Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Clúster de Alta Disponibilidad y Equilibrado de Carga para la Aplicación FarHos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Pablo Flores Soriano

Tutor: Pedro Juan López Rodríguez

2019-2020

Resumen

El siguiente proyecto configura un clúster de alta disponibilidad con equilibrado de carga que permite garantizar un funcionamiento ininterrumpido al eliminar los puntos únicos de fallo y crecer en prestaciones añadiendo servidores. El TFG desarrolla e implementa un demostrador basado en el uso de contenedores Docker y máquinas virtuales con la intención de documentar las ventajas que ofrece Docker frente a la virtualización tradicional.

La propiedad de alta disponibilidad se garantiza eliminando los componentes del sistema que tras un fallo ocasionen la parada de todo el servicio. Estos elementos son conocidos como puntos únicos de fallo. Para conseguirlo se han configurado copias (réplicas) de todas las máquinas de modo que si una cesa su actividad otra pueda ocupar su lugar.

El equilibrado de carga se consigue a través de un software director llamado HAProxy encargado de repartir las peticiones de los clientes entre un conjunto de servidores. Además, este programa también asume la tarea de monitorizar el estado de los servidores para seleccionar solo aquellos que estén activos. Para evitar que se convierta en un punto único de fallo se han instalado dos equilibradores de carga en configuración activo/pasivo.

El punto de acceso al clúster se configura a través de una dirección IP virtual (VIP) cuya asignación es manejada por el software Keepalived, encargado de monitorizar el estado de las máquinas *load balancer* maestro y backup. Siempre que la máquina principal esté activa se le fijará la VIP a su interfaz de red. En caso de detectar una parada, la VIP pivota a la máquina de respaldo, de modo que el fallo es totalmente transparente para el cliente.

Se plantean dos objetivos. El primero se centra en configurar y desplegar el clúster. Mediante un conjunto de pruebas cualitativas se ha comprobado que las peticiones se reparten según el algoritmo de round-robin y que el servicio sigue disponible tras el fallo de un nodo. Para este conjunto de pruebas se ha utilizado la aplicación FarHos Prescripción, proporcionada por la empresa Visual-Limes, para la consulta de la prescripción médica de pacientes.

El segundo objetivo se centra en estudiar las ventajas que ofrece el uso de Docker frente a máquinas virtuales. Para alcanzarlo se ha utilizado una aplicación Web que genera una elevada carga de cómputo en los nodos servidores (cálculo de integrales definidas). Las pruebas se han realizado ejecutando el clúster sobre una máquina física de alta gama. Mediante ensayos cuantitativos se ha llegado a la conclusión que Docker es ventajoso en tanto que reduce sustancialmente el consumo de memoria RAM y su velocidad de despliegue es mucho mayor. No obstante, la capacidad computacional (uso de CPU) es equivalente en contenedores y máquinas virtuales.

Palabras clave: Replica, sincronizado, punto único de fallo, equilibrado de carga, alta disponibilidad, haproxy, keepalived, ip virtual, cuantitativa, cualitativa.

Abstract

The following project configures a high-performance service cluster to ensure uninterrupted operation by eliminating single points of failure and growing in performance by adding servers. The TFG develops and implements a demonstrator based on the use of Docker containers and virtual machines with the intention of documenting the advantages offered by Docker over traditional virtualisation.

The high availability property is guaranteed by eliminating the system components that after a failure cause the stop of the whole service. These elements are known as single points of failure. To achieve this, replication has been configured so that if one ceases its activity another can take its place.

Load balancing is achieved with a software called HAProxy, which distributes client requests to a set of servers. In addition, this program also assumes the task of monitoring the state of the servers in order to select only those that are active. To prevent it from becoming a single point of failure, two load balancers have been installed in a active/passive configuration.

The access point to the cluster is configured through a virtual IP address (VIP) whose assignment is handled by the Keepalived software, in charge of monitoring the state of the master and backup load balancer machines. Whenever the master machine is active the VIP will be fixed to its network interface. In the event of a shutdown being detected, the VIP pivots to the backup machine, so that the failure is completely transparent to the customer.

There are two objectives. The first one is focused on configuring and deploying the cluster. A set of qualitative tests have shown that requests are distributed according to the Round-Robin algorithm and that the service is still available after a node failure. For this set of tests, the “Farhos Prescripción” application, provided by the company Visual-Limes, was used to consult patients' medical prescriptions.

The second objective is to study the advantages offered using Docker compared to virtual machines. To achieve this, a web application that demands a high load to the server nodes has been used. The tests were run on a high-end physical machine. Through quantitative tests, it has been concluded that the Docker is better than virtual machines, as it substantially reduces RAM memory consumption and its deployment is much faster. However, the computational capacity (CPU usage) is equivalent in containers and virtual machines.

Keywords : Replica, synchronized, single point of failure, load balancing, high availability, haproxy, keepalived, virtual ip, quantitative, qualitative.

Tabla de contenidos

Tabla de Ilustraciones

Tabla de Figuras

Prólogo

Los primeros modelos de las tecnologías de la comunicación se manifiestan a comienzos del siglo XX en una sociedad que solo conocía la mensajería en forma de carta. La comunicación nunca ha existido como hoy la conocemos. En la Antigüedad, los espacios de comunicación físicos eran el ágora, posteriormente serían la iglesia y las plazas de los pueblos o ciudades siendo muy difícil la difusión de información entre la sociedad ajena a dichos espacios. En el Renacimiento, la aparición de la imprenta y las mejoras en las redes de transporte hicieron posible mantener a las personas informadas sobre los acontecimientos que ocurrían a su alrededor, pero contactar con el otro extremo del planeta seguía siendo muy complicado.

No fue hasta la Primera Guerra Mundial que Europa advirtió sobre la imperiosa necesidad de disponer de métodos de comunicación efectivos. Así, con una motivación orientada a prevalecer la economía y ganar la guerra se hicieron grandes esfuerzos de investigación en tecnología de transporte y telecomunicaciones.

A partir de la década de 1980 se hizo popular el concepto de “tecnologías de la información” fruto de aunar los términos tecnología, información y comunicación. El nacimiento de las TIC ha provocado un cambio radical sin precedentes en la evolución económica y social de la humanidad al permitir una comunicación efectiva e instantánea en cualquier momento. Nunca se había podido conectar tantas culturas y facilitar el intercambio de información entre países de forma inmediata, y su altísimo potencial económico ha provocado que, paulatinamente, el mundo entero gire en torno a las TIC.

Actualmente no se puede hablar de un “incremento en la popularidad” de Internet. Internet es indispensable para el soporte de todo organismo público y/o privado. Más aún, los recientes acontecimientos relacionados con la pandemia global han esclarecido la verdadera necesidad de que los sistemas informáticos se apoyen en una infraestructura que garantice de forma prioritaria una disponibilidad absoluta, y como añadido secundario, importante pero no imprescindible, su alto rendimiento.

1. Introducción

Como se ha comentado en el prólogo, la humanidad ha entrado en una era de la información donde los usuarios precisan servicios web cada vez más rápidos, disponibles y confiables. Ante el crecimiento exponencial del uso de internet, los sitios web han tenido que adaptarse rápidamente a una demanda de peticiones sin precedentes que necesitan ser atendidas en cualquier momento.

Además, un perfil de usuario cada vez más crítico con la calidad de servicio que se le ofrece ha obligado a las entidades proveedoras de servicios informáticos en la red a desarrollar metodologías para construir servicios escalables, de alto rendimiento y altamente disponibles. En consecuencia, se ha producido un cambio en el paradigma de las aplicaciones tradicionales. Estas han pasado de ser ejecutadas en máquinas autónomas y dedicadas a ejecutarse en un servidor, que generalmente suele ser un clúster de computadores ya que es el mejor modo de proporcionar los recursos necesarios para ofrecer escalabilidad, alto rendimiento y disponibilidad con un coste asequible. [3]

1.1 Clúster

El concepto de "clúster" varía su significado en función del contexto en el que se utilice. En el libro "Building Clustered Linux Systems – Robert W. Lucke" [4] se hace la siguiente definición de clúster: "A closely coupled, scalable collection of interconnected computer systems, sharing common hardware and software infrastructure, providing a parallel set of resources to services or applications for improved performance, throughput, or availability".

En el ámbito informático, pues, podemos definir un clúster como "una colección escalable de sistemas informáticos interconectados, que comparten una infraestructura software y hardware común, y proporcionan un conjunto de recursos paralelos para incrementar el rendimiento y la disponibilidad de los servicios o las aplicaciones".

Para conseguir un sistema informático de tales características es necesario recurrir a un "clúster" de ordenadores, ya que solo una infraestructura tecnológica como tal es capaz de proporcionar inmunidad ante fallos, escalabilidad y alto rendimiento.

Se puede distinguir entre varios tipos de clúster en función de la finalidad para la cual hayan sido diseñados, aunque en última instancia todos ellos poseen la misma naturaleza. Su arquitectura viene dada por el tipo de servicio que se va a proporcionar. Así, en el ámbito científico los clústeres se diseñan orientados al cálculo computacional mientras que en el ámbito de las tecnologías de la información (TI) se pretende proporcionar alta disponibilidad.

High Performance Computing (HPC)

Este tipo de clústeres están diseñados para entornos con grandes necesidades de cómputo masivo. Por ejemplo, se utilizan varios servidores para ejecutar algoritmos científicos complejos de forma paralela sobre varios procesadores. En concreto, cada nodo resuelve

una porción del problema, comunicándose y sincronizándose con los otros nodos cuando sea necesario. La librería OpenMPI[1] proporciona los recursos necesarios para conseguir esta funcionalidad.

High Performance Service (HPS)

Se utiliza habitualmente en entornos con una alta demanda de peticiones, como por ejemplo una aplicación web conocida. Este diseño implica ejecutar múltiples tareas independientes sobre varios procesadores a la vez. Puede causar confusión con el anterior ya que aparentan ser similares.
La diferencia entre HPS y HTC radica en que el primero está orientado al cálculo de costosos algoritmos donde todos los nodos trabajan en una sola tarea, mientras que el segundo pretende atender un elevado número de peticiones de distintos clientes equilibrando la carga entre sus múltiples servidores.
Además, aparece un agente especial conocido como "equilibrador de carga", que consiste en una máquina independiente situada en la parte front-end de una aplicación encargada de recibir las peticiones y distribuirlas entre un conjunto de nodos trabajadores (workers) para asegurar un reparto equitativo de la carga de trabajo.

High Availability (HA)

Los clústeres de alta disponibilidad tienen un solo objetivo: garantizar, en la medida de lo posible, la disponibilidad de un recurso crítico y la reducción de tiempos de inactividad no programados. Mediante técnicas de monitorización y replicación de recursos se pretende garantizar que, cuando un servidor sufre una interrupción, siempre haya otro servidor que pueda respaldarlo, de manera que el recurso pueda continuar disponible.

Este proyecto abarca el diseño e implementación de un clúster de alta disponibilidad con equilibrado de carga para ofrecer HPS, con lo que se van a exponer las técnicas necesarias para equilibrar la carga de peticiones de una aplicación dada, a la vez que se garantiza la disponibilidad continuada de la aplicación.

1.2 Servicio de Alta Disponibilidad con Equilibrado de carga

Como se ha dicho anteriormente, un clúster de alta disponibilidad pretende garantizar, en la medida de lo posible, el funcionamiento ininterrumpido de una aplicación. Sin embargo, es necesario ser conscientes de que a día de hoy es imposible garantizar la disponibilidad absoluta de un servicio, más aún si se pretende hacer un balance equitativo entre las necesidades de negocio y el coste económico de obtener y mantener la infraestructura.

La replicación de recursos es la técnica más utilizada para incrementar el grado de alta disponibilidad, cuyo objetivo es la supresión de los puntos únicos de fallo (SPOF - *Single Point of Failure*) mediante la replicación de servidores, de modo que ante el fallo de uno sus copias pueden continuar atendiendo peticiones.

[1] Open MPI – Open Source High Permormance Computing. Disponible en: https://www.open-mpi.org/

Utilizar esta técnica permite tener un número ‘n’ de servidores trabajadores capaces de realizar la misma función. Así, si uno de ellos sufre una interrupción todos los demás pueden continuar prestando el servicio.

Por otra parte, para evitar el desperdicio de recursos se instala un equilibrador de carga que garantice un reparto equitativo de peticiones y además sea capaz de conocer el estado de los servidores, con objetivo de tomar las medidas oportunas en caso de que uno de ellos falle (intentar recuperarlo, notificar el error, redirigir las peticiones al resto de servidores…).

1.3 Objetivos

El proyecto que se va a desarrollar tiene dos objetivos principales. El primero es configurar un clúster de alta disponibilidad con equilibrado de carga que permita, a cualquier servicio web, crecer en prestaciones añadiendo servidores, y garantizar un funcionamiento ininterrumpido al eliminar puntos únicos de fallo.
El segundo objetivo es analizar las diferencias entre dos tecnologías de virtualización para implementar los nodos del clúster, el uso de contenedores Docker o de máquinas virtuales tradicionales. En función de los resultados, se va a determinar si es ventajoso configurar un clúster utilizando Docker.

1.4 Estructura

En esta sección se va a comentar la estructura que sigue el proyecto.

En el apartado “Estado del arte” se hace un breve análisis del contexto tecnológico con relación a los clústeres de tipo HPS/HA. A continuación, se estudian las distintas herramientas que se pueden utilizar para llevar a cabo la propuesta de proyecto, analizando sus pros y sus contras.

Una vez elegidas las herramientas que se van a utilizar, se expone un esquema de la infraestructura escogida para posteriormente detallar la configuración de todas sus partes.

Más adelante, en la sección “evaluación de la solución” se expone el conjunto de pruebas cuantitativas y cualitativas realizadas para comprobar la satisfacción de los objetivos designados al comienzo del proyecto. Una vez obtenidos los resultados, se extraen las conclusiones pertinentes.

Por último, se hace una reflexión sobre todo el proyecto para valorar lo aprendido y su relación con los estudios cursados en la carrera.

Al final del documento se encuentras las referencias bibliográficas, los anexos con los datos estadísticos obtenidos en las pruebas, una breve descripción de la empresa que ha proporcionado la aplicación para probar el clúster y un glosario de términos.

2. Estado del arte

La excelente relación entre el coste de los clústeres y las prestaciones que permiten alcanzar ha provocado que estos se conviertan en una solución ampliamente utilizada. Además, debido a su naturaleza pueden adaptarse perfectamente a cualquier ámbito, desde pequeños entornos y medianas empresas hasta los mayores servicios web que actualmente se conocen.

Como se ha comentado, los clústeres pueden ser de distintos tipos en función al propósito para el que hayan sido configurados. En relación con la prestación de servicios web se utiliza una configuración HPS/HA en la cual se va a centrar el proyecto a partir de este apartado.

Para acceder al clúster es necesario instalar un repartidor de carga o *load balancer* (LB) encargado de distribuir las peticiones de los clientes entre el conjunto de servidores disponibles, conocidos como servidores reales, nodos trabajadores o *workers*. El LB, además de ser el punto de acceso al servicio, proporciona la funcionalidad de equilibrado de carga que permite repartir las peticiones para que todos los *workers* tengan la misma carga de trabajo.
Los *load balancer* se pueden configurar con un dispositivo hardware específico, aunque es una solución cara y poco flexible. No obstante, existen soluciones software gratuitas capaces de proporcionar tal funcionalidad. Dos ejemplos son HAProxy[2] y Linux Virtual Server[3], cuyas ventajas y desventajas se estudian en el siguiente apartado.[2]

En cuanto a la alta disponibilidad, todos los clústeres proporcionan esta característica en mayor o menor medida, aunque este no sea su principal objetivo. La finalidad de un clúster de alta disponibilidad es garantizar un servicio ininterrumpido y un rendimiento constante.
Gran parte de estos propósitos se pueden conseguir mediante herramientas software encargadas de detectar fallos en el hardware o en el servicio y redirigir las peticiones a un nodo sustituto, como por ejemplo Keepalived[4] o Corosync[5]+Pacemaker[6]. Sin embargo, también es imprescindible abordar algunos aspectos hardware durante el diseño de la infraestructura, como por ejemplo garantizar que no exista ningún punto único de fallo [4].
Más aún, no es suficiente con replicar todos los nodos y monitorizarlos. Para garantizar el mayor grado de disponibilidad posible es necesario tener en cuenta otros aspectos, como utilizar un almacenamiento compartido para asegurar la persistencia de datos a través de *Network File System (NFS)* o una *Storage Area Network (SAN)*. También es interesante el uso de redes diferenciadas para evitar que todo el tráfico se dirija por la misma red, evitando la congestión y la consecuente pérdida de paquetes. Por último, es necesario contemplar el uso de dispositivos de *fencing* que aíslen un nodo cuando falle para evitar que interfiera con el resto del clúster. Por ejemplo, se puede utilizar la técnica STONITH (*Shoot The Other Node In The Head*) para desconectar al nodo fallido. [5]

[2] HaProxy – The Reliable, High Performance TCP/HTTP Load Balancer. Disponible en: http://www.haproxy.org/
[3] Linux Virtual Server. Disponible en: https://en.wikipedia.org/wiki/Linux_Virtual_Server
[4] Keepalived. Disponible en: https://keepalived.org/
[5] Corosync – The Corosync Cluster Engine. Disponible en: http://corosync.github.io/corosync/
[6] Pacemaker. Disponible en: https://clusterlabs.org/pacemaker/

2.1 Crítica al estado del arte

Hoy en día la infraestructura de un clúster se configura en máquinas físicas independientes, aunque también existen configuraciones en máquinas virtuales.
La tecnología de virtualización surgió en 1960 de la mano de IBM dando la capacidad de ejecutar aplicaciones o sistemas operativos en un entorno independiente del hardware (lo que se conoce como propiedad de aislamiento).
Hasta hace poco tiempo, la virtualización de máquinas virtuales ha sido la única forma de poder ejecutar varios sistemas operativos sobre el mismo conjunto de recursos hardware. Además de todas las ventajas y posibilidades que ofrece esta tecnología, permite configurar una infraestructura virtual como si se tratase de una física. Así, en el caso de clúster, es posible aprovecharse de las ventajas que ofrece un clúster sin tener que asumir el coste económico de cada máquina necesaria.

No obstante, hasta hace poco la virtualización ha estado enfocada a la ejecución de un sistema operativo completo ya que es la única manera de conseguir la propiedad de aislamiento entre la máquina virtualizada (huésped) y la máquina anfitriona.
En 2013 se popularizó la tecnología de contenedores bajo el nombre comercial de Docker. Si bien esta tecnología existía hace años (FreeBSD Jails, Solaris Containers…) no fue hasta la aparición de Docker que se extendió su uso entre los usuarios.
La principal diferencia entre ambas es que, en lugar de virtualizar el hardware y el sistema operativo completo, un contenedor supone un entorno aislado ejecutándose sobre el sistema operativo anfitrión. Esto implica menor consumo de memoria RAM y una velocidad de inicialización mucho mayor. Más adelante se profundiza en las diferencias entre ambas tecnologías.

Hoy en día prácticamente la totalidad de los clústeres que no se apoyan en una infraestructura física lo hacen en una de máquinas virtuales. No obstante, las ventajas en cuanto a eficiencia que aporta Docker pueden llevarlo a ser una solución alternativa y mucho más ligera a la hora de configurar un clúster.

2.2 Propuesta

Se quiere hacer una prueba de concepto de un clúster de alta disponibilidad con equilibrado de carga. Además, se va a introducir Docker como alternativa a la virtualización.

Se propone configurar un clúster de alta disponibilidad con equilibrado de carga. El proyecto supone una prueba de concepto de la combinación de las tecnologías de virtualización y contenedores para aprovechar las ventajas de ambas. De una mano, es imprescindible utilizar máquinas virtuales para crear la infraestructura del sistema dado que es necesario contar con cierto número de recursos hardware virtualizados, sobre todo interfaces de red. No obstante, el servicio web que proporcione el clúster se va a ejecutar dentro de un contenedor, de modo que su replicación será mucho menos costosa. Por tanto, los *load balancer* se van a instalar en máquinas virtuales, y los servidores reales en Dockers.

Para validar el funcionamiento del clúster se cuenta con la aplicación de prescripciones médicas “Farhos Prescripción” proporcionada por la empresa Visual – Limes. Sin embargo, dado que el propósito del proyecto pone la nota en realizar una prueba de concepto sobre la combinación de las tecnologías anteriormente mencionadas en un clúster, no se va a entrar en detalle sobre el funcionamiento interno ni en las funcionalidades de la aplicación. Por otra parte, se pretende que la infraestructura configurada no sea invasiva en la mencionada aplicación, de modo que pueda ser una solución genérica útil para cualquier servicio web.

En el anexo a este documento puede encontrarse una breve descripción sobre la empresa y el producto.

3. Identificación y análisis de posibles soluciones

La solución para este proyecto vendrá dada por el planteamiento de la tecnología adecuada. Se garantizará un sistema altamente disponible siempre que se eliminen los puntos únicos de fallo replicando cada nodo[7]. Para conseguir optimizar el rendimiento se optará por un software que proporcione equilibrado de carga, como puede ser HAProxy o Linux Virtual Server.

Por otra parte, el servicio replicado podrá desplegarse en un entorno de máquinas virtuales, apostando por las ventajas de la creciente tecnología de contenerización o instalándose en un clásico servidor baremetal.

A continuación, se expone un análisis de las diferentes posibilidades.

3.1 HA Proxy vs. Linux Virtual Server (LVS)

El principal objetivo es que la arquitectura del sistema sea totalmente transparente para el usuario. De esta manera, éste interactúa como si fuera un único servidor.
Las opciones más comunes son:

3.1.1 Linux Virtual Server

Linux Virtual Server (LVS) forma parte del kernel de Linux. Proporciona una solución simple pero eficiente para implementar un servidor altamente disponible y escalable.
Redirige los paquetes a través de una red interna hacia los servidores que se distribuyen el trabajo actuando como nodos trabajadores. La característica más determinante para su elección es que trabaja en el nivel 4 según el modelo OSI, y redirige tráfico tanto TCP como UDP.

3.1.2 HAProxy

HAProxy es un software con una gran reputación que proporciona una solución rápida y eficiente. Actualmente es la solución adoptada por muchas plataformas como: GitHub, Twitter, o Stack Overflow. Implementa un servidor proxy para aplicaciones TCP y HTTP y puede operar, por tanto, en el nivel 4 como en el nivel 7 del modelo OSI. Adicionalmente, está dotada de una interfaz gráfica para desplegar y configurar el software.

En cualquiera de los dos casos, cuando un cliente realiza una petición se requieren dos conexiones TCP: una del cliente al LB, y otra del LB al servidor. Por este motivo, los servidores reales no conocen la dirección IP del cliente. En el supuesto de precisar la

[7] Existen otros puntos únicos de fallo como la alimentación eléctrica del sistema o el hecho de disponer de una sola interfaz de red hardware. Estos deben contemplarse en una implementación real si se quiere conseguir el máximo grado de alta disponibilidad.

información de la dirección, se considerarán distintas opciones optando por la más apropiada.

La elección se ha formulado en base a las diferencias que ofrecen en cuanto al modelo OSI. HAProxy ofrece servicio en el nivel 7, lo que amplía las posibilidades en el equilibrado de carga. Teniendo en cuenta su gran aceptación se puede considerar la solución más adecuada para el propósito de este proyecto.

3.2 KeepAlived vs. Corosync + PaceMaker

Incrementar el número de servidores reales que atienden peticiones hace que la disponibilidad del sistema aumente y con ello, se reducirá el riesgo de posibles pérdidas del sistema. A más servidores reales se reduce la probabilidad de que fallen todos al mismo tiempo.
El equilibrador de carga elegido (HAProxy) es capaz de conocer el estado de los servidores, con lo cual únicamente enviará peticiones a aquellos que estén activos.
Sin embargo, hay dos aspectos necesarios para garantizar la alta disponibilidad que no se están contemplando.

1. Monitorizar al "load balancer" para que no cese su actividad.
2. En el caso de que cese, ¿cómo se reemplaza al no estar replicado? El equilibrador de carga todavía es un punto único de fallo a solucionar.

La alternativa habitual es desplegar dos *load balancers* en configuración activo - pasivo y que un tercer agente monitorice su estado. En caso de que el maestro falle, deberá pasarle el control al nodo de respaldo (backup).
Este cambio debe ser transparente para el resto del sistema. Para conseguirlo, se ha de configurar en los *load balancer*s una dirección IP virtual (VIP) para cada interfaz. Si el nodo maestro deja de funcionar, la VIP pivota automáticamente al nodo pasivo tomando el control hasta que el primero vuelve a estar operativo.

3.2.1 Corosync + Pacemaker

Corosync es un sistema de comunicaciones encargado de gestionar la mensajería entre los nodos de un clúster. También proporciona un sencillo sistema de alta disponibilidad capaz de reiniciar un servicio de aplicación en caso de detectar un fallo.

Pacemaker es un gestor de recursos de alta disponibilidad. Es capaz de detectar fallos tanto a nivel de aplicaciones como del servidor y tomar las decisiones pertinentes en función de su configuración. Para su funcionamiento necesita proveerse del sistema de mensajeria proporcionado por Corosync.

Así, combinando ambas herramientas, el administrador del clúster es capaz de monitorizar todo el sistema, detectar errores y configurar un conjunto de acciones automáticas mediante scripts para gestionar la infraestructura.

3.2.2 Keepalived

Todo lo expuesto al comienzo del apartado 3.2 se conseguirá con: 'KeepAlived'.

KeepAlived es un software que implementa las herramientas necesarias para monitorizar el estado de un nodo sirviéndose del protocolo de elección VRRP (Virtual Redundancy Routing Protocol). [6]

VRRP asigna la responsabilidad a un nodo de entre un set de nodos disponibles en la red local. El elegido se conoce como 'master'. Dicho protocolo elimina los puntos únicos de fallo al asignar dinámicamente un nuevo responsable en caso de fallo del 'master'.

Además, KeepAlived permite el uso de scripts ajustando la monitorización a nuestras necesidades.

Como se ha visto, Corosync + Pacemaker se utiliza para compartir un conjunto de recursos compartidos, incluyendo la compartición de direcciones IP. No obstante, proporciona más funcionalidad de la necesaria en este proyecto y su configuración es más compleja que la de KeepAlived. Por esto, se ha optado por utilizar la herramienta KeepAlived.

3.3 Máquinas virtuales, Docker y Baremetal

A continuación, se explican los conceptos de 'máquina virtual' y 'contenedor' para definir sus diferencias y desvanecer posibles confusiones.

3.3.1 Máquinas Virtuales

Una máquina virtual es un sistema operativo completo ejecutándose de forma aislada sobre otro sistema operativo anfitrión, por medio de un software llamado *hipervisor* (ej. VirtualBox, VMWare…). El hipervisor, o monitor de máquinas virtuales, se encarga de controlar las máquinas haciendo que todos sistemas operativos compartan el hardware disponible.

El siguiente esquema muestra la arquitectura de las máquinas virtuales:

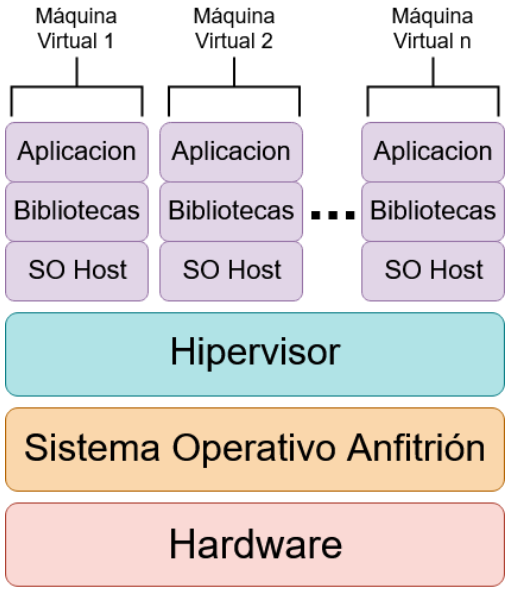

Ilustración 1. Arquitectura de las máquinas virtuales

Con esta tecnología es posible tener varios sistemas operativos ejecutándose sobre una máquina, compartiendo los recursos hardware, pero manteniendo aislados su memoria y espacio en disco.

3.3.2 Contenedores

Similar a la tecnología anterior, los contenedores también aíslan las aplicaciones en un entorno replicable, estable y portable.

[7] Sin embargo, a diferencia de las máquinas virtuales, las cuales permiten abstraer el hardware del sistema operativo, los contenedores nacieron para abstraer la aplicación en sí misma creando un entorno completamente aislado que contiene únicamente aquello que precisa para funcionar.

A continuación, se muestra un esquema similar al anterior:

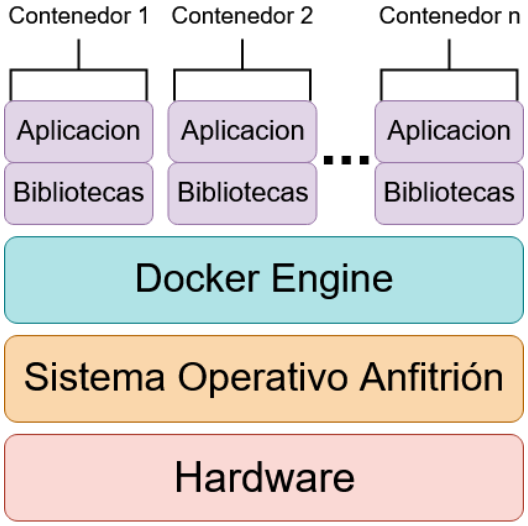

Ilustración 2. Arquitectura de los contenedores

La única diferencia aparente es la supresión de la capa del sistema operativo huésped y que *Docker*[8] *Engine* ha sustituido al *Hipervisor*. Docker Engine es el elemento encargado de lanzar y gestionar los contenedores (de forma análoga igual que hacía el hipervisor con las máquinas virtuales). Sin embargo, el concepto de tecnología es muy diferente y aquí es donde aparece la primera gran diferencia: en lugar de reservar previamente los recursos hardware de manera discreta (asignando nº de procesadores, cantidad de memoria RAM…) hace que todos los contenedores compartan los recursos del sistema operativo anfitrión ajustándolos dinámicamente a las necesidades de cada aplicación consiguiendo una optimización máxima.

Cada contenedor establece su propia estructura de directorios, posee interfaces de red y es accesible mediante dirección IP y puertos. Así, la aplicación percibe su entorno como si se estuviera ejecutando en un sistema operativo aislado.

[8] Docker. Disponible en: https://docs.docker.com/get-started/overview/

3.3.3 Bare-metal

Un servidor "bare-metal"[9] es un servidor cuyos recursos están destinados única y exclusivamente a un cliente. Bajo este concepto la aplicación estaría desplegada en un entorno con el mayor grado de aislamiento posible: todos los recursos están disponibles para ella. Además, desde el punto de vista de la seguridad, es la opción más segura ya que desaparecen todas las vulnerabilidades que puedan acompañar a un hipervisor software.

Sin embargo, dado que no disponemos de la infraestructura necesaria para montar un clúster en servidores *bare-metal* (harían falta muchas máquinas físicas independientes) se podría plantear como solución un servidor *bare-metal* con un hipervisor de tipo 1.

Un hipervisor de tipo 1 se instala directamente sobre el hardware sin ningún tipo de software ni sistema operativo que lo ejecute. Por este motivo han demostrado un rendimiento y estabilidad muy altos, ya que son muy ligeros y manejan el hardware exclusivamente para las máquinas virtuales que gestionan.

Con esta solución la máquina física en la que se ejecuta el hipervisor queda reducida exclusivamente a propósitos de virtualización. Es por esto que este modelo se encuentra mayormente en entornos empresariales, con máquinas dedicadas a este designio. Por tanto, no es una solución adecuada para la prueba a la que este proyecto acontece.

3.4 La solución adoptada

Conforme lo visto, un contenedor es capaz de proporcionar a una aplicación el mismo grado de aislamiento que una máquina virtual. Además, al contener sólo los elementos necesarios para su funcionamiento el despliegue es mucho más rápido. Por último, ya que su tamaño es mucho menor es posible mantener más instancias activas ocupando menos recursos.

Sin embargo, si nos paramos a analizar la mayoría de los servidores reales (disponibles en empresas, centros educativos, hospitales…) nos encontramos con la siguiente situación: el servidor (o servidores) es una máquina que proporciona multitud de servicios y ya cuenta con una sobrecarga añadida. Desplegar una multitud de contenedores en el sistema operativo anfitrión no es muy adecuado por la alta posibilidad de que se den conflictos de configuración con los servicios que están en funcionamiento.

Además, es necesario un detallado estudio del consumo de recursos para establecer el límite que Docker Engine puede alcanzar sin ahogar al resto de aplicaciones.

Por este motivo se considera necesario añadir una capa más e introducir los contenedores en una máquina virtual. Esta máquina virtual tendrá que ser replicada para no convertirse en un punto único de fallo.

A priori parece que no estemos obteniendo ningún beneficio con esta solución: estamos replicando máquinas virtuales, con el incremento de consumo que ello implica. Sin embargo, se puede obtener beneficio.

[9] Bare-metal Server. Disponible en: https://en.wikipedia.org/wiki/Bare-metal_server

En un clúster clásico cada máquina virtual implicará una instancia del servicio, de manera que el orden de escalado es directamente proporcional al número de máquinas virtuales. Supongamos que cada máquina requiere de 2 GB de RAM para funcionar, y la aplicación necesita 1 GB. Desplegando dos instancias el consumo total es de 6 GB de memoria RAM, y con cuatro instancias el consumo es de 12 GB.

Añadiendo Docker a la ecuación, cada máquina virtual podría tener varias instancias de la aplicación. Por ejemplo, con dos máquinas virtuales y dos instancias en cada una, el servicio estará replicado cuatro veces, lo que hará crecer el rendimiento del sistema al poder procesar más peticiones a la par que incrementa el grado de disponibilidad al haber más réplicas disponibles.

Siguiendo el planteamiento anterior, el consumo de este servicio con cuatro instancias seria de 8 GB de memoria RAM, frente a los 12 GB mencionados anteriormente.

4. Descripción de la solución adoptada

En este apartado se expone la configuración realizada para desplegar el clúster. Primero se muestra un esquema de la solución adoptada que permite obtener una idea general de la infraestructura del sistema. Después, se procede a detallar la configuración de cada herramienta utilizada.

4.1 Esquema de la solución adoptada

La solución adoptada es un clúster compuesto por cuatro nodos virtuales: dos equilibradores de carga y dos máquinas virtuales con dos contenedores cada una, esto implica que cada máquina virtual ejecuta dos instancias de la aplicación (una en cada contenedor).
El almacenamiento permanente es una base de datos ubicada en un servidor externo accesible por VPN, por lo que su instalación y configuración es ajena y no se contempla en este proyecto.
Todos los nodos ejecutan Debian 9 como sistema operativo. Se ha intentado reducir al máximo el tamaño del clúster por lo que todas las máquinas se han configurado sin interfaz gráfica.

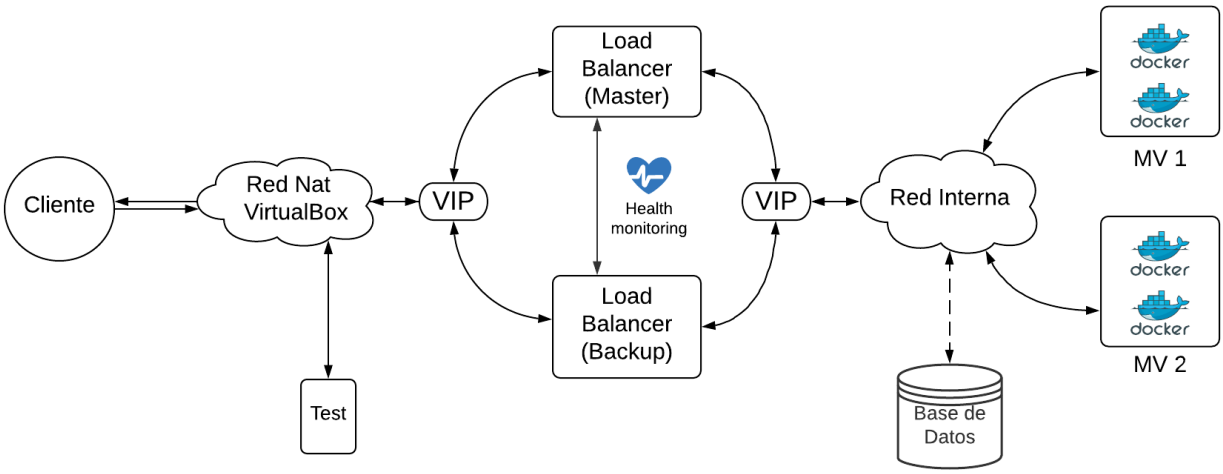

Ilustración 3. Esquema de la solución adoptada

A continuación, se muestran los pasos seguidos en la creación del clúster.

Inicialmente se han creado las máquinas virtuales en VirtualBox asignándoles la configuración hardware necesaria. Seguidamente, se procede a instalar Debian 9 en todas ellas. [1] Una vez instalado el sistema operativo, se han configurado las interfaces de red editando el fichero /etc/network/interfaces. A continuación, se muestra un extracto de la configuración realizada para las interfaces de la máquina Load Balancer Master:

```
auto enp0s3
iface enp0s3 inet static
address 10.0.2.21
netmask 255.255.255.0
gateway 10.0.2.21

auto enp0s8
iface enp0s8 inet static
address 192.168.1.5
netmask 255.255.255.0
```

Figura 1. Extracto del fichero /etc/network/interfaces

La configuración del resto de máquinas se ha realizado de acuerdo con los datos expuestos en las siguientes tablas.

Red	Dirección IP
Red NAT VirtualBox	10.0.2.0/24
Red Interna	192.168.1.0/24

Figura 2. Rango de direcciones de la red

Nodo	Dirección IP
Load Balancer Master	Interfaz enp0s3 → 10.0.2.21 VIP1 → 10.0.2.210 Interfaz enp0s8 → 192.168.1.5 VIP2 → 192.168.1.101
Load Balancer BackUp	Interfaz enp0s3 → 10.0.2.22 VIP1 → 10.0.2.210 Interfaz enp0s8 → 192.168.1.6 VIP2 → 192.168.1.101
Máquina Virtual 1	192.168.1.100
Máquina Virtual 2	192.168.1.110

Figura 3. Configuracion de las interfaces de red

Como se puede observar, los *load balancer* disponen de dos interfaces de red para conectar la red NAT (interfaz enp0s3) con la red interna (interfaz enp0s8). Ambos comparten la misma dirección IP virtual la cual se configurará desde KeepAlived más adelante.

Por otra parte, los dos servidores reales (MV1 y MV2) habilitan una interfaz de red (enp0s3) conectada a la red interna.

Una vez realizada la configuración de las interfaces de cada máquina es necesario dar acceso a la red a todas ellas. Los equilibradores de carga ya disponen de acceso a internet por estar conectados a la red NAT. Sin embargo, la red interna no da acceso al exterior por lo que es

necesario configurar los servidores reales para que puedan acceder a internet a través de las máquinas *load balancer*.

Dicha configuración se hace de la siguiente manera:

- Los servidores reales deben configurar la VIP2 como su puerta de enlace predeterminada.
- La máquina LB debe habilitar el servicio NAT:
    - Configurando de la variable del *kernel* net.ipv4.ip_forward=1.
    - Configurando en las "iptables" FORWARD ACCEPT para la interfaz enp0s3.

4.1 Configuración de Docker

Este proyecto ha sido apoyado por la empresa Visual-Limes y cuenta con una aplicación real enfocada a la prescripción médica en hospitales. No se va a hacer especial mención al funcionamiento interno de la aplicación ya que únicamente interesa conocer su funcionamiento a nivel de peticiones de red.

Funcionamiento de la aplicación

La aplicación expone cuatro servicios: dos HTTP ubicados en los puertos 9000 y 8080 y dos RMI (Remote Method Invocation)[10] ubicados en los puertos 9001 y 9002. A continuación se describe la funcionalidad que proporciona cada puerto.

- En el puerto 9000 un servidor apache proporciona el servicio HTTP necesario para descargar la aplicación por primera vez. No precisa persistencia, esto es, que todas las peticiones de un cliente sean atendidas siempre por el mismo servidor. Por esto, las peticiones HTTP se reparten siguiendo el algoritmo de round-robin.
- Los puertos 9001 y 9002 proporcionan el servicio RMI. A través de ellos, el programa "Farhos Prescripción" es capaz de intercambiar objetos con el cliente y la base de datos para ofrecer el servicio de consulta de prescripciones. A diferencia del servicio anterior, éste precisa persistencia por lo que las peticiones TCP hacen uso de las stick-tables para asignar un servidor a un cliente
- Por último, el tercer contenedor expone el puerto 8080. Ejecuta un servidor Tomcat[11] que recibe peticiones HTTP y se encarga de proporcionar un acceso directo a la prescripción de un paciente sin necesidad de ejecutar la aplicación y realizar una búsqueda. Para ello, recibe a través de los parámetros de la URL los datos del paciente necesarios para realizar la consulta a la base de datos. Tras obtener su prescripción médica asociada, construye un archivo ejecutable que inicia la aplicación y muestra directamente la prescripción del paciente deseado. Del mismo modo que en el primer servicio, las peticiones de éste se reparten siguiendo el algoritmo de round-robin.

Configuración de Docker

En cuanto a Docker, su filosofía es ubicar un microservicio por contenedor. Así, en base a dicha buena práctica y a la estructura de la aplicación se ha decidido construir tres contenedores. A

[10] Java Remote Method Invocation. Disponible en: https://es.wikipedia.org/wiki/Java_Remote_Method_Invocation
[11] Apache Tomcat. Disponible en: http://tomcat.apache.org/

continuación se detallan los pasos a seguir para desplegar un Docker y para seguidamente mostrar la configuración utilizada en el proyecto.

Para desplegar un contenedor primero es necesario crear una "imagen" a partir de un fichero llamado Dockerfile[12], el cual contiene una secuencia de instrucciones para montarla.

El funcionamiento del fichero es el siguiente: las instrucciones se ejecutan en orden y siempre debe comenzar con la directiva "FROM <image>:<tag>", que especifica la imagen base desde la cual se va a construir el contenedor. A continuación, se pueden especificar una serie de comandos mediante la directiva "RUN" para configurar la imagen según sea necesario.

Después, es posible añadir archivos al contenedor mediante "COPY <src> <dest>" o "ADD <src> <dest>". Ambas se utilizan para mover archivos desde <src> hasta la dirección en el contenedor <dest>, pero ADD permite además que el directorio de origen sea una URL.

Por último, es imprescindible añadir un comando que ejecute un servicio mediante las directivas "CMD" y/o "ENTRYPOINT". La diferencia entre ambas es la siguiente:

- ENTRYPOINT indica el ejecutable que se va a utilizar. Si no se especifica dicho comando, el valor por defecto es "/bin/sh -c".
- CMD proporciona los parámetros a utilizar en el ejecutable.

Es importante destacar que los contenedores Docker no ofrecen persistencia, es decir, cualquier dato que se haya generado durante su ejecución será destruido cuando el contenedor cese su actividad. Si se desea mantener algún dato generado (por ejemplo, los logs del servidor) es necesario utilizar la etiqueta "VOLUME" para especificar un directorio compartido entre el contenedor y el sistema operativo anfitrión.

Como se ha comentado anteriormente, se han construido tres contenedores distintos. El primero expone el puerto 9000. La siguiente imagen muestra el fichero Dockerfile utilizado para crear la imagen del contenedor llamado "traeJar".

```
FROM centos:7
ENV container docker
ENV PRESCRI_RUTA=/usr/local/prescriplant
ENV TEMP_RUTA=/tmp/

RUN yum -y update && yum -y install \
      nano \
      httpd.x86_64 \
      && rm -rf /var/cache/yum*; \
      sed  '/AddDefaultCharset/c  AddDefaultCharset  ISO-8859-1'
      /etc/httpd/conf/httpd.conf; \
      mkdir /usr/local/prescriplant; \
      rm /tmp/*;
COPY pres.conf /etc/httpd/conf.d
```

[12] Dockerfile reference. Disponible en: https://docs.docker.com/engine/reference/builder/

```
COPY client /usr/local/prescriplant/client
VOLUME [ "/sys/fs/cgroup" ]
EXPOSE 9000
ENTRYPOINT ["/usr/sbin/httpd", "-D", "FOREGROUND"]
```

Figura 4. Fichero Dockerfile del contenedor "traeJar"

El segundo contenedor expone los puertos 9001 y 9002 para proporcionar el servicio RMI.
La siguiente imagen muestra el fichero Dockerfile utilizado para crear la imagen del contenedor llamado “rmiServ”.

```
FROM centos:7
ENV container docker
ENV PRESCRI_RUTA=/usr/local/prescriplant
ENV TEMP_RUTA=/tmp/
#Preparar el sistema
#AÑADIR pserv con la ruta del jdk-8u162-linux-x64
COPY pserv /etc/init.d
COPY pserv /usr/sbin/
COPY jdk-8u162-linux-x64.rpm /tmp/
RUN rpm -ivh /tmp/jdk-8u162-linux-x64.rpm; \
      chmod +x /etc/rc.d/rc.local;\
      chmod +x /etc/init.d/pserv; \
      echo     "/bin/bash     /etc/init.d/pserv     start"     >>
      /etc/rc.d/rc.local; \
      echo "exit 0" >> /etc/rc.d/rc.local; \
      mkdir /usr/local/prescriplant; \
      yum -y install bzip2; \
      rm /tmp/*
COPY serv /usr/local/prescriplant/serv
VOLUME [ "/sys/fs/cgroup" ]
EXPOSE 9001
EXPOSE 9002
CMD ["/usr/sbin/init"]
```

Figura 5. Fichero Dockerfile del contenedor "rmiServ"

El tercer contenedor expone el puerto 8080. La siguiente figura es un extracto del fichero Dockerfile utilizado para crear la imagen “farhosIn”.

```
#COPY WAR FILE
COPY tomcat-users.xml /usr/local/tomcat/conf
COPY farhosin.xml /usr/local/tomcat/
COPY FarhosIn.war /usr/local/tomcat/temp
RUN mkdir /usr/local/tomcat/webapps/FarhosIn; \
      unzip usr/local/tomcat/temp/FarhosIn.war -d
      /usr/local/tomcat/webapps/FarhosIn; \
      rm -rf usr/local/tomcat/temp/FarhosIn.war

COPY lanzador.jnlp /usr/local/tomcat/webapps/FarhosIn
EXPOSE 8080
```

```
CMD ["catalina.sh", "run"]
```

Figura 6. Extracto del fichero Dockerfile del contenedor "farhosIn"

En cuanto la forma de acceder a los contenedores, el modo más común es mediante mapeo de puertos, es decir, un puerto de la máquina anfitriona se mapea a un puerto del contenedor Docker y el acceso se hace a través de la dirección <IPHost>:<Puerto>.

En este proyecto se han desplegado dos instancias de cada contenedor por lo que se podrían elegir dos puertos al azar. Sin embargo, si queremos que el clúster haga una gestión *elástica* de la carga (una aproximación mucho más cercana a una situación real) es necesario establecer una función que permita asignar puertos dinámicamente. De este modo, el sistema será capaz de levantar o destruir tantos nodos como necesite acorde a la carga de peticiones en un momento dado.

Se ha planteado la siguiente función:

Contenedor 1 (JAR)	JAR + x	*JAR = 20 000 x = nº instancias
Contenedor 2 (RMI)	RMI1 + y RMI2 + y	*RMI1 = 21 000 RMI2 = 21 500 y = nº instancias
Contenedor 3 (TOMCAT)	TOMCAT + z	*TOMCAT = 22 000 z = nº instancias

Figura 7. Asignación de puertos a los contenedores

La siguiente imagen ilustra la configuración de puertos utilizada en el proyecto:

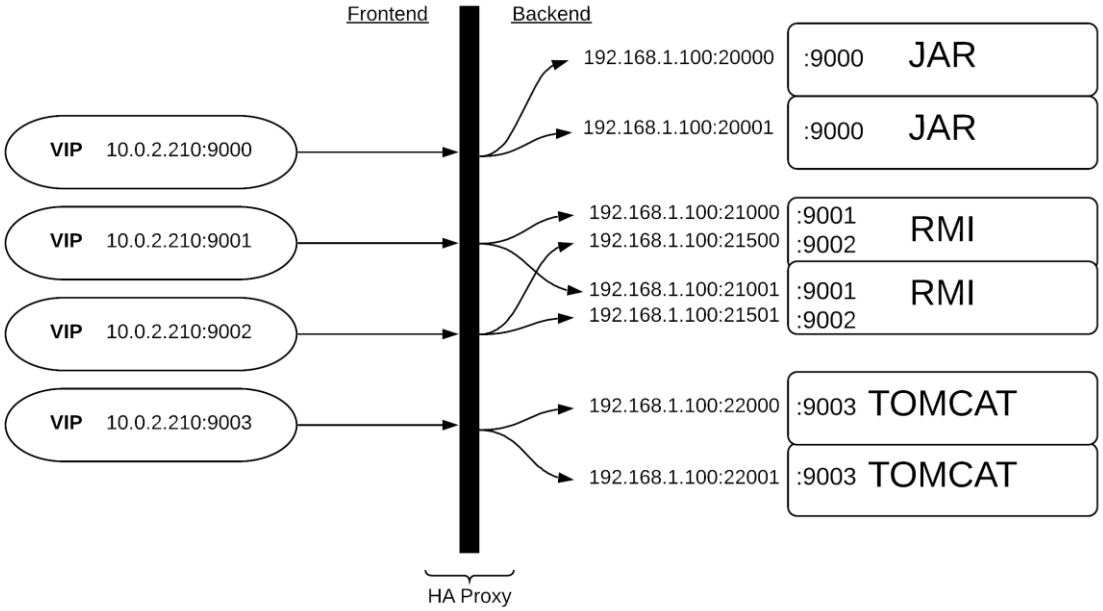

Ilustración 4. Esquema de la configuracion de puertos utilizada

4.2 Configuración de HA Proxy

HAProxy es una herramienta que ofrece muchas posibilidades de configuración [10] para proporcionar un equilibrado de carga a medida de las necesidades de cada usuario. Toda la configuración de HAProxy se especifica en el fichero "/etc/haproxy/haproxy.cfg" el cual se divide en secciones.

En los siguientes apartados se muestra la configuración adoptada y una breve descripción de cada sección.

`frontend <tag>`

Cada sección *frontend* detalla la configuración relativa a un servicio expuesto a los clientes. Las directivas utilizadas son *bind*, *mode* y *default_backend*.

- `bind <IP>:<port>`

  La directiva bind soporta varios tipos de configuración especificados como argumento en la misma línea. Para este caso, el argumento <IP>:<port> refiere a la dirección IP virtual y puerto por la cual será accesible el servicio en particular.

- `mode <http>|<tcp>`

  Si no se especifica el modo de equilibrado de carga por defecto es http (nivel 7). Si se especifica tcp HAProxy trabajará en el nivel 4.

- `default backend <backend name>`

  Especifica el *backend* asociado al servicio que define el *frontend.*

La siguiente figura muestra la configuración adoptada.

```
frontend traeJar

     bind 10.0.2.210:9000
     default_backend servers-traeJar

frontend rmiServ1

     mode tcp
     option tcplog
     log global
     bind 10.0.2.210:9001
     default_backend servers-rmiServ1

frontend rmiServ2

     mode tcp
     option tcplog
     log global
     bind 10.0.2.210:9002
     default_backend servers-rmiServ2

frontend farhosIn

     bind 10.0.2.210:8080
     default_backend servers-farhosIn
```

Figura 8. Extracto fichero haproxy.cfg. Configuración de la sección "frontend"

Se han definido cuatro servicios:

1. traeJar con dirección 10.0.2.210:9000 y modo de equilibrado http.
2. rmiServ1 con dirección 10.0.2.210:9001 y modo de equilibrado tcp.
3. rmiServ2 con dirección 10.0.2.210:9002 y modo de equilibrado tcp.
4. traeJar con dirección 10.0.2.210:8080 y modo de equilibrado http.

`backend <tag>`

Cada sección *backend* define el conjunto de servidores reales que darán servicio a una entrada *frontend*.

- `balance <mode>`
  Especifica el algoritmo de equilibrado de carga.
- `mode <http>|<tcp>`
  Define el modo en que trabajan los servidores reales. Si no se especifica, el valor por defecto es http (nivel 7). Si se especifica 'tcp' los servidores recibirán peticiones tcp (nivel 4).
- `server <tag> <IP:port> [check]`
  Declara un servidor real.

Para cada entrada *frontend* se ha definido una sección *backend* con los servidores encargados de dar cada servicio, tal y como se puede ver en la figura que se muestra a continuación:

```
backend servers-traeJar

     balance roundrobin
     server traejar1 192.168.1.100:20000 check
     server traejar2 192.168.1.100:20001 check
     server traejar11 192.168.1.110:20000 check
     server traejar22 192.168.1.110:20001 check


backend servers-rmiServ1

     mode tcp
     balance roundrobin
     stick-table type ip size 1m expire 1000s peers mypeers
     server rmi1 192.168.1.100:21000 check
     server rmi2 192.168.1.100:21001 check
     server rmi11 192.168.1.110:21000 check
     server rmi22 192.168.1.110:21001 check


backend servers-rmiServ2

     mode tcp
     balance roundrobin
     stick-table type ip size 1m expire 1000s peers mypeers
     server rmi1 192.168.1.100:21500 check
     server rmi2 192.168.1.100:21501 check
     server rmi11 192.168.1.110:21500 check
     server rmi22 192.168.1.110:21501 check


backend servers-farhosIn

     balance roundrobin
     server farhosIn1 192.168.1.100:22000 check
     server farhosIn2 192.168.1.100:22001 check
     server farhosIn11 192.168.1.110:22000 check
     server farhosIn22 192.168.1.110:22001 check
```

Figura 9. Extracto fichero haproxy.cfg. Configuración de la sección "backend"

Los servidores RMI manejan las peticiones de la aplicación una vez que el usuario se ha autenticado con sus credenciales. Por este motivo es necesario garantizar la *afinidad* de un usuario con un mismo servidor. Es decir, todas las peticiones que realice un usuario deben ser atendidas por el servidor que lo ha autenticado en la aplicación, de lo contrario la sesión se perdería en cada cambio de servidor.
Como solución se ha hecho uso del mecanismo de *stick-tables* que proporciona HAProxy. Una *stick-table* permite almacenar en sus filas datos de cada petición para después establecer reglas de equilibrado de carga.

En concreto se han configurado dos tablas diferentes, una para cada sección rmi-frontend, las cuales almacenan la dirección IP origen de cada petición y la asignan a un solo servidor del conjunto de servidores especificado.

Es importante destacar que, dado el diseño propio de la aplicación, dos secciones RMI diferentes hacen referencia a dos servicios RMI que llevan a cabo operaciones distintas. Por este motivo, cada uno de ellos apunta a un conjunto de servidores backend que no debe confundirse.

El atributo "peers" hace referencia a la breve sección de peers. Con esto es posible compartir el contenido de las tablas entre los equilibradores de carga maestro-backup. Esta configuración es útil para que cuando uno de ellos falle la instancia de respaldo pueda disponer del contenido de la tabla maestra.

```
peers mypeers
     peer debian 10.0.2.22:10000
```

Figura 10. Extracto fichero haproxy.cfg. Configuración de la sección "peers"

4.3 Configuración de KeepAlived

Del mismo modo que ocurre con HAProxy, KeepAlived también se configura a través de un fichero dividido en secciones. Las más relevantes se exponen a continuación:

`vrrp_instance <String>`

Esta sección define y configura una interfaz de red como interfaz virtual para el protocolo VRRP. A continuación, se muestra la configuración utilizada en el nodo Maestro:

```
vrrp_instance VI_1 {

     state MASTER
     interface enp0s3
     virtual_router_id 51
     priority 101

#En esta dirección se encuentra el servicio de LB
     virtual_ipaddress {
             10.0.2.210
     }
}

vrrp_instance VI_2 {

     state MASTER
```

```
    interface enp0s8

    virtual_router_id 52
    priority 101

    virtual_ipaddress {
            192.168.1.101
    }
}
```

Figura 11. Extracto fichero keepalived.conf. Configuración de la sección "vrrp_instance"

En este nodo se han configurado sus dos interfaces de red (enp0s3 y enp0s8) con las direcciones ip virtuales 10.0.2.210 y 192.168.1.101, respectivamente. Además, se ha establecido una prioridad superior a la configurada en el nodo de respaldo. Este requisito es imprescindible, ya que el protocolo elije como "Maestra" la instancia que notifica en los paquetes de difusión ARP una prioridad superior.

vrrp_sync_group <String>

En esta sección se definen el conjunto de interfaces virtuales cuyas VIPs migrarán al nodo de respaldo en caso de fallo. Su cometido principal es garantizar que ambas interfaces siempre migrarán su VIP a la vez. Así, en caso de que en una de ellas se detecte un cambio en las prioridades notificadas por los paquetes ARP y determine un cambio de estado entre los nodos, dicho cambio se realizará en ambas interfaces de forma sincronizada.

```
vrrp_sync_group SG1 {
    group {
            VI_1
            VI_2
    }
}
```

Figura 12. Extracto fichero keepalived.conf. Configuración de la sección "vrrp_sync_group"

4.4 Configuración adicional

El proceso de HAProxy se encuentra en estado "running" en ambas máquinas desde su inicio. Sin embargo, solo la máquina maestra tiene la dirección VIP establecida en la interfaz de red, lo que causará un error en el nodo de respaldo al intentar vincularse a una dirección IP que no tiene asignada. Para solucionar este problema es necesario habilitar la variable de *kernel* "net.ipv4.ip_nonlocal_bind".[13]

Para hacerlo se ha editado el fichero /etc/sysctl.conf añadiendo la siguiente línea:

```
net.ipv4.ip_nonlocal_bind = 1
```

[13] **Packet forwarding and nonlocal binding.** Consultado en: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/load_balancer_administration/s1-initial-setup-forwarding-vsa

5.Evaluación de la solución

La evaluación del funcionamiento del clúster de alta disponibilidad se va a realizar desde dos perspectivas. La primera, cualitativa, va a validar el funcionamiento de la alta disponibilidad. En concreto, se realizarán pruebas para comprobar que el servicio continúa estando disponible ante la caída de cualquier nodo, además de que el reparto de carga se está llevando a cabo de acuerdo con los parámetros definidos. En segundo lugar, mediante un conjunto de pruebas cuantitativas se extraerán estadísticas sobre el consumo de recursos para cuantificar el hardware necesario.

5.1 Evaluación cualitativa

Para la evaluación cualitativa se han realizado distintas pruebas que verifican el funcionamiento de los tres puntos clave de la alta disponibilidad y el equilibrado de carga. Primero, se va a comprobar que HAProxy está repartiendo la carga de acuerdo con la configuración establecida. A continuación, para validar la alta disponibilidad, se va a simular la parada tanto de un servidor de aplicación como de un equilibrador de carga estando el clúster en reposo, es decir, sin ninguna petición activa. Por último, se van a repetir las pruebas anteriores con peticiones en curso para observar si la aplicación es capaz de responder correctamente ante la pérdida de algún paquete.

5.1.1 Validación del equilibrado de carga

Como se ha mencionado en la sección cuatro, la aplicación utiliza los protocolos HTTP y TCP. Las peticiones HTTP reparten siguiendo el algoritmo Round-Robin. Las peticiones TCP, sin embargo, hacen uso de las stick-tables para asignar un servidor a un cliente.

Este funcionamiento se puede comprobar de varias formas:

- Modificar el programa para que incluya en las respuestas el identificador del servidor que atiende la petición. Dado que se está desarrollando una solución no invasiva, toda modificación de la aplicación queda descartada.
- Analizar la página de estadísticas de HAProxy y deducir el funcionamiento por inspección visual. Esta opción no se contempla por ser poco rigurosa.
- Analizar el archivo de log de HAProxy, ubicado en el directorio /var/log/haproxy.log, donde se registra cada petición y el servidor que la atiende. Se ha escogido esta opción para hacer un seguimiento de las peticiones y comprobar que el reparto es correcto.
- Acceder a las métricas de HAProxy a través del *socket* Unix [8]. Este medio se va a utilizar para consultar el contenido de las stick-tables y comprobar la asignación de servidores.

En primer lugar, se han realizado diez peticiones HTTP a la aplicación. Observando el archivo de logs del equilibrador puede observarse como cada petición ha sido atendida por un servidor siguiendo el algoritmo Round-Robin. El nombre de los servidores que sirven las peticiones HTTP son traejar1, traejar2, traejar 11 y traejar22, tal y como se puede observar en el archivo de configuración de HAProxy (Figura 9).

```
Jun 15 13:50:50 debian haproxy[2513]: 10.0.2.200:50229
[15/Jun/2020:13:50:50.727] traeJar servers-traeJar/traejar2
0/0/0/2/2 404 400 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 13:50:53 debian haproxy[2513]: 10.0.2.200:50229
[15/Jun/2020:13:50:53.048] traeJar servers-traeJar/traejar11
0/0/0/2/2 404 397 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 13:50:53 debian haproxy[2513]: 10.0.2.200:50229
[15/Jun/2020:13:50:53.150] traeJar servers-traeJar/traejar22
0/0/1/0/1 404 400 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 13:51:10 debian haproxy[2513]: 10.0.2.200:50229
[15/Jun/2020:13:51:10.868] traeJar servers-traeJar/traejar1
0/0/1/0/1 404 397 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 13:51:10 debian haproxy[2513]: 10.0.2.200:50228
[15/Jun/2020:13:51:10.870] traeJar servers-traeJar/traejar2
0/0/1/0/1 404 400 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 13:51:12 debian haproxy[2513]: 10.0.2.200:50228
[15/Jun/2020:13:51:12.744] traeJar servers-traeJar/traejar11
0/0/1/4/5 404 397 - - ---- 3/2/1/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 13:51:12 debian haproxy[2513]: 10.0.2.200:50229
[15/Jun/2020:13:51:12.748] traeJar servers-traeJar/traejar22
0/0/1/1/2 404 400 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 13:51:14 debian haproxy[2513]: 10.0.2.200:50229
[15/Jun/2020:13:51:14.786] traeJar servers-traeJar/traejar1
0/0/0/1/1 404 400 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 13:51:14 debian haproxy[2513]: 10.0.2.200:50228
[15/Jun/2020:13:51:14.788] traeJar servers-traeJar/traejar2
0/0/0/1/1 404 397 - - ---- 3/2/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 13:51:28 debian haproxy[2513]: 10.0.2.200:50228
[15/Jun/2020:13:51:28.903] traeJar servers-traeJar/traejar11
0/0/1/1/2 404 397 - - ---- 3/2/1/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"
```

Figura 13. Extracto del log de HAProxy. Prueba balanceo Round-Robin

Por otra parte, una vez el usuario se autentica en la aplicación se le debe asignar un servidor para evitar que pierda los datos de su sesión. Para verificar este comportamiento se ha ejecutado la aplicación y consultado la prescripción de un paciente desde dos

máquinas distintas. Si observamos el archivo de logs de HAProxy podemos verificar que cada cliente se ha asignado a un servidor, y siempre es atendido por el mismo.

```
Jul 14 17:21:20 debian haproxy[586]: 10.0.2.221:49753
[14/Jul/2020:17:21:02.791] rmiServ2 servers-rmiServ2/rmi1
1/0/18030 271 – 2/1/1/1/0 0/0

Jul 14 17:21:57 debian haproxy[586]: 10.0.2.221:49749
[14/Jul/2020:17:20:00.428] rmiServ2 servers-rmiServ2/rmi1
1/0/117140 1830470 cD 1/0/0/0/0 0/0

Jul 14 17:21:57 debian haproxy[586]: 10.0.2.221:49749
[14/Jul/2020:17:20:00.428] rmiServ2 servers-rmiServ2/rmi1
1/0/117140 1830470 cD 1/0/0/0/0 0/0

Jul 14 17:23:35 debian haproxy[586]: 10.0.2.221:49754
[14/Jul/2020:17:22:00.739] rmiServ2 servers-rmiServ2/rmi1
1/0/94261 144613 – 1/0/0/0/0 0/0

Jul 14 17:23:35 debian haproxy[586]: 10.0.2.221:49754
[14/Jul/2020:17:22:00.739] rmiServ2 servers-rmiServ2/rmi1
1/0/94261 144613 – 1/0/0/0/0 0/0

Jul 14 17:24:02 debian haproxy[586]: 10.0.2.220:49762
[14/Jul/2020:17:24:02.855] rmiServ2 servers-rmiServ2/rmi2
1/0/50 1331 – 1/0/0/0/0 0/0

Jul 14 17:24:02 debian haproxy[586]: 10.0.2.220:49762
[14/Jul/2020:17:24:02.855] rmiServ2 servers-rmiServ2/rmi2
1/0/50 1331 – 1/0/0/0/0 0/0

Jul 14 17:24:17 debian haproxy[586]: 10.0.2.220:49774
[14/Jul/2020:17:24:17.009] rmiServ2 servers-rmiServ2/rmi2
1/0/8 1060 – 1/0/0/0/0 0/0

Jul 14 17:24:17 debian haproxy[586]: 10.0.2.220:49774
[14/Jul/2020:17:24:17.009] rmiServ2 servers-rmiServ2/rmi2
1/0/8 1060 – 1/0/0/0/0 0/0

Jul 14 17:24:18 debian haproxy[586]: 10.0.2.220:49775
[14/Jul/2020:17:24:18.355] rmiServ2 servers-rmiServ2/rmi2
1/0/59 1060 – 1/0/0/0/0 0/0
```

Figura 14. Extracto del log de HAProxy. Afinidad.

Como se puede observar en el extracto anterior, al cliente con dirección IP 10.0.2.221 se le ha asignado el servidor rmi1, y al cliente con dirección IP 10.0.2.210 se le ha asignado el servidor rmi2.

Como se ha mencionado anteriormente, es posible acceder a las métricas de HAProxy a través del socket Unix. Para habilitar esta opción es necesario añadir las siguientes líneas al fichero de configuración de HAProxy:

```
global

    # [...]

    stats socket /run/haproxy/haproxy.sock mode 660 level admin
    stats timeout 2m # Wait up to 2 minutes for input
```

Figura 15. Configuración HAProxy - Socket Unix

Aunque es posible consultar las métricas a través de la consola de comandos, la herramienta de gestión HATop[14] facilita el trabajo. Así, a través de dicha herramienta, es posible obtener el contenido de las stick-tables y verificar la asignación de servidores.

```
* Tue Jul 14 17:27:26 2020
> show table servers-rmiServ1
#table: servers-rmiServ1, type: ip, size: 1038576, used:2
0x5651127c71e4: key=10.0.2.220 use=0 exp=849414 server_id=2
0x5651127c2d04: key=10.0.2.221 use=0 exp=552778 server_id=1

* Tue Jul 14 17:27:34 220
> show table servers-rmiServ2
#table: servers-rmiServ2, type: ip, size: 1048576, used: 2
0x5651127c2e44: key=10.0.2.220 use=0 exp=841935 server_id=2
0x5651127c2da4: key=10.0.2.221 use=0 exp=665856 server_id=1
```

Figura 16. Contenido de las stick-tables de HAProxy

5.1.2 Validación de la alta disponibilidad

Eliminar los puntos únicos de fallo permite garantizar que, ante el cese de actividad de un nodo, el servicio siga estando disponible. A partir de aquí, cuantas más réplicas se podrá garantizar mayor disponibilidad del servicio.

En este proyecto todos los nodos se han replicado una vez, por lo que ante la parada de uno de ellos la otra instancia podrá seguir atendiendo peticiones. Para validar esta característica es necesario verificar el comportamiento del clúster ante una parada tanto en la parte de los servidores como en el equilibrador de carga.

Se pueden dar dos escenarios: el primero, que el clúster se encuentre en reposo y no esté procesando ninguna petición en ese instante. El segundo, que el fallo se produzca mientras se están llevando a cabo peticiones.

Para ambos escenarios se van a realizar pruebas que permitan observar el comportamiento tanto del conjunto de servidores como de los equilibradores de carga.

[14] HATop – Interactive ncurses client for HAProxy. Disponible en: http://feurix.org/projects/hatop/

5.1.2.1 Pruebas en reposo

Simulación de interrupción de un servidor

Se han realizado cinco peticiones HTTP con las cuatro réplicas de los servidores "*traejar*" activas. A continuación, se ha forzado la parada de una réplica, tal y como puede observarse en el mensaje: "traejar1 is DOWN". Sin embargo, el servicio sigue disponible ya que el resto de las peticiones son atendidas por los servidores *traejar2, traejar11, traejar22.*

```
Jun 15 18:22:54 debian haproxy[5155]: 10.0.2.220:49724
[15/Jun/2020:18:22:54.722] traeJar servers-traeJar/traejar2
0/0/1/1/2 404 400 - - ---- 2/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 18:23:10 debian haproxy[5155]: 10.0.2.220:49724
[15/Jun/2020:18:23:10.924] traeJar servers-traeJar/traejar11
0/0/1/0/1 404 397 - - ---- 2/2/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 18:23:10 debian haproxy[5155]: 10.0.2.220:49724
[15/Jun/2020:18:23:10.928] traeJar servers-traeJar/traejar22
0/0/0/1/1 404 400 - - ---- 2/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 18:23:12 debian haproxy[5155]: 10.0.2.220:49724
[15/Jun/2020:18:23:12.921] traeJar servers-traeJar/traejar1
0/0/1/0/1 404 397 - - ---- 2/2/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 18:23:12 debian haproxy[5155]: 10.0.2.220:49724
[15/Jun/2020:18:23:12.926] traeJar servers-traeJar/traejar2
0/0/0/1/1 404 400 - - ---- 2/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 18:23:44 debian haproxy[5155]: Server servers-
traeJar/traejar1 is DOWN, reason: Layer4 connection problem,
info: "Connection refused", check duration: 0ms. 1 active and
0 backup servers left. 0 sessions active, 0 requeued, 0
remaining in queue.

Jun 15 18:23:44 debian haproxy[5155]: Server servers-
traeJar/traejar1 is DOWN, reason: Layer4 connection problem,
info: "Connection refused", check duration: 0ms. 1 active and
0 backup servers left. 0 sessions active, 0 requeued, 0
remaining in queue.

Jun 15 18:23:53 debian haproxy[5155]: 10.0.2.220:49752
[15/Jun/2020:18:23:53.087] traeJar servers-traeJar/traejar2
0/0/1/0/1 404 397 - - ---- 1/1/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"
```

```
Jun 15 18:23:53 debian haproxy[5155]: 10.0.2.220:49752
[15/Jun/2020:18:23:53.093] traeJar servers-traeJar/traejar11
0/0/0/1/1 404 400 - - ---- 1/1/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"

Jun 15 18:23:55 debian haproxy[5155]: 10.0.2.220:49752
[15/Jun/2020:18:23:55.246] traeJar servers-traeJar/traejar22
0/0/0/0/0 404 397 - - ---- 1/1/0/1/0 0/0 "GET
/prescriplant/css/images/ajax-loader.png HTTP/1.1"

Jun 15 18:23:55 debian haproxy[5155]: 10.0.2.220:49752
[15/Jun/2020:18:23:55.252] traeJar servers-traeJar/traejar2
0/0/0/1/1 404 400 - - ---- 2/2/0/1/0 0/0 "GET
/prescriplant/css/images/icons-18-white.png HTTP/1.1"
```

Figura 17. Extracto del log de HAProxy. Prueba de parada de un servidor

Simulación de interrupción del equilibrador de carga

HAProxy se encuentra configurado en modo activo - pasivo para evitar convertirse en un punto único de fallo. Con esta configuración, el nodo maestro es el único nodo activo y accesible a través de la dirección IP virtual establecida mientras que el nodo backup se encuentra en reposo a la espera de sustituir al maestro en caso de fallo.

Con ambas máquinas activas, Keepalived configura las dos interfaces de red del Maestro con las direcciones IP virtual especificadas. Esto puede comprobarse a través del comando “ip addr”, el cual ofrece la siguiente salida:

```
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:9a:40:35 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.21/24 brd 10.0.2.255 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet 10.0.2.210/32 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe9a:4035/64 scope link
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:b5:94:bf brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.5/24 brd 192.168.1.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet 192.168.1.101/32 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:feb5:94bf/64 scope link
       valid_lft forever preferred_lft forever
```

Figura 18. Comando "ip addr" en LB Maestro

En la figura anterior puede observarse como el LB Maestro posee dos direcciones IP en cada interfaz de red: primero su propia dirección estática configurada manualmente, y después la VIP.

KeepAlived emite continuamente hacia toda la red local paquetes ARP que informan del identificador de la interfaz de red virtual y su prioridad, de modo que siempre se elige nodo principal aquel de mayor prioridad. En caso de fallo del maestro, estos paquetes dejan de emitirse, por tanto, el nodo de respaldo pasará a tener la mayor prioridad. Así, KeepAlived lleva a cabo la conmutación de las VIP a la máquina LB Backup. La siguiente figura muestra cómo el programa detecta el fallo y realiza el cambio instantáneamente.

```
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_1) Backup received
priority 0 advertisement
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_2) Backup received
priority 0 advertisement
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_1) Receive
advertisement timeout
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_2) Receive
advertisement timeout
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_2) Entering MASTER
STATE
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_2) setting VIPs.
Jun 18 17:46:15 debian Keepalived_vrrp[567]: Sending gratuitous ARP
on enp0s8 for 192.168.1.101
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_2)
Sending/queueing gratuitous ARPs on enp0s8 for 192.168.1.101
Jun 18 17:46:15 debian Keepalived_vrrp[567]: Sending gratuitous ARP
on enp0s8 for 192.168.1.101
Jun 18 17:46:15 debian Keepalived_vrrp[567]: VRRP_Group(SG1)
Syncing instances to MASTER state
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_1) Entering MASTER
STATE
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_1) setting VIPs.
Jun 18 17:46:15 debian Keepalived_vrrp[567]: Sending gratuitous ARP
on enp0s3 for 10.0.2.210
Jun 18 17:46:15 debian Keepalived_vrrp[567]: (VI_1)
Sending/queueing gratuitous ARPs on enp0s3 for 10.0.2.210
Jun 18 17:46:15 debian Keepalived_vrrp[567]: Sending gratuitous ARP
on enp0s3 for 10.0.2.210
Jun 18 17:46:15 debian Keepalived_vrrp[567]: Sending gratuitous ARP
on enp0s3 for 10.0.2.210
```

Figura 19. Extracto del log de KeepAlived en el nodo BackUp ante la caída del nodo maestro.

Como se puede observar, las interfaces de red enp0s3 y enp0s8 (correspondientes a la interfaz virtual VI_1 y VI_2, respectivamente) han detectado el cambio de prioridades. Después, han cambiado su estado a “MASTER” y se les ha asignado la VIP. Con el comando “ip addr” se puede observar como, desde ahora, el nodo backup ha adoptado el rol del Maestro.

```
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:49:38:88 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.22/24 brd 10.0.2.255 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet 10.0.2.210/32 scope global enp0s3
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe49:3888/64 scope link
       valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:aa:af:94 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.6/24 brd 192.168.1.255 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet 192.168.1.101/32 scope global enp0s8
       valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:feaa:af94/64 scope link
       valid_lft forever preferred_lft forever
```

Figura 20. Salida de "ip addr" en la máquina LB BackUp.

Mientras el nodo Maestro no vuelva a estar activo, la máquina de respaldo mantendrá configuradas sus dos interfaces de red con dos direcciones IP: estática y VIP. De este modo, el servicio sigue operativo siendo el fallo totalmente transparente para el usuario y el resto de las máquinas que interactúen con el equilibrador de carga.

Cuando se recupere el nodo principal volverá a emitir paquetes ARP informando de una prioridad superior. Entonces, el nodo de respaldo volverá a pasar a estado "BACKUP" y las VIP conmutarán al nodo Maestro. La siguiente figura ilustra este cambio.

```
Jun 18 19:23:19 debian Keepalived_vrrp[567]: (VI_2) Master received
advert from 192.168.1.5 with higher priority 101, ours 50
Jun 18 19:23:19 debian Keepalived_vrrp[567]: (VI_2) Entering BACKUP
STATE
Jun 18 19:23:19 debian Keepalived_vrrp[567]: (VI_2) removing VIPs.
Jun 18 19:23:19 debian Keepalived_vrrp[567]: VRRP_Group(SG1)
Syncing instances to BACKUP state
Jun 18 19:23:19 debian Keepalived_vrrp[567]: (VI_1) Entering BACKUP
STATE
Jun 18 19:23:19 debian Keepalived_vrrp[567]: (VI_1) removing VIPs.
```

Figura 21. Extracto del log de KeepAlived. Recuperación del nodo principal.

Se ha comprobado que, ante una caída de cualquier nodo, el servicio sigue activo para nuevas peticiones. Sin embargo, queda por comprobar si las peticiones que había en curso se llegan a resolver, o de lo contrario se pierden hasta agotar su *timeout.*

Para resolver esta hipótesis se van a realizar dos pruebas, una para observar el comportamiento de la aplicación en la parte HTTP, y otra para la parte TCP.

5.1.2.2 Pruebas con peticiones en curso

Para testar este escenario se va a simular una parada del equilibrador de carga mientras haya peticiones en curso con el objetivo de observar el comportamiento de la aplicación ante la pérdida de paquetes.

"Failover" con petición HTTP en curso.

Se va a ejecutar el programa de Farhos Prescripción, habiendo eliminado todos los archivos de la caché de Java. La aplicación está configurada para comprobar que se dispone en local de todas las librerías necesarias que aseguran su correcto funcionamiento. Al iniciar la aplicación habiendo eliminado todos los recursos locales de Java se hace una petición GET al servidor que iniciará la descarga de las librerías. Mientras se lleve a cabo la descarga de los archivos .jar se va a simular una caída del equilibrador de carga para comprobar la respuesta de la aplicación.

Al realizar la prueba, el programa ha notificado una pérdida de conexión. Sin embargo, la descarga ha quedado bloqueada y la aplicación se ha cerrado tras mostrar el mensaje de error. En base a este comportamiento se puede deducir que no es capaz de soportar interrupciones durante una petición HTTP en curso.

"Failover" con petición TCP en curso

Tras iniciar sesión en la aplicación todas las acciones de consulta de prescripciones médicas se llevan a cabo como peticiones RMI. El protocolo RMI (Java Remote Method Invocation) trabaja sobre el protocolo TCP. Por ello, para validar esta prueba se va a obtener la prescripción de un paciente de prueba y a simular una parada en los servidores TCP que estaban atendiendo esa petición. Recordemos que una vez el usuario se autentica en la aplicación se le asigna un servidor para evitar perder los datos de su sesión.

Al realizar la prueba el programa ha quedado bloqueado. Tras agotar un "timeout" se ha notificado un "error en la obtención de prescripción".

Los resultados de las dos últimas pruebas permiten deducir que la aplicación FarHos no implementa mecanismos para reintentar peticiones perdidas, ya que de lo contrario tras agotar un *timeout* el programa emitiría una nueva petición, la cual sería atendida por el equilibrador de carga de respaldo.
Dado que no se tiene acceso al código fuente de la aplicación, y el proyecto pretende desarrollar una solución no invasiva, es imprescindible establecer como precondición que no existe tratamiento posible para que la pérdida de una petición en curso no implique un error en la acción que se está llevando a cabo en ese instante. Dicho en otras palabras, no se puede garantizar el correcto funcionamiento ante la pérdida de una petición en curso a causa de la caída de un nodo.
Esta implicación reduce el grado de alta disponibilidad de la solución adoptada. Para poder garantizar la más estricta disponibilidad del servicio es necesario revisar los mecanismos de tolerancia a fallos que implementa actualmente la aplicación y readaptarlos a las necesidades del proyecto.

5.2 Evaluación cuantitativa

A continuación, se van a hacer pruebas que permitan medir las prestaciones del clúster. Además, se va a evaluar el comportamiento de Docker frente a las máquinas virtuales. En concreto, se van a realizar tres tipos de pruebas:

- Primero se va a utilizar el conocido programa de evaluación de servidores web Apache Benchmark[15] para extraer datos sobre el tiempo de respuesta y las peticiones por segundo que el clúster es capaz de procesar.
- Segundo, se va a analizar el consumo de CPU al procesar un elevado número de peticiones y la memoria requerida para distintas configuraciones que se detallan más adelante.
- Tercero, se va a obtener el tiempo de despliegue de una máquina virtual y de un contenedor para su comparación.

La aplicación Farhos es útil para ilustrar las prestaciones obtenidas en cuanto a alta disponibilidad, ya que al proporcionar diferentes servicios permite explorar los dos modos de configuración de HAProxy (modo HTTP y modo TCP) así como el uso de las stick-tables.

Sin embargo, Farhos Prescripción no es útil para evaluar la capacidad computacional del clúster. Esto es debido a que la carga de trabajo inducida por la aplicación es muy baja, ya que las peticiones no requieren cálculo complejo. Más aún, al requerir de una conexión a una base de datos externa, un número elevado de peticiones puede provocar que la red se convierta en el cuello de botella, haciendo ininterpretables las estadísticas obtenidas.

Por ello, para este apartado se ha implementado un programa en PHP que genere una carga de CPU ajustables. En particular, se ha desarrollado un sencillo programa para resolver integrales definidas. Por simplicidad, la función a integrar y los límites de integración están definidos en el código. Para ajustar el tiempo de ejecución se le pasa el número de intervalos para hacer el sumatorio. El código del programa puede consultarse en el Anexo I.

Dicha función es accesible vía HTTP a través de la siguiente URL:

```
http://<VIP>:<puerto>/integral.php?n=<intervalos>
```

Donde <VIP>:<puerto> es la dirección IP virtual y el puerto de la nueva sección frontend definida en HAProxy, la cual da acceso al conjunto de servidores reales configurados como contenedores Docker.

<intervalos> es el número de intervalos utilizados para realizar el sumatorio. Este parámetro permite ajustar el tiempo de ejecución ya que cuanto mayor es el intervalo, más costoso es el cálculo que realizar.

[15] Ab – Apache HTTP Server benchmarking tool. Disponible en: https://httpd.apache.org/docs/2.4/programs/ab.html

5.2.1 Descripción del entorno de evaluación

Se van a realizar tres tipos de pruebas en función de la carga inducida en el servidor. Así, las pruebas de alta carga comprenden un conjunto de peticiones lo suficientemente costosas como para ocupar el 100% de la CPU.

- Baja carga: se consigue un tiempo de respuesta muy bajo con un número de intervalo n = 100.
- Media carga: se consigue un tiempo de respuesta moderado (0,1 segundos) con n = 1.000.000.
- Alta carga: se consigue un tiempo de respuesta elevado (1 segundo) con n= 10.000.000.

Con el objetivo de comparar la diferencia en cuanto a tiempo de respuesta entre un servidor configurado en un contenedor Docker o en una máquina virtual se han propuesto dos configuraciones. Todas ellas se ejecutan en una máquina física de gama media, dotada con un procesador de 32 núcleos de procesamiento y 512 GB de memoria.

Con la primera configuración se van a realizar las pruebas solo con máquinas virtuales, con 1 núcleo del procesador asignado a cada una y 4 GB de memoria RAM. Se van a ejecutar las pruebas de carga anteriormente mencionadas con una configuración de 1, 2, 4 y 8 máquinas virtuales.

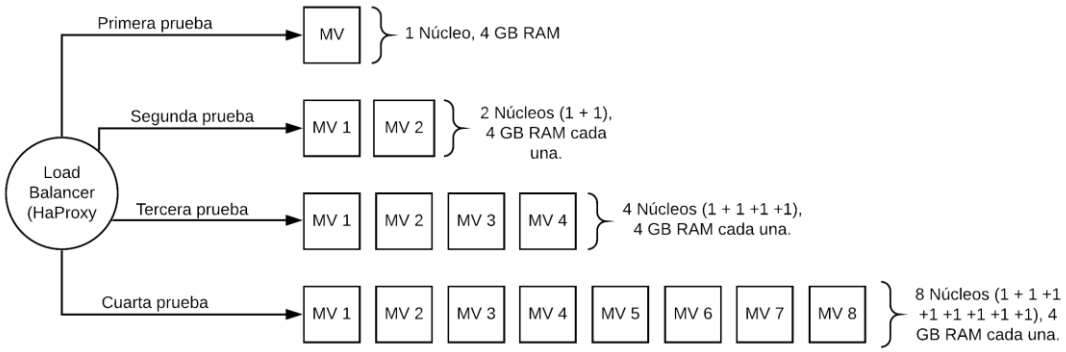

Ilustración 5. Esquema configuración 1 - Solo máquinas virtuales

La segunda configuración introduce el uso de Docker y dispone el clúster con 1 MV - 8 Docker, 2 MV – 4 Docker y 4 MV – 2 Docker. Se van a asignar 4 GB de memoria RAM a cada máquina virtual y un número de núcleos de CPU igual al número de contenedores que ejecute. Es importante destacar que en esta configuración siempre va a haber un total de ocho servidores reales, por lo que todos los datos obtenidos se deben interpretar atendiendo a que siempre van a haber ocho núcleos atendiendo peticiones.

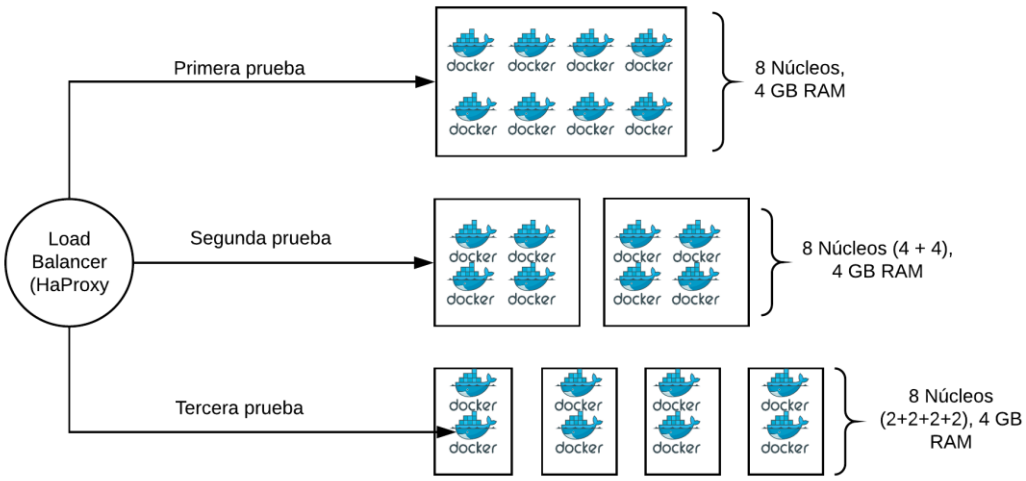

Ilustración 6. Esquema configuración 2 - Máquinas virtuales + Docker

5.2.2 Pruebas con Apache Benchmark

En cada configuración mencionada en el apartado anterior se va a utilizar Apache Benchmark para lanzar un conjunto de peticiones a los servidores de acuerdo con diversos grados de concurrencia y obtener dos valores estadísticos: *request per second* y *time per request*. Con ellos se van a construir las gráficas que permitan ilustrar el punto de saturación del clúster en función del número de servidores disponibles.

Así, la orden utilizada es:

```
ab -n <numero_peticiones> -c <grado_concurrencia> <URL>
```

Donde <numero_peticiones> es el número de peticiones que se van a lanzar y <grado_concurrencia> el número de peticiones concurrentes. Por ejemplo, si n = 100 y c = 2 se lanzan 100 peticiones de dos en dos.

<URL> es la dirección que permite acceder al servidor.

Como se ha mencionado anteriormente, esta configuración va a evaluar la capacidad del clúster para ejecutar un número elevado de peticiones de baja, media y alta carga con distintos grados de concurrencia.

Así, se han planteado dos hipótesis que pretenden ser validadas con los resultados obtenidos.

- Primero, se puede suponer que a medida que aumenta el número de máquinas virtuales (y por tanto de servidores) el servicio será capaz de atender un mayor número de peticiones concurrentes. No obstante, es importante prestar atención al punto de saturación del clúster, indicativo de que todos los servidores han alcanzado el 100% de uso de CPU. Dicho límite está condicionado por el número de servidores (o núcleos de CPU) y el grado de concurrencia, tal y como se va a ver a continuación. En definitiva, aumentar el número de servidores incrementará el número de peticiones por segundo hasta alcanzar el límite en que el grado de concurrencia sea igual al número de servidores (o núcleos).

- Segundo, cuanto mayor sea el número de servidores menor será el tiempo de respuesta. Además, mientras el número de servidores sea menor al grado de concurrencia de las peticiones el clúster no alcanzará el 100% de uso de CPU y el tiempo de respuesta se mantendrá constante. Una vez alcanzado el punto de saturación, el tiempo de respuesta comenzará a incrementarse.

5.2.2.1 Pruebas de baja carga

Las pruebas de baja carga suponen un conjunto de peticiones que el servidor es capaz de procesar en unos pocos milisegundos. Por este motivo, el tráfico en la red es mucho más elevado ya que las peticiones se contestan al instante.

Se ha enviado un conjunto de cien peticiones (-n 100) con diferentes grados de concurrencia para cada configuración de máquinas virtuales, o máquinas virtuales con Docker. Los resultados numéricos pueden consultarse en el Anexo II. Dichos datos estadísticos se han representado en dos gráficas para facilitar la comparación de peticiones por segundo y tiempo de respuesta entre MVs y contenedores.

Antes de interpretar las gráficas es importante matizar las características que se quieren observar: en primer lugar, la variación del número de peticiones por segundo y el tiempo de respuesta en función del número de servidores reales. Segundo, la diferencia entre la utilización de solo máquinas virtuales y la adición de Docker.

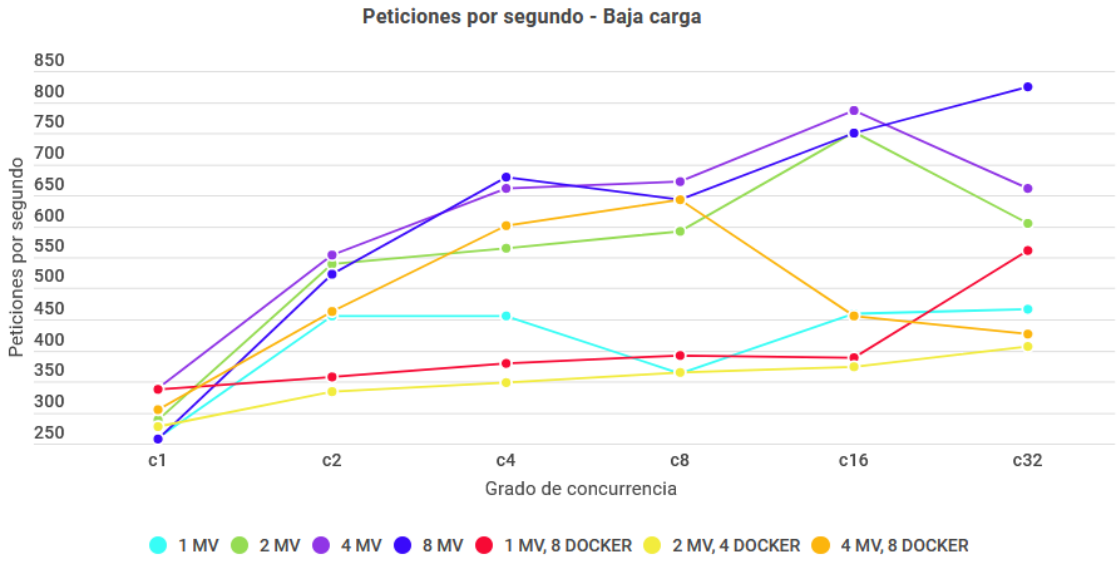

Ilustración 7. Baja carga - Peticiones por segundo

La gráfica anterior representa las peticiones por segundo que es capaz de atender el clúster en función del número de peticiones concurrentes a procesar en un instante de tiempo. No se observa una tendencia clara de mejora al aumentar el número de servidores reales. Tal y como se probará más adelante, es debido a que con peticiones que demandan poco cómputo el

equilibrador de carga (HaProxy) se convierte en un cuello de botella ya que el tráfico en la red se dispara.

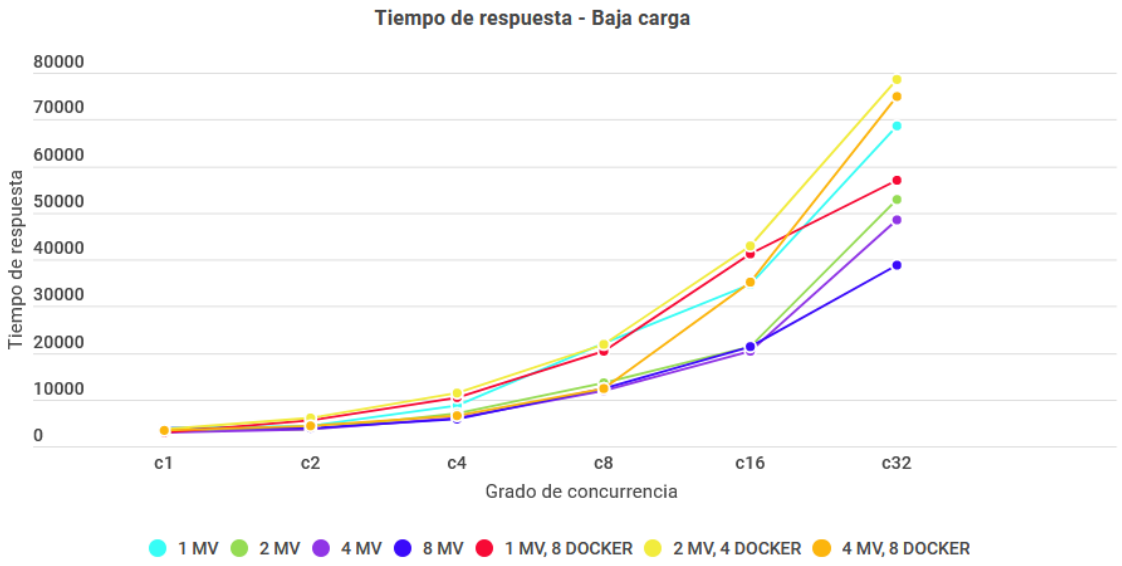

Ilustración 8. Baja carga - Tiempo de respuesta

En cuanto al tiempo de respuesta, se puede observar una tendencia a reducirse en tanto que aumenta el número de servidores reales. No obstante, algunas líneas se cruzan con otras, un indicativo de que hay otros aspectos que están afectando al tiempo de respuesta. Por ejemplo, la línea de color rojo (1 MV con 8 Docker) presenta en c16 un tiempo de respuesta superior a la línea de color cian (1 MV). Este resultado no coincide con la hipótesis planteada ya que, en teoría, ocho núcleos deben ofrecer un tiempo de respuesta menor a un solo núcleo, sin embargo, debido al poco cómputo que requiere esta prueba, en este caso no es así.

5.2.2.2 Pruebas de media carga

Aunque en el análisis con peticiones de bajo coste se pueden intuir ciertas tendencias, los resultados no son concluyentes porque presentan ciertos valores que no concuerdan con las hipótesis.

En las pruebas con peticiones de media carga los resultados empiezan a mostrar una tendencia más concorde al planteamiento inicial.

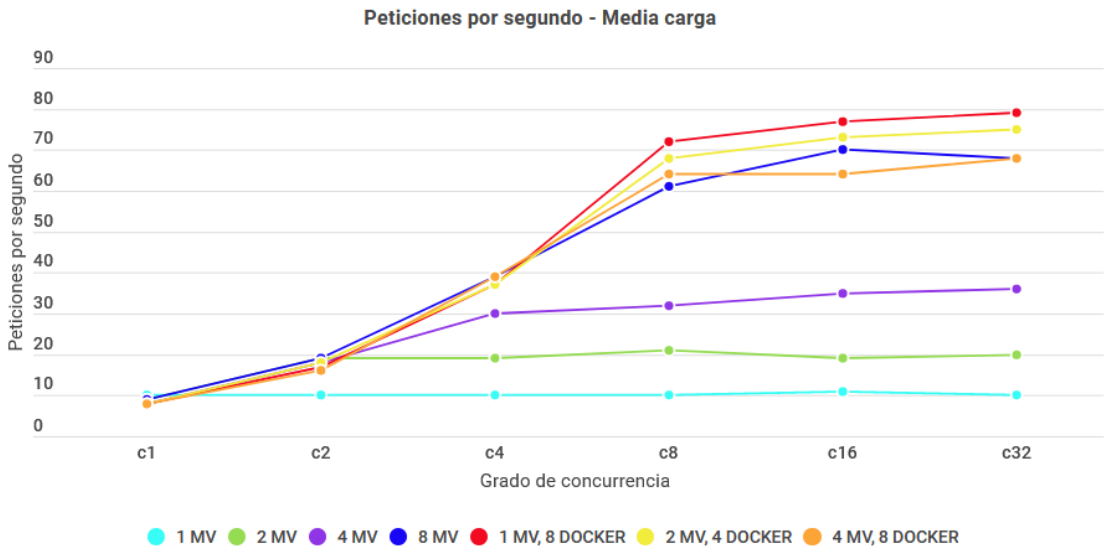

Ilustración 9. Media carga - Peticiones por segundo

En la gráfica anterior se puede observar como el número de peticiones por segundo crece a medida que aumenta el grado de concurrencia hasta alcanzar el punto de saturación. Como se ha comentado anteriormente, el punto de saturación se alcanza cuando el grado de concurrencia es igual al número de servidores reales (o núcleos de CPU).

Por otra parte, no se observa diferencia entre las peticiones por segundo obtenidas en la configuración con 8 máquinas virtuales y las que combinan MVs con Docker. Aparentemente, se puede intuir que el uso de contenedores no aporta ninguna mejora en cuanto al número de peticiones por segundo. No obstante, los resultados no son lo suficientemente rigurosos como para ratificar tal afirmación.

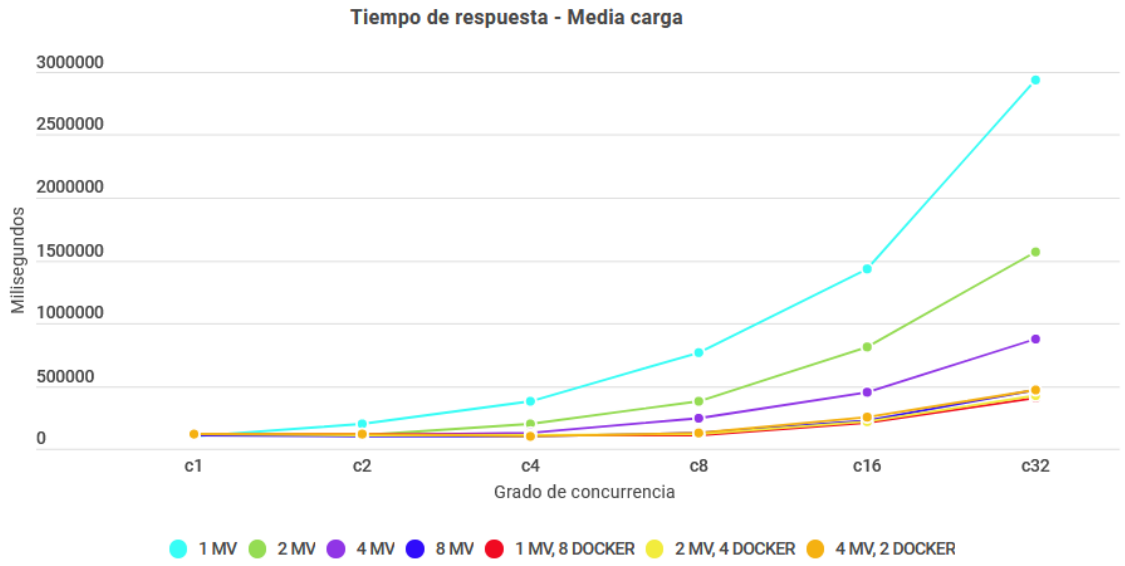

Ilustración 10. Media carga - Tiempo de respuesta

En cuanto al tiempo de respuesta, se observa una tendencia a reducirse en tanto que el número de servidores aumenta. También se observa como una vez alcanzado el punto de saturación, el servidor es incapaz de atender más peticiones concurrentes por lo que el tiempo de respuesta se incrementa.

Por otra parte, las líneas relativas a las configuraciones con ocho servidores se solapan, indicativo de que el tiempo de respuesta es independiente al uso de contenedores o no.

5.2.2.3 Pruebas de alta carga

Los datos obtenidos a partir del conjunto de pruebas de alta carga proporcionan la evidencia definitiva que permite confirmar las hipótesis iniciales. Al ser peticiones de alto coste (aproximadamente 1 segundo por petición) los servidores se convierten en el cuello de botella porque cada núcleo solo es capaz de procesar una petición por segundo. Así, el tráfico en la red se reduce por lo que se elimina cualquier retardo provocado por una red saturada.

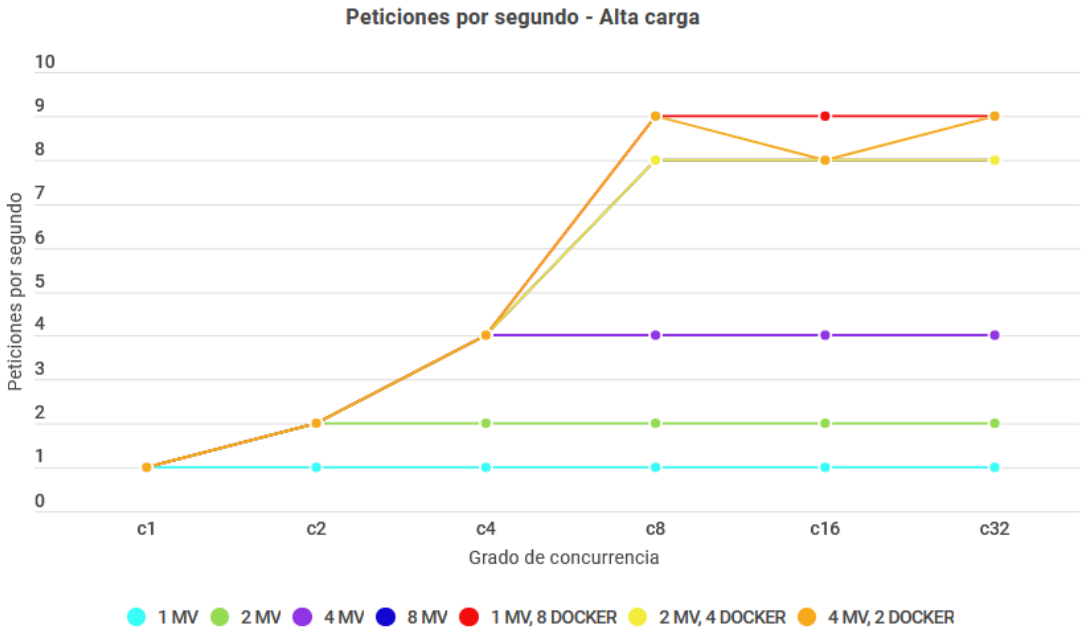

Ilustración 11. Alta carga - Peticiones por segundo

En la gráfica anterior se observa claramente la variabilidad de las peticiones por segundo en función del grado de concurrencia y el número de servidores. Una vez el número de servidores es igual al grado de concurrencia, las peticiones por segundo se mantienen constantes.

Por otra parte, se puede afirmar que el uso de solo máquinas virtuales o contenedores no influye en el número de peticiones que es capaz de procesar el clúster ya que las líneas se solapan.

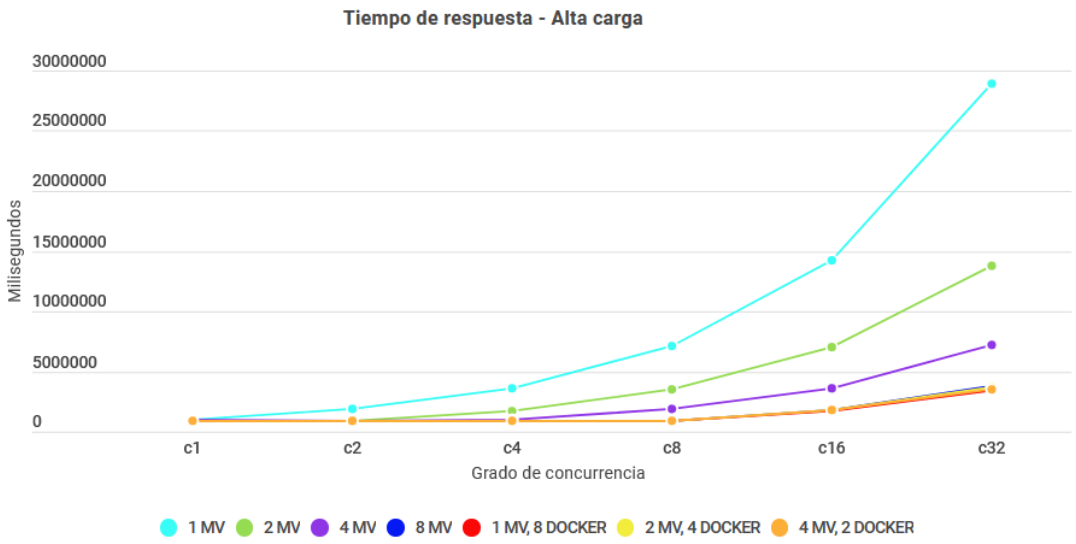

Ilustración 12. Alta carga - Tiempo de respuesta

En el caso del tiempo de respuesta, los resultados son muy similares a los obtenidos con el conjunto de peticiones de media carga: aumentar el número de servidores permite reducir el tiempo de respuesta.

En definitiva, los resultados obtenidos evidencian la veracidad de las hipótesis planteadas inicialmente. En primer lugar, aumentar el número de servidores (ya sean máquinas virtuales o máquinas virtuales con varios contenedores) incrementa el número de peticiones por segundo. Esto implica que el clúster es capaz de atender más peticiones (y por tanto, a más clientes) cuanto mayor sea el número de servidores.

En segundo lugar, si se añaden servidores el clúster puede ofrecer tiempos de respuesta menores. Además, éste se mantiene constante hasta que se alcanza el punto de saturación, por lo que un número elevado de servidores permite garantizar que el servicio puede atender hasta cierto número de peticiones ofreciendo las respuestas en un tiempo establecido.

Por último, cabe destacar que no se ha observado ninguna diferencia entre utilizar un conjunto de máquinas virtuales como servidores (hasta ocho) o reducir el número de MVs manteniendo constante el número de servidores utilizando contenedores. Este comportamiento se debe a que las prestaciones en cuanto a tiempo de respuesta y peticiones por segundo no dependen del uso de una tecnología u otra. Más aún, a efectos de la red Docker es visto como una máquina virtual más, por lo que no ofrece ninguna ventaja a la hora de procesar peticiones más rápidamente.

5.2.3 Análisis del consumo de recursos

Tras observar las prestaciones obtenidas en cuanto a tiempo de respuesta de una petición y número de peticiones atendidas por segundo, se va a analizar el consumo de recursos que ha realizado el clúster.

Para tal propósito se va a utilizar la orden de Linux 'top'[16], capaz de proporcionar información en tiempo real acerca del uso de la memoria y CPU, entre otras cosas. Sus parámetros *-d* y *-n* permiten ajustar el intervalo de segundos entre cada muestra y el número total de muestras deseadas, respectivamente. Así, con la siguiente orden se obtienen un total de veinte muestras, una cada segundo, con objetivo de representar la evolución del uso de CPU a lo largo del tiempo. Para calcular el consumo de memoria RAM se ha obtenido una muestra, ya que solo se pretende conocer el consumo base.

```
top -d 1 -b -n 20 > consumoRecursos.txt
```

5.2.3.1 Consumo de memoria

En este apartado se quiere analizar la utilización de memoria RAM requerida por ocho máquinas virtuales distintas y por una máquina virtual con ocho contenedores Docker en su interior, equivalente a ocho servidores distintos. El objetivo es determinar la ventaja que ofrece la utilización de contenedores para replicar servidores frente a utilizar solamente máquinas virtuales.

En primer lugar, se han ejecutado ocho máquinas virtuales. A continuación, se muestra un extracto de la salida de la orden *top* con las líneas correspondientes al proceso de cada MV. La primera columna (PID), señalada en negrita, corresponde con el PID del proceso que ejecuta la máquina virtual. La sexta columna (RES) corresponde con la cantidad de memoria RAM utilizada por el proceso.

```
PID   USER                          RES
18805 pabfloso   20   0 5017256 581692 514602 S   3,0   0,1
5:29.24 VBoxHeadless
18842 pabfloso   20   0 2902784 574624 507560 S   3,3   0,1
1:41.52 VBoxHeadless
18873 pabfloso   20   0 2904832 585088 515752 S   3,0   0,1
1:39.09 VBoxHeadless
18966 pabfloso   20   0 2890496 582076 515752 S   2,0   0,1
1:39.75 VBoxHeadless
19024 pabfloso   20   0 2915072 579460 511640 S   2,0   0,1
1:40.94 VBoxHeadless
18995 pabfloso   20   0 2919168 591080 519832 S   1,7   0,1
1:41.00 VBoxHeadless
18905 pabfloso   20   0 2894592 577848 509608 S   1,3   0,1
1:40.20 VBoxHeadless
```

[16] Top – Linux man page. Disponible en: https://linux.die.net/man/1/top

```
18937 pabfloso   20    0 2915072 587080 521896 S    1,3   0,1
1:37.71 VBoxHeadless
```

Figura 22. Salida "top" - Solo máquinas virtuales

Después, en una de las máquinas virtuales se ha ejecutado Docker y se han iniciado ocho instancias del contenedor que proporciona el servicio web. Así, una sola máquina virtual es capaz de mantener ocho servidores. Luego, se ha obtenido de nuevo el consumo de memoria de dicha MV. Como es de esperar, el consumo de memoria se ha elevado hasta alcanzar los 756180 KB.

```
18805 pabfloso   20    0 5017256 756180 701092 S    8,3   0,5
6:08.29 VBoxHeadless
```

Figura 23. Salida "top" - Una MV con ocho contenedores

También es posible conocer el consumo de memoria por parte del *Daemon* de Docker.

```
1418 root        20    0 2558344  82064  26984 S    0,3   0,0
13:56.79 dockerd
```

Figura 24. Salida "top" - Docker Daemon

Como se ha visto, cada máquina virtual independiente sin ejecutar Docker consume en promedio 582368.5 KB de memoria RAM. Con esto, desplegar ocho servidores utilizando solo máquinas virtuales requiere 4658948 KB de memoria, es decir, 4.44 GB.

No obstante, utilizando contenedores es posible conseguir desplegar ocho servidores con 756180 KB, esto es, 738.46 MB.

Cabe destacar que para mantener la propiedad de alta disponibilidad es necesario suprimir cualquier punto único de fallo. Por ello, es requisito desplegar como mínimo dos máquinas virtuales. En cualquier caso, queda claro que la combinación de máquinas virtuales con Docker para alcanzar un mayor número de instancias del servicio permite ahorrar una considerable cantidad de memoria RAM.

Como añadido adicional, es interesante conocer la forma en que Docker gestiona la utilización de memoria dinámica. Su máxima es reutilizar los recursos del sistema tanto como sea posible. Cuando se ejecutan varios contenedores basados en la misma imagen, internamente Docker solo despliega una instancia de la imagen, y todos los contenedores se referencian a dicha instancia. Después, cualquier cambio sucedido en un contenedor independiente se añade como “capa” superior sobre la instancia base, de modo que todos los contenedores referencian la misma imagen inicial, y cada uno maneja los cambios que va realizando.

5.2.3.2 Uso de CPU

Baja carga

En el apartado 5.2.2.1 las estadísticas muestran que en pruebas de baja carga las prestaciones del clúster no presentan notables variaciones en cuanto al tiempo de respuesta ofrecido. La hipótesis planteada para tal efecto es que incrementar el tráfico en la red convierte al equilibrador de carga (HaProxy) en un cuello de botella, ya que se satura al procesar todas las peticiones y respuestas que le llegan.

Las gráficas que se muestran a continuación confirman dicho planteamiento.

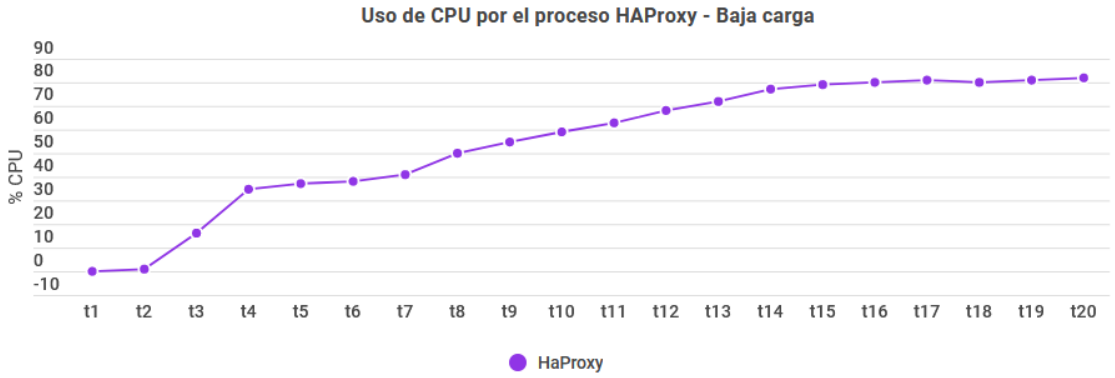

Ilustración 13. Baja carga - Uso de CPU por HAProxy

La gráfica anterior representa el uso de CPU por el proceso HAProxy ante un conjunto de peticiones con baja demanda de cómputo. Como se puede observar, a medida que se avanza en el tiempo el uso de CPU aumenta hasta alcanzar su límite en t20. Veinte segundos después de ejecutar ApacheBenchmark, HAProxy ha quedado saturado por el elevado número de peticiones y respuestas que debe procesar.

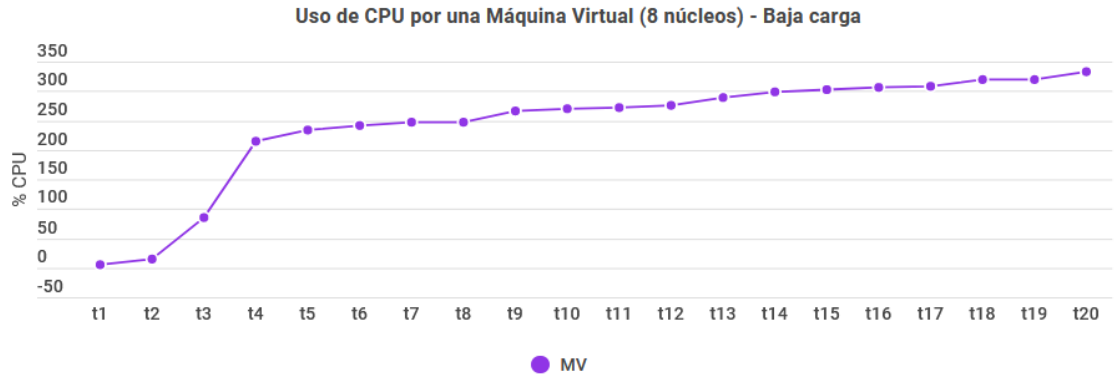

Ilustración 14. Baja carga - Uso de CPU por una MV

Esta última gráfica representa el uso de CPU por una máquina virtual con ocho núcleos asignados. Aquí, a medida que se avanza en el tiempo, el porcentaje de uso de CPU aumenta hasta alcanzar el pico en aproximadamente un 350%. Es necesario tener en cuenta que la máquina cuenta con ocho núcleos, por lo que el punto de saturación se alcanzaría en los 800%, lo que implica que el servidor no se está saturando.

En base a las estadísticas obtenidas, se puede afirmar que un conjunto elevado de peticiones de bajo coste provoca un incremento del tráfico en la red, lo cual satura el equilibrador de carga impidiendo que redirija todas las peticiones a tiempo, convirtiéndose en el cuello de botella del clúster. Los servidores, por su parte, apenas alcanzan un 43,75% de su capacidad computacional total (350%/800%).

Alta carga

En el apartado 5.2.2.3 las estadísticas muestran que las peticiones con alta demanda de cómputo provocan un comportamiento totalmente opuesto al obtenido con peticiones de bajo coste. Cuando una petición requiere un tiempo relativamente elevado para procesarse (por ejemplo, 1 segundo) los servidores se convierten en el cuello de botella del clúster.

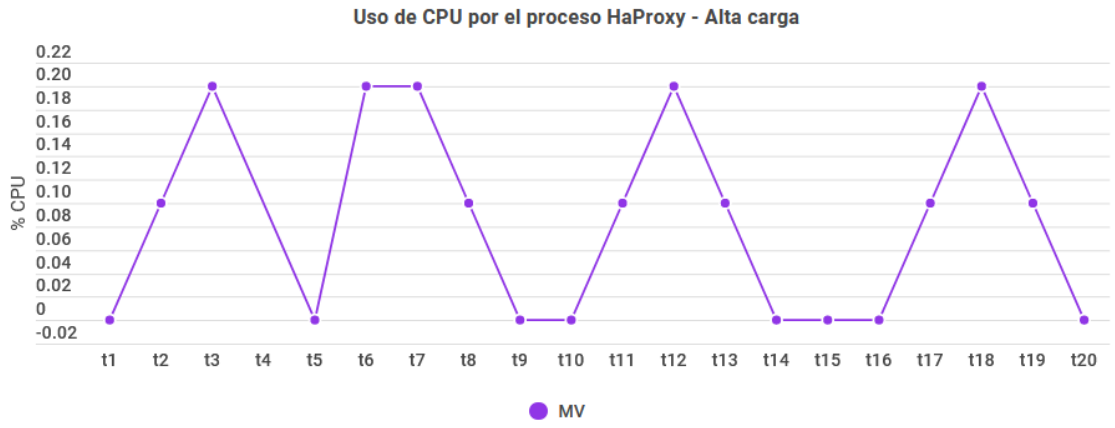

Ilustración 15. Alta carga - Uso de CPU por HAProxy

En la gráfica anterior se puede observar el uso de CPU por el proceso HaProxy ante un conjunto de peticiones de alto coste. Como se puede apreciar, su utilización oscila entre el 0 y 0,2%. Este comportamiento tan diferente al obtenido en las pruebas de baja carga se produce porque los servidores tardan mucho tiempo en procesar una petición. Así, a medida que se van procesando las respuestas, HaProxy las va redirigiendo al cliente, pero el conjunto de paquetes es muy reducido.

La siguiente gráfica muestra como el servidor ha quedado saturado al alcanzar el 800% de uso de su CPU, es decir, la utilización de sus ocho núcleos es del 100%.

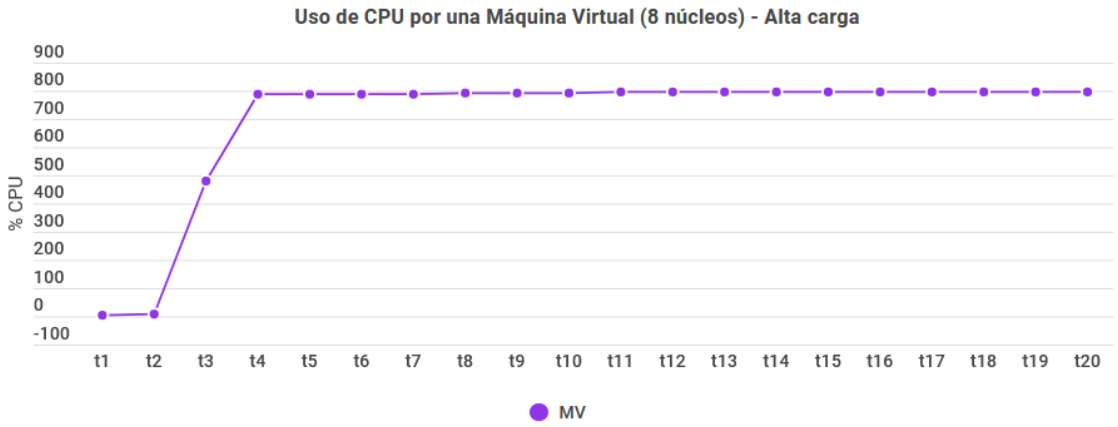

Ilustración 16. Alta carga - Uso de CPU por una MV

Por último, se ha querido analizar la carga de CPU con ocho máquinas virtuales distintas y un núcleo asignado a cada una. En la gráfica siguiente se puede observar una curva similar a la obtenida utilizando una sola máquina con ocho núcleos. En ambos casos la capacidad computacional es la misma, ya que se tiene el mismo número de núcleos de CPU.

La única ventaja que proporciona este segundo planteamiento es el aumento de la disponibilidad del servicio. Es decir, al configurar la aplicación en más de un servidor se elimina el punto único de fallo. Más aún, cuanto mayor sea el número de servidores menor es la probabilidad de que todos ellos fallen a la vez, lo que se garantiza que el servicio sea altamente disponible.

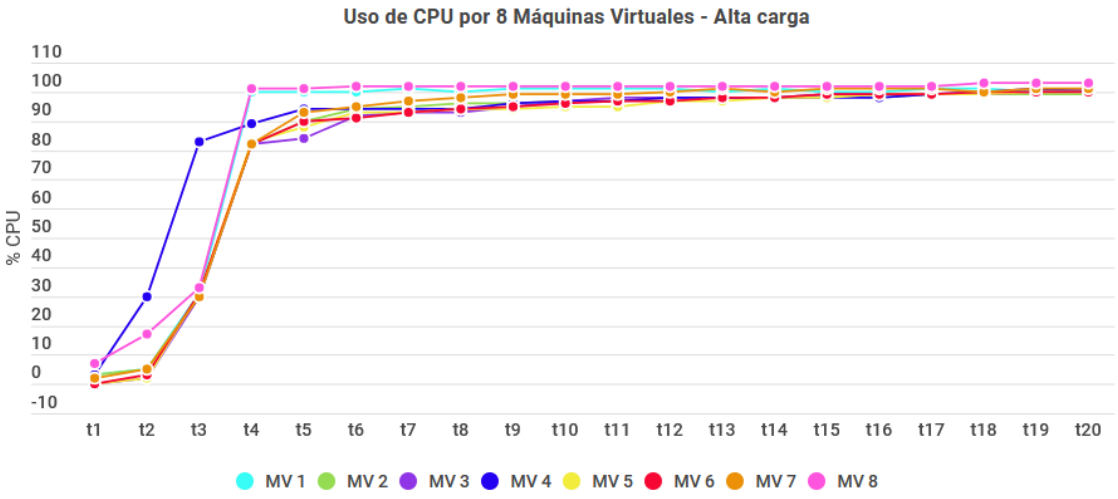

Ilustración 17. Alta carga - Uso de CPU por ocho MVs

5.2.4 Tiempo de despliegue

Se quiere hacer una comparativa entre las ventajas que ha ofrecido configurar la infraestructura del clúster utilizando contenedores para la replicación de los servidores de aplicación, en contraposición al uso de máquinas virtuales para tal efecto. En los apartados anteriores se han analizado las diferencias en cuanto a uso de CPU y consumo de memoria RAM.

Además de la disminución del consumo de recursos que incorpora Docker, también promete proporcionar despliegues de infraestructura mucho más agiles. Para comprobar tal característica, se va a obtener el tiempo de despliegue de una máquina virtual y un contenedor Docker para su comparación.

Para conocer los tiempos de despliegue de un contenedor hay que ejecutar el siguiente comando[17]:

```
time Docker run –rm <nombre_imagen> ping 8.8.8.8
```

Al hacerlo, se creará un contenedor a partir de la imagen <nombre_imagen> y se ejecutará la orden ping. Al terminar, se apaga y devuelve el tiempo transcurrido.

```
real   0m1,612s
user   0m0,072s
sys    0m0,080s
```

Figura 25. Salida "time" - Tiempo de despliegue de un contenedor

Por otra parte, conocer el tiempo de despliegue de una máquina virtual es más complicado, por lo que se ha de recurrir a un script. En particular, se ha utilizado el que se muestra en la figura 26, cuyo funcionamiento es el siguiente: primero, obtiene el instante de tiempo en que se ejecuta. A continuación, inicia la máquina virtual y comienza a lanzarle *pings*. Cuando un paquete *ping* recibe contestación, obtiene el instante de tiempo en que se ha recibido el paquete. Por último, se calcula la diferencia entre los dos tiempos obtenidos, la cual supone el tiempo desde que la máquina virtual se ha iniciado hasta que su interfaz de red ha estado activa para recibir y responder paquetes. Dicho tiempo es equivalente al tiempo de despliegue de la máquina virtual.

```
SERVER=$1
TIME_1=`date +%s`
#Aqui ponemos la orden que lanza el servicio
ssh -p 3322 pabfloso@mcom.disca.upv.es vboxmanage startvm
“Worker1”
OUT=1
while [ $OUT -ne 0 ];
do
ping -c1 $SERVER
```

[17] Time – Linux man page. Disponible en: https://man7.org/linux/man-pages/man1/time.1.html

```
OUT=$?
done
TIME_2=`date +%s`
echo $TIME_1
echo $TIME_2
ELAPSED=`expr $TIME_2 - $TIME_1`
echo "Tiempo transcurrido(s):" $ELAPSED
```

Figura 26. Script para calcular el tiempo de despliegue de una MV

La salida del script anterior es:

```
1597423351
1597423391
Tiempo transcurrido(s): 40
```

Figura 27. Salida Script. Tiempo de despliegue de una MV.

En comparación, el tiempo en levantar una máquina virtual es de 40 segundos, mientras que el tiempo en levantar un contenedor es de 1,612 segundos. En cuanto a tiempo de despliegue, los resultados se inclinan muy favorablemente en favor de Docker, tal y como promete la utilización de esta tecnología.

5.2.5. Apache jMeter

En este apartado se ha querido utilizar una herramienta alternativa a Apache Benchmark para realizar pruebas de carga sobre la aplicación de consulta de prescripciones médicas. En concreto, se va a analizar el tiempo de respuesta del servicio "farhosIn".

Apache jMeter[18] es una aplicación con interfaz gráfica de código abierto diseñada para ejecutar pruebas de carga sobre una variedad de servicios y evaluar su rendimiento. Principalmente se utiliza para el testeo de aplicaciones web, conexiones a bases de datos JDBC, conexiones TCP, servicios web…

Es importante destacar que las pruebas realizadas en este apartado tienen el objetivo de presentar Apache jMeter como herramienta alternativa a Apache Benchmark para realizar un análisis de prestaciones. Se pretende mostrar el funcionamiento básico del programa y las posibilidades más comunes a la hora de obtener resultados. No obstante, la evaluación cuantitativa realizada en los apartados anteriores es suficiente para comprender las capacidades y limitaciones del clúster, por lo que no se va a hacer hincapié en la interpretación de los resultados obtenidos con jMeter.

5.2.5.1 Prueba con Apache jMeter

En primer lugar, se ha descargado la aplicación sobre una máquina virtual con el rol de cliente desde donde se van a ejecutar el conjunto de peticiones de prueba. Tras iniciar jMeter se ha

[18] Apache jMeter. Disponible en: https://jmeter.apache.org/

añadido al "Plan de Pruebas" un "Grupo de Hilos" en el cual se han a declarado el conjunto de peticiones que se van a realizar. En concreto, se quiere obtener la prescripción médica de diez pacientes por lo que se han añadido diez muestreadores de tipo "Peticion HTTP" al Grupo de Hilos con su correspondiente ruta y especificación de parámetros. Además, se quiere repetir cien veces cada petición, por lo que la propiedad "Numero de Hilos" se ha definido como 100.

Después, se han especificado los tipos de informe (receptores) que el programa debe generar con los resultados de la ejecución. Se ha optado por generar los siguientes informes:

- Árbol de Resultados: el receptor "Árbol de Resultados" muestra el resultado obtenido para cada petición. Permite conocer el número de peticiones que se han llevado a cabo correctamente y las que han sufrido algún error.
- Informe Agregado: el receptor "Informe Agregado" representa en una tabla datos estadísticos como la media, mediana, percentil 95%, mínimo y máximo del tiempo de respuesta, así como porcentaje de error, kilobytes recibidos y enviados y otros valores a elegir por el usuario.
- Gráfico: el receptor "Gráfico" representa en una gráfica personalizable todos los datos estadísticos recogidos en el Informe Agregado.
- Gráfico del Tiempo de Respuesta: el receptor "Gráfico del Tiempo de Respuesta" representa en una gráfica el tiempo de respuesta de cada conjunto de peticiones.

Cabe recordar que las peticiones realizadas a la aplicación de consulta prescripciones inducen una carga computacional muy baja, por lo que el conjunto de pruebas que se va a mostrar a continuación es equivalente a las pruebas de bajo coste realizadas en los apartados anteriores al servicio de cálculo de integrales. No obstante, en este caso se requiere consultar una base de datos externa accesible por VPN, por lo que pueden experimentarse retrasos debido a esta conexión.

A continuación, se muestran los resultados obtenidos para la prueba realizada:

Etiqueta	# Muestras	Media	Mediana	90% Line	95% Line	99% Line	Mín	Máx	% Error	Rendimiento	Kb/sec	Sent K...
Paciente_1	100	1276	1196	1845	1918	2650	683	2675	0,00%	16,3/sec	1,92	7,00
Paciente_2	100	1588	1783	2024	2116	2615	734	2706	0,00%	13,3/sec	2,30	5,71
Paciente_3	100	1835	1921	2094	2103	2130	1017	2140	0,00%	11,2/sec	1,74	4,82
Paciente_4	100	1993	2026	2104	2119	2141	1625	2162	0,00%	10,1/sec	1,19	4,35
Paciente_5	100	2001	2013	2099	2108	2128	1792	2144	0,00%	9,8/sec	0,79	4,20
Paciente_6	100	2005	1985	2084	2106	2142	1877	2719	0,00%	9,0/sec	0,72	3,85
Paciente_7	100	1971	1951	2049	2069	2590	1738	2614	0,00%	9,2/sec	2,19	3,96
Paciente_8	100	1872	1905	1972	1980	2015	1219	2041	0,00%	10,1/sec	1,37	4,32
Paciente_9	100	1656	1796	1953	1964	1975	845	1981	0,00%	11,4/sec	2,29	4,91
Paciente_10	100	1374	1325	1944	1970	1977	708	1993	0,00%	13,3/sec	1,44	5,72
Total	1000	1757	1916	2075	2099	2566	683	2719	0,00%	46,4/sec	6,53	19,91

Ilustración 18. jMeter - Informe Agregado

La figura anterior corresponde al receptor "Informe Agregado". Cada fila de la columna "Etiqueta" representa el conjunto de cien peticiones (ver # Muestras) a un paciente de la base de datos. Como se puede observar, el resto de las columnas muestran los datos estadísticos obtenidos durante la prueba.

La figura siguiente es una representación gráfica de los datos obtenidos en el Informe Agregado. El usuario que realiza las pruebas puede definir en la leyenda los parámetros que quiere representar. Para el ejemplo se ha elegido mostrar la media, mediana y valores mínimo y máximo del tiempo de respuesta.

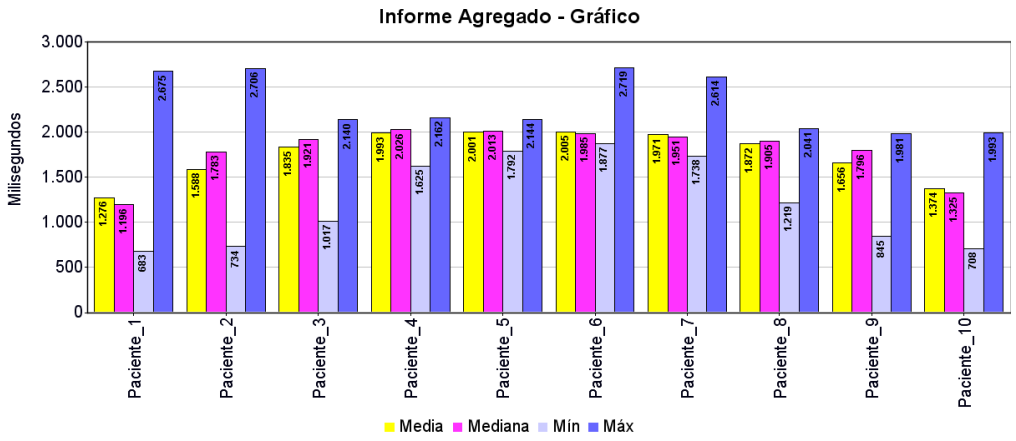

Ilustración 19. jMeter - Informe Agregado (Gráfico)

Por último, la gráfica siguiente representa la evolución del tiempo de respuesta de cada conjunto de peticiones a lo largo del tiempo que ha durado la prueba. Aquí, el usuario tiene la posibilidad de adaptar los rangos y los intervalos de los valores tanto del eje de abscisas como de ordenadas para ajustar las líneas de la gráfica como crea conveniente para su interpretación.

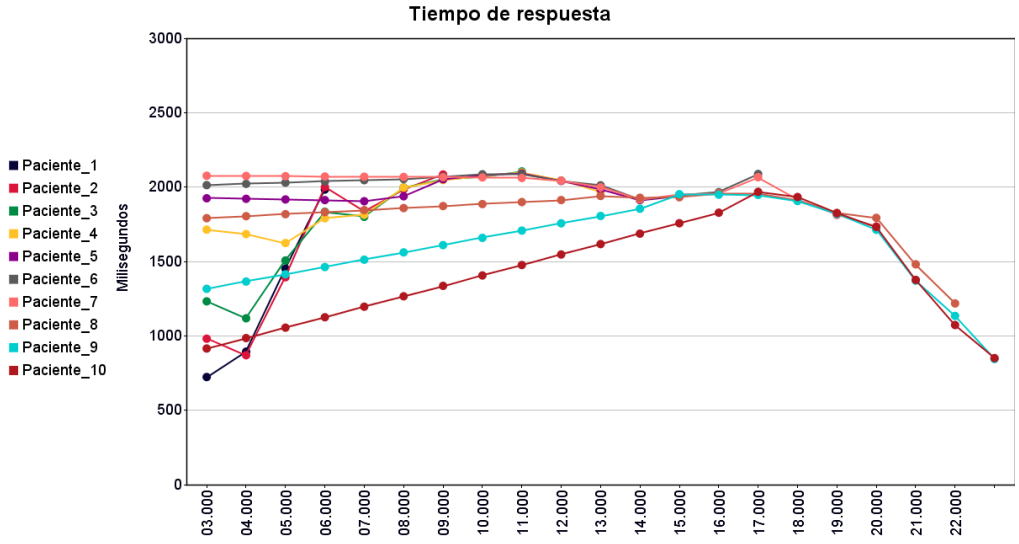

Ilustración 20. jMeter - Tiempo de respuesta

6. Conclusiones

La finalidad de este proyecto es la configuración de un clúster de alta disponibilidad con equilibrado de carga combinado con el uso de máquinas virtuales y contenedores Docker y desplegar sus servidores valiéndonos de las ventajas que ambas ofrecen.

Se ha realizado un análisis cualitativo de la funcionalidad con la aplicación Farhos Prescripción, y un análisis cuantitativo con un servicio que demanda una alta potencia de cómputo en los nodos servidores.

En base a los resultados de ambos análisis se puede afirmar que los objetivos establecidos al inicio del proyecto se han alcanzado satisfactoriamente.

En particular, las pruebas cualitativas han validado el correcto funcionamiento del clúster, es decir, han garantizado el funcionamiento ininterrumpido del servicio que planteaba el primer objetivo. También las pruebas cuantitativas demuestran que aumentar el número de servidores incrementa la capacidad computacional o, dicho de otro modo, permite que el servidor crezca en prestaciones, tal y como se especifica en el primer objetivo del proyecto.

Por último, queda demostrado que Docker es una tecnología totalmente funcional para este tipo de infraestructuras. Más aún, las ventajas que ofrece en cuanto al bajo consumo de memoria RAM y velocidad de despliegue lo sitúan como una alternativa muy potente a las máquinas virtuales tradicionales para desplegar el servicio ofrecido.

Conocimientos adquiridos

El principal conocimiento conseguido en este proyecto ha sido la utilización de la herramienta Docker con la finalidad de encapsular la aplicación en un contenedor. Para ello ha sido necesario estudiar y comprender el formato del fichero de instrucciones “Dockerfile” y el fichero de despliegue “Docker-Compose”. También conocer la estructura de un contenedor, el modo en que el *Daemon* de Docker gestiona los recursos de la máquina y adquirir conocimientos de instalación y configuración de las herramientas HaProxy y KeepAlived, pues ambas resultan imprescindibles para conformar el clúster.

Por otra parte, manejar Debian como sistema operativo ha puesto en práctica las capacidades de administración de los sistemas Linux. Gracias a ello se han ampliado los conocimientos en el uso de los comandos para configurar el entorno y las interfaces de red y para la utilización de otras herramientas sin interfaz gráfica.

Por último, se ha aprendido a utilizar Apache jMeter como herramienta alternativa para las pruebas de carga, cuyo manejo es interesante si se pretende disponer de una aplicación con interfaz gráfica.

Problemas encontrados

La principal dificultad en el proceso del proyecto ha sido encapsular Farhos Prescripción dentro de un contenedor Docker. Al no ser una aplicación pensada para desplegarse en un entorno de contenedores su acceso está programado para hacerse a través de una dirección IP, por lo que todas las llamadas RMI se basan en este formato. No obstante, el acceso a un contenedor dentro de una máquina virtual se realiza a través de <dirección_IP>:<puerto>, un requisito que obligó a replantear los ficheros de configuración de la aplicación.

También se ha tenido que modificar el acceso a la base de datos. En una instalación de Farhos real, la BBDD se encuentra en la misma máquina que ejecuta la aplicación. Sin embargo, por cuestiones de seguridad y protección de datos, el acceso a la base de datos se ha tenido que hacer de forma remota a través de una conexión VPN.

Por otra parte, la aplicación Farhos presenta una limitación a la hora de realizar una evaluación cuantitativa del sistema: sus peticiones tienen un muy bajo coste computacional. Como se ha visto anteriormente, es necesario realizar pruebas de bajo, medio y alto coste algorítmico, por lo que ha sido necesario desarrollar otra aplicación que proporcionase tal funcionalidad.

Por último, la infraestructura inicial en la que se desarrolló el proyecto no era adecuada para la evaluación cuantitativa debido a la limitación de recursos que ofrece un ordenador convencional. Por este motivo, se tuvo que migrar todas las máquinas virtuales a un computador de gama media dotado con un elevado número de núcleos y gran cantidad de memoria.

Aportación al proyecto

Finalmente, el desarrollo de este proyecto me ha brindado grandes aportaciones tanto personales como laborales. De una mano he adquirido grandes competencias en la gestión y desarrollo de proyectos, así como en la redacción de memorias documentadas y estructuradas formalmente. El entrenamiento de la capacidad resolutiva me ha proporcionado una experiencia muy enriquecedora para el enfrentamiento de futuros retos tecnológicos.

Por otra parte, el proyecto ha despertado el interés de Visual – Limes por la importancia de un sistema de alta disponibilidad en los entornos hospitalarios en los que la empresa implementa su software. Gracias a esto he tenido la oportunidad de desplegar un clúster de alta disponibilidad en un entorno de producción. No obstante, este proyecto comercial es una adaptación sin Docker ya que, tras un análisis de requisitos, la empresa ha determinado que no aporta las ventajas suficientes como para invertir en tiempo de aprendizaje, además del riesgo que implica utilizar una herramienta nueva con necesidades de configuración complejas para garantizar los estándares de seguridad que demanda la normativa.

7. Relación del trabajo con los estudios cursados

Tras finalizar el proyecto he realizado un ejercicio de introspección para determinar la relación entre el proyecto ejecutado y los estudios cursados en el grado, teniendo en cuenta que la rama de especialización cursada es “Tecnologías de la Información”. Pienso que es importante reflexionar sobre las competencias transversales que he puesto en práctica para conseguir objetivos satisfactorios.

En primer lugar, he utilizado Docker como una de las herramientas principales en el clúster y le he dotado de máxima importancia centrando gran parte del proyecto en el análisis de sus ventajas. Docker se ha utilizado en la asignatura del grado “Tecnologías de sistemas de la información en la red”, y es una tecnología con muchas posibilidades que ha ganado gran importancia en el sector en los últimos años.

He puesto en práctica los conocimientos obtenidos en las asignaturas “Diseño y configuración de redes de área local” y “Redes corporativas”, por la necesidad de plantear la configuración de red del clúster. En consecuencia, configurar los sistemas operativos basados en Linux también requiere de cierto dominio tecnológico que he adquirido en el grado.

Las herramientas de la línea de órdenes que permiten obtener la utilización de los recursos para analizar las capacidades del sistema y estudiadas en la asignatura “Diseño, configuración y evaluación de los sistemas informáticos” han sido de mucha utilidad para el análisis cuantitativo del clúster.

El paso por el grado universitario ha destacado y mejorado un conjunto de competencias transversales que han contribuido en mi crecimiento profesional y personal y han sido de vital importancia para el desarrollo de este proyecto.

Por ejemplo, la competencia de compresión e integración que permite relacionar e integrar conceptos en situaciones complejas me ha servido para relacionar las distintas herramientas para que se unifiquen por un objetivo común.

La habilidad de aplicación y pensamiento práctico que permite aplicar a la práctica las capacidades y recursos personales para alcanzar los objetivos dadas las circunstancias. Esta competencia estrechamente relacionada con la de análisis y resolución de problemas, ha sido necesaria saber aplicarla a todas las capacidades para superar los obstáculos que han surgido a lo largo del desarrollo.

La capacidad de pensamiento crítico ha sido necesaria para la reflexión sobre los pros y los contras de diferentes soluciones y para valorar sus implicaciones. Tanto en la identificación y análisis de posibles soluciones como en la valoración de las ventajas y desventajas de Docker,

he intentado aplicar un pensamiento crítico para determinar de manera objetiva las conclusiones.

Finalmente, otras competencias como aprendizaje permanente, planificación y gestión del tiempo e instrumental específica las he puesto en práctica para la elaboración el TFG. Todas ellas han sido imprescindibles para llevar a cabo un proyecto de estas características.

8. Bibliografía

[1] ROLAND MAS, RAPHAEL HERTZOG. *The Debian Administrator's Handbook.* Freexian SARL, 2012.

[2] PEDRO LOPEZ, ELVIRA BAYDAL. *Teaching high-performance service in a cluster computing course*. DISCA Department, Universitat Politècnica de Valencia, 2018.

[3] PEDRO LOPEZ, ELVIRA BAYDAL. *On a course on computer cluster configuration and administration.* DISCA Department, Universidad Politècnica de Valencia, 2017.

[4] ROBERT W. LUCKE. *Building Clustered Linux System.* Prentice Hall PTR, 2005.

[5] SANDER VAN VUGHT. *Pro Linux High Availability Clustering.* Apress, 2014.

[6] RFC 3768. *Virtual Router Redundancy Protocol (VRRP).* Abstract. Abril 2004. Disponible en: https://tools.ietf.org/html/rfc3768

[7] What is a container? A standarized unit of software.Disponible en: https://www.docker.com/resources/what-container

[8] EVAN MOUZAKITIS. *How to collect HAProxy metrics*. 2018. Disponible en: https://www.datadoghq.com/blog/how-to-collect-haproxy-metrics/

[9] WILLY TARREAU. *HAProxy Configuration Manual.* Octubre 2019. Disponible en: https://cbonte.github.io/haproxy-dconv/1.7/configuration.html

[10] *Keepalived Configuration Manual.* Mayo 2020 Disponible en: https://www.keepalived.org/manpage.html

Anexo I. Servicio cálculo de integrales definidas.

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

$n=$_GET["n"];

#$inf=$_GET["inf"];
#$sup=$_GET["sup"];
$inf=0;
$sup=1;

$inc=($sup-$inf)/$n;

$area=0.0;
$x=$inf;

while ($x<=$sup)
       {
#       Aqui se escribe la funcion:
#      $area=$area+FUNCION($x)
#      $area=$area+(sin($x)*sin($x)+cos($x)*cos($x));
#      $area=$area+sin($x);
       $area=$area+tanh($x);
       $x=$x+$inc;
       }

$result=$area*$inc;

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

echo "<br>Calculo de Integrales Definidas<br><br>";
#printf ("Funcion: sin(x)*sin(x)+cos(x)*cos(x)<br>");
#printf ("Funcion: sin(x)<br>");
printf ("Funcion: tanh(x)<br>");
printf ("Limites: inferior=%.5f superior=%.5f;
Intervalos=%d<br>",$inf,$sup,$n);
printf ("Resultado= %.5f<br>",$result);
printf ("Tiempo de ejecucion= %.5f segundos<br>",$exectime);
printf ("<br>El servidor es %s<br>", $_SERVER['SERVER_ADDR']);
?>
</body>
</html>
```

Anexo II. ApacheBenchmark - Baja carga

Solo máquinas virtuales

1 MV	Request per second	Time per request
c1	260,66	3,836
c2	456,22	4,384
c4	456,93	8,754
c8	363,71	21,995
c16	460,55	34,741
c32	466,58	68,585

2 MV	Request per second	Time per request
c1	288,81	3,463
c2	539,41	3,708
c4	564,30	7,088
c8	591,57	13,523
c16	751,46	21,292
c32	604,55	52,932

4 MV	Request per second	Time per request
c1	340,95	2,933
c2	554,17	3,609
c4	661,44	6,047
c8	672,92	11,888
c16	787,84	20,309
c32	661,48	48,376

8 MV	Request per second	Time per request
c1	258,81	3,863
c2	522,61	3,827
c4	680,91	5,875
c8	643,94	12,423
c16	750,1	21,330
c32	825,63	38,758

Máquinas Virtuales + Docker

1 MV, 8 DOCKER	Request per second	Time per request
c1	337,39	2,964
c2	358,24	5,583
c4	380,64	10,509
c8	391,28	20,446
c16	388,41	41,194
c32	561,90	56,950

2 MV, 4 DOCKER	Request per second	Time per request
c1	278,34	3,593
c2	333,37	5,999
c4	349,29	11,452
c8	365,72	21,874
c16	373,14	42,880
c32	407,81	78,468

2 MV, 4 DOCKER	Request per second	Time per request
c1	278,34	3,593
c2	333,37	5,999
c4	349,29	11,452
c8	365,72	21,874
c16	373,14	42,880
c32	407,81	78,468

Anexo III. Apache Benchmark – Media carga

Solo máquinas virtuales

1 MV	Request per second	Time per request
c1	10,22	97,812
c2	10,32	193,822
c4	10,64	376,066
c8	10,41	768,578
c16	11,14	1,436,428
c32	10,91	2,932,953

2 MV	Request per second	Time per request
c1	9,14	109,433
c2	19,24	103,930
c4	19,81	201,914
c8	21,33	375,032
c16	19,77	809,433
c32	20,38	1,569,967

4 MV	Request per second	Time per request
c1	8,95	111,786
c2	18,76	106,607
c4	30,90	129,456
c8	32,86	243,452
c16	35,33	452,836
c32	36,5	876,762

8 MV	Request per second	Time per request
c1	9,15	109,319
c2	19,46	102,777
c4	39,98	100,062
c8	61,88	129,287
c16	70,41	227,244
c32	68,11	469,843

Máquinas Virtuales + Docker

1 MV, 8 DOCKER	Request per second	Time per request
c1	8,69	115,122
c2	17,66	113,228
c4	37,37	107,040
c8	72,72	110,008
c16	77,98	205,176
c32	79,79	401,076

2 MV, 4 DOCKER	Request per second	Time per request
c1	8,66	115,527
c2	18,93	105,656
c4	37,94	105,428
c8	68,08	117,517
c16	73,45	217,848
c32	75,46	424,052

4 MV, 2 DOCKER	Request per second	Time per request
c1	8,69	115,130
c2	16,93	118,101
c4	39,13	102,213
c8	64,38	124,263
c16	64,50	248,060
c32	68,46	467,397

Anexo IV. Apache Benchmark – Alta Carga

Solo máquinas virtuales

1 MV	Request per second	Time per request
c1	1,05	948,498
c2	1,07	1,873,768
c4	1,12	3,574,600
c8	1,12	7,122,913
c16	1,12	14,263,578
c32	1,11	28,948,958

2 MV	Request per second	Time per request
c1	1,07	934,045
c2	2,28	878,729
c4	2,28	1,756,397
c8	2,28	3,515,740
c16	2,29	6,982,066
c32	2,32	13,786,023

4 MV	Request per second	Time per request
c1	1,05	954,607
c2	2,12	945,013
c4	4,23	946,008
c8	4,22	1,896,095
c16	4,40	3,635,797
c32	4,46	7,169,000

8 MV	Request per second	Time per request
c1	1,07	933,397
c2	2,17	920,908
c4	4,45	899,822
c8	8,73	916,737
c16	8,92	1,793,245
c32	8,53	3,753,183

Máquinas Virtuales + Docker

1 MV, 8 DOCKER	Request per second	Time per request
c1	1,11	899,135
c2	2,21	906,417
c4	4,53	882,504
c8	9,09	879,807
c16	9,37	1,707,736
c32	9,29	3,446,283

2 MV, 4 DOCKER	Request per second	Time per request
c1	1,11	903,084
c2	2,16	927,801
c4	4,49	890,688
c8	8,68	921,677
c16	8,87	1,804,225
c32	8,71	3,673,094

4 MV, 2 DOCKER	Request per second	Time per request
c1	1,10	912,532
c2	2,15	928,827
c4	4,42	905,736
c8	9,02	886,555
c16	8,98	1,782,603
c32	9,06	3,531,009

Anexo V. Consumo de recursos – Memoria RAM

Solo máquinas virtuales

Tiempo	MV1	MV2	MV3	MV4	MV5	MV6	MV7	MV8
t1	574476	584804	577388	586740	582044	591048	579104	581948
t2	507560	584804	577388	586740	582044	591048	579104	581948
t3	507560	584804	577388	586740	582044	591048	579104	581948
t4	507560	584804	577388	586740	582044	591048	579104	581948
t5	507560	584804	577388	586740	582044	591048	579104	581948
t6	507560	584804	577388	586740	582044	591048	579104	581948
t7	507560	584804	577388	586740	582044	591048	579104	581948
t8	507560	584804	577388	586740	582044	591048	579104	581948
t9	507560	584804	577388	586740	582044	591048	579104	581948
t10	507560	584804	577388	586740	582044	591048	579104	581948
t11	507560	584804	577388	586740	582044	591048	579104	581948
t12	507560	584804	577388	586740	582044	591048	579104	581948
t13	507560	584804	577388	586740	582044	591048	579104	581948
t14	507560	584804	577388	586740	582044	591048	579104	581948
t15	507560	584804	577388	586740	582044	591048	579104	581948
t16	507560	584804	577388	586740	582044	591048	579104	581948
t17	507560	584804	577388	586740	582044	591048	579104	581948
t18	507560	584804	577388	586740	582044	591048	579104	581948
t19	507560	584804	577388	586740	582044	591048	579104	581948
t20	507560	584804	577388	586740	582044	591048	579104	581948

1 Máquina virtual, 8 Docker

Tiempo	MV
t1	756180
t2	756180
t3	756180
t4	756180
t5	756180
t6	756180
t7	756180
t8	756180
t9	756180
t10	756180
t11	756180
t12	756180
t13	756180
t14	756180
t15	756180
t16	756180
t17	756180
t18	756180
t19	756180
t20	756180

Anexo VI. Consumo de recursos – Uso de CPU

Baja carga

Consumo por HaProxy

Tiempo	% CPU
t1	5,9
t2	10,9
t3	480,6
t4	788,2
t5	788,2
t6	789,1
t7	789,2
t8	790,1
t9	791,1
t10	792,1
t11	794,1
t12	795
t13	795
t14	796
t15	796
t16	796
t17	796
t18	796
t19	797
t20	797

Consumo por MV

Tiempo	% CPU
t1	6,2
t2	14,7
t3	85,3
t4	214,7
t5	234,4
t6	242,3
t7	247,5
t8	248,5
t9	267,6
t10	270,3
t11	272,5
t12	275,5
t13	290,1
t14	299
t15	302,2
t16	306,6
t17	308,9
t18	319,6
t19	319,6
t20	333,7

Alta carga

Consumo por HaProxy

t1	0
t2	0,1
t3	0,2
t4	0
t5	0
t6	0,2
t7	0,2
t8	0,1
t9	0
t10	0
t11	0,1
t12	0,2
t13	0,1
t14	0
t15	0
t16	0
t17	0,1
t18	0,2

Consumo por MV

t1	5,9
t2	10,9
t3	480,6
t4	788,2
t5	788,2
t6	789,1
t7	789,2
t8	790,1
t9	791,1
t10	792,1
t11	794,1
t12	795
t13	795
t14	796
t15	796
t16	796
t17	796
t18	796

t19	0,1
t20	0

t19	797
t20	797

Alta Carga – 8 Máquinas Virtuales

Tiempo	%MV1	%MV2	%MV3	%MV4	%MV5	%MV6	%MV7	%MV8
t1	3,9	3,9	0	3,9	0	0	2,9	7,8
t2	5,9	5,9	2,9	30,4	2,9	3,9	5,9	17,6
t3	31,4	30,4	30,4	83,3	31,4	31,4	30,4	33,3
t4	100	82,2	82,2	89,1	83,2	82,4	82,2	101
t5	100	90,2	84,2	94,1	88,2	90,1	93,1	101
t6	100	94,1	92,2	94,1	93,1	91,2	95	102
t7	101	95,1	93,1	94,1	93,4	93,1	97	102
t8	100	96,0	93,1	94,1	94,1	94,1	98	102
t9	101	96,0	95,0	96	94,1	95,1	99	102
t10	101	97,0	97,1	97	95	96	99	102
t11	101	97,0	98	97	95	97	99	102
t12	101	98,0	98	98	97	97	100	102
t13	100	98,0	98	98	97	98	101	102
t14	101	98,0	98	98	98	98	100	102
t15	100	99	99	98	98	99	101	102
t16	100	98	99	98	99	99	101	102
t17	101	99	99	99	99	99	101	102
t18	101	99	99	100	99	100	100	103
t19	100	99	100	101	100	100	101	103
t20	100	99	101	100	100	100	101	103

Consumo por la MV

Tiempo	% CPU
t1	6,2
t2	14,7
t3	85,3
t4	214,7
t5	234,4
t6	242,3
t7	247,5
t8	248,5
t9	267,6
t10	270,3
t11	272,5
t12	275,5
t13	290,1
t14	299
t15	302,2
t16	306,6
t17	308,9

t18	319,6
t19	319,6
t20	333,7

Anexo VII. Visual – Limes y Farhos

Visual Limes nace el año 1993 fruto de la visión de sus socios fundadores de aportar sus conocimientos en el desarrollo de sistemas de gestión de la información aplicadas al sector de la salud, observada la gran necesidad del uso de estas tecnologías en dicho sector de la sanidad por parte de los profesionales para mejorar la calidad asistencial de los pacientes al disponer de más información en tiempo real.

En este mismo año 1993 nace la primera versión de Nefrosoft®, software departamental para la gestión de la información de pacientes enfermos renales en servicios de nefrología de hospitales y clínicas de hemodiálisis.

Solo 4 años después del lanzamiento de Nefrosoft, en Visual Limes hicimos el lanzamiento de nuestro software para la gestión de farmacias de hospital Farhos® este desarrollo se realizó con la participación de Farmacéuticos de Hospital convirtiendo a Farhos en un referente a nivel nacional por lo completo que es en cobertura de las necesidades del servicio de farmacia de un hospital y lo robusto y fiable que es.

FarHos es un Sistema de Información basado en módulos para la Gestión de Farmacia Hospitalaria y Oncología. El conjunto de sus módulos resuelve todos los aspectos de la gestión de la farmacia de hospital (compras, gestión del almacén, oncología, prescripción electrónica médica asistida, distribución de medicamentos, dispensación individualizada a pacientes ingresados y ambulatorios, gestión de preparación de mezclas intravenosas, gestión de estudios de farmacocinética, generación de informes y estadísticas, integraciones con almacenes robotizados y otros sistemas de información hospitalarios).

El TFG se centra en el módulo de módulo de Prescripción Electrónica de FarHos.

Glosario

- **Back end**. Arquitectura interna de un servicio que asegura el correcto funcionamiento de todos sus componentes.

- **Capa, *layer*** (modelo OSI). El modelo OSI (Open System Interconnection) es un estándar que tiene por objetivo interconectar sistemas de distinta procedencia para que puedan intercambiar información en la red. El modelo está dividido en siete capas (layer) o niveles de abstracción.

- ***Cookie***. Pequeña información enviada por un sitio web y almacenada en el navegador del usuario, de manera que el sitio web puede consultar la actividad previa del navegador.

- **Daemon (Demonio como servicio)**. Proceso que se ejecuta en segundo plano en lugar de estar bajo el control directo de un usuario interactivo.

- **Escalable**. Un sistema se dice que es escalable si puede manejar la adición de usuarios y recursos sin sufrir una pérdida apreciable de rendimiento o un incremento de su complejidad administrativa.

- **Fencing**. Proceso de aislar un nodo de un clúster o proteger los recursos compartidos cuando un nodo presenta un comportamiento anómalo.

- **Front end.** Parte de un servicio que conecta e interactúa con los usuarios que lo utilizan.

- **Instancia**. Proceso en el que se ejecuta un servicio.

- **Equilibrado de carga.** Técnica utilizada para dividir y compartir el trabajo entre varios procesos u ordenadores de la manera más equitativa posible.

- **Nodo trabajador, *worker*, servidor real…** Servidor que implementa el servicio ofrecido encargado de procesar las peticiones y emitir respuestas.

- **Persistencia y afinidad**. Ambas implican el hecho de asignar un cliente a un servidor durante un periodo de tiempo. Sin embargo, se diferencian por el tipo de información manejada. **Afinidad** utiliza información tanto de la capa de transporte (*layer 4*) como de la capa de aplicación (*layer* 7), como por ejemplo la dirección IP y el puerto. **Persistencia** utiliza información sólo de la capa de aplicación, como por ejemplo las *cookies*.

- **Punto único de fallo** o *Single Point of Failure*. Componente de un sistema que, en caso de fallo, ocasiona la detención global del sistema completo.

- **Replicación (de recursos)**. Proceso mediante el cual se genera una copia exacta de un elemento del sistema para mejorar la fiabilidad, accesibilidad y tolerancia a fallos.

- **Servicio web**. Interfaz mediante la que dos máquinas o aplicaciones se comunican entre sí. Facilita un servicio a través de Internet.