UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Implementación de un balanceador de cargas en una red SDN mediante el controlador Ryu

TESIS

Que para obtener el título de

Ingeniero en Telecomunicaciones

P R E S E N T A N

Arreola González Cynthia Estefanía

Jimenez Herrera Alex

DIRECTOR DE TESIS

Dr. Luis Francisco García Jiménez

Ciudad Universitaria, Cd. Mx., 2022.

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

Quisiera agradecerles a mis padres por el infinito apoyo y comprensión que me han brindado a lo largo de mi vida, ya que no solamente me ha impulsado a salir adelante, sino que me ha ayudado en todo lo que me propongo. Jamás podré compensarles por todo el amor que me han dado, pero daré lo mejor de mí para demostrarles cada día que su esfuerzo ha valido la pena.

Agradezco a mis docentes. En especial a mi tutor el Dr. Luis Francisco García Jiménez y al Prof. Víctor Damián Pinilla Morán por su ayuda, paciencia y dedicación. Llevaré siempre conmigo todas sus enseñanzas, ya que a pesar de mis dudas y dificultades me apoyaron, demostrándome no solo su calidad académica sino también la humana.

Gracias también a los amigos que me han acompañado en este camino, que sus metas también se cumplan y su felicidad perdure por sobre cualquier obstáculo. Porque sin ellos quizá me hubiera vuelto loca desde el comienzo, y todos esos maravillosos momentos que pasamos juntos seguirán motivándome a seguir.

A mi compañero de tesis por su comprensión y paciencia, de corazón espero que tenga mucho éxito. Que sepa que sin él no hubiera sido posible tener este trabajo. Estoy segura de que se destacará en todo lo que haga, y le deseo lo mejor.

Y finalmente, agradezco el apoyo brindado por parte del proyecto DGAPA-PAPIIT IA102822, a la UNAM y a la Facultad de Ingeniería por ser una segunda casa durante estos años, por permitirme conocer a todas estas personas maravillosas y darme los pilares que me apoyarán en mi desarrollo profesional.

-Cynthia Estefanía Arreola González

Agradezco profundamente en primer lugar a mi familia, que me ha apoyado en todo momento sin importar las circunstancias desde el día que comencé mi educación. A mis papás, mis abuelos y hermanos, por recordarme y demostrarme que los límites los pone uno mismo, en donde es posible lograr tus metas y ser mejor cada día si en verdad te esfuerzas por lograrlo.

A los amigos y compañeros con los que he compartido aula, algunos desde el primer día de la carrera y algunos otros que, siempre me han apoyado y me hicieron seguir creyendo en mi cuando me costaba trabajo hacerlo. Así como a las personas que fui conociendo en el camino y que se convirtieron en una otra gran fuente de apoyo e inspiración.

De la misma forma, agradezco a todos los profesores de la carrera, en especial a mi tutor el Dr. Luis Francisco García Jiménez por el apoyo, consejos y paciencia durante la elaboración del trabajo, así como por la calidad de conocimientos brindados en las materias que imparte. De igual manera, a los sinodales por sus recomendaciones y comentarios para realizar un mejor trabajo.

Finalmente, agradezco por parte del proyecto DGAPA-PAPIIT IA102822, a la Facultad de Ingeniería y a la UNAM por ser una segunda casa durante todo este trayecto, así como por permitirme compartir espacios con tan grandes personas, compañeros y docentes que hicieron un camino más agradable.

-Alex Jiménez Herrera

Índice general

Resumen **1**

1. Introducción **2**
1.1. Planteamiento del problema 2
1.2. Objetivo 3
1.2.1. Objetivos particulares 3
1.3. Hipótesis 3
1.4. Metodología 3
1.5. Contribución 4
1.6. Estructura de la tesis 4

2. Antecedentes **5**
2.1. Estado del arte 6

3. Herramientas de desarrollo **8**
3.1. Componentes físicos 8
3.2. Protocolo OpenFlow 9
3.2.1. Iniciación del canal OpenFlow 10
3.2.2. Tablas OpenFlow y Flow Entries 10
3.2.3. Controlador 10
3.3. Open vSwitch 12
3.4. Instalación del Sistema Operativo Raspbian y configuraciones previas 12
3.4.1. Asignación de direcciones IP para clientes y servidores 12
3.4.2. Instalación de Open vSwitch en las Raspberry 13
3.5. Configuración inicial de los Switches 14
3.5.1. Switch 1 14
3.5.2. Switch 2 14
3.5.3. Inicialización de los *switches* 14
3.5.4. Instalación del controlador Ryu 15
3.5.5. Inicialización del controlador 15
3.5.6. Conexión del controlador con el switch 16
3.6. Configuración de los clientes y servidores 17
3.6.1. iPERF 17

4. Implementación **19**
4.1. Topología implementada 19
4.2. Comunicación cliente-servidor 20
4.3. Balanceadores de carga 23
4.3.1. Balanceador Aleatorio 23
4.3.2. Balanceador Round-robin 23

5. Pruebas y desempeño **25**
5.1. Prueba Aleatoria 25
5.2. Prueba Round-robin 29
5.3. Comparación de resultados 32

6. Conclusiones **36**
6.1. Conclusiones generales 36

Apéndices **37**

A. Datos balanceador random **37**

B. Datos balanceador Round-robin **40**

C. Código en Python implementado en el controlador para el balanceador Round-robin. **43**

D. Programa en Python implementado en el controlador para el balanceador aleatorio **47**

Índice de figuras

3.1. Tarjetas *Raspberry* y adaptador USB-Ethernet. 8
3.2. Capas de una SDN [1]. 9
3.3. Inicialización del controlador. 16
3.4. Conexión desde el controlador. 16
3.5. Conexión desde el switch. 17
3.6. Servidor escuchando peticiones. 18
3.7. Archivo de salida. 18

4.1. Topología implementada. 20
4.2. Flujo de las peticiones. 20
4.3. Balanceador con Openflow. 21
4.4. Diagrama de flujo de la aplicación. 22
4.5. Balanceador de carga. 23
4.6. Balanceador Round Robin. 24

5.1. Peticiones recibidas vs. peticiones totales. 26
5.2. Porcentaje de peticiones perdidas. 27
5.3. Throughput. 28
5.4. Velocidad de transmisión del algoritmo Aleatorio. 28
5.5. Peticiones del algoritmo Round-robin. 29
5.6. Porcentaje de paquetes perdidos del algoritmo Round-robin. 31
5.7. Throughput Round-robin. 32
5.8. Velocidad de transmisión del algoritmo Round-robin. 32
5.9. Peticiones recibidas. 33
5.10. Pérdidas de paquetes. 34
5.11. Throughput total. 34
5.12. Velocidad de transmisión. 35

Índice de tablas

3.1. Tabla de direcciones IP. 13
3.2. Opciones a utilizar en iPerf. 17

4.1. Tabla direcciones. 21

5.1. Promedios de peticiones para Random. 26
5.2. Server 1. 27
5.3. Promedios de peticiones Round Robin. 30
5.4. Promedios generales Round Robin. 31

A1. Server 2. 37
A2. Server 3. 38
A3. Random Promedios. 39

B1. Server 1. 40
B2. Server 2. 41
B3. Server 3. 42

Resumen

Las Redes Definidas por Software (*SDN*) proveen un método fácil de configuración y administración de dispositivos (virtuales o físicos) de manera centralizada. Este enfoque al contrario de las redes tradicionales, donde se utilizan dispositivos de hardware dedicados (*routers* y *switches*) altamente costosos, es flexible y escalable en tiempo real, debido a que las funciones de control y de conmutación de paquetes se disocian, permitiendo así una operación y administración de la red más inteligente. En este sentido, las funciones de control pueden implementarse en uno o varios dispositivos descentralizados (mediante software) que gestionan las reglas de distribución y conmutación del tráfico. Esta característica hace que las SDN sean una evolución con respecto a las redes tradicionales, especialmente porque no dependen de un hardware propietario cuyas licencias suelen ser costosas.

La principal tarea de un servidor, es atender la mayor parte de peticiones posibles para garantizar una calidad de servicio. Sin embargo, existen situaciones que pueden degradar su funcionamiento, como la falta de memoria o capacidad de procesamiento conforme crece la cantidad de tareas. Es aquí donde las SDN tienen una ventaja relevante en comparación al uso de las redes tradicionales, gracias al hecho de contar con un controlador que puede analizar el rendimiento de la red y evitar caídas en los servicios mediante un balanceador de cargas. Esto es posible gracias a que todos los dispositivos de red dentro de una SDN pueden ajustar y distribuir su carga dinámicamente, evitando que se saturen.

En esta tesis, en contraste a trabajos previos donde se utiliza el emulador de redes *Mininet*, se propone el uso de dispositivos físicos (*Raspberries*) para el estudio e implementación de balanceadores de carga en una SDN. Específicamente, se implementan dos algoritmos de balanceo: *Random* y *Round-robin*, los cuales son programados en *switches* con soporte del protocolo *OpenFlow* para determinar cuál de ellos permite un mejor balance de tráfico en una topología dada.

Capítulo 1

Introducción

Desde los primeros días de Internet, la cual fue pensada como un conjunto de dispositivos de comunicación para llevar la información de un punto a otro, se ha incrementado exponencialmente el número de servicios ofrecidos, desde la simple transferencia de un archivo mediante el protocolo FTP (File Transfer Protocol), hasta la solicitud de acceso a contenido en servidores de *streaming* o bien, conexiones en la nube para la administración remota de una empresa que se encuentra del otro lado del mundo. Para que estos servicios puedan ser utilizados, se hace uso de sistemas que encaminan la información a su destino llamados *switches* y *routers*, que se encargan principalmente de elegir los caminos adecuados para que cada paquete de información llegue a su destino en la menor cantidad de saltos.

Actualmente, el principal problema de los *switches* y *routers* es que el plano de control (responsable de llenar las tablas de encaminamiento y elegir protocolos) y el plano de datos (conmutación de paquetes de una interfaz a otra) se encuentran en el mismo dispositivo. Esto los hace poco flexibles ya que contienen una programación previa que no se adapta a los cambios propios de la red y que solo son capaces de seguir políticas individualmente. Además, suelen ser dispositivos a los que no es posible realizarles modificaciones de software debido a los costos elevados de licenciamiento.

Por otro lado, cuando una gran cantidad de usuarios se conectan a un servicio de red es posible que los dispositivos lleguen a su capacidad máxima y las peticiones se atiendan de forma deficiente. Esto puede deberse a dos principales causas; la incapacidad de los servidores de atender las peticiones por falta de memoria, o bien, la saturación de la red debido a rutas estáticas de encaminamiento. Ante esta problemática, surge un nuevo paradigma llamado Redes Definidas por Software (SDN), en donde el plano de control y el plano de datos se disocian. De esta manera, los dispositivos de red conservan el plano de datos, mientras que el plano de control puede encontrarse de forma descentralizada de cada dispositivo. Ahora, este dispositivo llamado controlador se encarga de gestionar a todos los dispositivos con los que cuenta la red. Gracias a esto las redes son más flexibles y escalables, ya que tienen un agente especializado que toma las decisiones de encaminamiento desde una perspectiva global y permite tomar mejores decisiones.

Actualmente, la necesidad de agilizar los procesos de encaminamiento ha impulsado la creación de nuevos algoritmos que distribuyen el tráfico entre los servidores conocidos como *balanceadores de carga*. Las SDN, gracias a su gestión centralizada, pueden ayudar a que el tráfico se distribuya de forma más equitativa entre múltiples servidores con diferentes ubicaciones geográficas y evitar un congestionamiento en la red.

1.1. Planteamiento del problema

Desde 1990, las redes se han convertido en una necesidad para los proveedores de servicios a nivel mundial. Por ello, la constante mejora en la dinámica de las redes para disminuir el error humano y aumentar el tráfico de datos se ha incrementado cada vez más.

Las peticiones a los servidores que hospedan las páginas que accedemos diariamente,

se hace desde un navegador que en pocas décimas de segundo muestra la información deseada. Sin embargo, del lado del servidor el proceso es más complicado debido a que los dispositivos de red envían la información hacia los servidores de dominio que transforman los nombres como *google.com* en una dirección IP, ya sea en su versión *"IPV4"*, o *"IPV6"*. Además, los servicios de páginas que esperan gran cantidad de tráfico tienen servidores ubicados en diferentes puntos geográficos. Por lo que existe un proceso que lleva a los usuarios a las zona geográfica más cercanas o menos utilizadas. Este proceso es conocido como un balanceador de carga o de tráfico, el cual debe tomar en cuenta el número de solicitudes, o bien, debe implementar algún algoritmo inteligente de balanceo de tráfico que tome factores como cantidad de memoria, capacidad de hilos, número de *sockets*, saturación de caminos, etc.

1.2. Objetivo

Analizar dos algoritmos de balanceo de carga (Aleatorio y Round-robin) en una red SDN, y comparar cuál de ellos se adapta mejor en una topología de red hecha con dispositivos físicos que soportan el protocolo *OpenFlow* ante pruebas de estrés.

1.2.1. Objetivos particulares

- Instalar y configurar el protocolo *Openflow* en los dispositivos de red.
- Conectar los dispositivos de red bajo una topología definida.
- Instalar y configurar el controlador Ryu.
- Implementar los algoritmos de balanceo de carga en el controlador Ryu.
- Analizar el desempeño del balanceador de carga Aleatorio.
- Analizar el desempeño del balanceador de carga Round-robin.
- Comparar resultados entre ambos algoritmos.

1.3. Hipótesis

Es posible implementar algoritmos de balanceo de carga mediante un controlador SDN que evite la saturación en los servidores bajo una topología dada.

1.4. Metodología

El desarrollo de este trabajo se divide en dos etapas:

- Primero, se instala el sistema operativo Raspbian-Linux en los microprocesadores (*switches*), posteriormente, se les da soporte del protocolo Openflow. También se instala el controlador Ryu en un dispositivo con mayor capacidad de procesamiento. Finalmente, se implementan los algoritmos de balanceo de carga en una topología definida.

- Segundo, se realizan diferentes experimentos para medir el desempeño entre los dos algoritmos de balanceo con el fin de determinar las capacidades bajo estrés de cada algoritmo. Finalmente, se concluye cuál balanceador tiene mejor desempeño para la topología utilizada.

1.5. Contribución

- Implementación de una SDN a través de *switches* virtuales instalados en tarjetas *Raspberry* Pi utilizando Ryu como controlador.
- Análisis de algoritmos de balanceo en una SDN; Aleatorio y *Round-robin*.

1.6. Estructura de la tesis

La estructura de este proyecto de investigación se divide de la siguiente forma:

- El capítulo 2 presenta una revisión de los algoritmos de balanceo de cargas, su uso en las SDN y los trabajos de investigación relacionados con esta tecnología.
- El capítulo 3 presenta las herramientas necesarias para la infraestructura del proyecto, así como los pasos de instalación del sistema operativo de cada microcontrolador, del controlador y el software de conmutación para los switches.
- El capítulo 4 presenta los algoritmos de balanceo, desde su descripción hasta la diferencia entre los algoritmos elegidos para este trabajo.
- El capítulo 5 presenta los resultados obtenidos bajo diferentes escenarios de pruebas.
- El capítulo 6 presenta las conclusiones finales, así como la verificación de la hipótesis y las perspectivas de investigación.

Capítulo 2

Antecedentes

En este capítulo, primero se presentan los conceptos básicos utilizados. Posteriormente, se muestran algunos trabajos relacionados y la relevancia de este proyecto de tesis.

TCP (*Transmission Control Protocol*) es un protocolo que garantiza el intercambio de datos entre dos puntos de comunicación. Para ello, se adiciona tráfico de control que corrige los posibles errores. A este mecanismo se le conoce como un canal orientado a conexión. Mientras que el protocolo UDP (*User Datagram Protocol*) es un mecanismo que no garantiza el intercambio total de información, también conocido como no orientado a conexión. Sin embargo, provee de mayor velocidad de transmisión y menor carga en la red [2]. Es importante recalcar que no hay comparación entre ambos protocolos ya que depende del objetivo o uso.

Un *framework* es un sistema que permite agilizar la creación de un proyecto, un marco o plantilla que esquematiza las tareas a realizar. Los *frameworks* son una técnica que provee estructura a la metodología de trabajo y modela un esqueleto de una aplicación de manera personalizada y reutilizable en las aplicaciones. En [3] lo definen como una técnica de reutilización que representa una arquitectura de software que modela las relaciones generales de las entidades del dominio. De tal manera que, en resumen, se utilizan con el objetivo de facilitar el desarrollo de aplicaciones, software o proyectos.

La calidad de servicio (QoS) es una medida de la eficiencia de los sistemas de redes. Por ejemplo, en [4], se menciona que las características más importantes a la calidad de servicio son el rendimiento, la fiabilidad, escalabilidad, capacidad, robustez, manejo de excepciones, precisión, integridad, accesibilidad, disponibilidad, interoperabilidad y seguridad. Los cuales engloban requisitos no funcionales de un servicio de comunicaciones, pero que son relevantes al momento de catalogar si un servicio es útil para las necesidades del usuario o no. Específicamente, para este trabajo de tesis, se toma el concepto de *Throughput*, el cual se define en [5] como la tasa promedio de éxito en la entrega de un mensaje sin considerar los encabezados del paquete, ACK, retransmisiones, etc., por lo que quedan solo los datos puros del usuario. Esta medición cuantifica el rendimiento de la red y establece la capacidad de la misma.

La velocidad de transmisión, se define como la cantidad de datos (*bits*) que se pueden llevar de un extremo a otro en los diferentes medios. Depende de la necesidad de transmisión que tenga la comunicación entre dispositivos y la cantidad de dispositivos conectados. Por ejemplo, en [6] lo definen como la cantidad de información que se puede transmitir en un momento dado a lo largo de una línea de datos.

Los balanceadores de carga se definen como aquellos dispositivos que implementan algoritmos capaces de distribuir el tráfico de red entrante hacia un grupo o conjunto de servidores a los cuales comúnmente se les conoce como *server farm* o *server pool*. Los sitios que soportan grandes cantidades de tráfico contratan proveedores que ofrecen el servicio de balanceo y puede manejarse como un servicio adicional si se tienen servidores alojados en la

nube. Uno de los proveedores más grandes es *Amazon Web Services* mediante *Elastic Load Balancing*, el cual puede ser un balanceador de carga de aplicaciones o de red. Si bien, puede tratarse de una opción, dicha solución es de un proveedor privado y lo que sucede detrás de dicho balanceador solo lo conocen *Amazon* [1]. Para fines de este proyecto de tesis, se desarrolló un balanceador de carga *Open Source* o de código abierto (accesible al público), escalable, el cual es capaz de distribuir la carga de red.

El controlador SDN es un dispositivo centralizado ya sea físico o virtual que se comunica con los dispositivos de red conocidos como *switches*, los cuales ejecutan las reglas y políticas de reenvío de tráfico. De esta manera, permite a los dispositivos de red el envío o retransmisión de información [7]. Estos dispositivos actúan de tal manera que los dispositivos puedan compartir información y se puedan comunicar entre sí. Para fines de este proyecto, se utilizan *switches* administrables [8], los cuales son programados para manejar de manera dinámica el tráfico de red. Sin embargo, existen *switches* no administrables más comúnmente utilizados en las redes domésticas, los cuales no pueden realizar cambios en su comportamiento.

El protocolo *OpenFlow*, es un estándar en la arquitectura de las redes definidas por software SDN, el cual define la comunicación entre el controlador de la SDN y los dispositivos de red. En [9], se describe como un protocolo flexible que está basado en un *switch* Ethernet con una o más tablas de flujo internas y una interfaz estándar para agregar y remover entradas de dicha tabla.

2.1. Estado del arte

A continuación se presentan algunos trabajos de investigación con el fin de resaltar la relevancia de este proyecto de tesis.

Existen numerosas técnicas y algoritmos para el balanceo de carga. Estos son utilizados para agilizar el acceso a las peticiones hechas por los usuarios. Estas peticiones responden al tipo de servicio o aplicación, al estado de la red o a los servidores disponibles al momento de la solicitud. Actualmente, se han diseñado métodos que permiten la combinación de distintos algoritmos para elegir al mejor servidor. Por ejemplo, una ventaja de las SDN con respecto a las redes tradicionales es el uso de un controlador que observa el comportamiento global y es manejado mediante software que permite que los dispositivos de red puedan trabajar de una manera más equilibrada y a la vez dinámica. Esto se hace a través de interfaces de programación, mejor conocidas como API. Por ejemplo, en [10] los autores crean una SDN en un dispositivo final como un *smartphone* que cuenta con al menos dos interfaces de red (4G y WiFi) con el fin de balancear el tráfico del usuario. Para ello, utilizan los algoritmos *Round-robin* y *Hash* basado en *switches* virtuales *Open vSwitch* y *Ofdatapath*. Sin embargo, los autores utilizan el simulador *GNS3* con tráfico HTTP en lugar de un dispositivo físico. Los autores afirman que la SDN interna en el dispositivo es transparente para las aplicaciones, y para la infraestructura de red por lo que se puede implementar de manera simple y sin gran carga de procesos.

En [11] se toma un enfoque en una red de topología *Data Center* para balancear la carga mediante el algoritmo *Dijkstra* el cual selecciona la ruta de menor costo y carga. Para ello los autores utilizan el controlador *Floodlight* y el emulador *Mininet* basado en *switches OpenFlow*.

Los autores en [12] presentan un algoritmo para crear un equilibrio de carga dinámico utilizando centros de datos SDN utilizando el algoritmo *OSPF*. Este es implementado en el emulador *Mininet* y el controlador *Floodlight* en base al lenguaje de programación Python.

En "Aplicación de Balanceo De Carga Dinámico para servidores"se propone un balanceador de carga basado en SDN, escrito por Minda C. y Pacheco R., donde el algoritmo de balanceo de carga está basado en un sistema de criterios en tiempo de ejecución desde distintos puntos de la red con el fin de seleccionar el servidor con las mejores condiciones de respuesta. La aplicación está escrita en lenguaje Python y se utilizó el controlador Ryu. Los autores utilizan la herramienta de emulación Mininet para la evaluación de resultados y las herramienta *HttpPerf* y *Wireshark*. Además, muestran que la respuesta se reduce hasta en un $50\,\%$

comparado con el método de Round-robin.

A diferencia de los trabajos previos, en este proyecto de tesis se utilizan dispositivos reales (*Raspberry Pi*) en lugar de un emulador como es Mininet. Esto con fin de analizar el comportamiento bajo estrés de una red SDN con límites físicos de memoria y de ancho de banda. Además de mostrar la instalación y configuración en los dispositivos físicos que oculta un emulador.

Capítulo 3

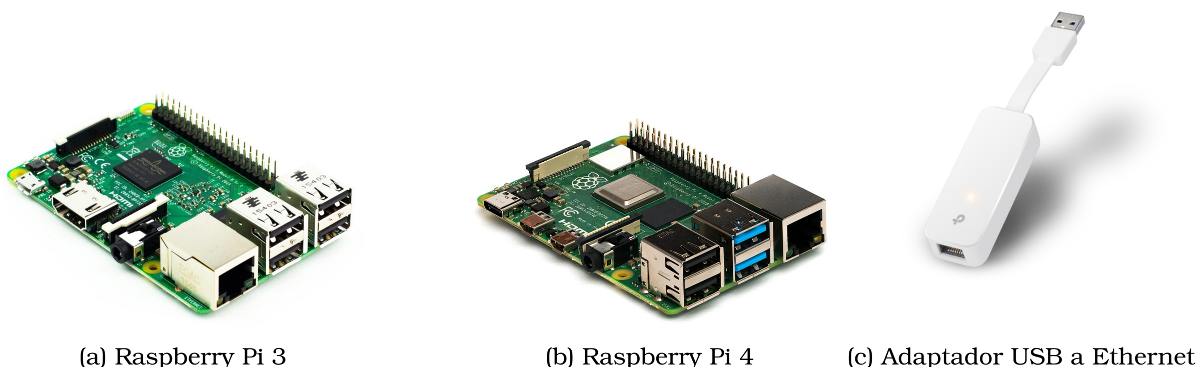
Herramientas de desarrollo

3.1. Componentes físicos

En este trabajo de tesis solo se utilizan dispositivos bajo la licencia de software libre GNU. Particularmente, se utilizan *switches* virtuales creados con el software *Open vSwitch* [13] (version 2.14.0) instalados en microprocesadores *Raspberry Pi* [14] (modelos Pi 3 B+ y Pi 4) bajo el sistema operativo *Raspbian* [15].

Las *Raspberry Pi* son microprocesadores de bajo costo (en comparación a una computadora tradicional) utilizadas en una amplia gama de proyectos. Por ejemplo, pueden albergar pequeños servidores, proyecto de IoT, como control de casas inteligentes, control de sistemas de riego, o emuladores de videojuegos, entre muchas más. Dadas las posibilidades de conectarse al monitor de alguna computadora mediante un puerto HDMI o mini HDMI según la tarjeta, se convierten en un dispositivo altamente útil para la creación de proyectos de todo tipo. Para este proyecto, se usa *Raspbian* como sistema operativo, el cual es una distribución de Linux Debian. Es importante mencionar que, las *Raspberry Pi 4*, cuentan con dos puertos USB 2.0 y dos puertos USB 3.0, mientras para el caso de las Raspberry Pi 3, cuentan con cuatro puertos USB 2.0. Así como un puerto Ethernet para ambos modelos.

Para la conexión física entre las *Raspberry* se utilizaron cables Ethernet RJ45 mediante los puertos integrados en los microprocesadores, así como los adaptadores USB a Ethernet (ver figura 3.1), para tener un mayor número de puertos (5 puertos Ethernet en total) en cada microcontrolador y así extender la conectividad. Gracias a esto, se pueden implementar diferentes topologías en cuanto a tipo y tamaño.



(a) Raspberry Pi 3 (b) Raspberry Pi 4 (c) Adaptador USB a Ethernet

Figura 3.1: Tarjetas *Raspberry* y adaptador USB-Ethernet.

3.2. Protocolo OpenFlow

La comunicación entre los dispositivos de red (*switches*) y el controlador ocurre a través del protocolo *OpenFlow* [16]. Es necesario puntualizar que el controlador es un dispositivo centralizado, que puede ser físico o virtual, el cual se comunica con los dispositivos de red mediante la capa 2 y 3 del modelo OSI.

De forma general, las SDN se componen de tres capas como se muestra en la figura 3.2.

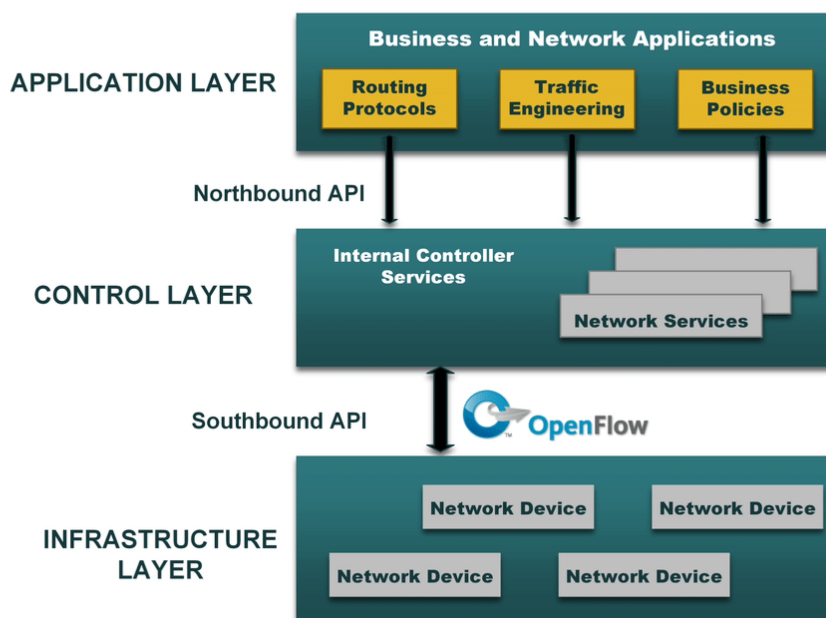


Figura 3.2: Capas de una SDN [1].

1. **Application layer:** Aquí es donde se ejecutan las aplicaciones como balanceadores de carga, *firewalls*, servidores DNS, etc. En el presente trabajo, es donde se encuentran los *scripts* que se encargan de realizar, de forma dinámica, el balanceo de carga del tráfico de datos.

2. **Control layer:** En esta capa se encuentra el controlador centralizado. Éste actúa como un puente entre la aplicación y los *switches*, procesa la información de la capa de aplicación y la convierte en *flujos* que son cargados en la tabla de flujos de cada *switch* vía el protocolo *OpenFlow*. Así como también puede ser utilizado para monitorizar los *switches* y sus puertos en un ambiente de administración.

3. **Data/Infrastructure layer:** En esta capa se encuentra el plano de reenvío o encaminamiento de paquetes que tienen los switches o dispositivos físicos conectados. Estos dispositivos consultan sus tablas de flujo para conmutar los paquetes a través de la red.

El protocolo *OpenFlow* (OF) se fundamenta en la comunicación entre un dispositivo controlador de la SDN y uno o más dispositivos simples de red. Este protocolo tiene sus orígenes en el año 2008 en la Universidad de Stanford, y fue incorporado en el *backbone* de la red de Google entre 2011 y 2012. Actualmente, es administrado por la *Open Network Foundation* (OFN) [17].
Es un estándar utilizado entre el controlador y los *switches* virtuales, por lo que la comunicación entre estos y el controlador no afecta al resto de la red. Estos mensajes son exclusivos entre cada uno de los *switches* y el controlador y su información no es expuesta hacía los otros *switches* de la red. Las especificaciones del protocolo *OpenFlow*, se detallan en el documento *Open Flow Switch Specification* [18].

3.2.1. Iniciación del canal OpenFlow

Este protocolo trabaja sobre el protocolo TCP. Por lo que, para la versión 1.3 (y anteriores) utilizada en este proyecto, el puerto por defecto es el 6633. Además, es necesario una conectividad por medio de IP entre el controlador y los *switches* para establecer comunicación. El canal *OpenFlow* se concreta después del *three-way handshake* exitoso del protocolo TCP. La inicialización del protocolo *OpenFlow* se puede dividir en los siguientes pasos:

- Los *switches* envían un paquete "HELLO" para iniciar la comunicación del canal OF, este mensaje contiene la versión máxima de *OpenFlow*. El controlador responde al mensaje HELLO con su máxima versión de *OpenFlow* soportada, para que posteriormente los *switches* negocien la versión a utilizar.

- Una vez que la versión ha sido negociada, el controlador envía un mensaje de "FEATURE_REQUEST", el cual pregunta al *switch* sobre sus capacidades de *OpenFlow*, así como el número de tablas de flujo o *"flow tables"*, acciones, etc. Finalmente, el *switch* responde con un mensaje "FEATURE_REPLY" que contiene sus capacidades, así como un identificador único llamado *Datapath ID (DPID)*.

3.2.2. Tablas OpenFlow y Flow Entries

Las *Flow Tables* pueden entenderse como las tablas MAC de los *switches* tradicionales que guardan la dirección de hardware de cada dispositivo conectado. Estas tablas guardan las *Flow Entries* o entradas de flujo que le dicen a los *switches* de la SDN que hacer con cada paquete que entra por algún puerto. Para ello, el *switch* hace un *match* o enlace con diferentes parámetros como la dirección IP, el número de puerto, MAC *Address*, VLAN ID, protocolo de transporte, puerto fuente o puerto destino, y selecciona la *Flow Entry* que mejor coincida para ejecutar la acción asociada a esa entrada. Entre las acciones existentes está tirar el paquete (drop), reenviarlo hacía otro puerto, *flooding*, o enviarlo al controlador para que decida que hacer con él.

Si el *switch* no tiene un *Flow Entry* que coincida con un paquete entrante, el *switch* puede tener una entrada "TABLE_MISS", la cual tiene prioridad mínima y las acciones pueden ser tirar el paquete o enviarlo al controlador. Cuando el controlador recibe este tipo de paquete, lo envía al plano de aplicación, el cual se encargará de procesar el paquete e informar al controlador si un nuevo *Flow Entry* necesita ser insertado en el *Flow Table* del *switch*. Si es el caso, el paquete insertará un nuevo *Flow Entry* en el switch. De esta forma, si un paquete con el mismo *match* es recibido por el *switch*, ahora tiene una *Flow Entry* adecuado para el manejo de dicho paquete. Esto mejora la eficiencia de la red, ya que se pueden tomar decisiones en tiempo real.

3.2.3. Controlador

Como se mencionó, una de las capas del protocolo *OpenFlow* es la de control, por lo que es necesario hacer énfasis en esta parte de la arquitectura, ya que es un punto muy importante en el desarrollo de este trabajo de tesis. El controlador es el encargado de conectar y configurar los dispositivos de la red, así como determinar el mejor encaminamiento para el tráfico de la capa de aplicación. Los controladores pueden simplemente dedicarse a la administración de la red, manejando las comunicaciones entre las aplicaciones y dispositivos de forma eficiente, o modificar los flujos de la red dependiendo de las necesidades.

El controlador transmite la información hacia los dispositivos de red por medio de las *Southbound API's* (ver figura 3.2) y de las aplicaciones por medio de las *Northbound API's*. Al ser implementado en software, se puede administrar el tráfico de forma dinámica y específica.

Para elegir un controlador que permita la implementación de la SDN, en primer lugar, se buscó aquellos que fueran *Open Source*, así como la facilidad de implementación. A continuación se muestran los controladores más populares, de los cuales se mencionan sus características principales [19].

- *OpenDayLigth* [20]: Es de los controladores *Open Source* más populares y probados hasta el día de hoy, es multiprotocolo y está construido utilizando el *framework* Java OSGi. Una de las grandes ventajas que tiene sobre los demás controladores, es la posibilidad de actualizar scripts durante la ejecución gracias a los OSGi containers, logrando así una mayor flexibilidad y funcionalidad.

- *Open Networking Operating System (ONOS)* [21]: Se trata de un controlador *Open Source* modular que también utiliza el *framework* Java OSGi. Al utilizar dicho *framework* tiene la capacidad de poder conectar o desconectar diferentes funcionalidades en tiempo de ejecución. ONOS tiene el soporte de Linux Foundation Networking, lo cual le proporciona una extensa documentación, así como una gran comunidad de desarrollo. ONOS es la mejor opción para *Communication Service Providers*, ya que tiene una extensa lista de *Nortbound* y *Southbond* API's, de esta manera los prestadores del servicio no tienen que desarrollar sus propias aplicaciones, y es posible escalar fácilmente según las necesidades del proyecto. Sin embargo, para proyectos pequeños, resulta demasiado extenso, lo que conlleva al desperdicio de recursos que el controlador no va aprovechar.

- *Ryu* [22]: Es un *framework Open Source* de SDN basado en componentes. Provee componentes de software con API's preexistentes, lo cual facilita a los desarrolladores crear nuevos sistemas de administración y control de aplicaciones. Ryu tiene soporte para diferentes protocolos y diversos dispositivos de red como *OpenFlow*, *Netconf*, *OF-config*, entre otros. El código es totalmente abierto bajo la licencia de Apache 2.0, y se encuentra completamente escrito en Python. La estructuración del controlador Ryu es diferente a la de los controladores ya mencionados, ya que su núcleo únicamente soporta un número limitado de aplicaciones, y no es posible realizar cambios en tiempo de ejecución. De igual forma, los usuarios tienen que escribir el código para lograr el comportamiento deseado en la red. Al utilizarse para proyectos pequeños existe una gran comunidad y foros de apoyo, así como una gran documentación con ejemplos sencillos.

- *Faucet* [23]: Es un controlador ligero *OpenSource* para SDN. Está construido sobre Ryu y habilita operadores de red para que operen sus redes de la misma forma en que se operan los *clusters* de servidores. Al igual que RYU, está escrito completamente en Python, con API's predefinidas, las cuales son fáciles de modificar por los desarrolladores según las necesidades de cada proyecto.

- *POX* [24]: Proporciona un framework de comunicación con *switches* SDN utilizando el protocolo *OpenFlow* o bien OVSDB protocol. POX utiliza Python como lenguaje de programación. Se ha convertido en una herramienta popular como primer acercamiento a las redes definidas por software, así como para propósitos de investigación en el mismo tema. Es posible utilizar POX como un controlador de SDN inmediatamente después de su instalación, con los componentes básicos. Aunque para generar escenarios mas complejos para el controlador, los desarrolladores tendrán que escribir las aplicaciones. Al momento de realizar el presente trabajo, la documentación acerca del controlador de la página oficial es insuficiente hablando del alcance, escalabilidad y flexibilidad de una SDN, únicamente se habla de instalación y ejemplos simples bajo el emulador de SDN Mininet.

Con base a las ventajas y desventajas presentadas anteriormente, se realizó una comparación entre los diferentes controladores, así como se evaluó cuál de ellos se adapta de mejor forma al alcance de este proyecto. Aunque algunos controladores tienen la posibilidad de realizar cambios en acciones de tiempo de ejecución, se consideró que esta característica tiene una mayor utilidad en redes de mayor tamaño, por lo que excede los fines de este proyecto, sin embargo, se puede contemplar como una opción en caso de ampliar las capacidades de la red. Asimismo, se decidió que el uso de interfaces gráficas para la monitorización de la red, así como sus estadísticas, eran innecesarias dado que es posible obtener esta información desde la terminal de Linux. De igual manera se tomó en cuenta el lenguaje de programación, eligiendo aquellos controladores en donde los *scripts* estén escritos en Python. Especialmente por su fácil sintaxis y comprensión del código.

En conclusión, el controlador utilizado para el desarrollo de este trabajo fue Ryu. Ya que es un controlador fácil de implementar, con sencillez en la lectura de código, y no requiere de una extensa cantidad de módulos como los que utilizan el *framework* Java OSCGi, además de que cuenta con una extensa documentación. En la sección 3.5.4 se describe el proceso de descarga e instalación del controlador Ryu.

3.3. Open vSwitch

Open vSwitch es un software multicapa bajo la licencia de código abierto Apache 2.0, el cual funciona como *switch* virtual en entornos de máquinas virtuales, y está diseñado para tener soporte multiplataforma en sistemas basados en virtualización Linux tales como Xen Server, KVM y VirtualBox. En este entorno virtualizado, el conmutador virtual o *switch* tiene dos funciones principales, las cuales consisten en conmutar el tráfico entre las máquinas virtuales y la comunicación entre máquinas virtuales hacia redes externas.

La mayor parte del código fuente se encuentra escrito en lenguaje C, y puede ser llevado fácilmente a otros ambientes. La versión actual al momento de redactar la presente tesis, admite las siguientes características:

- Estándar 802.1Q VLAN, con puertos *Trunk* y de acceso.
- Configuración de calidad de servicio (QoS)
- Tunneling
- Base de datos transaccional con enlaces C y Python
- Reenvío de alto rendimiento utilizando un módulo del kernel de Linux

Open vSwitch proporciona dos protocolos para su gestión: *OpenFlow*, que se dedica a gestionar el comportamiento de los *switches* a través de tablas de flujo; y *OVSDB* que se utiliza para conocer el estado de los puertos.

3.4. Instalación del Sistema Operativo Raspbian y configuraciones previas

Para la instalación del sistema operativo, fue necesario descargar desde la página oficial de *Raspberry* [15] la imagen *Raspberry Pi Imager* [25], la cual permite instalar el sistema operativo Raspberry Pi OS (Raspbian) en una tarjeta SD a través de una computadora mediante una interfaz gráfica. Una vez realizada la instalación del sistema operativo, se realizan los siguientes pasos.

3.4.1. Asignación de direcciones IP para clientes y servidores

Primero se configuraron las direcciones IP privadas de clase C. Para ello, se asigna una IP estática en cada dispositivo en el archivo */etc/dhcpcd.conf*. La asignación de direcciones IP, se muestra en la tabla 3.1. Es importante destacar que las configuración de cada dispositivo se realizó únicamente para la interfaz eth0, que es la integrada en las *Raspberry*, ya que es la interfaz por la cual los clientes y servidores aceptarán y enviarán el tráfico de control. Sin embargo, se puede utilizar cualquier otra interfase.

3.1: Archivo /etc/dhcpcd.conf

```
interface eth0
static ip_address = 192.168.10.xx
static routers = 192.168.10.254
static domain_name_servers = 192.168.10.254 8.8.8.8
```

Direcciones IP asignadas	
Dispositivo	Switches
SW1	192.168.10.10
SW2	192.168.10.20
C1	192.168.10.11
C2	192.168.10.12
C3	192.168.10.14
C4	192.168.10.15
SRV1	192.168.10.30
SRV2	192.168.10.40
SRV3	192.168.10.50
Controller	192.168.10.100

Tabla 3.1: Tabla de direcciones IP.

3.4.2. Instalación de Open vSwitch en las Raspberry

El procedimiento de instalación para el software *Open vSwitch* se realizó con los siguientes comandos (ver listado 3.2).

3.2: Instalacion Open vSwitch.

```
sudo su
apt-get update
wget http://openvswitch.org/releases/openvswitch-2.14.0.tar.gz
tar -xvf openvswitch
apt-get install python-simplejson python-qt4 libssl-dev ython-twisted-conch
apt-get install automake autoconf gcc uml-utilities libtool uild-essential pkg-config
uname -r
apt-get install -y linux-headers-4.9.0-6-rpi
./configure --with-linux=/lib/modules/4.9.0-6-rpi/build
make && make install
cd datapath/linux
modprobe openvswitch
cat /etc/modules
echo "openvswitch" >> /etc/modules
cat /etc/modules
cd ../..
touch /usr/local/etc/ovs-vswitchd.conf
mkdir -p /usr/local/etc/openvswitch
apt-get install uuid-runtime
```

3.3: Comandos adicionales para la instalación.

```
echo "export PATH=$PATH:/usr/local/share/openvswitch/scripts" > .bash_profile
echo ". ~/.bash_profile" >> .bashrc
```

```
apt-get install uuid-runtime
ovsdb-tool create /usr/local/etc/openvswitch/conf.db vswitchd/vswitch.ovsschema
```

Los comandos mostrados en la lista 3.3 son necesarios para crear la variable de entorno de *Open vSwitch*, así como de la herramienta UUID (Universal Unique Identifier) que asigna un número de proceso a los *switches*, y por último, la creación de una base de datos necesaria para el funcionamiento de Open vSwitch.

3.5. Configuración inicial de los Switches

La topología utilizada en esta tesis consta de dos *switches* (ver figura 4.1) identificados como SW1 y SW2, donde el SW1 tiene una conexión física con el controlador. En el listado 3.4, se muestra la configuración inicial para el *switch* SW1, mientras que en el listado 3.5 se muestra la configuración para el *switch* SW2. Es importante mencionar que esta configuración se realiza mediante los comandos "ovs-vsctl" y "ovs-ctl" propios de *Open vSwitch* que permiten crear los *switches* virtuales de capa 3. Además permiten agregar los puertos *ethernet* conectados por medio de los adaptadores USB-Ethernet. El comando `ovs-ctl add-br br0` crea un *switch* virtual llamado `br0`. Mientras que el comando `ovs-vsctl add-port br0 eth2` añade un puerto al *switch* `br0` sobre la interfaz `eth2` y le asigna un número. Finalmente, para el SW1 se asigna la conexión hacia el controlador y la versión del protocolo (ver listado 3.4).

3.5.1. Switch 1

3.4: Comandos para la configuración del Switch 1.

```
ovs-ctl start --system-id=random
ovs-vsctl add-br br0
ovs-vsctl add-port br0 eth0 -- set Interface eth0 ofport=10
ovs-vsctl add-port br0 eth1 -- set Interface eth0 ofport=1
ovs-vsctl add-port br0 eth2 -- set Interface eth0 ofport=2
ovs-vsctl add-port br0 eth3 -- set Interface eth0 ofport=3
ovs-vsctl add-port br0 eth4 -- set Interface eth0 ofport=4
ovs-vsctl set-controller br0 tcp:192.168.1.100:6633
ovs-vsctl set Bridge br0 protocols=OpenFlow13
```

3.5.2. Switch 2

3.5: Comandos de instalación de Ryu SDN.

```
ovs-ctl start --system-id=random
ovs-vsctl add-br br0
ovs-vsctl add-port br0 eth0
ovs-vsctl add-port br0 eth1
ovs-vsctl add-port br0 eth2
ovs-vsctl add-port br0 eth3
ovs-vsctl add-port br0 eth4
ovs-vsctl set Bridge br0 protocols=OpenFlow13
```

3.5.3. Inicialización de los *switches*

Para realizar el proceso de una forma más rápida, se crearon *scripts* en *bash* mediante los comandos listados en 3.4 y 3.5, para que únicamente sea necesario ejecutarlos y así activar los *switches*. Posteriormente se le asigna la dirección IP mostrada en la Tabla 3.1

3.6: Script para inicializar el switch.

```
#! /bin/bash

source ~/.bash_profile
ovs-ctl start --system-id=random
ip addr add 192.168.10.xx/24 dev br0
```

3.5.4. Instalación del controlador Ryu

Como se mencionó en la Sección 3.2.3, el controlador elegido fue Ryu. Para la instalación del controlador existen dos maneras hasta el momento de la redacción de este trabajo, la primera opción y la más rápida es utilizando el comando *pip install ryu*, o si se prefiere, es posible instalarlo desde el código fuente en el repositorio oficial en GitHub [26].

Los comandos para la instalación del controlador se muestran a continuación:

3.7: Comandos de instalación de Ryu SDN.

```
root@user:~# apt-get update
root@user:~# apt-get install pip
root@user:~# apt-get install git
root@user:~# apt-get install python-setuptools python-pip -y
root@user:~# apt-get install python-eventlet python-routes
root@user:~# apt-get install python-webob python-paramiko -y
root@user:~# pip install tinyrpc ovs oslo.config msgpack-python eventlet enum34
root@user:~# pip install eventlet --upgrade
root@user:~# pip install -r tools/pip-requires
root@user:~# git clone git://github.com/osrg/ryu.git
root@user:~# cd ryu
root@user:~/ryu# python ./setup.py install
```

3.5.5. Inicialización del controlador

Una vez ejecutados los comandos del apartado anterior, es posible comprobar la instalación exitosa del mismo con el comando listado en 3.8.

3.8: Comando de inicialización del controlador.

```
root@user:~# ryu-manager --verbose
```

El comando por sí solo no realiza ninguna acción, ya que todavía no hay una conexión física entre los *switches* y el mismo controlador. Sin embargo, la opción `--verbose` indica detalles sobre la ejecución. El resultado de este comando se muestra a continuación:

```
root@alexx: ~
root@alexx:~# ryu-manager --verbose
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK ofp_event
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPHello
  CONSUMES EventOFPEchoReply
```

Figura 3.3: Inicialización del controlador.

Para detener la ejecución de Ryu, es necesario presionar la combinación de teclas **Ctrl+C**.

3.5.6. Conexión del controlador con el switch

Una vez configurado el *switch* SW1 y el controlador, se puede verificar que ambos están configurados de forma adecuada. Primero, se conecta el puerto *Ethernet* del controlador hacia el puerto `eth0` del *switch* SW1. La Figura 3.4 muestra el comando `controller:~ # ryu manager --verbose`, que indica la inicialización del canal *OpenFlow* con los mensajes OFPHELLO para la negociación del protocolo, así como las características que ofrecerá el controlador a los *switches* y sus capacidades tal como se indicó en la Sección 3.2.1. Posteriormente, se muestra la conexión exitosa con el *switch*, mostrando su dirección IP, así como el puerto destino.

Por otro lado, la figura 3.5, lista el estado del controlador. Se sabe que la conexión es adecuada y exitosa, debido a que el atributo "`is_connected`" se encuentra en "`true`", además, los valores del atributo "`target`" deben coincidir con los comandos mostrados en la Sección 3.5.1.

```
root@alexx: ~
root@alexx:~# ryu-manager --verbose
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK ofp_event
  CONSUMES EventOFPHello
  CONSUMES EventOFPPortStatus
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPEchoReply
connected socket:<eventlet.greenio.base.GreenSocket object at 0x7f504a1d5250> address:('192.168.10.10', 6075
6)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x7f504a24be10>
move onto config mode
switch features ev version=0x4,msg_type=0x6,msg_len=0x20,xid=0x8ee31d6f,OFPSwitchFeatures(auxiliary_id=0,cap
abilities=79,datapath_id=202481593439924,n_buffers=0,n_tables=254)
move onto main mode
```

Figura 3.4: Conexión desde el controlador.

```
pi@SW1: ~
root@SW1:~# ovs-vsctl list controller
_uuid               : 9d5dbb02-d290-4ae8-bb90-d6137747a614
connection_mode     : []
controller_burst_limit: []
controller_queue_size: []
controller_rate_limit: []
enable_async_messages: []
external_ids        : {}
inactivity_probe    : []
is_connected        : true
local_gateway       : []
local_ip            : []
local_netmask       : []
max_backoff         : []
other_config        : {}
role                : other
status              : {last_error="Connection refused", sec_since_connect="8", sec_since_disconnect="16", st
ate=ACTIVE}
target              : "tcp:192.168.10.100:6633"
type                : []
root@SW1:~#
```

Figura 3.5: Conexión desde el switch.

3.6. Configuración de los clientes y servidores

3.6.1. iPERF

Para generar tráfico entre los clientes hacia los servidores, se utilizó la herramienta iPerf [27]. La cual permite hacer estadísticas de tráfico de una red. Se decidió utilizar esta herramienta debido a los múltiples parámetros y características que se pueden incluir, como elección entre protocolos de transporte (TCP o UDP), reportes detallados de ambos lados (cliente y servidor), medición de paquetes perdidos, *delay* y *troughput* máximo.

iPerf por otro lado permite realizar conexiones en paralelo (*threads*) desde un cliente hacia diferentes servidores, con lo cual un cliente puede generar múltiples peticiones de forma simultánea, incrementando así el tráfico total de la red. La versión de iPerf utilizada es la 2.0.5.

Las opciones del comando de iPerf, así como una breve descripción se encuentran en la tabla 3.2.

Opción	Descripción	Ejemplo
-s	Corre iPerf en modo servidor	iperf -s
-c	Ejecuta iPerf en modo cliente (debe especificarse la dirección IP del servidor al que se quiere conectar)	iperf -c 192.168.1.10
-u	Utiliza el protocolo UDP en vez de TCP (debe colocarse en el cliente y servidor en caso de que se utilice)	iperf -s -u — iperf -c 192.168.10.10 -u
-t	Define el tiempo en segundos para transmitir datos (solo para clientes)	iperf -c 192.168.10.10 -u -t 20
-P	Número de conexiones simultaneas a realizar con el servidor (solo cliente)	iperf -c 192.168.10.10 -u -t 20 -P 10

Tabla 3.2: Opciones a utilizar en iPerf.

Por ejemplo, la figura 3.6 muestra el comando utilizado en los servidores, la opción *-s*, indica el modo servidor, y la opción *-u* indica que será tráfico UDP el que recibirá de los clientes.

```
alex@alexx: ~
root@Server4:~# iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  176 KByte (default)
------------------------------------------------------------
```

Figura 3.6: Servidor escuchando peticiones.

La figura 3.7 muestra la salida con la información de tráfico para un experimento aleatorio, donde se observa un resumen de información del tráfico de red, se muestra: la dirección IP del servidor, la dirección IP y el puerto del cliente al que está atendiendo. Así como el intervalo de tiempo durante el cual se recibió información. Además, se muestra la cantidad de datos transferidos, el ancho de banda, el *Jitter*, el total de paquetes enviados y el total de paquetes perdidos, así como el porcentaje que representan aquellos que se consideran perdidos.

```
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  176 KByte (default)
------------------------------------------------------------
[  3] local 192.168.10.12 port 5001 connected with 192.168.10.14 port 39270
[  4] local 192.168.10.12 port 5001 connected with 192.168.10.14 port 60980
[  5] local 192.168.10.12 port 5001 connected with 192.168.10.11 port 55503
[  6] local 192.168.10.12 port 5001 connected with 192.168.10.13 port 51086
[  7] local 192.168.10.12 port 5001 connected with 192.168.10.11 port 55633
[  8] local 192.168.10.12 port 5001 connected with 192.168.10.15 port 54620
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  4]  0.0-10.0 sec  1.24 MBytes  1.05 Mbits/sec   0.025 ms    5/  892 (0.56%)
[  5]  0.0-10.0 sec  1.24 MBytes  1.05 Mbits/sec   0.030 ms    6/  892 (0.67%)
[  6]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.045 ms    3/  892 (0.34%)
[  8]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.042 ms    2/  892 (0.22%)
```

Figura 3.7: Archivo de salida.

Capítulo 4

Implementación

Retomando el planteamiento del problema de la Sección 1.1, los balanceadores de carga toman la decisión sobre que servidor se encargará de responder las peticiones recibidas hacia cierto servicio o página en específico. En el desarrollo de este proyecto se realizaron las peticiones de comunicación hacia un servidor iPerf 3.6.1 a través de una dirección IP "virtual" correspondiente al balanceador de carga. Ya que el *switch* que actúa como balanceador de carga, visto desde el lado del cliente, es donde se encuentra el servicio deseado. Este balanceador no cuenta con ningún servidor iPerf dentro del mismo, ya que su objetivo es el reenvío de paquetes, tal como lo hace un *router* tradicional. El balanceador de carga fue configurado como SW1, el cual tuvo una conexión directa con el controlador, y a su vez se conecta a los clientes y a los servidores tal como se muestra en la figura 4.1.

El *switch* SW1 recibe peticiones hacia su dirección IP por el puerto UDP 5001. Al no tener ningún servidor corriendo que sea capaz de atender dichas peticiones, éste enviará un mensaje OFPT_Packet_In hacia el controlador, ya que el *switch* no cuenta con ninguna tabla de flujo (Flow Table), por lo que no puede decidir por sí mismo el puerto *ethernet* por el que debe encaminar el paquete. El controlador, recibe el mensaje *OpenFlow* y según el *script* que esté ejecutándose, tomará una decisión para enviar el paquete, las cuales se describirán más adelante. De esta forma, si el SW1 recibe otro paquete con las mismas características, éstas harán *match* con su *Flow Table* actualizado y así sabrá por que puerto enviar el paquete para que pueda ser atendida dicha petición. Es importante mencionar que el controlador ejecuta un *script* a la vez, el cual corresponde al algoritmo de balanceo de carga. De tal forma que, al recibir los paquetes *OpenFlow*, toma la decisión sobre a cuál servidor se puede enviar dicho paquete estableciendo un flujo en cada sentido: del cliente al servidor elegido, y del servidor elegido al cliente, para que el tráfico de datos no sea interrumpido y pueda completarse la transferencia de datos entre cliente-servidor.

El controlador al conocer la topología de forma global es capaz de almacenar las direcciones IP de los servidores y los puertos disponibles del SW1. De está forma, puede tomar decisiones y responder al *switch* con un OFPT_Packet_Out indicando hacia que puerto enviar las peticiones, así como guardar la información en la *Flow Table* del *switch* para que los paquetes del mismo cliente sean enviados al servidor elegido por el controlador.

4.1. Topología implementada

La figura 4.1 muestra la topología implementada, en la cual se concentran los clientes hacia el *switch* SW1 que está conectado directamente al controlador. Por ello, es necesario recordar que las peticiones se realizan hacia la dirección IP del *switch* SW1, que a su vez actúa como balanceador de carga.

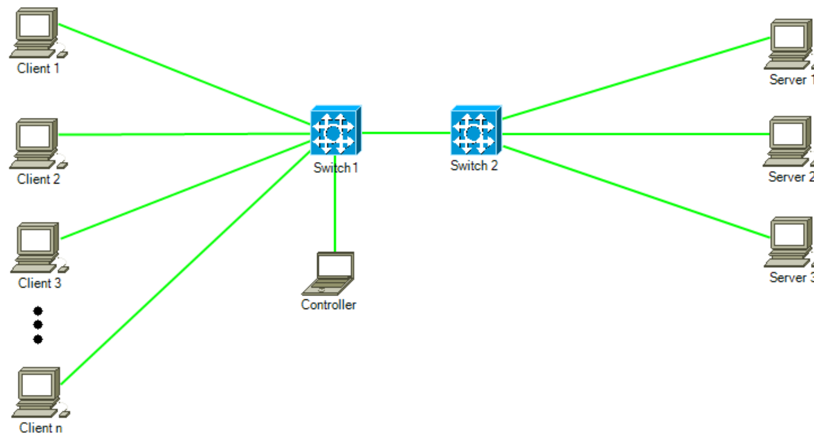


Figura 4.1: Topología implementada.

4.2. Comunicación cliente-servidor

La figura 4.2 muestra el proceso general de comunicación entre un cliente y un servidor cuando existe un balanceador de carga [28]. En el paso uno, el cliente hace una petición TCP/UDP hacia el *switch*. Cabe resaltar que, visto desde la perspectiva del cliente, éste realiza una conexión directa hacia el servidor. Sin embargo, es el *switch* el que realiza la conexión hacia el servidor sin que el cliente sea notificado sobre la presencia del balanceador de carga.

En el punto dos, el *switch* detecta que es una petición dirigida hacia su dirección IP con petición a un servicio, por lo que se comunica con el controlador para que éste decida hacia que servidor enviar la petición. Una vez elegido el servidor, se instalan dos flujos en las *Flow Tables* del *switch*: un flujo que cambie las direcciones MAC e IP destino hacia las del servidor elegido, así como encargarse de instalar la acción en el *switch* para que todos los paquetes futuros que coincidan (dirección IP/MAC origen y destino, así como el puerto de transporte) sean reenviados por un determinado puerto *ethernet*.

En el punto tres, el servidor recibe la petición, la procesa y la contesta en el punto cuatro, utilizando la IP del cliente ahora como dirección destino. Al llegar la respuesta del servidor hacia el *switch*, los *headers* harán *match* con la condición instalada en el punto dos, por lo que nuevamente cambiará los campos de la dirección IP y MAC del servidor (fuente) por la del *switch*. De esta manera, cuando el cliente reciba respuesta del servidor, parecerá que fue el *switch* quien respondió la petición.

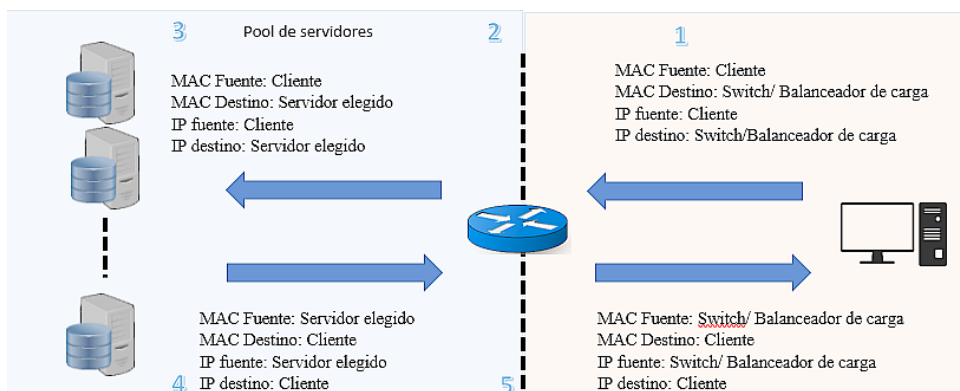


Figura 4.2: Flujo de las peticiones.

La tabla 4.1 extiende la explicación de un balanceador de carga con el soporte *OpenFlow*,

también muestra las direcciones MAC e IP simplificadas de un cliente, un *switch* y un servidor, respectivamente. Se utiliza el puerto UDP 5001 para el servidor (puerto utilizado por iPerf), y el puerto 9999 para el cliente.

Dispositivo	Dirección MAC	Dirección IP
Usuario	00:00:00	10.0.1.17
Switch	AA:AA:AA	10.0.1.1
Server	FF:FF:FF	10.0.10.21

Tabla 4.1: Tabla direcciones.

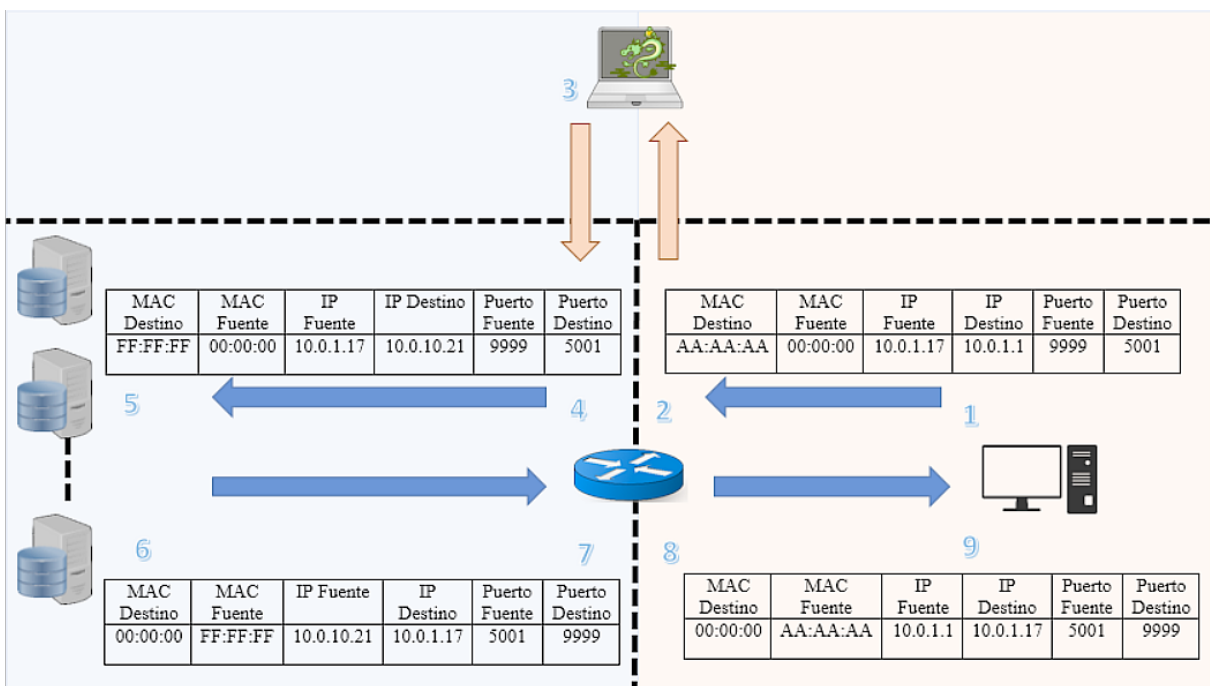


Figura 4.3: Balanceador con Openflow.

La figura 4.3 muestra el proceso de comunicación entre el cliente, el *switch* y el servidor. El cliente realiza la petición hacia el servicio iPerf, especificando la dirección IP del *switch* (paso 1). El *switch* detecta que la petición es dirigida hacia un servidor iPerf (mediante la dirección IP destino y puerto destino). En este momento el *switch* desconoce hacia donde encaminar dicho paquete, por lo que mandará un paquete `OFPT_Packet_IN` hacia el controlador (paso 2). Por su parte, el controlador al recibir un mensaje `OFPT_Packet_IN` desde el *switch*, deberá decidir hacia que servidor enviar los paquetes subsecuentes de ese cliente, ejecutando algún algoritmo de balanceo para hacer la elección. Una vez elegido el servidor por el controlador, éste responderá hacia el *switch* con un mensaje `OFPT_Packet_Out` con instrucciones para agregar nuevas entradas a la *Flow Table* del *switch* (paso 3). Estas entradas permiten que los próximos paquetes del mismo cliente se encaminen hacia el mismo servidor. Se agregan dos flujos en el *switch* para que tanto el tráfico de ida como de regreso se completen de forma satisfactoria entre cliente y servidor.

En el paso cuatro, el *switch* sabe que si un paquete hace *match* con las condiciones mencionadas, deberá cambiar la dirección MAC e IP destino del paquete, para que pueda ser recibida por un servidor, de igual forma, elige el puerto *Ethernet* del servidor seleccionado

para enviar el paquete por dicha interfaz. En el paso 5, el servidor responde la petición y es enviada al *switch* nuevamente (paso 6). En el punto siete, el *switch* recibe la respuesta del servidor hacia el cliente, pero el paquete hace *match* con el flujo instalado en el punto cuatro, por lo que ahora el *switch* sabe que si recibe un paquete de ese servidor hacia la dirección IP del cliente, ya no es necesario enviar el paquete al controlador para determinar el camino que debe tomar dicho paquete, ahora sabe que debe cambiar las direcciones fuente, por las del *switch*, para que parezca que es el *switch* el que atendió el servicio.

En el punto 8 se realiza el cambio de direcciones en los *headers* del paquete, y se envía por la interfaz correcta. Finalmente, en el punto nueve, el cliente recibe la respuesta del servidor. En este momento, la ruta en ambas direcciones la conoce el *switch* por lo que no preguntará más al controlador hacia que servidor encaminar los paquetes.

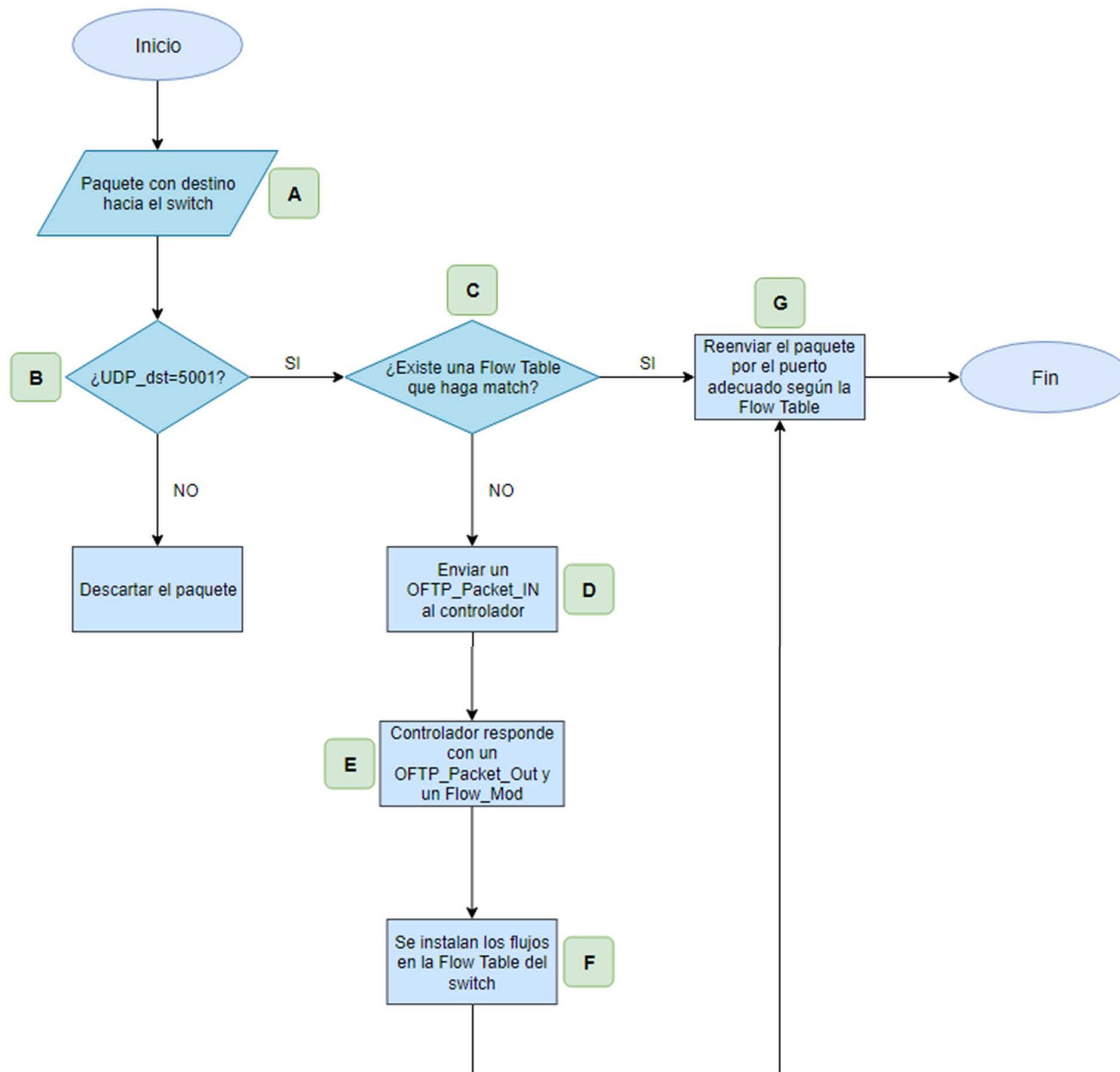


Figura 4.4: Diagrama de flujo de la aplicación.

La figura 4.4 muestra el diagrama de flujo del proceso de comunicación previamente explicado.

4.3. Balanceadores de carga

La Figura 4.5 ilustra un balanceador de carga. Los clientes con un identificador único envían peticiones hacia el *switch* conectado al controlador, éste corre la aplicación de balanceo de carga y elige un servidor, posteriormente el *switch* redirige cada petición hacia uno de ellos. Por ejemplo, la petición 1 y 3 son enviadas a un servidor, mientras la petición 2 y 5 son enviadas a un segundo servidor.

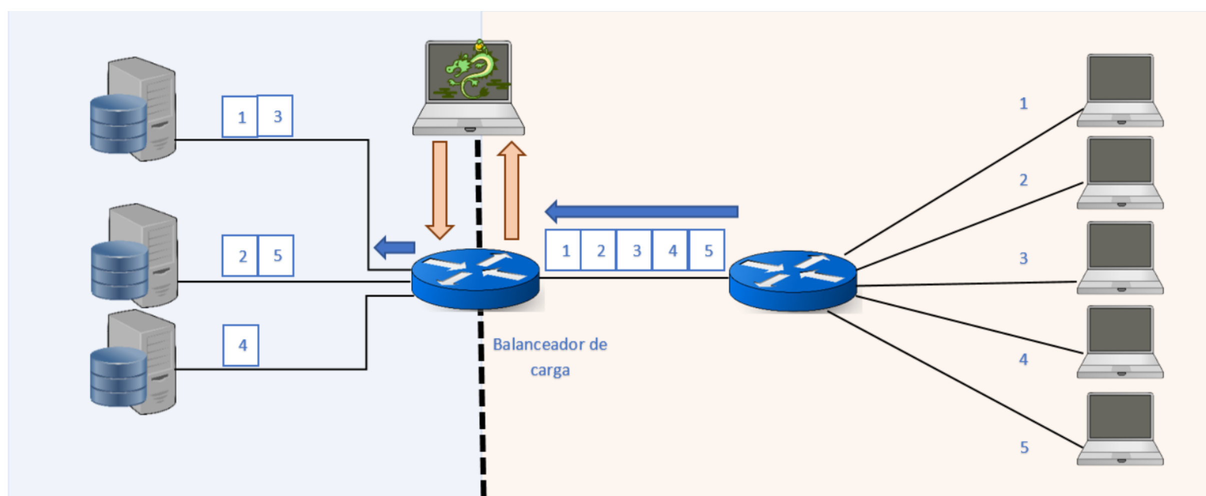


Figura 4.5: Balanceador de carga.

4.3.1. Balanceador Aleatorio

El primer balanceador de carga implementado en este proyecto es llamado *Random* o Aleatorio, esto es debido a su característica de seleccionar un servidor dentro de una lista tomada de la topología de red sin ninguna consideración especial sobre el estado del servidor o de características de la red. Para ello se utiliza la biblioteca *random* de Python, con el método `choice()`, tal como se puede observar en la línea 118 del código implementado en el Apéndice D, en donde se realiza la selección aleatoria de un elemento de la lista de servidores. El *switch* al momento de recibir una petición hacia el servicio IPerf, tal como se explica en la Figura 4.4 y comunicarse con el controlador, el controlador elige un servidor utilizando este criterio y responde al *switch* para instalar los flujos, y hacer el cambio de *headers* necesario con la información del servidor elegido.

4.3.2. Balanceador Round-robin

Round-robin es un algoritmo sencillo que elige todos los elementos de una lista, generalmente desde el primer elemento hasta el último de forma consecutiva o viceversa. El algoritmo Round-robin es utilizado por programadores de procesos en las CPU, ya que divide el tiempo de proceso en segmentos iguales para cada tarea sin priorizar algún proceso. En otras palabras, distribuye las peticiones enviadas por los clientes de forma equitativa entre los servidores sin tomar ningún factor de peso sobre la decisión, así, los servidores reciben el mismo número de peticiones y responden con la misma capacidad, ya que los servidores tienen exactamente la misma capacidad de procesamiento. La distribución de peticiones para este balanceador de carga se puede observar en la Figura 4.6.

Para la implementación de este algoritmo se agregó una variable que se incrementaba con cada petición hacia el servicio iPerf, y se aplicaba la operación módulo del número de servidores disponibles, por lo que cada petición es dirigida al siguiente servidor y después de elegir el último, continua con el primero de la lista. Esto se muestra en la línea 125 y 126 del Apéndice C.

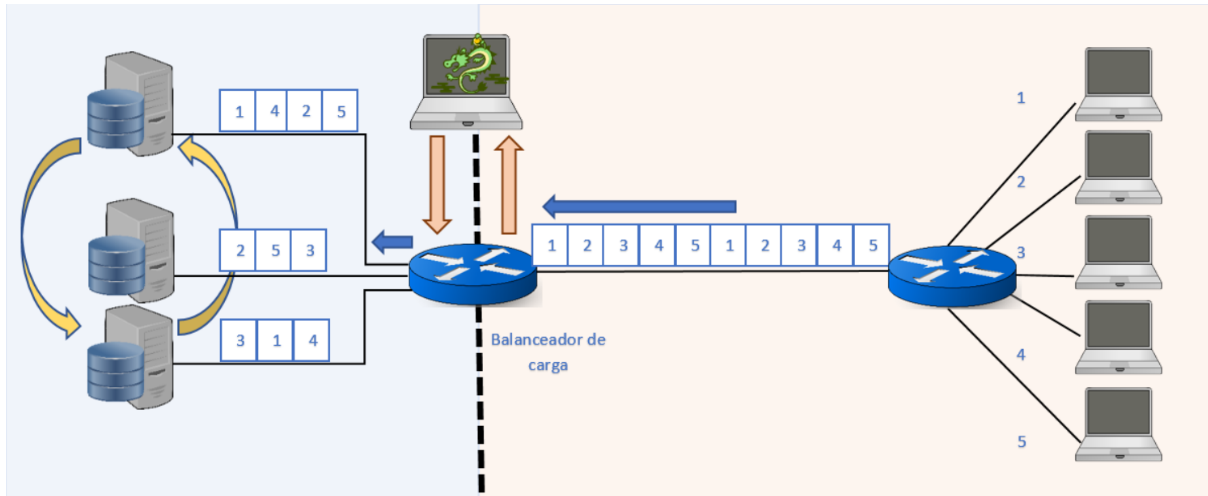


Figura 4.6: Balanceador Round Robin.

Capítulo 5

Pruebas y desempeño

Para cuantificar la eficiencia de un balanceador de carga es necesario tomar en cuenta diferentes parámetros, especialmente en una SDN que es capaz de cambiar su estrategia de reenvío de paquetes en tiempo de ejecución. Específicamente en este trabajo, se tomaron las siguientes métricas: número de peticiones recibidas por servidor, número de paquetes perdidos, throughput, y velocidad de transmisión. Para cada una de estas métricas se realizaron 20 pruebas para obtener valores promedios. Se utilizó la herramienta iPerf para generar tráfico entre los clientes y servidores, de tal manera que sea posible crear flujos de datos. En todos los experimentos se tomó la topología mostrada en la figura 4.1. Solo se cambió el software del balanceador de carga para analizar los dos casos de estudio (Aleatorio y Round-robin). En todos los experimentos se mantuvo constante el número de servidores (3) y se varió el número de peticiones creadas por los clientes. Es importante mencionar que ambos algoritmos se ejecutan en el *switch* que conecta a los servidores. Es importante recalcar, que la velocidad de transmisión por *default* utilizado por iPerf al generar tráfico UDP, es de 1 Mbps, el cual puede modificarse hasta 100 Mbps, pero se decidió dejar el valor por *default* para lograr la mayor capacidad de procesamiento por parte de los servidores al incrementar el número de peticiones que reciben.

5.1. Prueba Aleatoria

El algoritmo Aleatorio es utilizado para encaminar el tráfico de una manera sencilla sin tener que sacrificar tiempo de procesamiento para decidir a cuál servidor dirigir las peticiones. De esta manera, el controlador determina sin distinción alguna el servidor que recibirá la solicitud. La figura 5.1, muestra tres servidores disponibles. En esta prueba se varió el número de peticiones desde 4 hasta 200. Se puede ver que el número de peticiones recibidas no varía de forma considerable entre los tres servidores. Al incrementar el número de peticiones totales hasta acercarse a un valor de 150, se puede observar que el rendimiento de los servidores tiende a caer, y a partir de este valor los servidores ya no son capaces de recibir todas las peticiones.

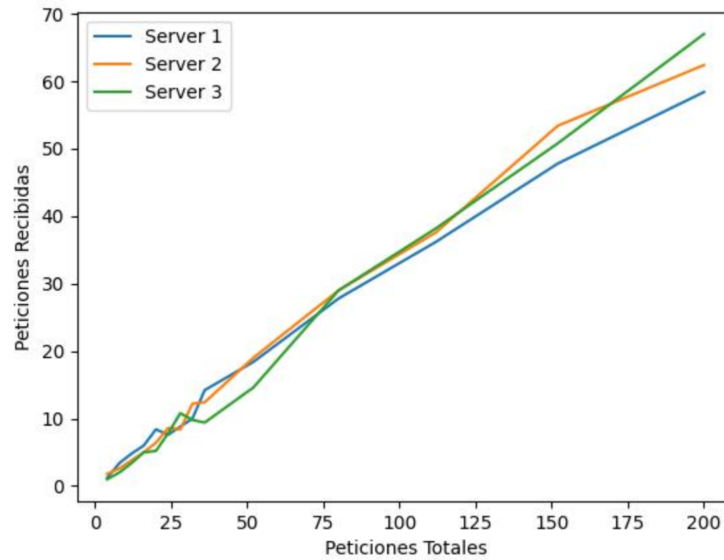


Figura 5.1: Peticiones recibidas vs. peticiones totales.

Tabla 5.1: Promedios de peticiones para Random.

Clientes Totales	Servidor 1		Servidor 2		Servidor 3	
	Peticiones	%	Peticiones	%	Peticiones	%
4	1.2	30	1.8	45	1	25
8	3.4	42.5	2.6	32.5	2	25
12	4.8	40	3.8	31.66	3.4	28.33
16	6	37.5	5	31.25	5	31.25
20	8.4	42	6.4	32	5.2	26
24	7.6	31.66	8.6	35.83	7.8	32.5
28	8.8	31.42	8.4	30	10.8	38.57
32	10	31.25	12.2	38.12	9.8	30.62
36	14.2	39.44	12.4	34.44	9.4	26.11
52	18.4	35.38	19	36.53	14.6	28.08
80	27.8	34.75	29	36.25	29	36.25
112	36.2	32.32	37.6	33.57	38.2	34.11
152	47.8	31.44	53.4	35.13	50.8	33.42
200	58.4	29.2	62.4	31.2	67	33.5
Promedio		34.92		34.53		30.62

La tabla 5.1 muestra el número promedio de peticiones recibidas por servidor mientras se aumenta el número de peticiones. Estas van entre el 30 % y 35 % del total enviadas. En un caso ideal, el balanceador debiese distribuir la carga de forma equitativa entre los tres servidores, es decir, un 33 % de peticiones recibidas por servidor. Sin embargo, los valores obtenidos no son tan distintos del valor ideal. Por ejemplo, se puede ver en la tabla que para 16 clientes, el Servidor 1 recibe 6 peticiones que representan el $37.5\,\%$ del total de peticiones,

mientras que el Servidor 3, para ese mismo número de clientes obtiene un 31.25 % de las peticiones.

Por otro lado, la Figura 5.2, muestra el porcentaje de paquetes perdidos por cada servidor. Se puede observar que conforme el número de peticiones por servidor aumenta, la cantidad de paquetes perdidos también aumenta. Por ejemplo, el porcentaje de paquetes perdidos alcanza el 78.86 % de perdidas al haber recibido 58.4 peticiones en el servidor 1. Estos datos se pueden observar en la Tabla 5.2.

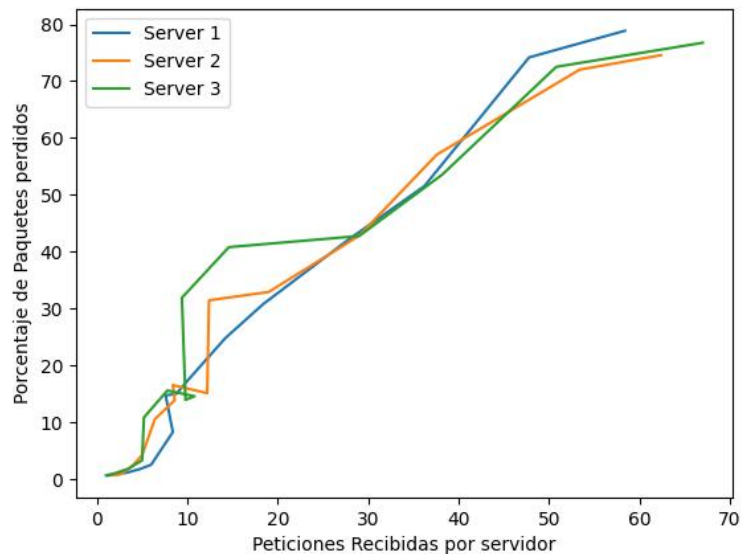


Figura 5.2: Porcentaje de peticiones perdidas.

Tabla 5.2: Server 1.

Peticiones Recibidas	**Troughput [Mbps]**	**Bandwidth [Mbps]**	**Porcentaje de perdidas**
1.2	1.272 666 667	1.05	0.635 039 223
3.4	1.266	1.047 058 824	1.186 552 405
4.8	1.258	1.043 333 333	1.811 560 37
6	1.248 266 667	1.038 333 333	2.547 437 621
8.4	1.175 047 619	1.002 214 286	8.296 430 667
7.6	1.092 210 526	0.955 552 631 6	14.739 884 39
8.8	1.088 772 727	0.945 636 363 6	15.027 753 73
10	1.059 52	0.929 48	17.296 400 88
14.2	0.963 383 098 6	0.870 154 929 6	24.807 473 8
18.4	0.886 178 260 9	0.842 076 087	30.809 902 05
27.8	0.741 433 093 5	0.716 489 208 6	42.105 517 83
36.2	0.620 796 685 1	0.620 944 751 4	51.519 393 48
47.8	0.330 624 937 2	0.424 073 849 4	74.183 169 01
58.4	0.270 721 301 4	0.311 030 821 9	78.867 910 05

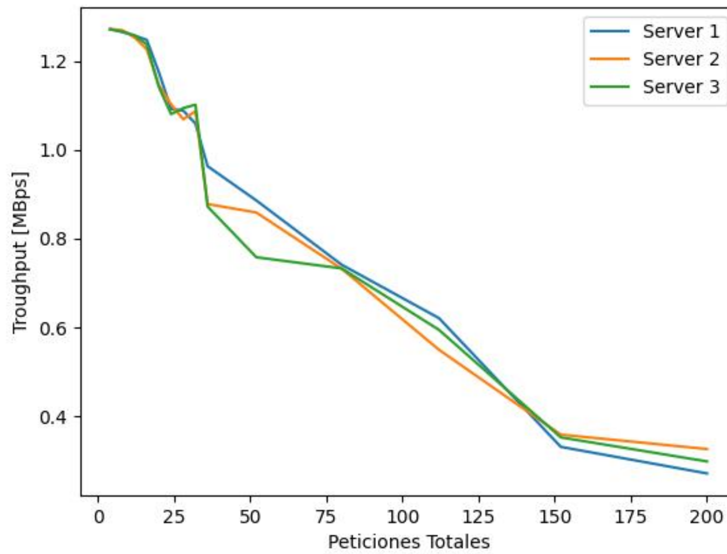


Figura 5.3: Throughput.

Como se mencionó, el *Throughtput* es una métrica de gran interés que permite cuantificar el beneficio de utilizar un algoritmo de balanceo, ya que mide la cantidad de información útil recibida por unidad de tiempo. La figura 5.3 muestra los resultados de los datos obtenidos en Megabits por segundo. Se puede ver en la figura que el *throughput* disminuye conforme crece el número de peticiones totales.

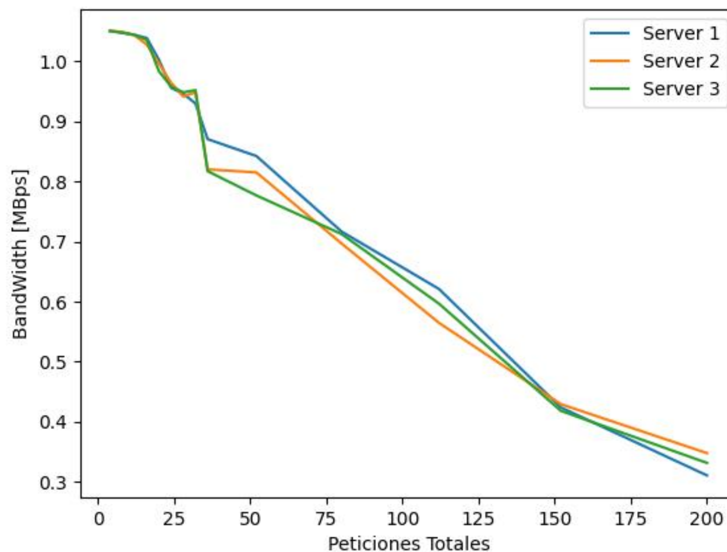


Figura 5.4: Velocidad de transmisión del algoritmo Aleatorio.

La velocidad de transmisión de la red mostrada en la figura 5.4, muestra un comportamiento similar a la figura 5.3, donde la velocidad disminuye conforme el número de peticiones aumenta. Además, ambos escenarios tienen un cambio de pendiente para las 150 peticiones. Como se mencionó, a partir de las 150 peticiones el algoritmo tiende a perder más paquetes y esto se ve reflejando en el cambio de pendiente.

5.2. Prueba Round-robin

La figura 5.5 presenta la relación entre el número total de peticiones contra la cantidad de peticiones recibidas por servidor. Se puede observar que, a partir de las 150 peticiones totales (en promedio 53 peticiones por servidor) los servidores comienzan a saturarse. Por ejemplo, el servidor 2, a partir de este número no es capaz de recibir más peticiones.

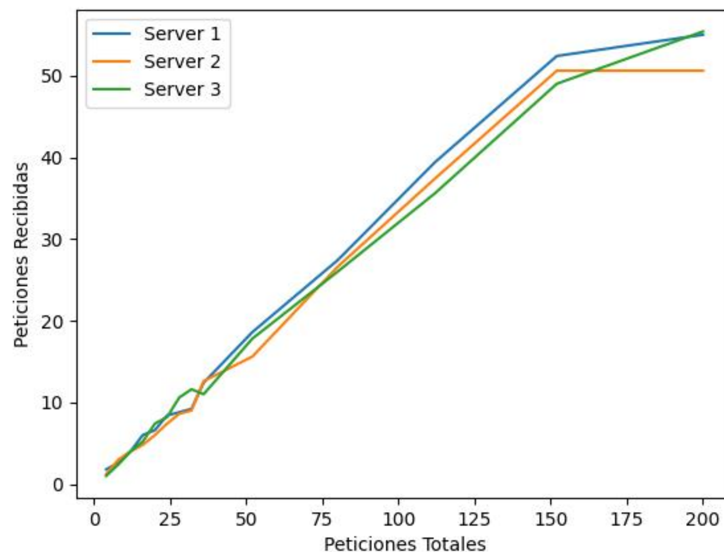


Figura 5.5: Peticiones del algoritmo Round-robin.

La tabla 5.3 presenta los promedios de peticiones para los tres servidores. Se puede ver en esta tabla que para el caso de 12 peticiones totales, el balanceador de carga alcanza el valor ideal. La tabla 5.4 presenta los valores promedios obtenidos en el experimento. Por lo cual, para efectos prácticos de medición del experimento, se determinó que 150 peticiones totales, fue el número máximo de peticiones que soportan los servidores, ya que a partir de allí su rendimiento se ve muy empobrecido.

Tabla 5.3: Promedios de peticiones Round Robin.

Clientes Totales	Servidor 1		Servidor 2		Servidor 3	
	Peticiones	%	Peticiones	%	Peticiones	%
4	1.8	45	1	25	1.2	30
8	2.6	32.5	2.4	30	3	37.5
12	4	33.33	4	33.33	4	33.33
16	6	37.5	5.2	32.5	4.8	30
20	6.6	33	7.4	37	6	30
24	8.4	35	8.2	34.17	7.4	30.83
28	8.8	31.49	10.6	37.86	8.6	30.71
32	9.2	28.75	11.6	36.25	9	28.12
36	12.4	34.44	11	30.55	12.6	35
52	18.6	35.77	17.8	34.23	15.6	30
80	27.4	34.25	26	32.5	26.6	33.25
112	39.4	35.18	35.6	31.78	37.4	33.39
152	52.4	34.47	49	32.24	50.6	33.29
200	55	27.5	55.4	27.7	50.6	25.3
Promedio		34.15		34.51		30.48

La figura 5.6 muestra el número de peticiones recibidas por servidor con respecto al porcentaje de paquetes perdidos. Se puede ver en esta figura un comportamiento muy parecido para cada uno de los servidores, en donde el número de paquetes perdidos va aumentando conforme aumentan las peticiones recibidas. También se puede ver que el comportamiento tiene varios cambios de pendiente y a partir de 26 peticiones recibidas por servidor, las gráficas tienen un comportamiento que varía muy poco una de la otra hasta las 50 peticiones recibidas por servidor, donde comienzan a decrecer. La tabla 5.3 muestra los valores obtenidos en el experimento para cada servidor.

Tabla 5.4: Promedios generales Round Robin.

Peticiones Recibidas	Troughput [Mbps]	Bandwidth [Mbps]	Porcentaje de perdidas
1.333 333 333	1.301 111 111	0.975 718 52	9.730 124
2.666 666 667	1.243 282 051	1.05	0.488 578 44
4	1.242 166 667	1.048 166 667	0.681 988 04
5.333 333 333	1.232 620 636	1.041 590 475	1.403 026 904
6.666 666 667	1.223 999 999	1.035 703 967	2.120 378 999
8	1.206 844 808	1.025 468 292	3.509 666 377
9.333 333 333	1.148 568 933	0.978 345 949 4	8.335 123 665
9.933 333 333	1.134 609 420	0.980 327 505 2	9.434 176 038
12	1.087 594 705	0.954 647 472 3	13.554 300 15
17.333 333 33	0.921 777 201	0.815 253 23	27.314 219 41
26.666 666 67	0.845 034 739	0.785 270 081 2	30.293 683 24
37.466 666 67	0.712 526 316	0.734 203 935 8	43.979 082 77
50.666 666 67	0.520 375 770	0.513 236 982 6	59.351 244 26
53.666 666 67	0.530 966 646	0.527 193 051 7	58.515 127 79

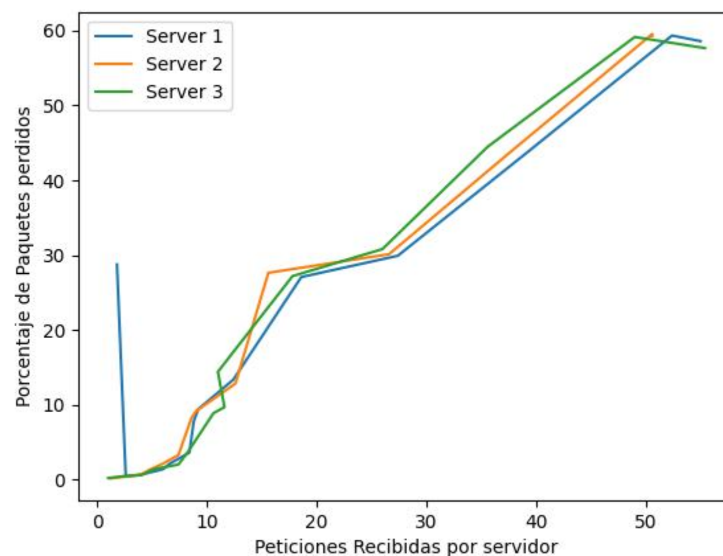


Figura 5.6: Porcentaje de paquetes perdidos del algoritmo Round-robin.

En cuanto a los mensajes recibidos de manera exitosa o *Throughput*, la figura 5.7 muestra valores en un rango entre 0.52 Mbps y 1.4 Mbps. Se puede observar que en el rango entre 50 y 125 peticiones existe una caída por parte del tercer servidor, mientras los otros dos servidores tienen un mejor desempeño. Después de ese rango, el comportamiento de los tres servidores fue prácticamente el mismo. Además se puede ver que al llegar a las 150 peticiones recibidas en total (o 50 peticiones recibidas por servidor) el *throughput* permanece casi constante.

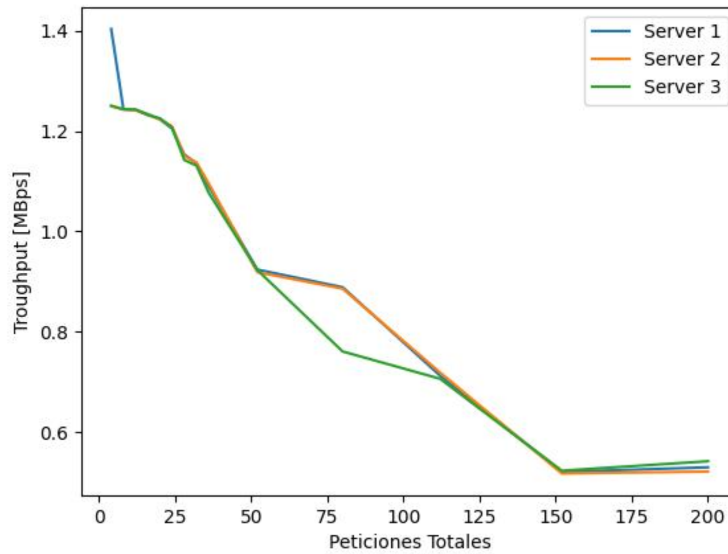


Figura 5.7: Throughput Round-robin.

La figura 5.8 muestra una velocidad de transmisión máxima de 1.05 Mbps y una mínima de 0.5 Mbps, que corresponde a un rango entre 50 y 120 peticiones. La figura también muestra que al llegar a las 150 peticiones en total, el desempeño de los servidores llega a un límite, recordando que es la capacidad máxima de peticiones de cada servidor.

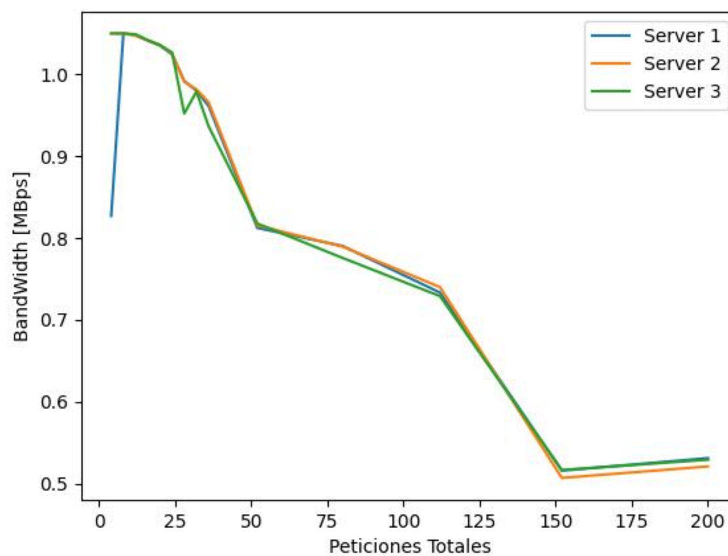


Figura 5.8: Velocidad de transmisión del algoritmo Round-robin.

5.3. Comparación de resultados

A continuación se realiza la comparación de desempeño entre los balanceadores de carga implementados. Esto con el fin de determinar cuál de ellos bajo condiciones de carga puede

considerarse mejor, o a partir de que rango de peticiones presenta un mejor rendimiento. Para esta comparación, se promedian los resultados de cada balanceador. De igual forma, los datos promediados de cada balanceador pueden consultarse en las tablas 5.1 y 5.3.

En la Figura 5.9, se muestran los promedios de peticiones recibidas por servidor respecto al número total de peticiones enviadas. Se puede ver en esta figura que desde 4 peticiones hasta 150, el comportamiento de ambos algoritmos es muy similar. Sin embargo, a partir de las 150 peticiones recibidas el algoritmo de Round-robin decrece más rápido en comparación con el algoritmo Aleatorio.

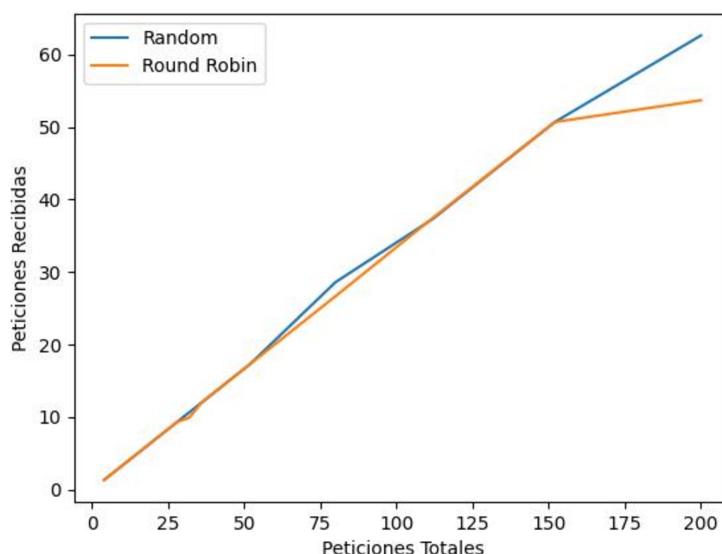


Figura 5.9: Peticiones recibidas.

La figura 5.10 muestra el número peticiones totales contra el porcentaje de paquetes perdidos. En esta figura se puede observar que, en el rango de 4 a 200 peticiones totales, el balanceador Round-robin presenta un mejor desempeño. En específico, este algoritmo presenta un $55\,\%$ de menor pérdida de paquetes con respecto al balanceador Aleatorio.

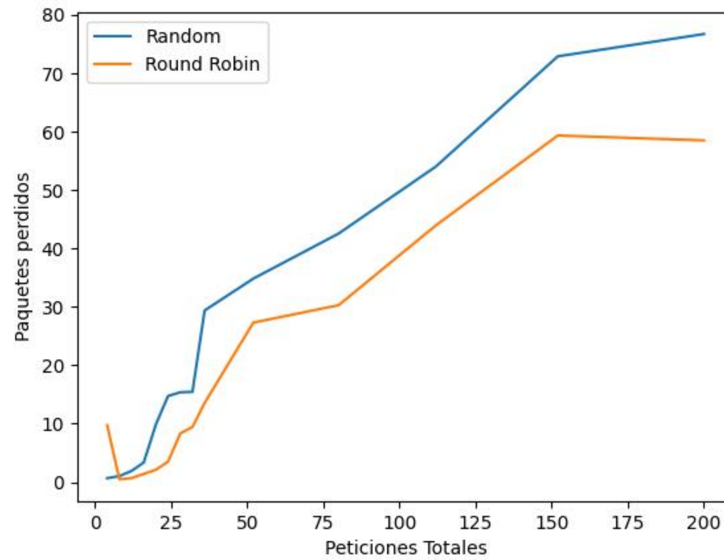


Figura 5.10: Pérdidas de paquetes.

La Figura 5.11 presenta el *throughput* total de ambos balanceadores de carga. El algoritmo Round-robin logró transferir más paquetes por unidad de tiempo comparado con el balanceador Aleatorio. En promedio el algoritmo Round-robin alcanza un 15 % de más paquetes transferidos por unidad de tiempo.

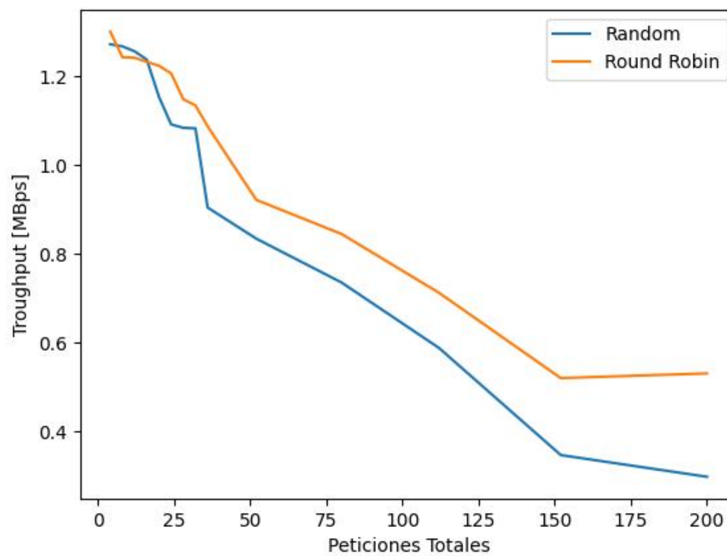


Figura 5.11: Throughput total.

Por último, en la figura 5.12, se puede observar la velocidad de transmisión obtenida en ambos experimentos. Recordando que la velocidad de transmisión de una red, es la medida de cuanta información puede ser transmitida en un canal por unidad de tiempo, es importante mencionar que no siempre se puede alcanzar toda la capacidad. Nuevamente podemos observar que el balanceador Round-robin tiene un mejor desempeño. Para este

experimento alcanza una velocidad de transmisión 10 % mayor en promedio que el logrando con el balanceador Aleatorio.

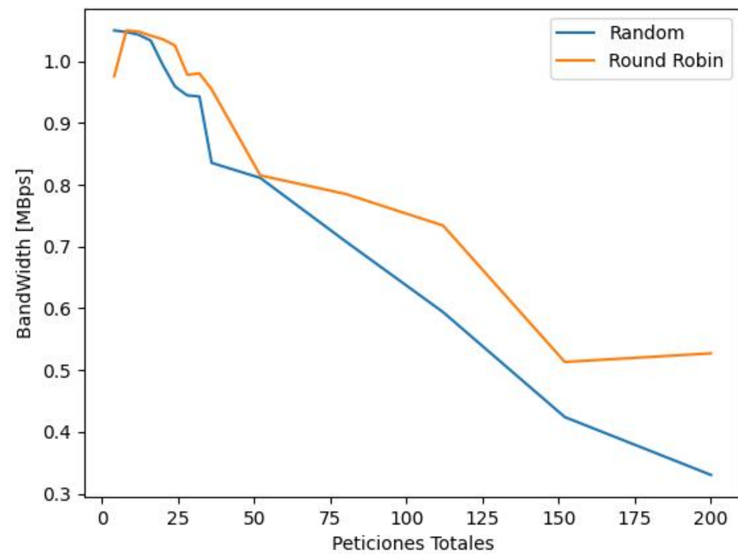


Figura 5.12: Velocidad de transmisión.

Capítulo 6

Conclusiones

6.1. Conclusiones generales

Este trabajo propone el análisis de dos balanceadores de carga (Aleatorio y Round-robin) en una red SDN bajo la inspección del controlador Ryu. A través de los resultados mostrados en las figuras de la Sección 5.3 se puede concluir que el desempeño del balanceador Round-robin supera al desempeño del obtenido por el balanceador Aleatorio, considerando que ambos experimentos fueron realizados bajo las mismas condiciones y bajo la misma topología de red.

Cabe resaltar que el comportamiento del algoritmo Round-robin, al recibir 50 peticiones por servidor, llegan a un punto límite en donde todos los servidores comienzan a degradar su rendimiento. En los resultados promediados de la Sección 5.3, puede observarse el comportamiento de cada algoritmo bajo diferentes métricas. Más aún, las figuras 5.11 y 5.12, correspondientes al *throughput* y a la velocidad de transmisión. Estas figuras muestran que al llegar 150 peticiones, el balanceador Round-robin tiene una pendiente casi nula, manteniendo un valor casi constante de 0.52 Mbps, como se puede ver en la tabla 5.4.

Por otro lado, el balanceador Aleatorio no presenta un tope máximo de peticiones (al menos hasta las 200 peticiones en los experimentos) como lo presenta el balanceador Round-robin. Sin embargo, para 150 peticiones totales, el número de paquetes perdidos ronda el 70 %, mientras para el balanceador Round-robin ronda el 57 %. Aunque para ambos casos el número de paquetes perdidos es muy alto, el balance Round-robin permite aceptar un 13 % más de paquetes. Y en lo referente a la velocidad de transmisión y el *throughput* (figuras 5.11 y 5.12) se observa que el desempeño del balanceador Round-robin cuenta con un mejor rendimiento de red en comparación con el balance Aleatorio.

Basándose en el análisis de resultados, y apoyándonos de los diferentes datos recabados, podemos concluir que el balanceador Round-robin presenta un mejor desempeño en todas las métricas analizadas, así como también confirmar la hipótesis planteada en la sección 1.3. De acuerdo a los resultados obtenidos se comprueba que el algoritmo Round-robin presenta un mejor encaminamiento al repartir eficientemente el tráfico, evitando la saturación de los servidores considerados. Sin embargo, el algoritmo aleatorio también cuenta con una técnica útil para evitar la saturación de una red con menor cantidad de servidores, por lo que el balance de cargas a través de un diseño de software se puede adecuar a las necesidades de la red mucho mejor que en una red convencional, con un menor costo y mayor eficacia.

Es posible implementar algoritmos más complejos en un trabajo futuro que considere el número de peticiones que atiende cada servidor por minuto, la memoria de cada servidor, o el número de procesos que tenga cada servidor. De igual manera, es posible agregar un mayor número de servidores o modificar la topología para visualizar los efectos que generen en los tipos de balanceadores mencionados en este trabajo.

Apéndice A

Datos balanceador random

Tabla A1: Server 2.

Peticiones Recibidas	Troughput [Mbps]	Bandwidth [Mbps]	Porcentaje de perdidas
1.8	1.272 888 889	1.05	0.697 037 590 2
2.6	1.269 846 154	1.048 461 538	0.862 068 965 5
3.8	1.253 684 211	1.042 631 579	2.135 441 246
5	1.227 12	1.027 52	4.230 148 772
6.4	1.146	0.995 968 75	10.533 688 19
8.6	1.102 930 233	0.963 790 697 7	13.887 296 04
8.4	1.069 142 857	0.941 047 619	16.543 038 58
12.2	1.087 278 689	0.948 065 573 8	15.133 367 62
12.4	0.878 093 548 4	0.819 887 096 8	31.459 861 93
19	0.858 981 052 6	0.814 821 052 6	32.930 599 15
29	0.732 438 620 7	0.696 606 896 6	42.814 075 01
37.6	0.549 471 276 6	0.564 621 276 6	57.092 253 56
53.4	0.358 172 734 1	0.429 534 082 4	72.032 510 77
62.4	0.325 675 705 1	0.348 125 673 1	74.574 127 96

Tabla A2: Server 3.

Peticiones Recibidas	Troughput [Mbps]	Bandwidth [Mbps]	Porcentaje de perdidas
1	1.2716	1.05	0.672 645 739 9
2	1.2674	1.047	1.053 811 659
3.4	1.258	1.043 529 412	1.793 485 428
5	1.238 16	1.0352	3.317 790 531
5.2	1.141 769 231	0.983 076 923 1	10.869 752 61
7.8	1.080 871 795	0.957 487 179 5	15.601 287 65
10.8	1.094 407 407	0.948	14.591 027 93
9.8	1.102 204 082	0.951 571 428 6	13.951 148 11
9.4	0.872 382 978 7	0.816 914 893 6	31.912 458 88
14.6	0.758 106 849 3	0.776 534 246 6	40.813 632 18
29	0.733 020 689 7	0.712 337 931	42.770 796 5
38.2	0.594 415 706 8	0.596 528 795 8	53.578 011 81
50.8	0.351 988 346 5	0.418 441 496 1	72.521 158 95
67	0.297 791 283 6	0.331 851 850 7	76.755 013 85

Tabla A3: Random Promedios.

Peticiones Recibidas	Troughput [Mbps]	Bandwidth [Mbps]	Porcentaje de perdidas
1.333 333 333	1.272 385 185	1.05	0.668 240 851 1
2.666 666 667	1.267 748 718	1.047 506 787	1.034 144 344
4	1.256 561 404	1.043 164 775	1.913 495 681
5.333 333 333	1.237 848 889	1.033 684 444	3.365 125 641
6.666 666 667	1.154 272 283	0.993 753 319 6	9.899 957 156
8	1.092 004 185	0.958 943 502 9	14.742 822 7
9.333 333 333	1.084 107 664	0.944 894 660 9	15.387 273 41
10.666 666 67	1.083 000 923	0.943 039 000 8	15.460 305 54
12	0.904 619 875 2	0.835 652 306 7	29.393 264 87
17.333 333 33	0.834 422 054 3	0.811 143 795 4	34.851 377 79
28.6	0.735 630 801 3	0.708 478 012 1	42.563 462 83
37.333 333 33	0.588 227 889 5	0.594 031 607 9	54.063 219 62
50.666 666 67	0.346 928 672 6	0.424 016 475 9	72.912 279 58
62.6	0.298 062 763 4	0.330 336 115 2	76.732 350 62

Apéndice B

Datos balanceador Round-robin

Tabla B1: Server 1.

Peticiones Recibidas	Troughput [Mbps]	Bandwidth [Mbps]	Porcentaje de perdidas
1.8	1.403 333 333	0.827 155 56	28.765 296
2.6	1.243 846 154	1.05	0.482 925 15
4	1.242	1.048	0.689 461 88
6	1.233 666 67	1.041 333 33	1.416 292 97
6.6	1.223 333 33	1.035 333 33	2.150 428 05
8.4	1.204 761 9	1.024 333 33	3.638 159 3
8.8	1.153 227 27	0.991 818 18	7.811 863 02
9.2	1.135 478 26	0.980 086 96	9.424 351 73
12.4	1.090 080 65	0.961 451 61	13.369 738 2
18.6	0.923 796	0.812 473 12	27.081 83
27.4	0.888 686 13	0.790 379 56	29.927 007 3
39.4	0.712 545 92	0.733 438 78	43.977 647 1
52.4	0.520 517 18	0.515 954 2	59.346 428 2
55	0.529 789 45	0.5312	58.596 765 8

Tabla B2: Server 2.

Peticiones Recibidas	**Troughput [Mbps]**	**Bandwidth [Mbps]**	**Porcentaje de perdidas**
1	1.25	1.05	0.238 228 699 6
2.4	1.243 333 333	1.05	0.482 062 780 3
4	1.243	1.049	0.622 197 309 4
5.2	1.231 695 238	1.041 771 429	1.433 482 81
7.4	1.225 333 333	1.035 778 571	2.039 557 976
8.2	1.206 583 333	1.026 747 222	3.582 149 975
10.6	1.142 246 97	0.952 242 924	8.905 365 312
11.6	1.130 95	0.979 073 333 3	9.707 772 795
11	1.077 020 925	0.936 586 045 1	14.431 036 83
17.8	0.922 945 858 7	0.817 760 930 1	27.206 177 21
26	0.760 643 649 6	0.775 784 067 1	30.821 536 27
35.6	0.706 301 846 1	0.729 103 134 9	44.498 609 85
49	0.523 012 500 1	0.516 705 364 5	59.151 888 84
55.4	0.541 880 053 5	0.529 311 961 6	57.661 903 16

Tabla B3: Server 3.

Peticiones Recibidas	Troughput [Mbps]	Bandwidth [Mbps]	Porcentaje de perdidas
1.2	1.25	1.05	0.186 846 038 9
3	1.242 666 667	1.05	0.500 747 384 2
4	1.2415	1.0475	0.734 304 932 7
4.8	1.2325	1.041 666 667	1.359 304 933
6	1.223 333 333	1.036	2.171 150 972
7.4	1.209 189 189	1.025 324 324	3.308 689 856
8.6	1.150 232 558	0.990 976 744 2	8.288 142 663
9	1.1374	0.981 822 222 2	9.170 403 587
12.6	1.095 682 54	0.965 904 761 9	12.862 125 42
15.6	0.918 589 743 6	0.815 525 641	27.654 651 03
26.6	0.885 774 436 1	0.789 646 616 5	30.132 506 15
37.4	0.718 731 182 8	0.740 069 892 5	43.460 991 37
50.6	0.517 597 628 5	0.507 051 383 4	59.555 415 74
50.6	0.521 230 434 8	0.521 067 193 7	59.286 714 4

Apéndice C

Código en Python implementado en el controlador para el balanceador Round-robin.

```
1  #    /* -*- python -*- */
2  # Licensed under the Apache License, Version 2.0 (the "License");
3  # you may not use this file except in compliance with the License.
4  # You may obtain a copy of the License at
5  #
6  #    http://www.apache.org/licenses/LICENSE-2.0
7  #
8  # Unless required by applicable law or agreed to in writing, software
9  # distributed under the License is distributed on an "AS IS" BASIS,
10 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
11 # implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15 from ryu.base import app_manager
16 from ryu.controller import ofp_event
17 from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
18 from ryu.controller.handler import set_ev_cls
19 from ryu.ofproto import ofproto_v1_3
20 from ryu.lib.packet import packet
21 from ryu.lib.packet import ethernet
22 from ryu.lib.packet import ether_types
23
24 from ryu.lib.packet import in_proto
25 from ryu.lib.packet import ipv4
26 from ryu.lib.packet import icmp
27 from ryu.lib.packet import tcp
28 from ryu.lib.packet import udp
29
30 from rasps import *
31 #from random import *
32 import random
33
34 class SimpleSwitch13(app_manager.RyuApp):
35     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
36
37     def __init__(self, *args, **kwargs):
```

```
38          super(SimpleSwitch13, self).__init__(*args, **kwargs)
39          self.mac_to_port = {}
40          self.counter=0
41
42      @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
43      def switch_features_handler(self, ev):
44          datapath = ev.msg.datapath
45          ofproto = datapath.ofproto
46          parser = datapath.ofproto_parser
47          idle_timeout = 0
48
49          match = parser.OFPMatch()
50          actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
51                                            ofproto.OFPCML_NO_BUFFER)]
52          self.add_flow(datapath, 0, idle_timeout,match, actions)
53
54      def add_flow(self, datapath, priority, idle_timeout, match, actions, buffer_id=None):
55          ofproto = datapath.ofproto
56          parser = datapath.ofproto_parser
57
58          inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
59                                               actions)]
60          if buffer_id:
61              mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
62                                      priority=priority,
63                                      idle_timeout=idle_timeout,
64                                      match=match,
65                                      instructions=inst)
66          else:
67              mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
68                                      idle_timeout=idle_timeout,
69                                      match=match, instructions=inst)
70          datapath.send_msg(mod)
71
72      @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
73
74
75      def _packet_in_handler(self, ev):
76
77          if ev.msg.msg_len < ev.msg.total_len:
78              self.logger.debug("packet truncated: only %s of %s bytes",
79                                ev.msg.msg_len, ev.msg.total_len)
80          msg = ev.msg
81          datapath = msg.datapath
82          ofproto = datapath.ofproto
83          parser = datapath.ofproto_parser
84          in_port = msg.match['in_port']
85
86          pkt = packet.Packet(msg.data)
87          eth = pkt.get_protocols(ethernet.ethernet)[0]
88          pkt_eth=pkt.get_protocol(ethernet.ethernet)
89          if eth.ethertype == ether_types.ETH_TYPE_LLDP:
90              # ignore lldp packet
91              return
92          dst = eth.dst
93          src = eth.src
94
95          destinoeth=pkt_eth.dst
```

```
96
97          dpid = datapath.id
98          self.mac_to_port.setdefault(dpid, {})
99
100         self.mac_to_port[dpid][src] = in_port
101         self.mac_to_port[dpid][switchs.mac]=switchs.port
102         pkt_ip=pkt.get_protocol(ipv4.ipv4)
103
104
105         if pkt_ip:
106             self.logger.info('Paquete IP recibido de: %s
107             hacia: %s ',pkt_ip.src,pkt_ip.dst)
108             protocol=pkt_ip.proto
109
110             if pkt_ip.dst==switchs.ip:
111
112                 if protocol == in_proto.IPPROTO_TCP:
113                     pkt_tcp=pkt.get_protocol(tcp.tcp)
114
115                 elif protocol == in_proto.IPPROTO_UDP:
116                     pkt_udp=pkt.get_protocol(udp.udp)
117                     self.logger.info('PETICION UDP')
118                     if pkt_udp.dst_port == 5001:
119                         self.logger.info('PETICION HACIA IPERF')
120                         self.logger.info('\t\t IPDESTINO: %s
121                         \n\t\tIPFUENTE: %s
122                         \n\t\tSRC_PRT=%s
123                         \n\t\t DST_PRT= %s',
124                         pkt_ip.dst, pkt_ip.src, pkt_udp.src_port, pkt_udp.dst_port)
125                         selserver= RR(self.counter)
126                         self.counter=self.counter+1
127                         self.logger.info('\n\nServidor seleccionado: IP: %s,
128                         \tPuerto: %s,
129                         \tMAC: %s',
130                         selserver.ip,selserver.port,selserver.mac)
131                         self.logger.info('Cambiando de la IP: %s,
132                         \tPuerto: %s,
133                         \tMAC: %s,
134                         \n MacOrigen:%s',
135                         pkt_ip.dst, switchs.port,pkt_eth.dst,pkt_eth.src)
136
137
138                         match=parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,
139                                                 ipv4_src=pkt_ip.src,
140                                                 ipv4_dst=pkt_ip.dst,
141                                                 ip_proto=protocol,
142                                                 udp_src=pkt_udp.src_port,
143                                                 udp_dst=pkt_udp.dst_port)
144
145                         reversematch = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,
146                                                             ipv4_src=selserver.ip,
147                                                             #desde el servidor
148                                                             ipv4_dst=pkt_ip.src,
149                                                             #hacia el cliente
150                                                             ip_proto=protocol,
151                                                             udp_src=pkt_udp.dst_port,
152                                                             udp_dst=pkt_udp.src_port )
153
154                         action=[parser.OFPActionSetField(eth_dst=selserver.mac),
```

```
155                                 parser.OFPActionSetField(ipv4_dst=selserver.ip),
156                                 parser.OFPActionOutput(selserver.port)]
157 
158                         reverseaction=[parser.OFPActionSetField(eth_src=switchs.mac),
159                                         parser.OFPActionSetField(ipv4_src=switchs.ip),
160                                         parser.OFPActionOutput(in_port)]
161 
162 
163 
164 
165                         self.add_flow(datapath, 1, idle_timeout=10,
166                                         match=match,actions=action)
167                         self.add_flow(datapath, 1, idle_timeout=10 ,
168                                         match=reversematch,actions=reverseaction)
169 
170 
171 
172 
173         if dst in self.mac_to_port[dpid]:
174             out_port = self.mac_to_port[dpid][dst]
175 
176         else:
177             out_port = ofproto.OFPP_FLOOD
178 
179         actions = [parser.OFPActionOutput(out_port)]
180         data = None
181         if msg.buffer_id == ofproto.OFP_NO_BUFFER:
182             data = msg.data
183 
184         out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
185                                   in_port=in_port, actions=actions, data=data)
186         datapath.send_msg(out)
```

Apéndice D

Programa en Python implementado en el controlador para el balanceador aleatorio

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types

from ryu.lib.packet import in_proto
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp
from ryu.lib.packet import tcp
from ryu.lib.packet import udp

from rasps import *
import random

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
```

```
38            self.counter=0
39
40        @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
41        def switch_features_handler(self, ev):
42            datapath = ev.msg.datapath
43            ofproto = datapath.ofproto
44            parser = datapath.ofproto_parser
45            idle_timeout = 0
46
47
48            match = parser.OFPMatch()
49            actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
50                                              ofproto.OFPCML_NO_BUFFER)]
51            self.add_flow(datapath, 0, idle_timeout,match, actions)
52
53        def add_flow(self, datapath, priority, idle_timeout, match, actions, buffer_id=None):
54            ofproto = datapath.ofproto
55            parser = datapath.ofproto_parser
56
57            inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
58                                                 actions)]
59            if buffer_id:
60                mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
61                                        priority=priority,
62                                        idle_timeout=idle_timeout,
63                                        match=match,
64                                        instructions=inst)
65            else:
66                mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
67                                        idle_timeout=idle_timeout,
68                                        match=match, instructions=inst)
69            datapath.send_msg(mod)
70
71        @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
72
73
74        def _packet_in_handler(self, ev):
75            if ev.msg.msg_len < ev.msg.total_len:
76                self.logger.debug("packet truncated: only %s of %s bytes",
77                                  ev.msg.msg_len, ev.msg.total_len)
78            msg = ev.msg
79            datapath = msg.datapath
80            ofproto = datapath.ofproto
81            parser = datapath.ofproto_parser
82            in_port = msg.match['in_port']
83
84            pkt = packet.Packet(msg.data)
85            eth = pkt.get_protocols(ethernet.ethernet)[0]
86            pkt_eth=pkt.get_protocol(ethernet.ethernet)
87            if eth.ethertype == ether_types.ETH_TYPE_LLDP:
88                # ignore lldp packet
89                return
90            dst = eth.dst
91            src = eth.src
92
93            destinoeth=pkt_eth.dst
94
95            dpid = datapath.id
```

```
96          self.mac_to_port.setdefault(dpid, {})
97
98
99          self.mac_to_port[dpid][src] = in_port
100         self.mac_to_port[dpid][switchs.mac]=switchs.port
101
102         pkt_ip=pkt.get_protocol(ipv4.ipv4)
103
104
105         if pkt_ip:
106             protocol=pkt_ip.proto
107
108             if pkt_ip.dst==switchs.ip:
109
110                 if protocol == in_proto.IPPROTO_TCP:
111                     pkt_tcp=pkt.get_protocol(tcp.tcp)
112
113
114                 elif protocol == in_proto.IPPROTO_UDP:
115                     pkt_udp=pkt.get_protocol(udp.udp)
116                     if pkt_udp.dst_port == 5001:
117
118                         selserver=random.choice(lservers)
119
120                         match=parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,
121                                               ipv4_src=pkt_ip.src,
122                                               ipv4_dst=pkt_ip.dst,
123                                               ip_proto=protocol,
124                                               udp_src=pkt_udp.src_port,
125                                               udp_dst=pkt_udp.dst_port)
126
127                         reversematch = parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP,
128                                                     ipv4_src=selserver.ip,
129                                                     #desde el servidor
130                                                     ipv4_dst=pkt_ip.src,
131                                                     #hacia el cliente
132                                                     ip_proto=protocol,
133                                                     udp_src=pkt_udp.dst_port,
134                                                     udp_dst=pkt_udp.src_port )
135
136                         action=[parser.OFPActionSetField(eth_dst=selserver.mac),
137                                 parser.OFPActionSetField(ipv4_dst=selserver.ip),
138                                 parser.OFPActionOutput(selserver.port)]
139
140                         reverseaction=[parser.OFPActionSetField(eth_src=switchs.mac),
141                                         parser.OFPActionSetField(ipv4_src=switchs.ip),
142                                         parser.OFPActionOutput(in_port)]
143
144                         self.add_flow(datapath, 1, idle_timeout=10 ,
145                                     match=match,actions=action)
146                         self.add_flow(datapath, 1, idle_timeout=10 ,
147                                     match=reversematch,actions=reverseaction)
148
149
150
151         out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
152                                   in_port=in_port, actions=actions, data=data)
153         datapath.send_msg(out)
```

Bibliografía

[1] L. Ochoa-Aday, C. Cervelló-Pastor, and A. Fernández-Fernández, “Current trends of topology discovery in openflow-based software defined networks,” *Department of Network Engineering at Universitat Politècnica de Catalunya (UPC)*, p. 6, 09 2015.

[2] F. Candelas and J. Pomares, “Protocolos de transporte tcp y udp. 2009,” Público, 2009, manual de práctica. [Online]. Available: https://rua.ua.es/dspace/bitstream/10045/11606/1/Pr3-2009-10.pdf

[3] L. Lisandro, “Frameworks,” in *Framework para el Desarrollo Ágil de Sistemas Web*, 2009, tesina 3, pp. 16–17. [Online]. Available: http://sedici.unlp.edu.ar/bitstream/handle/10915/4000/Tesis.%20Framework%20para%20el%20desarrollo%20%C3%A1gil%20de%20sistemas%20web.pdf-PDFA1b.pdf?sequence=2&isAllowed=y#:~:text=El%20Framework%20propuesto%20tiene%20por,herramienta%20de%20prototipaci%C3%B3n%20si%20el

[4] R. García, “Curso de doctorado web semántica: Tecnologías semánticas aplicadas a la definición de qos,” Universidad de Oviedo, Tech. Rep., 2008. [Online]. Available: http://di002.edv.uniovi.es/~labra/cursos/Doc08UniOvi/artiRodrigo.pdf

[5] E. Tamariz, M. Coyotecatl, R. Torrealba, and R. Ambrosio, “Análisis del parámetro throughput en una red ad hoc y manet en el estándar 802.11ac,” *Revista*, vol. 3, no. 7 1-9v, p. 4, 2017. [Online]. Available: https://www.ecorfan.org/spain/researchjournals/Aplicacion_Cientifica_y_Tecnica/vol3num7/Revista_de_Aplicacion_Cientifica_y_Tecnica_V3_N7_1.pdf

[6] R. Kanu, S. Kuyoro, S. Ogunlere, and A. Adegbenjo, “Management and control of bandwidth in computer networks,” *International Journal of Computer Networks and Wireless Communications (IJCNWC)*, vol. 2, 07 2012.

[7] H. Kavana, V. Kavya, B. Madhura, and K. Neha, “Load balancing using sdn methodology,” *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT)*, vol. 07, no. 05, 2018. [Online]. Available: https://www.ijert.org/research/load-balancing-using-sdn-methodology-IJERTV7IS050103.pdf

[8] “Switching & routing,” 2022. [Online]. Available: https://www.gfr.com.mx/portafolio/switching-routing#:~:text=La%20principal%20diferencia%20entre%20un,t%C3%ADpicos%20utilizan%20enrutamiento%20de%20software.

[9] V. O., “Reglas openflow: Mejorando el algoritmo de detección de interacciones,” Master’s thesis, Instituto Tecnológico de Costa Rica. Escuela de Ingeniería en Computación, 2015. [Online]. Available: https://repositoriotec.tec.ac.cr/bitstream/handle/2238/6677/Oscar_Mario_Vasquez_Leiton.pdf?sequence=1&isAllowed=y

[10] S. L. Anees Al-Najjar and M. Portmann, “Pushing sdn to the end-host, network load balancing using openflow,” *The Thirteenth IEEE International Workshop on Managing Ubiquitous Communications and Services*, vol. 1, pp. 3,4, 2016.

[11] U. Zakia and H. B. Yedder, “Dynamic load balancing in sdn-based data center networks,” *Simon Fraser University*, vol. 1, pp. 244–246, 2017.

[12] H. Noman and M. Jasim, "Load balancing using sdn methodology," *International Journal of Engineering Research & Technology (IJERT)*, vol. 7, pp. 244–246, 05 2018.

[13] L. Foundation, "Open vswitch," 2016, accedido en diciembre de 2021. [Online]. Available: https://www.openvswitch.org/

[14] Raspberry, "Raspberry pi," accedido en febrero de 2022. [Online]. Available: https://www.raspberrypi.com/for-home/

[15] Raspbian, "Raspbian," accedido en marzo de 2022. [Online]. Available: https://www.raspberrypi.com/software//

[16] A. Rawal, "Introduction to openflow protocolh," 2021, accedido en marzo de 2022. [Online]. Available: https://www.section.io/engineering-education/openflow-sdn/

[17] L. Larry, C. Vinton, C. David, K. Robert, K. Leonard, L. Daniel, P. Jon, R. Larry, and S. Wolff, "Breve historia de internet," 2017. [Online]. Available: https://www.internetsociety.org/es/internet/history-internet/brief-history-internet/

[18] "Openflow switch specification version 1.3.2 (wire protocol 0x04)," Abril 2013, accedido en mayo de 2022. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[19] D. Verma, "Software defined load balancing over an openflow-enabled network," Master's thesis, UNIVERSITY OF TEXAS, 2012. [Online]. Available: 'https://rc.library.uta.edu/uta-ir/bitstream/handle/10106/26824/VERMA-THESIS-2017.pdf?sequence=1'

[20] P. Farzaneh, "Comparison of software defined networking (sdn) controllers. part 3: Opendaylight (odl)," 2019, accedido en diciembre de 2021. [Online]. Available: https://aptira.com/comparison-of-software-defined-networking-sdn-controllers-part-3-opendaylight-odl/

[21] Pakzad, "Comparison of software defined networking (sdn) controllers. part 2: Open network operating system (onos)," 2019, accedido en diciembre de 2021. [Online]. Available: https://aptira.com/comparison-of-software-defined-networking-sdn-controllers-part-/2-open-network-operating-system-onos/

[22] ——, "Comparison of software defined networking (sdn) controllers. part 5: Ryu," 2019, accedido en enero de 2022. [Online]. Available: https://aptira.com/comparison-of-software-defined-networking-sdn-controllers-part-5-ryu/

[23] ——, "Comparison of software defined networking (sdn) controllers. part 6: Faucet," 2019, accedido en febrero de 2022. [Online]. Available: https://aptira.com/comparison-of-software-defined-networking-sdn-controllers-part-6-faucet/

[24] H. Noman and M. Jasim, "Pox controller and open flow performance evaluation in software defined networks (sdn) using mininet emulator," *IOP Conference Series: Materials Science and Engineering*, vol. 881, p. 9, 08 2020.

[25] Raspberry, "Raspberry pi imager," accedido en febrero de 2022. [Online]. Available: https://www.raspberrypi.com/news/raspberry-pi-imager-imaging-utility/

[26] RYU, "Ryu getting started ryu documentation." 2014, accedido en mayo de 2022. [Online]. Available: https://ryu.readthedocs.io/en/latest/getting_started.html

[27] iPerf, "The ultimate speed test tool for tcp, udp and sctp." 2014, accedido en noviembre de 2021. [Online]. Available: https://iperf.fr/iperf-doc.php

[28] K. Technologies, "Load balancing algorithms and techniques," 2019, last accessed 16 February 2022. [Online]. Available: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/