Universidad de Los Andes
Facultad de Ingeniería
División de Estudios de Postgrado
Postgrado en Computación

Adaptación de los algoritmos nativos de LVS para incorporar la carga de los servidores reales al proceso de balanceo

Autor: Ing. Marco Antonio Camejo Medina
Tutor: Gilberto Diaz

Trabajo de grado presentado ante la ilustre Universidad de Los Andes como requisito parcial para optar al grado de Magister Scientiae en Computación

Mérida, Diciembre 2013

A mi hijo Santiago Andrés

Tabla de Contenidos

Índice de Tablas

Índice de Figuras

Agradecimientos

Ante todo, vaya mi más sincero agradecimiento al profesor Gilberto, por su confianza y guía tanto técnica como personal en todo momento.
Gracias también a todo el personal del postgrado en computación, especialmente a la señora Luisa, por su apoyo y buena voluntad siempre.

Resumen

Este proyecto tiene como objetivo la incorporación de la carga de los servidores reales de un sistema de distribución de tráfico web basado en el Servidor Virtual de Linux, al proceso de balanceo. Los algoritmos de balanceo implementados en el Servidor Virtual de Linux hacen uso de un conjunto de pesos (uno por cada servidor real), asignados de forma manual por el administrador del sistema. En este trabajo se diseñó y desarrolló un mecanismo que permite actualizar de manera automática el valor de esos pesos, en función de la carga de cada servidor real. Además, se realizó un conjunto de pruebas de carga, para evaluar el impacto de esta solución, y realizar comparaciones de desempeño con respecto a un sistema basado en el Servidor Virtual de Linux tradicional.

Descriptores: Servidor Virtual de Linux, LVS, Balanceo de carga, Linux, Kernel

Capítulo 1

Introducción

La Internet ha facilitado el acceso a contenidos, servicios y aplicaciones a millones de personas alrededor del mundo. Es común hoy en día encontrar en servicios web que deben atender simultáneamente a una gran cantidad de visitantes y transferir un volumen alto de datos. Prestar a los visitantes un servicio de calidad, con alta disponibilidad y tiempos de respuesta razonables, requiere soluciones computacionales eficientes. Estas soluciones generalmente siguen 2 estrategias: utilizar un servidor único de altas prestaciones o mantener una granja o *cluster* de servidores que se comporten como si fuesen una única computadora.

La estrategia de utilizar un servidor único, generalmente no es suficiente para satisfacer la demanda del servicio. Además de que los costos de inversión son elevados, resultan difíciles de expandir y representan un único punto de falla. Por el contrario, la estrategia de utilizar granjas de servidores hacen que el sistema sea altamente escalable y aumenta la fiabilidad del servicio.

Los clusters de servidores son muy utilizados en el ámbito científico (donde generalmente se necesita un rendimiento alto) y en el ámbito empresarial (orientando principalmente a la búsqueda de alta disponibilidad y eficiencia). En particular, para el problema del tráfico web, se utilizan clusters con balanceo de carga, que son clusters de alto rendimiento, cuyo objetivo es disminuir la latencia y aumentar la capacidad de

respuesta del sistema completo.

El balanceo de carga se puede realizar mediante hardware y mediante software. Los dispositivos de balanceo de carga por hardware (*Hardware Load-Balancing Devices*) son unidades físicas, diseñadas específicamente para este propósito, que operan en diferentes capas del modelo OSI, mientras que las soluciones basadas en software hacen uso de hardware tradicional [Bourke, 2001].

El balanceo de carga por hardware, al estar programado directamente sobre los dispositivos, hace el balanceo más rápido y efectivo, sin embargo, los costos de dichos dispositivos son elevados y la mantenibilidad y desarrollo menos accesibles [Bourke, 2001].

1.1. Antedecentes

El problema del balanceo de carga en un cluster de servidores ha sido abordado desde diferentes perspectivas a lo largo de los últimos años. A continuación se presentan algunos de los trabajos más resaltantes en esta área.

Una de las primeras soluciones propuestas, se basa en la implementación del balanceo mediante **servidores DNS**, aprovechando el hecho de que éstos permiten asociar más de una dirección IP a un nombre de dominio. Así, cada vez que se haga una consulta al servidor DNS, este devolverá como resultado una de las direcciones IP de la lista que posee, utilizando un mecanismo de Round-Robin [Kozierok, 2005].

Esta solución, conceptualmente muy simple y fácil de implementar, no toma en cuenta la carga de los servidores reales, ni la disponibilidad de los mismos, por lo que resulta poco útil cuando se desea un sistema de alta disponibilidad y alto rendimiento.

Diversos autores han propuesto sistemas de balanceo de carga basados en DNS, entre estos:

- Harish, V. C. y Owens, Brad [Harish and Owens, 1999] lograron mejorar la solución básica que utiliza el algoritmo de Round-Robin, introduciendo un componente de monitoreo en los servidores DNS, que permite conocer en todo momento el nivel de ocupación de los servidores. Cada servidor informa al servidor DNS su estado, a través del envío de mensajes UDP; el servidor DNS decide a cual servidor enviar la petición basado en ese estado.

- Jong-Bae Moon y Myung Ho Kim[Moon and Kim, 2005] proponen un método de balanceo de carga dinámico, utilizando actualizaciones DNS y el mecanismo tradicional de Round-Robin, en el que se agregan o se eliminan entradas del servidor DNS, de acuerdo al nivel de carga del servidor, sin necesidad de modificar los servidores DNS.

Otra solución propuesta al problema del balanceo, involucra el uso de un **Proxy Inverso** o **Proxy en Reversa** (*Reverse Proxy*). Un proxy inverso, es un proxy que actúa como puerta de entrada a un conjunto de servidores. El proxy inverso recibe las peticiones de los usuarios, y las redirige a los servidores reales [Kew, 2004].

En 1998, [Zhang, 2000] introduce el **Servidor Virtual de Linux** (*Linux Virtual Server*, LVS), una herramienta de balanceo de carga para el sistema operativo Linux, que busca lograr niveles altos de rendimiento y disponibilidad.

Patrick O'Rourke y Mike Keefe [O'Rourke and Keefe, 2001], con la cooperación de Intel Corporation, realizaron una serie de experimentos para evaluar el desempeño del Servidor Virtual de Linux, concluyendo que representa una alternativa viable a otras soluciones de balanceo de carga basadas en software privativo. Concluyen también que la relación Beneficio-Costo del Servidor Virtual de Linux es al menos dos veces mejor que las soluciones basadas en hardware existentes en ese momento.

Jong-Bae Moon, Yong-Yoon Cho y Young-Chul Kim [Moon et al., 2004] implementan una nueva arquitectura para clusters de servidores web, basadas en el Servidor Virtual de Linux, orientada a la construcción de sitios web de tamaño pequeño y mediano. En esta arquitectura, cuando un servidor se encuentra muy cargado, pasa el trabajo a otro servidor con menor ocupación.

Primiano Tucci [Tucci, 2007] desarrolló un balanceador de carga basado en software, denominado *Linux Network Load Balancing*, en el que utiliza un algoritmo de planificación basado en una tabla hash de pesos, la cual se actualiza dinámicamente, de acuerdo a la carga del procesador o el porcentaje de memoria disponible de los servidores reales.

En el trabajo de Elizabeth Guerrero [Guerrero, 2009] se introdujo el uso de la ocupación actual de cada servidor como criterio para la planificación de tareas en un Servidor de Balanceo de Carga, basado en el Servidor Virtual de Linux. El algoritmo propuesto, comparado con el algoritmo por Round-Robin ponderado, implementado en el Servidor Virtual de Linux, arrojó mejores resultados cuando los niveles de ocupación de los servidores reales eran altos.

1.2. Planteamiento del problema

Los servidores de balanceo de carga utilizan el número de conexiones activas de los servidores reales, como criterio fundamental para la asignación de tareas. Bajo este enfoque, no se toma en cuenta la capacidad de respuesta de los servidores reales al momento de asignar las tareas, en términos de su capacidad de procesamiento y uso de otros recursos de hardware (memoria, interfaces de entrada/salida, etc).

Como resultado de no considerar estos factores, los algoritmos de planificación pueden asignar tareas a servidores, que aún cuando se encuentren atendiendo pocas conexiones, su capacidad de respuesta se encuentre comprometida por poseer una carga alta de procesamiento. El caso inverso también puede presentarse: en determinado momento, el nodo director puede obviar un servidor que pudiese dar una buena respuesta (si su nivel de ocupación fuese bajo), debido a que se encuentre atendiendo más conexiones que otros.

1.3. Objetivo general

Modificar el sistema de balanceo de carga de los algoritmos nativos del Servidor Virtual de Linux para que tomen en cuenta el nivel de ocupación de los nodos del cluster.

1.4. Objetivos específicos

- Analizar el funcionamiento y arquitectura del Servidor Virtual de Linux.
- Estudiar los algoritmos de balanceo de cargas implantados en el Servidor Virtual de Linux.
- Diseñar un mecanismo de comunicación entre el elemento balanceador y los servidores reales del Servidor Virtual de Linux, que permita a estos últimos comunicar periódicamente su nivel de ocupación al primero.
- Diseñar un mecanismo que actualice los valores de los pesos utilizados por el sistema de balanceo de carga del Servidor Virtual de Linux, tomando en cuenta el nivel de ocupación de los nodos del cluster.
- Proponer una solución, desarrollada bajo el concepto de software libre, que combine los dos mecanismos descritos en los objetivos anteriores, para mantener actualizados los pesos utilizados por el sistema de balanceo de carga del Servidor Virtual de Linux actualizados, en base a los niveles de ocupación de los nodos del cluster.
- Evaluar la correctitud de la solución propuesta.
- Comparar los resultados obtenidos mediante la solución propuesta, con los obtenidos utilizando los algoritmos nativos del Servidor Virtual de Linux.

1.5. Justificación e importancia

El problema de balanceo de carga en general es aplicable en multitud de contextos, y ha sido ampliamente estudiado. Específicamente, el cómo distribuir tráfico web entre un conjunto de servidores, ha tomado gran importancia en los últimos años, debido al incremento sostenido de usuarios conectados a Internet en todo el mundo. El Servidor Virtual de Linux es una herramienta que ha sido ampliamente utilizada por más de 10 años en el balanceo de tráfico web; sin embargo, como ya se ha mostrado, en años recientes diversos estudios se han concentrado en mejorar características específicas de esta herramienta, en búsqueda de una mayor eficiencia.

Una debilidad importante del Servidor Virtual de Linux es la falta de información en tiempo real sobre la carga de los servidores. Este estudio propone un mecanismo de incorpa dinámicamente la carga de los servidores al proceso de balanceo del Servidor Virtual de Linux, y presenta un conjunto de experimentos para la evaluación del impacto que esto causa en el desempeño del sistema.

1.6. Alcance y limitaciones

Para conseguir el objetivo de esta investigación se trabajó en dos aspectos: en primer lugar, el envío/recepción de los valores de carga de los servidores que comprenden el Servidor Virtual de Linux, y en segundo lugar, en el mecanismo de incorporación de estas cargas al proceso de balanceo.

Para la comunicación entre los equipos que forman el Servidor Virtual de Linux se implementaron dos programas (demonios del sistema operativo), que utilizan el protocolo de transporte UDP. Uno de estos programas actúa como servidor (instalado en el nodo balanceador) y el otro como cliente (instalado en los servidores web).

Para la incorporación de las cargas al proceso de balanceo, se implementó un módulo para el núcleo del sistema operativo, encargado de actualizar los pesos utilizados por el Servidor Virtual de Linux (y asignados de forma manual), en función de las cargas.

Durante la fase experimental, se llevaron a cabo pruebas de carga, tanto al Servidor Virtual de Linux original, como al Servidor Virtual de Linux con el mecanismo de actualización dinámica de pesos. Estas pruebas se limitaron al estudio de tráfico HTTP, con contenido HTML estático.

Los experimentos realizadas fueron aplicados a un Servidor Virtual de Linux con 4 equipos (un nodo balanceador de carga y tres servidores web), todos con las mismas características de hardware, y con la misma distribución de Linux: Ubuntu 12.04 con kernel versión 3.2.1.

1.7. Estructura de la monografía

Este documento está estructurado como se describe a continuación.

El capítulo 2 corresponde al basamento teórico en el cual se sustenta la investigación. El capítulo se divide en dos secciones: una dedicada al balanceo de carga y otra donde se estudian los detalles, tanto conceptuales como de implementación, del Servidor Virtual de Linux.

El capítulo 3 se refiere al marco metodológico de la investigación, y está dividido en dos secciones principales: una primera sección cubre la metodología de desarrollo, y en una segunda sección se detalla la metodología experimental utilizada para evaluar el desempeño de la solución propuesta.

Posteriormente, en el capítulo 4 se presenta formalmente el diseño propuesto para solución del problema planteado, partiendo desde los requerimientos de software, la arquitectura y funcionamiento a nivel conceptual, hasta los detalles de la implementación.

En el capítulo 5 se describen las características de los equipos e infraestructura de red utilizados durante la fase experimental, y se presentan los resultados obtenidos.

En el capítulo 6 se presentan las conclusiones a las que se llegó luego de finalizar el trabajo experimental, y las perspectivas y recomendaciones planteadas para futuras investigaciones en esta área.

Por último, se agregan dos apéndices: en el apéndice A se describen los archivos que componen el sistema de actualización dinámica de pesos, y se dan detalles de como ejecutarse, mientras que en el apéndice B se incluye el código fuente de todos los programas desarrollados.

Capítulo 2

Marco teórico

En este capítulo se presentan formalmente los conceptos teóricos sobre balanceo de carga utilizados en esta investigación. Se exponen también las características fundamentales del Servidor Virtual de Linux, las técnicas de reenvío de paquetes y los algoritmos de balanceo utilizados por este, así como también los detalles de su implementación.

2.1. Balanceo de carga

Como se adelantaba al comienzo de esta investigación, el balanceo de carga es un concepto que ha sido utilizado exitosamente en diversos ámbitos, desde el cálculo científico hasta las telecomunicaciones. Esta investigación en particular se centra en el uso de balanceo de carga, aplicado al problema de distribución de tráfico web entre varios servidores conectados a través de una o varias redes.

En un sistema de balanceo de tráfico web, los clientes (usuarios de Internet) realizan peticiones a una dirección IP, denominada **dirección IP virtual** (VIP, por sus siglas en Inglés, *Virtual IP*). Esta dirección VIP es manejada por un elemento **balanceador**, que se encarga de decidir cual de los servidores será el encargado de responder cada petición entrante y reenviársela, para que finalmente, este devuelva la respuesta al

cliente [Bourke, 2001].

2.1.1. Topología de red

Uno de los aspectos más importantes que definen un sistema de balanceo de carga es la topología de la red, que conecta servidores con el elemento balanceador. Existen dos posibilidades [Bourke, 2001]:

- Topología basada en una sola red (flat-based): las VIP y los servidores se encuentran dentro de la misma subred. Es la topología más sencilla de implementar y de administrar.
- Topología basada en redes separadas (NAT-based): las VIP y los servidores se encuentran en dos subredes diferentes, generalmente dos redes de área local virtuales (VLANs). Esta topología es más difícil de implementar, pero permite mejores configuraciones de seguridad.

2.1.2. Conectividad física

La conectividad física se refiere al tipo de conexión existente entre los servidores y el elemento balanceador, y puede ser de dos formas [Bourke, 2001]:

- Una conexión (One-armed): Existe una única conexión desde el balanceador de carga hacia la infraestructura de red. Es la configuración ideal para una topología basada en una sola red.
- Doble conexión (Two-armed): Bajo este esquema, en el elemento balanceador existen dos conexiones, cada una enlazando a una VLAN distinta: una VLAN para la subred que se conecta con los usuarios (red pública) y otra VLAN para la subred que se conecta con los servidores reales (red privada). Esta es la configuración utilizada para una topología basada en redes separadas, ya que provee dos enlaces a subredes separadas.

2.1.3. Evaluación de desempeño

Para la evaluación del desempeño de un sistema de balanceo de carga se realizan pruebas de carga (*Load testing*). Una prueba de carga consiste en someter al sistema a niveles de demanda específicos (y controlados), por un tiempo determinado, y medir su respuesta [Bourke, 2001].

Las pruebas de carga se utilizan para estudiar el comportamiento del sistema bajo condiciones de operación normal y bajo condiciones de sobrecarga, ayudando a determinar la capacidad máxima de operación, así como los cuellos de botella que puedan existir.

Los elementos más básicos que podrían definir una prueba de carga son:

- Número total de peticiones (N)
- Tasa de peticiones por unidad de tiempo (T)

En base a estos elementos, una prueba de carga consistiría en enviar peticiones al sistema bajo estudio, a una razón de T peticiones por unidad de tiempo, hasta completar un total de N peticiones, y medir su desempeño.

Diversas métricas pueden ser utilizadas para comparar el desempeño de dos sistemas de balanceo de carga. Particularmente, para estudiar el desempeño de servicios web, las métricas empleadas habitualmente son:

- **Tasa promedio de respuesta**: es la cantidad promedio de peticiones respondidas por unidad de tiempo.
- **Tiempo promedio de respuesta**: es el tiempo promedio que el sistema tarda en devolver una respuesta al cliente.

Para obtener la mayor información posible acerca del rendimiento del sistema estudiado, se realizan experimentos a diferentes niveles de demanda, hasta encontrar su

punto de saturación o punto máximo de operación, es decir, el nivel de demanda para el cual el sistema ya no es capaz de responder el 100 % de las peticiones.

Por ejemplo: se puede comenzar con 2 niveles, un nivel de demanda bajo y uno alto; si para ambos casos, el sistema no se satura, entonces quiere decir que el segundo nivel no es lo suficientemente alto; en ese caso, se escogería otro nivel superior (más alto que el original) y se repite la prueba; si ahora se logra saturar al sistema, significa que el punto de operación máximo es menor al escogido, por lo tanto, se repetiría la prueba con un nivel menor. Y así se podrían continuar realizando pruebas, hasta lograr un nivel de precisión aceptable.

Otra forma de realizar el estudio, consiste en repetir los experimentos una determinada cantidad de veces, aumentando cada vez la demanda en una razón fija (R). Es decir, se realizan N peticiones al servidor a razón de una determinada tasa de demanda T1; luego, se realizan otra N peticiones, ahora a una razón de T2 peticiones por unidad de tiempo (con T2 = T1+R) ; se repite el proceso hasta realizar N peticiones a razón de Tf peticiones por unidad de tiempo (con Tf = T1 + n*R).

- Número de peticiones totales en cada experimento (N)
- Tasa de peticiones inicial (Ti)
- Tasa de peticiones final (Tf)
- Razón de incremento de la tasa (R)

Por ejemplo: sean N=1000 peticiones, Ti=100 peticiones/segundo, Tf=200 peticiones/segundo y R=25 peticiones/segundo, entonces se realizarían los experimentos de la siguiente forma:

- En el primer experimento se realizarían 1000 peticiones, a razón de 100 peticiones/segundo.
- En el segundo experimento se realizarían 1000 peticiones, a razón de 125 (100+25) peticiones/segundo.

- En el tercer experimento se realizarían 1000 peticiones, a razón de 150 (100+2*25) peticiones/segundo.
- Se repetiría el proceso hasta realizar 1000 peticiones, a razón de 200 peticiones/segundo.

El rango de la tasa de demanda ([Ti, Tf]) depende de las características del servidor o sistema bajo estudio, así como de las características de la red. De igual forma, la razón de incremento de la tasa (R) depende en gran medida de que tan precisos deben ser los resultados.

En la figura 2.1 se muestra un ejemplo de una prueba de carga contra un servidor web, en la que se estudia el comportamiento de la tasa de respuesta. Se observa como bajo una demanda baja, el servidor es capaz de responder todas las peticiones (crecimiento lineal), pero cuando la demanda es lo suficientemente grande (en este ejemplo, alrededor de 80 peticiones/segundo), la capacidad de respuesta comienza a afectarse.

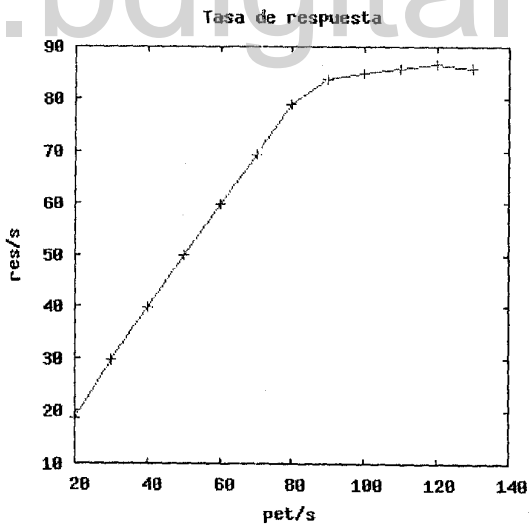


Figura 2.1: Prueba de carga. Métrica: tasa de respuesta. A partir de 80 peticiones/segundo, la capacidad de respuesta del servidor se degrada

2.2. El Servidor Virtual de Linux

El Servidor Virtual de Linux (*Linux Virtual Server*, LVS) es una solución de balanceo de carga de alto rendimiento y alta disponibilidad, basada en software, que funciona sobre un cluster de servidores bajo el sistema operativo Linux [Zhang, 2000]. El funcionamiento del LVS es transparente al usuario, y además provee tolerancia a fallos, alta escalabilidad y confiabilidad.

La arquitectura del LVS está formada por un conjunto de servidores que se encargan de atender las peticiones de los visitantes, denominados **servidores reales**, y uno o más servidores, que se encargan de distribuir la carga entre los primeros, denominados **directores**. La conexión entre el(los) nodo(s) director(es) y los servidores reales puede ser a través de una red de área local (*Local Area Network*, LAN) o una red de área amplia (*Wide Area Network*, WAN). En la figura 2.2 se muestra esta arquitectura.

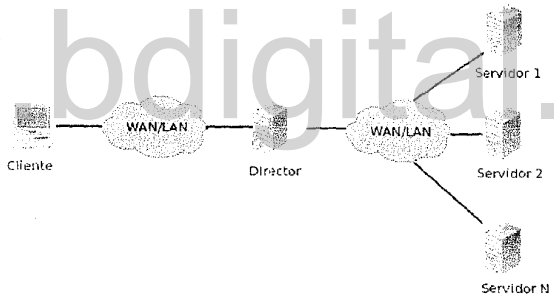


Figura 2.2: Arquitectura del Servidor Virtual de Linux

El LVS se puede implementar mediante dos mecanismos, ambos programados como un conjunto de módulos para el núcleo de Linux:

- **IP Virtual Server (IPVS)**: implementa el balanceo de carga a nivel de capa 4 del modelo OSI (capa de transporte). El IPVS se distribuye incorporado al Kernel de Linux, a partir de la versión 2.6.10, liberada el 24 de diciembre de 2004 [Zhang, 2000].

- **Kernel TCP Virtual Server (KTCPVS)**: implementa el balanceo de carga a nivel de la capa 7 del modelo OSI (capa de aplicación). La ventaja de realizar el balanceo en esta capa, es que se puede realizar la asignación de tareas a los servidores reales en base al contexto de las peticiones, es decir, permite crear reglas de balanceo en base al contenido solicitado (por ejemplo: utilizar un conjunto de servidores para responder contenido estático y otro grupo de servidores para contenido dinámico); sin embargo, por esta misma razón, resulta menos escalable que el IPVS [Zhang, 2000].

2.2.1. Técnicas de reenvío de paquetes en el LVS

En la primera sección de este capítulo se trató el tema de las características de las redes utilizadas en el balanceo de carga. En esta sección se describen las técnicas de funcionamiento del LVS, es decir, las técnicas empleadas para hacer llegar las peticiones de los clientes a los servidores reales y luego las respuestas desde los servidores reales hasta los clientes, de acuerdo a las configuraciones de red.

En este sentido, el Servidor Virtual de Linux utiliza 3 técnicas: Traducción de Direcciones de Red (*Network Address Translation*), Encadenamiento IP (*IP Tunneling*) y Enrutamiento Directo (*Direct Routing*). A continuación se detalla cada una de ellas.

Traducción de direcciones de red

La traducción de direcciones de red o NAT (por sus siglas en inglés), es una técnica que se utiliza para intercambiar paquetes entre dos redes, convirtiendo en tiempo real las direcciones de red entre ambas. Es el método que permite que direcciones de red privadas puedan acceder a la Internet.

Bajo esta técnica, el funcionamiento del LVS es el siguiente:

1. El nodo director se encarga de recibir las peticiones de los clientes.

2. El nodo director selecciona el servidor que responderá la petición (de acuerdo al algoritmo de balanceo).
3. El nodo director realiza la traducción de dirección para reenviarle la petición al servidor seleccionado.
4. El servidor envía la respuesta de vuelta al director.
5. El director realiza la traducción de dirección inversa, para hacer el reenvío de la respuesta al cliente.

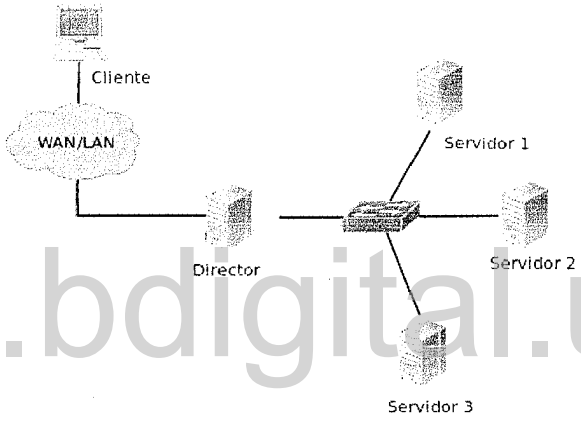


Figura 2.3: LVS con Traducción de direcciones de red

Es una técnica sencilla de implantar, sin embargo, tiene la desventaja de que el nodo director puede convertirse en un cuello de botella, cuando el número de peticiones es muy grande, debido a que debe reescribir todos los paquetes de datos desde y hacia los servidores reales.

Encapsulamiento IP

El Encapsulamiento IP (*IP Tunneling*) es una técnica utilizada para el envío de paquetes IP, insertos en otros paquetes IP, es decir, la parte de datos del segundo paquete IP es el primer paquete IP.

Bajo esta técnica, el funcionamiento del LVS es el siguiente:

1. El nodo director se encarga de recibir las peticiones de los clientes.
2. El nodo director selecciona el servidor que responderá la petición (de acuerdo al algoritmo de balanceo).
3. El nodo director encapsula el paquete IP original dentro de otro paquete IP y lo envía al servidor seleccionado.
4. El servidor desencapsula el paquete envíado por el director y envía la respuesta al cliente, sin tener que hacer pasar los paquetes a través del director. Para ello, es necesario que los servidores posean configurada una interfaz con la dirección IP pública (dirección IP Virtual).

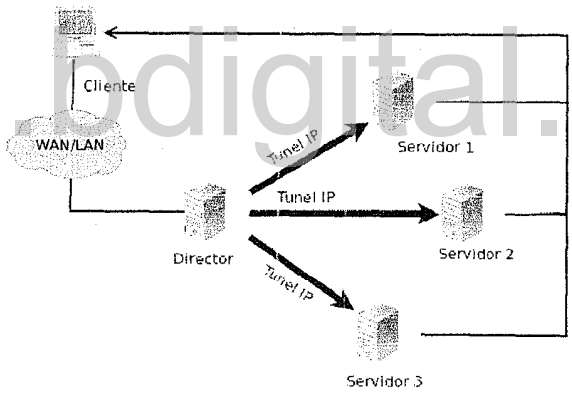


Figura 2.4: LVS con Encapsulamiento IP

La ventaja más importante que posee, respecto a la técnica de traducción de direcciones de red, es que al enviar las respuestas desde los servidores hasta los clientes directamente, el nodo director no representa un cuello de botella.

La limitación principal de esta técnica, es que todos los elementos del sistema deben soportar el encapsulamiento/desencapsulamiento de paquetes IP.

Enrutamiento directo

Es la técnica más utilizada, debido a su sencillez tanto conceptual como de implementación. Se basa en la utilización de una red de área local (con un único segmento de red) y un cambio de direcciones IP-MAC para el reenvío de paquetes.

Bajo esta técnica, el funcionamiento del LVS es el siguiente:

1. El nodo director se encarga de recibir las peticiones de los clientes.
2. El nodo director selecciona el servidor que responderá la petición (de acuerdo al algoritmo de balanceo).
3. El nodo director reenvía el paquete a la dirección MAC del servidor seleccionado.
4. El servidor envía la respuesta al cliente, sin tener que hacer pasar los paquetes a través del director. Para ello, al igual que con la técnica de encapsulamiento IP, es necesario que los servidores posean configurada una interfaz con la dirección IP pública (dirección IP Virtual).

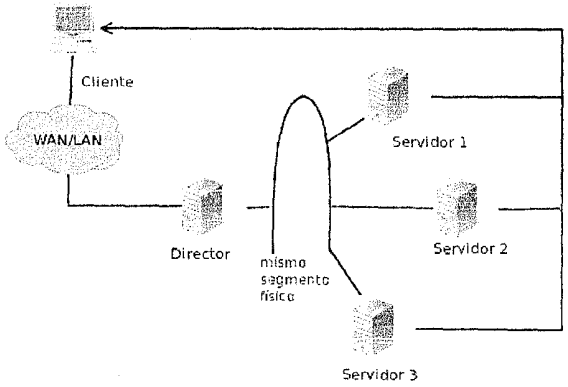


Figura 2.5: LVS con Enrutamiento directo

Con esta técnica, al igual que con la técnica de encapsulamiento IP, el nodo director no representa un cuello de botella, ya que las respuestas no deben pasar por él.

Además, el coste computacional de realizar el cambio IP-MAC, es menor que el de el encapsulado/desencapsulado de paquetes, por lo que su rendimiento es mejor.

2.2.2. Problema ARP

Para utilizar las técnicas de encapsulamiento IP o enrutamiento directo, si los servidores reales y director(es) se encuentran en un mismo segmento de red, se debe resolver un problema adicional, que surge a raiz del uso compartido de la dirección IP Virtual, y está directamente relacionado con el Protocolo ARP, por lo que se le denomina Problema ARP [Zhang, 2000].

El Protocolo de Resolución de Direcciones o Protocolo ARP (por sus siglas en inglés, *Address Resolution Protocol*), es el utilizado por los enrutadores para determinar las direcciones físicas (direcciones MAC) asociadas a las direcciones IP del segmento de red que enruta. Para lograrlo, los enrutadores envían un mensaje ARP a la dirección de broadcast de la subred, y el equipo con la dirección IP pedida, envía como respuesta su dirección física.

Dado que cuando se tiene un sistema LVS dentro de la subred, la dirección IP virtual es compartida por el(los) director(es) y servidores reales, cualquiera de ellos podría responder a la petición ARP, no solamente el nodo director. Si alguno de los servidores reales respondiera, su dirección física sería entonces asociada con la dirección IP virtual y almacenada en la tabla ARP del enrutador. Como resultado, mientras no se limpie la tabla ARP, cada nueva petición que llegue a la dirección IP virtual, sería envíada directamente al servidor real, originando diferentes problemas. No se realizaría balanceo y toda la carga la recibiría el servidor real en cuestión, y si este llegara a fallar, el cliente no recibiría respuesta, aún cuando los demás servidores reales esten activos y libres de carga.

El Problema ARP consiste entonces en cómo garantizar que sea exclusivamente el nodo director el que responda a las peticiones ARP enviadas por el enrutador, para lo cual se pueden adoptar diferentes enfoques, que se describen a continuación.

Enfoque 1: Utilizar una interfaz oculta

Es el enfoque más antiguo, consiste en ocultar la VIP en los servidores reales, de manera que estos ignoren las peticiones ARP. Se basa en el uso de una bandera "hidden" (oculta) sobre la interfaz asociada a la VIP. Tiene la desventaja de que se debe aplicar un parche al núcleo (y recompilarlo).

Enfoque 2: Política de enrutamiento

Otro enfoque que ha sido utilizado para resolver el problema ARP, consiste en utilizar una política de enrutamiento que bloquee el acceso a la dirección VIP desde la red local, evitando así que los servidores reales respondan a las peticiones ARP del enrutador.

Este enfoque tiene la ventaja de ser de fácil implementación, sin embargo, hace que el LVS no se pueda utilizar desde la red local.

Enfoque 3: Política de respuestas ARP

A partir de la versión 2.4.26 del núcleo de Linux, se incluyen las banderas *arp_announce* y *arp_ignore*, las cuales permiten establecer diferentes niveles de restricción para anunciar la dirección IP local al responder peticiones ARP. Una correcta configuración de estas dos banderas en los servidores reales, permite resolver de manera sencilla el problema ARP, y es la solución recomendada para los núcleos actuales.

2.2.3. Algoritmos nativos de planificación de tareas del LVS

Para distribuir la carga entre los servidores reales, el LVS implementa los siguientes algoritmos de planificación:

- Planificación por Round-Robin (*Round-Robin Scheduling*).

- Planificación por Round-Robin ponderado (*Weighted Round-Robin Scheduling*).
- Planificación por menor número de conexiones (*Least-Connection Scheduling*).
- Planificación por menor número de conexiones ponderado (*Weighted Least-Connection Scheduling*).
- Planificación por menor número de conexiones local (*Locality-Based Least-Connection Scheduling*).
- Planificación por menor número de conexiones local con réplicas (*Locality-Based Least-Connection with Replication Scheduling*).
- Planificación por hashing de destino (*Destination Hashing Scheduling*).
- Planificación por hashing de origen (*Source Hashing Scheduling*).
- Planificación por menor retardo esperado (*Shortest Expected Delay Scheduling*).
- Planificación por servidores sin peticiones en espera (*Never Queue Scheduling*).

Los algoritmos de planificación por Round-Robin y Round-Robin ponderado son los más sencillos. El primero se basa en un esquema de Round-Robin tradicional: se mantiene una lista con los servidores reales, y las peticiones son asignadas, a medida que van llegando, de acuerdo al orden de la lista. En el algoritmo de Round-Robin ponderado, a cada servidor real se le asigna un peso (un valor entero que representa su capacidad de procesamiento), y las tareas son asignadas por el(los) nodo(s) director(es) de acuerdo a ese peso.

Los algoritmos de planificación *Least-Connection* y *Weighted Least-Connection* se basan en el número de conexiones activas. En el primero se entrega la petición al servidor con el menor número de conexiones activas establecidas. El segundo algoritmo se basa en el primero, pero pondera el número de conexiones activas de cada servidor real, por un peso asignado estáticamente, que busca representar la capacidad de procesamiento del servidor.

El algoritmo *Locality-Based Least-Connection* funciona de manera similar al algoritmo *Least-Connection*, pero mantiene una tabla caché por dirección IP destino; si el servidor solicitado se encuentra en la tabla caché y además no se encuentra sobrecargado (donde la sobrecarga significa que el servidor tenga más conexiones activas que el valor de su peso), le asignará la petición, en caso contrario, utilizará el algoritmo *Weighted Least-Connection* para seleccionar el servidor. *Locality-Based Least-Connection with Replication*, es muy parecido al anterior, pero en vez de tener un servidor por destino, mantiene un conjunto de servidores. Si ninguno de los servidores del conjunto está disponible, utiliza el algoritmo *Weighted Least-Connection* para seleccionar el servidor.

Los algoritmos por *hashing* (destino y origen) designan el servidor que responderá la petición entrante, mediante una búsqueda en una tabla hash estática, por su IP destino o IP origen, según sea el caso. Si el servidor real se encuentra caído o sobrecargado (donde la sobrecarga significa que su número de conexiones activas es al menos el doble que el valor de su peso), entonces el director no asignará la tarea a ningún servidor y el usuario no obtendrá una respuesta satisfactoria.

El algoritmo por menor retardo esperado, asigna la petición al servidor que en ese momento tenga la menor demora esperada, definida esta para el i-ésimo servidor como $(Ci+1)/Ui$, donde Ci es el número de conexiones activas y Ui es el peso de ese servidor. El algoritmo por servidores sin peticiones en espera, si encuentra a un servidor inactivo (sin conexiones activas), le asignará la carga, en caso contrario, responderá como el algoritmo por menor retardo esperado.

2.2.4. Implementación del IP Virtual Server

El IP Virtual Server (IPVS), la implementación de capa 4 del Servidor Virtual de Linux, desde la versión 2.6.28 del núcleo de Linux, se encuentra en el directorio `/net/netfilter/ipvs/`. En este directorio se implementan tanto el núcleo del LVS, como los algoritmos de planificación presentados anteriormente.

El núcleo del LVS está estructurado de forma modular. Cada módulo se encarga de manejar aspectos específicos del LVS. Por ejemplo `ip_vs_ftp` se encarga de realizar tareas relativas al protocolo ftp, `ip_vs_sync` se encarga de manejar la sincronización de servidores reales con el director, y `ip_vs_est` implementa un estimador de tasas utilizado desde otras partes del LVS.

Los algoritmos de planificación se encuentran implementados también como módulos del núcleo de Linux, y son considerados como servicios para el LVS. Cuando uno de estos módulos se carga, se asocia el servicio correspondiente con el LVS, y por lo tanto, cada vez que llegue una petición a la VIP, se ejecutará ese algoritmo para seleccionar el servidor que responderá al cliente.

Dentro de cada módulo se encuentran implantadas, además del algoritmo de balanceo per se, algunas de las siguientes funciones:

- Función de inicialización (init_service): se ejecuta cuando se asocia el algoritmo al LVS. Se utiliza generalmente para inicializar las variables utilizadas por el algoritmo de balanceo; por ejemplo, el algoritmo de Round-Robin ponderado, la utiliza para crear una lista con los pesos de los servidores reales.
- Función de actualización (update_service): se ejecuta al actualizar o eliminar alguno de los servidores reales. Algunos algoritmos de balanceo utilizan variables internas cuyo valor es una función de los pesos dados a los servidores reales, por lo que, de actualizarse algún peso, deben recalcularse los valores de esas posibles variables; es entonces en la función de actualización, donde se incluyen las rutinas necesarias para ese recálculo.
- Función de finalización (done_service): se ejecuta cuando se termina el servicio, es decir, cuando se descarga el módulo correspondiente. Se usa generalmente para liberar la memoria utilizada por variables propias del algoritmo (módulo) de balanceo.

En la tabla 2.1 se muestra un resumen de los algoritmos de planificación implementados en el LVS, junto con el nombre abreviado del algoritmo, una columna que indica

si utiliza pesos o no, y 3 columnas adicionales que indican cuales algoritmos tienen definidas funciones de inicialización, actualización y finalización.

Los únicos algoritmos que no hacen uso de pesos son los de Round-Robin y Menor número de conexiones; el resto de los algoritmos hacen uso de los pesos asignados por el administrador del sistema, como factor de ponderación.

Los algoritmos de Hashing por destino y Hashing por origen hacen uso de la función `update_service`, pues cada vez que se modifica (o se elimina) alguno de los servidores reales, se debe actualizar la tabla hash utilizada para realizar el balanceo. Los otros algoritmos que utilizan `update_service` son el de Round-Robin y Round-Robin ponderado, ya que estos utilizan una lista interna de servidores que debe actulizarse. Los demás algoritmos, en cambio, no mantienen internamente copias de la lista de servidores (ni variables que pudiesen cambiar si estos cambian), por lo cual no necesitan ejecutar esta rutina cada vez que se actualiza la lista global de servidores reales.

2.3. Indicadores de carga de procesador en Linux

En las dos primeras secciones de este capítulo se detallaron los conceptos fundamentales del problema de balanceo de carga en servidores web, y los detalles conceptuales y de implementación de una de las soluciones de balanceo de carga basadas en software más utilizada hoy en día, el Servidor Virtual de Linux.

Al describir los algoritmos de balanceo implantados en el Servidor Virtual de Linux, se destacó el uso de un conjunto de pesos asignados a los servidores reales, como indicadores de su capacidad de procesamiento. Lo que se pretende en esta monografía, es hacer que esos pesos se actualicen dinámicamente, en función de la carga de los procesadores de los servidores reales, entendiéndose la **carga de procesador** como la cantidad de trabajo que ese procesador tiene en determinado período de tiempo.

Dos tipos de indicadores se han utilizado tradicionalmente para representar la carga de un procesador: un tipo de indicador basado en el porcentaje de utilización y el otro

algoritmo	abrev.	pesos	init_service	update_service	done_service
Round-Robin	rr	No	Sí	Sí	No
Round-Robin ponderado	wrr	Sí	Sí	Sí	Sí
Menor número de conexiones	lc	No	No	No	No
Menor número de conexiones ponderado	wlc	Sí	No	No	No
Menor número de conexiones local	lblc	Sí	Sí	No	Sí
Menor número de conexiones local con réplicas	lblcr	Sí	Sí	No	Sí
Hashing destino	dh	Sí	Sí	Sí	Sí
Hashing origen	sh	Sí	Sí	Sí	Sí
Menor retardo esperado	sed	Sí	No	No	No
Sin peticiones de espera	nq	Sí	No	No	No

Tabla 2.1: Algoritmos de planificación del LVS

basado en el largo de la cola de ejecución. Sin embargo, de acuerdo al estudio comparativo realizado por [Ferrari and Zhou, 1988], el uso del largo de la cola de ejecución del procesador, arroja mejores resultados.

Los sistemas operativos Unix/Linux, realizan dentro de su núcleo el cálculo de 3 valores de carga promedio del procesador, basados en el largo de la cola de ejecución: el promedio de carga en el último minuto, el promedio de carga en los últimos 5 minutos y el promedio de carga en los últimos 15 minutos. Se pueden acceder a estos indicadores mediante un archivo virtual del sistema de archivos de procesos: `/proc/loadavg`, o a través de los comandos `uptime`, `w` y `top`.

Capítulo 3

Metodología

Con esta investigación se busca modificar el Servidor Virtual de Linux, introduciendo un mecanismo de actualización dinámica para uno de los parámetros utilizados en el proceso de balanceo (los pesos de los servidores reales). Se trata de una investigación de evaluación o estudio de proyecto, que comprende dos grandes etapas: una etapa de desarrollo del proyecto, es decir de la aplicación, y otra etapa de evaluación experimental de la solución planteada.

3.1. Metodología de desarrollo

Considerando las características del proyecto, y la estructura del equipo de desarrollo, en esta investigación se optó por un enfoque de desarrollo ágil de software.

[Fuggetta and Bstract, 2003] menciona entre las metodologías adecuadas para el desarrollo de proyectos de Software Libre el Prototipado Rápido, el Desarrollo Rápido de Aplicaciones y el Desarrollo Ágil. En particular, las prácticas de Desarrollo Ágil aplicadas al software libre han probado en los últimos años resultar exitosas, a tal punto que son utilizadas en el desarrollo de proyectos tan importantes como la distribución de Linux Ubuntu [Taft, 2008].

Las metodologías ágiles se caracterizan por emplear un efoque incremental, basado en ciclos de iteraciones cortos, que buscan proveer una mejora al producto en cada paso o iteración [Abrahamsson et al., 2002].

Otros de los aspectos fundamentales de estas metodologías son la simplicidad en el desarrollo y la cooperación constante entre clientes y desarrolladores. Estos aspectos, combinados con los ciclos de desarrollo cortos, hacen que los proyectos desarrollados mediante metodologías ágiles puedan corregir efectivamente (y en corto tiempo) los errores y además, sean adaptativos.

En el trabajo de [Goldman and Gabriel, 2005] se analizan los principios del desarrollo ágil [Beck et al., 2001], concluyendo que son, en general, cónsonos con el desarrollo de software de código abierto, siendo la fuente principal de similitud, el énfasis de ambos esquemas en el continuo diseño del software.

3.1.1. Metodología PXP

Existen diversas metodologías calificadas como ágiles, entre las cuales se pueden destacar las siguientes:

- Adaptive Software Development (ASD).
- Agile Unified Process (AUP).
- Dynamic Systems Development Method (DSDM).
- Extreme Programming (XP).
- Feature Driven Development (FDD).
- Scrum.

Luego de estudiar estas metodologías, se decidió utilizar para el desarrollo de esta investigación, la metodología Extreme Programming (Programación Extrema), la

cual se adapta a los requerimientos y características del proyecto. *Extreme Programming* es una metodología ágil, desarrollada originalmente por Kent Beck, que fomenta la simplicidad de las soluciones y está especialmente orientada a grupos de trabajo pequeños.

Estudios recientes han refinado Extreme Programming, para el caso específico en el que los equipos de desarrollo son de una sola persona; es así como surge la **Programación Extrema Personal** (*Personal Extreme Programming*, PXP), definida y evaluada por [Agarwal and Umphress, 2008]. Personal Extreme Programming no utiliza el método tradicional (estático) de cascada, sino que en su lugar emplea un ciclo de vida dinámico e iterativo multietapa que se describe a continuación.

1. Toma de Requisitos. Durante esta etapa se reunen los requisitos funcionales y no funcionales del sistema. Se realiza en conjunto con quienes serán los usuarios del sistema.

2. Planificación. En esta etapa, la lista de requisitos del sistema se traduce en tareas específicas. Cada tarea puede estar compuesta por tareas de menor complejidad. En esta etapa además, antes de la planificación de tareas, se escogen aspectos fundamentales del sistema: sistema operativo, lenguaje de programación, modelo de aplicación, etc.

3. Inicialización de la iteración. Marca el inicio de un proceso iterativo, que busca ir mejorando cada vez más la solución. En esta etapa se seleccionan las tareas específicas sobre las cuales se enfocará la iteración.

4. Diseño. Durante esta etapa se modelan los módulos, clases y demás elementos de software que serán utilizadas durante la iteración.

5. Implementación. Durante esta fase se genera el código fuente. Se deben implementar todos los elementos especificados en la fase de diseño y no se deben producir errores de compilación.

6. Prueba del sistema. Durante esta etapa se realizan pruebas a todas las funcionalidades desarrolladas en la etapa anterior.

7. Retrospección. Determina el final de una iteración. Durante esta etapa se analizan los datos recolectados en las etapas anteriores y bien se genera un producto final o una versión candidata (*Release Candidate*) y se inicia una nueva iteración, volviendo a la fase de inicialización de iteración.

La Programación Extrema Personal comienza con la solución más simple, y en cada iteración se van agregando funcionalidades. Durante todo el proceso, el desarrollador mantiene un registro con información sobre las tareas planificadas, sugerencias para posibles mejoras y errores encontrados.

3.1.2. Escenario de desarrollo

De los escenarios descritos por [Akpata and Riha, 2004] en los cuales la Programación Extrema puede ser utilizada por un equipo de una persona, este proyecto clasifica dentro de la categoría *Desarrollador Privado*, en el cual el cliente es el mismo desarrollador y por consiguiente posee control total del proceso de desarrollo, y cuyo objetivo es el de crear una herramienta de software de código fuente abierto.

3.2. Metodología de evaluación

Para evaluar el desempeño de la solución planteada, se condujeron pruebas de carga (*Load testing*), a fin de realizar comparaciones con el desempeño de los algoritmos originales utilizados por el LVS. La comparación se realiza en base a las siguientes variables:

- **Tasa promedio de respuesta**
- **Tiempo promedio de respuesta**

Estas pruebas de carga se realizaron utilizando herramientas de generación de tráfico web, de distribución libre:

1. **httperf**: Es una herramienta para medir el rendimiento de servidores web; funciona enviando peticiones al sistema a una tasa fija y midiendo la tasa y tiempo de respuesta. Trabaja en sistemas operativos Linux y HP-UX.

2. **Autobench**: Es un script desarrollado en lenguaje Perl, que utiliza iterativamente httperf, permitiendo incrementar el número de conexiones en cada iteración, extrayendo información de la salida de httperf. Funciona bajo plataformas Linux.

El conjunto de experimentos aplicados fue planificado para realizarse en 5 etapas, las cuales se describen a continuación.

3.2.1. Etapa 1: Caracterización de la respuesta utilizando algoritmos clásicos

El objetivo de este conjunto de experimentos es estudiar el comportamiento de los algoritmos de balanceo del LVS, en condiciones naturales. Las pruebas se realizaron sin aplicar sobrecarga en ninguno de los servidores reales.

3.2.2. Etapa 2: Estudio de la respuesta con actualización dinámica de pesos

El objetivo de este conjunto de experimentos es estudiar el comportamiento del sistema, cuando se utiliza la actualización dinámica de pesos, sin aplicar sobrecarga en los servidores reales. Así, se analiza de que forma esta solución afecta el desempeño de los algoritmos originales, en términos de las tasas y tiempos de respuesta.

Se aplicaron los mismos experimentos que en la etapa anterior, con el fin de poder realizar comparaciones posteriormente.

3.2.3. Etapa 3: Caracterización de la respuesta ante servidores cargados

El objetivo de este conjunto de experimientos es estudiar el comportamiento de los algoritmos clásicos ante servidores cargados. Se realizaron las mismas pruebas que en las dos secciones anteriores, pero aplicando sobrecarga en los servidores reales: primero a un servidor, luego a dos y finalmente a los tres servidores.

3.2.4. Etapa 4: Estudio de la respuesta con actualización dinámica de pesos ante servidores cargados

El objetivo de este conjunto de experimentos es evaluar el desempeño del sistema utilizando actualización dinámica de pesos ante servidores cargados. Las pruebas son las mismas aplicadas en la sección anterior, pero con actualización dinámica de pesos, es decir, para cada algoritmo se realizaron las pruebas de carga con un servidor sobrecargado, luego con dos servidores sobrecargados, y finalmente, con los tres servidores sobrecargados.

3.2.5. Etapa 5: Comparación de los resultados

Una vez concluidas las fases experimentales, y en base a los resultados obtenidos, se realiza una comparación del desempeño del sistema original con el desempeño del sistema propuesto, en los distintos niveles de carga estudiados (ningún servidor cargado, un servidor cargado, dos servidores cargados y tres servidores cargados).

En la tabla 3.1 se presenta en resumen, las etapas que comprende la metodología experimental.

Etapa	Serv. sobrecargados	Act. dinámica	Objetivo
1	No	No	Caracterizar la respuesta utilizando algoritmos clásicos
2	No	Sí	Estudiar la respuesta con actualización dinámica de pesos
3	Sí	No	Caracterizar la respuesta ante servidores cargados
4	Sí	Sí	Estudiar la respuesta con actualización dinámica de pesos ante servidores cargados
5	-	-	Comparar los resultados obtenidos

Tabla 3.1: Resumen de etapas de la metodología de evaluación

Capítulo 4

Diseño e implementación

Este capítulo contiene el diseño de los componentes de software necesarios para lograr la actualización dinámica de los pesos de los servidores reales de un sistema de balanceo basado en LVS, así como los detalles de implementación.

4.1. Diseño

El objetivo de este proyecto es modificar el sistema de balanceo de carga del Servidor Virtual de Linux, para que se actualicen de manera dinámica los pesos utilizados en el proceso de balanceo, en función de la carga real de los servidores. A continuación se detallan los requerimientos determinados en la primera etapa de la metodología de desarrollo PXP, para posteriormente, en base a esos requerimientos, presentar la arquitectura propuesta y su funcionamiento.

4.1.1. Requerimientos

- Se debe crear un mecanismo de comunicación entre los servidores reales y el nodo director, de manera que los primeros le puedan informar periódicamente al último,

su estado de carga. Esta información debe almacenarse en el nodo director, para que pueda ser accedida desde el núcleo del Servidor Virtual de Linux.

- El formato en el que se almacene la información de carga de los servidores reales en el nodo director debe ser lo más simple posible, para que el costo computacional relativo al proceso de recuperación o lectura sea mínimo.
- El núcleo del LVS debe modificarse para que antes de ejecutar el algoritmo de balanceo, actualice los pesos de los servidores, de acuerdo a la información de carga.
- La solución propuesta debe ser lo menos intrusiva posible, ya que el núcleo del Servidor Virtual de Linux, así como los algoritmos de balanceo utilizados por este, se encuentran implementados en el kernel de Linux.
- La solución propuesta debe dar la posibilidad al administrador del sistema de activar y desactivar la actualización dinámica de pesos en cualquier momento.
- El mecanismo de actualización de cargas debe ser independiente del LVS, de manera tal que pueda ser susituido sin problemas por otro mecanismo que guarde la información de carga en el mismo formato.

4.1.2. Arquitectura

A partir de los requerimiento anteriores, se identifican dos componentes de software: uno encargado del envío y recepción de cargas desde los servidores reales hacia el nodo director, y otro encargado de la actualización de los pesos asignados a los servidores reales, dentro del ámbito del Servidor Virtual de Linux.

En la figura 4.1 se presenta la arquitectura del sistema propuesto, basada en estos dos componentes.

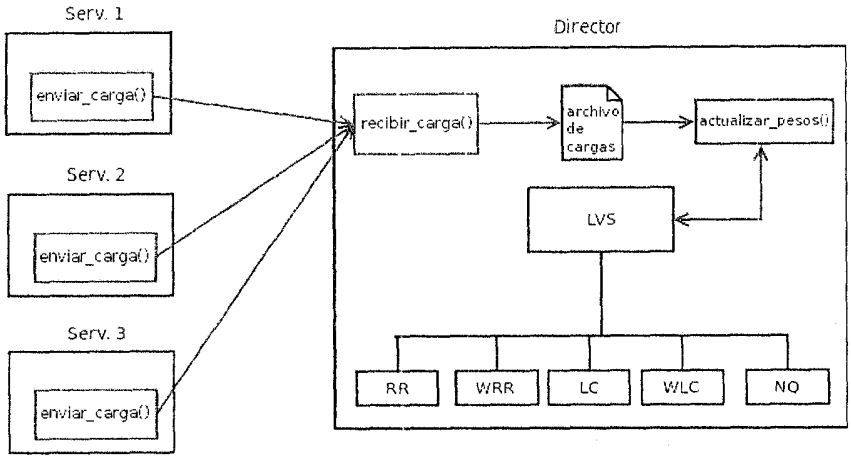


Figura 4.1: Funcionamiento del sistema LVS con actualización dinámica de pesos

Componente de actualización de cargas

El objetivo de este componente es mantener actualizado al nodo director con los valores de carga de los servidores reales, y funciona de manera independiente al Servidor Virtual de Linux. Los valores de carga se mantienen en un archivo del sistema, para que puedan ser accedidos desde el espacio de Kernel (donde funciona el Servidor Virtual de Linux).

Desde los servidores reales se deben enviar periódicamente los valores de carga, que deben ser recibidos y procesados en el nodo director. Los datos son transmitidos entonces en forma unidireccional (desde los servidores hacia el nodo director), y el componente puede ser subdividido en dos:

- Sub-componente de envío de cargas: ubicado en los servidores reales, su única función es la de enviar periódicamente su carga actual.
- Sub-componente de recepción de cargas: ubicado en el nodo director, su función es la de recibir los valores de carga enviados por los servidores reales y mantener el archivo de cargas actualizado con los datos más recientes.

Componente de actualización de pesos

El objetivo de este componente es actualizar la lista de pesos asignado a los servidores reales, en función de la carga, al momento de ejecutar el algoritmo de balanceo.

Este componente utiliza los datos del archivo de cargas (mantenido por el componente de actualización de cargas) para actualizar una lista interna de servidores reales (con sus pesos), con la cual alimenta la lista de destinos del Servidor Virtual de Linux. A diferencia del componente de actualización de cargas, la actualización de pesos debe realizarse en el espacio de kernel.

4.1.3. Funcionamiento

El componente de actualización de cargas funciona de la siguiente manera (figura 4.2):

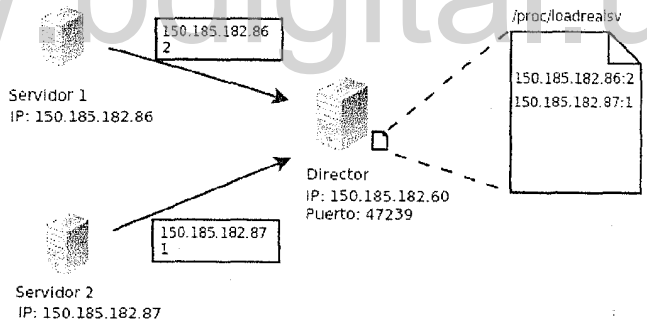


Figura 4.2: Funcionamiento del componente de actualización de cargas

1. En el nodo director se mantiene abierto un puerto específico para la recepción de mensajes desde los servidores reales.

2. Un servidor real envía un paquete con su carga actual a la dirección/puerto de escucha del nodo director.

3. El nodo director abre el archivo de cargas y busca la dirección IP del servidor real; Si no la encuentra, agrega la tupla (IP:Carga) al archivo. Si la encuentra, sólo actualiza el valor de carga con el que acaba de recibir.

4. Se guarda el archivo de cargas.

El componente de actualización de pesos, funciona de la siguiente manera:

1. Crea una lista de servidores reales, con sus cargas.

2. Cada vez que se actualiza el archivo de cargas, se actualiza la lista de cargas con los nuevos valores.

3. Cuando llega una petición, el Servidor Virtual de Linux funciona normalmente, pero actualiza los pesos de los servidores reales (de acuerdo a la lista de cargas) antes de ejecutar el algoritmo de balanceo.

4.2. Implementación

Al tratarse de una modificación al Servidor Virtual de Linux, queda intrínsecamente definido el sistema operativo (Linux) y el lenguaje de programación utilizado para desarrollar las piezas principales de software (lenguaje C). La implementación, que se detalla a continuación, fue realizada sobre un núcleo versión 3.2.1 (versión estable más reciente al momento de realizar el desarrollo).

4.2.1. Componente de actualización de cargas

Al no funcionar directamente sobre el Servidor Virtual de Linux, ni el sub-componente de envío de carga ni el de recepción requieren estar programados en espacio de Kernel, sino como procesos o servicios del sistema operativo. En el trabajo de [Guerrero, 2009] se utilizaron exitosamente Sockets de datagrama, que funcionan sobre

el protocolo de transporte UDP, como el medio de comunicación utilizado para enviar datos de carga desde los servidores reales hasta el nodo director, y es la estrategia seguida en esta monografía.

Sub-componente de envío de cargas

Este componente está constituido por un demonio (`lvsload_udp_s.c`) y un script de inicialización (`lvsload_svr`). El proceso de envío de carga se encuentra implementado en el demonio; el script se utiliza para controlar su inicio, pausa y reinicio, siguiendo el formato de la Linux Standard Base (LSB).

Sub-Componente de recepción de cargas

Este componente, al igual que el de envío, está constituido por un demonio (`lvsload_udp_d.c`) y un script de inicialización (`lvsload_dir`).

4.2.2. Componente de actualización de pesos

Este componente se programó como un módulo para el núcleo, complementario al Servidor Virtual de Linux, de manera de que las modificaciones del código original sean mínimas, y además que pueda ser cargado o descargado por el administrador del sistema.

Como se explicó en el segundo capítulo de esta monografía, el Servidor Virtual de Linux está estructurado de forma modular, dentro del Kernel del sistema operativo, dejando a los algoritmos de balanceo como módulos de servicio, permitiendo a los desarrolladores programar nuevos módulos de balanceo, como el propuesto por [Guerrero, 2009].

Luego de estudiar exhaustivamente la estructura y funcionamiento del LVS, se determinó que el algoritmo `ip_vs_schedule()`, implementado en el archivo `ip_vs_core.c`,

es el encargado de invocar la ejecución del algoritmo de balanceo activo. Como primera solución se implementó la rutina de actualización de pesos justo antes de esta invocación, evitando así realizar modificaciones en cada uno de los módulos de balanceo.

En una segunda iteración del sistema, para cumplir con los requerimientos de diseño, se decidió realizar un módulo donde se implantaron los algoritmos relacionados con la actualización de pesos, para separar su implantación de `ip_vs_core.c`. A este nuevo módulo se le nombró `ip_vs_load.c`. De esta manera, la modificación al núcleo del LVS fue mínima (sólo se agrega la llamada al algoritmo de actualización de pesos).

Sin embargo, esta nueva solución tenía el inconveniente de que el módulo de actualización de pesos debería estar cargado siempre (pues de no estarlo, cuando `ip_vs_schedule()` invocase a la función de actualización, y esta no existiese en la tabla de símbolos, se produciría un error grave en el núcleo).

En una nueva iteración se solventó este problema, introduciendo una función como "trampolín" a la función de actualización de pesos:

- Dentro de `ip_vs_schedule()`, antes de invocar al algoritmo de balanceo, se invoca a la función trampolin denominada `before_sched_callback()`.
- En la función de inicialización de `ip_vs_core.c` se hace a `before_sched_callback()` apuntar a una función vacía, implementada en el mismo archivo (`ip_vs_dummy_callback()`). De esta forma, cuando el módulo de carga no esté cargado, el sistema puede funcionar normalmente.
- En la función de inicialización del módulo de carga (`ip_vs_load.c`), se hace a `before_sched_callback()` apuntar a la función de actualización de pesos (`ip_vs_load_callback()`).
- En la función de finalización de `ip_vs_load.c` (cuando ya no se desea utilizar más el módulo de cargas) se vuelve a hacer que `ip_vs_before_sched_callback()` apunte a `ip_vs_dummy_callback()`. De esta forma, se garantiza que cuando se deje de usar el módulo, el sistema pueda seguir funcionando.

La función `ip_vs_schedule()` requirió una modificación adicional: la ejecución de la rutina de actualización del módulo de balanceo (en el caso de existir). Para no ejecutar la rutina innecesariamente, se hizo uso de los valores de retorno de la función trampolín. Se utiliza el valor `1` como retorno de la función `ip_vs_dummy_callback()` (en cuyo caso no se ejecutaría la rutina de actualización) y `0` como valor de retorno de `ip_vs_load_callback()` (en cuyo caso se debe ejecutar la rutina de actualización). Véase la figura 4.3

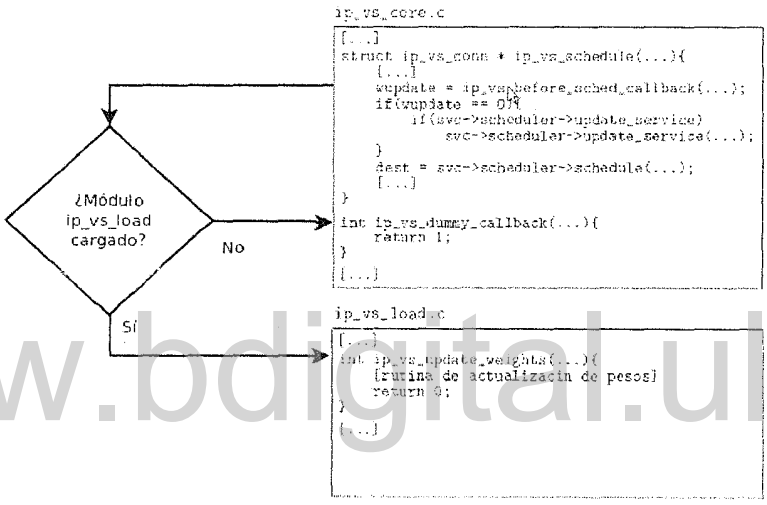


Figura 4.3: Implementación del componente de actualización de pesos

Función de actualización de pesos

Hasta este punto se han descrito las modificaciones realizadas al LVS para lograr invocar a una función de actualización de pesos (`ip_vs_load_callback()`), antes de que sea ejecutado el algoritmo de balanceo, garantizando así que este cuente con pesos que reflejen el estado actual de carga de cada uno de los servidores reales.

Esa función de actualización funciona de la siguiente manera: el módulo `ip_vs_load` mantiene una lista con las direcciones IP de los servidores reales, junto con sus valores de carga. Cada vez que el demonio de actualización de cargas en el nodo director

Carga	Peso
0	3
1	2
2	1
3 o más	0

Tabla 4.1: Esquema de asignación de pesos según la carga

escribe el archivo de cargas, `ip_vs_load` actualiza la lista con los nuevos valores. Así, la función de actualización de pesos sólo debe recorrer la lista de destinos del LVS y actualizar el peso, en función de su carga más reciente.

Función de transformación de cargas a pesos

El módulo de actualización de pesos debe encargarse de transformar los valores de carga leídos del archivo `/proc/loadrealsv` a pesos, tomando en cuenta que la relación entre estos valores es inversamente proporcional, es decir, mientras mayor sea la carga, menor será el peso que debe tener ese servidor.

El esquema de pesos utilizado, fue el mismo utilizado en el trabajo de [Guerrero, 2009], el cual consiste en una transformación discreta que se muestra en la tabla 4.1.

Función de inicialización

Durante la función de inicialización del módulo se invoca a la rutina de creación del archivo de cargas, se guarda una referencia a la función `ip_vs_before_sched_callback` en la variable `ip_vs_dummy_reference`, y se sobreescribe con la referencia a la función `ip_vs_load_callback`.

```
static int __init ip_vs_load_init(void)
{
    create_proc_file();
    ip_vs_dummy_reference = ip_vs_before_sched_callback;
    ip_vs_before_sched_callback = &ip_vs_load_callback;
    return 0;
}
```

Función de finalización

Durante la función de finalización se elimina el archivo de cargas y se restaura la función `ip_vs_before_sched_callback` con la función original (a la cual se guardó una referencia en la función de inicialización en la variable `ip_vs_dummy_reference`).

```
static void __exit ip_vs_load_cleanup(void)
{
        remove_proc_entry(PROC_FILE_NAME, NULL);
        ip_vs_before_sched_callback = ip_vs_dummy_reference;
}
```


Con esta función, se concluyen los detalles de implementación del sistema; en los apéndices A y B se encuentra la totalidad del código fuente, así como la descripción de cada uno de los archivos generados y se ejemplifica el uso de las herramientas de prueba. En el siguiente capítulo se presenta el trabajo experimental realizado, describiendo desde la preparación de la plataforma de hardware, hasta el análisis de resultados.

Capítulo 5

Pruebas y análisis de resultados

Como se describió en el marco metodológico, una vez concluída la fase de diseño de los componentes de software, se procedió a evaluar el impacto de la solución propuesta, realizando comparaciones de rendimiento con respecto a un sistema de balanceo de carga basado en el Servidor Virtual de Linux sin modificar. Las medidas de desempeño consideradas son las siguientes:

1. Tasa promedio de respuesta (peticiones/segundo).
2. Tiempo promedio de respuesta (segundos).

A continuación se describe la plataforma de hardware y software utilizada, y posteriormente se presentan los resultados de la experimentación, agrupado en dos secciones: una primera sección correspondiente a pruebas remotas (fuera de la red en la cual se encuentra el sistema LVS), y una segunda sección de pruebas realizadas dentro de la misma red del cluster de balanceo. En ambos grupos de experimentos, se aplicó el procedimiento de 5 etapas descrito en el marco metodológico.

5.1. Configuración del sistema LVS

El sistema de balanceo de carga fue implementado en el Laboratorio del Postgrado en Computación de la Universidad de Los Andes, en Mérida, Venezuela.

Tanto para el nodo director como para los servidores reales se utilizaron equipos con las mismas características de Hardware: procesador Intel Pentium 4 de 2.8Ghz de velocidad, 512MB de memoria RAM. El sistema operativo utilizado fue Ubuntu 12.04, con Kernel 3.2.1.

5.1.1. Configuración del nodo director

El nodo director se configuró con un kernel 3.2.1 modificado para incluir el módulo de manejo de carga desarrollado en esta monografía.

Se instaló el paquete ipvsadm versión 1.25 para la administración del LVS.

5.1.2. Configuración de los servidores reales

El servidor web utilizado fue Apache versión 2.2.22. Adicionalmente se instaló Stress versión 1.0.4 para aplicar sobrecarga (en las pruebas que lo requirieron).

5.1.3. Configuración de red

La conexión de los equipos (nodo director y servidores reales) dentro del Laboratorio se realiza mediante un equipo conmutador (switch), que constituye además el punto de conexión a la Internet. La topología de esta red es de tipo estrella simple. El conmutador utilizado es modelo 3Com 4228G.

Las características del sistema LVS son las siguientes:

Equipo	Dirección IP
Director	150.185.182.60/24
Servidor 1	150.185.182.86/24
Servidor 2	150.185.182.87/24
Servidor 3	150.185.182.90/24
VIP	150.185.182.69/24

Tabla 5.1: Direcciones IP utilizadas por el sistema LVS

- Topología de red: basada en una sola red (flat-based).
- Conectividad física: Una conexión (one-armed).
- Técnica de reenvío de paquetes: Enrutamiento directo (direct routing).

Se utilizaron direcciones IP públicas para el nodo director, para los servidores reales y para la IP virtual, dentro del rango 150.185.182.0/24, de acuerdo a la tabla 5.1:

En la figura 5.1 se muestra la configuración de red utilizada.

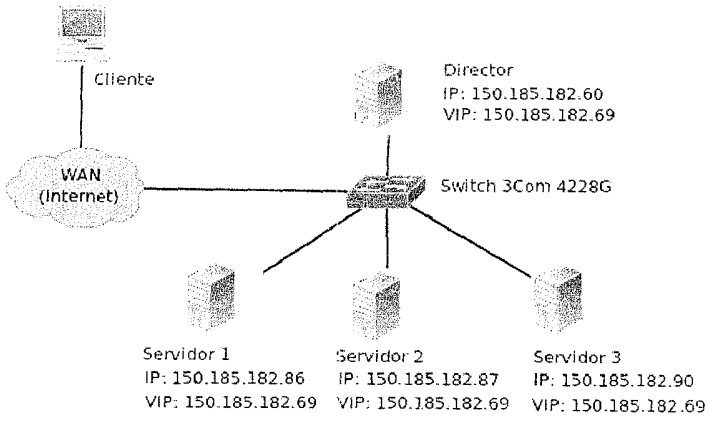


Figura 5.1: Configuración de red utilizada para las pruebas

Para solventar el problema ARP, originado por el uso de la técnica de Enrutamiento directo para el reenvío de paquetes, se siguió el enfoque de uso de políticas de respuesta ARP, descrito en el capítulo 2.

5.2. Pruebas desde clientes remotos

Estas pruebas se realizaron desde clientes remotos, ubicados físicamente en la ciudad de Barquisimeto, edo. Lara, utilizando una conexión a Internet de 1024Kbps (subida)/512Kbps (bajada).

Las características de los equipos utilizados como clientes para estas pruebas son las siguientes: procesador Intel Pentium Dual Core de 2.8Ghz de velocidad, 2GB de memoria RAM, sistema operativo Ubuntu 12.04 con Kernel 3.2.1.

5.2.1. Descripción de las pruebas

Todas las pruebas fueron automatizadas, utilizando:

- Httperf versión 0.9.0
- Autobench versión 2.1.2

Autobench fue configurado para realizar 2 tipos de pruebas de carga, diferenciados por el número total de peticiones:

- Prueba tipo 1:
  - Número total de peticiones por experimento: 1000
  - Rango de tasa de peticiones/segundo: [20, 130]
  - Razón de incremento de peticiones en cada paso: 10

Según estos parámetros, se comienza realizando 20 peticiones/segundo hasta alcanzar 1000 peticiones, luego se incrementa a 30 peticiones/segundo hasta lograr las 1000 peticiones, y así sucesivamente hasta completar 1000 peticiones a razón de 130 peticiones/segundo.

- Prueba tipo 2:
  - Número total de peticiones por experimento: 2000
  - Rango de tasa de peticiones/segundo: [20, 130]
  - Razón de incremento de peticiones en cada paso: 10

  La única diferencia con el caso anterior es que se realizan 2000 peticiones por cada experimento, en vez de 1000.

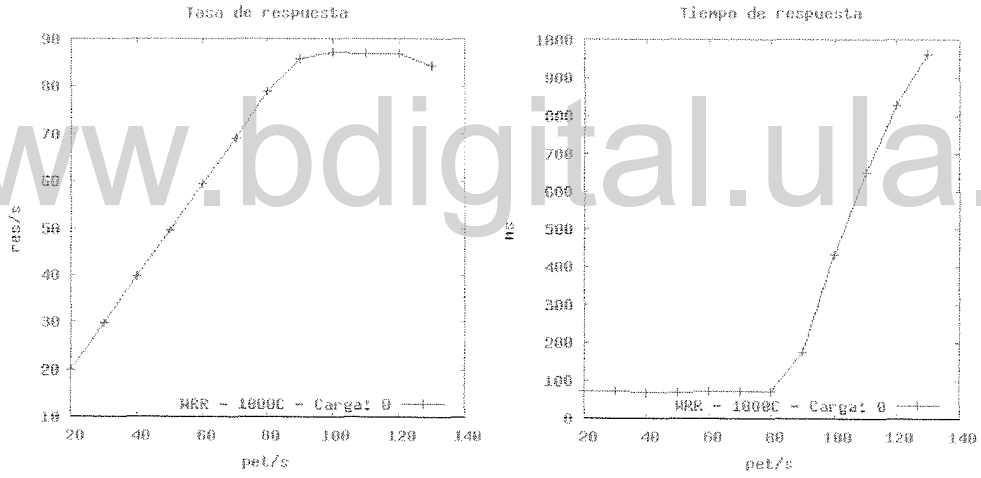
Los algoritmos de balanceo seleccionados para las pruebas fueron el algoritmo por Round-Robin ponderado (WRR) y el algoritmo por menor número de conexiones ponderado (WLC). Estos algoritmos fueron seleccionados, ya que hacen uso de los pesos de los servidores reales, lo cual es necesario para llevar a cabo comparaciones entre el uso de estos pesos asignados estáticamente y el uso de pesos asignados dinámicamente en función de su carga.

Se realizaron 3 réplicas de cada prueba y se graficaron los promedios de tasa de respuesta y de tiempo de respuesta, en cada caso.

5.2.2. Pruebas utilizando algoritmos clásicos sin servidores cargados

Siguiendo el marco metodológico experimental de 5 etapas, descrito en el capítulo 3, el primer conjunto de experimentos aplicados estuvo orientado a estudiar la respuesta del sistema LVS en condiciones naturales, es decir, sin sobrecargar ninguno de los servidores reales, ni realizar la actualización dinámica de pesos.

En las figuras 5.2 y 5.3 se muestran los resultados obtenidos realizando un total de 1000 conexiones por paso (Prueba tipo 1), tanto para algoritmo Round-Robin ponderado como para el algoritmo por menor número de conexiones ponderado. Se puede observar un comportamiento similar en ambos casos, manteniendo la tasa de respuesta un comportamiento lineal hasta aproximadamente 80 peticiones/seg, con tiempos de respuesta por debajo de los 100ms. A partir de este punto, se comienza a apreciar la saturación que impide un crecimiento lineal en la tasa de respuesta, alcanzando un tope entre las 80 respuestas/segundo y 90 respuestas por segundo. También se observa como a partir del punto de saturación, los tiempos de respuesta comienzan a crecer exponencialmente, sobrepasando los 500ms al llegar a 130 peticiones/segundo, en ambos casos.



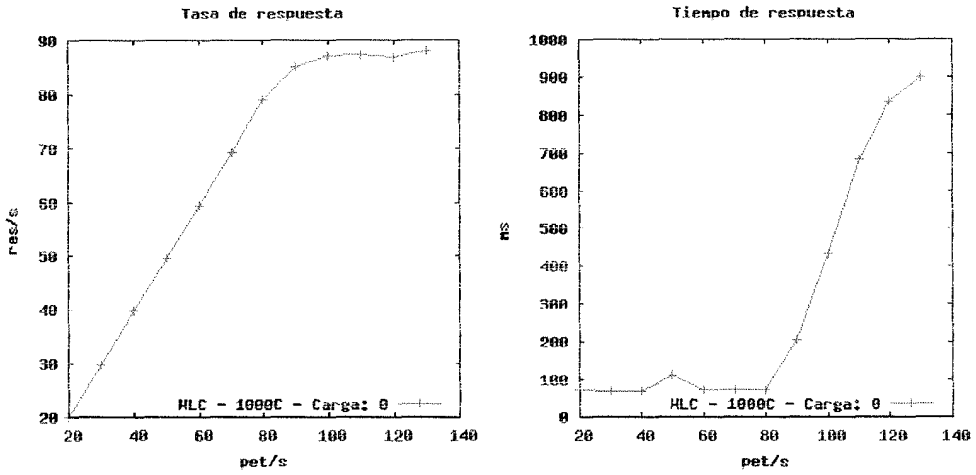


Figura 5.3: Algoritmo: WLC. 1000 conexiones/paso: sin actualización dinámica

En las figuras 5.4 y 5.5 se muestran los resultados obtenidos realizando un total de 2000 conexiones por paso (Prueba tipo 2), para ambos algoritmos. Se observa un comportamiento similar a los anteriores en cuanto a tasa de respuesta, creciendo de forma lineal hasta las 80 peticiones/segundo, y alcanzando máximos cercanos a las 90 respuestas/segundo. También se observa un crecimiento exponencial en los tiempo de respuesta, pero en este caso, ligeramente superior, debido a que en cada paso se realizan un mayor número de conexiones (2000 conexiones).

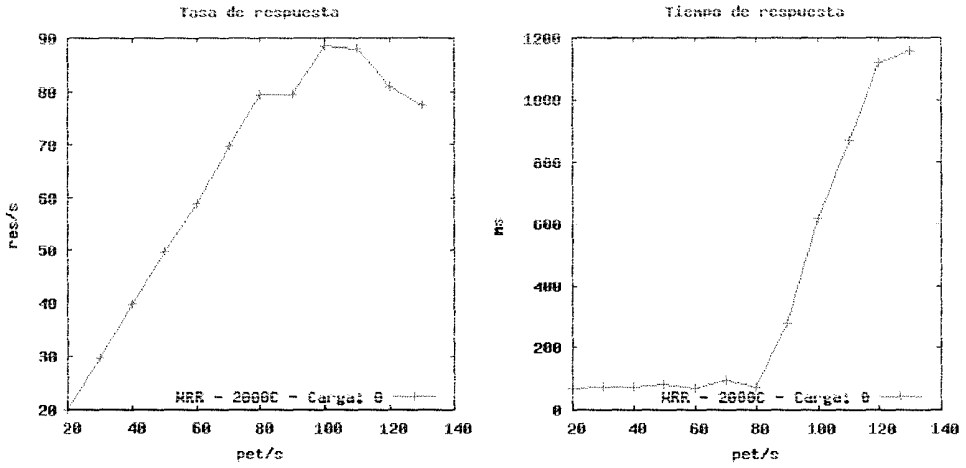


Figura 5.4: Algoritmo: WRR. 2000 conexiones/paso: sin actualización dinámica

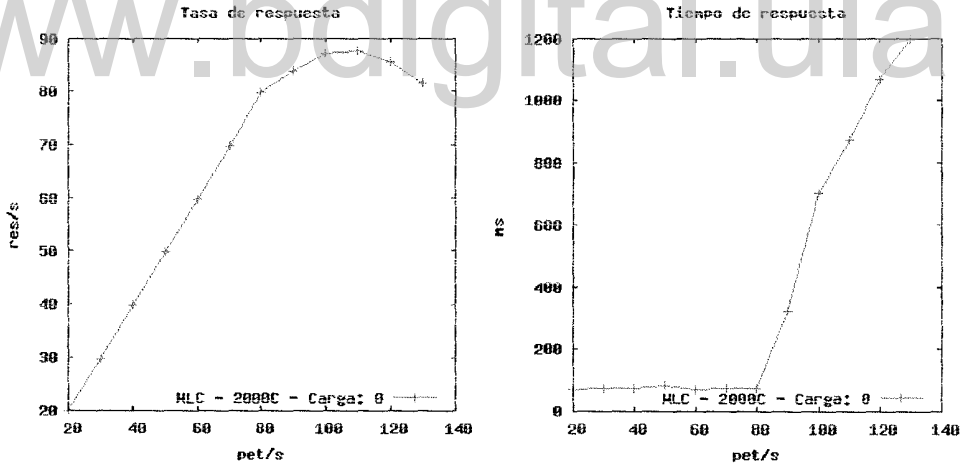


Figura 5.5: Algoritmo: WLC. 2000 conexiones/paso: sin actualización dinámica

5.2.3. Pruebas utilizando actualización dinámica de pesos sin servidores cargados

Continuando con la metodología experimental, el siguiente conjunto de experimentos, correspondientes a la etapa 2, tuvieron como objetivo el estudio del sistema LVS en condiciones naturales (sin sobrecargar ninguno de los servidores reales), una vez introducido el módulo de actualización dinámica de pesos.

Se realizaron los mismos experimentos descritos en la sección anterior, pero ahora utilizando la actualización dinámica de pesos.

En las figuras 5.6 y 5.7 se muestran los resultados obtenidos con 1000 conexiones por paso (Prueba tipo 1), tanto para algoritmo Round-Robin ponderado como para el algoritmo por menor número de conexiones ponderado.

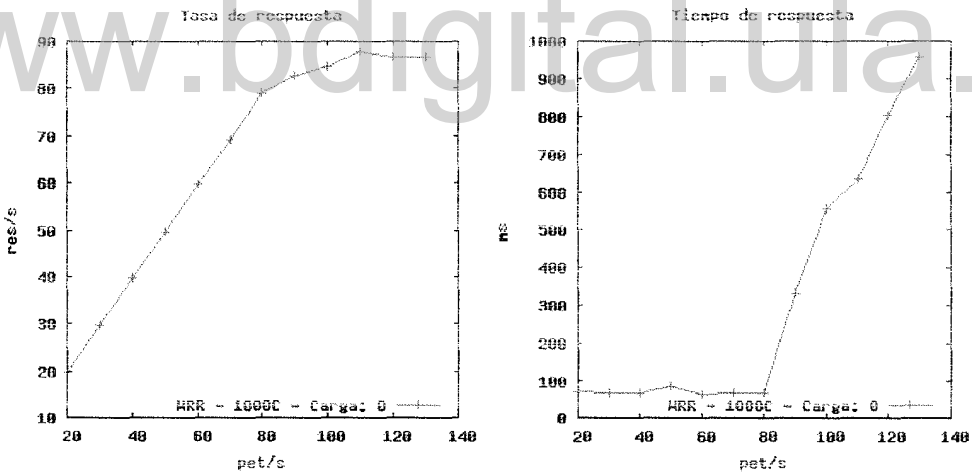


Figura 5.6: Algoritmo WRR, 1000 conexiones/paso con actualización dinámica

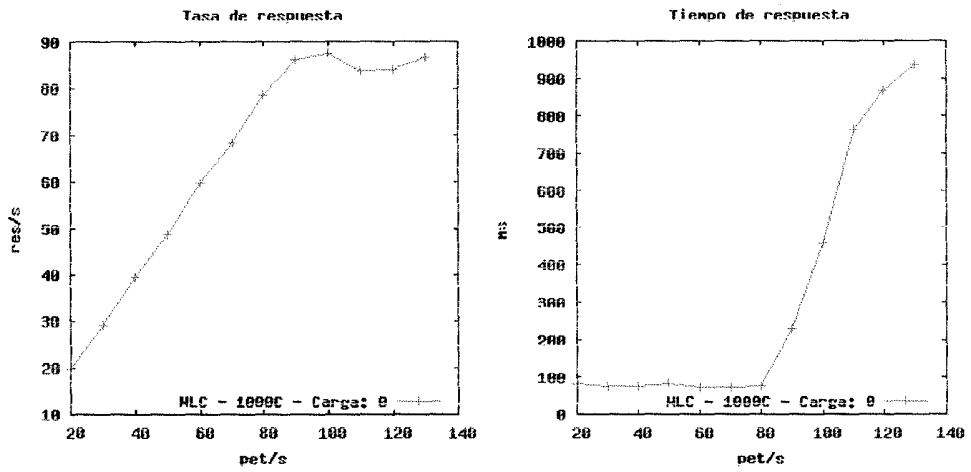


Figura 5.7: Algoritmo: WLC. 1000 conexiones/paso: con actualización dinámica

El comportamiento es similar al obtenido anteriormente, en ausencia del módulo de actualización dinámica de pesos: respuesta lineal hasta el punto de saturación (aproximadamente en 80 peticiones/segundo), alcanzando máximos cercanos a las 90 respuestas/segundo, y tiempos de respuesta con crecimiento exponencial a partir del punto de saturación, oscilando cerca de los 1000ms en el último punto de prueba (130 peticiones/segundo).

Resultados similares se obtuvieron en el caso de 2000 conexiones por paso (Prueba tipo 2), los cuales se anexan a este documento.

5.2.4. Pruebas utilizando algoritmos clásicos con servidores cargados

La siguiente etapa del marco experimental (etapa 3 de la metodología experimental) consistió en estudiar el comportamiento del sistema LVS original, aplicando sobrecarga a los servidores reales.

Los experimentos aplicados fueron los mismos de las etapas anteriores, pero primero aplicando sobrecarga a uno de los servidores reales, luego a dos de los servidores reales, y finalmente a los 3 servidores reales.

En las figuras 5.8 y 5.9 se presentan los resultados obtenidos con 1000 conexiones por paso (Prueba tipo 1), tanto para algoritmo Round-Robin ponderado como para el algoritmo por menor número de conexiones ponderado. Como referencia, en estas gráficas se muestran también los resultados presentados previamente con servidores no cargados (Carga 0).

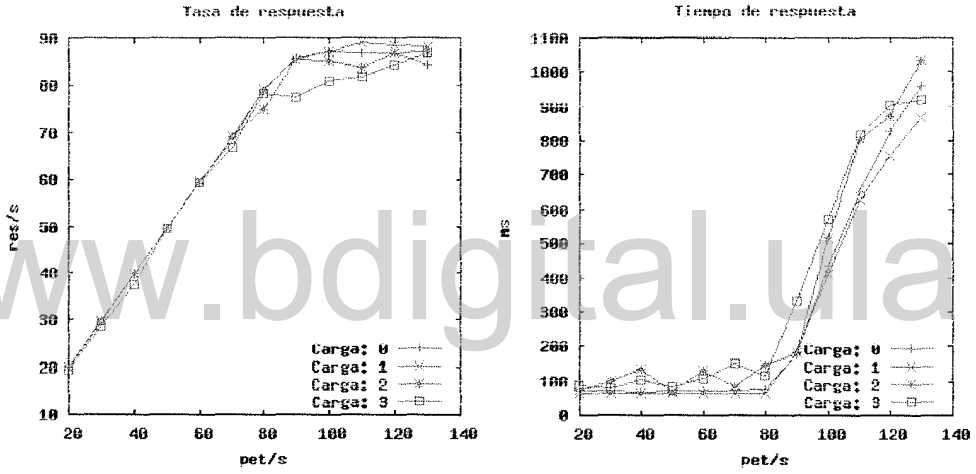


Figura 5.8: Algoritmo: WRR. 1000 conexiones/paso: sin actualización dinámica

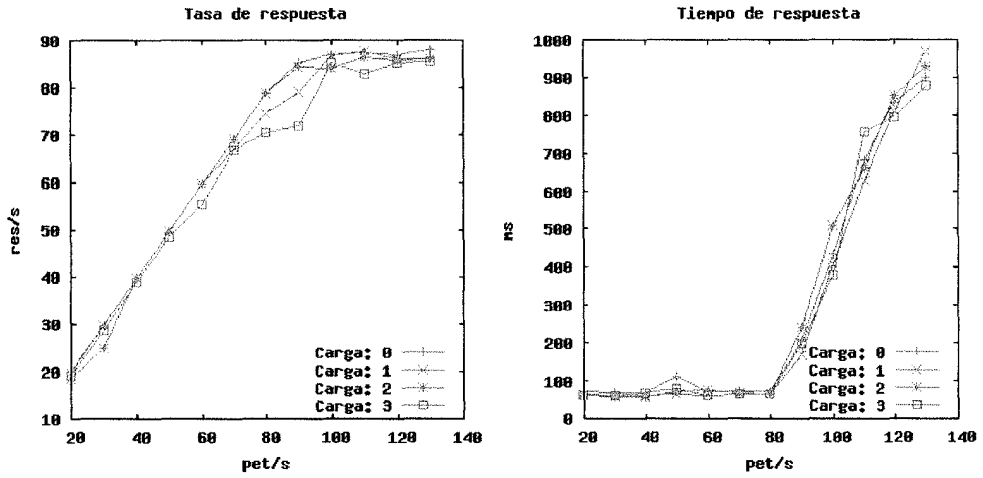


Figura 5.9: Algoritmo: WLC. 1000 conexiones/paso; sin actualización dinámica

5.2.5. Pruebas utilizando actualización dinámica de pesos con servidores cargados

La cuarta etapa de la metodología experimental consistió en la repetición de las pruebas de la tercera etapa, pero con el módulo de actualización dinámica de pesos funcionando.

En las figuras 5.10 y 5.11 se presentan los resultados obtenidos con 1000 conexiones por paso (Prueba tipo 1), tanto para algoritmo Round-Robin ponderado como para el algoritmo por menor número de conexiones ponderado. Como referencia, en estas gráficas se muestran también los resultados presentados previamente con servidores no cargados (Carga 0).

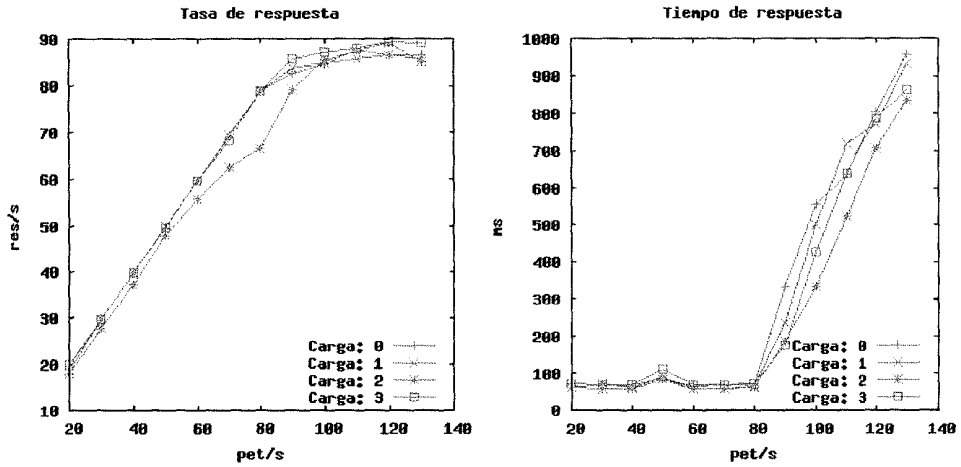


Figura 5.10: Algoritmo: WRR. 1000 conexiones/paso; con actualización dinámica

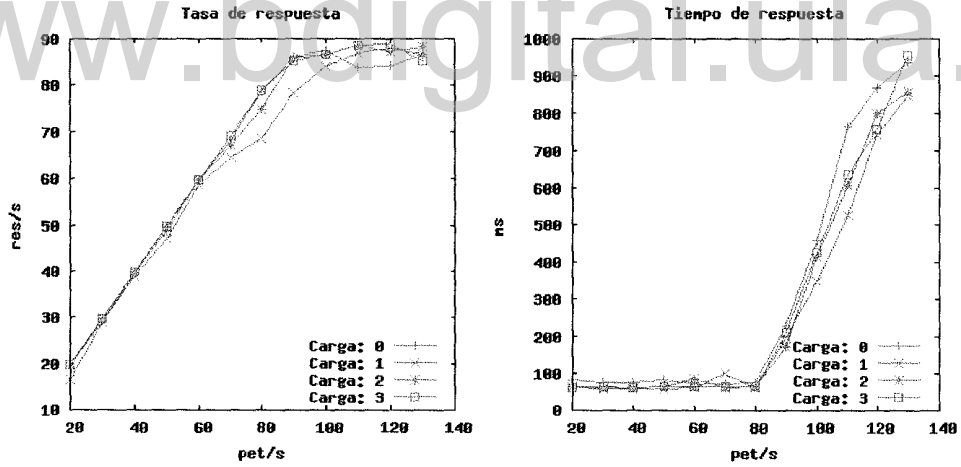


Figura 5.11: Algoritmo: WLC. 1000 conexiones/paso; con actualización dinámica

5.2.6. Comparación de resultados

Finalmente, una vez terminadas todas las pruebas correspondientes a este grupo de experimentos, como etapa final de la metodología, se compararon los resultados obtenidos utilizando el sistema LVS original, con los obtenidos cuando se utilizó el mecanismo de actualización dinámica de pesos.

En los gráficos siguientes se compara el desempeño del LVS original con el desempeño del LVS con actualización dinámica de pesos, tanto en ausencia de carga (Servidores cargados: 0), como en presencia de carga (Servidores cargados: 1, 2 o 3), utilizando los resultados ya presentados de forma separada.

Sin servidores cargados

Los resultados muestran que cuando ningún servidor se encuentra sobrecargado, el LVS mantiene el mismo comportamiento para los dos algoritmos de balanceo estudiados (Round-Robin ponderado y Menor número de conexiones ponderado).

En las figuras 5.12 y 5.13 se comparan los resultados obtenidos para el algoritmo de balanceo por menor número de conexiones ponderado, para 1000 conexiones y para 2000 conexiones respectivamente.

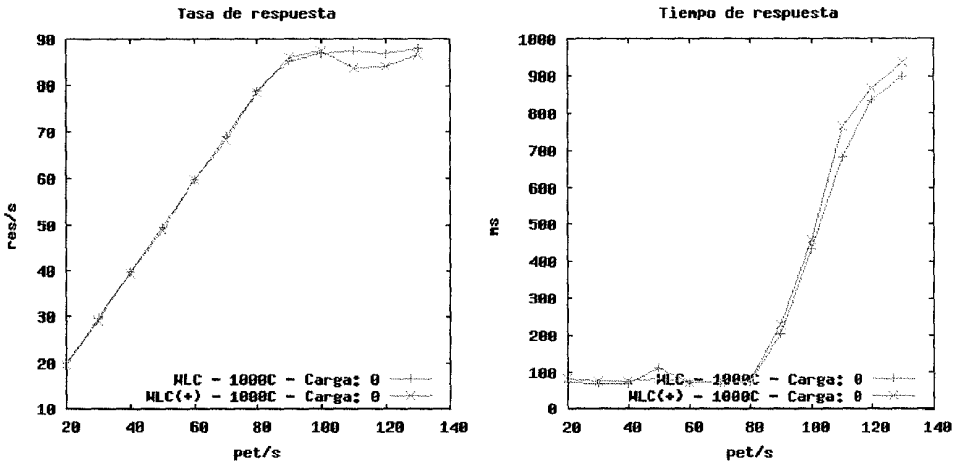


Figura 5.12: Algoritmo: WLC. 1000 conexiones/paso; sin servidores cargados

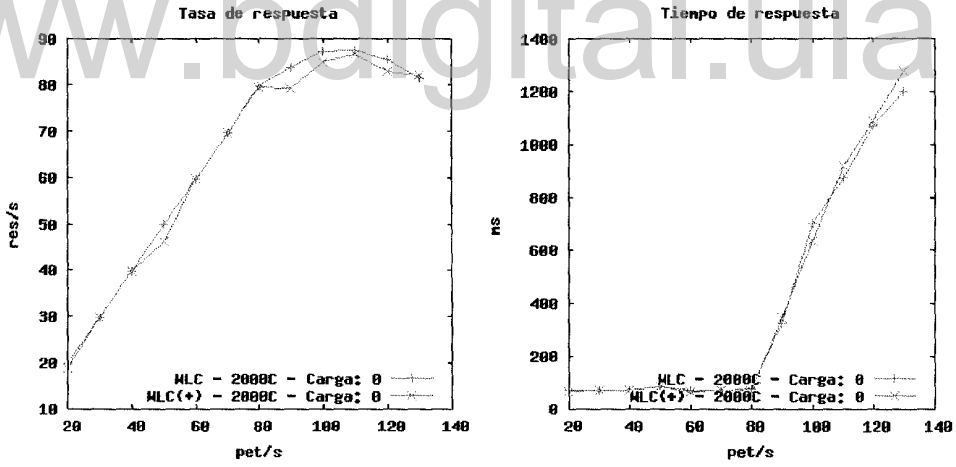


Figura 5.13: Algoritmo: WLC. 2000 conexiones/paso; sin servidores cargados

Con servidores cargados

En las figuras 5.14 y 5.15 se comparan los resultados obtenidos para el algoritmo de balanceo por menor número de conexiones ponderado, para 1000 conexiones y para 2000 conexiones respectivamente, con un servidor cargado.

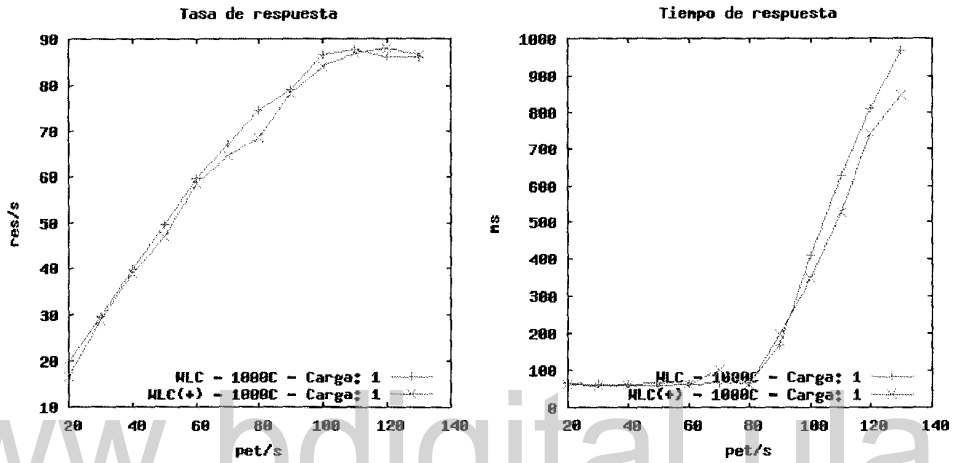


Figura 5.14: Algoritmo: WLC. 1000 conexiones/paso; con 1 servidor cargado

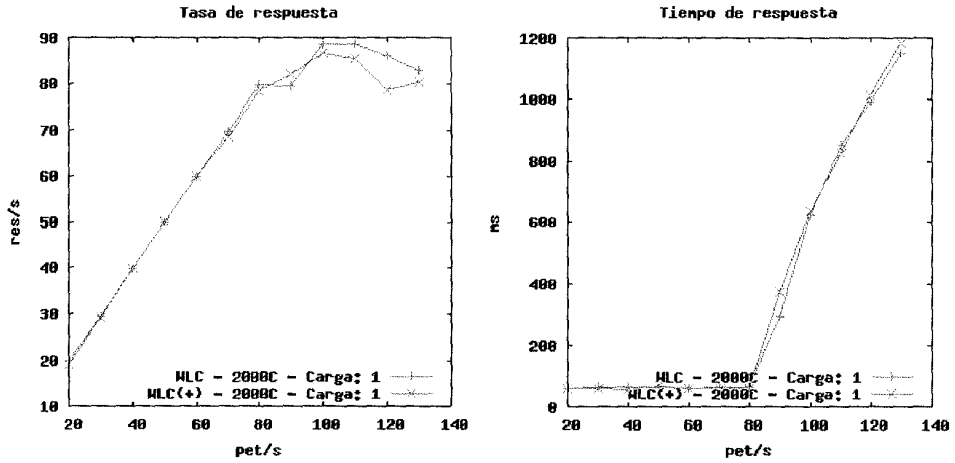


Figura 5.15: Algoritmo: WLC. 2000 conexiones/paso; con 1 servidor cargado

En las figuras 5.16 y 5.17 se comparan los resultados obtenidos para el algoritmo de balanceo por menor numero de conexiones ponderado y para el algoritmo por Round-Robin ponderado, para 1000 conexiones en ambos casos, y con dos servidores cargados.

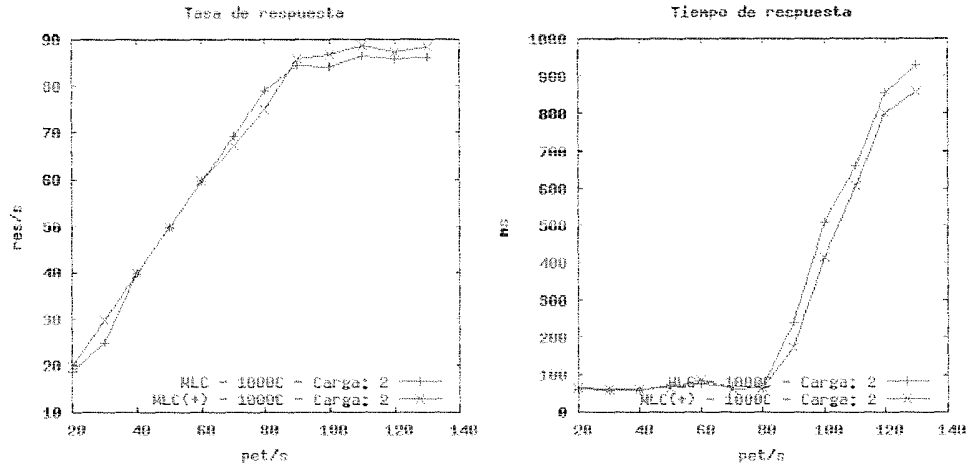


Figura 5.16: Algoritmo: WLC, 1000 conexiones/paso: con 2 servidores cargados

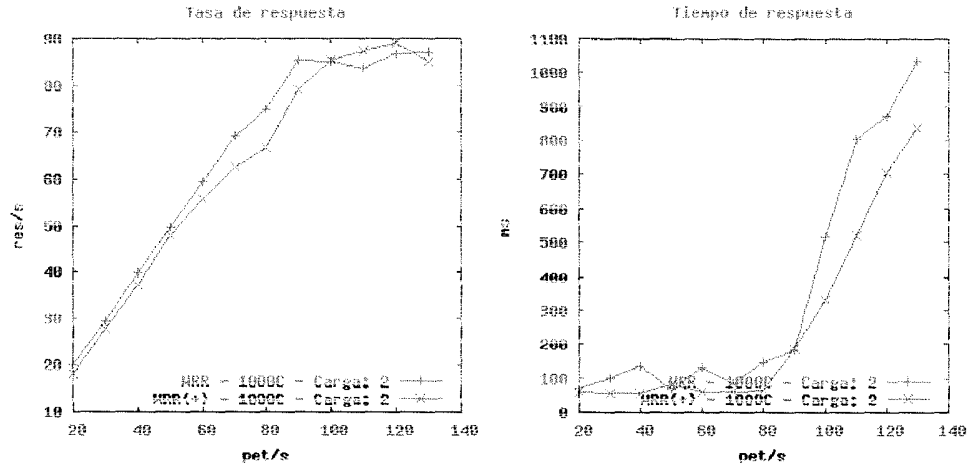


Figura 5.17: Algoritmo: WRR, 1000 conexiones/paso: con 2 servidores cargados

En las figuras 5.18 y 5.19 se comparan los resultados obtenidos para el algoritmo de balanceo por menor número de conexiones ponderado y para el algoritmo por Round-Robin ponderado, para 1000 conexiones en ambos casos, y con 3 servidores cargados.

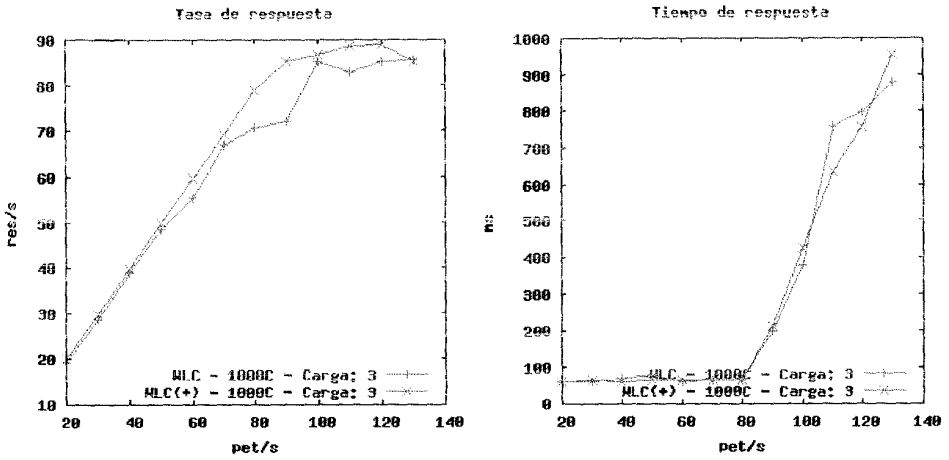


Figura 5.18: Algoritmo: WLC. 1000 conexiones/paso; con 3 servidores cargados

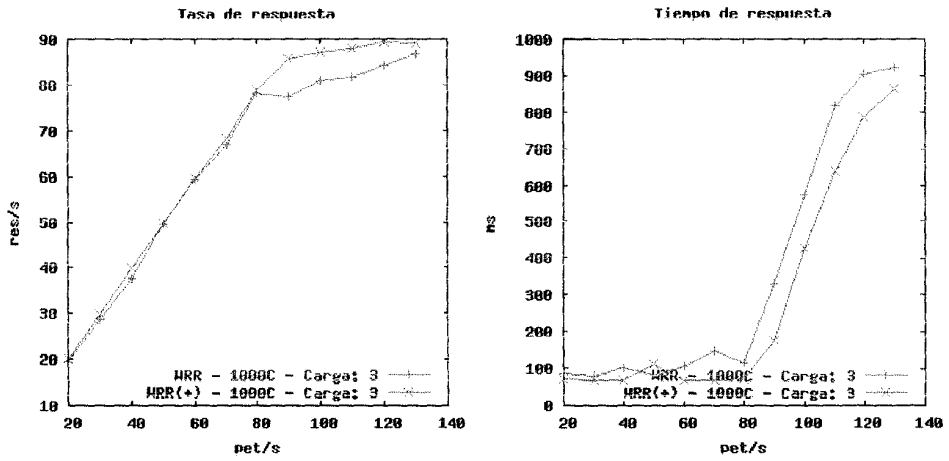


Figura 5.19: Algoritmo: WRR. 1000 conexiones/paso; con 3 servidores cargados

5.3. Pruebas desde la red interna

Una vez finalizadas las pruebas utilizando clientes remotos, se decidió realizar un conjunto de pruebas, ubicando los clientes dentro de la red interna de la Universidad de Los Andes. Para ello, se ubicaron los clientes en la sede del Centro Nacional de Cálculo Científico (CeCalCULA), en el Parque Tecnológico de Mérida.

La motivación principal para realizar este conjunto de pruebas, fue el poder realizar un número más alto de peticiones, sin saturar el ancho de banda de los clientes.

En este nuevo conjunto de experimentos también se siguió la metodología de 5 etapas descrita en el marco metodológico. Los experimentos realizados, fueron similares a los descritos en la sección anterior, pero en este caso realizando un número mayor de peticiones. Se utilizaron los mismos algoritmos estudiados en la sección anterior (Round-Robin ponderado y Menor número de conexiones ponderado).

5.3.1. Descripción de las pruebas

Estas pruebas de carga fueron automatizadas con las mismas herramientas utilizadas para las pruebas desde clientes remotos:

- Httperf versión 0.9.0
- Autobench versión 0.1.9

Las características de cada prueba de carga son las siguientes:

- Número total de peticiones por experimento: 9000
- Rango de tasa de peticiones/segundo: [100, 300]
- Razón de incremento de conexiones en cada paso: 20

De acuerdo a estos parámetros, en cada prueba se comienza realizando 100 conexiones/segundo hasta completar 5000 conexiones, luego se realizan 125 conexiones/segundo hasta completar nuevamente 5000 conexiones, y así sucesivamente hasta completar 5000 conexiones a razón de 300 conexiones/segundo.

Al igual que con los clientes remotos, se realizaron 3 réplicas de cada prueba.

5.3.2. Pruebas utilizando algoritmos clásicos sin servidores cargados

A continuación se presentan los resultados correspondientes a la etapa 1 de la métodología experimental, referente a la caracterización de la respuesta utilizando el sistema LVS original.

En las figuras 5.20 y 5.21 se presentan los resultados obtenidos para el algoritmo de balanceo por menor número de conexiones ponderado y para el algoritmo por Round-Robin ponderado, sin servidores cargados.

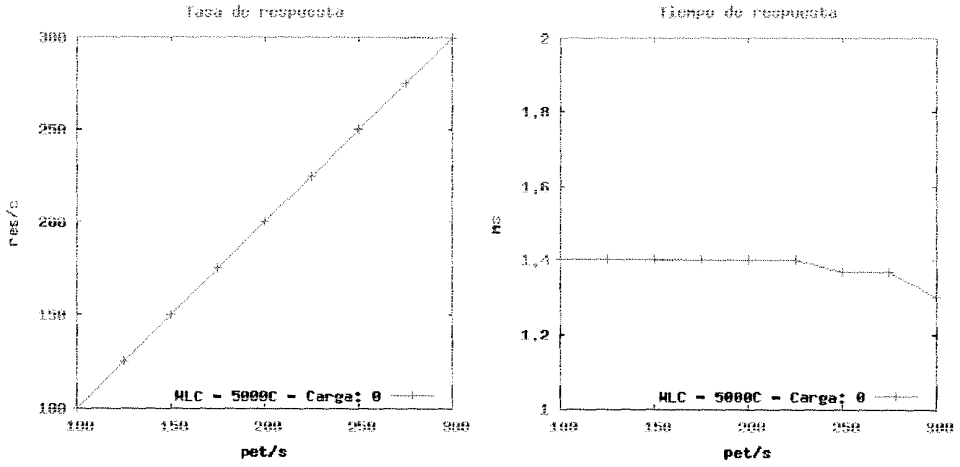


Figura 5.20: Algoritmo: WLC. 5000 conexiones/paso; sin actualización dinámica

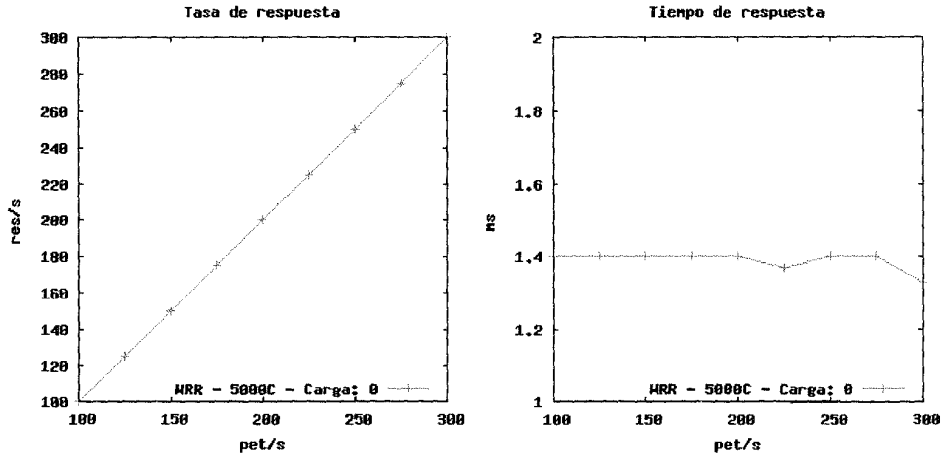


Figura 5.21: Algoritmo: WRR. 5000 conexiones/paso; sin actualización dinámica

En ambos casos, el sistema es capaz de responder el 100 % de las peticiones recibidas y los tiempos de respuesta oscilan alrededor de 1.4 sengundos.

5.3.3. Pruebas utilizando actualización dinámica de pesos sin servidores cargados

Siguiendo con la metodología experimental, la segunda etapa corresponde al estudio del sistema sin servidores sobrecargados, una vez introducido el componente de actualización dinámica de pesos.

Se realizaron los mismos experimentos descritos en la sección anterior, pero ahora utilizando el módulo de actualización dinámica de pesos. Los resultados obtenidos se presentan en las figuras 5.22 y 5.23.

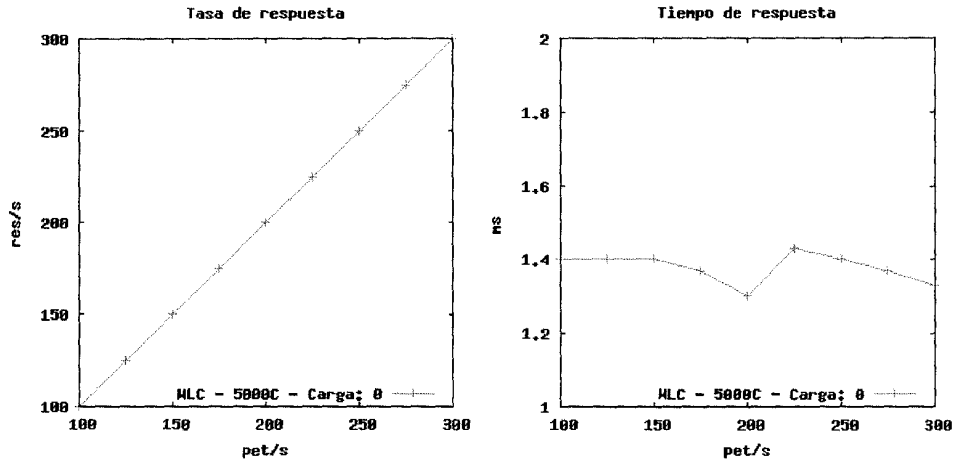


Figura 5.22: Algoritmo: WLC. 5000 conexiones/paso; con actualización dinámica

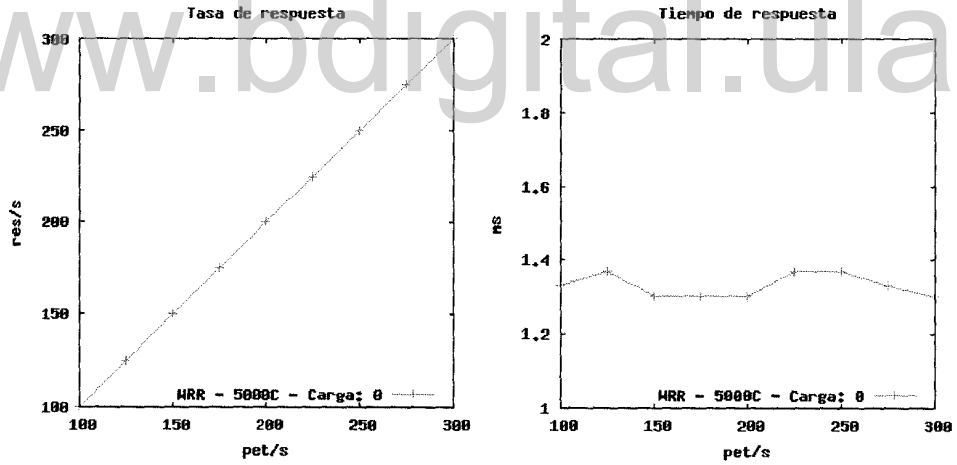


Figura 5.23: Algoritmo: WRR. 5000 conexiones/paso; con actualización dinámica

Se observan resultados similares a los obtenidos cuando no se utilizó la actualización dinámica de pesos: el sistema está en capacidad de responder todas las peticiones recibidas, y los tiempos de respuesta oscilan alrededor de 1.4 segundos, inclusive cuando el número de peticiones alcanca el máximo valor probado (300 peticiones/segundo).

5.3.4. Pruebas utilizando algoritmos clásicos con servidores cargados

A continuación se presentan los resultados obtenidos en la tercera etapa de la metodología experimental, en la cual se busca estudiar el comportamiento del sistema LVS original cuando se sobrecargan los servidores reales.

En las figuras 5.24 y 5.25 se presentan los resultados obtenidos para los algoritmos de balanceo por Round-Robin ponderado y por menor número de conexiones ponderado, sin actualización dinámica de pesos, cuando los servidores fueron sobrecargados (Carga 1: un servidor cargado; Carga 2: dos servidores cargados; Carga 3: tres servidores cargados). En las gráficas se repiten los resultados presentados previamente, en los que ningún servidor fue sometido a carga externa (Carga 0).

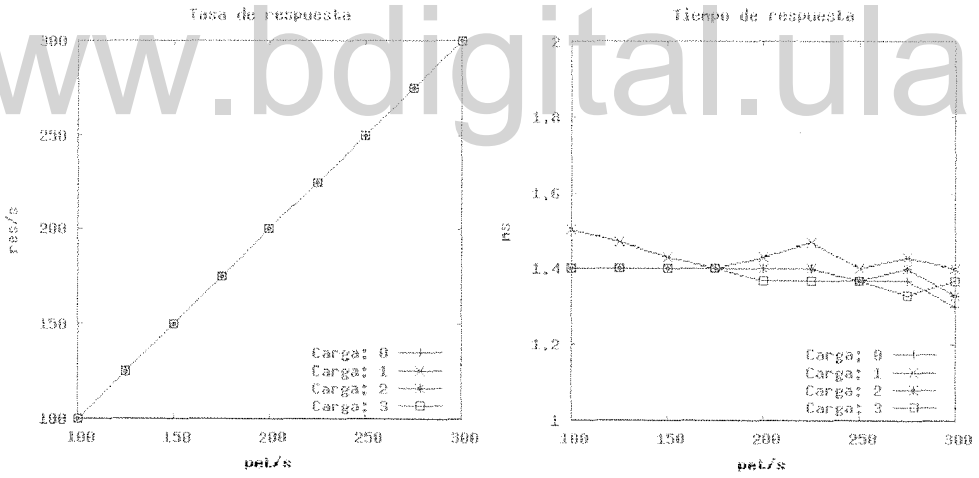


Figura 5.24: Algoritmo: WLC. 5000 conexiones/base. Sin actualización dinámica

carga de los servidores reales al proceso de balanceo

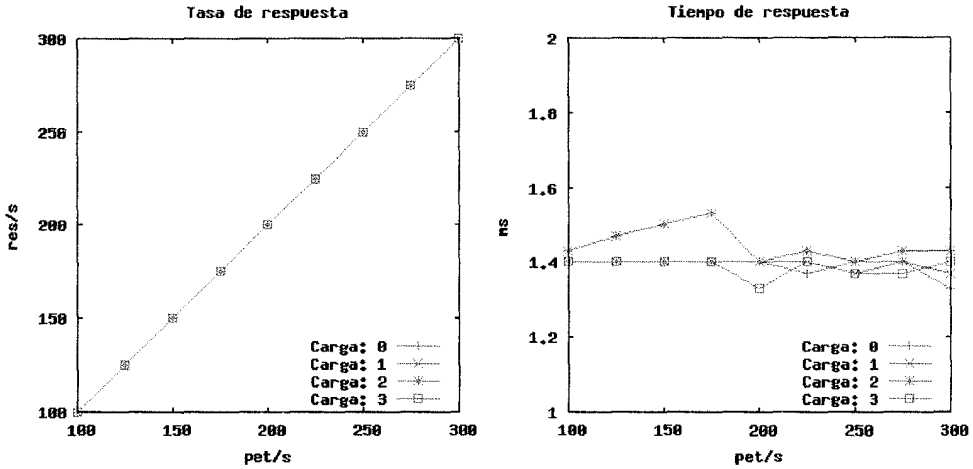


Figura 5.25: Algoritmo: WRR. 5000 conexiones/paso; sin actualización dinámica

5.3.5. Pruebas utilizando actualización dinámica de pesos con servidores cargados

En la cuarta etapa de la metodología experimental, se repitieron los experimentos realizados en la etapa anterior, pero ahora utilizando el módulo de actualización dinámica de pesos.

En las figuras 5.26 y 5.27 se presentan los resultados obtenidos para los algoritmos de balanceo por Round-Robin ponderado y por menor número de conexiones ponderado, con actualización dinámica de pesos, cuando los servidores fueron sobrecargados (Carga 0: ningún servidor cargado; Carga 1: un servidor cargado; Carga 2: dos servidores cargados; Carga 3: tres servidores cargados).

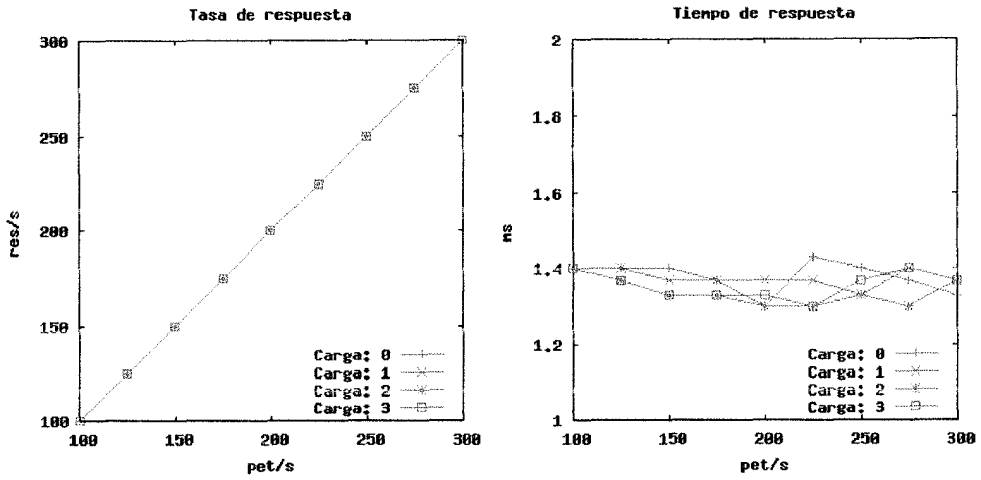


Figura 5.26: Algoritmo: WLC. 5000 conexiones/paso; con actualización dinámica

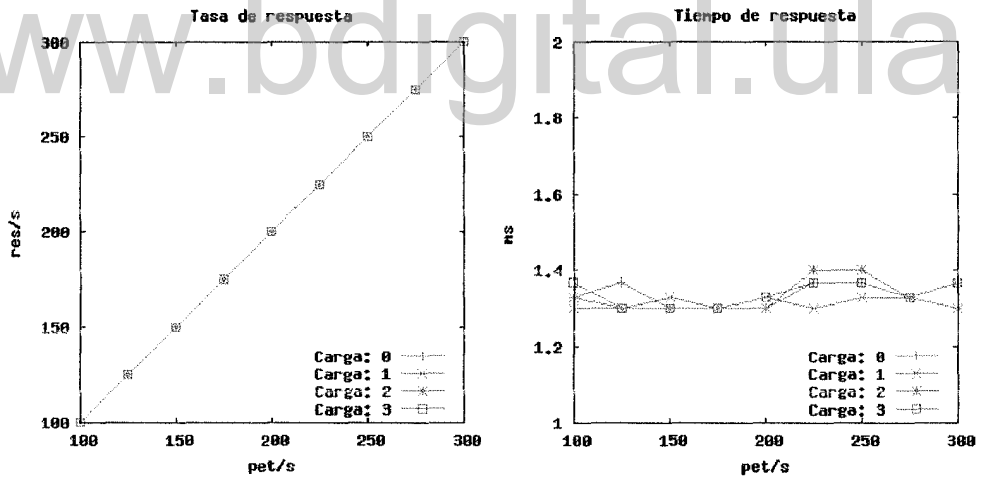


Figura 5.27: Algoritmo: WRR. 5000 conexiones/paso; con actualización dinámica

5.3.6. Comparación de resultados

Como etapa final de la metodología experimental, al igual que se hizo con las pruebas desde clientes remotos, se compararon los resultados obtenidos utilizando el sistema LVS original, con los obtenidos cuando se utilizó el mecanismo de actualización dinámica de pesos.

Las comparaciones se realizan, en terminos del número de servidores sobrecargados, y se presentan en dos partes: una primera parte correspondiente a las pruebas en que no se aplicó ningún tipo de sobrecarga (es decir, servidores sobrecargados: 0), y una segunda parte correspondiente a las pruebas en las que se sobrecargaron los servidores (servidores sobrecargados: 1, 2 o 3).

Sin servidores cargados

En las figuras 5.28 y 5.29 se comparan los resultados obtenidos para los algoritmos de balanceo por menor número de conexiones ponderado y por Round-Robin ponderado, respectivamente.

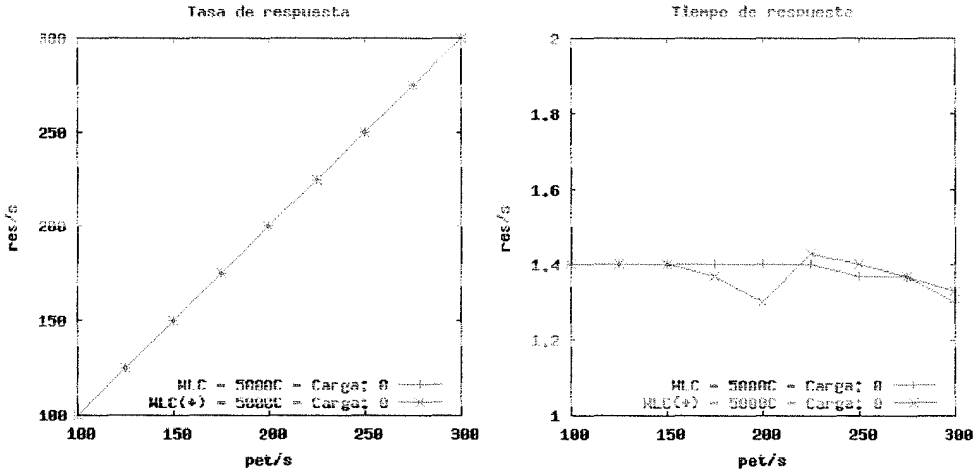


Figura 5.28. Algoritmo: WLC. 5000 conexiones/paso, sin servidores cargados

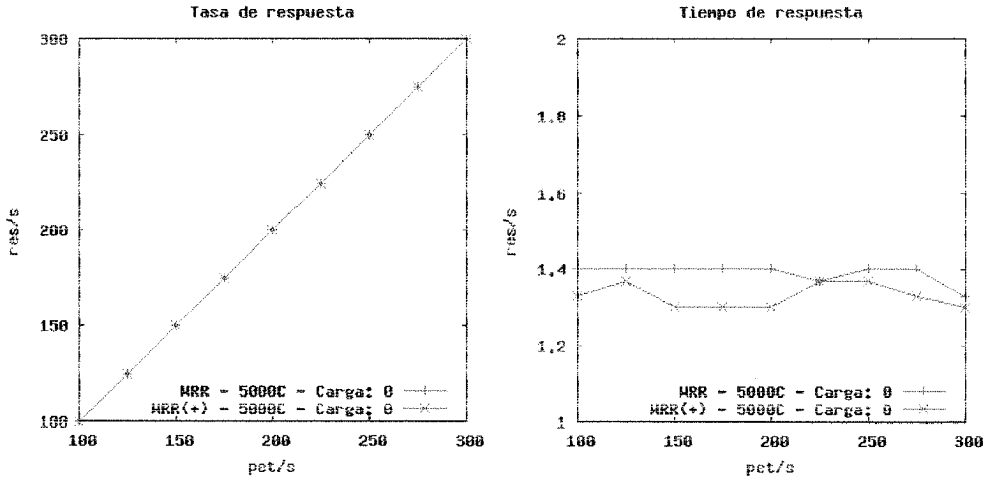


Figura 5.29: Algoritmo: WRR. 5000 conexiones/paso; sin servidores cargados

Como ya se observó en todos los resultados presentados por separado, la capacidad de respuesta es del 100 % para el sistema original y para el sistema con actualización dinámica de pesos. En estas dos gráficas se observan además tiempos de respuesta similares, casi constantes, alrededor de 1.4 segundos.

Con servidores cargados

En la figura 5.30 se comparan los resultados obtenidos para el algoritmos de balanceo por Round-Robin ponderado, con un (1) servidor cargado.

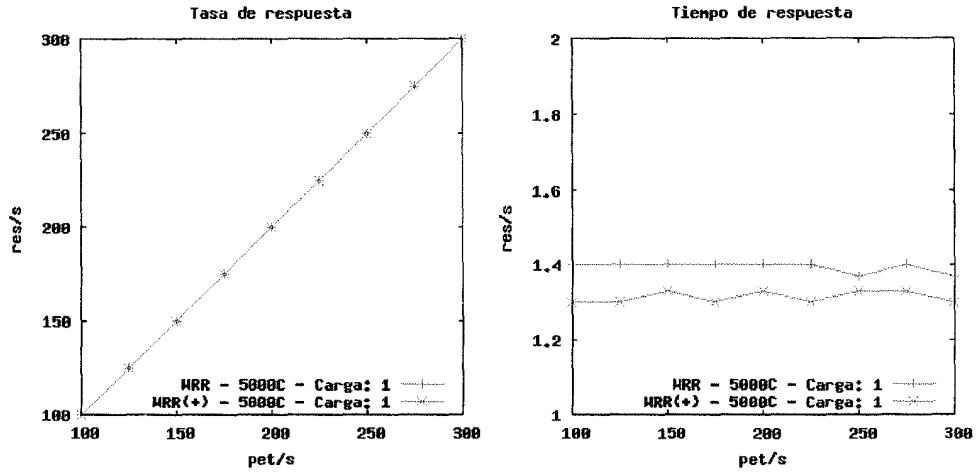


Figura 5.30: Algoritmo: WRR. 5000 conexiones/paso; con 1 servidor cargado

En la figura 5.31 se comparan los resultados obtenidos para el algoritmos de balanceo por Round-Robin ponderado, con dos (2) servidores cargados.

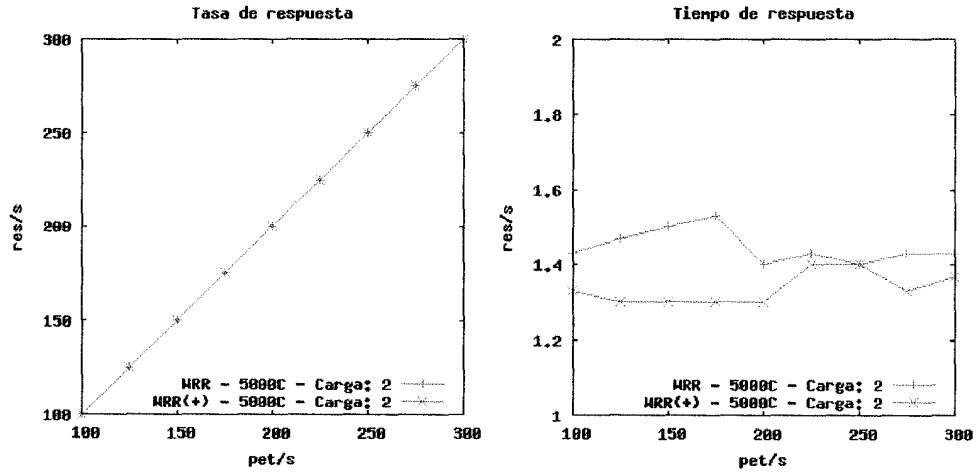


Figura 5.31: Algoritmo: WRR. 5000 conexiones/paso; con 2 servidores cargados

En las dos gráficas anteriores se observa que a pesar de encontrarse los tiempos de

respuesta cercanos a 1.4 segundos, en ambos casos el tiempo de respuesta del sistema con actualización dinámica de pesos fue menor (mejor) que cuando no se utilizó.

En la figura 5.32 se comparan los resultados obtenidos para el algoritmos de balanceo por Round-Robin ponderado, con tres (3) servidores cargados.

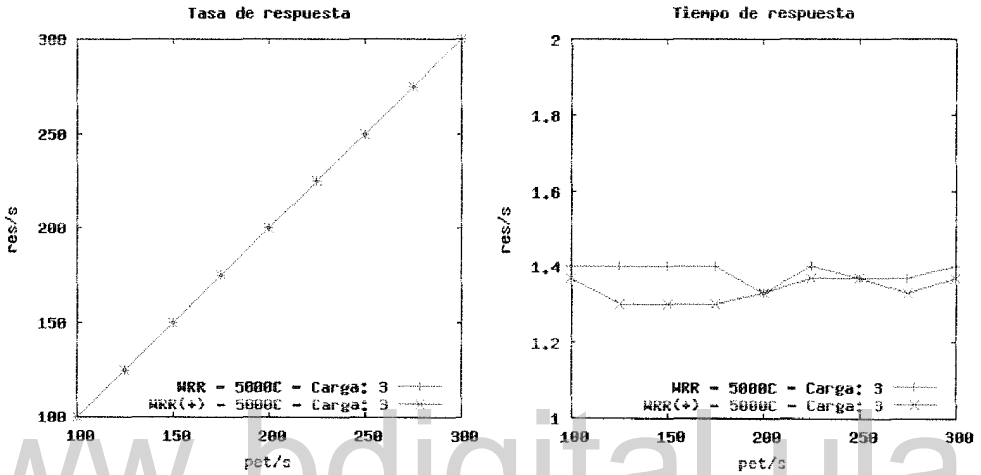


Figura 5.32: Algoritmo: WRR. 5000 conexiones/paso; con 3 servidores cargados

En este caso, no se observa una diferencia significativa entre los dos sistemas, como ocurrió durante las pruebas realizadas sin servidores cargados.

5.4. Resumen

En este capítulo se presentaron los resultados de dos grandes grupos de experimentos: un primer grupo en el cual se ubicaron los clientes de manera remota, por lo cual las peticiones/respuestas se transmitieron a través de la Internet; y un segundo grupo, con los clientes ubicados dentro de la red interna de la Universidad de Los Andes. El primer grupo estaba orientado a probar el sistema a las condiciones habituales encontradas en la Internet, tomo la limitación del ancho de banda y las variaciones propias

de este tipo de conexión). Mediante el segundo grupo, se sometió el sistema a prueba dentro de una Intranet, con un mayor ancho de banda, lo que permitió realizar un mayor número de peticiones por segundo en cada experimento.

Los resultados obtenidos cuando los clientes se ubicaron de manera remota se pueden resumir en los siguientes puntos:

- **Con sobrecarga 0** (ningún servidor sobrecargado).
  - En cuanto a tasa de respuesta los resultados para ambos sistemas (LVS original y LVS con actualización de pesos) fueron muy similares, con diferencias menores al 1 % en el punto crítico, es decir a 80 peticiones/segundo (78,90 % Vs. 78,67 % en el caso del algoritmo por menor número de conexiones ponderado, y 78,77 % Vs. 78,80 % en el caso del algoritmo por Round-Robin ponderado).
  - En cuanto al tiempo promedio de respuesta, se obtuvieron resultados mixtos: para el algoritmo de menor número de conexiones ponderado, el comportamiento fue mejor con el sistema LVS original, mientras que para el algoritmo por Round-Robin ponderado fue mejor el LVS con actualización dinámica de pesos; aunque las diferencias fueron pequeñas (3,3 segundos en el punto crítico y 15 segundos en un punto de alta demanda en el primer caso, y 3,84 segundos y 24,8 segundos en el segundo).
- **Con sobrecarga 1** (1 servidor sobrecargado).
  - En cuanta a tasa de respuesta, las variaciones entre el sistema LVS y el original se mantienen bajas, mantiéndose por debajo del 3 % para ambos algoritmos.
  - En cuanto al tiempo promedio de respuesta, los resultados obtenidos mediante el sistema LVS con actualización dinámica de pesos fueron mejores, obteniendo diferencias de hasta 49,43 segundos en puntos de alta demanda (761,37 segundos Vs. 810,80 segundos).

- **Con sobrecarga 2** (2 servidores sobrecargados).
  - En cuanta a tasa de respuesta, el LVS con actualización dinámica de pesos arroja mejores resultados, especialmente en los puntos de alta demanda, sin embargo, las diferencias se mantienen por debajo del 3 %.
  - En cuanto al tiempo promedio de respuesta, la diferencia entre los dos sistemas es notable a favor del LVS con actualización dinámica de pesos, obteniendo mejores tiempos siempre, especialmente en puntos de alta demanda, donde las diferencias observadas llegan a estar por encima de 150 segundos, lo cual representa alrededor de un 20 % de mejora, bajo las condiciones de prueba.
- **Con sobrecarga 1** (1 servidor sobrecargado).
  - En cuanta a tasa de respuesta, las variaciones nuevamente se mantienen bajas, no se evidencia una mejor respuesta en alguno de los sistemas.
  - En cuanto al tiempo promedio de respuesta, se obtuvieron resultados mixtos, con diferencias menores a los 5 segundos en el punto crítico, y menos de 50 segundos en puntos de alta demanda.

Los resultados obtenidos cuando los clientes se ubicaron en la red interna de la ULA se pueden resumir en los siguientes puntos:

- **Con sobrecarga 0.** Tanto el sistema LVS original, como el sistema LVS con actualización dinámica de pesos mostraron una tasa de respuesta del 100 %. El tiempo promedio de respuesta se ubicó alrededor de 1.4 segundos en ambos casos.
- **Con sobrecarga 1.** Tanto el sistema LVS original, como el sistema LVS con actualización dinámica de pesos mostraron una tasa de respuesta del 100 %. El tiempo promedio de respuesta nuevamente se ubicó cercano a los 1.4 segundos en todas pruebas, sin embargo fue menor el del LVS con actualización dinámica de pesos en la mayoría de las pruebas relizadas, aunque las variaciones no son significativas, alcanzando una diferencia máxima de 0.1 segundos, lo cual representa un 7 % de mejora, nuevamente, en el mejor de los casos.

- **Con sobrecarga 2.** Tanto el sistema LVS original, como el sistema LVS con actualización dinámica de pesos mostraron una tasa de respuesta del 100 %. El tiempo promedio de respuesta del LVS fue menor que el del LVS original en todas las pruebas, aunque las diferencias nuevamente fueron muy bajas (menores al 7 %).

- **Con sobrecarga 3.** Tanto el sistema LVS original, como el sistema LVS con actualización dinámica de pesos mostraron una tasa de respuesta del 100 %. En cuanto al tiempo de respuesta, las diferencias se redujeron con respecto al caso anterior, arrojando el LVS con actualización dinámica mejores tiempos en algunas pruebas, y el LVS original en otras.

Capítulo 6

Conclusión

En esta investigación se buscaba modificar el sistema de balanceo de carga del Servidor Virtual de Linux, para incorporar el nivel de ocupación de los servidores reales al proceso de balanceo.

Entre los objetivos específicos propuestos para lograr este propósito, se planteó en primer lugar el estudio del funcionamiento y la arquitectura del Servidor Virtual de Linux. En tal sentido, se proporcionó una visión general del problema de balanceo de carga, y se formalizaron los detalles conceptuales del Servidor Virtual de Linux, así como las técnicas de reenvío de paquetes que este utiliza y una visión actual del problema ARP y las estrategias para solucionarlo.

Se estudiaron los algoritmos de balanceo de carga con los que cuenta el Servidor Virtual de Linux, y se presentaron sus características fundamentales tanto conceptuales como de implementación, haciendo énfasis en el uso de pesos, como parámetro representativo de la capacidad de procesamiento de los servidores reales.

Se planteó diseñar dos componentes de software, siendo el primero un mecanismo de comunicación que permite a los servidores reales informar periódicamente al nodo director de su nivel de ocupación. En este sentido, se desarrollaron dos sub-componentes

de software, en forma de demonios/servicios del sistema operativo: uno para ser ejecutado en los servidores reales y cuyo objetivo es enviar el nivel de ocupación de dicho servidor (tal como es calculado por el núcleo de Linux, y accesible mediante el archivo virtual `/proc/loadrealsv`) a través de paquetes de datagrama sobre el protocolo de UDP, que son recibidos por el segundo sub-componente, ubicado en el nodo director, y almacenados en un archivo virtual, el cual fue nombrado `/proc/loadrealsv`.

Con respecto al mecanismo para actualizar los valores de los pesos utilizados por el Servidor Virtual de Linux, se propuso una solución consistente en un módulo para el núcleo del sistema operativo, que se encarga de modificar los valores de los pesos asignados a los servidores reales, a partir del archivo virtual `/proc/loadrealsv`. Es decir, este archivo es el elemento que conecta los dos componentes de software, y para él se propuso un formato simple para el almacenamiento de los valores de carga (Una tupla `IP:Carga` por cada servidor real), de manera de que su lectura sea lo más rápida posible.

Para comparar los resultados obtenidos mediante la herramienta propuesta con los obtenidos utilizando un Servidor Virtual de Linux sin modificar, se plantearon una serie de pruebas de carga, que se ejecutaron en dos grupos: un grupo ubicando los clientes de forma remota, y otro grupo dentro de la red interna de la ULA.

A partir de los resultados obtenidos mediante estas pruebas, se concluye que cuando todos los servidores están sobrecargados o cuando todos se encuentran en ausencia de carga, el desempeño del sistema es el mismo tanto si se usa la actualización dinámica de pesos o si no se usa. Este resultado demuestra que la incorporación del mecanismo de actualización de pesos no afecta el rendimiento del Servidor Virtual de Linux.

Cuando existen diferencias de carga entre los servidores, es decir, cuando sólo se sobrecargaba uno o dos servidores, el desempeño del sistema utilizando actualización dinámica de pesos fue mejor que cuando se utilizaba el sistema original. En estos casos, el LVS original continúa enviando peticiones a los servidores en base a unos pesos que no representan sus estados actuales de carga, mientras que con la actualización dinámica, el algoritmo de balanceo trabaja con pesos representativos, obteniendo mejores

resultados.

Las diferencias evidenciadas en este último caso se presentaron principalmente en el tiempo promedio de respuesta, obteniendo mejoras de hasta un 20 % con respecto al LVS original en el grupo de pruebas remotas, y hasta un 7 % en grupo de pruebas desde la Intranet de la ULA. En cuanto a la tasa de respuesta, aún cuando también se presentaron mejoras cuando se incorporó el mecanismo de actualización dinámica de pesos, las diferencias fueron mucho menores, rondando el 3 % en el mejor de los casos.

6.1. Aportes del proyecto

La modificación principal que hizo sobre el núcleo del Servidor Virtual de Linux, fue la adición de una función denominada genéricamente `before_sched_callback()`, que se ejecuta antes del algoritmo de balanceo, y que funge como "trampolín"para la ejecución de otras funciones. En esta investigación, el trabajo se centró en la actualización de pesos en función a determinado criterio (la carga real de los servidores), sin embargo, este trampolín deja abierta la posibilidad de ser utilizado para implementar otros criterios de actualización de pesos o realizar otras tareas, al mismo nivel de ejecución, en otros temas de investigación sobre el LVS.

Se logró implementar un mecanismo sencillo de comunicación entre servidores reales y nodo director, utilizando sockets de datagrama, que opera de forma automatizada, y que además es independiente del Servidor Virtual de Linux.

El módulo de actualización de pesos, se basa en el archivo de cargas (`/proc/loadrealsv`), es decir, no depende directamente del componente de actualización de cargas, por lo que este último pudiese ser sustituído por otro, siempre que se siga el mismo formato de escritura del archivo, y el módulo de actualización de pesos seguiría funcionando normalmente.

El módulo de actualización de pesos, además, es completamente independiente de los algoritmos de balanceo, por lo que el desarrollo y puesta en funcionamiento de un

nuevo algoritmo, es transparente al programador.

6.2. Lecciones aprendidas

La primera lección que se desprende de este trabajo, es que la carga de los servidores reales, influye directamente en el desempeño de un sistema de balanceo de carga basado en el Servidor Virtual de Linux. Tomar en cuenta los valores de carga, como indicadores del estado de los servidores reales, permite realizar un mejor balanceo.

Con respecto al desarrollo de los elementos de software, es destacable resaltar la conveniencia del uso de una metodología ágil, basada en iteraciones cortas. Esta estrategia permitió resolver de forma paulatina, cada uno de los problemas que se plantearon durante la fase de análisis de requerimientos.

Durante la fase experimental de esta investigación, se evidenció la necesidad de contar con los equipos y la infraestructura de red adecuada para poder conducir pruebas que permitan comparar efectivamente el desempeño de sistemas como el planteado. Las cantidad y características de los equipos que conforman el sistema de balanceo, así como su ubicación, deben ser considerados al definir los objetivos de las pruebas. La cantidad y ubicación de los clientes, y las características de la red mediante la cual se conectan al sistema LVS, son también determinantes sobre la calidad de las pruebas de carga.

6.3. Perspectivas

En base a los aportes de este proyecto, así como las lecciones aprendidas durante su desarrollo, en esta sección se plantean algunas recomendaciones para futuros proyectos.

En cuanto al mecanismo de actualización de pesos, se dejan planteados varios escenarios para su futuro estudio.

En primer lugar, los equipos utilizados como servidores reales en esta investigación

poseen un único procesador, sin embargo, es común hoy en día que los servidores posean dos o más núcleos de procesamiento. En este caso, los valores de carga arrojados por *loadavg* deberían ser normalizados, tomando en cuenta el número de procesadores de cada servidor real. Se deja planteada la modificación de la solución propuesta para implementar estos cambios, así como el estudio del desempeño del sistema bajo este escenario.

Otro planteamiento que se deja para futuros estudios, es la incorporación de un mecanismo de suavizado de pesos, haciendo que los pesos de los servidores reales no dependan exclusivamente de la última actualización de carga, sino también de su(s) estado(s) anterior(es). Una posible fórmula de cálculo sería: $w(t) = \alpha * w(t) + (1 - \alpha) * w(t-1)$, en donde α es un valor real entre 0 y 1 que representa la ponderación que se le da al peso actual (si $\alpha = 1$ se le daría todo el peso al valor actual, si $\alpha = 0$ el sistema no cambiaría de peso nunca). El estudio de los valores de α que hacen que un sistema se desempeñe mejor que otro bajo determinadas condiciones (como pueden ser la cantidad de servidores reales o el tipo de contenido, entre otros), permitirían obtener mayor conocimiento sobre en cuales casos la actualización dinámica de pesos es de utilidad y en qué casos es despreciable su aporte.

En cuanto al mecanismo de comunicación entre servidores reales y nodo director, se deja a consideración de futuros estudios, evaluar el uso de otros protocolos diferentes al UDP, como TCP o UDP Lite.

En esta investigación se logró incorporar exitosamente la carga real de los servidores al proceso de balanceo. Otros criterios, como el uso de memoria o la carga de interfaces de comunicación, podrían ser también incorporados al proceso en futuros estudios, con el fin de evaluar su impacto en el desempeño.

Por limitaciones de tiempo para la fase experimental, las pruebas de carga fueron realizadas sobre servidores web con información estática replicada en cada servidor real. Sería interesante evaluar el rendimiento de la solución propuesta con otro tipo de datos, como pueden ser información dinámica utilizando un servidor de base de datos, o contenido multimedia mediante *streaming*.

Bibliografía

[Abrahamsson et al., 2002] Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. (2002). Agile software development methods. Technical report, VTT Publications.

[Agarwal and Umphress, 2008] Agarwal, R. and Umphress, D. (2008). Extreme programming for a single person team. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 82–87, New York, NY, USA. ACM.

[Akpata and Riha, 2004] Akpata, E. and Riha, K. (2004). Can extreme programming be used by a lone programmer? In *International Conference on Systems Integration*, pages 167–175.

[Beck et al., 2001] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development.

[Bookman, 2002] Bookman, C. (2002). *Linux Clustering: Building and Maintaining Linux Clusters*. Sams.

[Bourke, 2001] Bourke, T. (2001). *Server Load Balancing*. O'Reilly Media, 1 edition.

[Brisco, 1995] Brisco, T. (1995). Dns support for load balancing. RFC 1794 (Informational).

[Dzhurov et al., 2009] Dzhurov, Y., Krasteva, I., and Ilieva, S. (2009). Personal extreme programming - an agile process for autonomous developers. In *Proceedings*

of the International Conference Software, Services and Semantic Technologies 2009, pages 252–259.

[Ferrari and Zhou, 1988] Ferrari, D. and Zhou, S. (1988). An empirical investigation of load indices for load balancing applications. In *Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, Performance '87, pages 515–528, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

[Fuggetta and Bstract, 2003] Fuggetta, A. and Bstract, A. (2003). Open source software – an evaluation. *Journal of Systems and Software*, 66:77–90.

[Goldman and Gabriel, 2005] Goldman, R. and Gabriel, R. P. (2005). *Innovation happens elsewhere - open source as business strategy*. Elsevier.

[Guerrero, 2009] Guerrero, E. (2009). Diseño de un servidor de balanceo de carga basado en lvs, tomando como criterio de distribución la ocupación actual de cada servidor del cluster. Master's thesis, Universidad de Los Andes.

[Harish and Owens, 1999] Harish, V. C. and Owens, B. (1999). Dynamic load balancing dns. *Linux J.*, 1999.

[Highsmith, 2002] Highsmith, J. (2002). What is agile software development? *CrossTalk - The Journal of Defense Software Engineering*.

[Highsmith and Cockburn, 2001] Highsmith, J. and Cockburn, A. (2001). Agile software development: The business of innovation. *IEEE Computer*, 34(9):120–122.

[Highsmith, 1999] Highsmith, J. A. (1999). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing Company, Incorporated.

[Kew, 2004] Kew, N. (2004). Apache week. running a reverse proxy with apache. Internet. Accesado 25-junio-2012.

[Kozierok, 2005] Kozierok, C. M. (2005). Dns name server load balancing. `http://www.tcpipguide.com/free/t_DNSNameServerLoadBalancing.htm`. Internet. Accesado 25-junio-2012.

[Moon et al., 2004] Moon, J.-B., Cho, Y.-Y., and Kim, Y.-C. (2004). A high-performance lvs system for webserver cluster. In *International Conference on Computational Intelligence*, pages 48–55.

[Moon and Kim, 2005] Moon, J.-B. and Kim, M. H. (2005). Dynamic load balancing method based on dns for distributed web systems. In *E-Commerce and Web Technologies Conference*, pages 238–247.

[Mosberger and Jin, 1998] Mosberger, D. and Jin, T. (1998). httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37.

[O'Rourke and Keefe, 2001] O'Rourke, P. and Keefe, M. (2001). Performance evaluation of linux virtual server. In *Large Installation System Administration Conference*, pages 79–92.

[Park et al., 2006] Park, G., Gu, B., Heo, J., Yi, S., Han, J., Park, J., Min, H., Piao, X., Cho, Y., Park, C., Chung, H., Lee, B., and Lee, S. (2006). Adaptive load balancing mechanism for server cluster. In *Computational Science and Its Applications - ICCSA 2006*, volume 3983 of *Lecture Notes in Computer Science*, pages 549–557. Springer Berlin / Heidelberg.

[Stallman, 1984] Stallman, R. (1984). The free software definition. `http://www.gnu.org/philosophy/free-sw.html`. Internet. Accesado 30-junio-2012.

[Taft, 2008] Taft, D. K. (2008). Agile plus open source equals developer success. *eWeek.com*, 25(August 04, 2008).

[Tucci, 2007] Tucci, P. (2007). Linux network load balancing. `http://lnlb.sourceforge.net/`. Internet. Accesado 25-junio-2012.

[Zhang, 2000] Zhang, W. (2000). Linux virtual server for scalable network services. In *Proceedings of the Ottawa Linux Symposium 2000.*

Apéndice A

Ejemplos de Uso

En este apéndice se describen los archivos que forman parte del desarrollo de esta monografía, y se explica como deben utilizarse. Adicionalmente, se dan ejemplos de uso de las herramientas de prueba, utilizadas durante la fase de experimentación.

A.1. Componente de actualización de cargas

Los elementos que conforman el componente en actualización de cargas son los siguientes:

- **lvsload_udp_d**: demonio de envío de cargas. Se debe instalar en los servidores reales, en el directorio `/usr/local/bin`
- **lvsload_udp_s**: demonio de recepción de cargas. Se debe instalar en el nodo director, en el directorio `/usr/local/bin`

Para iniciar el demonio de recepción de carga, se debe ejecutar la siguiente instrucción:

```
sudo /usr/local/bin/lvsload_udp_d DIRECTOR_PORT TIME_INTERVAL
```

Para iniciar el demonio de envío de carga, se debe ejecutar la siguiente instrucción:

```
sudo /usr/local/bin/lvsload_udp_s DIRECTOR_IP DIRECTOR_PORT TIME_INTERVAL
```

Durante la fase experimental se usaron los siguientes valores: `TIME_INTERVAL` = 10, `DIRECTOR_IP` = 150.185.182.60, `DIRECTOR_PORT` = 47239, `TIME_INTERVAL` = 10.

Adicionalmente, se programaron dos scripts para poder administrar estos demonios como otros servicios del sistema operativo:

- **lvsload_svr**: script para automatizar el funcionamiento del demonio de envío de cargas. Se debe instalar en el nodo director, en el directorio `/etc/init.d`.
- **lvsload_dir**: script para automatizar el funcionamiento del demonio de recepción de cargas. Se debe instalar en el nodo director, en el directorio `/etc/init.d`.

Estos scripts además facilitan la configuración de los parámetros mencionados anteriormente, en caso de que se requiera algún cambio. De esta forma, para iniciar los demonios, se ejecuta una de las siguientes instrucciones, según corresponda:

```
sudo /etc/init.d/lvsload_dir start #demonio de recepción de cargas (director).
sudo /etc/init.d/lvsload_svr start #demonio de envío de cargas (serv. reales).
```

Pueden iniciarse primero los demonios en los servidores reales y luego el demonio de recepción de carga, o viceversa. En cada iteración, el demonio de recepción de carga verifica si el archivo `/proc/loadrealsv` se encuentra creado, y de ser así, escribirá en él los valores de carga actuales.

A.2. Componente de actualización de pesos

El componente de actualización de pesos, como se describió en el Capítulo 4, está programado como un módulo del núcleo del sistema operativo, que puede ser cargado y descargado por el administrador del sistema como cualquier otro módulo.

Como requisito para la ejecución del módulo, debe estar corriendo el Servidor Virtual del Linux. Mientras el módulo de actualización de pesos no se encuentre activo, el LVS utilizará los pesos asignados estáticamente, una vez que se comience a ejecutar el módulo, el LVS actualizará los pesos de forma dinámica, de acuerdo a los valores de carga que se encuentren en el archivo `/proc/loadrealsv`.

A.3. Uso de las herramientas de prueba

En las máquinas utilizadas como clientes, se utilizó autobench de la siguiente manera para automatizar las pruebas:

```
autobench --single_host --output_fmt=tsv --host1=150.185.182.69
  --low_rate=LOW_RATE --high_rate=HIGH_RATE --rate_step=STEP_SIZE
  --num_conn=NUM_CONN --num_call=1 --file=FILENAME
```


Los parámetros `LOW_RATE`, `HIGH_RATE`, `STEP_SIZE` y `NUM_CONN` varían en cada prueba. Por ejemplo, para hacer una prueba desde un cliente remoto, con un total de 2000 conexiones por paso, con una tasa inicial de 20 peticiones/segundo, una tasa final de 130 peticiones/segundo y un valor de incremento de tasa de 10, se ejecutaría de la siguiente manera:

```
autobench --single_host --output_fmt=tsv --host1=150.185.182.69
  --low_rate=20 --high_rate=130 --rate_step=10
  --num_conn=2000 --num_call=1 --file=extern_wrr_2000c_test2.csv
```

El nombre del archivo de salida sigue una convención utilizada en esta investigación para describir unívocamente cada prueba. El prefijo "extern" indica que la prueba se hace desde un cliente remoto, "wrr" indica que es una prueba del algoritmo WRR (Round-Robin ponderado), 2000c indica que se realizan 2000 conexiones por paso y "test2" indica que se trata de los resultados de la segunda (de tres) réplicas de la prueba, bajo las condiciones antes descritas.

Apéndice B

Código fuente

En este apéndice se incluye el código fuente completo de los programas.

B.1. Demonio de envío de cargas

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <math.h>

#define tam_buffer 100
#define tam_shorter 10
#define NAME_FILE                    "///proc//loadavg"

int main( int argc, char *argv[] )
{

  FILE *file_load;
  char *d, line_load[tam_buffer], onemin[tam_shorter];
  double OneMin, FiveMins, FifteenMins, result, *iptr;
  int i = 0, firstMin, socket_udp, tam, num_bytes, time_sleep;
  struct sockaddr_in addressip;

  /* validation input arguments */
  if( argc != 4 )
  {
```

```
  exit( -1 );
}
time_sleep = atoi(argv[3]);

/*************************************************************
* Daemonization
*************************************************************/
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
  exit(EXIT_FAILURE);
}

if (pid > 0) {
  exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
  exit(EXIT_FAILURE);
}

/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);


/*************************************************************
* Create socket and loop
*************************************************************/

/* init socket */
addressip.sin_family = AF_INET;
addressip.sin_port = htons( atoi( argv[2] ) );
inet_aton( argv[1], &(addressip.sin_addr) );
bzero( &(addressip.sin_zero), 8 );
socket_udp = socket(AF_INET, SOCK_DGRAM, 0);

tam = sizeof(struct sockaddr);
iptr = (double *) malloc (sizeof(double));
do
{
  if((file_load = fopen(NAME_FILE,"r")) == NULL )
  {
    exit(1);
  }
  else
  {
    while(!feof(file_load))
    {
      d = fgets(line_load,50,file_load);
      sscanf(line_load,"%lf %lf %lf", &OneMin, &FiveMins, &FifteenMins);
      // rounding load in the one minute to int
```

```
        OneMin = OneMin*10;
        result = modf(OneMin,iptr);
        if ((result >= 0.00) && (result < 0.50))
          firstMin = (int) floor(OneMin);
        else
          firstMin = (int) ceil(OneMin);
        sprintf(onemin,"%d",firstMin);
        // Send load to lvs
        num_bytes = sendto(socket_udp, onemin, strlen(onemin),
                           0, (struct sockaddr *)&addressip, tam);
      }
      i++;
      fclose (file_load);
      sleep(time_sleep);
    }

  }while(onemin != NULL);

  // Close the connection:
  close(socket_udp);

  return 0;
}
```


B.2. Script de envío de cargas

```
#!/bin/bash
#
RETVAL=0;

DIRECTOR_IP="192.168.122.25"
DIRECTOR_PORT="47239"
TIME_INTERVAL="10"

start() {
  echo "Starting lvsload_svr"
  /usr/local/bin/lvsload_udp_s $DIRECTOR_IP $DIRECTOR_PORT $TIME_INTERVAL
  echo "lvsload_svr started"
}

stop() {
  echo "Stopping lvsload_svr"
  killall -e lvsload_udp_s
  echo "lvsload_svr stopped"
}

restart() {
  echo "Stopping lvsload_svr"
  killall -e lvsload_udp_s
  echo "lvsload_svr stopped"
  echo "Starting lvsload_svr"
  /usr/local/bin/lvsload_udp_s $DIRECTOR_IP $DIRECTOR_PORT $TIME_INTERVAL
  echo "lvsload_svr started"
}
```

```
case "$1" in
start)
  start
  ;;
stop)
  stop
  ;;
restart)
  restart
  ;;
force-reload)
  restart
  ;;
*)
  echo $"Usage: $0 {start|stop|restart}"
  exit 1
  ;;
esac

exit $RETVAL
```

B.3. Demonio de recepción de cargas


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <time.h>

#include "listas.h"
#include "loadrs.h"

#define tam_load 8
#define tam_buffer 256
#define NAME_FILE                    "///proc//loadrealsv"
#define INTERVALO 10
#define tam_ip 20

int main(int argc,char *argv[])
{

  time_t tbegin, tnow;
  struct sockaddr_in iprealsv;
  struct server_info *temp, *server_real, *cur_real_server;
  char server_load[tam_load], addressip[tam_ip], cur_add[tam_ip], buffer[tam_buffer];
  LISTA list_realsv;
  FILE *file_load;
  double tdiff;
  unsigned pos, total;
  int band, socket_udp, tam, numbytes = 1, n = 0, interval = 1;
```

```
/* validation input arguments */
if(argc != 3)
{
  exit( -1 );
}

/*************************************************************
* Daemonization
*************************************************************/
pid_t pid, sid;

/* Fork off the parent process */
pid = fork();
if (pid < 0) {
  exit(EXIT_FAILURE);
}

if (pid > 0) {
  exit(EXIT_SUCCESS);
}

/* Change the file mode mask */
umask(0);

/* Create a new SID for the child process */
sid = setsid();
if (sid < 0) {
  exit(EXIT_FAILURE);
}

/* Close out the standard file descriptors */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);


/*************************************************************
* Create socket and loop
*************************************************************/


list_realsv = crea_lst((unsigned) sizeof(struct server_info));

temp = (struct server_info *) malloc (sizeof(struct server_info));

server_real = (struct server_info *) malloc (sizeof(struct server_info));

cur_real_server = (struct server_info *) malloc (sizeof(struct server_info));

interval = atoi(argv[2]);

/* init socket */
iprealsv.sin_family = AF_INET;
iprealsv.sin_port = htons( atoi(argv[1]) );
iprealsv.sin_addr.s_addr = INADDR_ANY;
bzero( &(iprealsv.sin_zero), 8 );

socket_udp = socket( AF_INET, SOCK_DGRAM, 0 );
tam = sizeof( struct sockaddr );
```

```
if ((bind( socket_udp, (struct sockaddr *)&iprealsv, tam )) == -1)
  printf("\n>:ERROR: bind function \n");

/* initialization time to interval between writing in file /proc/loadrealsv */
tbegin = time(NULL);

/* loop to receive load from real servers */
while (numbytes > 0)
{
  n++;
  numbytes = recvfrom(socket_udp, buffer, tam_buffer-1, 0, (struct sockaddr *)&iprealsv
  buffer[numbytes] = '\0';
  strcpy(server_real->server_load, buffer);
  server_real->ip_server.sin_addr.s_addr = iprealsv.sin_addr.s_addr;
  strcpy(addressip,inet_ntoa(iprealsv.sin_addr));
  server_real->update_time = time(NULL);
  addressip[strlen(addressip)] = '\0';
  strcpy(server_load,buffer);
  cur_real_server = (struct server_info*)pri_lst(list_realsv);
  band = 1;
  total = num_lst(list_realsv);

  /* loop to find the current real server in the list */
  while ((pos = pos_lst(list_realsv) <= total) && (total != 0) && (cur_real_server != NU
  {
    strcpy(cur_add, inet_ntoa(cur_real_server->ip_server.sin_addr));
    cur_add[strlen(cur_add)] = '\0';
    if (strcmp(cur_add, addressip) == 0)
    {
      strcpy(cur_real_server->server_load, buffer);
      cur_real_server->update_time = time(NULL);
      band = 0;
      pos = 0;
      break;
    }

    if (pos < total)
      cur_real_server = (struct server_info*)sig_lst(list_realsv);
    else
      break;
  }

  /* if not found then the current real server adds in list */
  if (band != 0)
    ins_lst(list_realsv, server_real);

  /* take time current*/
  tnow = time(NULL);

  /* compare difference between initial time and current time with INTERVALO
   * to write in file /proc/loadrealsv */
  if (((tdiff = difftime(tnow, tbegin))) > interval)
  {
    file_load = fopen(NAME_FILE,"w");
    if (file_load == NULL)
    {
      //uncomment next line if the daemon must exit when file does not exist
      //exit (EXIT_FAILURE);
      //comment next line if the daemon must exit when file does not exist
```

```
        continue;

      }
      temp = (struct server_info*)pri_lst(list_realsv);

      /* iterate over list and each entry write in file */
      do
      {
        if (((tdiff = difftime(tnow, temp->update_time)) < (2*interval)) || ((tdiff = dif:
        {
          strcpy(addressip, inet_ntoa(temp->ip_server.sin_addr));
          strcpy(server_load, temp->server_load);
          if (fprintf(file_load, "%s:%s:\n", server_load, addressip))
            printf("in file is: real_server_load: %s, address_ip: %s \n", server_load, ad
        } else
          sup_lst(list_realsv);
      }while ((temp = (struct server_info*)sig_lst(list_realsv)) != FINLST);
      tbegin = tnow;
      fclose(file_load);
    }
  }

  /* close connection */
  close( socket_udp );
  free(list_realsv);
  free(server_real);
  free(cur_real_server);
  free(temp);
  return 0;
}
```


B.4. Script de recepción de cargas

```
#!/bin/bash
#
RETVAL=0;

DIRECTOR_PORT="47239"
TIME_INTERVAL="10"


start() {
  echo "Starting lvsload_dir"
  /usr/local/bin/lvsload_udp_d $DIRECTOR_PORT $TIME_INTERVAL
  echo "lvsload_dir started"
}

stop() {
  echo "Stopping lvsload_dir"
  killall -e lvsload_udp_d
  echo "lvsload_dir stopped"
}

restart() {
```

```
  echo "Stopping lvsload_dir"
  killall -e lvsload_udp_d
  echo "lvsload_dir stopped"
  echo "Starting lvsload_dir"
  /usr/local/bin/lvsload_udp_d $DIRECTOR_PORT $TIME_INTERVAL
  echo "lvsload_dir started"
}

case "$1" in
start)
  start
  ;;
stop)
  stop
  ;;
restart)
  restart
  ;;
force-reload)
  restart
  ;;
*)
  echo $"Usage: $0 {start|stop|restart}"
  exit 1
  ;;
esac

exit $RETVAL
```


B.5. Módulo de actualización de pesos

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <net/ip_vs.h>
#include <linux/proc_fs.h>
#include <linux/fs.h>
#include <linux/gcd.h>

#define PROC_FILE_NAME "loadrealsv"
#define PROC_MAX_SIZE 992 /* Buffer maxime size */

// this object holds information about the loadrealsv file
static struct proc_dir_entry * proc_file_meta;

/* The buffer used to store character for this module */
static char load_buffer[PROC_MAX_SIZE];

/* The size of the buffer */
static unsigned long load_buffer_size = 0;

/*
 * In ip_vs_core.c is defined before_sched_callback pointing to a empty function.
 * before_sched_callback is a callback function that is executed before the
 * scheduling algorithm.
 *
```

```
 * When ip_vs_load_rel is loaded, it overwrites the empty function with the one
 * defined here.
 *
 * original_callback_reference is used save the reference to the original (empty)
 * function, to be restored if ip_vs_load_rel is unloaded
 */
int (*ip_vs_dummy_reference)(struct ip_vs_service *, struct sk_buff *);

extern int (*ip_vs_before_sched_callback)(struct ip_vs_service *, struct sk_buff *);

/*
 * This function is called when the /proc file is written
 *
 */
int loadfile_write(struct file *file, const char *buffer, unsigned long count, void *data
{
    int i, numbytes;

    /* get buffer size */
    load_buffer_size = count;
    if (load_buffer_size > PROC_MAX_SIZE ) {
        load_buffer_size = PROC_MAX_SIZE;
    }

    for (i = 0; i < PROC_MAX_SIZE; i++) {
        load_buffer[i]='\0';
    }
    /* write data to the buffer */
    if ( copy_from_user(load_buffer, buffer, load_buffer_size) ) {
        return -EFAULT;
    }
    numbytes = strlen(load_buffer);
    load_buffer[ numbytes ] = '\0';
    return load_buffer_size;
}

/*
 * This function is called when the /proc file is read
 *
 */
int loadfile_read(char *buffer,
              char **buffer_location,
              off_t offset, int buffer_length, int *eof, void *data)
{
    int ret;

    if (offset > 0) {
        ret  = 0;
    }
    else {
        /* fill the buffer, return the buffer size */
        memcpy(buffer, load_buffer, load_buffer_size);
        ret = load_buffer_size;
    }

    return ret;
}


/*
```

```
 * This function is used to compute load to weight: while smaller is the load,
 * greater is the weight, meaning major capacity for receiving requests
 */
int ip_vs_load_to_weight(char *load)
{
    int iload;

    sscanf(load, "%d", &iload);
    if (iload < 3 && iload >= 0)
        return(3-iload);
    else if(iload == 3)
        return (1);
    else
        return(0);
}

/**
* This function is called from scheduling algorithm, to update
* the weight of the destinies with their last one minute load.
*/
int ip_vs_load_callback(struct ip_vs_service *svc, struct sk_buff *skb)
{

    int j, flagload = 0, flagip = 0, load_ip = 0, idem_ip = 1;
    struct ip_vs_dest *dest;
    /* vars to separate and store tokens from the load_buffer */
    char *address_ip = NULL, *load = NULL, *load_word = NULL;
    char* bf = NULL;
    char** bbf;
    char tmp_buffer[PROC_MAX_SIZE];

    load_word = kmalloc((sizeof(char)*100), GFP_ATOMIC);
    address_ip = kmalloc((sizeof(char)*100), GFP_ATOMIC);
    load = kmalloc((sizeof(char)*100), GFP_ATOMIC);

    // iterate over list to update the weight of dest with load
    // of real server
    list_for_each_entry(dest, &svc->destinations, n_list) {
        idem_ip = 1;
        flagload = 0;
        flagip = 0;
        strcpy(tmp_buffer, load_buffer);
        bf = tmp_buffer;
        bbf = &bf;

        // loop to gets the tokens from load_buffer
        while ((load_word = strsep(bbf, ":")) != NULL) {
            // with the length of word decide between load or address ip
            j = strlen(load_word);

            if (j > 5) { // address ip
                strcpy(address_ip, load_word);
                flagip = 1;
            }

            else {  // load value
                strcpy(load, load_word);
                load_ip = ip_vs_load_to_weight(load);
                flagload = 1;
            }
```

```
            if (flagload == 1 && flagip == 1) {
                flagload = 0;
                flagip = 0;
                if ((dest->addr).ip == (in_aton(address_ip))) {
                    atomic_set(&dest->weight, load_ip);
                    idem_ip = 0;
                    break;
                }
            }

        }

        // if dest is not in list, its weight update 0
        if (idem_ip != 0)
            atomic_set(&dest->weight, 0);
    }

    kfree(load_word);
    kfree(address_ip);
    kfree(load);
    return(0);
}

int create_proc_file(void)
{
    proc_file_meta = create_proc_entry(PROC_FILE_NAME, 0644, NULL);
    if(proc_file_meta == NULL) {
        remove_proc_entry(PROC_FILE_NAME, NULL);
        return -ENOMEM;
    }

    proc_file_meta->read_proc  = loadfile_read;
    proc_file_meta->write_proc = loadfile_write;
    proc_file_meta->mode          = S_IFREG | S_IRUGO;
    proc_file_meta->uid           = 0;        //especify the user ID of the file
    proc_file_meta->gid           = 0;        //especify the group ID of the file
    proc_file_meta->size          = 37;         //especify the length of the file to show

    return 1;
}
EXPORT_SYMBOL(create_proc_file);

static int __init ip_vs_load_init(void)
{
    create_proc_file();
    ip_vs_dummy_reference = ip_vs_before_sched_callback;
    ip_vs_before_sched_callback = &ip_vs_load_callback;

    return 0;
}

static void __exit ip_vs_load_cleanup(void)
{
    remove_proc_entry(PROC_FILE_NAME, NULL);
    ip_vs_before_sched_callback = ip_vs_dummy_reference;
}

module_init(ip_vs_load_init);
module_exit(ip_vs_load_cleanup);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Marco Camejo");
MODULE_DESCRIPTION("Realtime Load information of the nodes for LVS");
```