UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Infraestructura flexible multi-cliente multi-servidor para aplicaciones de tail latency en entornos Cloud

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Li, Zhilin

Tutor/a: Petit Martí, Salvador Vicente

Cotutor/a: Pons Escat, Lucía

Cotutor/a: Sahuquillo Borrás, Julio

CURSO ACADÉMICO: 2023/2024

Agrademientos

Con este trabajo pongo fin a una etapa relevante de mi vida. Por eso, quiero dar mis más sinceros agradecimientos a las personas que me han ayudado en este proyecto. En primer lugar, quiero dar las gracias a mis tutores, Salvador, Lucía y Julio, por su conocimiento y orientación, así como la paciencia para soportarme durante todo este tiempo. Me han mostrado cómo es el mundo de la investigación.

Agradezco también a mi familia por todo el amor, el apoyo y motivación incondicional que me han desde siempre. Asimismo, a todos los compañeros y amigos que he conocido en este tiempo, gracias por acompañarme en estos 5 años de trayecto. En especial, a Carolina, Roser, María, Qiyue, Shiwan, Luis y Alejandro por su amistad y constante apoyo.

Por último, doy las gracias a la UPV por proveerme del entorno y de los recursos cruciales para mi desarrollo y formación académica. Han sido fundamentales no solo para el proyecto, sino también para mi futuro.

Gracias a todos; sin vosotros, el proyecto no habría tenido éxito.

Resum

Tradicionalment, els objectius de prestacions dels sistemes distribuïts s'enfocaven principalment a reduir el temps mitjà de resposta de les aplicacions. No obstant això, en els entorns *cloud* actuals, la mètrica a millorar és la latència de cua o *tail latency* en compte de la latència de mitjana. Aquesta mètrica representa el temps de resposta més llargs patit per les peticions de les aplicacions, habitualment els del percentil 95 o superior. La latència de cua és important perquè si augmenta excessivament pot fer perillar els acords de nivell de servici o SLA, la qual cosa afectaria l'experiència dels usuaris. Esta mètrica és altament sensible a la infraestructura del sistema, la distribució de càrrega entre els servidors i les característiques de les aplicacions.

Alguns dels *benchmark suites* de gran ús com Tailbench presenten serioses limitacions respecte a un sistema real, la qual cosa restringix el rang d'estudis d'avaluació que es poden realitzar. En concret, operen en un entorn molt controlat on el nombre de clients, el nombre de peticions i la càrrega són fixos; un escenari no representatiu dels servicis distribuïts.

El present treball perseguix eliminar estes limitacions, centrant-se en el disseny d'una infraestructura multi-client i multi-servidor basada en Linux Virtual Server que permeta estudiar entorns realistes de manera flexible amb càrregues dissenyades per a l'avaluació de la latència de cua, com són les pertanyents a la *suite* de *benchmarks* Tailbench. Per a això, es necessari adaptar les aplicacions Tailbench per a la seua execució en esta nova infraestructura. Les aplicacions modificades, Tailbench++, permeten definir situacions en les quals els servidors esperen un nombre de clients indefinit, i cada client realitza un nombre variable de sol·licituds, permetent alterar la seua càrrega dinàmicament.

L'objectiu del treball és desenvolupar un entorn que permeta realitzar estudis sobre el *cloud* de manera més precisa. En concret, l'entorn perseguix considerar, entre altres indicadors, la interferència en els recursos compartits dels servidors, com la cache d'últim nivell, i l'amplada de banda de memòria principal. Entre els estudis als quals es dedicarà l'entorn cal ressaltar, la implementació i anàlisi d'estratègies de balanceig de càrrega.

Paraules clau: Balanceig de càrrega, latència de cua, aplicacions de latència crítica, computació en el núvol, Linux Virtual Server, Tailbench, multi-client, multi-servidor, infraestructura experimental, benchmark, servicis distribuïts

Resumen

Tradicionalmente, los objetivos de prestaciones de los sistemas distribuidos se enfocaban principalmente en reducir el tiempo medio de respuesta de las aplicaciones. Sin embargo, en los entornos *cloud* actuales, la métrica a mejorar es la latencia de cola o *tail latency* en vez de la latencia media. Esta métrica representa los tiempos de respuesta más largos sufridos por las peticiones de las aplicaciones, habitualmente los del percentil 95 o superior. La latencia de cola es importante porque si aumenta excesivamente, puede hacer peligrar los acuerdos de nivel de servicio o SLA, lo que afectaría a la experiencia de los usuarios. Esta métrica es altamente sensible a la infraestructura del sistema, la distribución de carga entre los servidores y las características de las aplicaciones.

Algunos de los *benchmark suites* de gran uso como Tailbench presentan serias limitaciones respecto a un sistema real lo que restringe el rango de estudios de evaluación que se pueden realizar. En concreto, operan en un entorno muy controlado donde el número de clientes, número de peticiones y carga son fijos; un escenario no representativo de los servicios distribuidos.

El presente trabajo persigue eliminar estas limitaciones, centrándose en el diseño de una infraestructura multi-cliente y multi-servidor basada en Linux Virtual Server que permita estudiar entornos realistas de forma flexible con cargas diseñadas para la evaluación de la latencia de cola, como son las pertenecientes a la *suite* de *benchmarks* Tailbench. Para lograr este fin, es necesario adaptar las aplicaciones Tailbench para su ejecución en esta nueva infraestructura. Las aplicaciones modificadas, Tailbench++, permiten definir situaciones en las que los servidores esperan un número de clientes indefinido, y cada cliente realiza un número variable de peticiones, permitiendo además poder alterar su carga dinámicamente en el tiempo.

El objetivo de este trabajo es desarrollar un entorno que permita realizar estudios sobre el *cloud* de manera más precisa. En concreto, el entorno persigue considerar, entre otros indicadores, la interferencia en los recursos compartidos de los servidores, como la cache de último nivel, y el ancho de banda de memoria principal. Entre los estudios a los que se dedicará el entorno cabe resaltar, la implementación y análisis de estrategias de balanceo de carga.

Palabras clave: Balanceo de carga, latencia de cola, aplicaciones de latencia crítica, computación en la nube, Linux Virtual Server, Tailbench, multi-cliente, multi-servidor, infraestructura experimental, benchmark, servicios distribuidos

Abstract

Traditionally, the main focus of distributed systems' performance objectives has been reducing the average application response time. However, in today's cloud environments, the metric to improve is tail latency rather than average latency. This metric represents the longest response times experienced by application requests, typically those in the 95th percentile or higher. Tail latency is important because increasing excessively can compromise service level agreements or SLAs, affecting the user experience. This metric is highly sensitive to system infrastructure, server load distribution, and application characteristics.

Some of the widely used benchmark suites, such as Tailbench, have serious limitations concerning a real system, which limits the scope of studies that can be performed. In particular, they operate in a very controlled environment where the number of clients, the number of requests, and the load are fixed, a scenario that is not representative of distributed services.

This work aims to remove these limitations, focusing on the design of a multi-client and multi-server infrastructure based on Linux Virtual Server that allows reproducing realistic environments with workloads designed for the evaluation of tail latency, such as those belonging to the Tailbench benchmark suite. For this purpose, the Tailbench applications need to be adapted to run on this new infrastructure. The modified applications, Tailbench++, allow for more realistic situations to be reproduced, in which the servers wait for an indefinite number of clients and where each client makes a variable number of requests, enabling them to alter the load dynamically.

The main objective of this work is to develop an environment that allows cloud studies to be carried out more accurately. Specifically, the environment aims to consider, among other indicators, interference among shared server resources, such as the last level cache and main memory bandwidth. Among the studies to which the environment will be dedicated, it is worth highlighting the implementation and analysis of load-balancing strategies.

Key words: Load balancing, tail latency, latency-critical applications, cloud computing, Linux Virtual Server, Tailbench, multi-client, multi-server, experimental infrastructure, benchmarking, distributed services

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

Hoy en día, Internet ofrece una variedad de servicios informáticos. Estos servicios ofrecidos son soportados por unas infraestructuras hardware-software que, dependiendo de las funcionalidades y el propósito de estos servicios, pueden ser más o menos complejas. Además, en un mundo cada vez más interconectado, donde el número de usuarios de internet es creciente [1], los servicios se enfrentan a un mayor nivel de estrés y de carga. Esto resulta en una necesidad relevante de mejorar el rendimiento, la escalabilidad, la eficiencia y la flexibilidad de las infraestructuras informáticas actuales.

Ante esta situación, tecnologías como los sistemas distribuidos y la computación en la nube han adquirido una importante relevancia con el objetivo de abordar estos desafíos. Estas tecnologías mencionadas permiten aprovechar adecuadamente los recursos de cómputo de forma dinámica, facilitando la escalabilidad de los servicios para adaptarse efectivamente al número de usuarios. Aunque han permitido tratar de manera significativa los problemas presentes en buena parte de los servicios informáticos [2], siguen existiendo ciertos desafíos que no pueden ser abordados apropiadamente del todo. Un ejemplo de estos retos sería la latencia de cola [3, 4].

Se trata de un problema que no tiene relevancia si el servicio ofrecido es intensivo en cálculos. Sin embargo, para las aplicaciones sensibles a la latencia, como pueden ser los sistemas en tiempo real, es una limitación grave, ya que cualquier pequeña variación en el tiempo de respuesta puede comprometer gravemente la experiencia y satisfacción del usuario. Por ejemplo, situaciones indeseadas como esperar mucho tiempo para los resultados de una búsqueda web. En el caso de que suceda, lo más probable es que los usuarios cambiarían de buscador web.

Para abordar estas dificultades, además de otras como pueden ser la gestión energética, la disponibilidad o la seguridad que presentan estas tecnologías y que influencian gravemente la calidad de los servicios informáticos actuales, es crucial desarrollar e implementar estrategias y soluciones efectivas. Para conseguirlo, es fundamental poder reproducir situaciones reales de los entornos de ejecución de estas aplicaciones, ya que permitiría estudiar y analizar en detalle los problemas y retos que presentan. Aunque este proceso no sea sencillo debido a la complejidad de las infraestructuras y las características de las cargas de trabajo de los servicios, configurar una infraestructura experimental y emular adecuadamente los servicios reales es esencial. De lo contrario, no es posible analizar las limitaciones y validar las soluciones desarrolladas apropiadamente, arrojando resultados y conclusiones que no son aplicables a entornos de producción. Todo esto podría suponer, en definitiva, no solo la ineficiencia del servicio, sino también una pérdida económica y competitiva del proveedor.

1.1 Motivación

1.1.1. Motivación personal

Antes de comenzar con el Grado en Ingeniería Informática, ya sentía especial interés por el mundo tecnológico. Me pasaba algunas de mis tardes investigando sobre el funcionamiento de los ordenadores, instalando software y otras herramientas para este fin. Gracias a ello, adquirí unos conocimientos básicos sobre la informática. Sin embargo, no fue hasta cursar la carrera que me di cuenta de la complejidad del funcionamiento de un ordenador gracias a las asignaturas que cursé. De todas ellas, me fascinaban concretamente las materias impartidas en las asignaturas FSO (Fundamentos de Sistemas Operativos) y ETC (Estructura de Computadores). Dos asignaturas que presentan el funcionamiento de un ordenador desde distintos puntos de vista. La primera ilustra el ordenador desde un nivel superior, abstrayendo los recursos físicos. Mientras que la segunda explica los componentes físicos del ordenador a un nivel hardware.

A medida que adquiría más conocimiento en la universidad, me di cuenta de que no se podían tratar los temas de la informática estudiados en las asignaturas por separado. Todo estaba relacionado para poder ofrecernos los servicios informáticos que disfrutamos hoy en día. Asimismo, la carrera de ADE (Administración de Empresas) me ha aportado una perspectiva distinta que ha reforzado también esta comprensión, así como el potencial que tiene la informática aplicado a otras disciplinas, como es el caso de los beneficios de las optimizaciones de los procesos de negocio en las empresas. Entonces, la idea de llevar a cabo un TFG (Trabajo de Fin de Grado) que pudiera combinar varios campos de la informática y que fuera de utilidad tanto para mis futuros estudios como para la sociedad, surgió en mi mente. Por este motivo, me uní al grupo de investigación GAP (Grupo de Arquitecturas Paralelas) de la UPV, gracias a mis tutores de este proyecto. Ellos me propusieron abordar el problema de la latencia de cola, un aspecto crucial para la eficiencia y efectividad de los sistemas distribuidos que soportan los servicios de internet en la actualidad. Como primer paso, debía diseñar e implementar un entorno experimental adecuado que soportaría los posteriores estudios. Se trata de una tarea compleja, ya que exige aplicar conocimientos de diversas áreas de la informática, como los sistemas operativos, las redes de computadores o la arquitectura de computadores. Sin embargo, me motivaba mucho la temática y el poder aplicar estos conocimientos aprendidos, razones por las que finalmente elegí este tema para mi TFG.

1.1.2. Motivación profesional

Mejorar las prestaciones de los servicios informáticos está cobrando cada vez más importancia en la actualidad. Esto no es debido únicamente al creciente número de internautas en todo el mundo, sino también a la proliferación de aplicaciones que tienen unos exigentes requisitos de rendimiento para asegurar un funcionamiento de calidad.

Un ejemplo de este tipo de servicios son las aplicaciones de latencia crítica los videojuegos en línea, los servicios de transmisión de vídeo, o la inteligencia artificial aplicada a la conducción de vehículos. Estas aplicaciones se caracterizan por soportar grandes cantidades de peticiones y responder en el mínimo tiempo posible cumpliendo unas restricciones impuestas para garantizar su calidad servicio. Garantizar el cumplimiento de estos exigentes requisitos es fundamental para ofrecer una experiencia satisfactoria a los usuarios. En este contexto, el uso de tecnologías como los sistemas distribuidos y la computación en la nube se ha consolidado como una solución ampliamente adoptada para abordar este desafío. Estas tecnologías destacan por sus características como la eficiencia, la escalabilidad, la flexibilidad, la fiabilidad, entre otras. Además, permiten

proporcionar un nivel de rendimiento adecuado para la mayoría de los servicios informáticos. Sin embargo, para abordar los exigentes requisitos de algunos servicios, como el mencionado en el párrafo anterior, requieren implementar ciertas estrategias y técnicas adicionales. Para desarrollar estas últimas, es crucial, como primer paso, disponer de un entorno experimental con unas aplicaciones de prueba adecuadas.

Estas infraestructuras y cargas de prueba deben cumplir unos ciertos requisitos. En primer lugar, la infraestructura debe reflejar fielmente la realidad de los entornos de producción, donde múltiples servidores proporcionan servicio a múltiples clientes. En segundo lugar, el entorno de pruebas debe ser flexible, permitiendo adaptarse y expandirse fácilmente según los requerimientos y escenarios que surjan. Asimismo, es esencial emplear cargas de trabajo que tengan el mismo comportamiento que los servicios informáticos. Cumpliendo estas características, se podrán realizar experimentos que reproduzcan escenarios realistas, permitiendo no solo identificar y analizar detalladamente problemas existentes, sino también validar soluciones desarrolladas. Todo esto posibilita la extracción de resultados y conclusiones que pueden ser extrapolados a los entornos operativos, evitando los costes e imprevistos que puedan surgir si se hiciera directamente sobre estos últimos.

Hasta el momento, el GAP (Grupo de Arquitecturas Paralelas) disponía de un entorno experimental [5]. No obstante, se trata de una infraestructura de pruebas que no es adecuada para estudiar las aplicaciones de los sistemas distribuidos actuales que están formados por múltiples servidores y clientes. El entorno consistía de un sistema de un único par cliente-servidor, limitando las capacidades de realizar pruebas con múltiples clientes y servidores. Por lo tanto, surge la necesidad de este trabajo de desarrollar una infraestructura experimental multi-cliente y multi-servidor. Este entorno es de especial relevancia para poder llevar a cabo trabajos futuros y proyectos con empresas que puedan mejorar e innovar el campo de la informática.

1.2 Objetivos

El objetivo principal del presente trabajo es diseñar e implementar un entorno de pruebas flexible que refleje fielmente la realidad de las infraestructuras de producción de los sistemas distribuidos y de la computación en la nube, donde múltiples clientes son atendidos por múltiples servidores. Esta infraestructura debe poder abordar los estudios, las investigaciones de desafíos que se enfrenta la informática y los proyectos que puedan surgir con empresas. Con este propósito, se definen los siguientes objetivos secundarios:

- Implementar una infraestructura *cloud* experimental, formada por una máquina *master* o director encargado de distribuir las conexiones de los clientes a los servidores que procesan y responden las solicitudes recibidas.
- Elegir y adaptar aplicaciones de latencia crítica que sean representativas para reproducir escenarios realistas. En concreto, el proceso servidor debe poder esperar un número indeterminado de clientes. A su vez, el cliente debe ser capaz de decidir las peticiones máximas a enviar y poder variar su carga durante la ejecución. Como resultado, nace Tailbench++, una adaptación del conjunto de aplicaciones Tailbench [6] que permite reproducir de manera más fiel los servicios informáticos actuales.
- Estudiar el comportamiento de los tiempos de respuesta de las aplicaciones de latencia crítica elegida en el entorno de pruebas diseñado, con el objetivo de validar el correcto funcionamiento tanto de las aplicaciones como de la infraestructura diseñada.

- Garantizar la correcta configuración de las interconexiones de red, lo que incluye la asignación adecuada de las direcciones IP (Internet Protocol) [7], con el fin de asegurar una comunicación efectiva entre las máquinas físicas y virtuales.
- Configurar el balanceador de carga para su correcto funcionamiento y estudiar si el distribuidor seleccionado presenta alguna limitación o espacio para futuras mejoras.

1.3 Impacto esperado

Se pretende diseñar una infraestructura de pruebas utilizando las máquinas del GAP, asegurando que ésta pueda ser escalable y adaptable, para llevar a cabo, junto con las cargas de trabajo adaptadas, futuros estudios que exploren temáticas, como por ejemplo estrategias de balanceo de carga para aplicaciones de latencia crítica, y que puedan innovar y optimizar las tecnologías informáticas.

1.4 Metodología

Para poder cumplir con los objetivos propuestos en el presente trabajo, se van seguir los siguientes pasos:

1. **Investigación sobre los temas a tratar**. Como primer paso, se va a recabar información relevante sobre las temáticas como la latencia de cola, las cargas de trabajo representativas y existentes, así como los sistemas y tecnologías que soportan estos servicios informáticos.
2. **Diseño de la infraestructura experimental**. Antes de estudiar las aplicaciones de latencia crítica, será necesario definir el entorno experimental y adaptarla para cumplir con los objetivos del presente trabajo.
3. **Elección de las aplicaciones**. Se seleccionará, entre un conjunto de aplicaciones, las más representativas para el estudio de la latencia de cola. Asimismo, se adaptarán para poder ser ejecutadas en el entorno de pruebas diseñado.
4. **Evaluación global**. Por último, se evaluarán las funcionalidades de las aplicaciones seleccionadas en la infraestructura experimental y el correcto funcionamiento de esta última mediante experimentos y pruebas.

1.5 Estructura de la memoria

La estructura de la memoria de este Trabajo Fin de Grado está formado por 7 capítulos, sin tener en cuenta el presente. A continuación, se ofrece una breve descripción de cada uno de ellos:

- El Capítulo 2 presenta algunos conceptos complementarios como la computación en la nube, la virtualización o la latencia de cola que ayudan a una mejor comprensión de los temas del presente TFG.
- El Capítulo 3 analiza trabajos previos e investigaciones de los temas relacionados, resaltando las principales diferencias con el trabajo presentado en este proyecto.

- El Capítulo 4 describe el diseño de la infraestructura de pruebas y las herramientas seleccionadas para su posterior implementación.
- El Capítulo 5 presenta los pasos a seguir para la configuración e implementación del entorno experimental.
- El Capítulo 6 detalla las aplicaciones de latencia crítica seleccionadas, las limitaciones que presentan y las modificaciones realizadas para solventarlas.
- El Capítulo 7 evalúa las nuevas funcionalidades de las aplicaciones y presenta algunos casos de estudio prácticos para ilustrar el uso de la infraestructura y las aplicaciones propuestas en este proyecto.
- Para finalizar, el Capítulo 8 relata las principales conclusiones de este trabajo y las posibles lineas de trabajo futuro a seguir.
- Se han incluido cuatro apéndices con información adicional. El Apéndice A presenta capturas adicionales del proceso de creación e instalación de una máquina virtual. El Apéndice B muestra ejemplos de los ficheros de configuración de red de las máquinas virtuales utilizadas para evaluar la infraestructura experimental. El Apéndice C presenta bloques de código de algunas funciones claves modificadas para adaptar las cargas Tailbench. Por último, el Apéndice D, detalla los Objetivos de Desarrollo Sostenible relacionados con el presente trabajo.

CAPÍTULO 2
Marco teórico

Con el objetivo de ayudar a entender mejor el contexto y visión del presente trabajo, en este capítulo se describen los conceptos teóricos relacionados, así como las tecnologías y herramientas empleadas para desarrollar el presente trabajo. En primer lugar, se presenta el modelo OSI de redes de comunicaciones. En segundo lugar, se introduce la computación en la nube y las técnicas de virtualización y balanceo de carga. Finalmente, se presentan las cargas de latencia crítica y la métrica de prestaciones latencia de cola empleada para evaluar estas cargas.

2.1 Modelo OSI

Las redes de comunicación han evolucionado con el paso del tiempo. Sus componentes, funcionalidades y arquitecturas actuales son mucho más sofisticados que antes. Desde su aparición hasta la actualidad, se han desarrollado múltiples modelos para definir las características y cualidades que deben tener estos sistemas de comunicación. Algunos ejemplos son el modelo TCP/IP (Transmission Control Protocol/Internet Protocol) [8] o el modelo OSI (Open Systems Interconnection) [9]. Este segundo es el que se va a detallar a continuación, debido a que es el que emplea Linux Virtual Server empleado en este proyecto.

El modelo OSI es un marco de referencia que define y clasifica las funcionalidades y tareas necesarias para el correcto intercambio de información entre los sistemas de comunicación [10, 11, 9]. Fue diseñado por la Organización Internacional de Normalización (ISO (International Standard Organization)) en 1980. Desde entonces, se ha convertido en una guía para el desarrollo de los protocolos de red, reglas, pasos o estándares a seguir con el objetivo de asegurar una transmisión de datos eficaz. Este marco define en total siete niveles o capas, cada una con un rol determinado.

- **Capa 7, o de aplicación**. Es el nivel más alto y encargado de proveer a los usuarios y programas la interfaz para el uso de los servicios de red como la transferencia de archivos o el correo electrónico.
- **Capa 6, o de presentación**. Su función principal es la traducción del formato de los mensajes que genera el nivel superior al formato de la red y viceversa.
- **Capa 5, o de sesión**. Su misión consiste en gestionar las conexiones, que incluye el establecimiento, el mantenimiento y la sincronización entre las partes.
- **Capa 4, o de transporte**. Se encarga de proporcionar una comunicación fiable y eficiente de extremo a extremo entre los procesos.

- **Capa 3, o de red**. Es responsable del direccionamiento y enrutamiento de los paquetes entre distintas redes.

- **Capa 2, o de enlace de datos**. Se hace cargo de la transferencia fiable de datos entre los nodos a través de un medio físico que no lo es.

- **Capa 1, o física**. Es el nivel más bajo y establece la codificación y representación de los valores 0 y 1 en señales adaptas al medio de transmisión.

2.2 Computación en la nube

Antes, los recursos de cómputo eran sumamente valiosos. Pocas personas y organizaciones tenían acceso a éstos. No obstante, con el desarrollo tecnológico, esta barrera que impedía su disponibilidad ha desaparecido. En la actualidad, estos recursos están al alcance de cualquier persona u organización. Una de las causas de la situación actual se debe a los avances de la computación en la nube, o *cloud computing* [12, 13, 14], especialmente sobre los de finales del siglo XX y principios del XXI, donde ha proliferado su uso y desarrollo [15]. Hoy en día, la computación en la nube juega un papel fundamental en la mayoría de los servicios informáticos que disfrutamos, y es utilizada por numerosas organizaciones para el despliegue de sus infraestructuras informáticas.

Según NIST (National Institute of Standards and Technology) [16], la computación en la nube es una innovación tecnológica que permite al usuario utilizar una serie de recursos informáticos, compartidos a través de Internet, de manera eficiente y flexible en base a sus necesidades. Estos recursos no pertenecen exclusivamente al usuario, sino que están disponibles bajo demanda, y normalmente se encuentran en grandes centros de datos. Se trata de una solución para abordar problemas que afrontan las organizaciones habitualmente como el alto coste de adquirir y mantener infraestructuras hardware y software privadas, y la dificultad de adaptar la capacidad de los recursos a la demanda de cada instante. Como consecuencia de ello, los recursos se infrautilizan cuando hay baja demanda mientras que se producen situaciones de contención en el caso contrario.

2.2.1. Características de la computación en la nube

A partir de la definición proporcionada por NIST, se establecen cinco características esenciales [12] de la computación en la nube:

- **Acceso ubicuo**. El usuario puede acceder a los recursos informáticos donde quiera y cuando quiera, siempre que tenga acceso a internet.

- **Autoservicio bajo demanda**. El cliente tiene capacidad de autoaprovisionarse los recursos que necesita.

- **Elasticidad**. El sistema puede adaptarse a los requerimientos de los clientes según su situación y nivel de carga añadiendo más recursos, o liberándolos.

- **Servicio medido**. El usuario paga solo por los recursos que utiliza.

- **Recursos compartidos**. Los recursos de la nube están distribuidos espacialmente entre numerosos servidores, y son utilizados por múltiples usuarios o clientes.

2.2.2. Modelos de servicio

Existen diferentes tipos de *cloud computing* dependiendo del modelo de servicio [17, 13, 18, 14] del proveedor, entre los cuales destacan los siguientes:

- ***Infrastructure as a Service***, o **IaaS**. En este modelo el proveedor ofrece recursos físicos, como almacenamiento, memoria principal o CPU, a través de máquinas virtuales, contenedores o incluso directamente sobre máquinas físicas. Esto implica que el cliente tiene un mayor control sobre la infraestructura que contrata.
- ***Platform as a Service***, o **PaaS**. En este caso el proveedor proporciona no solo recursos informáticos, sino también un entorno software ya configurado. Esto facilita en gran medida el despliegue de aplicaciones sobre el sistema para los clientes.
- ***Software as a Service***, o **SaaS**. A diferencia de los modelos ya mencionados, el proveedor ofrece a los usuarios aplicaciones ya configuradas y listas para su uso. Como resultado, el cliente no tiene que preocuparse por la gestión ni el mantenimiento de la infraestructura.

2.2.3. Modelos de despliegue

Según la manera de desplegarlo [18, 13, 12], se puede clasificar la computación en la nube en las siguientes categorias:

- **Nubes privadas**. La infraestructura *cloud* está al servicio de una sola organización. Es decir, las entidades (personas u organizaciones) externas no tienen permiso para su uso. Esto proporciona un mayor seguridad y control de los datos e información. En cuanto a la gestión del sistema, puede ser administrada tanto por un tercero como por la propia organización.
- **Nubes públicas**. A diferencia de la nube privada, la infraestructura puede ser utilizada por cualquier entidad que contrate el servicio con el proveedor, quien es también el encargado de su gestión.
- **Nubes híbridas**. Se trata de una mezcla entra nube privada y nube pública. La organización puede tener información confidencial en la parte privada, y exponer en la parte pública servicios y datos que puedan ser utilizados por entidades ajenas.
- **Nubes de comunidad**. Se podría considerar también como un tipo de nube privada. Sin embargo, la infraestructura está al servicio de un conjunto de organizaciones que tienen unos intereses comunes.

2.3 Virtualización

La virtualización es una tecnología que ha cobrado una creciente importancia con los avances tecnológicos en los últimos años. Especialmente con el auge de la computación en la nube, se ha convertido en una de las bases de los servicios informáticos que hoy en día se utilizan.

Podemos entender la virtualización como una técnica para simular un sistema real, sus características y comportamientos mediante software [19]. En el ámbito del *cloud computing*, la virtualización es uno de los pilares fundamentales para conseguir una alta escalabilidad, disponibilidad y eficiencia, además de facilitar el despliegue y control de los servicios distribuidos.

Para poder comprender mejor el diseño e implementación de la infraestructura *cloud* presentada en este trabajo, se introducen brevemente los siguientes conceptos relacionados con la virtualización:

- **Máquina anfitriona** o ***host***. Consiste en la máquina física real formada por los recursos físicos, y soporta el despliegue de múltiples máquinas virtuales a través de un hipervisor.
- **Máquina virtual** o **huésped**. Se trata de una simulación completa de una computadora física. Ejecuta su propio sistema operativo y presenta la mayoría de las características de un ordenador real. Accede a los recursos físicos reales del anfitrión a través de las abstracciones de estos ofrecidas por el hipervisor [19, 20].
- **Hipervisor**. Es el intermediario entre los recursos físicos del anfitrión y las máquinas virtuales. Su propósito es proveer a los huéspedes el entorno que necesitan para su correcto funcionamiento. Se pueden distinguir dos tipos de hipervisores dependiendo de si se comunican directamente o no con el hardware de la máquina física. El primer tipo, también conocido como *bare metal*, interactúa de manera directa con los recursos físicos. Mientras que el segundo, se sitúa sobre un sistema operativo y aprovecha las llamadas al sistema para relacionarse con los componentes reales [19].
- **Contenedor**. Al igual que las máquinas virtuales, los contenedores son huéspedes de una máquina anfitriona. Sin embargo, los contenedores son más ligeros ya que no tienen un sistema operativo propio, sino que comparten el de la máquina anfitriona a través de de un conjunto de librerías y dependencias [19, 21].

2.4 Balanceo de carga

Con el aumento continuo de los usuarios, un solo servidor no tiene la capacidad suficiente para procesar todas las peticiones que le llegan. Por ese motivo, múltiples servidores son configurados para ofrecer un mismo servicio. Sin embargo, con esta solución surge la problemática de cómo repartir estas solicitudes de forma eficaz y eficiente entre el conjunto de nodos, mejorando así el rendimiento general del servicio. Ésta es la razón por la cual el balanceo de carga, o *load balancing* ha adquirido tanta relevancia en la actualidad.

El balanceo (también denominado equilibrado) de carga es un concepto aplicable a diversos ámbitos de la informática. En este trabajo nos centramos en su aplicación a los sistemas y servicios distribuidos. En este contexto, se define balanceo de carga [22] como una técnica o proceso para organizar y distribuir el tráfico de conexiones de un servicio entre varios servidores. Para ello, es necesario un elemento clave, el balanceador/equilibrador de carga, o *load balancer*. Se trata de un componente que gestiona las conexiones de los usuarios, y, en base a una determinada política o algoritmo de balanceo de carga, decide qué servidor se hará cargo de cada conexión. Desde el punto de vista externo al sistema, todo este proceso es transparente. Es decir, para el usuario la funcionalidad es la misma que si solo hubiera un único servidor.

En el ámbito de la computación en la nube, el equilibrado de carga es un elemento esencial para asegurar la eficacia y eficiencia de las infraestructuras y sistemas [23]. Por un lado, permite mejorar la eficiencia de los recursos, evitando la sobrecarga de unos y la infrautilización de otros. Por otro lado, ayuda a disminuir el tiempo de respuesta y actuar ante posibles fallos del servicio, como la caída de algún servidor. Todo esto favorece en

gran medida la satisfacción de los usuarios y el cumplimiento de los requisitos de nivel de servicio o SLA (Service Level Agreement).

Dependiendo de la información con la que funcionan, los algoritmos de balanceo de carga se pueden clasificar en dos grupos[24, 25]:

- **Algoritmos estáticos**. Este tipo de algoritmos necesita datos previos sobre el sistema para su funcionamiento. Es decir, las distribución de la carga se basa en información conocida de antemano sobre los servidores o nodos. Esta información no se modifica ni se actualiza con el paso del tiempo. Por lo tanto, los algoritmos estáticos son más adecuados para entornos donde el nivel de tráfico presenta pocas variaciones.

- **Algoritmos dinámicos**. A diferencia de los anteriores, los algoritmos dinámicos actualizan su lógica interna en base al estado de los recursos. En este sentido, es necesario monitorizar el estado de los servidores. Como resultado, los algoritmos dinámicos permiten adaptar la distribución de la carga a las circunstancias de cada instante. Por este motivo, son más adecuados para entornos dinámicos o cambiantes.

2.5 Aplicaciones de latencia crítica

Con el avance tecnológico, las infraestructuras en la nube (tanto el hardware como el software) han evolucionado radicalmente, lo que ha impactado de forma relevante los servicios informáticos. En particular, ha habido un aumento progresivo y continuado de la implantación de aplicaciones o servicios de latencia crítica [26]. Estas aplicaciones se caracterizan por exigir unos tiempos de respuestas muy bajos para garantizar un correcto funcionamiento y experiencia de usuario aceptable ante grandes volúmenes de conexiones de usuarios [6, 26].

Algunas de las aplicaciones de latencia crítica más tradicionales son la búsqueda web, o los videojuegos en línea. Por otro lado, aplicaciones que soportan tecnologías emergentes como el internet de las cosas (IoT (Internet of Things)), la realidad aumentada o los sistemas de conducción autónoma también se considerarían sensibles a la latencia [27]. En gran parte de los servicios mencionados, se observa que es necesaria una interacción en tiempo real entre el usuario y el sistema.

2.6 Latencia de cola

Tradicionalmente, cuando las cargas de trabajo eran menos interactivas, para medir las prestaciones de los servidores se utilizaban métricas basadas en medias, como por ejemplo el número medio de peticiones procesadas por segundo o el tiempo medio de respuesta. Sin embargo, en el panorama actual estas métricas ya no se consideran adecuadas debido a la gran escala y complejidad de los centros de computo, además de la proliferación de las aplicaciones de latencia crítica, que abarcan gran parte de los servicios utilizados hoy en día [6, 28, 29].

Las aplicaciones de latencia crítica presentan estructuras complejas con múltiples componentes. Esto implica que cuando un usuario realiza una petición, su respuesta dependa de varios servidores. Como consecuencia de ello, el tiempo de respuesta del servicio está sujeto al servidor que tarda más en contestar. En este contexto, es incompatible el uso de métricas tradicionales, ya que no proporcionan una representación real de

la situación. En lugar de ello, surge la métrica de la latencia de cola o *tail latency*. Se trata principalmente del tiempo de respuesta del porcentaje de peticiones con un servicio más lento. Normalmente, se utilizan el percentil 95 o el 99 [28, 6, 3].

Para explicar la importancia que supone esta métrica, nos basaremos en el caso de ejemplo mencionando en [3]. Suponiendo un servicio que dispone 100 servidores, los cuales tienen una respuesta media de 10ms; si la latencia de cola en el percentil 99 es de 1s, esto significa que 1 de cada 100 peticiones tarda más de 1s. Entonces, en el caso de una aplicación de latencia crítica donde cada usuario debe recoger las respuestas de todos los servidores, alrededor del 63 % de los usuarios experimentaría un retraso mayor que 1s. Para demostrarlo, sea X la situación de que algún servidor tarde más de 1s, e Y la situación de que ningún servidor tarde más de 1s. Así:

$$P(X > 1s) = 1 - P(Y < 1s)$$
$$P(Y < 1s) = 0{,}99^{100} \approx 0{,}366$$
$$P(X > 1s) = 1 - 0{,}366 = 0{,}634$$

Esta variabilidad en los tiempos de respuesta de los servidores puede ser ocasionada por múltiples factores. Algunos de los más importantes que se mencionan en [3, 4] son los recursos compartidos, la gestión energética de los servidores, los procesos en segundo plano, o la concurrencia tanto a nivel software como hardware. Intentar mitigar en la totalidad todas estas fuentes es irrealizable. Sin embargo, se han desarrollado técnicas para controlar y reducir su efecto, como aislar los procesos en segundo plano, dividir las peticiones de mayor tamaño en múltiples peticiones de menor tamaño, distribuir mejor los procesos e hilos entre los núcleos, o utilizar políticas de balanceo más adecuadas.

Por último, cabe destacar que una latencia de cola alta no solo pone en peligro el rendimiento del sistema, violando el SLA [28], sino que también afecta a la experiencia de los usuarios, aumentando el nivel de frustración al no recibir las respuestas en el tiempo esperado y, como consecuencia de ello, el abandono de la aplicación [3], lo que puede dar lugar a importantes pérdidas económicas.

CAPÍTULO 3
Estado del arte

La importancia que presentan las infraestructuras para el estudio de sistemas en la nube y las aplicaciones de latencia crítica, ha conseguido que estas temáticas sean objeto de estudio de numerosos trabajos. En este capítulo se realiza una descripción del estado del arte, incluyendo las infraestructuras y cargas de prueba empleadas en la actualidad, así como las publicaciones más relacionadas con el presente trabajo. Al final del capítulo se presenta una comparativa de las aportaciones que realiza el presente trabajo con respecto a las deficiencias detectadas en el estado del arte.

3.1 Infraestructuras para el estudio de sistemas en la nube

En entornos de nube pública se suelen emplear máquinas virtuales para proporcionar aislamiento y privacidad para las aplicaciones de los huéspedes. La complejidad de estos sistemas se extiende más allá de las especificaciones del hardware, ya que incluye la pila de software que da soporte a la virtualización, el balanceo de la carga, la gestión de los recursos, la monitorización del cumplimiento del SLA, el soporte a múltiples huéspedes o inquilinos (*tenants*), etc.

Tal complejidad hace que sea difícil modelar sistemas en la nube. Debido a esto, como primer enfoque tentativo, se propusieron entornos de simulación de sistemas en la nube como CloudSim [30]. Estos entornos presentan un alto nivel de abstracción, ocultando muchos detalles del hardware, por lo que suelen usarse para proporcionar resultados orientativos y proporcionar guía para la implantación. Por ejemplo, permite estudiar el impacto en el rendimiento al variar el número de nodos del sistema o el número de núcleos en los procesadores de los nodos. En este sentido, se han realizado estudios sobre gestión de recursos en el *cloud* [31, 32, 33] empleando este entorno simulado.

Sin embargo, tal nivel de abstracción hace que estos sistemas proporcionen resultados que presentan grandes desviaciones (por ejemplo, más del 30 % [6] con solo un servidor) con respecto a entornos más realistas. Esto se debe, entre otras razones, a que estos entornos no son capaces de reproducir los efectos en las prestaciones causados por la interferencia entre máquinas virtuales cuando compiten por los principales recursos del sistema. Esta desventaja de los entornos simulados ha llevado a la creación de infraestructuras específicas para el estudio de los sistemas en la nube. Con respecto a estas infraestructuras, se pueden observar dos grandes tendencias. Una de ellas es usar infraestructuras de gran escala como Grid'500 [34], Cloudlab [35] y Chamaleon [36], las cuales permiten llevar a cabo experimentos en sistemas distribuidos geográficamente. Para usar estas infraestructuras, los usuarios reservan los recursos deseados por una cantidad limitada de tiempo y luego los configuran (por ejemplo, para evaluar un software que pretende desplegarse en sistemas en la nube).

En contraposición, se pueden utilizar infraestructuras de pequeña escala (formadas por un pequeño subconjunto de nodos) que habilitan el estudio de los sistemas en la nube sin depender de actores externos. Estas infraestructuras presentan una complejidad más reducida que las de gran escala y una mayor flexibilidad, ya que ofrecen un control estricto de la plataforma experimental y la carga de trabajo. De hecho, algunas organizaciones de nube pública (por ejemplo, Alibaba Cloud [37] y Google [38]) utilizan este tipo de infraestructuras, en las que se centra este trabajo.

El diseño e implementación de una infraestructura de pequeña escala representativa es desafiante ya que dicha infraestructura debe presentar tres características clave: i) incluir los principales tipos de nodos (servidor, cliente y almacenamiento), ii) soportar virtualización para huéspedes, iii) incorporar mecanismos de balanceo de la carga, y iv) permitir la gestión de los principales recursos compartidos, incluyendo memoria principal, caché de último nivel (LLC), componentes de red, etc. Sin embargo, la mayoría de las infraestructuras en el estado del arte presentan deficiencias tales como: i) estar compuestas por una sola máquina [39, 37, 40, 41, 42], ii) falta de soporte a la virtualización [39, 37, 40], iii) no tener en cuenta el balanceo de carga [37, 40, 41, 42, 43] y iv) descuidar la gestión de componentes importantes como la infraestructura de red [39, 37, 40] y el almacenamiento remoto [37, 40, 44, 45, 43].

3.2 *Suites* de *benchmarks* de latencia crítica

Para estudiar las aplicaciones de latencia crítica, se han diseñado numerosos conjuntos de aplicaciones de referencia o *suites* de *benchmarks*. Éstos son ejecutados en las infraestructuras informáticas obteniendo unos resultados que permiten evaluar el rendimiento. Algunas de las *suites* más utilizadas se presentan a continuación:

- **CloudSuite** [46]. Se trata de un conjunto de cargas de trabajo a gran escala populares en los centros de datos de los principales proveedores de servicios. Su propósito se enfoca al estudio de las influencias que tiene la microarquitectura de los procesadores sobre los servicios distribuidos actuales. Estas cargas se caracterizan por operar sobre una gran colección de datos divididos entre varias máquinas o nodos, los cuales reciben constantemente numerosas peticiones independientes. Están diseñadas específicamente para las infraestructuras de la nube, donde un nodo puede estar disponible en un momento y no estarlo en el siguiente, debido a factores que afectan el funcionamiento normal de la máquina. El conjunto está formado por ocho aplicaciones que abarcan áreas como sistemas de almacenamiento de datos mediante bases de datos no relacionales, procesamiento de datos a gran escala siguiendo el paradigma MapReduce [47], aplicaciones de transmisión de contenido multimedia, programas para cálculos de gran escala distribuidos y servicios web.

- **YCSB** [48]. A diferencia de la anterior, este conjunto de aplicaciones evalúa los sistemas de la nube utilizando solo un tipo de carga, relacionada con el almacenamiento y la gestión de datos. Se definen varios niveles de carga, cada uno formado por un conjunto de operaciones de lectura y escritura, con distintos tamaños de datos a leer o escribir, y diferentes patrones de acceso. Permite reproducir escenarios típicos de los centros de datos como actualización intensiva (50 % operaciones de lectura y 50 % operaciones de actualización), lectura intensiva (95 % lectura y 5 % actualización), solo lectura (100 % lectura), lectura de datos recientes (95 % lectura y 5 % inserción) y escaneo de registros (95 % escaneo y 5 % inserción). Cabe resaltar que los autores han habilitado la posibilidad de definir nuevos niveles de carga para adecuarse a las necesidades de evaluación de cada usuario.

- **BigDataBench** [49]. Este conjunto de aplicaciones tiene como propósito la evaluación de los sistemas *big data* a través de una variedad de cargas con datos reales y representativos. Consta en total de diecinueve cargas de trabajo que abarcan programas de servicios *online* como búsqueda web o comercio electrónico, aplicaciones de análisis de datos en tiempo real y aplicaciones de análisis de datos *offline*. Los dos últimos mencionados se diferencian en que el primero procesa datos nada más generarse, mientras que el segundo trata datos ya almacenados con anterioridad. Se generan diferentes métricas de prestaciones que permiten realizar una evaluación tanto desde punto de vista del usuario como desde el punto de vista del proveedor *cloud*.

- **Tailbench** [6]. Se trata de un conjunto de cargas de trabajo diseñadas para evaluar las prestaciones de los sistemas de los servicios informáticos desde el punto de vista de la latencia, y, en especial, de la latencia de cola. Las aplicaciones del conjunto tienen un rango amplio de tiempos de respuesta, desde nanosegundos a segundos. Esta formado por ocho cargas de trabajo en total, abarcando aplicaciones como la búsqueda web, el reconocimiento de imágenes y los sistemas de almacenamiento en la nube, entre otras.

Aunque CloudSuite, YCSB y BigDataBench son muy populares tanto en el ámbito comercial como en el académico, no son las más adecuadas para el presente trabajo. Esto se debe principalmente a dos motivos: por un lado, no se centran en el estudio de la latencia, clave para este trabajo; por otro lado, algunas presentan poca variedad en los tipos de cargas que utilizan, lo que no permite un estudio de un conjunto de servicios variado y representativo. Estas razones dan lugar a la elección de las aplicaciones Tailbench, que fueron diseñadas específicamente para el estudio de la latencia, y en particular, de la latencia de cola. Asimismo, incluye una variedad de aplicaciones con las que se pueden llevar a cabo pruebas realistas. Otra de las razones que motivan el uso de Tailbnech es que su configuración es considerablemente más sencilla que las otras cargas de trabajo mencionadas.

Sin embargo, como se explica en la sección 6.1.3, Tailbench presenta limitaciones para la realización de experimentos en estudios sobre políticas de balanceo de carga. En particular, limita de forma estática el número de clientes que se pueden conectar a un determinado servidor y el número de peticiones que cada servidor debe procesar. Estas limitaciones impiden realizar experimentos donde los clientes que se conectan a los servicios varían de forma dinámica y con un número de peticiones y tasa de acceso variables por cliente.

3.3 Estudios sobre la latencia de cola y el balanceo de la carga

Artículos como [3] y [4] tratan el tema de la latencia de cola desde distintos niveles o puntos de vista. El primero estudia la importancia y la dificultad de mantener una latencia baja en sistemas distribuidos a gran escala. Destaca que pequeños problemas de rendimiento pueden amplificar considerablemente los tiempos de respuesta de las peticiones. Como consecuencia de ello, la experiencia de los usuarios finales y la calidad del servicio se ven afectados significativamente. El trabajo insiste que es inviable eliminar todas variaciones que causan latencia, por lo que propone buscar soluciones similares al desarrollo de sistemas tolerantes a fallos. El segundo trabajo se centra en las latencias de cola de los nodos individuales que componen los sistemas distribuidos. Para realizar el estudio, asume que el funcionamiento de los servidores sigue un modelo de cola que consiste en tratar las peticiones en su orden de llegada, o FIFO (First In First Out). Los

resultados de los experimentos indican que los procesos en segundo plano, las políticas de balanceo del procesador, la memoria, la estructura interna de los programas y el ahorro de energía son algunas de las principales fuentes de problemas y propone algunas soluciones para contrarrestarlas.

Con respecto al balanceo de la carga en el ámbito de la computación en la nube, trabajos como [23, 50, 51, 52, 53] proporcionan una visión general del estado del arte. En [54], los autores estudian las interconexiones de la red de los centros de datos como fuente de latencia de cola. Destacan cómo diferentes patrones de tráfico de red cuando se emplea una arquitectura software basada en microservicios pueden dar lugar a un peor tiempo de respuesta. Se proponen dos métodos como solución a estos problemas. El primero se basa en optimizar las conexiones entre los conmutadores de red, mientras que el segundo radica en realizar un mejor balanceo de carga entre los servidores. El trabajo presentado en [55] persigue estudiar el impacto de las políticas de balanceo de carga, en concreto, *Round Robin* y *Least Connections*. Se evalúan los algoritmos tanto desde el punto de vista del rendimiento y utilización de los recursos como desde los tiempos de respuesta.

Cabe resaltar que ninguno los trabajos mencionados sobre balanceo de carga evalúa las latencias de cola. En muchos casos, esto se debe a que las aplicaciones o servicios evaluados no son adecuados para representar cargas sensibles a esta latencia. Asimismo, la infraestructura experimental suele ser excesivamente simple y/u opaca. Por ejemplo, algunos de los trabajos [55] se desarrollan en entornos de nube pública, lo que impide discernir si las aplicaciones de prueba se alojan todas en un mismo servidor físico o en distintos, además de dificultar el conocimiento de los detalles de la configuración hardware de los servidores.

3.4 Aportaciones del presente trabajo sobre el estado del arte

Tras analizar el estado del arte, se comprueba que existe una ausencia de infraestructuras adecuadas para el estudio de los sistemas en la nube que soporte la evaluación de la latencia de cola en políticas de balanceo de carga aplicadas a servicios de latencia crítica. Para solventar este problema, en este trabajo se presenta una infraestructura que cuenta con las siguientes características:

- Implementa los tres tipos principales de nodos presentes en los sistemas reales: nodos servidores que alojan a los huéspedes, nodos clientes que generan solicitudes y nodos de almacenamiento que proporcionan capacidades de almacenamiento remoto a los servidores.
- Soporta servicios virtuales.
- Permite la estudio de la latencia de cola en servicios de latencia crítica.
- Proporciona mecanismos para configurar políticas de balanceo de carga.
- Permite la realización de experimentos para evaluar las políticas de balanceo de carga, donde los clientes se conectan dinámicamente a la infraestructura, con un número de peticiones y tasas de acceso también variables dinámicamente.

En resumen, el presente trabajo supera las deficiencias de las infraestructuras existentes; lo que permitirá abordar, en futuros trabajos, el desarrollo de nuevas políticas de balanceo de carga que tengan en cuenta la latencia de cola y el impacto en ésta de la gestión de recursos.

CAPÍTULO 4
Diseño de la infraestructura experimental

En este capítulo se detallan el diseño de la infraestructura, las herramientas empleadas para la construcción de ésta y las características de las máquinas que la componen. En primer lugar, se presenta una visión general de toda la infraestructura, para después proceder a describir con más detalle el equipamiento hardware y la plataforma software.

4.1 Descripción general

La infraestructura informática empleada en el presente trabajo se divide en dos subsistemas: i) el equipamiento hardware, y ii) la plataforma virtual o software. El primero está compuesto por los elementos tangibles como los nodos anfitriones de las máquinas virtuales o nodos de cómputo, los nodos de almacenamiento, los conmutadores o *switches* de red, y los componentes necesarios para la asegurar la correcta interconexión del sistema. La Figura 4.1 muestra los nodos de cómputo, denominados *master*, *worker* 1 y *worker* 2, así como los nodos de almacenamientos, denominados *storage* 1, *storage* 2 y *storage* 3. Además, se muestra la interconexión de los nodos a través de conmutadores.

En cuanto a la infraestructura software, está formada por un conjunto de máquinas virtuales y el software utilizado para soportarlas, incluyendo el que se ejecuta en los nodos de almacenamiento. Cada máquina virtual tiene un rol determinado, que puede ser: i) cliente, que se encarga de realizar peticiones; o ii) servidor, que se responsabiliza del procesamiento de éstas. Los clientes se ejecutan en el nodo *master* y los servidores lo hacen en *worker* 1 y *worker* 2.

Para la creación y gestión de toda esta infraestructura se ha empleado Proxmox como sistema operativo en los nodos de cómputo anfitriones y Ubuntu en clientes y servidores virtuales. Para balancear las conexiones entre éstos, se emplea LVS (Linux Virtual Server). En los siguientes apartados se describe con más detalle cada uno de estos componentes.

4.2 Equipamiento hardware

Los nodos de cómputo *master*, *worker* 1 y *worker* 2 disponen de distinta configuración hardware, formando en conjunto de servidores heterogéneos. Estos nodos se interconectan a través de enlaces Ethernet de 10Gbps mediante el conmutador correspondiente a la red de cómputo (192.168.14.0/24, ver Figura 4.2). Por otro lado, los nodos *worker* 1 y *worker* 2 se comunican a través del conmutador de la red de almacenamiento

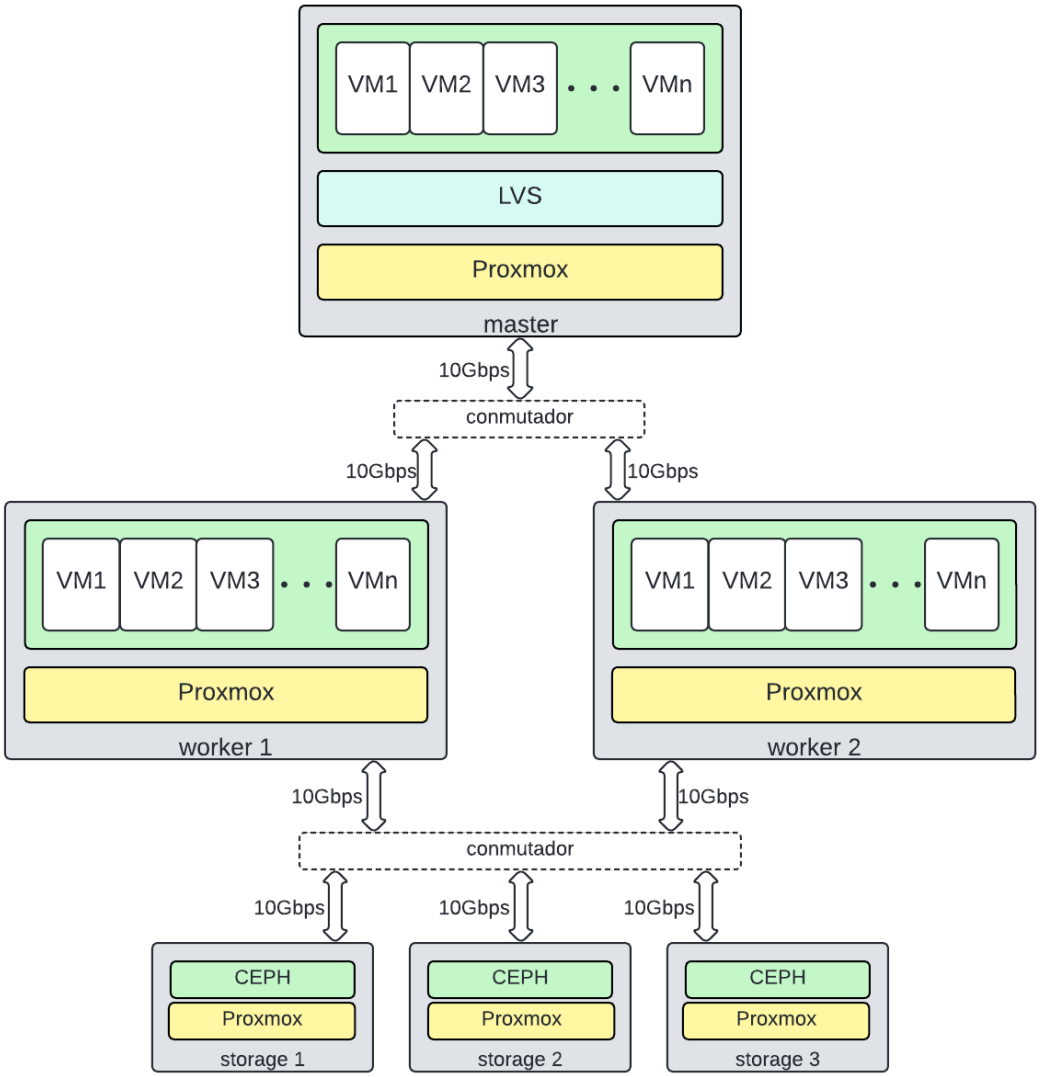


Figura 4.1: Nodos de cómputo y su interconexión en la infraestructura propuesta.

(192.168.15.0/24) con los nodos correspondientes. La razón de incluir dos redes, una de cómputo y otra de almacenamiento, no interconectadas reside en modelar la existencia de la división de este tráfico en los sistemas reales.

La Tabla 4.1 resume de forma simplificada las características más relevantes del hardware de los nodos de cómputo, las cuales se detallan seguidamente.

Máquina	Núcleos	Hilos por núcleo	F. base	F. máxima	Memoria
master	12	2	2,2GHz	2,9GHz	32GB
worker 1	8	2	2,1GHz	3,0GHz	32GB
worker 2	64	2	2,0GHz	4,0GHz	256GB

Tabla 4.1: Resumen del hardware de los nodos de cómputo.

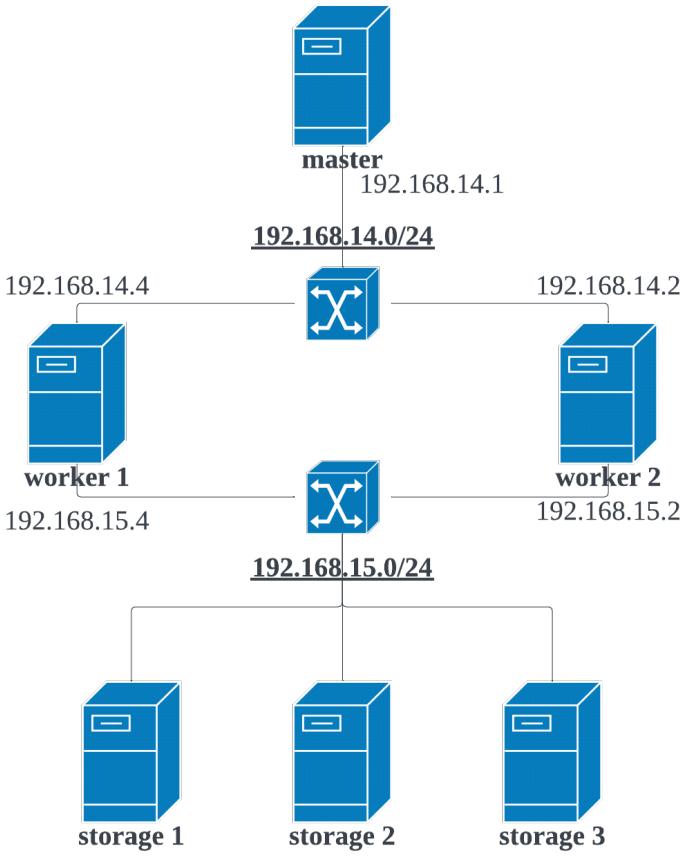


Figura 4.2: Interconexión entre los nodos.

4.2.1. Nodo *master*

Este nodo aloja a los clientes e incluye el balanceador de carga LVS que distribuye sus conexiones entre los servidores. Tiene como procesador un Intel Xeon E5-2658A v3[1], que dispone en total de 12 núcleos y 3 niveles de caché, con un tamaño de 30MB en el último nivel. Este procesador permite ejecutar hasta 2 hilos simultáneamente en cada núcleo, llegando a 24 hilos en su totalidad. La frecuencia base de funcionamiento es de 2,2GHz en situaciones de baja carga. Ésta puede ser aumentada temporalmente hasta los 2,9GHz. Sin embargo, para llevar a cabo el estudio se fijará la frecuencia al valor base. El equipo tiene a su disposición una memoria principal DRAM de 32GB a 1066,5MHz.

4.2.2. Nodo *worker 1*

A diferencia del *master*, este equipo tiene como procesador un Intel Xeon E5-2620 V4[2], el cual dispone en total de 8 núcleos y 20MB en el último nivel de caché. Al igual que el *master*, cada núcleo permite ejecutar hasta 2 hilos (16 hilos en total). Su frecuencia base es ligeramente inferior (2,1GHz), mientras que la frecuencia máxima puede alcanzar los 3GHz. En este trabajo se ha fijado la frecuencia al mismo valor que en el *master*, 2,2GHz. En cuanto a la memoria principal, tiene a su disposición dos módulos DDR4 a 1200MHz con un tamaño de 16GB cada uno (32GB en total).

[1]https://ark.intel.com/content/www/us/en/ark/products/86067/intel-xeon-processor-e5-2658a-v3-30m-cache-2-20-ghz.html

[2]https://www.intel.la/content/www/xl/es/products/sku/92986/intel-xeon-processor-e52620-v4-20m-cache-2-10-ghz/specifications.html

4.2.3. Nodo *worker 2*

Este nodo tiene instalado en sus dos zócalos dos procesadores Intel Xeon Gold 6438Y+[3], los cuales cuentan cada uno con 32 núcleos multihilo (total 128 hilos entre los 2 procesadores) y un tamaño de 60MB en la caché de último nivel. Comparado con los otros nodos, la frecuencia base de funcionamiento del procesador es la más baja (2GHz), mientras que la máxima puede llegar a los 4GHz. Al igual que los otros nodos, en este nodo se ha fijado la frecuencia a 2,2GHz. Con respecto a la memoria principal que posee este servidor, la configuración actual de la máquina ofrece un total de 256GB a 2400MHz con 16 módulos DDR5 de 16GB cada uno.

4.2.4. Switches Dlink DXS 1210-28T 24x10G Base T

La infraestructura cuenta con dos conmutadores de red Dlink DXS 1210-28T[4]. Uno de ellos se dedica a la red de cómputo mientras que el otro a la de almacenamiento. Este modelo de conmutador dispone de 24 puertos Ethernet de alto rendimiento de 10Gbps, y 4 puertos de fibra óptica adicionales de la misma velocidad. Su ancho de banda máximo es de 560Gbps. Está diseñado específicamente para redes de gran complejidad y asegurar una infraestructura de red efectiva y segura.

4.2.5. Nodos de almacenamiento

La infraestructura cuenta con tres nodos (*storage* 1, *storage* 2 y *storage* 3) de almacenamiento con un sistema de archivos distribuido CEPH [56]. Aunque no van a ser evaluados en este trabajo, se presentan brevemente algunas de las características más relevantes del hardware junto con la capacidad de los discos de este sistema. Los nodos *storage* 1 y 2 cuentan con el mismo modelo de procesador AMD EPYC 7272[5], el cual tiene 12 núcleos de 2 hilos cada uno (24 hilos en total). Este procesador funciona a una frecuencia base de 2,90GHz, y puede llegar a los 3,20GHz de frecuencia máxima. Las dos máquinas tienen a su disposición 64GB de memoria principal con 4 módulos de 16GB cada uno. Por otro lado, *storage* 3 tiene instalado un procesador Intel(R) Core(TM) i5-9400F[6] que cuenta con 6 núcleos de 1 hilo cada uno. Se frecuencia base de funcionamiento es de 2,90GHz, mientras que la frecuencia máxima es de 4,10GHz. Tiene a su disposición 32GB de memoria principal con 2 módulos de 16GB.

Dos de los nodos (*storage* 1 y *storage* 2) disponen de 3TB de memoria secundaria cada uno, mientras que el tercero implementa 3,5TB. En el momento en el que se escribe este trabajo, la capacidad utilizada para el almacenamiento distribuido es de 6TB (2TB de cada nodo).

4.3 Plataforma software

Una vez se ha descrito el hardware de la infraestructura, se introduce a continuación la plataforma software.

[3]https://www.intel.la/content/www/xl/es/products/sku/232382/intel-xeon-gold-6438y-processor-60m-cache-2-00-ghz/specifications.html

[4]https://www.dlink.com/en/products/dxs-1210-28t-10-gigabit-ethernet-smart-managed-switches

[5]https://www.amd.com/en/products/processors/server/epyc/7002-series.html

[6]https://www.intel.la/content/www/xl/es/products/sku/190883/intel-core-i59400f-processor-9m-cache-up-to-4-10-ghz/specifications.html

4.3.1. Proxmox

Como ya se ha destacado anteriormente, la virtualización constituye uno de los pilares de los servicios distribuidos actuales. Gracias a ella, la gestión de los recursos informáticos ha cambiado radicalmente. En la actualidad, existen en el mercado numerosas herramientas de virtualización como pueden ser VMware[7] o VirtualBox[8], entre otras. En particular, en este trabajo se ha optado por el uso de Proxmox [57]. El motivo de ello es que, además de ser capaz de proveer a los usuarios la virtualización, ofrece funcionalidades adicionales [58] tales como: i) una interfaz gráfica amigable que facilita la gestión y el control de los nodos de la infraestructura de forma conjunta, ii) permitir escalar la infraestructura añadiendo nuevos nodos, y iii) soportar la migración de máquinas virtuales entre nodos.

Proxmox es un sistema operativo de código abierto que permite gestionar clústeres de nodos físicos anfitriones con sus correspondientes huéspedes. Está basado en Debian GNU/Linux [59], al que incorpora un núcleo modificado. Permite la gestión tanto de máquinas virtuales como de contenedores al integrar dos tipos de tecnologías de virtualización: KVM (Kernel-based Virtual Machine)[9] y LXC (Linux Containers)[10]. La primera consiste en un módulo del núcleo de Linux que capacita al sistema para poder comportarse como un hipervisor *bare metal*. La segunda consiste en un entorno a nivel de usuario sobre el cual se pueden lanzar múltiples contenedores.

Como características adicionales, Proxmox facilita la gestión de red de las máquinas virtuales mediante redes y *switches* virtuales. Estos últimos son elementos software que tienen las mismas funcionalidades que un conmutador real. Asimismo, la plataforma soporta diferentes tipos de sistemas de almacenamiento de ficheros. En definitiva, Proxmox provee a los usuarios unas potentes herramientas y funcionalidades para la gestión tanto de máquinas físicas como virtuales.

4.3.2. Linux Virtual Server

En la actualidad, existen numerosos balanceadores de carga, como Nginx [60] o HAProxy [61], que son utilizados popularmente para la gestión del tráfico de los servicios distribuidos en los centros de cómputo. Sin embargo, para llevar a cabo el proyecto, se ha elegido el balanceador LVS (Linux Virtual Server) [62, 63]. Las razones principales de esta elección son su facilidad de configuración, su alta escalabilidad, y que al ser un balanceador que actúa en la capa de transporte (capa 4 del modelo OSI), es mucho más simple e introduce menores retardos comparado con los que trabajan en la capa de aplicación (capa 7 del OSI) [64].

LVS es una tecnología que distribuye las conexiones de los clientes a los servidores a través de la capa de transporte con el objetivo de proveer servicios de alta disponibilidad, y elasticidad [65] mediante la creación de clústeres o conjuntos de servidores de alto rendimiento [62]. Para entender el funcionamiento de LVS, es necesario conocer los elementos y componentes que intervienen en él. Los servidores que forman el conjunto pueden tomar dos tipos de roles. El primero de ellos es el director LVS. Se trata del corazón del grupo, ya que es el encargado de recibir las conexiones de los usuarios y redistribuirlas según la política de reparto que tenga configurada. Para ello tiene definido un listado con direcciones y puertos de los servicios que puede atender. El segundo rol

[7] https://www.vmware.com/
[8] https://www.virtualbox.org/
[9] https://linux-kvm.org/page/Main_Page/
[10] https://linuxcontainers.org/

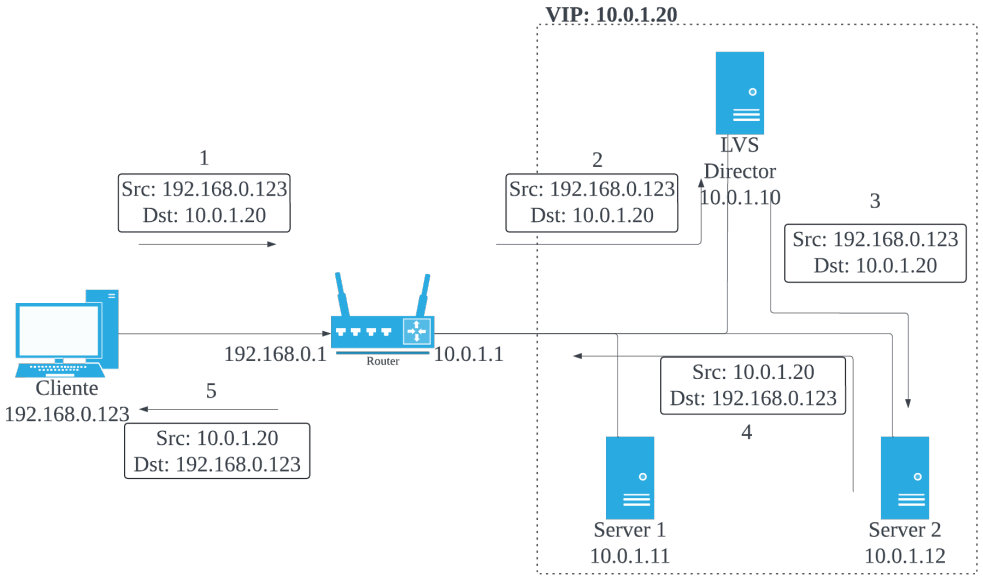


Figura 4.3: Modo de funcionamiento *Direct Routing* de LVS.

es el servidor real, las máquinas que tengan este papel asumen el procesamiento de las conexiones que reciben del primero.

Modos de enrutamiento

El proceso explicado anteriormente es transparente para los clientes, quienes solo tienen que acceder a una dirección específica. Dependiendo del modo de enrutamiento que se haya establecido, esta dirección, denominada también como dirección virtual, puede pertenecer solo al director, o es una dirección compartida por todo el grupo de servidores. LVS establece tres modos de enrutamiento [62, 65, 63]. En el presente trabajo, se ha elegido el modo ***Direct Routing*** (DR) o **Enrutamiento Directo** por su simplicidad de configuración y baja sobrecarga, ya que el director solo procesa las peticiones y no las respuestas, tal como se describe a continuación.

Bajo este modo, tanto el balanceador como los servidores deben estar dentro de una misma red local. En este contexto, todos tienen configurados en alguna interfaz la dirección virtual. En el caso de los servidores reales, esta interfaz debe ser la de *loopback*. Esta interfaz tiene un comportamiento diferente a las demás, ya que los paquetes que sean enviados por ésta tienen como destino la misma máquina. Es decir, el receptor y el emisor son la misma computadora. Las direcciones típicas de *loopback* pertenecen al rango de direcciones 127.0.0.0/8. Cuando el balanceador recibe un paquete, éste comprueba la existencia del servicio. Si existe, sustituye la dirección física (MAC) de destino de la trama por la del servidor elegido según el algoritmo de balanceo. El servidor recibe la petición y tras procesarla, responde directamente al cliente sin necesidad de la intermediación del director. El funcionamiento se puede apreciar con más detalle en la Figura 4.3, la cual representa el envío de una petición y su respuesta a través de los distintos interfaces de red.

Algoritmos de balanceo

LVS implementa distintas políticas o algoritmos de balanceo. Al funcionar en la capa 4, no puede inspeccionar el contenido de los mensajes para realizar un balanceo. Pero,

por otra parte, esto permite algoritmos de balanceo más simples y eficientes. Se destacan los siguientes algoritmos [62, 65, 66]:

- ***Round Robin*** (RR). Se trata de uno de los algoritmos más simples y conocidos. Todos los servidores son equivalentes para el balanceador, el cual reparte las conexiones de forma rotativa entre ellos.
- ***Weighted Round Robin*** (WRR). Es una versión modificada de RR donde a cada servidor se le asigna un peso que representa su capacidad. De esta manera, los servidores con mayores pesos reciben un mayor número de conexiones.
- ***Least Connections*** (LC). Se trata de un algoritmo dinámico, es decir, sus decisiones se basan en los estados actuales de los nodos. Cuando el balanceador recibe una nueva conexión, ésta es redirigida al servidor con el menor número de conexiones activas.
- ***Weighted Least Connections*** (WLC). Es una versión modificada de LC que, al igual que WRR, asigna a cada servidor un peso que representa su capacidad. Los servidores con mayores pesos reciben más conexiones, de manera que el número de conexiones activas de cada servidor es proporcional al valor seleccionado.
- ***Locality-Based Least Connections*** (LBLC). Su funcionamiento es el siguiente: cuando el balanceador recibe una nueva conexión hacia una dirección determinada, éste examina qué servidor es actualmente el encargado de recibir conexiones con dicha dirección. Si existe un servidor encargado y no está sobrecargado, es decir, su número de conexiones no supera su capacidad, entonces se le asigna la nueva conexión. Por el contrario, si no hay servidor encargado o está sobrecargado, se elige un nuevo servidor, mediante WLC, de entre los servidores que tienen una carga inferior a la mitad de su capacidad. El servidor elegido pasa a ser el nuevo encargado de la dirección destino.
- ***Locality-Based Least Connections with Replication*** (LBLCR). Es una versión ampliada de la anterior. La diferencia principal radica en que ahora es un grupo de servidores que se encarga de una dirección determinada. Cuando el balanceador recibe una nueva conexión hacia dicha dirección, aplica LC sobre el grupo. Si todos los servidores del grupo están sobrecargados, se elige un servidor de entre todos los demás que tenga el menor número de conexiones. Este servidor es añadido al grupo, reemplazando al servidor más saturado.
- ***Destination Hashing*** (DH). Su funcionamiento para seleccionar un servidor es muy simple. El servidor se elige a partir de una tabla **hash** estática indexada por la dirección destino de la conexión. Típicamente, la tabla se configura para que diversas direcciones destino se mapeen sobre el mismo servidor.
- ***Source Hashing*** (SH). A diferencia de DH, la tabla se indexa con la dirección origen de la conexión.
- ***Shortest Expected Delay*** (SED). Trata de escoger el servidor con el menor tiempo de respuesta esperado. Con este fin, cuando el balanceador recibe una conexión, calcula los retrasos esperados de cada servidor siguiendo la formula $D_i = \frac{C_i+1}{U_i}$. Siendo C_i el número de conexiones activas del i-ésimo servidor y U_i la capacidad o peso de ese servidor.
- ***Never Queue*** (NQ). La lógica de este algoritmo se basa en elegir siempre un servidor ocioso. En el caso de que ningún servidor esté ocioso, se aplica el algoritmo SED.

El presente trabajo se centra en diseñar una infraestructura para la evaluación de cargas de latencia crítica con soporte a diferentes políticas de balanceo, mientras que la evaluación de las políticas específicas se deja para trabajo futuro. Por esta razón, se ha evaluado únicamente la política de *Round Robin* para validar el funcionamiento de la infraestructura experimental y las cargas de prueba. *Round Robin* es una política simple que no necesita información adicional que distribuye uniformemente las conexiones de los clientes entre los servidores. Asimismo, es uno de los algoritmos más populares y utilizados en los entornos del *cloud computing* [67].

CAPÍTULO 5
Implementación de la infraestructura experimental

En este capítulo se detallan los pasos que se han seguido para la construcción de la infraestructura experimental presentada anteriormente. Primero se explica el proceso de instalación de una máquina virtual, y después se describen los pasos para realizar las configuraciones de red y la asignación de recursos físicos a las máquinas virtuales.

5.1 Instalación de las maquinas virtuales

Gracias a Proxmox, la instalación de las máquinas virtuales es un proceso fácil de llevar a cabo, ya que este sistema provee un entorno gráfico que permite la creación y gestión de todas las máquinas virtuales que se alojan en un nodo. La única dificultad encontrada durante el proceso es la descarga del archivo `iso` del sistema operativo Ubuntu, y su posterior localización dentro del sistema de archivos del nodo, ya que el archivo debe situarse en un ruta determinada (concretamente en `/var/lib/vz/template/iso`) para que Proxmox pueda detectarlo y utilizarlo.

Los pasos a seguir para la creación de una máquina virtual se detallan a continuación. En primer lugar, hay que descargar el archivo *iso* con la imagen de Ubuntu (concretamente la versión para servidores) para las máquinas virtuales. Este paso se realiza por línea de comandos. Para ello, nos conectamos al nodo mediante el comando `ssh`. La Figura 5.1 muestra los comandos que hay que ejecutar. La primera línea emplea el comando `wget` para descargar la imagen de Ubuntu, y la segunda mueve el archivo descargado a la carpeta mencionada para que pueda ser utilizado por Proxmox.

Para crear la máquina virtual a partir de la imagen de Ubuntu, se puede utilizar la interfaz gráfica de Proxmox. Esta interfaz es accesible a través de un navegador web en un puerto determinado del nodo que contendrá la máquina virtual. Tomando como ejemplo el *master*, la url de la interfaz es `https://192.168.14.1:8006`. Una vez dentro de la interfaz, hacemos clic derecho sobre el icono que representa al nodo, y seleccionamos la opción de creación de máquinas virtuales. Esto abre una ventana como la que se muestra en la Figura 5.2, donde se introducen los datos requeridos, como el nombre de la máquina virtual, el número de núcleos virtuales que va a disponer, la capacidad de la memoria principal, etc.

Para que la máquina virtual pueda arrancar, es necesario instalarle un sistema operativo. Esto se hace haciendo clic sobre su icono en la interfaz Proxmox (ver Figura 5.3). Esta acción abre una nueva ventana (Figura 5.4) que muestra la instalación de Ubuntu. A

```
wget https://releases.ubuntu.com/18.04/ubuntu-18.04.6-live-server-amd64.iso
mv ubuntu-18.04.6-live-server-amd64.iso /var/lib/vz/template/iso/
```

Figura 5.1: Comandos utilizados para la descarga de Ubuntu y su posterior localización dentro del sistema de archivos de Proxmox.

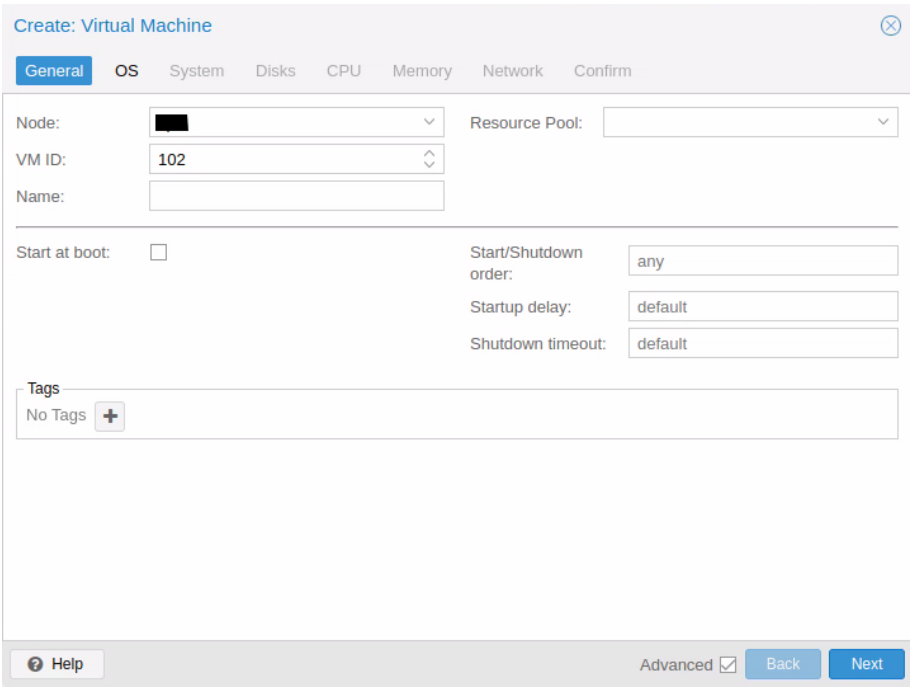


Figura 5.2: Diálogo para la creación de una máquina virtual.

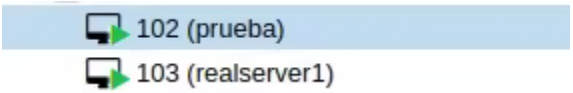


Figura 5.3: Lista de máquinas virtuales.

partir de aquí, solo es necesario seguir las instrucciones para completar la instalación. El Apéndice A muestra más imágenes de todo este proceso.

5.2 Configuración de la red

Siguiendo los pasos de la sección anterior, se han creado todas las máquinas virtuales empleadas en los escenarios evaluados en el Capítulo 7. Sin embargo, estas máquinas aún no se comunican entre ellas. En esta sección se presentan los pasos a seguir para lograr este objetivo. Para ello, será necesario configurar las interfaces de cada máquina virtual, asignándoles unas direcciones IP. Aunque estas asignaciones podrían hacerse durante la instalación del sistema operativo, el proceso es más claro utilizando herramientas específicas de configuración de red como *netplan*[1]. La Figura 5.5 muestra una plantilla de ejemplo para la configuración de los clientes y el director, mientras que la Figura 5.6 muestra otra para el caso de los servidores.

Se explica a continuación el significado de cada línea del fichero en la Figura 5.5. La primera línea (`network`) indica el comienzo de las configuración de red. La siguiente línea,

[1]https://netplan.io/

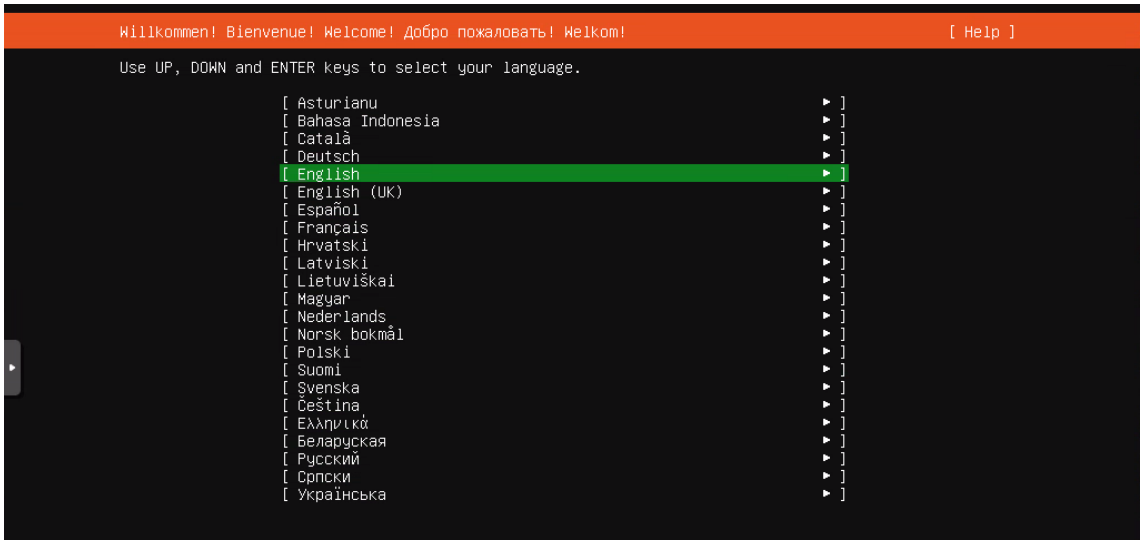


Figura 5.4: Instalación de Ubuntu.

```
network:
  <tipo de tecnologia>:
   <nombre del enlace>:
      addresses:
      - 192.168.14.x/24
      gateway4: 192.168.14.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
  version: 2
```

Figura 5.5: Plantilla del fichero de configuración de red para clientes y director.

```
network:
  <tipo de tecnologia>:
   <nombre del enlace>:
      addresses:
      - 192.168.x.x/24
      gateway4: 192.168.x.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
    nombre del enlace:
      addresses:
      - 192.168.14.x/24
  version: 2
```

Figura 5.6: Plantilla del fichero de configuración de red para servidores.

`tipo de tecnologia`, señala que las interfaces o conexiones que vienen a continuación utilizan el estándar que se indique. En este trabajo, todas utilizan el valor `ethernets`. Una vez especificado el tipo de tecnología, para configurar cada interfaz se debe concretar su nombre, que depende de la convención que use el sistema operativo. Las líneas que siguen representan las características de la interfaz. Se especifica primero las direcciones asignadas seguidas de la línea `gateway`, que describe la dirección de la puerta de enlace para poder salir al exterior de la red. Este último parámetro solo puede definirse una vez,

y en una sola interfaz si la máquina dispone de varias. Finalmente, `nameservers` establece la configuración de los servidores de nombre, incluyendo sus direcciones, que se encargan de resolver los nombres de dominio. Para conocer con más detalle los parámetros establecidos para cada máquina, en el Apéndice B se detallan las configuraciones de algunas de las máquinas virtuales creadas para evaluar la infraestructura experimental.

5.3 Configuración de LVS

En esta sección se detalla la configuración necesaria para el correcto funcionamiento del balanceador LVS en modo enrutamiento directo (ver Sección 4.3.2). En este modo, las peticiones pasan por el director pero las respuestas viajan directamente de los servidores a los clientes. Para que esto sea posible, es necesario que el director y los servidores estén en la misma red local. Además, todos los servidores deben compartir entre ellos una misma dirección virtual, que es la utilizada por los clientes para acceder a los servicios. En el caso del balanceador, esta dirección puede estar vinculada teóricamente a cualquier interfaz de red, mientras que para los servidores es necesario que esta interfaz sea la de *loopback*. Sin embargo, para simplificar la configuración, se va a asumir que el director utiliza también esta interfaz.

Para configurar la dirección virtual se debe hacer uso del comando `ip` con privilegios de *root* de Ubuntu. La siguiente línea muestra el comando a ejecutar.

```
1 sudo ip addr add 192.168.14.120/32 dev lo
```

Este comando añade la dirección 192.168.14.120 a la interfaz `lo` (*loopback*). El valor 32 tras la dirección indica que sólo se añade una dirección a la interfaz. Valores menores permiten añadir rangos de direcciones. Para ver si el comando ha surgido efecto, se puede ejecutar el siguiente comando, que muestra las direcciones que tiene asociada cada interfaz de red de la máquina.

```
1 ip addr show
```

Una vez establecida la dirección virtual en los servidores y el director, a continuación se realiza una configuración específica para cada uno de ellos. Se muestran primero los comandos que se deben ejecutar para establecer correctamente los servicios. Estos comandos especifican la dirección del servicio ofrecido por el director (dirección virtual), el puerto por el que escucha, y las direcciones y puertos de los servidores. Hay que tener en cuenta que en este modo de funcionamiento es necesario que el puerto de los servidores sea el mismo que el del director. El siguiente conjunto de comandos realiza las modificaciones mencionadas.

```
1 sudo ipvsadm -A -t 192.168.14.120:8080 -s rr
2 sudo ipvsadm -a -t 192.168.14.120:8080 -r 192.168.14.111:8080 -g
3 sudo ipvsadm -a -t 192.168.14.120:8080 -r 192.168.14.112:8080 -g
```

La primera orden establece una dirección y un puerto para un servicio que el director puede aceptar mediante el parámetro `-A -t 192.168.14.120:8080`. Además, establece la política de balanceo *Round Robin* mediante el parámetro `-s rr`. Las dos siguientes órdenes establecen las direcciones de los servidores (192.168.14.111 y 192.168.14.112) que pueden atender las peticiones del servicio especificado. Por último el parámetro `-g` especifica el modo de enrutamiento directo.

```
sudo ip addr add DIRECCION_VIRTUAL/32 dev INTERFAZ
sudo ipvsadm -A -t DIREC_VIRTUAL:PUERTO -s ALGORITMO_DISTRIBUCION
sudo ipvsadm -a -t DIREC_VIRTUAL:PUERTO -r DIREC_SERVIDOR_1:PUERTO -g
sudo ipvsadm -a -t DIREC_VIRTUAL:PUERTO -r DIREC_SERVIDOR_2:PUERTO -g
...
sudo ipvsadm -a -t DIREC_VIRTUAL:PUERTO -r DIREC_SERVIDOR_n:PUERTO -g
```

Figura 5.7: Configuración del director LVS.

```
sudo ip addr add DIRECCION_VIRTUAL/32 dev lo
sudo echo "1" > /proc/sys/net/ipv4/conf/lo/arp_ignore
sudo echo "2" > /proc/sys/net/ipv4/conf/lo/arp_announce
sudo echo "1" > /proc/sys/net/ipv4/conf/all/arp_ignore
sudo echo "2" > /proc/sys/net/ipv4/conf/all/arp_announce
```

Figura 5.8: Configuración de los servidores.

Las Figuras 5.7 y 5.8 resumen las configuraciones necesarias para el director y los servidores, respectivamente, en enrutamiento directo.

5.3.1. Filtrado del protocolo ARP

Debido a que los servidores y el director pertenecen a la misma red local y comparten una misma dirección virtual (la asignada al servicio), es posible que cuando un cliente realice una petición hacía esa dirección virtual, la petición no llegue al director, evitando que éste pueda redirigirla siguiendo el algoritmo de balanceo. Para evitar esta situación, se necesita modificar ciertos parámetros del sistema operativo en los servidores. Los siguientes comandos realizan esta configuración.

```
    sudo echo "1" > /proc/sys/net/ipv4/conf/lo/arp_ignore
    sudo echo "2" > /proc/sys/net/ipv4/conf/lo/arp_announce
    sudo echo "1" > /proc/sys/net/ipv4/conf/all/arp_ignore
    sudo echo "2" > /proc/sys/net/ipv4/conf/all/arp_announce
```

Estas órdenes modifican el comportamiento de los servidores con respecto al protocolo ARP (Address Resolution Protocol)[7], el cual permite conocer la dirección MAC de una interfaz a partir de su dirección IP. En general, una petición de cliente tiene como IP destino la dirección virtual del servicio. Mediante el protocolo ARP esta dirección virtual se traduce a una dirección MAC, que representa la interfaz encargada de atender el servicio dentro de la red local. El objetivo es conseguir que la traducción siempre lleve a la interfaz del director.

Las dos primeras órdenes afectan solo a la interfaz de *loopback*. El valor 1 establece que la interfaz solo responderá a las peticiones ARP que vengan dirigidas exclusivamente a ella y con su IP. Por otro lado, el valor 2 establece que el sistema solo responderá utilizando la dirección que sea relevante para la red a la que pertenece la interfaz. Por ejemplo, cuando se reciba una petición ARP por la interfaz de *loopback*, la máquina responderá con la MAC de la interfaz *loopback*, y no con otra MAC de otra interfaz. Las siguientes órdenes configuran el mismo comportamiento para todas las interfaces del sistema. De esta manera, la interfaz del director aparecerá en la red como la única correspondiente

a la dirección virtual ante los clientes, y por tanto el único nodo que puede aceptar las peticiones.

5.4 Configuración de los procesadores

Para evaluar el funcionamiento de la infraestructura, es necesario establecer la frecuencia de los procesadores y asignar los núcleos de cada nodo a las máquinas virtuales. Esto permite reducir las interferencias y mejorar la reproducibilidad de los resultados.

5.4.1. Establecimiento de la frecuencia de funcionamiento

Aunque las cargas de trabajo que se van a utilizar no son intensivas en el uso del procesador, se ha decidido fijar una frecuencia cercana a la base de los procesadores para no sobrecargar el sistema y minimizar la variabilidad en los resultados cuando el mismo experimento se repite. Observando las características de los procesadores de las dos máquinas físicas, todas tienen unas frecuencias base cercanas (2,2GHz en *master*, 2,1GHz en *worker 1* y 2GHz en *worker 2*). Teniendo esto en cuenta, el criterio seguido para fijar la frecuencia ha sido elegir la base máxima de las tres, 2,2GHz.

Para fijar la frecuencia se emplea el comando `cpupower`. A continuación, se muestran los comandos empleados:

```
1 sudo cpupower frequency-set -g performance
2 sudo cpupower frequency-set --min 2.2GHz --max 2.2GHz
```

La primera orden establece el administrador de energía del procesador en modo rendimiento o *performance*. Esto hace que el procesador funcione constantemente a la frecuencia máxima establecida sin importar la carga que esté soportando. La segunda orden establece la frecuencia del procesador a 2,2GHz, fijando tanto la frecuencia mínima como la máxima a este valor.

5.4.2. Asignación de los núcleos

En esta sección se explican los pasos que hay que realizar para llevar a cabo la asignación de los núcleos virtuales de las máquinas virtuales a *cores* de un procesador físico. En la infraestructura experimental, si no se restringieran los núcleos que pueden ser utilizados por cada máquina creada, podría suceder que éstas compitan por los mismos núcleos, causando interferencias que afectarían al rendimiento y la reproducibilidad de los resultados.

Hay que destacar que a nivel de sistema operativo, cada hilo (*thread*) de un núcleo físico se trata como un *núcleo lógico*. Por lo tanto, la asignación de núcleos a máquinas virtuales se realiza a nivel de núcleo lógico en vez de físico. En el caso de los procesadores utilizados por los nodos de cómputo en este trabajo, cada núcleo físico soporta 2 hilos, con lo que se desglosa en dos núcleos lógicos. En consecuencia, si no se asignaran los núcleos, dos máquinas virtuales podrían compartir el mismo núcleo físico, causando interferencia por los recursos compartidos en el núcleo (operadores, cachés L1 y L2, etc.). Para prevenir que esto suceda, se tiene que asegurar que las máquinas virtuales no compartan núcleo físico, lo cual se puede hacer imponiendo que los núcleos lógicos asignados a una máquina no pertenezcan al mismo núcleo físico de los núcleos lógicos asignados a otras máquinas.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
stepping        : 1
microcode       : 0xb000040
cpu MHz         : 2043.587
cache size      : 20480 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
```

Figura 5.9: Información de un hilo proporcionada por el sistema operativo.

Para poder llevar el proceso de asignación, se emplea el comando `taskset`. Este comando requiere los identificadores numéricos de los núcleos lógicos a asignar y los identificadores de los procesos (PID) correspondientes a las máquinas huéspedes en los anfitriones. Para averiguar el identificador de un núcleo lógico, es necesario acceder a la información del sistema mediante el siguiente comando.

```
cat /proc/cpuinfo
```

El resultado de este comando muestra un listado similar al de la Figura 5.9 para cada hilo del procesador. Los campos *processor* y *core id* del listado proporcionan, correspondientemente, información sobre la identificación y a qué núcleo físico pertenece cada hilo. Por otra parte, el campo *physical id* indica el zócalo de la placa donde está instalado el procesador al que pertenece el hilo. En cuanto a la numeración de los hilos, esta sigue un patrón secuencial que empieza desde 0 hasta el número total de hilos menos uno.

Como alternativa a examinar el valor del campo *core id* para conocer el núcleo físico al que pertenece un hilo, se ha detectado que existe un patrón circular en la asignación a los núcleos físicos. Por ejemplo, en la máquina *master*, que tiene 12 núcleos (24 hilos, 2 cada núcleo), el hilo 0 y el hilo 12 son del núcleo 0, el hilo 1 y el hilo 13 del núcleo 1, y así sucesivamente.

Con respecto a cómo averiguar el PID de los procesos correspondientes a cada máquina virtual, el siguiente comando permite listar este identificador para todas las máquinas virtuales activas en Proxmox.

```
ps aux | grep kvm
```

Una vez identificados los hilos y el PID de las máquinas virtuales, ya es posible realizar la asignación. A continuación, se presenta un ejemplo donde se asignan los hilos de una máquina virtual (PID 11111) a los hilos 0,1,2 (`0-2`) y 12,13,14 (`12-14`), pertenecientes a 3 núcleos físicos (0, 1 y 2).

```
sudo taskset -pc 0-2,12-14 11111
```

CAPÍTULO 6
Tailbench++

En este capítulo se describen la *suite* de *benchmarks* Tailbench y se explican las limitaciones que han motivado el desarrollo de Tailbench++, una versión mejorada de Tailbench adaptada a los requisitos de evaluación de algoritmos de balanceo de la carga. Posteriormente se describen las mejoras implementadas en Tailbench++. Finalmente, se comprueba el correcto funcionamiento de las modificaciones realizadas comparando los resultados de las aplicaciones antes y después de realizarlas.

6.1 La *suite* Tailbench

En esta sección se describe la *suite* Tailbench, comenzando por sus aplicaciones. Seguidamente se introducen las métricas de prestaciones que proporciona para medir la latencia o tiempo de respuesta. Finalmente, se discuten las principales limitaciones que motivan las modificaciones realizadas en este trabajo.

6.1.1. Aplicaciones

Tailbench incluye un conjunto de aplicaciones representativas de latencia crítica, las cuales se explican a continuación.

- `Xapian`. Se trata de un buscador escrito en lenguaje C++. Es utilizado por sitios web como Debian Wiki, y marcos de trabajo software como Catalyst. El índice de búsqueda empleado se construye a partir de la versión inglesa de Wikipedia de julio de 2013. Realiza las consultas eligiendo aleatoriamente los términos siguiendo una distribución Zipfian.
- `Masstree`. Se trata de un almacén de claves-valores o *key-values store* escalable, rápido y en memoria principal escrito en C++. Este tipo de almacén gestiona una gran cantidad de datos que son divididos y repartidos entre numerosos servidores. La petición de un cliente puede implicar varias decenas incluso centenas de solicitudes al almacén.
- `Moses`. Consiste en un sistema de traducción automática estadística escrito en C++. Estos sistemas son la base de servicios de traducción como los de Google. Está configurado para seleccionar aleatoriamente fragmentos de diálogo del corpus inglés-español de `opensubtitles.org`.
- `Sphinx`. Es un sistema de reconocimiento de voz escrito en C++. Estos sistemas son un componente importante en aplicaciones como Apple Siri o IBM Speech to Text. Se ha adaptado para elegir al azar enunciados de la base de datos CMU AN4.

- `Img-dnn`. Consiste en una aplicación de reconocimiento de escritura a mano que se basa en OpenCV. Se trata de un tipo de reconocimiento de imágenes y está configurado en este caso para elegir al azar muestras de la base de datos MNIST.
- `Specjbb`. Es una evaluación de prueba comparativa del *middleware* de Java, que es utilizado ampliamente por las empresas, y que debe satisfacer unos requisitos estrictos de latencia. Simula un sistema de una empresa mayorista con una infraestructura de nivel 3 que gestiona diferentes tipos de peticiones de los clientes.
- `Silo`. Se trata de una base de datos transaccional en memoria, utilizado popularmente en sistemas de procesamiento de transacciones en línea, u OLTP. Emplea TPC-C, un estándar de referencia OLTP de la industria para su funcionamiento.
- `Shore`. Al igual que Silo, se trata de una base de datos transaccional, pero en disco. Difiere significativamente en cómo guarda y accede a los datos. Emplea también TPC-C como base de la aplicación.

Hay que destacar que el conjunto de aplicaciones Tailbench tiene dos versiones: *networked* e *integrated*. En la primera versión, las aplicaciones se dividen en dos módulos, el cliente que envía las peticiones, y el servidor que las procesa. Éstos dos pueden ejecutarse en máquinas distintas, y comunicarse a través de la red, permitiendo considerar tanto la latencia de la red como la del servicio. Aunque esta versión permite la ejecución simultánea de los dos módulos en una misma computadora, Tailbench ofrece la versión *integrated* que simplifica el funcionamiento para este caso. La versión *integrated* unifica los dos módulos en un único programa, centrándose solo en el tiempo de servicio.

6.1.2. Métricas

Como resultado de los experimentos, Tailbench proporciona las siguientes métricas de interés para la evaluación de los sistemas. Todas se basan en el tiempo de extremo a extremo, o *sorjourn time*, que incluye, el tiempo de envío de la petición por la red, el tiempo que la petición está en la cola de espera del servidor, el tiempo de procesamiento de ésta, y el tiempo de devolución de la respuesta.

- **Media**. Se trata del valor medio de los tiempos de respuesta (latencia) de las peticiones de los clientes.
- **Percentil 95**. Representa el valor de latencia que deja por debajo el 95 % de los tiempos de respuesta. Es decir, el 5 % de las respuestas tienen una latencia por encima de este valor.
- **Percentil 99**. De forma similar a la métrica anterior, representa el valor de latencia solo superado en el 1 % de las peticiones.

6.1.3. Limitaciones

Aunque en este trabajo se considera que la *suite* Tailbench es la más adecuada para estudiar la latencia de cola, presenta importantes limitaciones en lo que respecta al estudio de políticas de balanceo de la carga. La mayor parte de estas limitaciones tienen que ver con la rigidez que impone el módulo servidor para configurar los experimentos. Se han detectado las siguientes:

1. Antes de que el módulo servidor pueda iniciar el proceso de peticiones, debe esperar a que un número determinado de clientes, establecido de antemano, se conecte.

2. Una vez que el servidor comienza a procesar solicitudes, no permite aceptar conexiones de nuevos clientes.
3. Si todos los clientes se desconectan, el servidor termina su ejecución.
4. El número de peticiones procesadas por el módulo servidor se determina de antemano.
5. Una vez que el servidor procesa el número de peticiones determinado, el proceso servidor finaliza y el experimento termina.

Estas limitaciones impiden la evaluación de algoritmos de balanceo de carga ya que estos experimentos requieren de varios servidores que se repartan las conexiones y peticiones de los clientes dinámicamente, por lo que no es posible determinar estos parámetros de antemano. Además de esto, un escenario representativo de los servicios informáticos de hoy en día requiere que el número de clientes y sus tasas de envío de peticiones puedan variar con el tiempo, llegando a ser cero en algunos momentos. En este trabajo se propone una modificación de Tailbench que provee estas características.

6.2 Tailbench++

En este trabajo se presenta Tailbench++, un conjunto de mejoras sobre Tailbench que le permite ser utilizada en estudios de evaluación de algoritmos de balanceo de la carga. Las modificaciones principales se han realizado en el núcleo del código compartido por las aplicaciones de Tailbench, denominado *harness*, y afectan solo a la versión *networked*. Estas modificaciones se describen a continuación.

- El cambio más relevante ha sido adaptar el código del servidor para que pueda aceptar las conexiones de un número indeterminado de clientes mientras que en paralelo procesa peticiones de conexiones ya aceptadas.
- La modificación previa también supone que el servidor no termina cuando todos los clientes se desconectan, sino que queda a la espera de nuevas conexiones.
- Se ha modificado el código del servidor y cliente para que sea este último quien determine el número de peticiones a procesar. En el código original este parámetro lo definía el servidor. Esto permite modelar que cada cliente *decida* cuántas peticiones realiza, lo cual es más realista que el servidor defina el límite de peticiones procesadas globalmente, tal como ocurría en el código original.
- Por último, se ha implementado la funcionalidad de poder variar la tasa de envío de peticiones de cada cliente dinámicamente durante su ejecución. En el código original esa tasa era fija para cada cliente. Esto permite modelar variaciones en la carga soportada por los servidores.

6.2.1. Modificaciones en el servidor

Durante un experimento para evaluar el balanceo de la carga, los servidores han de ser capaces de tratar con un número no determinado de clientes, que pueden ser asignados a éstos en distintos momentos por el balanceador de carga. Sin embargo, en el código original, el servidor no comienza a procesar peticiones hasta que el número de clientes conectados alcanza un valor especificado. Una vez que el servidor empieza a procesar peticiones, ya no acepta nuevas conexiones. Por último, cuando las todas las conexiones

```
size_t NetworkedServer::recvReq(int id, void **data) {
....
        if (clientFds.size() == 0) {
            // SERVER NEW CONNECTIONS //
            int maxFd = listenFd;
            fd_set readSet;
            FD_ZERO(&readSet);
            FD_SET(listenFd, &readSet);
            int ret = select(maxFd + 1, &readSet, nullptr, nullptr, nullptr);
            if (ret == -1) {
                std::cerr << "select() failed: " << strerror(errno) << std::
                    endl;
                pthread_mutex_unlock(&recvLock);
                exit(-1);
            }
            if (checkNewClient(&readSet)) {
                recvClientHead = 0;
            }
            continue;
        }
        int maxFd = -1;
        fd_set readSet;
        FD_ZERO(&readSet);
        // SERVER NEW CONNECTIONS //
        //add listening socket to the set
        FD_SET(listenFd, &readSet);
        maxFd = listenFd;

        for (int f : clientFds) {
            FD_SET(f, &readSet);
            if (f > maxFd)
                maxFd = f;
        }

        int ret = select(maxFd + 1, &readSet, nullptr, nullptr, nullptr);
        if (ret == -1) {
            std::cerr << "select() failed: " << strerror(errno) << std::endl;
            exit(-1);
        }
        //if new client is detected, actualiza fd_Set
        if (checkNewClient(&readSet)) {
            FD_ZERO(&readSet);
            maxFd = -1;
            for (int f : clientFds) {
                FD_SET(f, &readSet);
                if (f > maxFd)
                    maxFd = f;
            }
            ret = select(maxFd + 1, &readSet, nullptr, nullptr, nullptr);
            if (ret == -1) {
                std::cerr << "select() failed: " << strerror(errno) << std::
                    endl;
                pthread_mutex_unlock(&recvLock);
                exit(-1);
            }
        }
....
}
```

Figura 6.1: Fragmento de código de la función `recvReq`.

finalizan el servidor termina su ejecución. Este comportamiento no permite el dinamismo requerido cuando se estudian algoritmos de balanceo y tampoco refleja una situación real donde los servidores procesan peticiones al mismo tiempo que aceptan nuevas conexiones. El fragmento de código en la Figura 6.1 muestra las modificaciones realizadas sobre la función `recvReq`, que originalmente sólo se encargaba de recibir las peticiones de los clientes.

Esta función consiste en un bucle infinito que no finaliza hasta que se recibe correctamente un mensaje de algún cliente. Con las modificaciones, este método, además de mantener esta funcionalidad, permite comprobar si existen nuevas conexiones, distinguiendo dos casos: i) cuando no hay clientes conectados (líneas 3-19), y ii) cuando sí los hay (líneas 20-54). En ambos casos, se sigue una lógica similar: se comprueba si el descriptor `listenFd`, que atiende nuevas conexiones se ha activado o no. Esto se consigue mediante la función `select` que monitoriza el estado de cada descriptor del conjunto `readSet` (variable que contiene también los descriptores de las conexiones a los clientes existentes). Después, mediante la función `checkNewClient`, que comprueba si hay una nueva conexión entrante, se añade el descriptor de la conexión con el nuevo cliente a la lista de descriptores de conexiones de clientes activas `clientFds`.

En el código original el servidor finalizaba si todas las conexiones de los clientes se cerraban. En la versión modificada, el servidor no finaliza y queda esperando nuevas conexiones. El Apéndice C muestra al completo las funciones `recvReq` y `checkNewClient`.

6.2.2. Modificaciones en el cliente

En los sistemas informáticos de hoy en día, es el cliente quien decide normalmente cuándo accede a los servicios y cuántas peticiones realiza. Por lo tanto, el comportamiento que presenta Tailbench no refleja la realidad de los servicios distribuidos. En el código original, el servidor establece cuantas peticiones se procesan, mientras que los clientes se limitan a emitir peticiones sin límite. Una vez que se alcanza el número de peticiones establecido por el servidor, este finaliza.

Para corregir este comportamiento, se han realizado cambios en el código del cliente con el fin de dotarle de capacidad para establecer el número de peticiones enviadas. Estos cambios implican modificaciones en el tratamiento de los mensajes de control entre

```
1  void Client::finiReq(Response *resp) {
2  ....
3      // CLIENT CONTROLS QUERIES //
4      // if the client completes the number of requests, it finishes
5      ++numReqsCompleted;
6      if (numReqsCompleted == warmupreqs)
7      {
8          _startRoi();
9      }
10     else if (numReqsCompleted >= (warmupreqs + maxreqs))
11     {
12         dumpStats();
13         fprintf(stderr, "--> Client archiving total requests FINISH\n");
14         pthread_mutex_unlock(&lock);
15         syscall(SYS_exit_group, 0);
16     }
17 ....
18 }
```

Figura 6.2: Fragmento de código de la función `finiReq`.

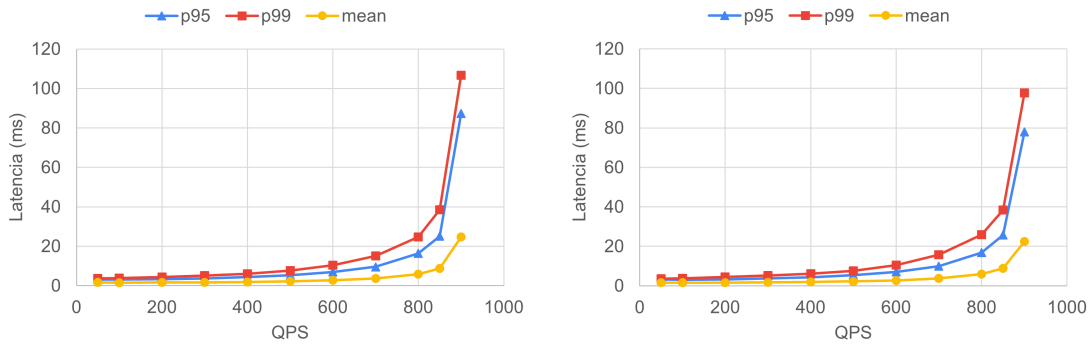
clientes y servidores. Asimismo, se han realizado modificaciones menores en el servidor que restringen su control de la finalización.

La Figura 6.2 muestra el fragmento de código de la función *finiReq* en el cliente que procesa las respuestas que recibe del servidor. Este fragmento comprueba si el número de peticiones completadas (`numReqsCompleted`), ha alcanzado o no el número total de peticiones establecidas por las variables `warmupreqs` y `maxreqs`. La primera indica el número de peticiones de calentamiento (*warmup*) y la segunda el número de peticiones máximas, y se instancian al principio de la ejecución. Durante el calentamiento, no se realizan mediciones sobre los tiempos de respuesta. Cuando el número de peticiones completadas es igual al número de peticiones de calentamiento (lineas 6-10), el cliente pasa al estado de medición de tiempos mediante la función `_startRoi`. Cuando la variable que lleva la cuenta de la peticiones realizadas supera la suma de `warmupreqs` y `maxreqs`, el cliente guarda los resultados mediante la función `dumpStats` en un fichero, y finaliza con la llamada al sistema, `syscall(SYS_exit_group, 0)`, que termina el proceso.

Además de incluir en el cliente la decisión de cuántas peticiones se envían, se ha añadido la posibilidad de variar la tasa de peticiones enviadas por segundo (*Queries per second* o QPS). Para ello, se han definido seis parámetros que deben ser pasados inicialmente al cliente, de los cuales cinco (QPS1...QPS5) son para indicar un valor de QPS y uno para indicar un intervalo que establece cuantas peticiones deben enviarse a la tasa QPS1. El cliente comienza enviando peticiones con tasa QPS1. Cuando completa el intervalo pasa a la tasa QPS2, y así sucesivamente. Para conseguir que todas las tasas sean mantenidas por el cliente durante una misma cantidad de tiempo, el intervalo se ajusta proporcionalmente. Por ejemplo, supongamos que tenemos dos tasas, 100 y 200 peticiones por segundo, y un intervalo de 1000 peticiones. En este caso, el cliente realizará las primeras 1000 peticiones de medición a 100 QPS. Después pasará a 200 QPS cambiando

```
1  ....
2  if (qpsCounter == qpsInterval)
3  {
4      qpsCounter = 0;
5
6      if(qpsVar == 1) {
7          qpsIni += qpsStep; // 10 ---- 100
8      } else if(qpsVar == 2) {
9          int prevqps = listQPS[idxListQPS%listQPS.size()];
10         int newqps = listQPS[(idxListQPS+1)%listQPS.size()];
11         double ratio = (double)newqps/prevqps;
12         qpsInterval = static_cast<int>(qpsInterval*ratio);
13
14         qpsIni = newqps*1e-9;
15         idxListQPS++;
16     }
17
18     dist = new ExpDist(qpsIni, seed, getCurNs());
19     fprintf(stderr, "\nQPS changed to %f\n", qpsIni / 1e-9);
20     fprintf(stderr, "\nQPS interval changed to %d\n", qpsInterval);
21 }
22 else
23 {
24     qpsCounter++;
25 }
26 ....
```

Figura 6.3: Fragmento de código de la función `startReq`.



(a) Xapian **(b)** Xapian++

Figura 6.4: Curvas de crecimiento de latencia de la aplicación `Xapian`.

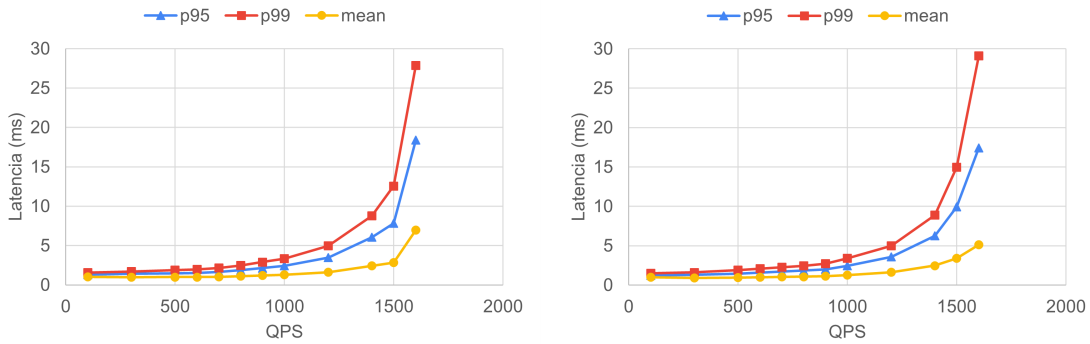
el intervalo a 2000 peticiones, lo que implica que el tiempo con la tasa QPS2 será el mismo que se tomó con la tasa QPS1.

Nótese que durante las peticiones de calentamiento, el cliente no varía su nivel de carga. Los cambios solo se dan cuando comienza la medición. Cuando el cliente finaliza con la tasa última tasa, si aún no se ha alcanzado el total de peticiones, se reinicia a la tasa QPS1.

La Figura 6.3 muestra el fragmento de código de la función `startReq` del cliente que implementa el comportamiento explicado. La variable `qpsCounter` lleva la cuenta de cuantas peticiones se han enviado hasta el momento. Cuando el valor de esta variable es igual al valor del intervalo indicado por la variable `qpsInterval`, se procede con el cambio de tasa. Si la variable `qpsVar`, recibida al principio del proceso, toma el valor 2, se ejecutará la lógica de la funcionalidad añadida en este trabajo. En caso de que tome el valor 1, actuará según la lógica anterior. La variable `idxListQPS` juega un papel fundamental en la lógica de variación de QPS, porque es utilizada para acceder a la lista de tasas. Las valores de las variables `prevqps` y `newqps`, tasa actual y siguiente respectivamente, dependen de ella. Con estas dos últimas variables, se calcula el valor de la variable `ratio` utilizada para asegurar que los intervalos duran el mismo tiempo. Una vez realizados todos los cálculos, se actualiza la tasa en la linea 18.

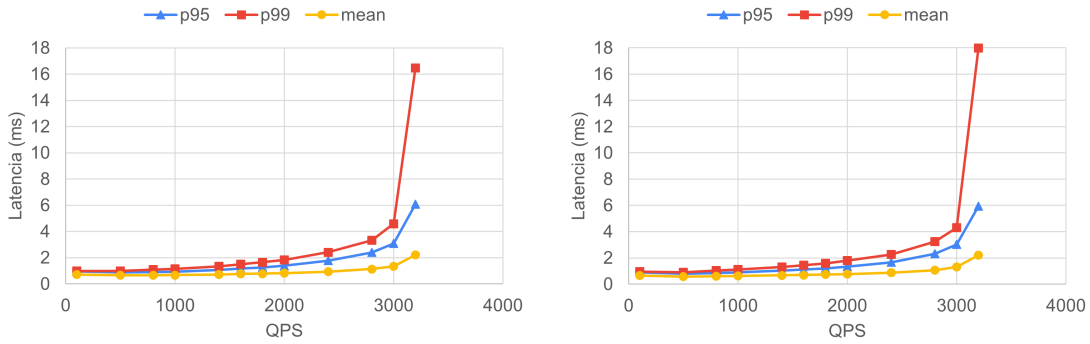
6.3 Comprobación del funcionamiento de Tailbench++

Explicadas las modificaciones introducidas por Tailbench++, en esta sección se comprueba el comportamiento de cada aplicación. En las figuras siguientes, se detalla el crecimiento de las latencias (eje Y) en función de los QPS del cliente (eje X). Por cada aplicación, se muestran dos figuras, una corresponde a Tailbench y la otra a Tailbench++. El objetivo es comprobar que las modificaciones introducidas en Tailbench++ no afectan al comportamiento de las aplicaciones en términos de latencia. En ambos casos, se han obtenido los datos con un solo cliente conectándose a un servidor y realizando 10000 peticiones de calentamiento y 30000 de medición. Se tratan de valores adecuados para calentar el sistema y obtener una muestra suficiente de mediciones. Todas las gráficas muestran las tres métricas proporcionadas por Tailbench: media, percentil 95 y percentil 99. En todas las ocasiones, la primera métrica crece en menor medida comparada con las dos posteriores. Los resultados se han obtenido con las máquinas virtuales cliente 1 y servidor 1 descritas en el apartado 7.1.



(a) Img-dnn **(b)** Img-dnn++

Figura 6.5: Curvas de crecimiento de latencia de la aplicación `Img-dnn`.



(a) Masstree **(b)** Masstree++

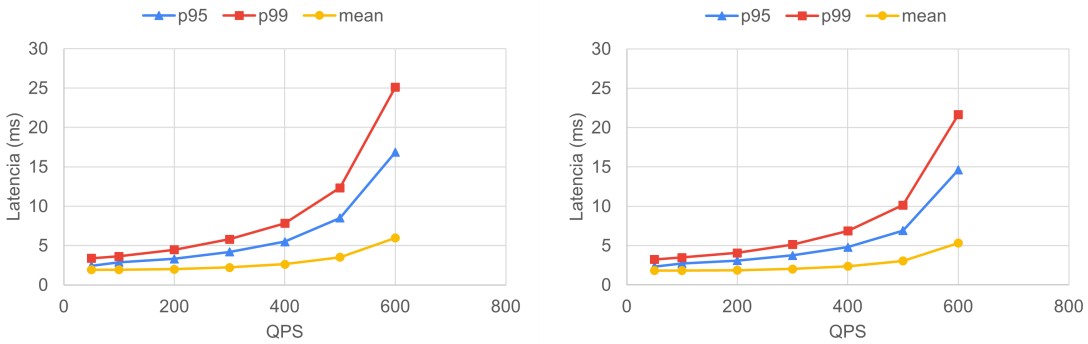
Figura 6.6: Curvas de crecimiento de latencia de la aplicación `Masstree`.

6.3.1. Xapian

En la Figura 6.4 se muestran las dos gráficas de crecimiento de latencia de la aplicación `Xapian`. La imagen de la izquierda es la versión original, mientras que la de la derecha es la modificada. Se observa en ambos casos que los tiempos de respuestas apenas crecen con la carga inicialmente. Sin embargo, a partir de las 600 peticiones por segundo, el comportamiento cambia súbitamente. Y a partir de las 800 QPS, el crecimiento es exponencial, la latencia de cola pasa de aproximadamente 20ms a 100ms con 900 QPS en el percentil 99, siendo en menor medida este aumento con el percentil 95, y mucho menor en el caso de la media.

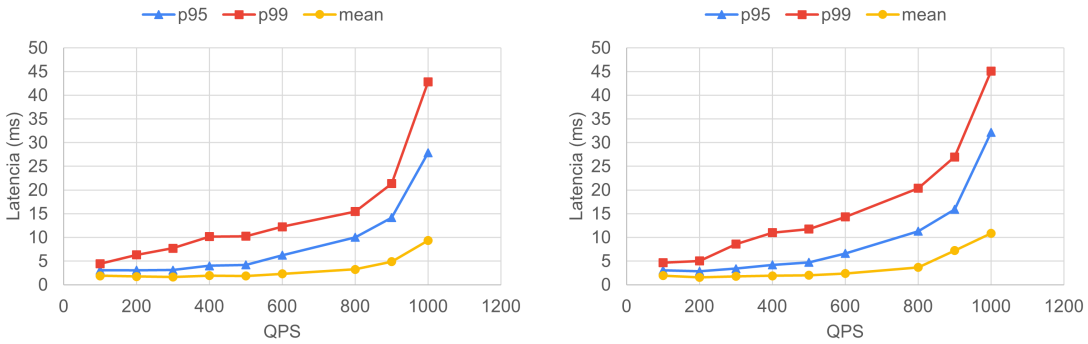
6.3.2. Img-dnn

Las curvas de latencias de `Img-dnn` se detallan en la Figura 6.5. Ambas figuras siguen un comportamiento similar, con un crecimiento limitado y estable al principio. Sin embargo, a partir de las 1000 peticiones por segundo se aprecia un aumento considerable de la latencia, que se eleva drásticamente a medida que aumentan los QPS. Las latencias pasan de 5ms con 1000 QPS, a más de 25ms en el caso del percentil 99 con 1600 peticiones por segundo, lo que indica que el sistema está saturado.



(a) Moses **(b)** Moses++

Figura 6.7: Curvas de crecimiento de latencia de la aplicación `Moses`.



(a) Shore **(b)** Shore++

Figura 6.8: Curvas de crecimiento de latencia de la aplicación `Shore`.

6.3.3. Masstree

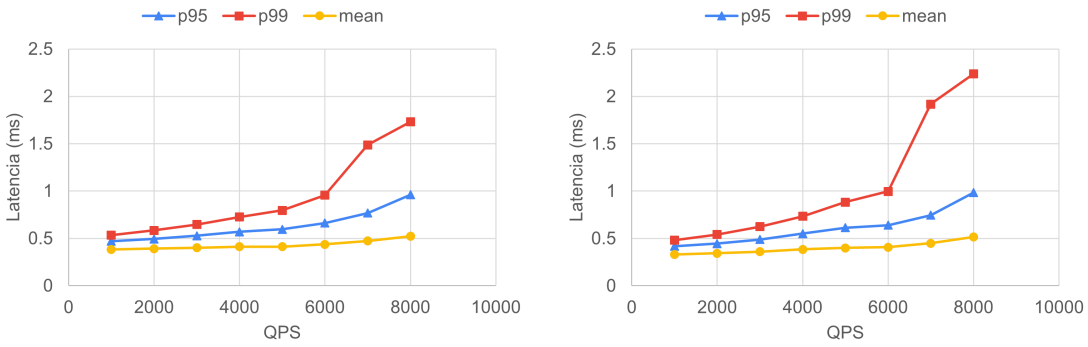
Con respecto a la aplicación de almacenamiento clave-valor `Masstree`, su comportamiento se aprecia en la Figura 6.6. A diferencia de las aplicaciones anteriores, su crecimiento exponencial se aprecia con más peticiones por segundo. Como es habitual, la latencia es baja y constante al inicio. Sin embargo, a partir de los 2000 QPS, el crecimiento es más pronunciado, incrementándose desde aproximadamente 2ms, en ese punto, hasta más de 16ms en el caso del percentil 99, y 6ms en el caso del percentil 95.

6.3.4. Moses

La Figura 6.7 muestra el crecimiento de la aplicación de traducción `Moses`. Su crecimiento es evidente desde el inicio. Aunque al principio los aumentos son moderados, a medida que las peticiones por segundo aumentan, las latencias experimentan un crecimiento exponencial. En el caso del percentil 99, los tiempos de respuestas crecen desde aproximadamente 5ms con 50 QPS hasta llegar a más de 20ms con 600 QPS. Por otra parte, el percentil 95 y la media muestran aumentos en menor medida, pasando de alrededor de 3ms a aproximadamente 15ms, y 5ms respectivamente.

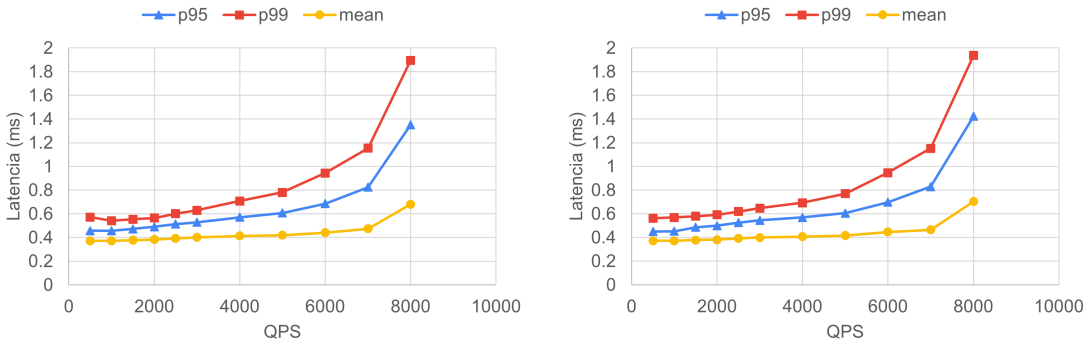
6.3.5. Shore

Las curvas de latencia de la aplicación `Shore` se muestran en la Figura 6.8. La versión original y la modificada tienen el mismo comportamiento. A diferencia del percentil 99,



(a) Silo **(b)** Silo++

Figura 6.9: Curvas de crecimiento de latencia de la aplicación `Silo`.



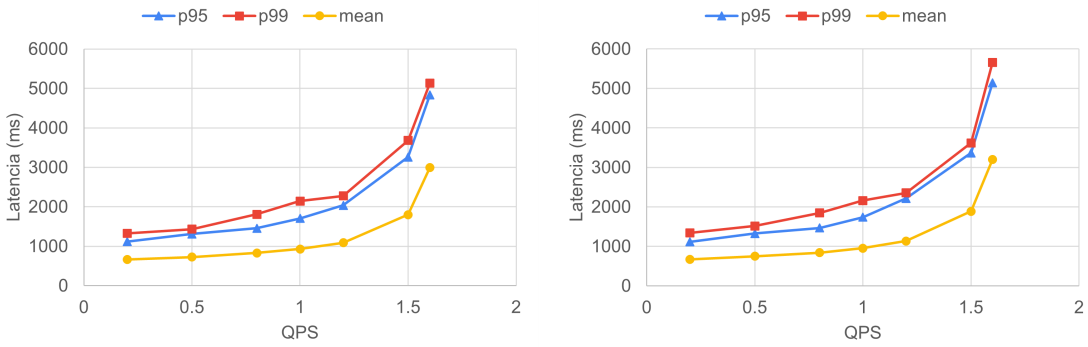
(a) Specjbb **(b)** Specjbb++

Figura 6.10: Curvas de crecimiento de latencia de la aplicación `Specjbb`.

que desde el inicio tiene unos valores de latencia pronunciados, el percentil 95 y la media mantienen un crecimiento similar y constante inicialmente. Esta situación cambia cuando el número de peticiones por segundo alcanza los 600 QPS. A partir de ese momento, el percentil 95 comienza a crecer drásticamente junto al percentil 99. En cambio, la media sigue manteniendo unos valores constantes. Después de las 800 peticiones por segundo, el crecimiento de esta aplicación se vuelve exponencial, pasando de aproximadamente de 20ms a más de 40ms en el percentil 99, de 15ms a más de 25ms en el percentil 95, y de 5ms a alrededor de 10ms.

6.3.6. Silo

El comportamiento de la aplicación `Silo` es bastante diferente a los demás. Se puede observar en las gráficas de la Figura 6.9 que el eje vertical toma valores mucho más grandes. Sin embargo, el crecimiento de las latencias es mucho menor comparado con las otras aplicaciones. Pero aún así, se aprecia un crecimiento exponencial cuando se alcanzan las 6000 peticiones por segundo. A partir de ese punto, se pasa de 1ms a aproximadamente 2ms con 8000 QPS en el caso del percentil 99. No obstante, este aumento es mucho menor para el percentil 95, y la media casi no varía. Esto demuestra que no es adecuado utilizar la media exclusivamente para medir las prestaciones de las cargas de latencia crítica.



(a) Sphinx **(b)** Sphinx++

Figura 6.11: Curvas de crecimiento de latencia de la aplicación `Sphinx`.

6.3.7. Specjbb

Como se aprecia en la Figura 6.10, `Specjbb` se mueve en rangos de latencia similares a los de *Silo*. El comportamiento de esta carga apenas varía cuando el número de peticiones por segundo es menor a 4000. Cuando se supera este límite, se observa que las curvas de cada métrica tienden a distanciarse entre ellas. A partir de los 6000 QPS, el percentil 99 y el percentil 95 toman un crecimiento exponencial. En cambio, tal como ocurre con *Silo*, para la media, los valores no varían considerablemente.

6.3.8. Sphinx

En cuanto a la aplicación `Sphinx`, se ha modificado el número de peticiones a realizar, ya que, por defecto, tiene unos tiempos de procesamiento bastante altos. Por lo tanto, con un nivel de peticiones por segundo relativamente bajo en comparación a otras aplicaciones, el sistema se satura con facilidad. Por este motivo, se ha rebajado tanto los QPS como el número de peticiones a realizar por el cliente, estableciéndose como 10 y 100 los valores de calentamiento y medición para estos últimos parámetros. En las gráficas (Figura 6.11) se aprecia que desde el inicio, con una tasa de 0,2 peticiones por segundo, la latencia observada es de aproximadamente 1s. A pesar de que este valor es significativamente mayor en comparación con los resultados de las demás aplicaciones, es el menor tiempo de respuesta que hemos obtenido para `sphinx`. A medida que se incrementa la tasa de peticiones por segundo, los valores de latencia también aumentan. Por ejemplo, cuando se alcanza una tasa de 1 QPS, los percentiles 95 y 99 toman valores cercanos a los 2s. El crecimiento de la latencia a partir de este punto se vuelve significativamente más pronunciado. Con solo con aumentar la tasa a 1,6 peticiones por segundo, ambos percentiles pasan superar los 5s y la media se eleva a los 3s.

6.3.9. Discusión

Vistas todas las curvas de crecimiento de latencia, se puede apreciar que existe un amplio rango de sensibilidades al número de peticiones por segundo. La latencia de algunas aplicaciones crece significativamente desde el inicio con tasas de peticiones por segundo pequeñas, mientras que en otras aplicaciones este crecimiento no se aprecia hasta un nivel determinado de QPS. Asimismo, las curvas ofrecidas por Tailbench++ no presentan comportamientos notablemente distintos a las de Tailbench. Por esta razón, se puede afirmar que los cambios realizados no han afectado de forma relevante a las cualidades y propiedades de las aplicaciones, manteniendo su representatividad.

CAPÍTULO 7
Evaluación

En este capítulo se realiza una evaluación de las nuevas funcionalidades introducidas por Tailbench++ y la infraestructura experimental propuesta. Primero se presenta el entorno experimental. Seguidamente se comprueba el correcto funcionamiento de las modificaciones realizadas sobre el código de Tailbench. En tercer lugar se evalúan las prestaciones del sistema con y sin balanceador. Finalmente, se exploran las posibilidades que ofrece la infraestructura para evaluar nuevas políticas de balanceo alternativas a las ya existentes.

7.1 Entorno experimental

En esta sección se presentan las características de las máquinas virtuales utilizadas para la evaluación de la infraestructura, así como las localizaciones de cada una de ellas en los nodos de cómputo y los recursos asignados.

7.1.1. Configuración de las máquinas virtuales

Se ha decidido emplear un sistema compuesto por cinco máquinas virtuales: tres funcionarán como clientes y dos como servidores. La Tabla 7.1 detalla las características de cada máquina virtual: número de núcleos físico y lógicos, cantidad de memoria, direcciones de red y localización. Los clientes se alojan, como ya se ha mencionado en el apartado 4.2.1, en el nodo *master*. Las máquinas virtuales que funcionan como servidores se alojan en el nodo *worker* 2. No se ha utilizado el nodo *worker* 1 porque en el momento en el que se escriben estas líneas, el nodo presenta problemas de excesiva variabilidad en los resultados, cuyas causas concretas se están inspeccionando. Puesto que el nodo *worker* 2 dispone de dos zócalos de procesador y una arquitectura de memoria principal no uniforme (NUMA), donde cada zócalo dispone de su propia caché de último nivel y módulos de memoria principal, se ha optado por situar los dos servidores en zócalos separados, minimizando la interferencia entre ellos con el objetivo de emular, en la medida de lo posible, dos nodos separados.

La Figura 7.1 ilustra la infraestructura virtual con las direcciones de cada máquina virtual y el nodo *master* asumiendo el rol de balanceador. Asimismo, la dirección 192.168.14.120 representa la dirección virtual compartida por los servidores y el director. La Figura 7.2 presenta toda la infraestructura experimental de forma más completa. En ella se observan los servidores físicos de la plataforma hardware y las máquinas virtuales que contiene cada uno.

Máquina	Núcleos físicos	Núcleos lógicos	Memoria (GB)	Dirección	Localización
Cliente 1	3	6	8	192.168.14.80	Master
Cliente 2	3	6	8	192.168.14.81	Master
Cliente 3	3	6	8	192.168.14.82	Master
Servidor 1	4	8	8	192.168.14.111	Worker 2
Servidor 2	4	8	8	192.168.14.112	Worker 2

Tabla 7.1: Características de las máquinas virtuales.

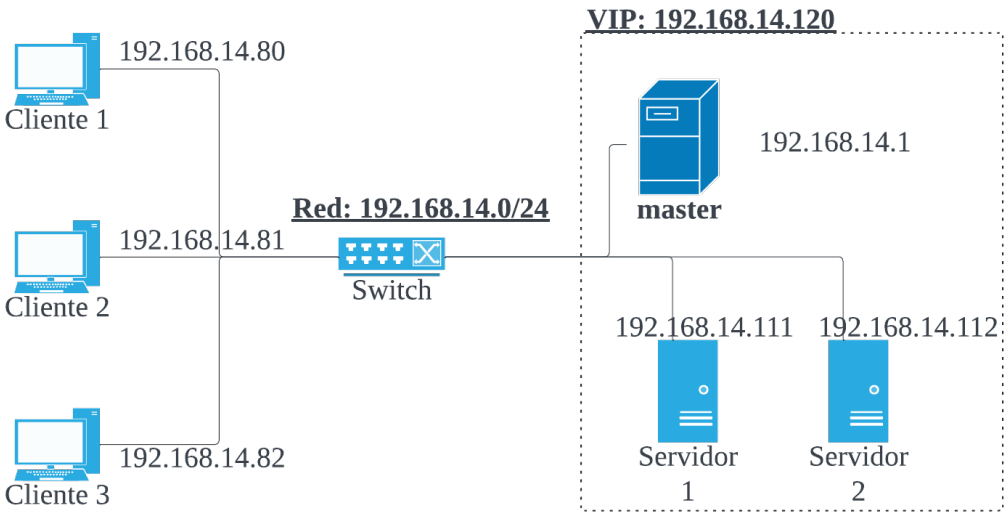


Figura 7.1: Infraestructura virtual.

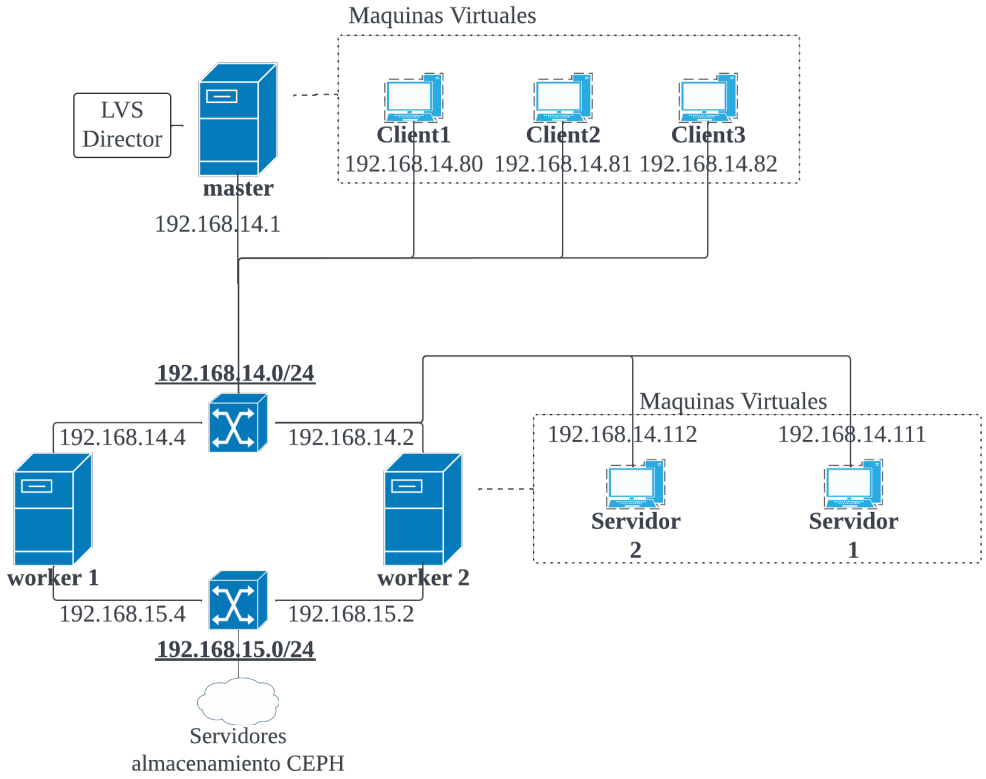


Figura 7.2: Infraestructura completa.

En lo que respecta al sistema operativo, a todas las máquinas virtuales se les ha instalado Ubuntu 18[1]. A pesar de que se trata de una versión de Ubuntu que actualmente se

[1]https://ubuntu.com/18-04

encuentra sin soporte, Tailbench++ tiene algunas dependencias con esta versión que impiden compilar todas las aplicaciones de la *suite* en versiones más actuales. Actualmente, se está trabajando para resolverlo para poder compilar las aplicaciones en versiones más recientes del sistema operativo.

7.1.2. Asignación de recursos a las máquinas virtuales

Nodo *master*

Como ya se ha especificado, la máquina *master* tiene 12 núcleos físico que soportan la ejecución de 2 hilos cada uno y 32GB de memoria principal. Puesto que, además de los clientes, aloja el balanceador de la carga, se ha decidido reserva 3 *cores* y 8GB para este fin. Los recursos restantes se han repartido entre los tres clientes. A cada uno se le asignan 3 núcleos físicos y 8GB de memoria principal. Recuérdese que para evitar que las máquinas virtuales compitan entre ellas por los recursos de los núcleos, deben especificarse que núcleos se asignan a cada cliente siguiendo la metodología explicada en la Sección 5.4.2.

Nodo *worker 2*

El nodo *worker 2* dispone de abundantes recursos, por lo que no es realista distribuirlos todos entre los dos servidores. Por ello, se ha decido asignar 8GB de memoria principal a cada servidor (al igual que en el caso de los clientes) y 4 núcleos físicos. Como se ha comentado más arriba, para minimizar las interferencias los servidores se alojan en zócalos distintos del nodo.

7.2 Evaluación de la infraestructura y las cargas Tailbench++

En esta sección se evalúan las nuevas funcionalidades de las cargas Tailbench++ así como las nuevas características de la infraestructura implantada. Para cada uno de los los experimentos, se describe su metodología, los parámetros experimentales y se presentan los resultados.

7.2.1. Verificación de las funcionalidades de Tailbench++

Como ya se ha explicado en la Sección 6.1.3, las aplicaciones Tailbench originales tienen limitaciones que no les hacen aptas para el estudios relacionados con el balanceo de la carga. A continuación, se realizan experimentos para comprobar el correcto funcionamiento de las modificaciones propuestas en Tailbench++ para superar estas limitaciones y adaptar las cargas a la infraestructura experimental.

Servidor++

Una de las modificaciones principales que se ha realizado es adaptar el servidor para que pueda aceptar un número conexiones no determinado a priori. Con tal fin, se ha modificado su código para evitar que no se procesen las peticiones hasta que se acepten un número predefinido de conexiones. Esto implica que el servidor ahora es capaz de procesar peticiones al mismo tiempo que acepta nuevas conexiones o termina conexiones existentes. Esto último significa que el servidor no finaliza cuando acaban todas las conexiones, al contrario de lo que pasaba en la versión original.

Cliente	QPS	Calentamiento	Medición
1	200	1000	6000
2	200	1000	6000

Tabla 7.2: Parámetros del experimento para testear el funcionamiento del servidor.

Para evaluar este funcionamiento, se ha utilizado la aplicación `Xapian` en uno de los servidores al que acceden dos clientes. Los clientes se inician en diferentes momentos y su ejecución no se solapa. Los parámetros de entrada son iguales para los dos clientes, y están detallados en la Tabla 7.2. Los clientes realizan 200 peticiones por segundo, con un número de 7000 peticiones en total, de las cuales 1000 son de calentamiento y 6000 son de medición.

La Figura 7.3 muestra el resultado del experimento Se muestran las 3 métricas proporcionadas por Tailbench: media (azul y verde), percentil 95 (rojo y naranja) y percentil 99 (amarillo y cían). Los sufijos C1 y C2 de la leyenda sirven para diferenciar entre el cliente 1 y el cliente 2. El eje horizontal representa el tiempo transcurrido en segundos, mientras que el eje vertical representa los tiempos de respuesta o latencia en milisegundos.

Los resultados demuestran el correcto funcionamiento de las modificaciones aplicadas al servidor. Por un lado, se distinguen claramente dos periodos de tiempo donde la latencia es distinta de cero. El primer periodo termina con la finalización del cliente 1. En este punto, con el código original de Tailbench, al no haber más clientes que atender, el servidor finalizaría también. Sin embargo, el hecho de haber un segundo periodo de latencias indica que el servidor ha seguido funcionando. Por otro lado, las latencias de los dos clientes tienen unos valores similares, lo cual significa que el sistema mantiene el mismo nivel de carga en ambos periodos, sin que el primero afecte al segundo. Esto es coherente ya que ambos clientes están configurados con el mismo QPS.

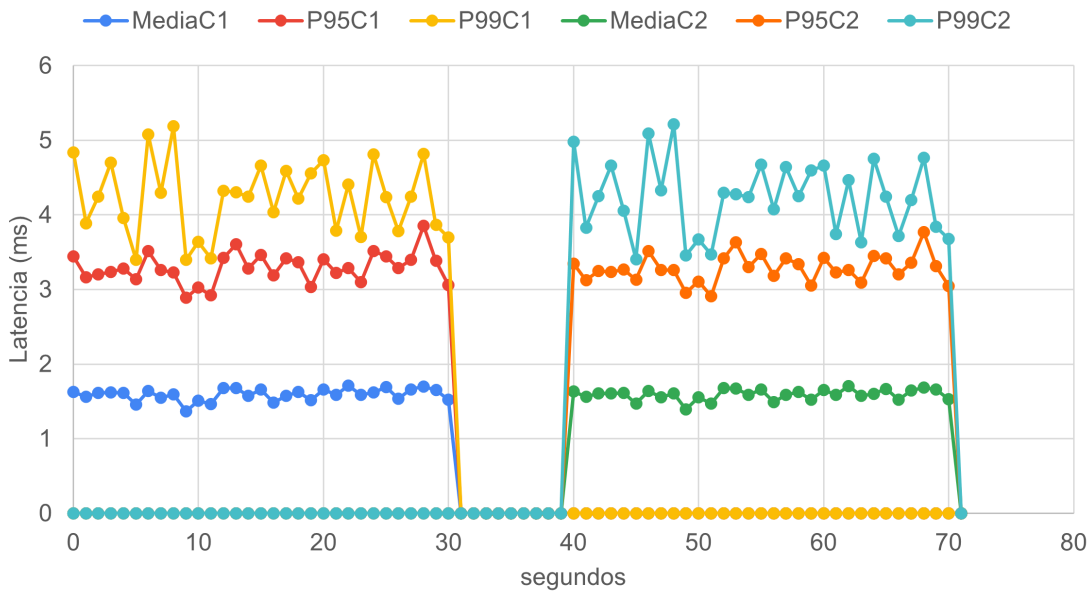


Figura 7.3: Latencias obtenidas con el acceso de dos clientes consecutivos.

Cliente	QPS	Calentamiento	Medición
1	200	1000	10000
2	200	1000	7000
3	200	1000	5000

Tabla 7.3: Parámetros del experimento para comprobar que el cliente establece el número de peticiones.

Cliente++: establecimiento de número de peticiones

En el caso del cliente, se pueden diferenciar dos modificaciones principales. La primera permite establecer cuántas peticiones realiza cada cliente. Para comprobar esta modificación, hemos ejecutado un servidor `Xapian` que recibe las conexiones de tres clientes en instantes de tiempo diferentes. Los tres clientes presentan distintos números de peticiones a realizar. Los parámetros del experimento vienen especificados en la Tabla 7.3. Todos los clientes acceden al servidor con una tasa de 200 peticiones por segundo y tienen el mismo número de peticiones de calentamiento. Comienza primero el cliente 1, y al cabo de unos 15s aproximadamente el cliente 2. Por último, después de otros 20s, el cliente 3 se inicia. Antes de que finalice el último cliente, habrán terminado los clientes 1 y 2.

La Figura 7.4 presenta los resultados del primer experimento. El eje horizontal representa el tiempo en segundos, mientras que el eje vertical muestra las latencias del percentil 95 en milisegundos. Hay tres líneas en la gráfica, cada una de las cuales corresponde a un cliente. Se puede apreciar una clara tendencia en la figura. A medida que los clientes se conectan, las fluctuaciones en las latencias aumentan. Comienza primero el cliente 1 con unas latencias de aproximadamente 3ms. Con la entrada del cliente 2, las oscilaciones de ambos crecen en cierta medida. Estas latencias se vuelven todavía más pronunciadas con la introducción del cliente 3. Durante el periodo donde los tres clientes acceden simultáneamente al servidor, se observa una alta variabilidad en las latencias, fruto de la contención. Sin embargo, con la finalización de los dos primeros clientes, el cliente 3 se estabiliza, volviendo a valores de latencia menores que 4ms.

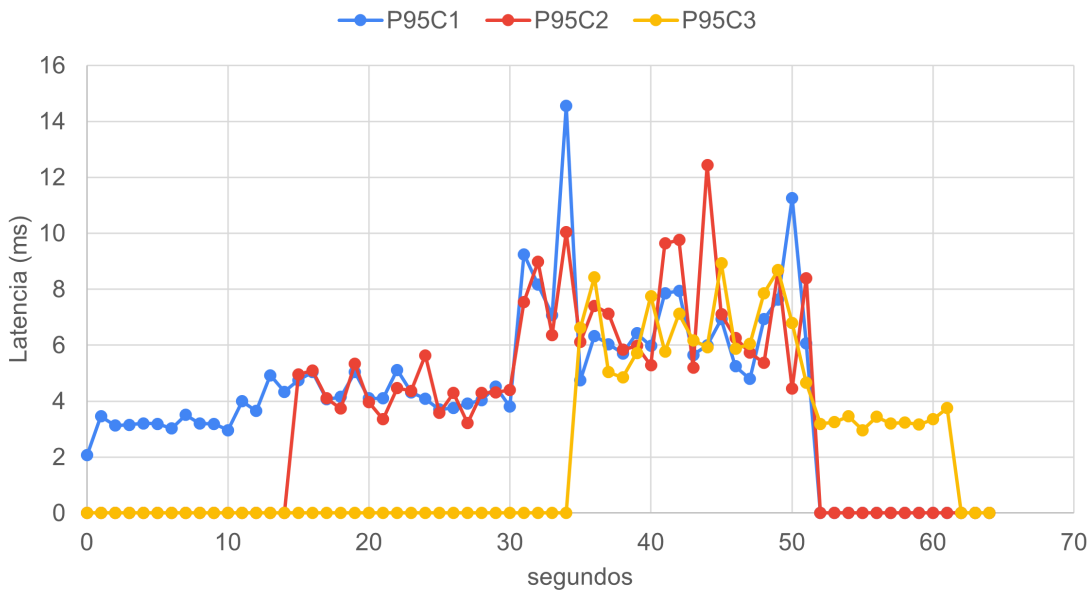


Figura 7.4: Latencias en el percentil 95 de tres clientes con distintos números de peticiones.

QPS 1	QPS 2	QPS 3	QPS 4	QPS 5	Intervalo inicial	Calentamiento	Medición
100	200	300	400	500	1000	1000	24000

Tabla 7.4: Parámetros del experimento para testear que el cliente varía el número de peticiones por segundo.

La gráfica demuestra el correcto funcionamiento de la capacidad de establecer el número de peticiones emitido por cada cliente. Aunque los tres presentan la misma tasa de peticiones por segundo, el tiempo de ejecución de cada uno es diferente debido a los distintos números de peticiones que realizan. Asimismo, el experimento vuelve a confirmar el correcto funcionamiento del servidor ante nuevas conexiones mientras se procesan las peticiones de las conexiones ya realizadas.

Cliente++: QPS variable

La segunda modificación al cliente permite variar las peticiones por segundo a lo largo del tiempo. Para comprobar esta modificación, se pondrán en marcha un servidor y un cliente de `Xapian`. El cliente se configura con un conjunto de tasas de acceso a utilizar durante su ejecución. La Tabla 7.4 muestra los valores de los parámetros de entrada del experimento. El cliente realizará 1000 peticiones de calentamiento a una tasa de 100 peticiones por segundo. Después, durante el periodo de medición, variará la tasa de envío siguiendo el orden de QPS especificado en la tabla, volviendo al final al valor inicial.

Los resultados se ilustran en la Figura 7.5. Los cambios de QPS se ven reflejados claramente en la gráfica a través de las variaciones de las latencias. A medida que se aumenta la tasa (aproximadamente cada 10s), los tiempos de respuesta se incrementan. Aunque esto es más evidente en los percentiles, en la media se aprecia mejor cuando se llega a las 500 peticiones por segundo. La reducción de las latencias después de los 50s viene dada por un retorno al valor de QPS inicial. Estos resultados demuestran efectivamente el correcto funcionamiento de esta nueva funcionalidad.

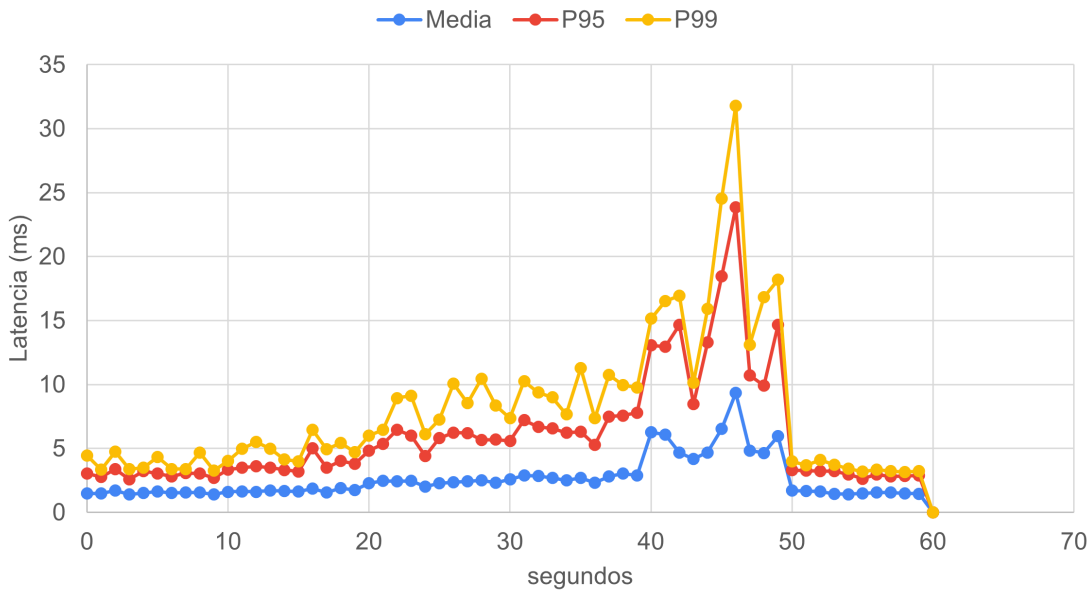


Figura 7.5: Latencias percibidas por un cliente `Xapian` que varía la tasa de peticiones durante su ejecución.

Cliente	Calentamiento	Medición
1	10000	30000
2	10000	30000
3	10000	30000

Tabla 7.5: Parámetros de entrada para todas las aplicaciones excepto `Sphinx` para la obtención de curvas de crecimiento de latencia con y sin balanceador.

Cliente	Calentamiento	Medición
1	10	100
2	10	100
3	10	100

Tabla 7.6: Parámetros de entrada para `Sphinx` para la obtención de curvas de crecimiento de latencia con y sin balanceador.

7.2.2. Validación de la infraestructura experimental

El objetivo principal de los experimentos previos es verificar las nuevas funcionalidades de Tailbench++, pero no consideraban las capacidades de la infraestructura propuesta para realizar estudios sobre el balanceo de la carga. Por este motivo, los siguientes experimentos tienen como objetivo demostrar cómo con la nueva infraestructura y Tailbench++ se pueden realizar estos estudios. Si no se especifica lo contrario, en los experimentos siguientes se utilizará el algoritmo de balanceo *Round Robin*.

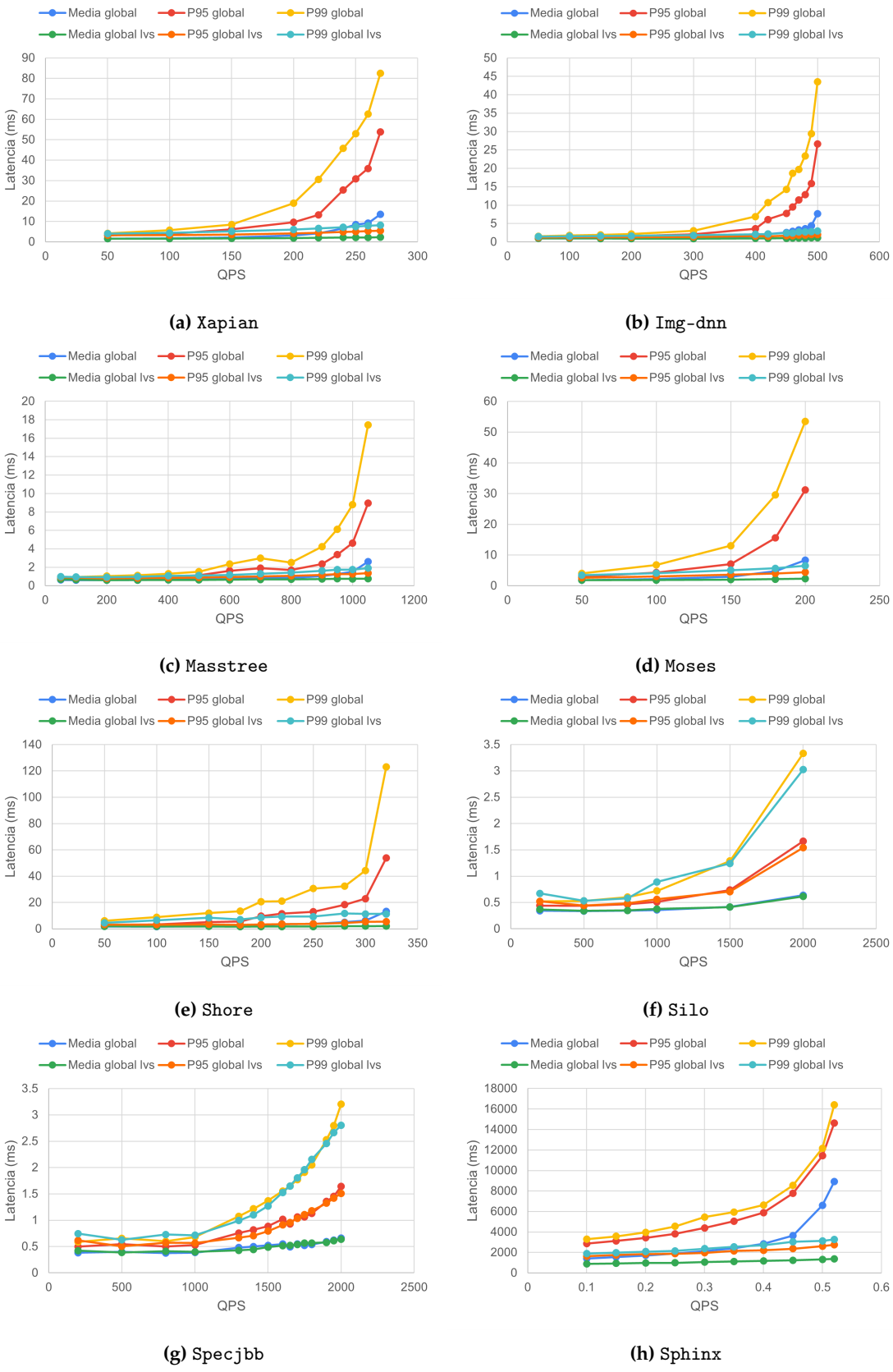
Curvas de crecimiento de latencia con y sin balanceador

El primer experimento tiene como objetivo analizar el efecto del balanceador sobre cada aplicación de Tailbench++. Para ello, se establece una situación base donde se lanzan tres clientes. Todos ellos se conectan a un mismo servidor, reflejando un caso sin balanceo. Los resultados de este experimento se comparan con otro donde la carga se balancea entre dos servidores.

Los parámetros de entrada para los clientes de todas las aplicaciones excepto `Sphinx` se muestran en la Tabla 7.5. En el caso de `Sphinx` son distintos (ver Tabla 7.6). Para las primeras, cada cliente realiza 10000 peticiones de calentamiento y 30000 de medición. En el caso de `Sphinx` se ha optado por otros valores, ya que, como ya se ha explicado en el apartado 6.3.8, esta aplicación presenta un largo tiempo de procesamiento. En su caso se han establecido 10 peticiones de calentamiento y 100 de medición.

La Figura 7.6 muestra los resultados para todas las aplicaciones: latencia media, percentil 95 y percentil 99. Cada curva representa el comportamiento medio (*global*) de los tres clientes. El eje horizontal representa la tasa de peticiones por segundo, mientras que el eje vertical simboliza la latencia en milisegundos. Los resultados de los dos escenarios se diferencian mediante el sufijo `lvs` en la leyenda, siendo las lineas azul, roja y amarilla el caso sin balanceo.

En general, como es de esperar, se observa que para 6 de las 8 aplicaciones (todas excepto `Silo` y `Specjbb`), contar con dos servidores y balancear la carga proporciona mejoras significativas tanto en la media como en los percentiles. Existen varias causas por las que esta situación puede darse, como la existencia de cuellos de botella en el almacenamiento o en la red, que no se solucionan simplemente por incrementar los recursos de cómputo. El análisis de la causa de este comportamiento se deja para trabajo futuro. A continuación, se presenta un pequeño análisis para cada una de las aplicaciones.



(a) `Xapian` **(b)** `Img-dnn`

(c) `Masstree` **(d)** `Moses`

(e) `Shore` **(f)** `Silo`

(g) `Specjbb` **(h)** `Sphinx`

Figura 7.6: Curvas de crecimiento de latencia con y sin balanceador.

Xapian. En la Figura 7.6a se puede apreciar que a tasas de peticiones bajas, por ejemplo a 50 QPS, las latencias obtenidas son, en las dos situaciones, menores que 10ms. Sin embargo, conforme aumenta la tasa, los tiempos de respuesta en el caso base tienden a crecer en mayor medida comparado con el escenario con balanceador. Esto es considerablemente más evidente en las métricas de latencia de cola, llegando a más de 80ms en el percentil 99 y más de 50ms en el percentil 95 con 270 peticiones por segundo. En cambio, con la misma tasa de peticiones, la existencia de un balanceador permite seguir manteniendo las latencias por debajo de los 10ms. Todo esto indica que el balanceo de carga reduce notoriamente los tiempos de respuesta en las aplicaciones de búsqueda web como `Xapian`.

Img-dnn. La Figura 7.6b muestra una tendencia parecida inicialmente. Es a partir de las 300 peticiones por segundo cuando las curvas del caso base comienza a crecer exponencialmente. Se pasan de unas latencias menores de 5ms hasta valores superiores a 40ms en el caso del percentil 99, y un incremento relevante también, aunque en menor grado, de más de 20ms en el percentil 95. Todo esto revela un deterioro importante del rendimiento del servidor bajo las cargas de los tres clientes conectados. A una tasa de 500 QPS, el 5 % de las solicitudes tardan más de 25ms en obtener respuesta. En situaciones más extremas, el 1 % de todas éstas superan los 40ms. Al contrario de lo que sucede en el caso sin balanceador, el escenario con un director que reparte las conexiones entre los dos servidores presenta una evidente mejora en las latencias. Se consigue mantener los tiempos de respuesta bajo los 5ms, demostrando que la distribución de carga mejora el rendimiento de aplicaciones de reconocimiento de imágenes como `Img-dnn`.

Masstree. En la Figura 7.6c se aprecia que los comportamientos de los dos escenarios son parecidos a sus respectivos en las cargas previas. Las prestaciones obtenidas son parecidas cuando la carga es baja. Conforme aumenta la tasa de peticiones por segundo de los clientes, el sistema sin balanceador comienza a experimentar ciertos aumentos de tiempos de respuesta, sobre todo en los percentiles. En contraposición, cuando se equilibra la carga las latencias se mantienen controladas. A partir de los 800 QPS, el rendimiento del sistema sin balanceo sufre un empeoramiento notable, disparándose exponencialmente las latencias. En cambio, gracias a la existencia de un balanceador, el crecimiento es más controlado en el segundo caso. Con una tasa de 1050 peticiones por segundo, el 1 % de las peticiones tardan más de 18ms aproximadamente en tener respuesta en el escenario sin balanceo. Con éste, el limite baja a los 2ms. En resumen, se puede afirmar que el balanceo de carga mejora significativamente el rendimiento de la aplicación `Masstree`.

Moses. Al contrario de las aplicaciones anteriores, en la Figura 7.6d se puede apreciar que la diferencia en el rendimiento de los dos casos aparece desde el principio con tasas bajas. Asimismo, las conclusiones que se pueden sacar son similares. Conforme aumenta la tasa, las latencias de cola crecen exponencialmente en el caso sin balanceador, lo cual, supone una degradación significativa del rendimiento del sistema. En cambio, la infraestructura con director presenta unas latencias más estables y controladas. Esto es lógico, ya que las cargas de los clientes se reparten entre dos servidores, dando mayor capacidad al sistema en global. En definitiva, la existencia de un intermediario optimiza los rendimientos de servicios similares a `Moses`.

Shore. En la Figura 7.6e, los patrones que se observan son similares a sus correspondientes en los resultados de las aplicaciones previas. Al igual que `Xapian` o `Masstree`, las latencias a niveles de carga baja de los dos escenarios no presentan mucha diferencia. Es a partir de los 200 peticiones por segundo cuando comienzan a distanciarse considerablemente los percentiles de cada escenario. El rendimiento del sistema sin balanceo se ve degradado aún más con tasas superiores a 250 QPS, donde el 1 % de todas la peticiones tarda más de 30ms en obtener respuesta. Conforme aumentan las peticiones por segundo, este porcentaje de solicitudes puede sufrir hasta 120ms de latencia. Todo esto está indicando que el sistema sin balanceador se está saturando con las cargas a las que

le someten los tres clientes. En cambio, la infraestructura con director, al repartir las conexiones de los clientes entre dos servidores, consigue controlar los tiempos de respuesta bajo los 20ms, ofreciendo mejores rendimientos.

Silo. En la Figura 7.6f se aprecia un comportamiento distinto a las aplicaciones vistas hasta ahora. Se observa que ambos casos (con y sin balanceador) presentan un crecimiento de latencias similar conforme aumenta la tasa de peticiones por segundo. Aunque se aprecia una pequeña mejora de latencias de cola en niveles de cargas altos, estos datos sugieren que la aplicación `Silo` no se beneficia de recursos de cómputo adicionales.

Specjbb. En la Figura 7.6g se observa cómo las curvas de latencias mostradas presentan un comportamiento similar a las de `Silo`, donde las diferencias de latencias entre los dos escenarios es insignificante. En los dos casos, las latencias se mantienen estables y bajas al inicio. Sin embargo, a medida que la tasa de peticiones por segundo aumenta (a partir de los 1000 QPS), el crecimiento de las latencias de cola es más evidente en ambos casos.

Sphinx. La Figura 7.6h muestra unos valores mucho mayores de latencia debido al alto tiempo de procesamiento de `Sphinx`. Se aprecia en esta gráfica que a niveles de cargas bajas, las latencias en los dos escenarios es bastante estable. Sin embargo, los valores de los percentiles del caso sin balanceador superan considerablemente a los de la situación con balanceador, indicando cierta mejora desde el inicio debido a las altas demandas de cómputo de la aplicación. Conforme aumenta la tasa de peticiones por segundo, el rendimiento del sistema base comienza a degradarse. Las diferencias entre las métricas, sobre todo las de latencias de cola, se expanden aún más. A partir de los 0,4 QPS, tanto la media como los percentiles del escenario sin balanceador crecen de manera exponencial. En comparación, distribuir la carga muestra mejores desempeños, manteniendo controlados y estabilizados los tiempos de respuesta. Para una tasa de 0,6 peticiones por segundo, en la situación sin balanceador en promedio las peticiones tardan más de 8s en obtener respuesta. En cambio, con equilibrado de la carga, este tiempo promedio se mantiene por debajo de los 2s. Asimismo, las diferencias entre las latencias de cola es aún mayor. El 1 % de las peticiones tardan más de 16s en la situación base, comparado con los 4s del caso mejorado. Con todo esto mencionado, se puede afirmar que la distribución de cargas optimiza el rendimiento de las aplicaciones similares a `Sphinx`.

Evolución de la latencia en el tiempo con y sin balanceador: clientes con la misma tasa

A diferencia del experimento anterior que estudia los cambios de latencia en función de la tasa de peticiones por segundo, este experimento consiste en analizar la variación de latencia a lo largo del tiempo utilizando solo una de las aplicaciones, concretamente `Xapian`. De la misma manera que en el experimento anterior, se diferencian dos escenarios. Por un lado, el escenario sin balanceo que consta de un servidor con tres clientes. Por otro, el escenario con balanceador formado por dos servidores y tres clientes gestionados por el director.

Independientemente del escenario, todos los clientes comienzan al mismo tiempo, con la misma tasa de peticiones por segundo y realizan el mismo número de peticiones de

Cliente	QPS	Calentamiento	Medición
1	250	1000	15000
2	250	1000	15000
3	250	1000	15000

Tabla 7.7: Parámetros de entrada para la obtención de la evolución temporal de las latencias con QPS fijo.

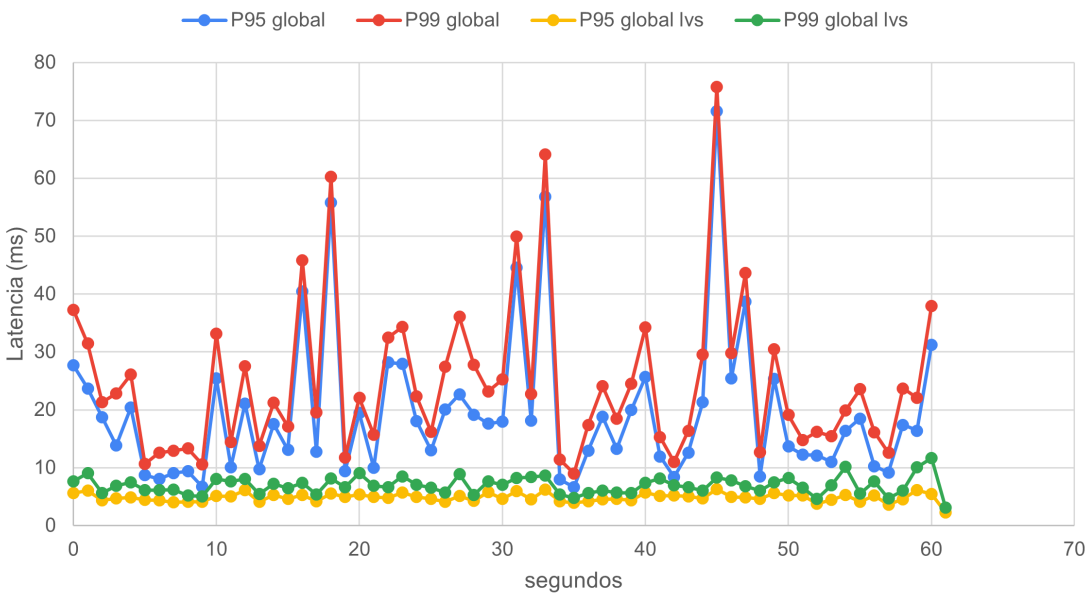


Figura 7.7: Evolución de las latencias de `Xapian` en los percentiles 95 y 99 con y sin balanceador y QPS fijo.

calentamiento y medición. Los valores de los parámetros se muestran en la Tabla 7.7. Los clientes funcionan a una tasa de 250 peticiones por segundo y realizan 16000 peticiones en total, de las cuales 1000 son de calentamiento y 15000 son de medición.

La Figura 7.7 presenta los resultados obtenidos. El eje horizontal representa el tiempo en segundos, mientras que el eje vertical simboliza la latencia en milisegundos. Esta gráfica solo muestra las latencias de cola (percentil 95 y percentil 99) globales. Por un lado, las lineas roja y azul representan el caso base sin balanceador. Por otro lado, la amarilla y la verde representan la situación con director. Se observan claramente diferencias significativas entre los dos escenarios. La situación base presenta mayor variabilidad durante todo el tiempo, desde valores de latencias cercanos a 10ms hasta valores extremos de más de 70ms. En cambio, las fluctuaciones del escenario con balanceo son mucho menores, produciendo resultados más estables y consistentes, ya que rara vez superan el límite de los 10ms. Todo esto indica que el rendimiento del sistema base se ha degradado significativamente, lo que conduce a una mala gestión de la peticiones de los clientes. En cambio, añadir recursos y balancear la carga muestra muchos mejores resultados. Esto reafirma que el balanceo de carga incrementa sustancialmente las prestaciones de los servicios de latencia crítica.

Impacto de las políticas de balanceo ante clientes con diferentes tasas

El siguiente caso de estudio se centra en evaluar el comportamiento del director LVS ante clientes con diferentes tasas. En concreto, se pretende analizar si el intermediario debe, en función de la tasa de peticiones, modificar su decisión de distribución. Nótese que los algoritmos incluidos en LVS no consideran la tasa de peticiones, sino el número de conexiones, por lo que este estudio abre la puerta al diseño de nuevas políticas de balanceo.

En este estudio, se va a comparar la política clásica *Round Robin*, ampliamente utilizada en los sistemas reales con una política que trata de balancear la tasa de peticiones entre los servidores. A esta última política la denominaremos *óptima*.

Cliente	QPS	Calentamiento	Medición
1	500	5000	30000
2	200	2000	12000
3	200	2000	12000

Tabla 7.8: Parámetros de entrada para el estudio del impacto del balanceo ante clientes con diferentes tasas.

Para evaluar ambas políticas, se utilizará la aplicación `Xapian` con dos servidores y tres clientes. Con respecto a los clientes, sus parámetros se detallan en la Tabla 7.8. El cliente 1 presenta una tasa de 500 peticiones por segundo, superior a los otros dos que tienen una tasa de 200 QPS. Para que los tres clientes se ejecuten durante un tiempo similar, se ha establecido que el primero, al tener una tasa mayor, realice más peticiones que las otros dos, siendo 5000 peticiones de calentamiento y 30000 de medición. En cambio, los dos restantes al tener una misma tasa, realizarán el mismo número de peticiones de calentamiento y de medición.

Los clientes comienzan su ejecución al mismo tiempo y la mantienen por el mismo periodo. En este caso, la política de balanceo óptima distribuye las conexiones de manera que los dos clientes con menor tasa se conectan al mismo servidor, dejando el servidor restante para el cliente con la mayor tasa, mientras que *Round Robin* simplemente distribuirá las conexiones de los clientes entre los servidores atendiendo a su orden de llegada al director. A continuación, se describen los resultados obtenidos por cada política.

Algoritmo de balanceo *Round Robin*. La Figura 7.8 muestra las latencias de cola (percentil 95) obtenidos por los tres clientes en el caso donde se aplica la politica por defecto *Round Robin*. El eje horizontal representa el tiempo en segundos, mientras que el eje vertical muestra la latencia en milisegundos. Se han utilizado tres colores para diferenciar cada cliente, la línea azul representa el cliente 1, la linea roja el cliente 2 y la amarilla el cliente 3. Se pueden apreciar mayores fluctuaciones en el primer cliente comparado con los otros dos, debido a su alta tasa de peticiones por segundo. Por otro lado, el cliente 2, que tiene la misma tasa que el cliente 3, obtiene unas latencias superiores a este último. Esto se debe a que *Round Robin* reparte las conexiones de manera que los clientes 1 y 2 se ejecutan en la misma máquina, mientras que el cliente 3 se ejecuta en solitario en el otro servidor.

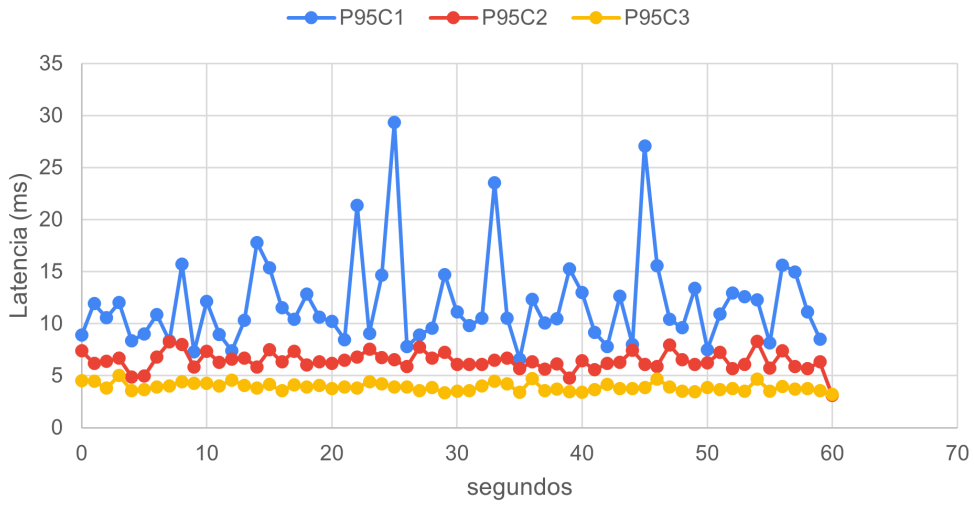


Figura 7.8: Latencias en el percentil 95 de los tres clientes para la política *Round Robin*.

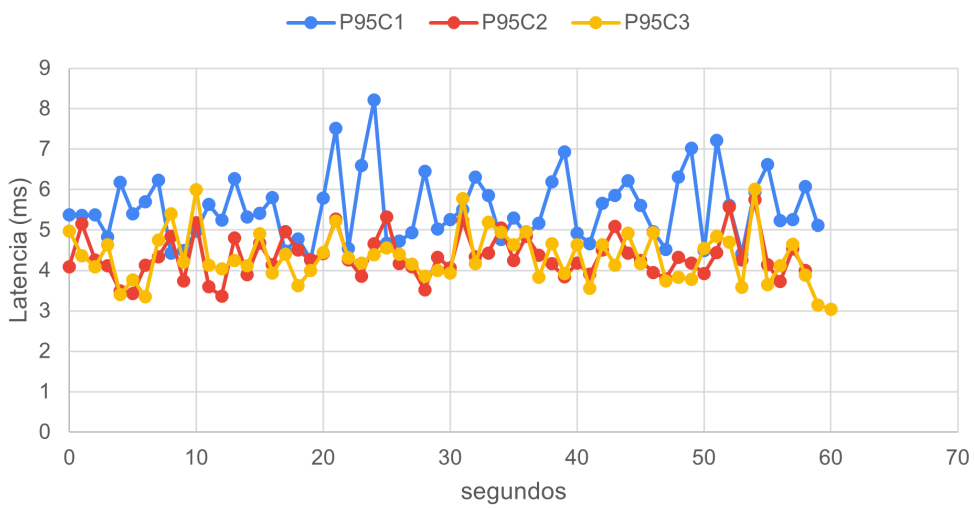


Figura 7.9: Latencias en el percentil 95 de los tres clientes para la política óptima.

Algoritmo de balanceo óptimo. La Figura 7.9 muestra los resultados obtenidos por la política o algoritmo óptimo. Se puede apreciar que este cambio en la distribución ha tenido un impacto significativo en las prestaciones del cliente 1, ya que presenta una menor latencia que con *Round Robin*. Además se han reducido las fluctuaciones de manera significativa: el pico máximo del cliente 1 ha pasado de 30ms a 9ms. Asimismo, se observa una superposición de las latencias de los clientes 2 y 3, indicando variaciones dentro de unos rangos similares. Lo cual es lógico al estar los dos en la misma máquina servidor y presentar la misma tasa. Si se comparan las latencias obtenidas por el cliente 2 en los dos escenarios, se detecta también una mejora de prestaciones. Antes, el percentil 95 tomaba valores superiores a 5ms la mayor parte del tiempo. En cambio, ahora está por debajo de ese valor generalmente. Esto se debe a que el cliente 2 ya no comparte servidor con el cliente 1, que presenta la mayor tasa. En cuanto al cliente 3, no parece haber sufrido una degradación significativa a pesar de compartir servidor con el cliente 2. Esto es debido a que la suma de los QPS de ambos clientes no es suficiente para elevar significativamente la latencia de cola con los recursos disponibles para un servidor.

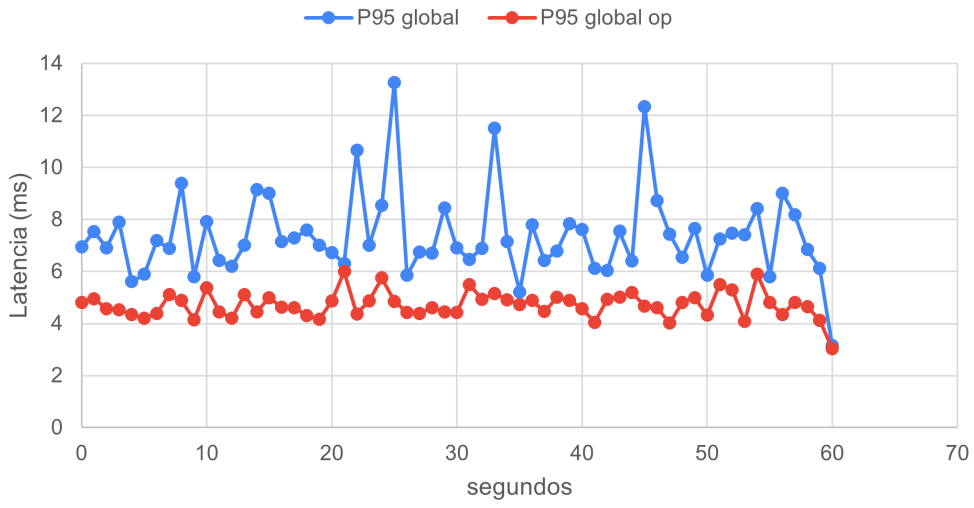


Figura 7.10: Comparación de las latencias en el percentil 95 globales para ambas políticas.

Comparativa. Por último, para analizar la mejora en conjunto, se presenta la Figura 7.10. Esta gráfica muestra la evolución de las latencias del percentil 95 globales del sistema en función del tiempo. La línea azul representa la política base, mientras que la linea roja el escenario optimizado. En general, se aprecian mayores retrasos y variabilidades en el caso sin optimización, mientras que el segundo caso presenta unas latencias de cola más consistentes y bajas. En definitiva, el análisis demuestra que el sistema optimizado exhibe mejores prestaciones, mayor estabilidad, así como latencias de cola más predecibles. Los resultados marcan una vía de trabajo futuro para el desarrollo de políticas de balanceo que tengan en cuenta las peticiones por segundo para mejorar las latencias de cola y la consistencia en las prestaciones de los sistemas distribuidos en la nube.

CAPÍTULO 8
Conclusiones

En este último capítulo se resumen las principales contribuciones que se han alcanzado en el presente Trabajo Fin de Grado. Asimismo, se relaciona el contenido del proyecto con asignaturas cursadas durante la carrera, además de exponer posibles líneas de trabajo futuro.

8.1 Principales contribuciones

Los servicios informáticos actuales están exigiendo cada vez más características específicas para satisfacer sus exigentes requisitos. Encontrar y desarrollar estrategias que permitan cumplir con el rendimiento de estas aplicaciones es fundamental para poder satisfacer a los usuarios y competir en el mercado. Por lo tanto, es crucial tener una infraestructura de pruebas que refleje fielmente los entornos reales, donde se pueda ejecutar cargas de trabajo que representen estas aplicaciones.

En este proyecto se ha desarrollado un entorno experimental que refleja adecuadamente las características de las infraestructuras de los servicios reales: múltiples máquinas con diferentes roles y un entorno virtual. El propósito de esta infraestructura es emplearla para analizar y estudiar los desafíos que presentan los entornos de producción, así como proyectos con las empresas. Para validar su correcto funcionamiento, se han creado cinco máquinas virtuales, de las cuales, tres funcionan como clientes y dos como servidores. Los clientes acceden a los servidores mediante un director que decide qué servidor va a servir dicha conexión. Se trata de un diseño utilizado comúnmente en servicios de gran escala.

Para validar la infraestructura, se ha empleado un conjunto de cargas de trabajo representativas de aplicaciones reales. Para poder reproducir un comportamiento típico de un sistema distribuido, se ha adaptado su diseño y añadido nuevas funcionalidades, entre las que se incluye el soporte a conexiones dinámicas de múltiples clientes o la variación de la tasa de envío a lo largo del tiempo de ejecución.

La infraestructura diseñada y las aplicaciones modificadas han permitido llevar a cabo un estudio de la latencia de cola y el efecto que tiene el balanceo de carga sobre ella. Los resultados del estudio corroboran la efectividad de esta técnica para reducir la latencia y mejorar las prestaciones de los servicios de latencia crítica. Esto se refleja, por ejemplo, con el experimento que compara las curvas de latencias globales del apartado 7.2.2. La infraestructura experimental que implementa el balanceo de carga presenta unos crecimientos de latencias de cola significativamente menores que el caso sin balanceador.

En conclusión, este trabajo ha logrado los objetivos establecidos. Se ha diseñado una infraestructura experimental funcional y realista, y seleccionado y adaptado un conjunto

representativo de aplicaciones de latencia crítica para su correcta ejecución, lo que ha permitido analizar las influencias del balanceo de carga sobre la latencia de cola.

8.2 Relación del trabajo con los estudios cursados

En este apartado se presentan las asignaturas cursadas durante el Grado en Ingeniería Informática que guardan mayor relación con los contenidos abordados en este trabajo.

- **Redes de Computadores**. Esta asignatura pretende introducir al alumno en el funcionamiento de las comunicaciones de los sistemas de información. Da las bases de los protocolos a seguir para garantizar una transmisión efectiva y eficiente de información entre los computadores. Ha sido de gran ayuda a la hora de configurar los parámetros de red de las máquinas virtuales y el balanceador.
- **Concurrencia y Sistemas Distribuidos**. La asignatura trata, por un lado, la gestión efectiva de múltiples tareas que se ejecutan al mismo tiempo. Por otro lado, introduce al alumno en el funcionamiento de los servicios distribuidos, así como los mecanismos y retos que presentan. Esta materia ha sido esencial a la hora de adaptar las aplicaciones seleccionadas.
- **Tecnología de Sistemas de Información en la Red**. Esta asignatura pretende mostrar al alumno diferentes opciones de diseño de servicios distribuidos fiables y escalables en la red. Su ayuda ha sido fundamental para comprender, al igual que la asignatura anterior, los servicios distribuidos actuales, así como la computación en la nube.
- **Fundamentos de Sistemas Operativos**. Se introduce al alumno en los conocimientos básicos de los sistemas operativos. Gracias a ella, ha sido más fácil comprender la importancia del balanceo de carga.
- **Gestión y Configuración de la Arquitectura de los Sistemas de Información**. Presenta al alumno la estructura y componentes de los centros de datos. Los conocimientos en esta materia me han permitido tener claro cómo diseñar la infraestructura experimental para que refleje de forma simplificada los entornos reales.
- **Estructura de Computadores**. La asignatura aborda el estudio de los componentes centrales que integran un ordenador. Los conocimientos adquiridos en esta materia ha permitido una mejor comprensión de la infraestructura hardware utilizada, concretamente las procesadores y la memoria principal.
- **Arquitectura e Ingeniería de Computadores**. Estudia el rendimiento de los computadores desde una perspectiva de prestaciones, proporcionando conceptos clave para entender la motivación de las actuales arquitecturas de computadores. Esta asignatura proporciona conceptos necesarios para la entender, por ejemplo, la asignación de núcleos lógicos (hilos) a máquinas virtuales o la localización de los núcleos virtuales que miminice la interferencia entre servicios.

Además de las competencias específicas adquiridas en estas asignaturas, el trabajo me ha permitido adquirir múltiples competencias transversales. Se destacan las siguientes:

- **Innovación y creatividad**. Se trata de una competencia que ha sido fundamental para el proyecto. Me ha permitido diseñar un entorno experimental que reflejara bien las infraestructuras reales de los servicios informáticos, así como optimizar las funcionalidades de las cargas de trabajo seleccionadas.

- **Responsabilidad y toma de decisiones**. Se ha aplicado esta competencia no solo en la búsqueda y el análisis de información, sino también en el aprendizaje autónomo de conocimientos y herramientas necesarios para llevar cabo el trabajo de implementación del entorno de pruebas y modificación de las aplicaciones.
- **Comunicación efectiva**. Gracias a esta competencia, se ha podido estructurar efectivamente y coherentemente el contenido del trabajo, plasmando los objetivos, ideas y resultados de manera precisa en la memoria.

8.3 Trabajo futuro

Con los resultados obtenidos en este trabajo, nuevos temas y desafíos se abren para futuros proyectos. Se presentan, a continuación, algunas de las líneas que se pretende desarrollar como trabajo futuro.

- **Evaluar el sistemas con distintas cargas**. En este trabajo solo se ha experimentado con las cargas Tailbench cuyas prestaciones vienen definidas por la latencia de cola, por lo que sería interesante emplear otras tipo de cargas como científicas, de *big data* o de *machine learning* con el fin recrear la heterogeneidad de aplicaciones que se ejecutan en el *cloud*.
- **Mejorar la infraestructura experimental**. Los entornos de producción implementan mecanismos ante posibles fallos del sistema, como las caídas de servidores o del director. Por este motivo, una posible optimización de la infraestructura sería prevenir estas situaciones, instalando balanceadores secundarios que tomen el control de la situación si el principal falla e incorporar al entorno más nodos.
- **Diseñar políticas de balanceo que considere la carga que los clientes somete a los servidores**. Según los resultados del experimento de la Sección 7.2.2, el balanceador LVS no implementa ningún algoritmo que considere la carga que los clientes someten a los servidores. Por este motivo, sería conveniente diseñar una política que distribuya las conexiones de los clientes a los servidores teniendo en cuenta la carga.
- **Aplicar inteligencia artificial a la distribución de carga**. Emplear técnicas de inteligencia artificial para estimar la carga recibida y ajustar el balanceo de carga dinámicamente mejorando el rendimiento del sistema.
- **Evaluar el impacto del sistema de almacenamiento distribuido.** Dado que la infraestructura desplegada permite emplear un sistema de almacenamiento distribuido, se pretende estudiar el impacto del sistema de almacenamiento distribuido en las prestaciones.

Bibliografía

[1] R. Fernández, "El uso de internet en el mundo," 2024, accessed: 18-Jun-2024. [Online]. Available: https://es.statista.com/temas/9795/el-uso-de-internet-en-el-mundo/

[2] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[3] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[4] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–14. [Online]. Available: https://doi.org/10.1145/2670979.2670988

[5] L. Pons, S. Petit, J. Pons, M. E. Gómez, C. Huang, and J. Sahuquillo, "Stratus: A hardware/software infrastructure for controlled cloud research," in *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2023, pp. 299–306.

[6] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[7] J. F. Kurose and K. W. Ross, *Computer networking : a top-down approach*, global ed. ed. Harlow: Pearson, 2022.

[8] K. R. Fall and W. R. Stevens, *Tcp/ip illustrated*. Addison-Wesley Professional, 2012, vol. 1.

[9] M. M. Alani, *Guide to OSI and TCP/IP models*. Springer, 2014.

[10] Y. Li, D. Li, W. Cui, and R. Zhang, "Research based on osi model," in *2011 IEEE 3rd International Conference on Communication Software and Networks*, 2011, pp. 554–557.

[11] S. Kumar, S. Dalal, and V. Dixit, "The osi model: overview on the seven layers of computer networks," *International Journal of Computer Science and Information Technology Research*, vol. 2, no. 3, pp. 461–466, 2014.

[12] N. Ruparelia, *Cloud Computing*, 1st ed. Cambridge, Massachusetts ; London, England: The MIT Press, 2016, this book provides a lucid overview of the implications of the cloud phenomenon and the opportunities and risks associated with it.

[13] Google Cloud. (2024) ¿qué es la computación en la nube? Accessed: 20-Jun-2024. [Online]. Available: https://cloud.google.com/learn/what-is-cloud-computing?hl=es#section-1

[14] J. Hurwitz and D. Kirsch, *Cloud Computing for Dummies*, 2nd ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2020.

[15] J. Surbiryala and C. Rong, "Cloud computing: History and overview," in *2019 IEEE Cloud Summit*, 2019, pp. 1–7.

[16] National Institute of Standards and Technology. (2024) Cloud computing project. Accessed: 20-Jun-2024. [Online]. Available: https://csrc.nist.gov/Projects/Cloud-Computing

[17] E. Gorelik, "Cloud computing models," S.M. thesis, Massachusetts Institute of Technology, Engineering Systems Division, 2013. [Online]. Available: https://dspace.mit.edu/handle/1721.1/79811

[18] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing — the business perspective," *Decision Support Systems*, vol. 51, no. 1, pp. 176–189, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167923610002393

[19] M. Portnoy, *Virtualization Essentials*, 3rd ed. Wiley, 2024. [Online]. Available: https://learning.oreilly.com/library/view/virtualization-essentials-3rd/9781394181568/

[20] VMware, "What is a virtual machine?" https://www.vmware.com/topics/glossary/content/virtual-machine.html, 2024, accessed: 2024-06-25.

[21] IBM, "Containers: A complete guide," 2023, accessed: 2024-06-27. [Online]. Available: https://www.ibm.com/es-es/topics/containers

[22] T. Bourke, *Server load balancing*. Sebastopol [etc: O'Reilly, 2001.

[23] P. Kumar and R. Kumar, "Issues and challenges of load balancing techniques in cloud computing: A survey," *ACM Computing Surveys*, vol. 51, no. 6, p. Article 120, 2019. [Online]. Available: https://doi.org/10.1145/3281010

[24] E. Jafarnejad Ghomi, A. M. Rahmani, and N. Nasiri, "A survey on load balancing algorithms in cloud computing," *Journal of Network and Computer Applications*, vol. 88, pp. 50–71, 2017.

[25] N. Mohamed and J. Al-Jaroodi, "A survey on load balancing algorithms for cloud computing," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, 2014, pp. 135–142.

[26] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion, "Energy proportionality and workload consolidation for latency-critical applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 342–355. [Online]. Available: https://doi.org/10.1145/2806777.2806848

[27] E. Huedo, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, "Opportunistic deployment of distributed edge clouds for latency-critical applications," *Journal of Grid Computing*, vol. 19, no. 2, 2021. [Online]. Available: https://doi.org/10.1007/s10723-021-09545-3

[28] M. R. Reynolds and J. Wang, "A hardware-efficient maximum-likelihood estimator for neural binary-event sources using clock-free time-encoding," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 9, no. 3, pp. 338–350, 2015.

[29] C. Delimitrou and M. Marty, "Tales of the tail: Past and future," *IEEE Micro*, pp. 1–7, 2024.

[30] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable Cloud computing environments and the CloudSim toolkit: Challenges and opportunities," in *Proceedings of HPCS*, 2009, pp. 1–11.

[31] G. Belalem, F. Z. Tayeb, and W. Zaoui, "Approaches to improve the resources management in the simulator CloudSim," in *Proceedings of ICICA*. Springer, 2010, pp. 189–196.

[32] B. Liang, X. Dong, Y. Wang, and X. Zhang, "Memory-aware resource management algorithm for low-energy cloud data centers," *Future Generation Computer Systems*, vol. 113, pp. 329–342, 2020.

[33] C. Liu, W. Li, J. Wan, L. Li, Z. Ma, and Y. Wang, "Resource Management in Cloud Based on Deep Reinforcement Learning," in *Proceedings of ICCCI*, 2022, pp. 28–33.

[34] S. Badia, A. Carpen-Amarie, A. Lèbre, and L. Nussbaum, "Enabling large-scale testing of IaaS cloud platforms on the grid'5000 testbed," in *Proceedings of the International Workshop on Testing the Cloud*, 2013, pp. 7–12.

[35] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The Design and Operation of CloudLab," in *Proceedings of USENIX ATC*, 2019, pp. 1–14.

[36] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock *et al.*, "Lessons learned from the chameleon testbed," in *Proceedings of USENIX ATC*, 2020, pp. 219–233.

[37] Q. Chen, S. Xue, S. Zhao, S. Chen, Y. Wu, Y. Xu, Z. Song, T. Ma, Y. Yang, and M. Guo, "Alita: Comprehensive Performance Isolation through Bias Resource Management for Public Clouds," in *Proceedings of SC*, 2020, pp. 1–13.

[38] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ML-driven performance debugging in microservices," in *Proceedings of ASPLOS*, 2021, pp. 135–151.

[39] A. Suresh and A. Gandhi, "ServerMore: Opportunistic Execution of Serverless Functions in the Cloud," in *Proceedings of SoCC*, 2021, p. 570–584.

[40] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in *Proceedings of HPCA*, 2020, pp. 193–206.

[41] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, "ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud," in *Proceedings of HPCA*, 2022, pp. 155–168.

[42] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in *Proceedings of HPCA*, 2020, pp. 167–179.

[43] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications," in *Proceedings of CLOUD*, 2018, pp. 82–89.

[44] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of ASPLOS*, 2019, pp. 107–120.

[45] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi, "Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments," in *Proceedings of SoCC*, 2019, p. 272–285.

[46] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2150976.2150982

[47] J. S. Dong, J. Sun, and J. Wang, "Checking and reasoning about semantic similarity between business process models," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC 2008)*, ser. ICSOC '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 71–82. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[49] Z. Jia, S. Blake, X. Wen, G. Fu, C. Luo, L. Wang, G. Yang, and C. Wang, "Bigdatabench: a big data benchmark suite from internet services," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 488–499.

[50] F. S. Prity and M. M. Hossain, "A comprehensive examination of load balancing algorithms in cloud environments: A systematic literature review, comparative analysis, taxonomy, open challenges, and future trends," *Iran Journal of Computer Science*, 2024. [Online]. Available: https://doi.org/10.1007/s42044-024-00183-y

[51] D. A. Shafiq, N. Jhanjhi, and A. Abdullah, "Load balancing techniques in cloud computing environment: A review," *Journal of King Saud University – Computer and Information Sciences*, vol. 34, pp. 3910–3933, 2022. [Online]. Available: https://doi.org/10.1016/j.jksuci.2021.02.007

[52] R. Tasneem and M. A. Jabbar, "An insight into load balancing in cloud computing," in *Proceeding of 2021 International Conference on Wireless Communications, Networking and Applications*, Z. Qian, M. Jabbar, and X. Li, Eds. Singapore: Springer Nature Singapore, 2022, pp. 1125–1140.

[53] S. K. Mishra, B. Sahoo, and P. P. Parida, "Load balancing in cloud computing: A big picture," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 2, pp. 149–158, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157817303361

[54] A. Althoubi, R. Alshahrani, and H. Peyravi, "Tail latency in datacenter networks," in *Proceedings of the 2020 International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, ser. Lecture Notes in Computer Science, vol. 12527. Springer Nature Switzerland AG, 2021, pp. 254–272.

[55] B. Alankar, G. Sharma, H. Kaur, R. Valverde, and V. Chang, "Experimental setup for investigating the efficient load balancing algorithms on virtual cloud," *Sensors*, vol. 20, no. 24, p. 7342, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/24/7342

[56] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, 2006, pp. 307–320.

[57] Proxmox, "Proxmox," https://www.proxmox.com/en/, 2024, [Accessed: 01-Jul-2024].

[58] W. Ahmed, *Mastering proxmox : Build virtualized environments using proxmox VE hypervisor*, third edition. ed. Birmingham, England;: Packt, 2017.

[59] Debian, "Debian – el sistema operativo universal," https://www.debian.org/index.es.html, 2024, [Accessed: 01-Jul-2024].

[60] Nginx, Inc. (2024) Nginx. Accessed: 18-Jun-2024. [Online]. Available: https://nginx.org/en/

[61] HAProxy Technologies, LLC. (2024) Haproxy. Accessed: 18-Jun-2024. [Online]. Available: https://www.haproxy.org/

[62] L. V. S. Project. (2024) Linux virtual server. Accessed: 18-Jun-2024. [Online]. Available: http://www.linuxvirtualserver.org/

[63] W. Zhang *et al.*, "Linux virtual server for scalable network services," in *Ottawa Linux Symposium*, vol. 2000, 2000.

[64] T. Sasajima and S. ya Nishizaki, "Event-driven implementation of layer-7 load balancer," in *Proceedings of the 2013 Advances in Information Technology*. Switzerland: Springer International Publishing, 2013, pp. 162–172. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-03783-7_15

[65] Q. Wei, G. Xu, and Y. Li, "Research on cluster and load balance based on linux virtual server," in *ICICA 2010, Part I, CCIS 105*, R. Zhu *et al.*, Eds. Springer-Verlag Berlin Heidelberg, 2010, pp. 169–176.

[66] I. Red Hat, "Red hat enterprise linux virtual server administration," 2004, accessed: 2024-06-23. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/4/html/virtual_server_administration/s1-lvs-scheduling-vsa#s2-lvs-sched-VSA

[67] T. Balharith and F. Alhaidari, "Round robin scheduling algorithm in cpu and cloud computing: A review," in *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*, 2019, pp. 1–7.

APÉNDICE A
Creación de una máquina virtual

En este apéndice se muestran las capturas de pantalla sobre el proceso de creación y configuración de una máquina virtual, con el fin de complementar la explicación que se incluye en la Sección 5.1.

A.1 Pasos a seguir en Proxmox

Se presentan, a continuación, los pasos seguidos en las pestañas principales de configuración del proceso de creación de maquinas virtuales en *Proxmox*. El contenido de la Figura A.1 trata sobre la pestaña de selección del sistema operativo. En ella se debe especificar dónde se encuentra la imagen y el nombre de la imagen. Los demás campos disponibles no se han modificado.

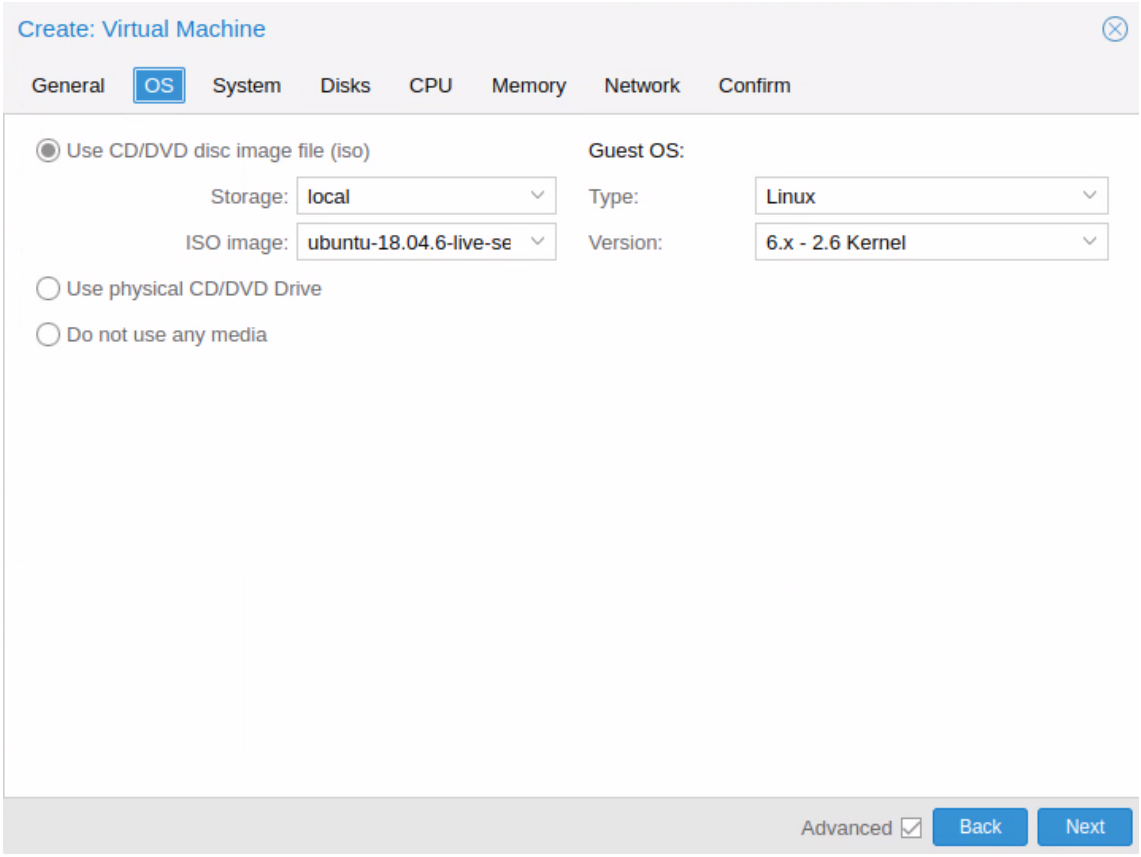


Figura A.1: Creación máquina virtual 1

La Figura A.2 ilustra la configuración de las características del procesador de una máquina virtual. En esta pestaña se define el número de núcleos que va a tener, así como otras características, que se han mantenido en los valores predeterminados.

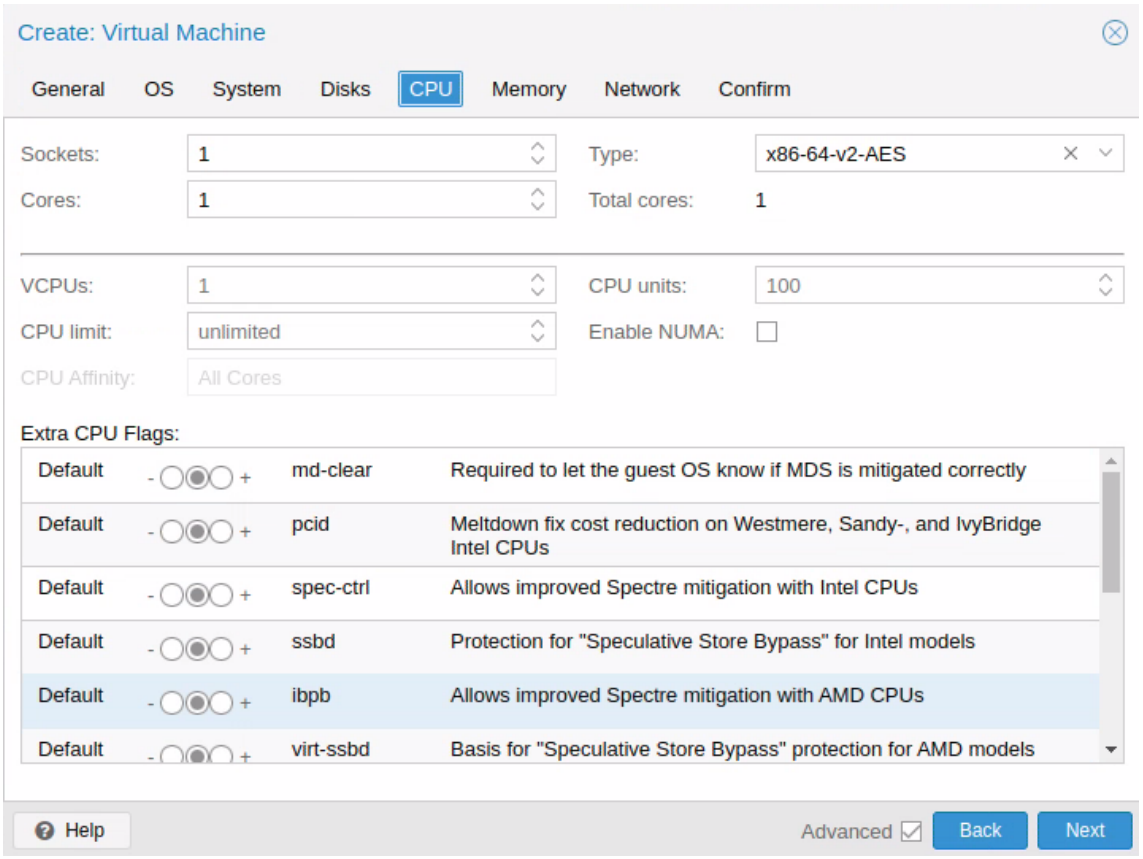


Figura A.2: Creación máquina virtual 2

En la Figura A.3 se presenta la pestaña donde se configura las cantidades mínima y máxima de memoria principal que tendrá una máquina virtual. Para las máquinas virtuales creadas en este trabajo, los valores de los dos campos son establecidos a 8GB.

El resto de pestañas que no se han presentado con capturas de pantallas son de menor importancia para el trabajo, a excepción de la pestaña disco, donde se estable la cantidad de memoria secundaria que tiene una máquina virtual. Sin embargo, esta característica que no afecta al funcionamiento de las máquinas ya que se proporciona un espacio suficiente para que no se creen situaciones indeseadas.

A.2 Instalación del sistema operativo

En este apartado se muestran algunas capturas de pantalla sobre el proceso de instalación del sistema operativo *Ubuntu* después de la creación de una máquina virtual. Vienen a complementar también las capturas de pantalla y explicación de la Sección 5.1. Se muestran las capturas donde se configura el idioma del proceso (Figura A.4), las particiones del sistema de ficheros (Figura A.5), la pestaña de creación de usuario (A.6) y la pestaña del progreso de instalación (A.7).

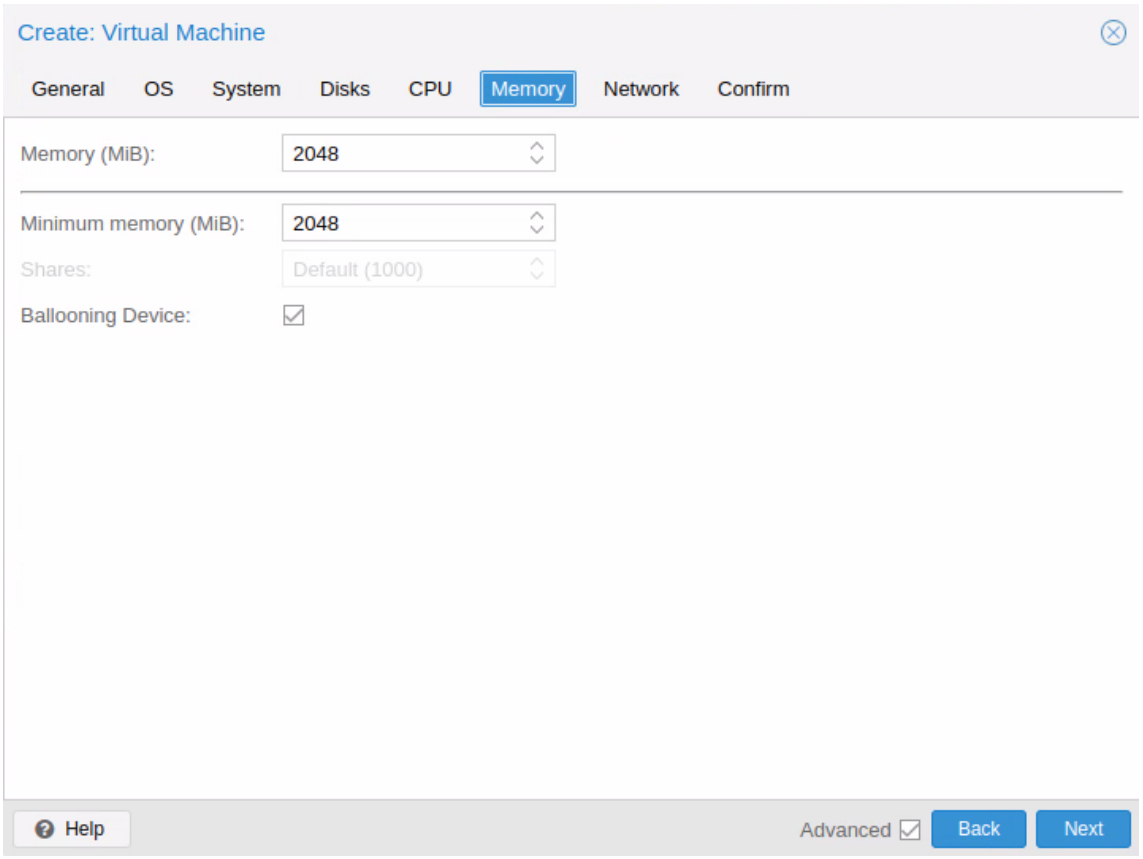


Figura A.3: Creación máquina virtual 3

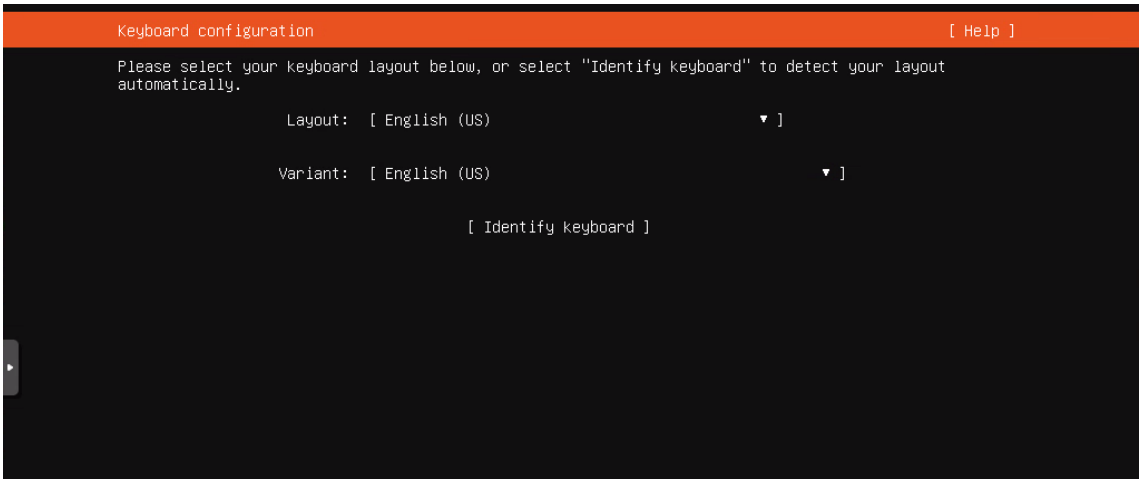


Figura A.4: Instalación de *Ubuntu* 1

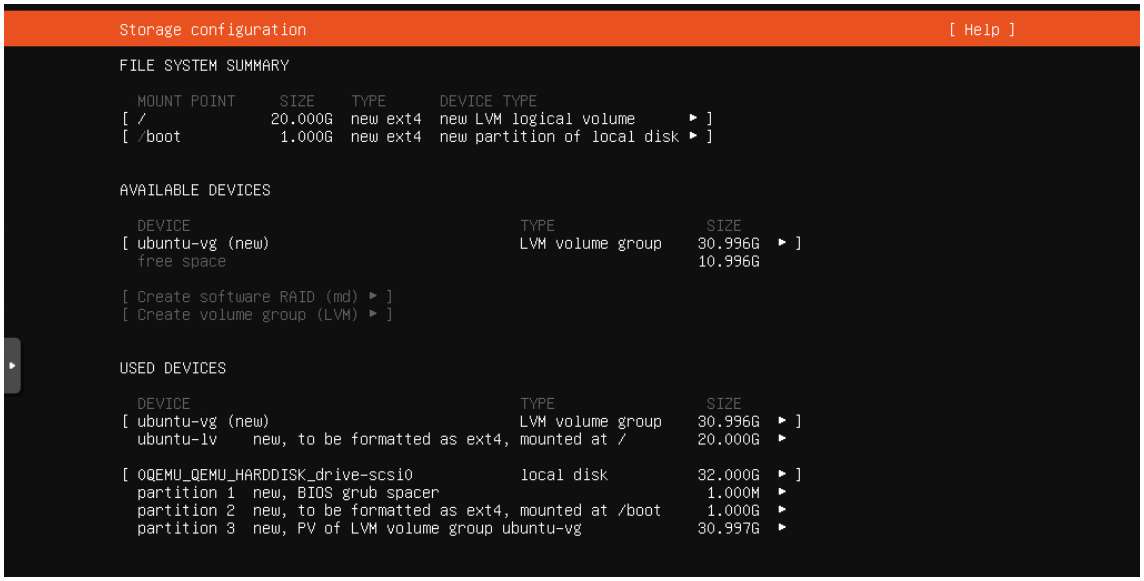


Figura A.5: Instalación de *Ubuntu* 2

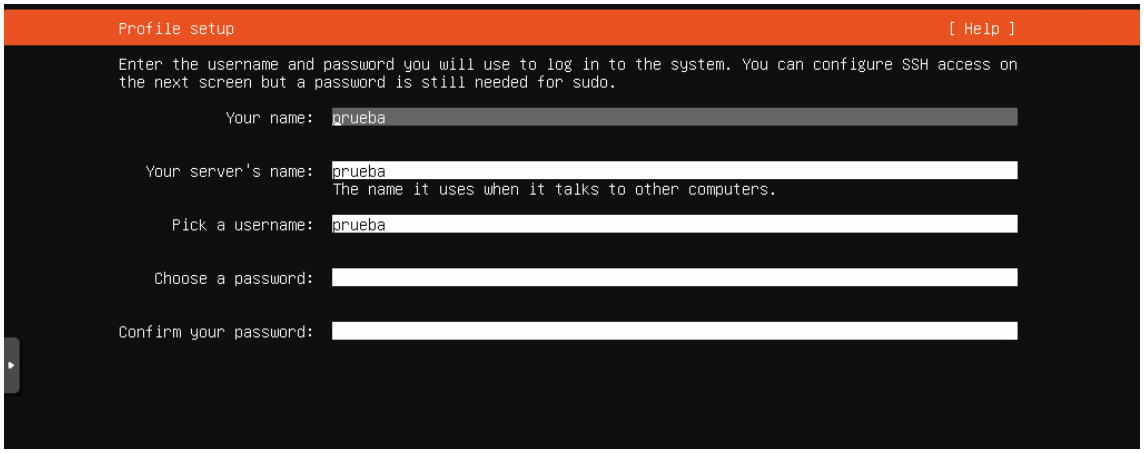


Figura A.6: Instalación de *Ubuntu* 3

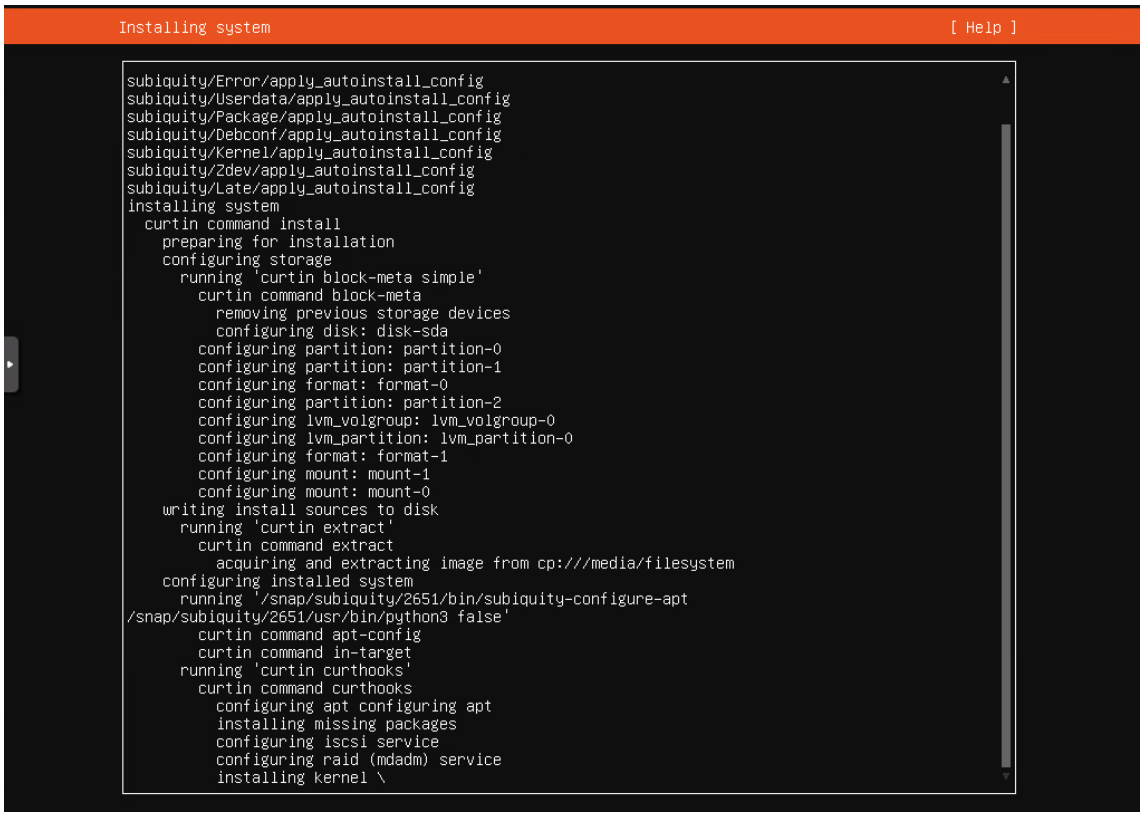


Figura A.7: Instalación de *Ubuntu* 4

APÉNDICE B
Fichero de configuración de red

En este apéndice se presentan los ficheros de configuración de red de las máquinas virtuales utilizadas para la evaluación de la infraestructura experimental y las cargas de trabajo. En ellas se detallan información como las direcciones de cada máquina, la puerta de enlace, los servidores de nombre, entre otras.

```
network:
  ethernets:
    ens18:
      addresses:
      - 192.168.14.80/24
      gateway4: 192.168.14.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
  version: 2
```

Figura B.1: Fichero de configuración de red del cliente 1

```
network:
  ethernets:
    ens18:
      addresses:
      - 192.168.14.81/24
      gateway4: 192.168.14.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
  version: 2
```

Figura B.2: Fichero de configuración de red del cliente 2

```
network:
  ethernets:
    ens18:
      addresses:
      - 192.168.14.82/24
      gateway4: 192.168.14.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
  version: 2
```

Figura B.3: Fichero de configuración de red del cliente 3

```
network:
  ethernets:
    ens18:
      addresses:
      - 192.168.3.111/24
      gateway4: 192.168.3.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
    ens19:
      addresses:
      - 192.168.14.111/24
  version: 2
```

Figura B.4: Fichero de configuración de red del servidor 1

```
network:
  ethernets:
    ens18:
      addresses:
      - 192.168.10.112/24
      gateway4: 192.168.10.1
      nameservers:
        addresses:
        - 8.8.8.8
        search: []
    ens19:
      addresses:
      - 192.168.14.112/24
  version: 2
```

Figura B.5: Fichero de configuración de red del servidor 2

APÉNDICE C

Código fuente de las modificaciones de *Tailbench++*

En este apéndice se incluye el código completo de las funciones más relevantes del código de Tailbench que han sido modificadas para adaptar estas aplicaciones a las nuevas funcionalidades. Concretamente, estas funciones son `recvReq`, que se muestra en el Bloque de código C.1; la función `checkNewClient`, que se presenta en el Bloque C.2; y la función `startReq`, que se detalla en el Bloque C.3. La primera función originalmente se encargaba de recibir las peticiones de los clientes, pero ahora es capaz de aceptar nuevas conexiones, haciendo uso de la segunda función que monitoriza la existencia de clientes entrantes. Asimismo, la tercera función, la emplean los clientes para generar nuevas peticiones, con las modificaciones, es capaz de variar la tasa de petición por segundo dinámicamente.

```
size_t NetworkedServer::recvReq(int id, void **data)
{
    pthread_mutex_lock(&recvLock);

    bool success = false;
    Request *req;
    int fd = -1;

    while (!success)
    {
        if (clientFds.size() == 0)
        {
            // SERVER NEW CONNECTIONS //
            int maxFd = listenFd;
            fd_set readSet;
            FD_ZERO(&readSet);
            FD_SET(listenFd, &readSet);
            int ret = select(maxFd + 1, &readSet, nullptr, nullptr, nullptr);
            if (ret == -1)
            {
                std::cerr << "select() failed: " << strerror(errno) << std::
                    endl;
                pthread_mutex_unlock(&recvLock);
                exit(-1);
            }

            if (checkNewClient(&readSet))
            {
                recvClientHead = 0;
            }
            // ---------------------- //

```

```
32              continue;
33          }
34
35          int maxFd = -1;
36          fd_set readSet;
37          FD_ZERO(&readSet);
38
39          // SERVER NEW CONNECTIONS //
40          //add listening socket to the set
41          FD_SET(listenFd, &readSet);
42          maxFd = listenFd;
43
44          for (int f : clientFds)
45          {
46              FD_SET(f, &readSet);
47              if (f > maxFd)
48                  maxFd = f;
49          }
50
51          int ret = select(maxFd + 1, &readSet, nullptr, nullptr, nullptr);
52          if (ret == -1)
53          {
54              std::cerr << "select() failed: " << strerror(errno) << std::endl;
55              exit(-1);
56          }
57
58          //if new client is detected, actualiza fd_Set
59          if (checkNewClient(&readSet))
60          {
61              FD_ZERO(&readSet);
62              maxFd = -1;
63              for (int f : clientFds)
64              {
65                  FD_SET(f, &readSet);
66                  if (f > maxFd)
67                      maxFd = f;
68              }
69              ret = select(maxFd + 1, &readSet, nullptr, nullptr, nullptr);
70              if (ret == -1) {
71                  std::cerr << "select() failed: " << strerror(errno) << std::
                        endl;
72                  pthread_mutex_unlock(&recvLock);
73                  exit(-1);
74              }
75          }
76          // ---------------------- //
77
78          fd = -1;
79
80          for (size_t i = 0; i < clientFds.size(); ++i)
81          {
82              size_t idx = (recvClientHead + i) % clientFds.size();
83              if (FD_ISSET(clientFds[idx], &readSet))
84              {
85                  fd = clientFds[idx];
86                  break;
87              }
88          }
89
90          recvClientHead = (recvClientHead + 1) % clientFds.size();
91
92          if (fd == -1)
93          {
94              continue;
```

```
95          }
96
97          int len = sizeof(Request) - MAX_REQ_BYTES; // Read request header first
98
99          req = &reqbuf[id];
100         int recvd = recvfull(fd, reinterpret_cast<char *>(req), len, 0);
101
102         success = checkRecv(recvd, len, fd);
103         if (!success)
104             continue;
105
106         recvd = recvfull(fd, req->data, req->len, 0);
107
108         success = checkRecv(recvd, req->len, fd);
109         if (!success)
110             continue;
111     }
112
113     uint64_t curNs = getCurNs();
114     reqInfo[id].id = req->id;
115     reqInfo[id].startNs = curNs;
116     activeFds[id] = fd;
117
118     *data = reinterpret_cast<void *>(&req->data);
119
120     if (!start_file)
121     {
122         start_file = true;
123         time_t rawtime;
124         struct tm *timeinfo;
125         time(&rawtime);
126         timeinfo = localtime(&rawtime);
127         fprintf(stderr, "\n[SERVER] Starting to reply clients -- %s", asctime(
                timeinfo));
128         if (!start_file_name.empty())
129         {
130             std::ofstream file{start_file_name};
131         }
132     }
133
134     pthread_mutex_unlock(&recvLock);
135
136     return req->len;
137 }
```

Listing C.1: Método 'recvReq' en la clase 'NetworkedServer'

```
1  bool NetworkedServer::checkNewClient(fd_set *fdSet)
2  {
3      struct sockaddr_storage clientAddr;
4      socklen_t clientAddrSize;
5      std::cout << "Checking new connection" << std::endl;
6      bool res = false;
7      if (FD_ISSET(listenFd, fdSet))
8      {
9          clientAddrSize = sizeof(clientAddr);
10         memset(&clientAddr, 0, clientAddrSize);
11
12         int clientFd = accept(listenFd,
13                               reinterpret_cast<struct sockaddr *>(&clientAddr),
14                               &clientAddrSize);
15
16         if (clientFd == -1)
17         {
```

```
            std::cerr << "accept() failed: " << strerror(errno) << std::endl;
            return false;
        }

        int nodelay = 1;
        if (setsockopt(clientFd, IPPROTO_TCP, TCP_NODELAY,
                       reinterpret_cast<char *>(&nodelay), sizeof(nodelay)) ==
                           -1)
        {
            std::cerr << "setsockopt(TCP_NODELAY) failed: " << strerror(errno)
                      << std::endl;
            close(clientFd);
            return false;
        }

        clientFds.push_back(clientFd);
        res = true;
    }
    return res;
}
```

Listing C.2: Método 'checkNewClient' en la clase 'NetworkedServer'

```
Request *Client::startReq() {

    uint64_t startNs = getCurNs();

    if (status == INIT)
    {
        pthread_barrier_wait(&barrier); // Wait for all threads to start up
        pthread_mutex_lock(&lock);

        if (!dist)
        {

            uint64_t curNs = getCurNs();
            if (!qpsVar)
            {
                dist = new ExpDist(lambda, seed, curNs);
            }
            else
            {
                dist = new ExpDist(qpsIni, seed, curNs);
            }
            status = WARMUP;
            pthread_barrier_destroy(&barrier);
            pthread_barrier_init(&barrier, nullptr, nthreads);
        }

        pthread_mutex_unlock(&lock);
        pthread_barrier_wait(&barrier);
    }
    if (status == ROI && qpsVar)
    {
        // COMPROBAR  E INCREMENTAR QPS
        pthread_mutex_lock(&lock);
        if (qpsCounter == qpsInterval)
        {
            qpsCounter = 0;

            if(qpsVar == 1) {
                qpsIni += qpsStep; // 10 ---- 100
            } else if(qpsVar == 2) {
                double newqps = listQPS[idxListQPS%listQPS.size()];
```

```
42                  double ratio = newqps/qpsIni;
43                  if(newqps < qpsIni) qpsInterval = static_cast<int>(qpsInterval*
                        ratio) + 1;
44                  else qpsInterval = (qpsInterval*ratio); //interval is
                        proportional to qps
45                  qpsIni = newqps;
46                  idxListQPS++;
47
48              }
49
50              dist = new ExpDist(qpsIni, seed, getCurNs());
51              fprintf(stderr, "\nQPS changed to %f\n", qpsIni / 1e-9);
52              fprintf(stderr, "\nQPS interval changed to %d\n", qpsInterval);
53          }
54          else
55          {
56              qpsCounter++;
57          }
58          pthread_mutex_unlock(&lock);
59      }
60      Request *req = new Request(); // It can be placed before the lock as it is
            independent of the threads and takes some not-negligible time
61      req->startNs = startNs;
62
63      pthread_mutex_lock(&lock);
64
65      req->startNs_lock_ad = getCurNs(); // Time when the lock is adquired
66      size_t len = tBenchClientGenReq(&req->data);
67      req->len = len;
68      req->id = startedReqs++;
69      req->genNs = dist->nextArrivalNs();
70      inFlightReqs[req->id] = req;
71      gen_requests++;
72      if (req->genNs > req->startNs)
73      {
74          timely_gen_requests++;
75      }
76      req->startNs_lock_re = getCurNs(); // Time when the lock is released
77      gen_req_ns += req->startNs_lock_re - req->startNs_lock_ad;
78
79      pthread_mutex_unlock(&lock);
80
81      if (req->genNs < req->startNs)
82      {
83          req->late = true;
84      }
85      else
86      {
87          req->late = false;
88      }
89
90      uint64_t curNs = getCurNs();
91      if (curNs < req->genNs)
92      {
93          sleepUntil(std::max(req->genNs, curNs + minSleepNs));
94          total_sleep_ns += std::max(req->genNs - curNs, minSleepNs);
95      }
96      return req;
97  }
```

Listing C.3: Método 'startReq' de la clase 'Client'

Objetivos de Desarrollo Sostenible

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				x
ODS 2. **Hambre cero.**				x
ODS 3. **Salud y bienestar.**				x
ODS 4. **Educación de calidad.**				x
ODS 5. **Igualdad de género.**				x
ODS 6. **Agua limpia y saneamiento.**				x
ODS 7. **Energía asequible y no contaminante.**				x
ODS 8. **Trabajo decente y crecimiento económico.**			x	
ODS 9. **Industria, innovación e infraestructuras.**	x			
ODS 10. **Reducción de las desigualdades.**				x
ODS 11. **Ciudades y comunidades sostenibles.**				x
ODS 12. **Producción y consumo responsables.**		x		
ODS 13. **Acción por el clima.**				x
ODS 14. **Vida submarina.**				x
ODS 15. **Vida de ecosistemas terrestres.**				x
ODS 16. **Paz, justicia e instituciones sólidas.**				x
ODS 17. **Alianzas para lograr objetivos.**				x

Table 1: Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Los Objetivos de Desarrollo Sostenible (ODS) son unas metas establecidas por las Naciones Unidas para erradicar la pobreza, proteger el planeta y asegurar la prosperidad de la humanidad como parte de una nueva agenda de desarrollo sostenible, y que deben ser cumplidas para el año 2030. Por este motivo, sería bastante interesante relacionar el presente TFG con estos objetivos. Se presenta a continuación los objetivos que son beneficiados con el trabajo.

- ODS 8. **Trabajo decente y crecimiento económico**. El propósito de esta meta es impulsar el crecimiento sostenible e inclusivo, y trabajo decente para todo el mundo. El presente trabajo ayuda a conseguir este objetivo, porque la implementación de un entorno experimental permitiría llevar a cabo estudios e investigaciones que innovarían los sistemas informáticos actuales, creando mayores oportunidades de empleo en el sector tecnológico, y como consecuencia de ello, una mejora de la situación económica para las personas.

- ODS 9. **Industria, innovación e infraestructuras**. Este objetivo pretende desarrollar unas infraestructuras sólidas, impulsar una industrialización sostenible y estimular la innovación. Esta meta es la más relacionada con el presente proyecto, porque gracias al diseño e implantación de una infraestructura experimental y cargas de trabajos realistas, se permite realizar estudios que contribuyan a la construcción de unas infraestructuras informáticas robustas e innovadoras que benefician no solo la industria, sino también otros sectores.

- ODS 12. **Producción y consumo responsables**. Esta última meta persigue un consumo y producción responsable para combatir los recursos limitados del planeta. El contenido de este trabajo aborda este objetivo creando una infraestructura experimental flexible utilizando el balanceo de carga y la virtualización, estas técnicas permiten aprovechar de manera eficiente los recursos informáticos. Asimismo, como otro ejemplo, se podría realizar investigaciones en este entorno de pruebas sobre temáticas que optimicen el consumo energético en las infraestructuras de producción.

ETS Enginyeria Informàtica
Camí de Vera, s/n, 46022, València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es