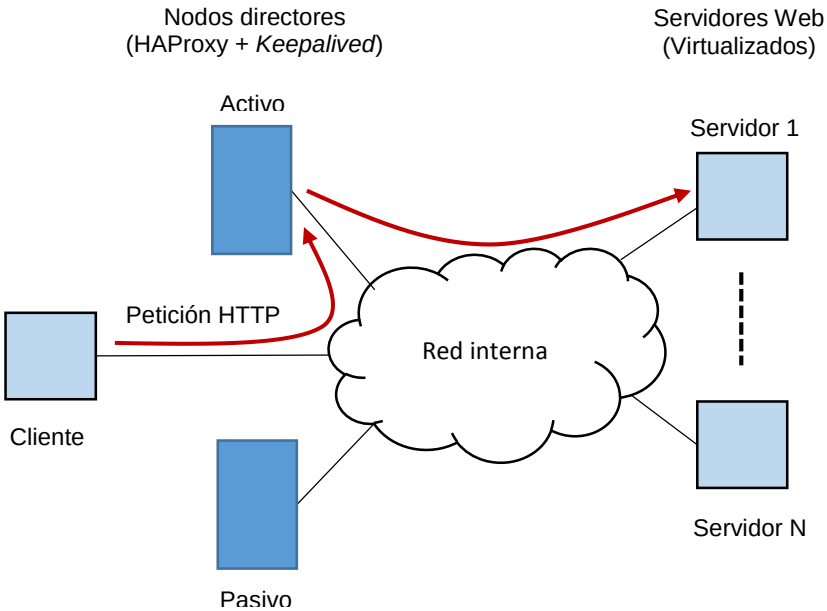


Figura 1. Estructura del servidor de Internet

1.2 Objetivos

A continuación se muestran los objetivos que se fijaron en el proyecto.

- Configurar un servidor de alta disponibilidad con equilibrado de carga.
- Configurar los servidores basados en máquinas virtuales, para mayor flexibilidad de su configuración y despliegue.
- Configurar los nodos directores para que realicen funciones de equilibrado de carga.
- Replicar los nodos directores para proveer alta disponibilidad.
- Optimizar el servidor configurado valiéndose de conceptos de afinidad en la ejecución de procesos y gestión de las interrupciones de red.
- Evaluar las prestaciones del servidor configurado y establecer comparativas.

1.3 Metodología aplicada

La ejecución de este proyecto contempló las siguientes etapas:

- Planificación: se realizó un análisis previo a la implementación del clúster de altas prestaciones, tomando en consideración diferentes escenarios de prueba con varias plataformas enfocadas a la virtualización y servicios web.
- Implementación: se basó en la configuración de los escenarios sobre los cuales se llevaron a cabo las pruebas. Se ha trabajado sobre dos escenarios; el primer escenario ha servido para poder seleccionar la plataforma de virtualización que alcance las mejores prestaciones, pero sobre todo proporcione mayor flexibilidad en su uso y despliegue en los nodos del clúster.

El segundo escenario se encuentra compuesto por un clúster con 6 nodos de cómputo, además de 2 nodos directores replicados encargados de equilibrar la carga entre los servidores. Adicional a esto se han considerado 3 nodos clientes para generar carga sobre el clúster.

En este segundo escenario se pondrán en funcionamiento las herramientas de alta disponibilidad y equilibrado de carga; donde a partir de diferentes configuraciones aplicaremos técnicas de optimización que permitan mejorar las prestaciones de nuestro clúster.

- Experimentación: se basó en la realización de un conjunto de pruebas con el fin de obtener datos relativos a la tasa de transacciones alcanzadas por unidad de tiempo, haciendo uso de las herramientas *apache benchmark* y *weighttp*; para de esta manera evaluar las diferentes plataformas.

- Recopilación de datos: se realizaron gráficos estadísticos para representar los datos obtenidos en la evaluación.

- Análisis de los datos: se exponen los resultados obtenidos para establecer comparativas en cuanto a las prestaciones de los sistemas.

1.4 Estructura del documento

La memoria del presente trabajo se compone de cuatro capítulos, que procedemos a exponer a continuación:

- **Capítulo 1, Introducción:** Este capítulo introduce el proyecto realizado y la metodología utilizada, así como una visión general del funcionamiento del servidor y los objetivos a alcanzar.

- **Capítulo 2, Conceptos previos:** Se exponen los conocimientos necesarios para el debido entendimiento y comprensión de los temas que se abordan en la realización del trabajo.

- **Capítulo 3, Plataformas de Virtualización:** Se han ejecutado diversas pruebas que permitan realizar una comparativa de prestaciones sobre diferentes plataformas de virtualización y de esta manera seleccionar la que mejor se adapte a nuestro clúster.

- **Capítulo 4, Servidor web de alta disponibilidad con equilibrado de carga:** En este capítulo se plantea la configuración del servidor de Internet, incluyendo los nodos servidores y los nodos directores. También se aplican diferentes estrategias para mejorar las prestaciones del equilibrador de carga, valiéndonos de conceptos de afinidad de procesos y optimización en la gestión de interrupciones de red. Como paso final, se provee al sistema de alta disponibilidad.

CAPÍTULO 2

Conceptos Previos

En este capítulo se exponen varios conceptos básicos relacionados con el trabajo desarrollado.

2.1 Virtualización

Las tecnologías de virtualización ofrecen la posibilidad de compartir los recursos de un computador entre múltiples sistemas operativos invitados o máquinas virtuales (VMs), haciendo uso del particionado de hardware/software, emulación, compartición de tiempo y recursos dinámicos.

De manera tradicional los sistemas operativos han sido los encargados de controlar el hardware, sin embargo, las tecnologías de virtualización añaden una nueva capa entre el sistema operativo y el hardware. Esta capa de virtualización proporciona una infraestructura que permite soportar múltiples VMs y mantenerlas aisladas e independientes unas de otras. A menudo se suele llamar hipervisor o monitor de máquina virtual (VMM) a la capa de virtualización.

Existen tres enfoques diferentes asociados a la virtualización: la para-virtualización (PV), la virtualización completa (FV) y la virtualización asistida por hardware (HVM), que se muestran en la Figura 2. La para-virtualización requiere modificación en el sistema operativo del huésped, básicamente con el objetivo de adaptarlo a las necesidades del hipervisor y de esta manera poder tener acceso a recursos restringidos del sistema. La virtualización completa no

requiere la modificación del sistema operativo huésped, ya que funciona a través de traducción binaria. Sin embargo, esta técnica puede introducir una mayor sobrecarga debido a que las instrucciones que manejan recursos protegidos, deben ser interceptadas y reescritas. Por último, la virtualización asistida por hardware permite que el hipervisor acceda a los recursos restringidos del sistema de manera más eficiente, ya que hace uso del soporte a virtualización provisto por los procesadores más modernos como Intel VT o AMD-V.

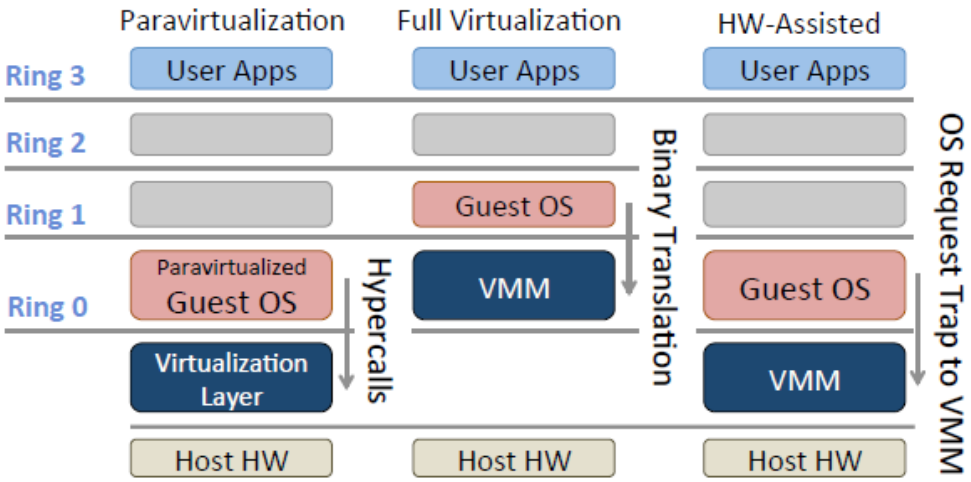


Figura 2. Enfoques de la capa de virtualización [2]

2.2 Equilibrado de carga

Un concepto muy utilizado dentro de los sistemas informáticos es el de equilibrado de carga. Empieza desde la planificación de procesos en un procesador multinúcleo y se extiende hasta la capa más alta del modelo TCP/IP, donde un conjunto de dispositivos de igual o diferente capacidad realizan una misma tarea de manera simultánea, con el objetivo de alcanzar una capacidad de procesamiento por encima de la capacidad individual de cada dispositivo. El objetivo es que la carga de trabajo se reparta entre los distintos dispositivos.

En los servidores de internet, el mecanismo que permite que todo esto sea posible son los llamados equilibradores de carga, que pueden actuar a diferentes niveles, ya sea a nivel de enlace, a nivel de red o a nivel de aplicación.

A nivel de enlace se puede mencionar la tecnología Link Aggregation (LAG) definida en el estándar IEEE 802.1AX-2008 [3], la cual permite agrupar de manera lógica varios enlaces de red, con el objetivo de alcanzar velocidades superiores a la nominal de cada interfaz física. Ver Figura 3.

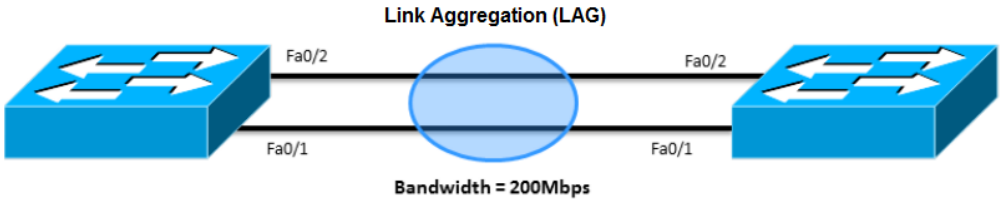


Figura 3. Equilibrado de carga a nivel de enlace [4]

El equilibrado de carga a nivel de red permite a uno o más nodos directores distribuir las peticiones de los clientes entre una serie de nodos servidores, basándose únicamente en la dirección de red provista por la cabecera de los paquetes. Una vez realizado el procesamiento, los paquetes devueltos por los nodos servidores pueden ser enviados directamente al cliente (Direct Routing) o volver a pasar por el nodo director cuando se emplea NAT.

En este proyecto trataremos los equilibradores de carga a nivel de aplicación, los cuales actúan sobre el contenido de una sesión TCP. Esto significa que existe la posibilidad de analizar determinados datos de la petición y utilizarlos como política en el reparto de la carga, lo que obliga a que el tráfico vuelva a pasar por el equilibrador de carga, añadiendo tiempo extra de procesamiento, lo cual no siempre ocurre en los equilibradores a nivel de red. Sin embargo, los equilibradores a nivel de aplicación pueden ofrecer amplias posibilidades y realizar diversas tareas complejas sobre el tráfico.

Uno de los desafíos de los equilibradores de carga es la persistencia de las sesiones, lo cual implica que las múltiples peticiones de un cliente sean enviadas

y respondidas por el mismo servidor. Esto se puede lograr de dos formas, almacenando la información del servidor objetivo en los repartidores de carga o bien en el cliente.

En la Figura 4 se muestra una configuración típica de un sistema con equilibrado de carga y alta disponibilidad que consta de dos etapas. En la primera etapa llamada *frontend*, un solo dispositivo o nodo director se encuentra activo repartiendo la carga hacia los servidores reales. Además, cada nodo director tiene dos interfaces de red para de esta manera regular el tráfico entre la red externa (Internet) y la interna (Privada). En la segunda etapa denominada *backend*, se encuentran los servidores reales o nodos de cómputo, que son los encargados de ejecutar las tareas solicitadas por los clientes.

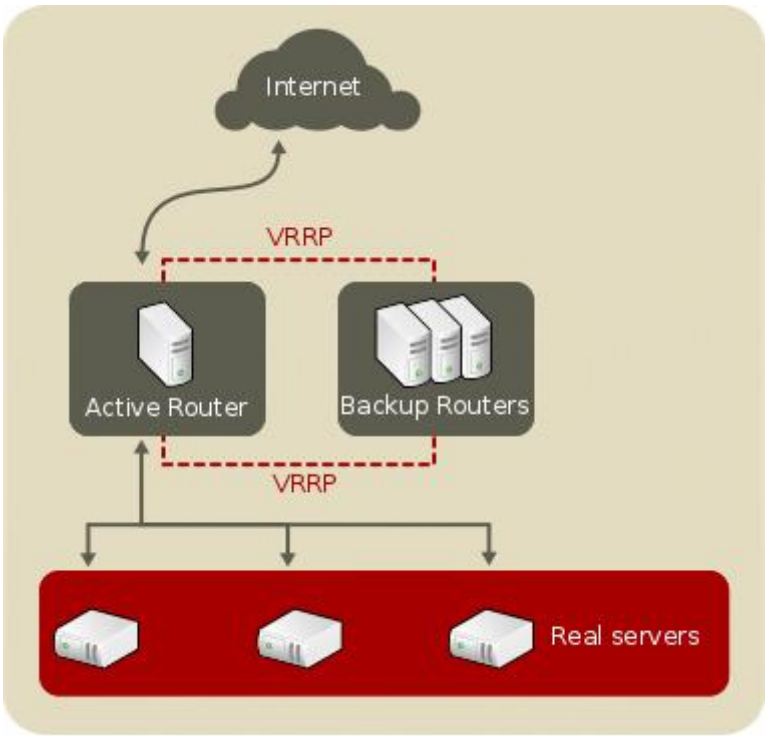


Figura 4. Configuración básica de equilibrado de carga y alta disponibilidad [5]

Como ya sabemos, el equilibrado de carga se puede conseguir a diferentes niveles. Sin embargo, para los servidores de internet es común trabajar con herramientas que realicen el reparto de carga en las capas 4 o 7. Una de las alternativas para equilibrar la carga en la capa 4, es la herramienta LVS, que

trabaja a nivel del núcleo del sistema operativo Linux, con lo cual se obtiene una mayor velocidad en el procesamiento de los paquetes. Sin embargo, las posibilidades son mucho más amplias con un equilibrador de carga en la capa 7, ya que los datos de una petición pueden ser analizados y en base a eso, decidir el mejor destino para los paquetes, permitiendo un análisis de grano fino sobre las peticiones. Una de las herramientas que permiten el reparto de carga a nivel de aplicación, es HAProxy [6], la cual se encuentra muy difundida y en constante actualización, siendo una herramienta utilizada por un gran número de sitios web conocidos, como por ejemplo Instagram [7].

2.2.1. HAProxy

HAProxy es un equilibrador de carga a nivel de aplicación basado en eventos, diseñado con el objetivo de reenviar los datos de manera rápida, ejecutando la menor cantidad de operaciones posibles [8].

El funcionamiento de HAProxy se puede resumir en los siguientes pasos:

- Aceptar conexiones entrantes de los sockets definidos en la configuración de cada *frontend*.
- Aplicar reglas de procesamiento en el *frontend* a las conexiones entrantes, las cuales pueden resultar en bloqueo o modificación de algunas cabeceras del segmento TCP, como por ejemplo el puerto destino.
- Reenviar las conexiones entrantes al *backend*, el cual representa el conjunto de servidores encargados de responder a alguna o a varias tareas específicas. Las peticiones son reenviadas haciendo uso de alguna estrategia de equilibrado de carga, como por ejemplo round robin. También puede repartirse la carga para garantizar que las peticiones de un cliente dado se atiendan siempre por el mismo servidor.

- Se puede volver aplicar otras reglas de procesamiento para los datos de respuesta, tanto en el *backend* como *frontend*.

- Emitir un registro detallado de lo que ha sucedido

Es importante destacar que HAProxy periódicamente analiza el estado de los servidores del *backend* para determinar si se encuentran activos, así como también puede realizar intercambio de información con otros nodos directores, en su caso.

2.3 Alta disponibilidad

La alta disponibilidad como su nombre indica, intenta que algún servicio siga estando activo frente a errores que puedan ocurrir a nivel de hardware o del sistema operativo, haciendo uso de técnicas de tolerancia a fallos en los diferentes puntos vitales del sistema, conocidos como puntos únicos de fallo.

Existen diferentes técnicas que pueden asegurar la alta disponibilidad, pero básicamente en un clúster se hace uso de la redundancia de los diferentes elementos. En nuestro clúster, nos interesa que el repartidor de carga sea redundante. Para ello, se definirá una dirección IP virtual (VIP) que flotará entre los dos nodos directores. En un momento dado, solo estará activa en uno de los dos nodos. En caso que falle el nodo activo, la VIP se asignará al otro nodo, el cual se hará cargo del servicio. Una de las posibilidades para configurar una VIP es a través del protocolo VRRP (Virtual Redundancy Routing Protocol), el cual envía anuncios desde el elemento activo hacia los demás dispositivos redundantes en intervalos periódicos. En el caso de que los anuncios no sean recibidos por los demás nodos, un nuevo elemento tomará el control del servicio.

2.3.1. Keepalived

Es una herramienta que implementa el protocolo VRRP. Tiene como objetivo principal dotar a sistemas Linux de alta disponibilidad y equilibrado de carga en la capa 4 o nivel de transporte, por lo que esta herramienta puede trabajar tanto con protocolos TCP como UDP [9].

En nuestro caso utilizaremos la herramienta *keepalived* para proveer de alta disponibilidad al conjunto de nodos directores gestionando la configuración de la VIP. De esta manera aseguramos que el reparto de carga se siga realizando hacia los nodos de cómputo, en caso de que falle el nodo activo del *frontend*.

2.4 Optimización de recursos

En esta parte trataremos distintos conceptos que nos permitirán sintonizar de manera más eficiente nuestro sistema, valiéndonos de conceptos tales como afinidad en la ejecución de procesos e interrupciones de red.

2.4.1. Planificación

Por defecto los procesos pueden ejecutarse sobre cualquier núcleo, lo cual conlleva que las aplicaciones compitan por los recursos de la CPU. Sin embargo, el hecho de que el planificador del sistema operativo cambie la CPU sobre la cual se está ejecutando un proceso, puede generar fallos de cache que ralentizan la aplicación. La disponibilidad de múltiples de núcleos de procesamiento en las máquinas actuales, permite ajustar los procesos para que se ejecuten siempre en un determinado núcleo, siguiendo algún criterio de asignación. Esto puede ayudar a que las aplicaciones alcancen un mayor rendimiento.

2.4.2. Afinidad

Dentro de un procesador multinúcleo pueden existir desde uno a varios dominios NUMA (*Non-Uniform Memory Access*). Cada uno de ellos representa un conjunto de procesadores con recursos independientes. Por ejemplo, es habitual que cada procesador tenga las caches L1 y L2 privadas, y que todos los procesadores de un dominio NUMA compartan la cache L3. Por otra parte, también es frecuente que cada dominio NUMA tenga asignada una parte de la memoria. Por lo tanto, cada procesador accederá a su propia memoria local de manera más rápida que a una ubicada en un dominio NUMA diferente.

La afinidad de memoria cache explota el hecho de tener niveles de cache compartidos entre los núcleos de procesamiento de un dominio NUMA. Para ello, se intenta que los procesos que mantienen relación entre ellos se ejecuten dentro de un mismo dominio NUMA. Esto minimiza la cantidad de datos que atraviesan los distintos dominios y por ende maximiza las posibilidades de aciertos en cache de cada NUMA, lo cual se traduce en una mejora global de las prestaciones [10].

2.4.3. Procesamiento de las interrupciones de red

Para poder aplicar las técnicas de optimización en la gestión de las interrupciones de red, será necesario entender cómo funciona la recepción de los paquetes. La Figura 5 muestra el camino que siguen los paquetes desde la recepción en la interfaz de red hasta la aplicación.

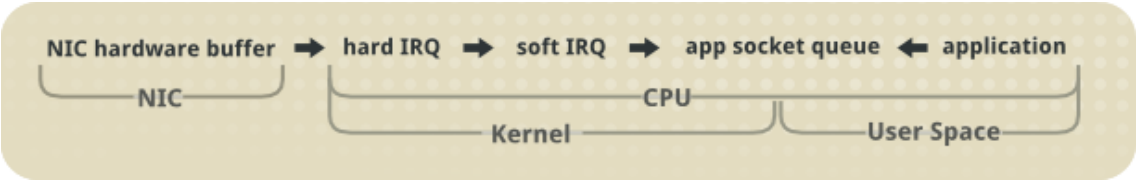


Figura 5. Esquema de recepción de paquetes [11]

En la primera etapa, la NIC (*Network Interface Card*) recibe la trama y la transfiere hacia un buffer hardware interno donde será almacenada. A partir de este punto la NIC indica a la CPU la presencia de una trama de red mediante una interrupción hardware. Esto causa que el controlador de la NIC reconozca la interrupción y programe la correspondiente interrupción software. Dentro de este contexto, el núcleo del sistema operativo elimina la trama del buffer de la NIC y la procesa a través de la pila de red, para luego enviarla al correspondiente socket de escucha, donde es asociada con la aplicación. Este proceso se repite de manera iterativa, hasta que el buffer de la NIC se quede sin tramas.

Para mantener un alto rendimiento al procesar los paquetes entrantes, existen diversas técnicas que nos permiten tratar las interrupciones de red de manera más eficiente. Algunas vienen dadas por el propio hardware, como por ejemplo las tarjetas de red con múltiples colas (RSS) o pueden venir de la mano de aplicaciones que ayuden a esta tarea, como RPS y RFS. En una primera aproximación podríamos decir que la solución hardware sería más eficiente en comparación con una solución basada en software, ya que en la mayoría de casos las soluciones software introducen cierta sobrecarga al sistema. Sin embargo, veremos que también con las soluciones software se podrán alcanzar prestaciones considerables y a un menor coste económico que la solución hardware.

2.4.4. RSS

Receive-Side Scaling (RSS) distribuye la recepción de paquetes de red hacia diferentes colas basadas en hardware, permitiendo que el tráfico de red entrante pueda ser distribuido entre diferentes procesadores, con el objetivo de aliviar los cuellos de botella que se puedan producir por la sobrecarga de que una sola CPU procese todas las interrupciones de red [12].

La distribución de los paquetes se puede realizar de diferentes formas. Algunas tarjetas de red permiten la aplicación de un filtro sobre cada paquete. De esta forma, los paquetes se asignan a un pequeño número de flujos lógicos,

los cuales son dirigidos hacia colas separadas para que pueden ser procesadas por distintas CPUs.

2.4.5. RPS

Receive Packet Steering (RPS) es similar a RSS, ya que permite direccionar los paquetes de red para que sean procesados por una CPU específica. Sin embargo, RPS se encuentra implementado a nivel de software, ayudando así a prevenir que la única cola de una tarjeta de red no se convierta en el cuello de botella del sistema.

RPS tiene ciertas ventajas sobre RSS ya que puede ser utilizado sobre cualquier interfaz de red. También es sencillo de configurar y no aumenta la tasa de interrupciones hardware del dispositivo de red. Sin embargo, introduce interrupciones entre los procesadores.

2.4.6. RFS

Receive Flow Steering (RFS) extiende el comportamiento de RPS pero de manera inteligente, ya que intenta aumentar la tasa de aciertos en cache, enviando los paquetes de red hacia la CPU asociada a la aplicación que hará uso de tales paquetes.

CAPÍTULO 3

Plataformas de Virtualización

El uso de los sistemas de virtualización por parte de los centros de datos se ha consolidado, debido a la flexibilidad que la virtualización ofrece en la gestión de los recursos de un servidor. Sin embargo, la disponibilidad de diversas plataformas de virtualización plantea un gran desafío a la hora de seleccionar la plataforma más adecuada, que se adapte a las necesidades del servicio a ofrecer y, sobre todo, explote al máximo las prestaciones del equipo anfitrión.

Entre las más conocidas podemos nombrar VirtualBox [13], XEN [14], KVM [15], vSphere [16], Hyper-V [17]. Sin embargo, en este trabajo se han considerado las tres primeras debido a que no son soluciones de pago. Para seleccionar la más adecuada, se ha realizado una evaluación de sus prestaciones. En este capítulo se muestran los resultados obtenidos y se justifica la elección realizada.

3.1 Arquitectura de los hipervisores seleccionados

En la sección 2.1 se ha hecho énfasis en los distintos enfoques que presentan los hipervisores en función de su implementación. Además, también es importante conocer la arquitectura básica de los hipervisores utilizados, para poder sacar mejores conclusiones de las pruebas realizadas.

3.1.1. Arquitectura de XEN

Xen es capaz de trabajar sobre cualquier tipo de virtualización PV, FV o HVM. Sin embargo, indistintamente del modo en que trabaje, las máquinas virtuales no privilegiadas (domU) no tienen acceso directo al hardware, quedándose totalmente aisladas del hardware como se muestra en la Figura 6. Es el dom0 o dominio administrativo, quien tiene los permisos especiales para acceder directamente al hardware y manipular las funciones del sistema e interactuar con las máquinas virtuales. Una de las características principales de Xen es que debe iniciarse antes que el sistema operativo, ya que de esta manera puede tomar control sobre el hardware y distribuir su uso entre las máquinas virtuales.

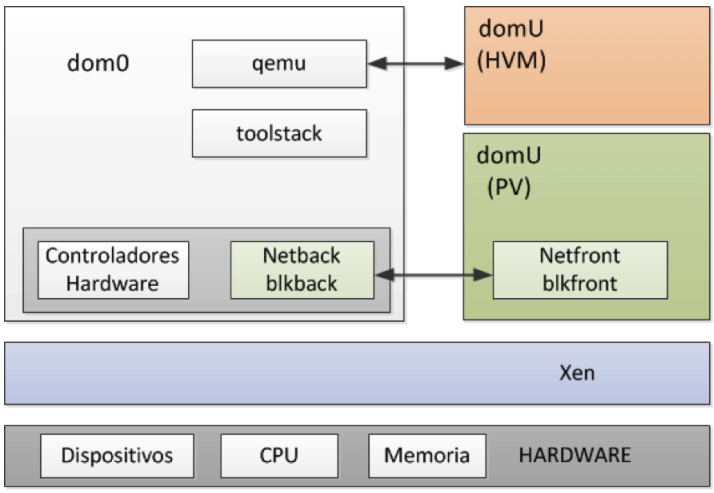


Figura 6. Arquitectura Xen [18]

3.1.2. Arquitectura de KVM

KVM es una solución de tipo FV, pero hace uso del soporte a virtualización que ofrecen los procesadores como Intel VT o AMD-V. Una de las características que favorece el uso extendido de esta plataforma, es que viene incluida en el núcleo de Linux desde la versión 2.6.20. En esta arquitectura, las máquinas virtuales son consideradas como procesos normales, por lo tanto la gestión de memoria y la planificación de procesos son los estándares del sistema operativo. KVM añade un modo de ejecución adicional llamado invitado como se muestra

en la Figura 7, el cual será el modo de ejecución para el sistema operativo huésped, siempre y cuando no existan operaciones de entrada/salida. El modo usuario es el que será utilizado para ejecutar las operaciones de entrada/salida del sistema invitado, permitiendo gestionar los dispositivos virtuales a nivel de usuario [19].

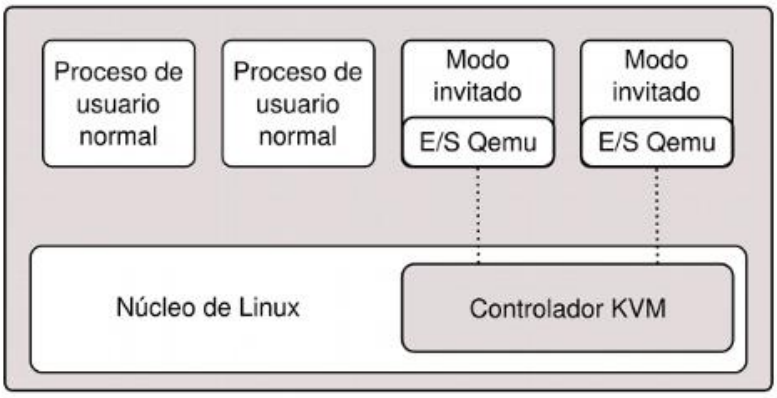


Figura 7. Arquitectura KVM

3.1.3. Arquitectura de VirtualBox

VirtualBox es una plataforma de virtualización que puede trabajar sobre múltiples sistemas operativos. Lo que hace es aplicar un parche al sistema operativo anfitrión, para que de esta manera el hipervisor se ejecute en un nivel por debajo del anfitrión y así poder servir a los requerimientos de las VMs. Otra de las características del virtualizador es que puede operar sin usar el soporte a virtualización por hardware que ofrecen los procesadores. Sin embargo, se recomienda usarlo para mejorar las prestaciones de las VMs.

El hipervisor también incorpora controladores para gestionar el acceso al hardware virtualizado, lo cual mejora la velocidad de ejecución de las máquinas virtuales, ya que simplifica la emulación de los dispositivos y la interacción entre el sistema anfitrión y el huésped [19].

3.2 Selección de la plataforma de virtualización

La metodología aplicada para la comparativa de prestaciones entre los hipervisores consistió en someter a estrés cada uno de los componentes principales del sistema, los cuales incluyen: CPU, memoria RAM, disco I/O y red, así como la ejecución de algunas aplicaciones de interés.

Las pruebas se han realizado sobre un servidor con las siguientes características: CPU Intel Core i5-760 de 2.80 GHz con 4 núcleos, memoria RAM DDR3 de 4096 MB y 500 GB de disco duro.

Las plataformas de virtualización puestas a prueba fueron KVM, Xen y Virtualbox. Para poder obtener resultados comparables, se ha utilizado la misma configuración de hardware para cada una de las plataformas virtuales, que consiste en: 4 núcleos de procesamiento, memoria RAM de 2048 MB y 20 GB de almacenamiento.

Para la configuración de red se han utilizado las interfaces de red virtuales por defecto. En el caso de KVM y XEN es la virtio; y para Virtualbox es la Intel PRO/1000 MT. Es importante destacar que la interfaz de red se ha configurado en modo puente. Así se evita la sobrecarga introducida por el empleo de NAT en la traducción de la dirección interna de la VM hacia la externa del equipo anfitrión.

3.2.1. Pruebas realizadas

Sobre cada plataforma virtual se han realizado seis grupos de pruebas:

- Compilación del núcleo de Linux
- *Benchmark* de la CPU
- *Benchmark* de memoria RAM
- *Benchmark* del sistema de almacenamiento
- *Benchmark* de la red

- Servidor Web

3.2.2. Compilación del núcleo de Linux

La compilación necesita utilizar la CPU, requiere memoria y acceder al disco, con lo que nos puede dar una primera impresión acerca de las prestaciones. Por lo tanto, el punto de partida para la comparación fue realizar la compilación del núcleo de Linux 3.13.1 (Ver anexo A.2) sobre el cual se estaban ejecutando las plataformas virtuales, y así comparar los tiempos de compilación entre las diferentes plataformas tal como se muestra en la Figura 8.

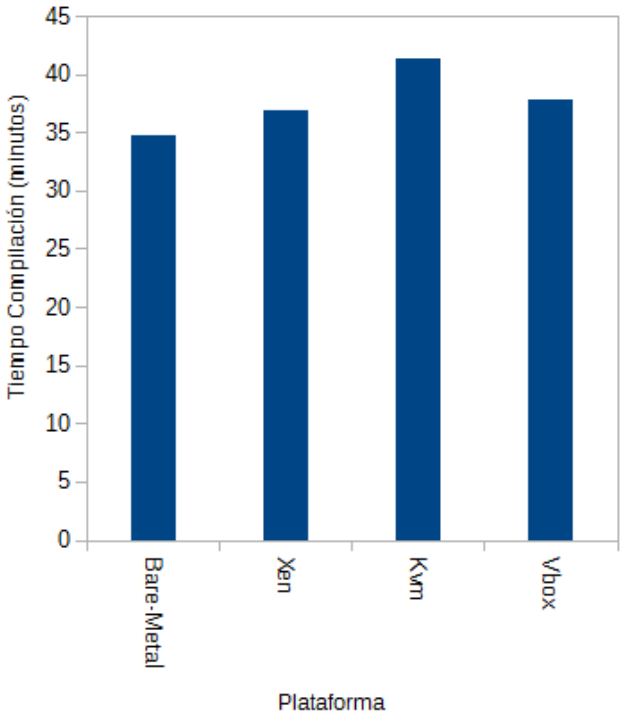


Figura 8. Tiempo de compilación del núcleo Linux 3.13.1

A simple vista se puede observar que todas tardan aproximadamente lo mismo, con una ligera desventaja de KVM.

3.2.3. CPU

Para las pruebas de CPU se ha utilizado la herramienta Bytemark [20], la cual está diseñada para generar cargas de trabajo de distintos tipos sobre la CPU, en las cuales constan operaciones aritméticas de enteros y coma flotante (Ver anexo A.3).

La Figura 9 muestra los resultados obtenidos para cada una de las plataformas virtuales, donde es evidente que se obtienen menores prestaciones con el hipervisor XEN, siendo un 12% inferior a la ejecución en Bare-Metal.

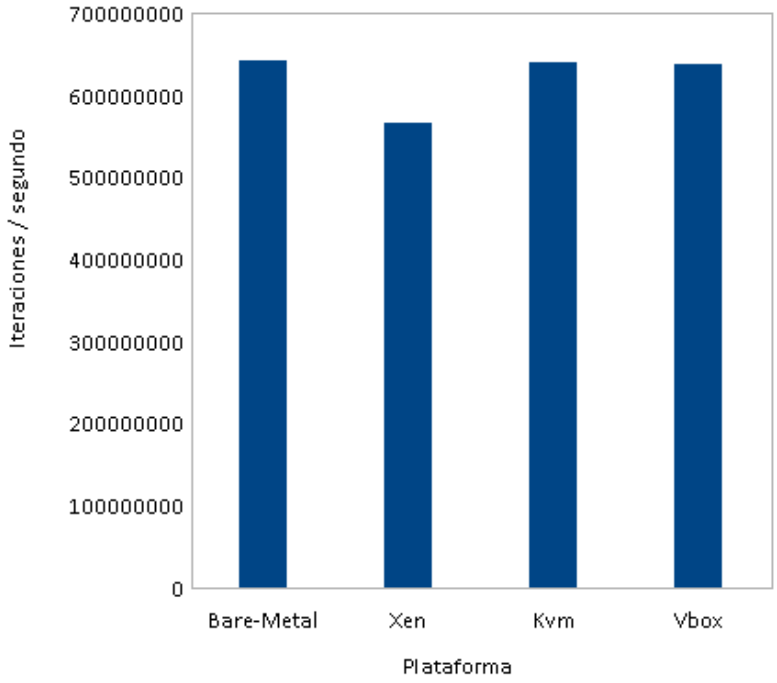


Figura 9. Benchmark CPU

De todas maneras se puede observar que los resultados no reflejan caídas muy considerables en las prestaciones y esto se debe a la tecnología de virtualización asistida por hardware del procesador Intel, la cual permite que las operaciones se ejecuten de manera directa sobre la CPU sin ayuda del hipervisor, obteniendo velocidades relativamente cercanas al Bare-Metal.

3.2.4. Memoria RAM

Para la RAM se ha utilizado la herramienta RAMspeed [21], la cual nos permitirá medir el ancho de banda disponible, haciendo uso de 4 algoritmos que reservan cierto espacio de memoria para diversas operaciones realizadas sobre la ALU, FPU, MMX y SSE respectivamente (Ver anexo A.4).

En la Figura 10 se puede observar que la velocidad máxima alcanzada por el Bare-Metal es de 11000 MB/s aproximadamente, lo cual se acerca a la velocidad teórica del tipo de memorias DDR3 utilizado para las pruebas. Sin embargo, los resultados muestran nuevamente una disminución de las prestaciones por parte de la plataforma XEN, donde la velocidad máxima alcanzada es de 10000 MB/s aproximadamente para las operaciones con coma flotante.

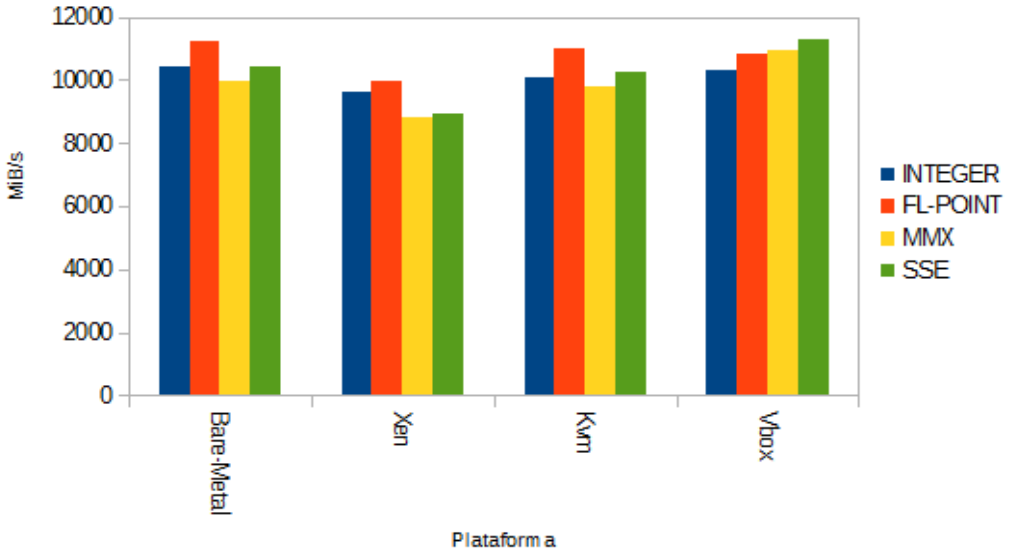


Figura 10. Benchmark memoria RAM

3.2.5. Almacenamiento

Para medir la velocidad de lectura y escritura sobre el disco se ha utilizado la herramienta Bonnie++ [22], donde es importante definir el tamaño de memoria RAM disponible, para así disminuir los efectos de la cache de los sistemas de archivos (Ver anexo A.5).

En la Figura 11 se puede observar que se alcanzan valores similares de lectura para cada una de las plataformas, aunque se reducen respecto de las obtenidas por el Bare-Metal. Sin embargo, las prestaciones de escritura para la plataforma XEN se reducen en aproximadamente 15,18% en relación con las demás plataformas.

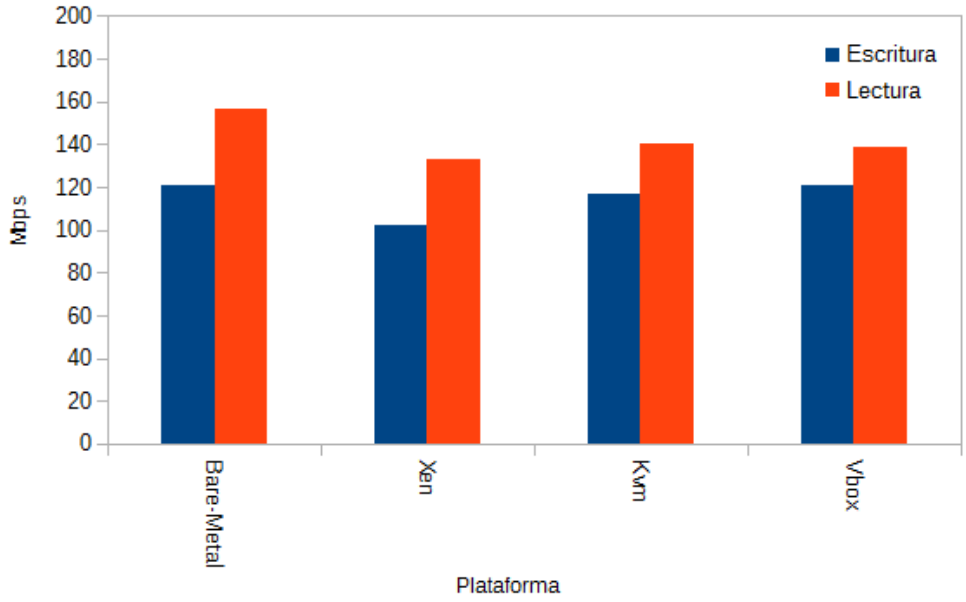


Figura 11. Benchmark de lectura y escritura en disco

3.2.6. Red

Uno de los aspectos fundamentales a considerar es la sobrecarga que induce la virtualización sobre la interfaz de red, por lo que se han realizado pruebas para medir la latencia lanzando 10 peticiones ICMP por segundo (Ver anexo A.6), obteniendo los resultados de la Figura 12, donde es evidente la sobrecarga que induce la tarjeta de red virtualizada en la plataforma VirtualBox, de aproximadamente el 51% en relación con el Bare-Metal.

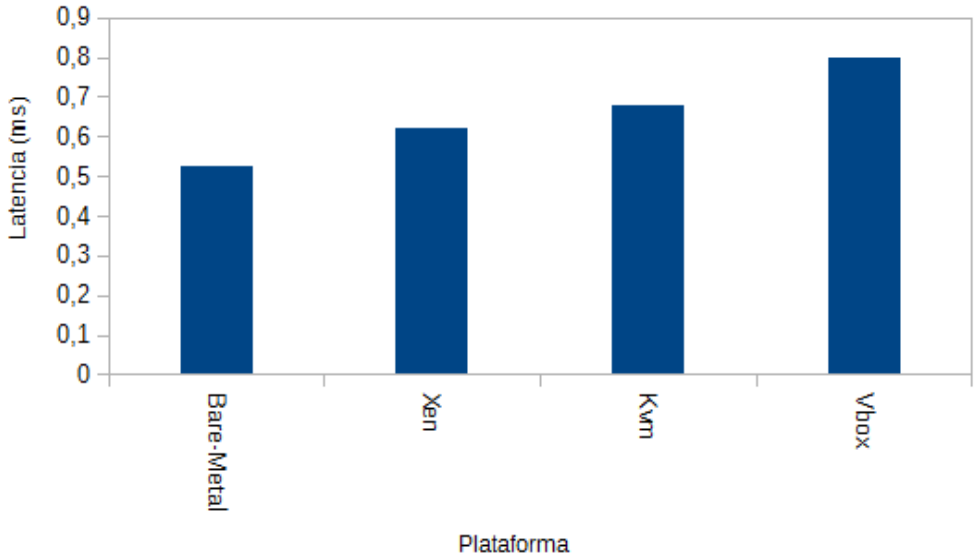


Figura 12. Benchmark interfaz de red

3.2.7. Ejecución de un servidor web

Otro punto importante que nos permitirá seleccionar la plataforma de virtualización adecuada, será ejecutar la aplicación que se pretende ofrecer como servicio (Ver anexo A.7). Para las pruebas se ha utilizado el servidor de aplicaciones web apache 2.4.7 ofreciendo una página html muy simple (index.html, ver anexo A.9). Las pruebas se han obtenido utilizando la herramienta apache benchmark (Ver anexo A.7)

La Figura 13 muestra los resultados obtenidos de las solicitudes por segundo procesadas por el servidor, donde se mantiene la tendencia observada en la Figura 12, en la cual VirtualBox obtuvo el valor de latencia más alto, que se ha visto reflejado en una menor cantidad de solicitudes por segundo servidas, llegando a un valor máximo de 3500 peticiones por segundo, frente a las casi 12000 alcanzadas por la máquina física. El resto de plataformas de virtualización obtienen resultados menores (en torno a 10000 - 11000 peticiones por segundo) aunque aceptables.

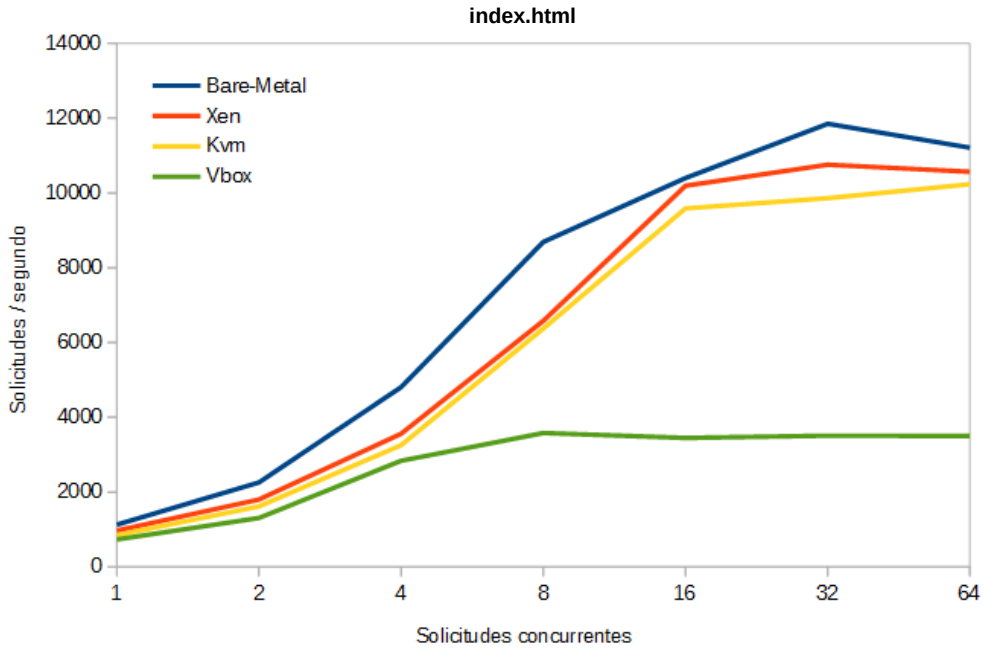


Figura 13. Benchmark servidor web (index.html)

3.2.8. Selección de la plataforma de virtualización

De acuerdo a las diferentes pruebas realizadas hemos podido notar los beneficios de la virtualización asistida por hardware del procesador, viéndose reflejado en una mínima degradación de las prestaciones en la parte de procesamiento y el sistema en general. Sin embargo, a lo largo de las pruebas hemos podido notar un ligero menor rendimiento por parte de la plataforma XEN, lo cual tiene su origen en la administración independiente de los recursos por parte del dom0, el cual controla las máquinas virtuales y el acceso a los diferentes recursos del sistema. En cambio KVM se ejecuta como un módulo del kernel, permitiendo así el uso de las características del sistema operativo por sí mismo.

VirtualBox no se queda atrás, ya que con las características de aceleración introducidas en las últimas versiones, ha mostrado ser una plataforma con mucho potencial, llegando a alcanzar prestaciones cercanas al Bare-Metal para la mayoría de las pruebas realizadas. Sin embargo, para las pruebas usando la aplicación apache, se han obtenido resultados muy por debajo de las demás

plataformas. Por lo tanto, en nuestro clúster haremos uso de la plataforma KVM, debido además a la flexibilidad que ofrece en la instalación sobre los nodos del clúster y sobre todo a la facilidad que nos brinda la interfaz en el momento de enlazar núcleos virtuales con físicos, siendo ésta la base para optimizar las prestaciones de las VMs, como veremos más adelante.

CAPÍTULO 4

Servidor web de alta disponibilidad con equilibrado de carga

En este capítulo se describe y se evalúa el clúster configurado para ofrecer servicios de Internet de alta disponibilidad con equilibrado de carga. En primer lugar dado que nuestro servidor ofrece un servicio web, se realizará la comparativa de prestaciones de diferentes aplicaciones web, entre las cuales se han considerado Apache [23], Nginx [24] y Gwan [25]. Como los nodos servidores que ofrecen el servicio web se encuentran virtualizados por motivos de flexibilidad, será importante determinar la distribución más adecuada de las VMs sobre los nodos del clúster, para que no afecte las prestaciones de manera considerable.

Seguidamente, haremos uso de la herramienta HAProxy sobre los nodos directores y determinaremos las prestaciones utilizando diferentes configuraciones de HAProxy. También analizaremos diversas técnicas para mejorar el procesamiento de las peticiones, utilizando técnicas de optimización a nivel de procesos y gestión de recursos de red. Finalmente, dotaremos al sistema de alta disponibilidad con la herramienta *keepalived*.

4.1 Diagrama de bloques del clúster

A partir de este capítulo trabajaremos sobre el clúster real del GAP (Grupo de Arquitecturas Paralelas) que se muestra en la Figura 14, donde cada nodo de

cómputo posee características similares, mostradas en la Tabla I (Para un mayor detalle ver anexo A.1). Una de las características de este clúster, es que todos los nodos se encuentran en la misma subred. Las peticiones llegarían desde Internet (habitualmente a través de un firewall) al nodo repartidor de carga, el cual se encarga de enviar las peticiones hacia los diferentes nodos de cómputo, encargados de procesar las solicitudes de los clientes y reenviar las respuestas a través del nodo director.

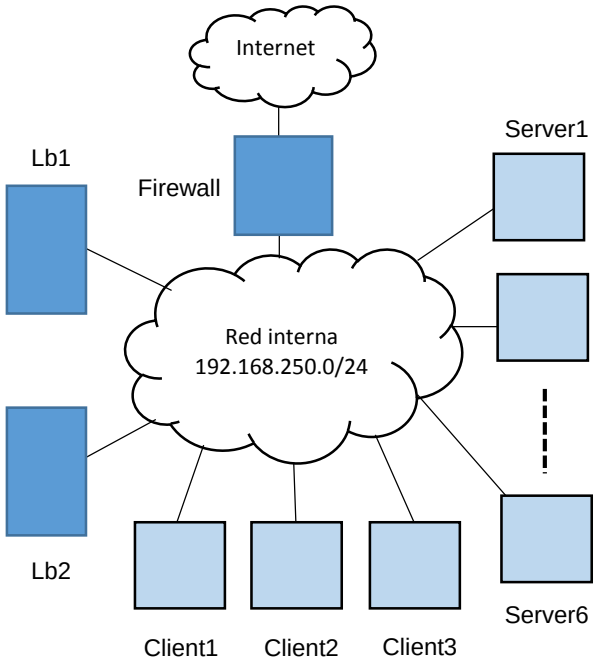


Figura 14. Esquema del clúster utilizado

Dispositivo	IP	Función	Características
Lb1	192.168.250.121	Equilibrador de carga (*Master*)	4 NUMA Quad-Core AMD Opteron(tm) Processor 8350 2GHz L1 cache 512 KB L2 cache 2 MB L3 cache 2 MB Memoria Ram 16384 MB DDR2 333 MHz
Lb2	192.168.250.111	Equilibrador de carga (*Backup*)	
Server1	192.168.250.126	Nodos de cómputo	
Server2	192.168.250.128		
Server3	192.168.250.132		
Server4	192.168.250.117		

Server5	192.168.250.108		
Server6	192.168.250.110		
Client1	192.168.250.113	Generadores de Carga de Trabajo	
Client2	192.168.250.115		
Client3	192.168.250.116		

Tabla I. Detalles del clúster

Sin embargo, en nuestro clúster hemos conectado los nodos cliente directamente a la subred interna, con el objetivo de realizar una evaluación de prestaciones del sistema en condiciones de máximo estrés. Estos nodos cliente serán los que realicen las peticiones al servidor configurado.

En una implementación real debería haber también un almacenamiento compartido con todos los archivos utilizados por el servicio (en nuestro caso apache) u aplicación. Sin embargo, en nuestro caso los servidores almacenarán localmente las páginas que sirven, para así evitar que el almacenamiento compartido se pueda convertir en el cuello de botella.

4.2 Nodos servidores

Los nodos servidores ejecutan varias máquinas virtuales, encargadas de ofrecer el servicio web a los clientes, por lo que en esta sección compararemos los servidores web Apache, Nginx y Gwan. Además, analizaremos la distribución más adecuada de las VMs sobre los nodos servidores.

Es importante destacar que cada VM posee una dirección IP que se encuentra en la misma subred del clúster. Para las diferentes pruebas se han tomado en consideración los rangos que se muestran en la Tabla II, en función del nodo servidor donde serán ejecutadas las VMs.

Dispositivo	Rango de IPs para VMs
Server1	192.168.250.181 - 192.168.250.188
Server2	192.168.250.189 - 192.168.250.196
Server3	192.168.250.197 - 192.168.250.204
Server4	192.168.250.205 - 192.168.250.212
Server5	192.168.250.213 - 192.168.250.220
Server6	192.168.250.221 - 192.168.250.228

Tabla II. Direcciones de red para las VMs

4.2.1. Selección del servidor web

Según estadísticas de la W3Techs (*World Wide Web Technology Surveys*) la plataforma más utilizada por la mayoría de sitios web es Apache con un 52,6%, siguiéndole Nginx con un 30,1% [26], siendo pues evidente la popularidad del servidor apache debido a su larga trayectoria en este mercado y sus continuas mejoras en términos de prestaciones.

Otra plataforma que hemos incluido en la comparativa es el servidor Gwan, debido a los buenos resultados de prestaciones que expone en su sitio web, además de ser un proyecto reciente y ligero [25]. También, el sitio web incluye abundante material sobre cómo evaluar correctamente los servidores web.

Para la comparativa de las plataformas web seleccionadas, se ha hecho uso de la herramienta weighttp (Ver anexo A.8) que lanza varios hilos en la ejecución y distintos valores de peticiones concurrentes. Se han realizado dos tipos de prueba, una solicitando una página sencilla en html (index.html, ver anexo A.9) y otra con una página sencilla en php (index.php, ver anexo A.10).

Los resultados solicitando la página index.html se observan en la Figura 15, donde las tres plataformas web presentan una tendencia similar. Sin embargo, se debe destacar que la plataforma nginx ha alcanzado un máximo de 17.238 peticiones por segundo.

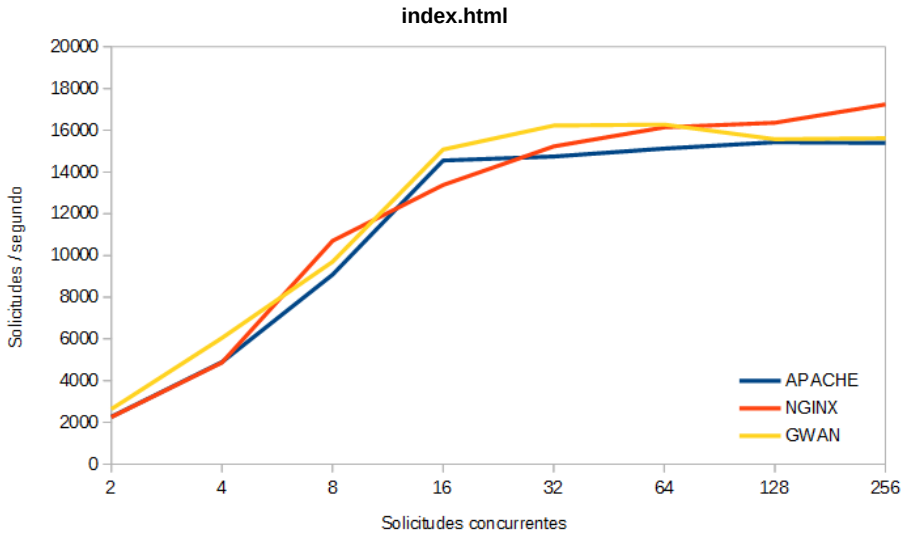


Figura 15. Benchmark plataformas web usando html

Los resultados para las pruebas utilizando la página index.php se pueden observar en la Figura 16, donde claramente se ve un bajo rendimiento por parte de la plataforma Gwan, llegando a un máximo de 200 peticiones por segundo aproximadamente. Es relevante destacar que las prestaciones de la plataforma Nginx mantienen una tendencia similar a la de Apache, con picos de 13.660 peticiones por segundo, lo cual muestra el gran potencial de esta joven plataforma, lanzada en el año 2004.

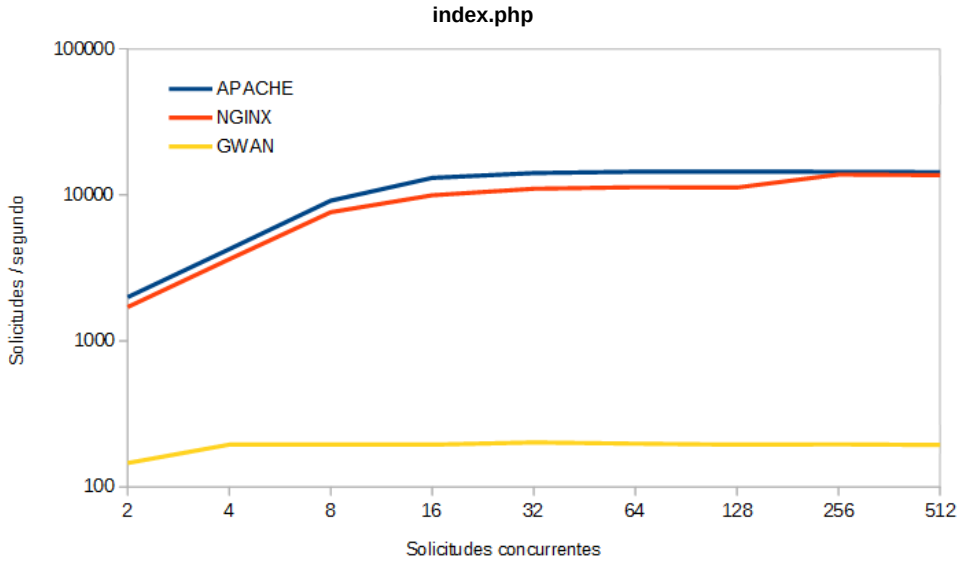


Figura 16. Benchmark plataformas web usando php

No obstante, tomando en consideración la popularidad de Apache, procederemos a seleccionarla en este trabajo. Además cabe mencionar que la configuración de Nginx con soporte para php requiere modificaciones adicionales en los archivos de configuración y la instalación de módulos extra para su correcto funcionamiento.

4.2.2. Disposición de las VMs sobre el *backend*

Otro desafío importante en la configuración de nuestro sistema, es la distribución inteligente de las VMs sobre cada uno de los nodos del clúster, lo cual supone que exista una asignación equilibrada de los recursos hacia cada una de las VMs. Por lo tanto, en este apartado se pretende determinar el número óptimo de VMs por cada nodo servidor real. Además, se desea comparar si es mejor la asignación manual de las VMs a las CPUs disponibles o la que realiza el sistema operativo.

Como hemos visto en el capítulo 2, la plataforma KVM trata a las VMs como procesos normales, por lo que en primera instancia podríamos dejar que el planificador del sistema operativo se encargue de repartir los recursos como mejor le convenga. Sin embargo, realizaremos diferentes pruebas para demostrar que esa no es la manera más eficiente de ejecutar las VMs.

En esta fase se ha utilizado uno de los nodos de cómputo del clúster, de manera específica el Server6, sobre el cual se han lanzado diferentes configuraciones de VMs manteniendo una distribución uniforme de los núcleos como se describe a continuación:

- 1 VM con 16 núcleos
- 2 VMs cada una con 8 núcleos
- 4 VMs cada una con 4 núcleos
- 8 VMs cada una con 2 núcleos
- 16 VMs cada una con 1 núcleo

Para las pruebas se ha utilizado la herramienta weighttp (Ver anexo A.8), lanzando 128 peticiones concurrentes, con dos hilos de ejecución, que solicita la página index.html (Ver anexo A.9). Para poder realizar las pruebas de manera simultánea sobre varias VMs, se han lanzado varias instancias del *benchmark.* A continuación se muestra para el caso de 2 VMs.

```
# Prueba con 2 VMs cada una con 8 núcleos

weighttp -n 100000 -c 128 -t 2 192.168.250.181/ > web1 &
weighttp -n 100000 -c 128 -t 2 192.168.250.182/ > web2
```

Cada escenario consta de una configuración base, donde la asociación de núcleos virtuales con los físicos la realiza el sistema operativo. Además, hemos probado una configuración optimizada, donde la asociación se realiza de manera manual sobre el archivo de configuración xml de cada VM. Por ejemplo, para 2 VMs con 8 núcleos cada una se muestra a continuación, donde *vcpupin* representa el núcleo virtual y *cpuset* el núcleo físico.

```
# Asignación manual para primera VM (2 VMs cada una con 8 núcleos)

..
<vcpu placement='static'>8</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='0'/>
    <vcpupin vcpu='1' cpuset='1'/>
    <vcpupin vcpu='2' cpuset='2'/>
    <vcpupin vcpu='3' cpuset='3'/>
    <vcpupin vcpu='4' cpuset='4'/>
    <vcpupin vcpu='5' cpuset='5'/>
    <vcpupin vcpu='6' cpuset='6'/>
    <vcpupin vcpu='7' cpuset='7'/>
  </cputune>
..
```

```
# Asignación manual para segunda VM (2 VMs cada una con 8 núcleos)
..
<vcpu placement='static'>8</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='8'/>
    <vcpupin vcpu='1' cpuset='9'/>
    <vcpupin vcpu='2' cpuset='10'/>
    <vcpupin vcpu='3' cpuset='11'/>
    <vcpupin vcpu='4' cpuset='12'/>
    <vcpupin vcpu='5' cpuset='13'/>
    <vcpupin vcpu='6' cpuset='14'/>
    <vcpupin vcpu='7' cpuset='15'/>
  </cputune>
..
```

Para que la asignación de núcleos se haga de manera automática en cada escenario, se ha realizado un script (Ver anexo A.14), el cual recibe como parámetro de entrada el número de VMs a utilizar. Se debe recalcar que se ha hecho una copia de cada archivo xml, omitiendo los tags *vcpu* y *cputune*, ya que serán generados de manera automática por el script.

Los resultados obtenidos se pueden observar en la Figura 17, donde en todos los casos las prestaciones son mejores cuando se realiza la asignación manual de los núcleos en las VMs, alcanzando un máximo de 7314 peticiones por segundo para la configuración de 4 VMs. Los resultados para las configuraciones con un número elevado de VMs tienden a variar mucho en cada una de las ejecuciones, lo cual puede atribuirse a los procesos *libvirt* que acompañan a las VMs en la administración de los recursos virtuales y que sin duda restan recursos al sistema y pueden interferir con otras aplicaciones.

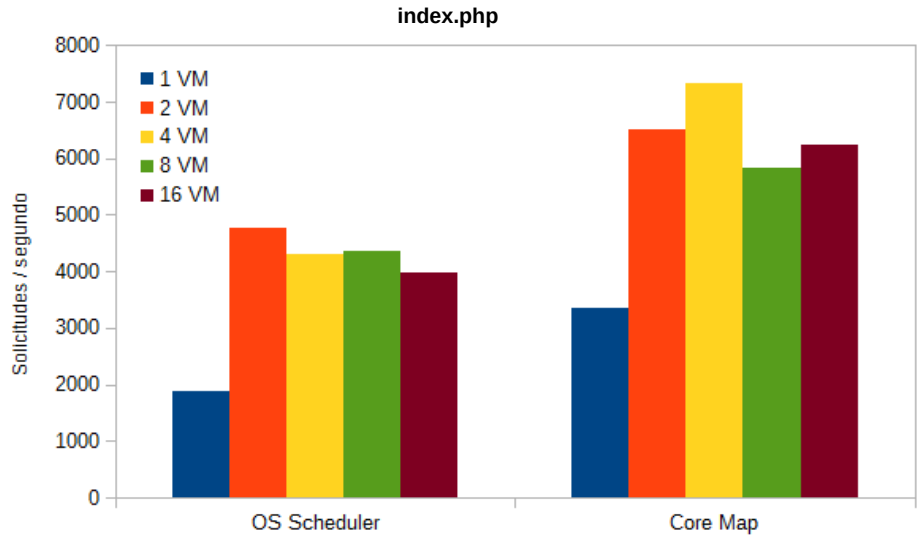


Figura 17. Comparación planificador OS vs asignación manual

La prueba anterior nos ha servido para comparar el comportamiento de las VMs utilizando el planificador del sistema operativo o con asignación manual. Sin embargo, para poder llegar a una conclusión acerca de la disposición de VMs sobre el *backend*, será necesario introducir una mayor carga de trabajo en los procesadores, para lo cual haremos uso de una página dinámica en php con la

carga de trabajo ajustable. (Cálculo de la constante pi por integración numérica, archivo pi.php, anexo A.11).

```
# Prueba con 2 VMs cada una con 8 núcleos

weighttp -n 100000 -c 128 -t 2 192.168.250.181/pi.php?n=100000 > web1 &
weighttp -n 100000 -c 128 -t 2 192.168.250.182/pi.php?n=100000 > web2
```

La Figura 18 muestra el resultado para el cálculo de pi con 100.000 iteraciones, donde se alcanzan 333 peticiones por segundo aproximadamente para las configuraciones de 4 y 8 VMs.

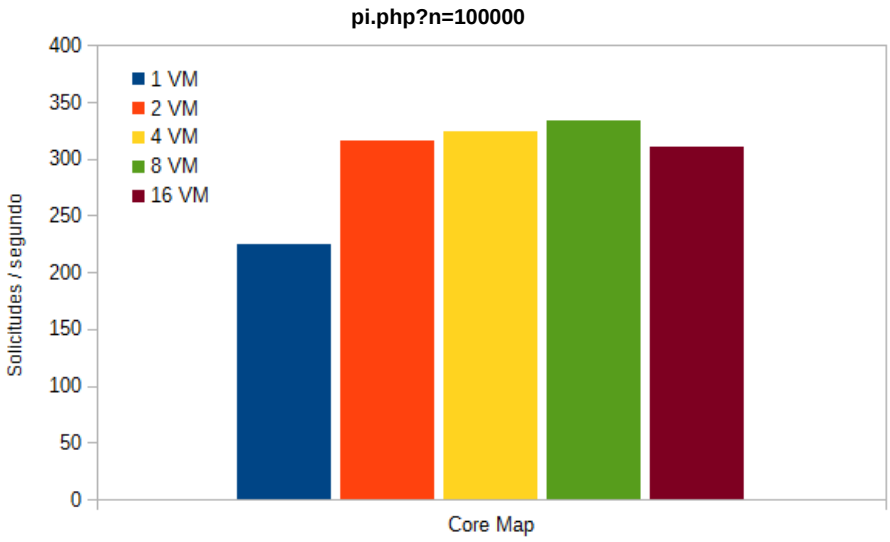


Figura 18. Prueba de carga utilizando asignación manual

En función de los resultados obtenidos, y teniendo en cuenta la mejora observada con 8 VMs para caso de páginas dinámicas, optaremos por las configuraciones de 4 y 8 VMs con asignación manual de los recursos, para las pruebas con el equilibrador de carga HAProxy.

4.3 Equilibrado de carga en los nodos directores: HAProxy

En esta sección se mostrará la configuración de HAProxy. También se hará una evaluación de sus prestaciones en diferentes escenarios, utilizando la disposición de 4 y 8 VMs sobre los 6 nodos de cómputo del clúster, con lo cual

obtendremos 24 y 48 VMs respectivamente para cada escenario. Se ha utilizado como base la configuración de HAProxy mostrada en el anexo A.15.

4.3.1. Configuración de HAProxy

El archivo de configuración de HAProxy está compuesto por diversos bloques que definen el funcionamiento del equilibrador de carga. El primer bloque *"global"* especifica los parámetros globales relacionados con la ejecución de la aplicación sobre el sistema operativo. Por ejemplo, como el fichero *.sock*, utilizado para la comunicación de HAProxy con otros procesos o recursos de red. La directiva log permite especificar el destino de los registros. También es posible definir el número máximo de conexiones concurrentes que recibirá el equilibrador de carga con la directiva *maxconn*, con el fin de proteger a los servidores del *backend* de posibles ataques de denegación de servicio.

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/HAProxy
        stats socket /var/run/HAProxy241.sock mode 660 level admin
        stats timeout 2m
        maxconn 1000
        user HAProxy
        group HAProxy
        daemon
..
```

El bloque *"defaults"*, permite configurar los parámetros por defecto a utilizar por la aplicación. Por ejemplo, la opción "*mode"* permite definir el protocolo, que en caso de ser http, permitirá analizar la solicitud del cliente a nivel de aplicación, usando funciones especiales provistas por HAProxy para el tratamiento de los datos. En caso de ser tcp, se reenvía la petición del cliente al servidor, pero sin examinar el contenido de la petición. Por lo tanto, no podrá utilizarse información del nivel de aplicación para conseguir la persistencia de sesión o realizar el reparto de la carga. Los valores relacionados con el tiempo, como los *timeouts* se expresan en ms, a no ser que se indique explícitamente otra magnitud.

```
..
defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
..
```

El bloque *“frontend”* permite definir la dirección IP del equilibrador de carga con su respectivo puerto asociado, así como diversos parámetros relacionados con el tratamiento de las solicitudes entrantes. Pueden definirse varias direcciones IP y/o puertos en caso de ser necesario. En el bloque *“backend”* se definen los nodos servidores encargados de responder las peticiones del cliente, siguiendo una política de reparto de carga definida por la directiva “*balance”*. Es importante destacar que el estado de los nodos servidores puede ser monitorizado, a través de la opción *check port*. Los servidores que no respondan a la monitorización no recibirán peticiones de los clientes.

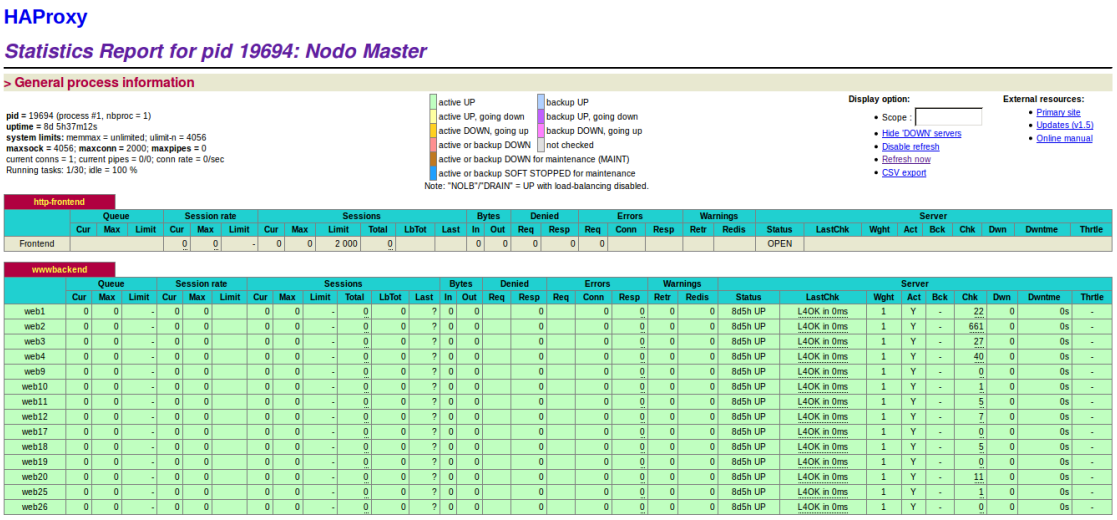
```
..
frontend http-frontend
    option httplog
    mode http
    bind 192.168.250.121:8081
    default_backend wwwbackend

backend wwwbackend
    mode http
    balance roundrobin
    server web1 192.168.250.181:80 check port 80
..
```

HAProxy proporciona una interfaz web que muestra el registro detallado del tráfico que atraviesa el equilibrador de carga, así como las solicitudes servidas por los nodos servidores y el estado del servicio. La directiva *“stats enable”* activa este servicio. Los restantes parámetros mostrados definen las características del mismo. Por ejemplo, *“stats hide-version”* elimina la información sobre la versión de HAProxy del informe de estadísticas. Se recomienda para sitios públicos o con contraseñas de acceso débiles.

```
..
listen stats
        mode http
        stats enable
```

```
        bind :9000
        stats uri /haproxy?stats
        stats show-desc Nodo Master
        stats hide-version
        stats refresh 5s
        stats realm HAProxy\ Statistics
        stats auth admin:admin
```

En la Figura 19 se observar la interfaz web con el registro de estadísticas, donde la dirección URL del servicio se encuentra definida por la directiva *"stats uri"* y el puerto asociado a través de la directiva *"bind"*, que para la configuración mostrada es http://localhost:9000/haproxy?stats.



HAProxy

Statistics Report for pid 19694: Nodo Master

> General process information

pid = 19694 (process #1, nbproc = 1)
uptime = 8d 5h37m12s
system limits: memmax = unlimited; ulimit-n = 4056
maxsock = 4056; maxconn = 2000; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 0/sec
Running tasks: 1/30; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Disable refresh
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

http-frontend

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	2 000	0			0	0	0	0	0					OPEN								

wwwbackend

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
web1	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	22	0	0s	-
web2	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	661	0	0s	-
web3	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	27	0	0s	-
web4	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	40	0	0s	-
web9	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	0	0	0s	-
web10	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	1	0	0s	-
web11	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	5	0	0s	-
web12	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	7	0	0s	-
web17	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	0	0	0s	-
web18	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	5	0	0s	-
web19	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	0	0	0s	-
web20	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	11	0	0s	-
web25	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	1	0	0s	-
web26	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	8d5h UP	L4OK in 0ms	1	Y	-	0	0	0s	-

Figura 19. Informe de estadísticas de HAProxy

4.3.2. Evaluación: ejecución de un proceso HAProxy

Como primer paso intentaremos determinar las prestaciones de HAProxy ejecutando un solo proceso HAProxy y considerando un *backend* de 24 VMs. Compararemos los resultados obtenidos en dos situaciones: dejando al planificador del sistema operativo ubicar el proceso y ubicándolo manualmente.

Es importante resaltar que al elegir la CPU donde ubicar el proceso HAProxy, no deben provocarse interferencias con otros procesos en ejecución. En particular, hay que evitar la CPU que se ocupa del procesamiento de las

interrupciones de red. Sin embargo, para aprovechar la afinidad NUMA, ambas deberían estar dentro del mismo dominio NUMA.

Por lo tanto, antes de ubicar el proceso HAProxy, se debe conocer la CPU que está recibiendo las interrupciones de red. Para ello, primero se debe determinar el número de la interrupción asociada al controlador de Ethernet, ejecutando la siguiente orden, y considerando que en nuestro caso el identificador de la tarjeta de red es *eth0*.

```
grep eth0 /proc/interrupts

32:   0     140      45       850264      PCI-MSI-edge      eth0
```

El primer valor de la respuesta representa el número de la interrupción, que servirá para localizar la CPU que recibe las interrupciones de red, a través de la siguiente orden.

```
cat /proc/irq/32/smp_affinity_list

6
```

En nuestro caso, considerando que las interrupciones de red ingresan por el núcleo 6, se ha ubicado el proceso HAProxy en el núcleo 5, dentro del mismo dominio NUMA (Ver Anexo A.1), a través de la siguiente orden.

```
taskset -cp 5 pid_proceso
```

Para las pruebas se ha utilizado la herramienta weighttp (Ver anexo A.8), con 128 peticiones concurrentes y un total de 100.000 solicitudes de la página index.php (Ver anexo A.10).

```
weighttp -n 100000 -c 128 192.168.250.121:8081/
```

En la Figura 20 podemos observar que la ejecución de HAProxy sobre un núcleo específico dentro del mismo dominio NUMA de las interrupciones de red, obtiene mayores prestaciones que la planificación utilizada por el sistema operativo, alcanzando alrededor de 10.056 peticiones por segundo, mientras que la del sistema operativo se queda en 8.659.

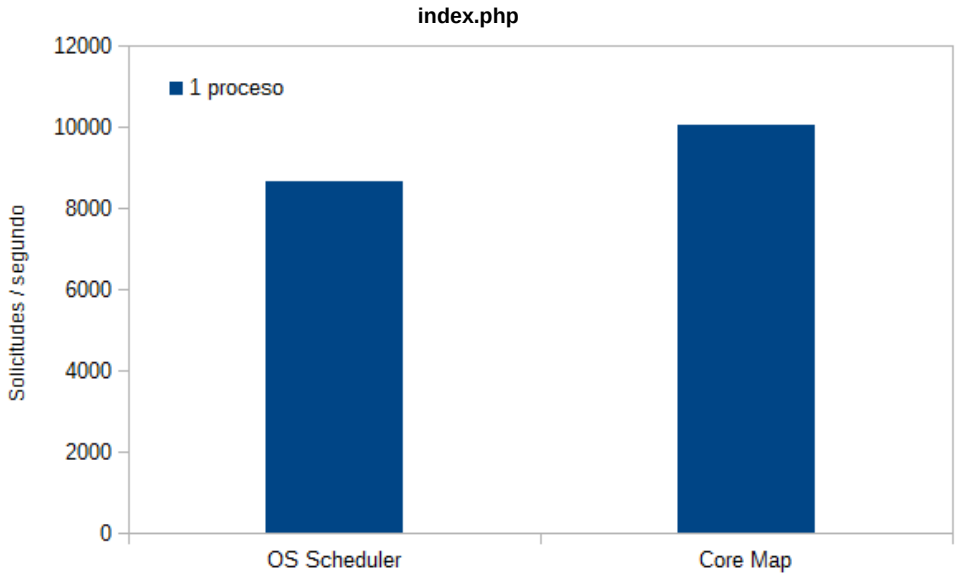


Figura 20. Ejecución de un proceso HAProxy con planificador OS vs asignación manual (index.php)

A pesar de que el planificador del sistema operativo ubicaba el proceso HAProxy sobre el mismo dominio NUMA, en muchas ocasiones se podía observar que el proceso interfería con las interrupciones de red, generando una disminución de las prestaciones.

Otra dato que se observa cuando el sistema operativo realiza la asignación es que las prestaciones son variables en función de los aciertos conseguidos en la elección de la CPU, durante las sucesivas planificaciones.

Se ha realizado la misma prueba pero ahora añadiendo más carga de trabajo al *backend*, haciendo uso de la herramienta en php que realiza el cálculo de pi con 100.000 iteraciones (pi.php, ver anexo A.11).

```
weighttp -n 100000 -c 128 192.168.250.121:8081/pi.php?n=100000
```

La Figura 21 muestra los resultados, donde la diferencia entre prestaciones es de tan solo 140 peticiones por segundo; lo cual se debe a que el retardo principal lo introducen los nodos del *backend*, debido a la mayor carga de procesamiento de la página solicitada. Por lo tanto, la interferencia entre los procesos HAProxy y las interrupciones de red no es apreciable debido a que el equilibrador de carga no se encuentra saturado.

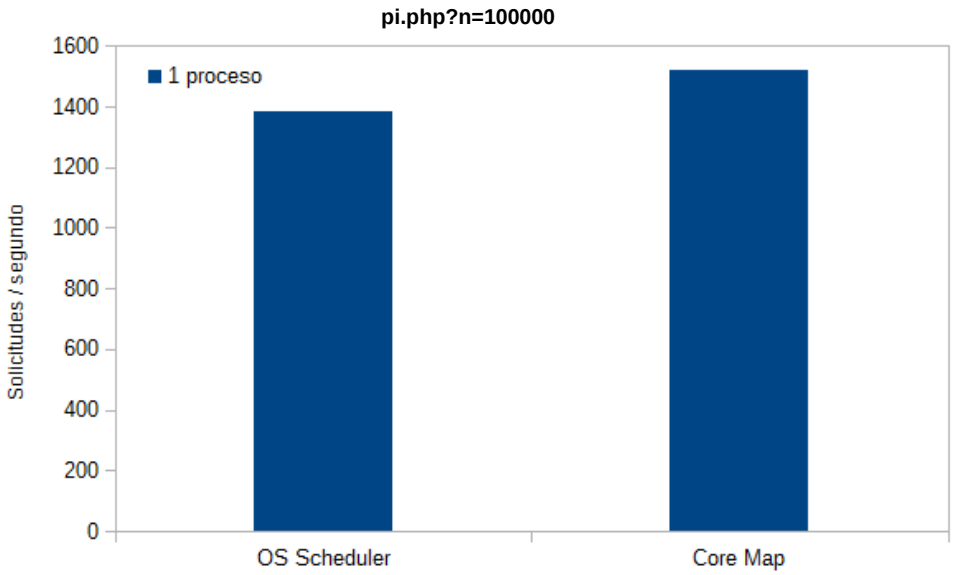


Figura 21. Ejecución de un proceso HAProxy con planificador OS vs asignación manual (pi.php)

Queda muy claro que la planificación manual e inteligente de los procesos HAProxy puede ayudar a obtener mejores prestaciones en el equilibrador de carga, por lo que en las siguientes pruebas haremos uso de una planificación manual de los procesos HAProxy.

Un análisis de la utilización de la CPU por parte del proceso HAProxy reveló un elevado valor (cercano al 100%), indicando claramente que el repartidor de carga es el cuello de botella.

4.3.3. Evaluación: ejecución de varios procesos HAProxy

El siguiente paso sería intentar aumentar las prestaciones lanzando varios procesos HAProxy, lo cual se ha hecho sobre los escenarios de 24 y 48 VMs en el *backend*.

Se ha utilizado un archivo de configuración HAProxy distinto para cada proceso, partiendo del archivo base mostrado en el anexo A.15. Para cada archivo de configuración se ha cambiado el nombre del archivo *.sock* y el puerto TCP por donde escucha el *frontend*.

```
# Configuración HAProxy proceso 1 (24 VMs)
..
stats socket /var/run/HAProxy241.sock mode 660 level admin
..
bind 192.168.250.121:8081
..
```

```
# Configuración HAProxy proceso 2 (24 VMs)
..
stats socket /var/run/HAProxy242.sock mode 660 level admin
..
bind 192.168.250.121:8082
..
```

```
# Configuración HAProxy proceso 3 (24 VMs)
..
stats socket /var/run/HAProxy243.sock mode 660 level admin
..
bind 192.168.250.121:8083
..
```

```
# Configuración HAProxy proceso 4 (24 VMs)
..
stats socket /var/run/HAProxy244.sock mode 660 level admin
..
bind 192.168.250.121:8084
..
```

Para lanzar los procesos HAProxy, se ha utilizado la siguiente orden, especificando el archivo de configuración con la opción -f y el archivo que contendrá el identificador del proceso con la opción -p. Además, para iniciar el servicio como demonio del sistema, se ha utilizada la opción -D.

```
haproxy –f haproxy-conf.cfg -D -p /var/run/haproxy.pid
```

En esta prueba se han ejecutado varios procesos HAProxy (desde 1 hasta 4), considerando un núcleo distinto al de las interrupciones de red para no causar interferencias. Por lo tanto, los procesos se han ubicado manualmente en el mismo dominio NUMA donde se atienden las interrupciones de red, pero sin utilizar el núcleo 6, en nuestro caso, encargado de procesar dichas interrupciones.

En concreto se han utilizado los núcleos 4, 5 y 7, aprovechando así la afinidad L3. Únicamente, para el caso de 4 procesos HAProxy ha sido necesario ubicar

uno de ellos en un dominio distinto. La Figura 22 muestra la asignación realizada mediante las órdenes siguientes.

```
taskset -cp 4 pid_proceso_HAProxy1
taskset -cp 5 pid_proceso_HAProxy2
taskset -cp 7 pid_proceso_HAProxy3
taskset -cp 8 pid_proceso_HAProxy4
```

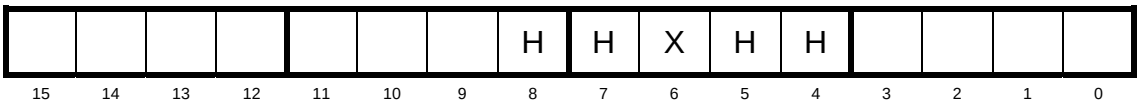


Figura 22. Esquema de pruebas con varios procesos HAProxy

Para generar la carga se ha utilizado la herramienta weighttp, lanzando las pruebas para que se ejecuten de manera simultánea.

```
# Generando carga sobre 4 procesos HAProxy

weighttp -n 100000 -c 128 192.168.250.121:8081/ > HAProxy1 &
weighttp -n 100000 -c 128 192.168.250.121:8082/ > HAProxy2 &
weighttp -n 100000 -c 128 192.168.250.121:8083/ > HAProxy3 &
weighttp -n 100000 -c 128 192.168.250.121:8084/ > HAProxy4
```

La Figura 23 muestra la comparativa entre varios procesos HAProxy solicitando una página sencilla en php (index.php, ver anexo A.10). Como puede observarse, se ha obtenido un aumento del 34% de las peticiones por segundo con tres procesos HAProxy, en comparación con un solo proceso. Un dato curioso es que las prestaciones disminuyen al utilizar 4 procesos HAProxy y no aumentan como era de esperar. Esto se debe a que el cuarto proceso no tiene afinidad con las interrupciones de red, ya que se encuentra en un dominio NUMA diferente.

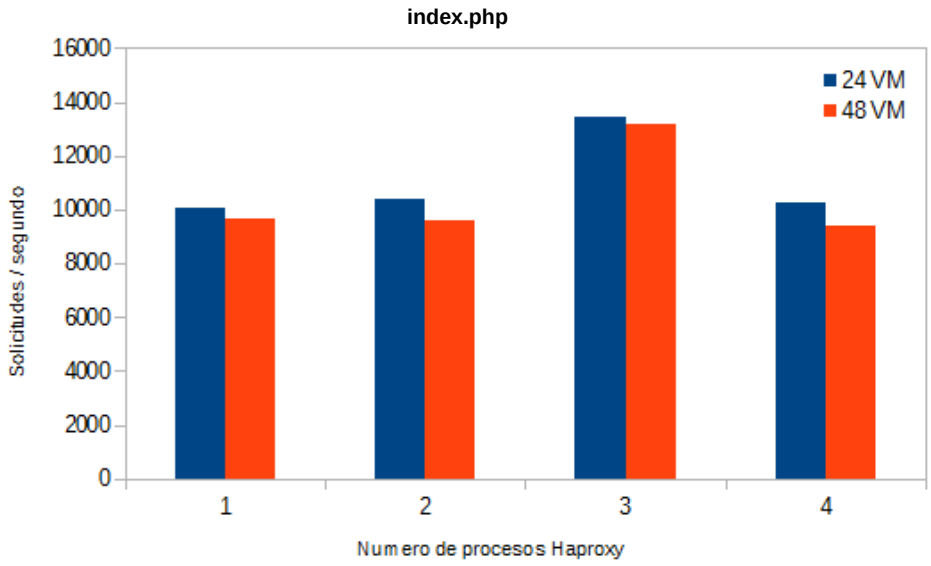


Figura 23. Ejecución de varios procesos HAProxy (index.php)

De igual manera se han hecho pruebas haciendo uso de la herramienta que calcula el valor de pi con 100.000 intervalos, lanzando 128 peticiones concurrentes (Ver anexo A.11). En la Figura 24 se puede observar que no existe ningún aumento de las peticiones por segundo utilizando varios procesos HAProxy, lo cual se debe a que el cuello de botella se encuentra en el procesamiento por parte de los servidores y no en el equilibrador de carga. También, se puede observar que las prestaciones disminuyen con un mayor número de VMs, lo cual es coherente con lo visto en la sección 4.2.2, donde los procesos *libvirt* que acompañan a las VMs en la administración de los recursos virtuales, restan recursos del sistema e interfieren con otras aplicaciones, como se ha comprobado en este caso.

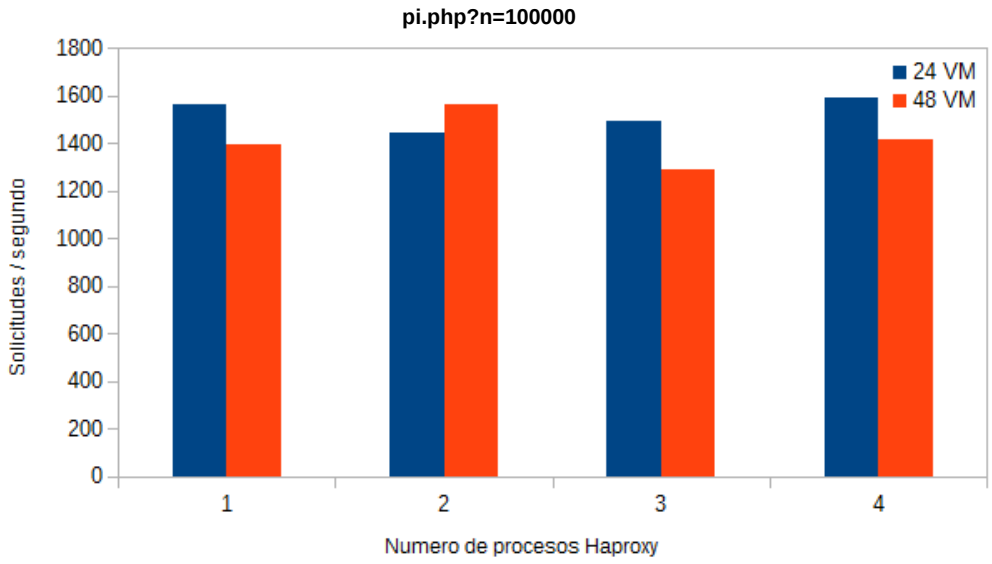


Figura 24. Ejecución de varios procesos HAProxy (pi.php)

El hecho de que, con la página sencilla, añadir más de 3 procesos HAProxy no mejore las prestaciones motivó que sospecháramos que el cuello de botella estaba en otro lugar. Un detallado análisis de la utilización del procesador, mostró que los procesos HAProxy no alcanzaban valores cercanos al 100% y, sin embargo, la CPU ocupada del procesamiento de los paquetes de red, mostraba una alta utilización. Este parecía ser el cuello de botella del sistema, por lo que se procedió a optimizar la gestión de las interrupciones de red.

4.4 Optimización en la gestión de las interrupciones de red

Es esta parte intentaremos aumentar las prestaciones de HAProxy, basándonos en optimizaciones en el procesamiento de recursos de red, haciendo uso de herramientas tales como RPS y RFS vistas en la sección 2.4. Debido a que las tarjetas de red utilizadas solo poseen una cola física, no podremos realizar pruebas usando RSS.

De acuerdo a las pruebas anteriores hemos obtenido un valor máximo de 13.467 peticiones por segundo con 3 procesos HAProxy para el escenario de 24 VMs sobre el *backend* con la página index.php. Sin embargo, trabajaremos en la optimización de las prestaciones de red valiéndonos de 4 procesos HAProxy, por dos motivos. En primer lugar, es de esperar que el mejorar el procesamiento de paquetes de red, genere más presión sobre los procesos HAProxy. En segundo lugar, 4 procesos HAProxy nos permitirá repartir los procesos e interrupciones de red de manera equitativa sobre los diferentes dominios NUMA.

4.4.1. Optimización mediante RPS

Para activar RPS se ejecuta la siguiente orden como usuario privilegiado, escribiendo el valor en hexadecimal que representa los núcleos a utilizar como RPS, donde el bit más significativo representa el núcleo con el identificador más alto y el bit menos significativo con el identificador más bajo.

```
echo RPS_mask > /sys/class/net/eth0/queues/rx-0/rps_cpus
```

En nuestro caso, hemos probado añadir 1 y 3 CPUs adicionales para la gestión de interrupciones de red. Para la selección de las CPUs más apropiadas, debemos tener en cuenta la ubicación de los procesos HAProxy en el sistema.

En esta parte del trabajo se proponen diferentes configuraciones, tomando en consideración la configuración inicial utilizada en la sección anterior que se mostró en la Figura 22.

Como punto de partida se ha considerado añadir 1 CPU adicional al procesamiento de paquetes de red (en el núcleo 2) y los núcleos 3, 4, 5 y 7 para el procesamiento de procesos HAProxy tal y como se muestra en la Figura 25.

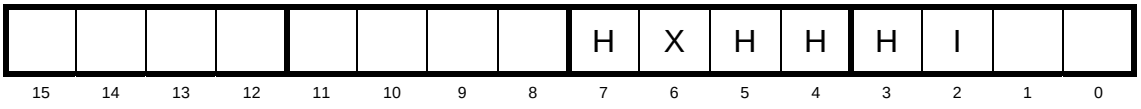


Figura 25. Esquema de pruebas con RPS (mask 0004)

La máscara RPS será 0004, donde 4 en binario (0100) representa la CPU número 2. Con esta configuración se emplean 2 dominios NUMA. En uno de ellos (CPU 4 al 7) se explota la afinidad entre tres procesos HAProxy y el que atiende las interrupciones de red.

En el otro dominio NUMA (CPU 0 al 3) también existe afinidad entre un proceso HAProxy y otro que colabora en la atención de interrupciones de red. Nótese, sin embargo, que no hay garantía de que al hacer el reparto de paquetes entre los dos procesos que gestionan interrupciones de red, los paquetes asignados a cada uno vayan destinados al proceso HAProxy que comparte el dominio NUMA.

Para la siguiente prueba se han destinado las CPUs 2, 10 y 14 para el procesamiento de las interrupciones de red, lo que se representa con la máscara RPS 4404. Para el procesamiento de procesos HAProxy se ha destinado las CPUs 3, 7, 11 y 15 como se muestra en la Figura 26.

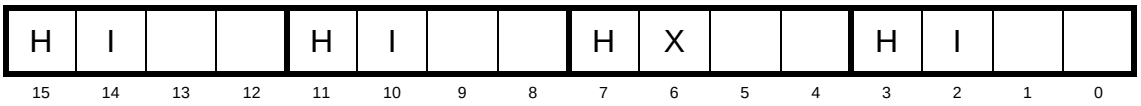


Figura 26. Esquema de pruebas con RPS (mask 4404)

En este caso se emplean 4 dominios NUMA. En cada dominio se ha ubicado un proceso HAProxy junto con otro que procesa interrupciones de red. Tal y como se ha indicado, no hay garantía de que los paquetes tratados por un proceso de atención de interrupciones, vayan destinados al proceso HAProxy con el que comparte dominio NUMA.

Como siguiente prueba, se han destinado las CPUs 4, 5 y 7 para el procesamiento de paquetes de red, lo que se representa con la máscara RPS 00b0. Además, se ha utilizado las CPUs 8, 9, 10 y 11 para procesos HAProxy como se observa en la Figura 27.

Figura 27. Esquema de pruebas con RPS (mask 00b0)

En este caso se emplean 2 dominios NUMA. En uno de ellos (CPU 4 al 7) se explota la afinidad entre los procesos que atienden interrupciones de red. Recuérdese que las atiende inicialmente la CPU 6, y que reparte el trabajo a las CPUs 4, 5 y 7.

La última prueba considerada, consistió en destinar las CPUs 2, 3 y 7 al procesamiento de paquetes de red, lo que se representa con la máscara RPS 008c. Además, se han utilizado las CPUs 0, 1, 4 y 5 para procesos HAProxy como se observa en la Figura 28.

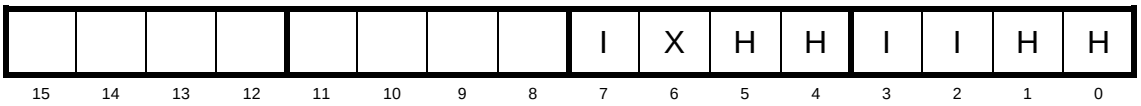


Figura 28. Esquema de pruebas con RPS (mask 008c)

En este caso se emplean dos dominios NUMA. En cada dominio NUMA se han ubicado dos procesos que atienden interrupciones de red, más otros dos procesos HAProxy. Se debe volver aclarar que no hay garantía de que los paquetes de red tratados, vayan destinados a los procesos HAProxy que comparten dominio NUMA.

La Tabla III muestra un resumen de las 4 configuraciones evaluadas.

RPS mask	**CPUs Interrupciones de red adicionales**	**CPUs HAProxy**
0004	2	3, 4, 5, 7
4404	2, 10, 14	3, 7, 11, 15
00b0	4, 5, 7	8, 9, 10, 11
008c	2, 3, 7	0, 1, 4, 5

Tabla III. Detalles de las pruebas con RPS

Así mismo, se ha utilizado la herramienta weighttp para generar las peticiones sobre cada uno de los procesos HAProxy de manera simultánea.

```
 # Generando carga sobre 4 procesos HAProxy

weighttp -n 100000 -c 128 192.168.250.121:8081/ > RPS1 &
weighttp -n 100000 -c 128 192.168.250.121:8082/ > RPS2 &
weighttp -n 100000 -c 128 192.168.250.121:8083/ > RPS3 &
weighttp -n 100000 -c 128 192.168.250.121:8084/ > RPS4
```

En la Figura 29 se puede observar los resultados solicitando la página index.php (Ver anexo A.10). La configuración RPS 008c obtiene prestaciones muy por encima de las demás configuraciones, llegando a un máximo de 23.093 peticiones por segundo, lo cual es un 71,5% mejor que lo máximo alcanzado en las pruebas sin optimizaciones de red.

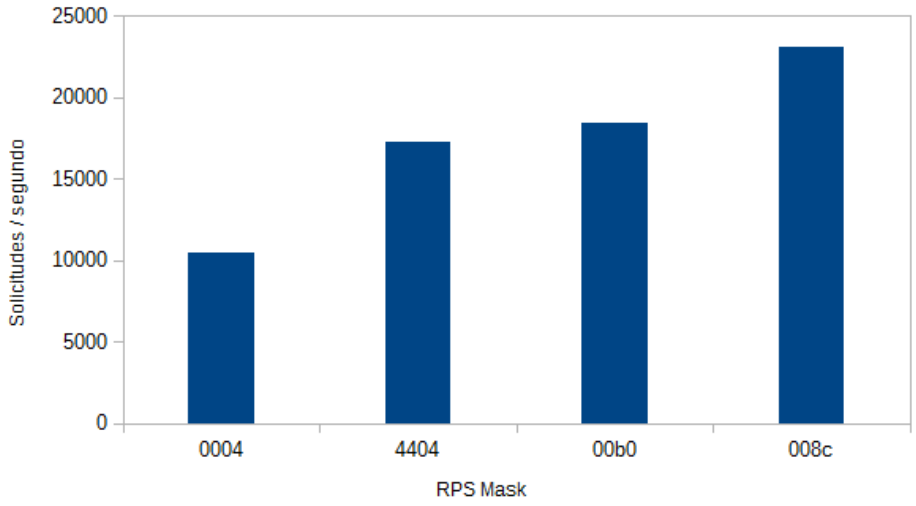


Figura 29. Ejecución de 4 procesos HAProxy con RPS

Si analizamos la configuración RPS 008c que obtiene el mejor resultado, nos podemos dar cuenta que el reparto de procesos HAProxy e interrupciones de red sobre los núcleos se realiza de manera equitativa, teniendo en cada dominio NUMA 2 procesos HAProxy y 2 interrupciones de red.

Es importante destacar la configuración RPS 00b0, ya que se han asignado todas las interrupciones de red dentro de un mismo dominio NUMA, lo cual destaca el concepto de afinidad, dado que las interrupciones de red se sirven del mismo dominio, permitiendo así obtener alrededor de 18.446 peticiones por segundo, que es la segunda mejor opción.

4.4.2. Optimización mediante RFS

El siguiente paso es hacer uso de la técnica RFS, que como se ha visto en la sección 2.4, nos permitirá optimizar los recursos de red de manera similar a RPS, pero con la diferencia de que RFS intenta aumentar la eficiencia de aciertos en cache, enviando los paquetes de red hacia la CPU que contiene la aplicación que hará uso de tales paquetes.

Para poder trabajar con RFS será necesario definir el número de conexiones concurrentes activas que se espera recibir en nuestro servidor, en el fichero rps_sock_flow_entries (32768). Además, también hay que indicar el número de posibles procesos consumidores, en el fichero rps_flow_cnt, por lo que para nuestro caso se divide el valor de 32768 entre 4, lo cual da como resultado 8192.

```
echo 32768 > /proc/sys/net/core/rps_sock_flow_entries
echo 8192 > /sys/class/net/eth0/queues/rx-0/rps_flow_cnt
```

Para las pruebas se ha seguido el mismo esquema de asignación de procesos HAProxy utilizado en RPS, el cual se puede observar en la Tabla IV.

RFS Test	HAProxy (Núcleo)
RFS-1	8, 9, 10, 11
RFS-2	3, 4, 5, 7
RFS-3	0, 1, 4, 5
RFS-4	3, 7, 11, 15

Tabla IV. Detalles de las pruebas con RFS

Al igual que en las pruebas con RPS, se ha utilizado la herramienta weighttp para generar las peticiones sobre cada uno de los procesos HAProxy de manera simultánea. Como en la evaluación anterior la página solicitada también ha sido index.php (Ver anexo A.10).

```
# Generando carga sobre 4 procesos HAProxy

weighttp -n 100000 -c 128 192.168.250.121:8081/ > RFS1 &
weighttp -n 100000 -c 128 192.168.250.121:8082/ > RFS2 &
weighttp -n 100000 -c 128 192.168.250.121:8083/ > RFS3 &
weighttp -n 100000 -c 128 192.168.250.121:8084/ > RFS4
```

Los resultados se pueden ver en la Figura 30, donde el máximo valor alcanzado se da en la prueba RFS-1 con un valor de 12.480 peticiones por segundo, lo cual no representa mejora en comparación con las pruebas sin optimizaciones de red y tres procesos HAProxy. Sin embargo, si representa mejora respecto a la configuración básica de 4 procesos HAProxy (que alcazaba alrededor de 10.000 peticiones por segundo, ver Figura 23). Es importante destacar que la configuración RFS-1 con mejor resultado, tiene todos los procesos HAProxy en el mismo dominio NUMA.

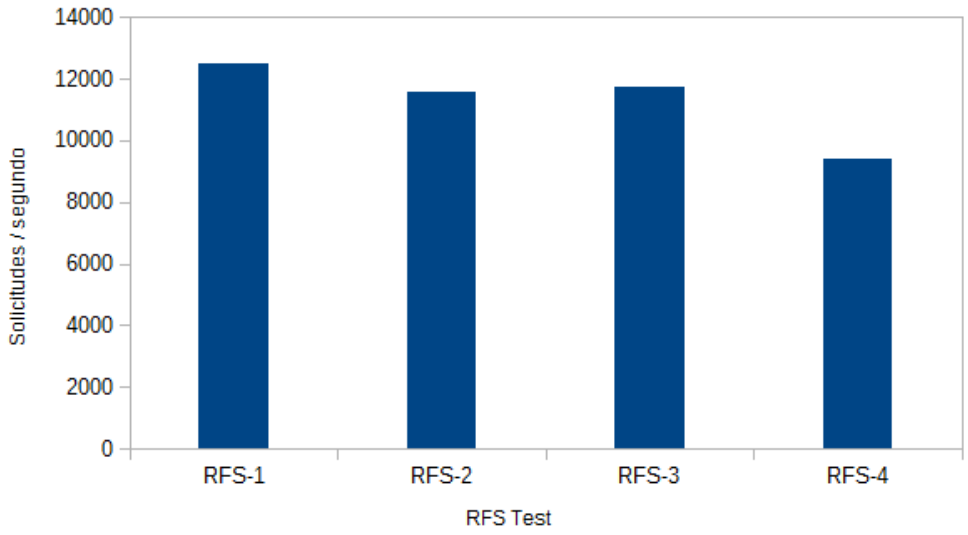


Figura 30. Ejecución de 4 procesos HAProxy con RFS

Analizando con más detalle la ejecución de las pruebas usando RFS, se pudo observar que cada núcleo que ejecuta un proceso HAProxy, gestiona de manera adicional interrupciones de red, lo cual va de acuerdo con la característica principal de RFS, que es direccionar los paquetes de red hacia los procesos que harán uso de los mismos. Sin embargo, a pesar de que esto mejora la afinidad entre los procesos, resta potencia de cómputo al proceso principal, degradando en cierto modo las prestaciones globales del equilibrador de carga.

4.5 Alta disponibilidad en los nodos directores: *keepalived*

Para dotar al sistema de alta disponibilidad en cuanto a los repartidores de carga, haremos uso de la herramienta *keepalived* descrita en la sección 2.3. La alta disponibilidad de los nodos servidores es gestionada directamente por el propio HAProxy, tal como se indicó en la sección 2.2.1. Como ya hemos visto, será necesario definir una o varias direcciones de red VIP, que serán usadas por *keepalived* para asignarlas al sistema activo que se encuentre prestando el servicio. En nuestro caso se ha utilizado una dirección VIP, específicamente la dirección 192.168.250.250. Recordemos que en los apartados anteriores al lanzar varios procesos HAProxy, los hemos diferenciado por su número de puerto.

Para poder asociar la dirección VIP con cada proceso HAProxy, será necesario editar el archivo base de la configuración HAProxy asociada a ese proceso (Ver anexo A.15), añadiendo el conjunto de equilibradores de carga (en nuestro caso dos, MASTER y BACKUP) con sus respectivas IP físicas y especificar la VIP con el puerto asociado en el *frontend*. La directiva *"peers"* utilizada en la configuración, permite que ambos equilibradores de carga sincronicen sus entradas, es decir que puedan mantener las conexiones establecidas en caso de algún fallo en el nodo maestro.

```
# Configuración HAProxy con VIP y servidor replicado

peers LB
    peer lb1 192.168.250.121:1234
    peer lb2 192.168.250.111:1234
..
frontend http-frontend
    option httplog
    mode http
    bind 192.168.250.250:8081
    default_backend wwwbackend
..
```

Ahora solo queda definir la configuración *keepalived* de los equilibradores de carga, donde especificaremos el rol de cada uno, su prioridad, las características de la autenticación, la dirección VIP y el servicio a monitorizar. En nuestro caso,

se ha decidido comprobar periódicamente el estado de la interfaz de red eth0, aunque es posible realizar una monitorización más sofisticada, por ejemplo, se puede monitorizar la salud del proceso HAProxy.

```
# Configuración keepalived MASTER

vrrp_instance HA1 {
        state MASTER
        interface eth0
        virtual_router_id 11
        priority 150
        advert_int 1
        authentication {
                auth_type PASS
                auth_pass $ cac
        }

        virtual_ipaddress {
                192.168.250.250
        }

        track_interface {
        eth0
      }
}
```

```
# Configuración keepalived BACKUP

vrrp_instance HA1 {
        state BACKUP
        interface eth0
        virtual_router_id 11
        priority 100
        advert_int 1
        authentication {
                auth_type PASS
                auth_pass $ cac
        }

        virtual_ipaddress {
                192.168.250.250
        }

        track_interface {
        eth0
      }
}
```

Una vez configurado el servicio *keepalived*, se procede a iniciarlo en cada uno de los equilibradores de carga y a probar su funcionamiento, comprobando que el nodo con rol MASTER tenga asignada la VIP.

```
   $ ip addr sh eth0

   2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
       link/ether 00:30:48:5f:f1:ea brd ff:ff:ff:ff:ff:ff
       inet 192.168.250.121/24 brd 192.168.250.255 scope global eth0
          valid_lft forever preferred_lft forever
       inet 192.168.250.250/32 scope global eth0
          valid_lft forever preferred_lft forever
```

Como podemos observar, aparece configurada la dirección VIP 192.168.250.250. Para verificar que la VIP es asignada al BACKUP cuando ocurre algún fallo en el nodo principal, se procede a desactivar el servicio *keepalived* en el MASTER. A continuación, comprobamos que la VIP está servida en la interfaz eth0 del BACKUP o también comprobando el siguiente aviso en el registro del sistema:

```
   $ tail /var/log/syslog

   Keepalived_vrrp[6891]: VRRP_Instance(HA1) Transition to MASTER STATE
   Keepalived_vrrp[6891]: VRRP_Instance(HA1) Entering MASTER STATE
   avahi-daemon[933]: Registering new address record for 192.168.250.250 on
eth0.IPv4.
```

Capítulo 5

Conclusiones

En este capítulo se expone un resumen de los objetivos alcanzados en el desarrollo del trabajo.

Se logró implementar un repartidor de carga de alta disponibilidad que ofrece servicios web. El repartidor se compone de 2 nodos repartidores y 6 nodos servidores. Los nodos repartidores decidieron implementarse directamente sobre el hardware para alcanzar elevadas prestaciones, mientras que los nodos servidores se implementaron como máquinas virtuales para aprovechar su flexibilidad en la configuración y despliegue.

En cuanto a las plataformas de virtualización, se eligieron las opciones más populares del mercado, disponibles de forma gratuita: KVM, XEN y Virtual Box. Se realizó un análisis comparativo de prestaciones entre ellas. Para los *benchmark* de CPU, RAM y disco, se obtuvo un ligero menor rendimiento por parte de la plataforma XEN y el resultado de VirtualBox fue de un 65,8% inferior a las demás plataformas, en la ejecución de servicios web. Por lo tanto, se desplegó KVM sobre los nodos servidores del clúster debido a la flexibilidad en la instalación y las prestaciones alcanzadas en las diferentes pruebas.

En segundo lugar, una vez elegida la plataforma KVM, se determinó la configuración idónea de los nodos servidores en cuanto a dos parámetros. Por una parte, se determinó el número idóneo de máquinas virtuales que se podían ejecutar en cada uno de los nodos físicos. Por otra parte, se compararon las prestaciones obtenidas asignando manualmente los núcleos del sistema a las

máquinas virtuales, con el resultado de la asignación hecha por el sistema operativo. En todo momento las prestaciones de la asignación manual fueron superiores en comparación con la planificación realizada por el sistema operativo. En cuanto al número de máquinas virtuales, las configuraciones de 4 y 8 máquinas virtuales por nodo físico resultaron ser las mejores.

En tercer lugar, se realizó la comparativa de los servidores web Apache, Nginx y Gwan. Las prestaciones obtenidas por los tres servidores fueron similares al momento de solicitar una página web estática. Sin embargo, Gwan obtuvo un bajo rendimiento en la solicitud de páginas web dinámicas usando php. Los resultados conseguidos por los otros dos servidores fueron similares, con una ligera desventaja por parte del servidor Nginx. Por este motivo, unido a que es el servidor web más consolidado y con mayor uso de los tres, se utilizó Apache sobre los nodos del clúster.

Se realizó la configuración de un servidor de alta disponibilidad con equilibrado de carga, haciendo uso de las herramientas *keepalived* y HAProxy, alcanzando prestaciones de 13.467 peticiones por segundo, haciendo uso de 3 procesos HAProxy distribuidos sobre un mismo dominio NUMA y sin optimizaciones en la gestión de interrupciones de red.

Se aumentaron las prestaciones hasta 23.093 peticiones por segundo (71,5% de mejora) utilizando la técnica RPS, que permite gestionar de manera más eficiente las interrupciones de red, aplicando una distribución equilibrada de procesos HAProxy e interrupciones de red sobre los distintos dominios NUMA.

Bibliografía

[1] Cisco VNI: Forecast and Methodology, 2015–2020. Disponible en: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html

[2] VMware, “Understanding full virtualization, paravirtualization, and hardware assist” VMware White Paper, 2007. Disponible en: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[3] IEEE Computer Society, “802.1AX-2008 IEEE Standard for Local and metropolitan area networks--Link Aggregation” Disponible en: https://standards.ieee.org/findstds/standard/802.1AX-2008.html

[4] “Basic Understanding of EtherChannel”. Disponible en: https://ciscoskills.net/2012/01/04/basic-understanding-of-etherchannel/

[5] RedHat, “A Basic Keepalived Load Balancer Configuration”. Disponible en: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Load_Balancer_Administration/ch-keepalived-overview-VSA.html#s1-lvs-basic-VSA

[6] HAProxy. http://www.haproxy.org/

[7] They use it. Disponible en: http://www.haproxy.org/they-use-it.html

[8] How HAProxy Works. Disponible: http://cbonte.github.io/haproxy-dconv/intro-1.7.html

[9] Keepalived. Disponible en: http://www.keepalived.org/

[10] IBM, Cache and memory affinity optimizations. Disponible en: https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.optimize/dso_cache_mem_aff.htm

[11] RedHat, Overview of Packet Reception. Disponible en: https://access.redhat.com/documentation/en-

US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-network-packet-reception.html

[12] Scaling in the Linux Networking Stack. Disponible en: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[13] Virtual Box. Disponible en: https://www.virtualbox.org/

[14] XEN. Disponible en: http://www.xenproject.org/

[15] KVM. Disponible en: http://www.linux-kvm.org/page/Main_Page

[16] vSphere. Disponible: https://www.vmware.com/es/products/vsphere

[17] Hyper-V. Disponible en: https://msdn.microsoft.com/es-es/library/hh831531(v=ws.11).aspx

[18] Ferran Serafini. Arquitectura Xen, 2013. Disponible en: https://www.josemariagonzalez.es/2013/02/13/arquitectura-xen.html

[19] Sergio Talens-Oliag. Herramientas de virtualización libres para sistemas GNU/Linux. Congreso Internet del Mediterráneo, 2010. Disponible en: http://www.uv.es/sto/charlas/2010_CIM/hvl-cim-2010-slides.pdf

[20] BYTEmark benchmark. http://www.tux.org/~mayer/linux/bmark.html

[21] RAMspeed. http://alasir.com/software/ramspeed/

[22] Bonnie++. http://www.coker.com.au/bonnie++/

[23] Apache HTTP server. https://httpd.apache.org/

[24] Nginx. https://www.nginx.com/

[25] Gwan. http://gwan.com/

[26] Usage of web servers for websites. Disponible en: http://w3techs.com/technologies/overview/web_server/all

[27] Kernel Linux. https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.13.1.tar.xz

[28] ApacheBench & HTTPerf. Disponible en: http://gwan.com/en_apachebench_httperf.html

[29] Weighttp. http://cgit.lighttpd.net/weighttp.git/snapshot/weighttp-master.tar.gz

Anexo A

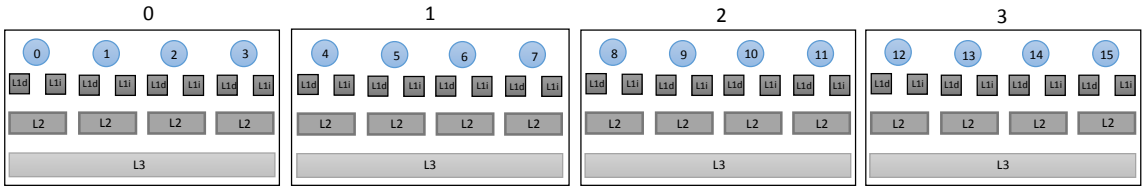
A.1 Arquitectura de los nodos del clúster

Ejecutando la siguiente instrucción *lscpu* se puede conocer la microarquitectura del procesador. En la columna *Node* podemos observar la cantidad de dominios NUMA del procesador (4 en este caso), el cual indica que son del 0 al 3 para el procesador AMD Opteron(tm) 8350. Adicional a esto podemos observar que las memorias L1 y L2 son privadas para cada uno de los núcleos, sin embargo la memoria L3 es compartida por los núcleos de los diferentes dominios NUMA.

lscpu –p
CPU,Core,Socket,Node,,L1d,L1i,L2,L3 0,0,0,0,,0,0,0,0 1,1,0,0,,1,1,1,0 2,2,0,0,,2,2,2,0 3,3,0,0,,3,3,3,0 4,4,1,1,,4,4,4,1 5,5,1,1,,5,5,5,1 6,6,1,1,,6,6,6,1 7,7,1,1,,7,7,7,1 8,8,2,2,,8,8,8,2 9,9,2,2,,9,9,9,2 10,10,2,2,,10,10,10,2 11,11,2,2,,11,11,11,2 12,12,3,3,,12,12,12,3 13,13,3,3,,13,13,13,3

```
14,14,3,3,,14,14,14,3
15,15,3,3,,15,15,15,3
```

En el siguiente gráfico se puede observar la distribución de memoria caché y núcleos sobre los diferentes dominios NUMA. Es importante volver a destacar que la memoria L3 es compartida por los núcleos de los diferentes dominios NUMA.



A.2 Compilación del kernel Linux

Para la compilación del Kernel Linux 3.13.1, será necesario descargar el código fuente disponible en la web oficial [27]. Adicionalmente, es necesario tener instalado algunos paquetes adicionales.

```
apt-get install make gcc ncurses-devel ncurses tar
```

Con el código fuente descargado, se accede a la carpeta descomprimida del kernel y se ejecuta la siguiente orden para generar el archivo de configuración del kernel, el cual especifica los módulos a incluir en la compilación, que para nuestro caso usaremos los parámetros por defecto.

```
make menuconfig
```

Con el archivo de configuración listo, se procede a ejecutar la siguiente orden con el fin de realizar la compilación utilizando varios hilos de ejecución y muestre el tiempo que ha tardado en la ejecución.

```
time sh –c “make –j 5”
```

A.3 *Benchmark* de CPU

Para someter a prueba la CPU, se ha utilizado la herramienta Bytemark la cual se ha obtenido de la web oficial [20]. Accediendo a la carpeta de la aplicación se ejecuta la orden make para compilarla. Una vez compilada se ejecuta la siguiente orden para iniciar el *benchmark*, otorgando una mayor prioridad al proceso, usando la instrucción nice.

```
nice -n -20 time ./nbench
```

A.4 *Benchmark* de memoria RAM

Para realizar el *benchmark* de memoria RAM, se ha utilizado la herramienta RAMspeed, la cual se ha obtenido de la web oficial [21]. Accediendo a la carpeta de la aplicación se ejecutan las siguientes órdenes, para poder hacer uso de los algoritmos implementados por la herramienta, que utilizan las unidades ALU, FPU, MMX y SSE.

```
./ramsmp -b 3
./ramsmp -b 6
./ramsmp -b 9
./ramsmp -b 12
```

A.5 *Benchmark* de disco

Para realizar el *benchmark* de disco, se ha utilizado la herramienta Bonnie++, disponible para su instalación desde el repositorio de Ubuntu.

```
apt-get install bonnie++
```

Para obtener la velocidad de lectura y escritura se ha utilizado la siguiente orden, donde se especifica el directorio a utilizar con la opción -d, el valor de memoria RAM disponible con la opción -r y el usuario seleccionado para ejecutar la aplicación con la opción -u.

```
bonnie++ -d /tmp -r 2048 -u root
```

A.6 *Benchmark* de Red

Para las pruebas de red se ha utilizado la orden ping, con un tiempo de espera de 0,1 segundos para cada envío de paquetes y con un total de 60 peticiones ICMP.

```
ping ip_servidor -i 0.1 -c 60
```

A.7 Servidores web: Apache *benchmark*

Para obtener las prestaciones en relación al número de solicitudes por segundo servidas por cada una de las plataformas virtuales, se ha hecho uso de la herramienta apache *benchmark*, la cual se puede descargar desde el repositorio de Ubuntu.

```
apt-get install apache2-utils
```

Esta herramienta realiza un número de peticiones sobre el servidor web, simulando la presencia de varios usuarios simultáneos. Para iniciar las pruebas se ejecuta la siguiente orden, la cual especifica el número de peticiones concurrentes con la opción -c y el número total de peticiones al servidor web, con la opción -n.

```
ab -n 100000 -c 1 ip_servidor/
```

Tras ejecutar el *benchmark*, se ofrecen los resultados, siendo los más importantes el número de peticiones por segundo alcanzadas y el tiempo medio de respuesta por petición.

A.8 Weighttp

A partir del segundo escenario descrito en el capítulo 4, haremos uso de otra herramienta de *benchmark* para servidores web, weighttp, ya que dentro del entorno de pruebas el cliente también se puede convertir en el cuello de botella [28], por lo que aprovecharemos la opción multithreading que incorpora weighttp para trabajar con varios núcleos en el cliente. Esta posibilidad no está implementada en el apache *benchmark*.

Para instalar la aplicación, primero descargamos el paquete [29]. Una vez descomprimido, accedemos a la carpeta y ejecutamos las siguientes órdenes que permitirán la compilación e instalación de la herramienta.

```
./waf configure
./waf build
./waf install
```

Para ejecutar el *benchmark* hacemos uso de la siguiente orden, donde la opción -n especifica el número total de peticiones hacia el servidor, la opción -c el número de peticiones concurrentes y la opción -t la cantidad de hilos a utilizar.

```
weighttp -n 100000 -c 2 -t 2 ip_servidor/
```

A.9 Archivo index.html

```
<html>
<body>Pagina de prueba para CAC</body>
</html>
```

A.10 Archivo index.php

```
<html>
<body>
Pagina de prueba para CAC
<?php
echo "<br><br>";
echo "Hoy es " . date ("d/m/Y") . "<br>";
```

```
   echo "Son las " . date ("h:i:s") . "<br>";
   echo "Soy el servidor " . $_SERVER['SERVER_NAME'] . "IP " .
$_SERVER['SERVER_ADDR'] . "<br>";
   ?>
   </body>
```

A.11 Cálculo de pi en php (pi.php)

```
   <html>
   <body>
   <?php
   $start=microtime(true);

   $area=0.0;

   $n=$_GET["n"];
   // $n=100000000;

   for ($i=0; $i<$n; $i++)
           {
           $x=($i+0.5)/$n;
           $area=$area+4.0/(1.0+$x*$x);
           }
   $result=$area/$n;

   $end=microtime(true);
   $exectime=$end-$start;

   echo "<br>Calculo de PI<br><br>";
   printf ("La cte. PI con n= %d es igual a %f<br>", $n, $result);
   printf ("Tiempo de ejecucion= %.5f segundos<br>",$exectime);
   printf ("<br>El servidor es %s<br>", $_SERVER['SERVER_ADDR']);
   ?>
   </body>
   </html>
```

Al programa se le pasa como parámetro el número de intervalos para calcular la constante pi por integración numérica. Variando el número de intervalos se modifica el tiempo de ejecución. La tabla siguiente muestra el tiempo de ejecución y las peticiones por segundo en función del número de intervalos, para los servidores utilizados en este trabajo.

Intervalos	Tiempo de ejecución (ms)	Peticiones por segundo
100.000	50,017	19,99
1.000.000	505,188	1,98
10.000.000	5151,370	0,19

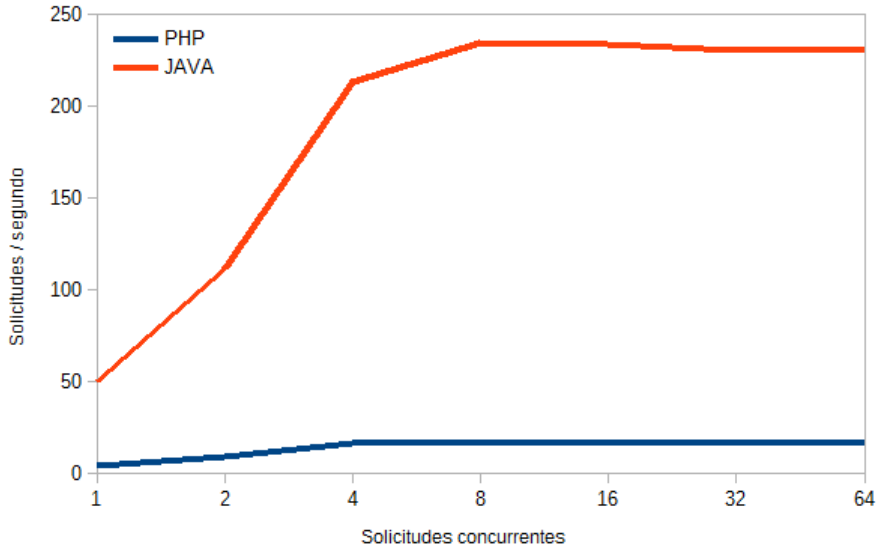
A.12 Cálculo de pi en java

```
   import java.io.IOException;
   import java.io.PrintWriter;
   import java.net.InetAddress;
   import java.sql.Date;
   import java.text.DateFormat;
   import java.text.SimpleDateFormat;
   import java.util.Calendar;

   import javax.servlet.ServletException;
   import javax.servlet.annotation.WebInitParam;
   import javax.servlet.annotation.WebServlet;
   import javax.servlet.http.HttpServlet;
   import javax.servlet.http.HttpServletRequest;
   import javax.servlet.http.HttpServletResponse;

   public class FirstServlet extends HttpServlet {
      private static final long serialVersionUID = 1L;
      public static final String HTML_START="<html><body>";
      public static final String HTML_END="</body></html>";
      double exec, start, end;

       public FirstServlet() {
           super();
           // TODO Auto-generated constructor stub
       }

      protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
             // TODO Auto-generated method stub

              double pi;
              DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd
HH:mm:ss");
              PrintWriter out = response.getWriter();

              if (request.getParameter("n") == null) {
                     out.println(HTML_START+"<br>Ingresar valor de n en la
URL"+HTML_END);
              } else {

                     pi = calcpi(Integer.parseInt(request.getParameter("n")));
                     exec = (end - start)/1000;

                     Calendar cal = Calendar.getInstance();

                     out.println(HTML_START);
                     out.println("<h2>Calculo de PI</h2><br>");
                     out.println("La cte. PI con n= ");
                     out.println(request.getParameter("n"));
                     out.println(" es igual a "+pi+"<br>");
                     out.println("<br>Tiempo de ejecucion= "+exec+"
segundos<br>");
                     out.println("<br>El servidor es
"+InetAddress.getLocalHost().getHostAddress()+"<br>");
```

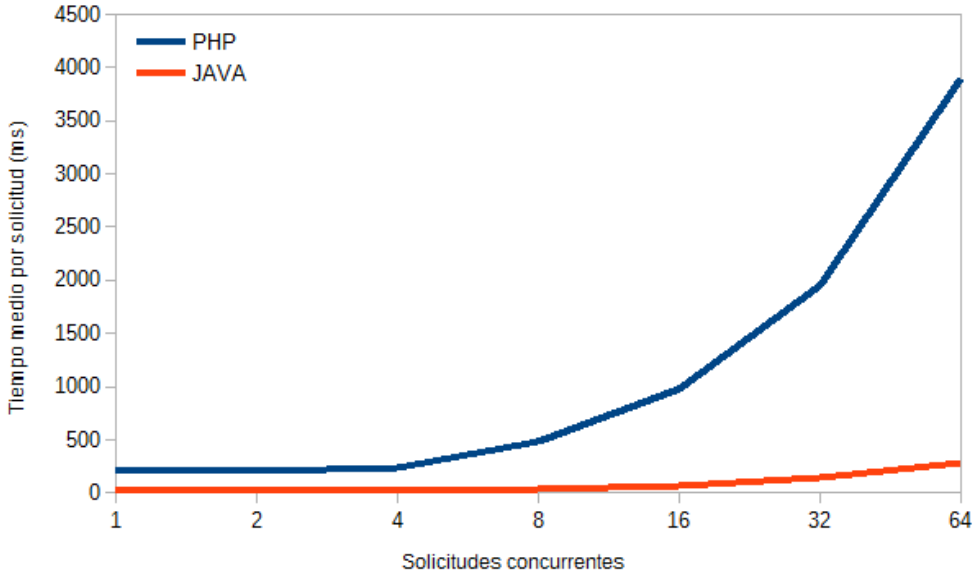
```
                    out.println("<br>Date=
"+dateFormat.format(cal.getTime())+"<br>");
                    out.println(HTML_END);

             }
      }

      public double calcpi(int n){

            double x, area = 0, result;

            start = System.currentTimeMillis();

            for (int i=0; i<n; i++){
                  x = (i + 0.5)/n;
                  area = area + 4.0/(1.0 + x*x);
            }

            result = area/n;

            end = System.currentTimeMillis();

            return result;
      }

      protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
            // TODO Auto-generated method stub
            doGet(request, response);
      }
   }
```

A.13 Prestaciones de aplicaciones usando Java Servlet

Las limitaciones en prestaciones de un sistema informático no solo se centran en la arquitectura del hardware y el sistema operativo, ya que el cuello de botella en muchas ocasiones puede estar en la aplicación y la tecnología de programación utilizada para su desarrollo. En la siguiente sección se realizó un análisis comparativo entre las tecnologías de programación PHP y Java, a través de una aplicación que calcula el valor de la constante pi mediante integración numérica. El programa acepta como dato el número de intervalos a considerar. En la gráfica se observan los resultados para n igual a 1000000 de intervalos, donde el servlet en java (Ver anexo A.12) alcanza un pico de 234 peticiones por segundo, lo que representa 14 veces más peticiones por segundo servidas en comparación con el mismo programa usando php (pi.php, ver anexo A.11).



En la siguiente gráfica es evidente como el programa en php tarda mucho más en responder a las solicitudes concurrentes conforme éstas aumentan, en cambio el servlet en java mantiene los tiempos de respuesta por debajo de los 276 ms, lo cual le permitiría seguir respondiendo una mayor cantidad de peticiones concurrentes.



A.14 Script para la asignación de núcleos sobre cada VM

```
   if [[ -z "$1" ]]
   then
   echo "Falta numero de VMs"
   exit 1
   fi

   l=15
   c=`expr 16 / $1`
   cpu=0
   vcpu=0

   for j in `seq 1 $1`;
   do
     sudo cp web$j.xml temp$j
   done

   for j in `seq 1 $1`;
   do

   sudo sed "13i \ \ <vcpu placement='static'>$c</vcpu>" temp$j > gen-
xml/web$j.xml
   sudo cp gen-xml/web$j.xml temp$j
   sudo sed "14i \ \ <cputune>" temp$j > gen-xml/web$j.xml
   sudo cp gen-xml/web$j.xml temp$j

   for i in `seq 1 $c`;
   do
       vcpu=`expr $i - 1`
       sudo sed "${l}i \ \ \ \ <vcpupin vcpu='$vcpu' cpuset='$cpu'/>" temp$j
> gen-xml/web$j.xml
       sudo cp gen-xml/web$j.xml temp$j
       l=`expr $l + 1`
       cpu=`expr $cpu + 1`
   done

   sudo sed "${l}i \ \  </cputune>" temp$j > gen-xml/web$j.xml
   sudo cp gen-xml/web$j.xml temp$j
   l=15

   done

   sudo rm -f temp*
   sudo chown root:root gen-xml/web*
   sudo chmod 600 gen-xml/web*
   sudo cp -ax gen-xml/web* /etc/libvirt/qemu/
   sudo rm -f gen-xml/web*

   for j in `seq 1 $1`;
   do
     sudo virsh define /etc/libvirt/qemu/web$j.xml
   done
```

A.15 Archivo de configuración base HAProxy

```
# 24 VMs en el backend

global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/HAProxy
        stats socket /var/run/HAProxy241.sock mode 660 level admin
        stats timeout 2m
        user HAProxy
        group HAProxy
        daemon
defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000

frontend http-frontend
    option httplog
    mode http
    bind 192.168.250.121:8081
    default_backend wwwbackend

backend wwwbackend
    mode http
    balance roundrobin
    server web1 192.168.250.181:80 check port 80
    server web2 192.168.250.182:80 check port 80
    server web3 192.168.250.183:80 check port 80
    server web4 192.168.250.184:80 check port 80
    server web9 192.168.250.189:80 check port 80
    server web10 192.168.250.190:80 check port 80
    server web11 192.168.250.191:80 check port 80
    server web12 192.168.250.192:80 check port 80
    server web17 192.168.250.197:80 check port 80
    server web18 192.168.250.198:80 check port 80
    server web19 192.168.250.199:80 check port 80
    server web20 192.168.250.200:80 check port 80
    server web25 192.168.250.205:80 check port 80
    server web26 192.168.250.206:80 check port 80
    server web27 192.168.250.207:80 check port 80
    server web28 192.168.250.208:80 check port 80
    server web33 192.168.250.213:80 check port 80
    server web34 192.168.250.214:80 check port 80
    server web35 192.168.250.215:80 check port 80
    server web36 192.168.250.216:80 check port 80
    server web41 192.168.250.221:80 check port 80
    server web42 192.168.250.222:80 check port 80
    server web43 192.168.250.223:80 check port 80
    server web44 192.168.250.224:80 check port 80
```

```
# 48 VMs en el backend

global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/HAProxy
        stats socket /var/run/HAProxy481.sock mode 660 level admin
        stats timeout 2m
        user HAProxy
        group HAProxy
        daemon

defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000

frontend http-frontend
    option httplog
    mode http
    bind 192.168.250.121:8081
    default_backend wwwbackend

backend wwwbackend
    mode http
    balance roundrobin
    server web1 192.168.250.181:80 check port 80
    server web2 192.168.250.182:80 check port 80
    server web3 192.168.250.183:80 check port 80
    server web4 192.168.250.184:80 check port 80
    server web5 192.168.250.185:80 check port 80
    server web6 192.168.250.186:80 check port 80
    server web7 192.168.250.187:80 check port 80
    server web8 192.168.250.188:80 check port 80
    server web9 192.168.250.189:80 check port 80
    server web10 192.168.250.190:80 check port 80
    server web11 192.168.250.191:80 check port 80
    server web12 192.168.250.192:80 check port 80
    server web13 192.168.250.193:80 check port 80
    server web14 192.168.250.194:80 check port 80
    server web15 192.168.250.195:80 check port 80
    server web16 192.168.250.196:80 check port 80
    server web17 192.168.250.197:80 check port 80
    server web18 192.168.250.198:80 check port 80
    server web19 192.168.250.199:80 check port 80
    server web20 192.168.250.200:80 check port 80
    server web21 192.168.250.201:80 check port 80
    server web22 192.168.250.202:80 check port 80
    server web23 192.168.250.203:80 check port 80
    server web24 192.168.250.204:80 check port 80
    server web25 192.168.250.205:80 check port 80
    server web26 192.168.250.206:80 check port 80
```

```
        server web27 192.168.250.207:80 check port 80
        server web28 192.168.250.208:80 check port 80
        server web29 192.168.250.209:80 check port 80
        server web30 192.168.250.230:80 check port 80
        server web31 192.168.250.231:80 check port 80
        server web32 192.168.250.212:80 check port 80
        server web33 192.168.250.213:80 check port 80
        server web34 192.168.250.214:80 check port 80
        server web35 192.168.250.215:80 check port 80
        server web36 192.168.250.216:80 check port 80
        server web37 192.168.250.217:80 check port 80
        server web38 192.168.250.218:80 check port 80
        server web39 192.168.250.219:80 check port 80
        server web40 192.168.250.220:80 check port 80
        server web41 192.168.250.221:80 check port 80
        server web42 192.168.250.222:80 check port 80
        server web43 192.168.250.223:80 check port 80
        server web44 192.168.250.224:80 check port 80
        server web45 192.168.250.225:80 check port 80
        server web46 192.168.250.226:80 check port 80
        server web47 192.168.250.227:80 check port 80
        server web48 192.168.250.228:80 check port 80
```