UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CARRERA DE INGENIERÍA EN ELECTRÓNICA Y REDES DE COMUNICACIÓN

“DESARROLLO DE UN ALGORITMO DE BALANCEO DE CARGA DINÁMICO DE MÚLTIPLES RUTAS PARA LA REDISTRIBUCIÓN DE FLUJOS EN UNA RED SDN QUE PERMITA MEJORAR EL RENDIMIENTO EN EL ENRUTAMIENTO DE PAQUETES TCP/IP.”

TRABAJO DE GRADO PREVIO A LA OBTENCIÓN DEL TITULO DE INGENIERÍA EN ELECTRÓNICA Y REDES DE COMUNICACIÓN

AUTOR: NEJER HARO DIEGO FERNANDO

DIRECTOR: MSC. CARLOS ALBERTO VÁSQUEZ AYALA

Ibarra-Ecuador

2023

UNIVERSIDAD TÉCNICA DEL NORTE

BIBLIOTECA UNIVERSITARIA

AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicada en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

DATOS DEL CONTACTO			
CÉDULA DE IDENTIDAD:	100468310-6		
APELLIDOS Y NOMBRES:	Néjer Haro Diego Fernando		
DIRECCIÓN:	Zumba 4-43 y Macas (Alpachaca)		
E-MAIL:	dfnejerh@utn.edu.ec		
TELÉFONO FIJO:	062-602-566	**TELÉFONO MÓVIL:**	0988252447
DATOS DE LA OBRA			
TÍTULO:	DESARROLLO DE UN ALGORITMO DE BALANCEO DE CARGA DINÁMICO DE MÚLTIPLES RUTAS PARA LA REDISTRIBUCIÓN DE FLUJOS EN UNA RED SDN QUE PERMITA MEJORAR EL RENDIMIENTO EN EL ENRUTAMIENTO DE PAQUETES TCP/IP.		
AUTOR (ES):	Néjer Haro Diego Fernando		
FECHA: DD/MM/AAAA	15/03/2023		
PROGRAMA:	☒ PREGRADO ☐ POSGRADO		
TÍTULO POR EL QUE OPTA:	Ingeniero en Electrónica y Redes de Comunicación		
ASESOR/DIRECTOR:	Msc. Carlos Alberto Vásquez Ayala		

2. CONSTANCIAS

El autor manifiesta que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto, la obra es original y que es la titular de los derechos patrimoniales, por lo que asume la responsabilidad sobre el contenido de la misma y saldrá en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 15 días del mes de marzo del 2023.

EL AUTOR:

………………………………………

Néjer Haro Diego Fernando

CI: 100468310-6

UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CERTIFICACIÓN

MAGISTER CARLOS VÁSQUEZ, DIRECTOR DEL PRESENTE TRABAJO DE TITULACIÓN CERTIFICA:

Que, el presente trabajo de Titulación "DESARROLLO DE UN ALGORITMO DE BALANCEO DE CARGA DINÁMICO DE MÚLTIPLES RUTAS PARA LA REDISTRIBUCIÓN DE FLUJOS EN UNA RED SDN QUE PERMITA MEJORAR EL RENDIMIENTO EN EL ENRUTAMIENTO DE PAQUETES TCP/IP.". Ha sido desarrollado por el señor Néjer Haro Diego Fernando bajo mi supervisión.

Es todo en cuanto puedo certificar en honor de la verdad.

Msc. Carlos Vásquez

100242498-2

DIRECTOR

DEDICATORIA

El presente trabajo de titulación quiero dedicarlo en primer lugar a Dios por bendecirme cada instante de mi vida personal y profesional, a mi papito Ernesto, a mis hermanos Verito, Edwin, Xavier por ser el pilar fundamental para alcanzar esta meta que creía casi imposible de conseguir, pero de manera más especial a mi mamita preciosa Blanca quien hasta el último momento de mi carrera universitaria me brindo su apoyo y que cuando menos creía yo en mí, ella me brindaba sus palabras de aliento para continuar y no decaer.

Y por último y no menos importantes, a mis mejores amigos: mi confidente Mafer, mi superamigo Cristian, mi consejera de malas decisiones Ibethcita y al inolvidable grupo de "Los Avengers" Anthony, Brayan, David, Edwin y Santiago.

Diego Fernando Nejer Haro

AGRADECIMIENTOS

Agradezco a Dios por la salud, por el don del conocimiento, por brindarme seguridad de que si algo me propongo lo consigo sin importar los obstáculos y por la persona que me he convertido.

A mi mamita y papito por ese amor y confianza que me dieron en toda mi carrera, pero en especial a mi madre que cuando me sentía derrotado ella me levantaba y me brindaba sus palabras de aliento.

A mis hermanos Verito, Edwin y Xavier por ese cariño y apoyo incondicional que me ofrecen desde que llegue a sus vidas como su llucho y siempre les viviré agradecido.

A mis sobrinos Junior, Gabriel, Benjamín, Erick e Isaac, por hacerme sentir el tío más feliz y que no me hace falta nada más en este mundo cuando los tengo a mi lado.

A mis amigos de la universidad Anthony, Brayan, David, Edwin, Richard y Santiago por esos desvelos, risas, locuras y más que todo por brindarme su amistad y sé que a pesar de que algunos tomamos caminos diferentes solo basta una fecha especial para volvernos a ver.

A mis amigos de vida Cristian, Ibethcita, Maferes, decirles gracias porque más que ser unos amigos son como mi familia, a quien confió totalmente y en quienes puede refugiarme en mis momentos más difíciles.

A mi padrecito Ruperto Colimba por brindarme esa confianza única y ver en mi algo tan especial que hasta yo mismo creí a ver perdido.

Finalmente, a mis docentes que me formaron como profesional desde mi ingreso a la Universidad y en especial a mi director Msc. Carlos Vázquez por su tiempo, paciencia y preocupación, durante todo este trayecto de titulación. Muchas gracias a todos.

Diego Fernando Nejer Haro

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

En el presente trabajo de titulación consiste en el desarrollo de un algoritmo de balanceo de carga dinámico de múltiples rutas para la redistribución de flujos en una red SDN que permita el mejorar el rendimiento en el enrutamiento de paquetes TCP/IP. El sistema propuesto se integra de dos redes SDN o testbeds emulada por la herramienta Mininet. Los testbeds de prueba se componen de un primer escenario con 5 equipos OpenvSwitch con múltiples enlaces y el segundo escenario una red en cascada con diversos equipos OpenvSwitch, aplicando servidores de VoIP y Streaming de Video, mismas que será administrada y gestionada por el controlador RYU.

En la selección del emulador y controlador de la red SDN se realizó conforme a las pautas presentadas en la norma ISO/IEC/IEEE 29148 para de esta forma asegurar la operatividad de las herramientas para el levantamiento de los testbeds propuestos. Para la evaluación del rendimiento de las redes se aplica conforme al modelo de UF1880 Gestión de redes telemáticas; donde se presenta las métricas a evaluar antes y después de aplicar el algoritmo de balanceo en los escenarios de prueba del presente trabajo de titulación.

El desarrollo y programación del algoritmo de balanceo se lo realiza en base al algoritmo DiGraph para la detección de rutas para posteriormente validar los anchos de banda de los enlaces disponibles y realizar la selección de la mejor ruta dentro de la red SDN. Posteriormente y para finalizar se aplica el algoritmo en los testbed de prueba para la obtención de resultados de las métricas de rendimiento como anchos de banda, paquetes perdidos, tasas de transferencia, latencia y comprobar la mejora con la aplicación del algoritmo desarrollado.

ABSTRACT

This graduation thesis involves the development of a dynamic load balancing algorithm with multiple paths for the redistribution of flows in an SDN network to improve the performance in TCP/IP packet routing. The proposed system integrates two SDN networks or testbeds emulated by the Mininet tool. The testbeds consist of a first scenario with 5 OpenvSwitch devices with multiple links, and the second scenario is a cascaded network with various OpenvSwitch devices, applying VoIP and Video Streaming servers, all of which will be managed and controlled by the RYU controller.

The selection of the SDN network emulator and controller was carried out in accordance with the guidelines presented in the ISO/IEC/IEEE 29148 standard to ensure the operability of the tools for the deployment of the proposed testbeds. To evaluate the performance of the networks, the UF1880 Telematic Network Management model is applied, which presents the metrics to be evaluated before and after applying the balancing algorithm in the test scenarios of this graduation thesis.

The development and programming of the balancing algorithm are based on the DiGraph algorithm for route detection, followed by validating the available link bandwidths and selecting the best path within the SDN network. Afterwards, the algorithm is applied to the testbeds to obtain performance metric results such as bandwidth, packet loss, transfer rates, and latency. The results are used to verify the improvement obtained with the application of the developed algorithm.

Capítulo 1. Antecedentes

1.1. Introducción

En este capítulo se presenta los ítems principales que motivó al desarrollo del este proyecto de investigación y titulación, los cuáles comprenden: La problemática que hace referencia a las complicaciones presentadas actualmente en las redes SDN y de forma general como se pretende dar solución a dicho problema; Una sección de dedicada a la presentación de objetivos propuestos a alcanzar para culminar con el desarrollo del presente proyecto; La sección de alcance donde se presenta de forma detalla las delimitaciones de esta propuesta y finalmente la justificación tanto tecnológica, teórica y metodológica que dan validación y viabilidad al desarrollo de este tema de titulación.

1.2. Problema

Con el gran crecimiento de la red de datos y la aparición de tecnologías emergentes como Cloud Computing y Big Data, las redes tradicionales son poco eficientes en la administración y en el manejo de flujos de tráfico excesivos (Neghabi, A., et al, 2018) , debido a que el plano de datos y el plano de control es manejado por equipos propietarios, donde dicho elemento tiene una perspectiva individual de la topología de red, es decir, toma sus propias decisiones de enrutamiento en base a sus políticas o protocolos de enrutamiento previamente configurados; bajo esta lógica, los elementos de red tienden a ser predecibles y aplicar los mismos criterios o mecanismos en cualquier circunstancia, ya sea una red sin sobrecarga o en una red sobrecargada de peticiones en dónde se tiene mayor pérdida de paquetes, retardo en las comunicaciones u otros aspectos. (Ramírez, M. & López, A., 2018)

Según varias aportaciones en el área científica de las redes de nueva generación, se ha trabajado arduamente en mecánicos y tecnologías como son las Redes Definidas por Software (SDN). Las redes SDN se plantean como una alternativa para reducir costos operativos y ajustarse a las necesidades reales de los entornos empresariales y Proveedores de Servicio de Internet (ISP’s) (Sayans, 2018). Al ser una red totalmente programable, una red SDN se caracteriza por deslindar el plano de control de los equipos de hardware, eliminando así las decisiones de control como el enrutamiento y habilitando tablas de flujo programables mediante el protocolo estándar OpenFlow. El comportamiento de la red se maneja mediante el uso de un controlador central que permite generar y establecer políticas de enrutamiento que serán anunciadas a los conmutadores y publicadas en sus tablas de flujo para determinar cómo procesar un determinado flujo hasta su destino. (Fernández, 2015)

Sin embargo, al igual que las redes tradicionales, una Red Definida por Software (SDN) presenta complicaciones especialmente en aspectos relacionados al enrutamiento de paquetes; por ejemplo, cuando se inicia un nuevo flujo en el plano de datos y no existe una política de enrutamiento dentro de las tablas de flujo, el dispositivo de conmutación reenvía el primer paquete de ese flujo al controlador SDN donde se asignará una ruta de reenvió relevante para el flujo. Una política de reenvío se define para cada flujo en las tablas de flujo del equipo SDN (Switch Openflow) (Ramírez, M. & López, A., 2018), por tal motivo existe una cantidad excesiva de paquetes que utilizan un flujo en particular lo cual provoca sobrecarga, pérdida de paquetes, retardo en las comunicaciones y otros aspectos que son cruciales sobre todo en servicios en tiempo real, creando de esta manera descontento en los usuarios por el uso de servicios intermitentes y de baja calidad (Irizar, C. & Calderón, C., 2017) frente a una deficiente administración de red en la asignación de recursos y acciones preventivas. Por lo tanto, el volumen de datos cada vez mayor y el tráfico de red requiere métodos eficientes de ingeniería y gestión de tráfico para garantizar la disponibilidad, escalabilidad y confiabilidad de la red (Jerome et al. , 2018). En este sentido se han implementado algoritmos de balanceo de carga poco efectivos con base a criterios de algoritmos estáticos entre los que se pueden mencionar:

Round Robin, Least Connections y Random; los cuales se basan en parámetros definidos independientemente del estado actual de la red.

El algoritmo Round Robin consiste en distribuir el tráfico rotativamente entre los enlaces, mientas que Least Connections distribuye según el número de conexiones que existan en el momento; por otra parte, Random distribuye el tráfico aleatoriamente sin importar el estado de los mismos (Kuster, 2015) (Betegón, 2018). Todos estos algoritmos fueron tomados en su momento como una alternativa para solucionar los problemas de congestión en la red tradicional de forma local (LAN) y en una red SDN para solventar problemas en la sobrecarga en enlaces, pero su mecanismo sigue siendo poco práctico para mejorar el rendimiento en la red. Frente a esta problemática y dificultades aún presentes es necesario desarrollar un balanceador de carga dinámico en base a algoritmos de la ruta más corta, que permitan obtener un estado de la red y redistribuir los tráficos de forma mucho más eficiente; estimando mejoras en el rendimiento de la red en base a métricas como la latencia, dispersión de retardo (jitter), tasas de transferencia, utilización del ancho de banda entre otras, de tal manera que permitan evaluar el rendimiento en la red de datos y verificar que el algoritmo desarrollado presentan mejoras ante otros algoritmos existentes como Round Robin, Last Connections y Random.

Con relación al problema planteado, se propone el levantamiento de una red de prueba SDN (testbed) mediante un controlador de código abierto y un simulador de red para la creación de escenarios de prueba. Un primer escenario consistirá en una red pequeña, con un número limitado de nodos con la finalidad de comprobar el correcto funcionamiento del algoritmo de balanceo de carga dinámico desarrollado; y un segundo escenario, una red extendida con nodos en cascada y la implementación de servidores sensibles al retardo como son VoIP y Streaming de Video con el objetivo de evaluar el rendimiento de la red ante un escenario crítico con demanda de tráfico excesivos. Posteriormente se realizará la implementación de los algoritmos existentes en el balanceo de carga estáticos y el propuesto en la siguiente investigación en el testbed implementando el modelo de análisis de rendimiento UF1880 que consta de las siguientes fases: La planificación, donde se detallará el propósito del análisis una vez realizado el monitoreo de la red; La definición de indicadores o métricas para valorar el estado de la red, los indicadores importantes para el rendimiento de la red considerados son: latencia, dispersión de retardo (jitter), tasas de transferencia, utilización del ancho de banda y pérdidas o errores de paquetes. La medición de estás métricas se lo hará mediante herramientas de análisis de rendimiento con la finalidad de determinar que el algoritmo desarrollado permita obtener mejoras en el rendimiento de la red de estudio.

Ante las altas demandas de tráfico en la actualidad y con la inmersión de nuevos dispositivos a la red de datos se han presentado nuevas tecnologías denominadas 5G que permiten solventar este requerimiento de demanda dando la posibilidad de integrar cada vez más equipos hacia internet, pero al igual que como una red tradicional las denominadas SDN poseen sus limitaciones si no son trabajadas. Al ser las SDN una red totalmente programable dan la oportunidad de realizar investigación que aporte a un mejor desempeño de la red entre aspectos como mejorar el rendimiento y administración de equipos de forma centralizada. El realizar el desarrollo de un algoritmo de balanceo de carga dinámico de múltiples rutas permitirá trabajar el enrutamiento y distribución de flujos de tráfico mejorando tiempos de respuesta y reducir la latencia en el transporte de flujos dentro de la red de transporte.

1.3. Objetivos

3.1.1. Objetivo General

Desarrollar un algoritmo de balanceo de carga dinámico basado en SDN que permita la redistribución de flujos en Redes Definidas por Software para mejorar el rendimiento en el enrutamiento de paquetes TCP/IP.

3.1.2. Objetivos Específicos

- Realizar una investigación bibliográfica relacionada al funcionamiento de Redes Definidas por Software (SDN) y algoritmos de balanceo de carga de múltiples rutas.
- Analizar herramientas de simulación que permitan despliegue de una red SDN basada en software libre aplicando la norma ISO/IEC/IEEE 29148.
- Determinar parámetros de medición en base al modelo de UF1880 Gestión de redes telemáticas, para evaluar el rendimiento de la red en base a estos parámetros establecidos.
- Desarrollar un algoritmo de balanceo de carga dinámico de múltiples rutas mediante lenguaje de programación en el controlador SDN para evaluar el enrutamiento de flujos de tráfico.
- Evaluar el funcionamiento del algoritmo de balanceo de carga SDN propuesto mediante herramientas de análisis de rendimiento de los parámetros de medición determinados.
- Establecer comparativas entre los parámetros de medición de rendimiento del algoritmo de balanceo de carga SDN desarrollado con relación a otros ya existentes en la equiparación de carga con la finalidad de determinar las mejoras presentadas la distribución y enrutamiento de paquetes TCP/IP.

1.4. Alcance

Las redes se han convertido en un recurso importante para el desarrollo de actividades de las personas, por lo tanto, brindar servicios de calidad es un objetivo para alcanzar por los administradores de red aplicando técnicas que permitan solventar problemáticas como congestión y retardo en las comunicaciones con la finalidad de mejorar la experiencia del usuario al utilizar servicios en red. Por tal motivo se propone el desarrollo de un algoritmo de balanceo de carga dinámico en el controlador de la Red Definida por Software permitiendo mejorar el rendimiento y evitar la congestión en las redes SDN. Los resultados esperados al implementar dicho algoritmo desarrollado son menor latencia, baja utilización de ancho de banda de los enlaces y menor número de paquetes errados en la recepción y compararlos con otros algoritmos poco eficiente en su aplicación para la misma finalidad (la distribuir el tráfico) conocidos como algoritmos de balanceo estáticos.

En primera instancia se realizará un estudio bibliográfico con el fin de establecer bases teóricas en las que se sustente el proyecto a desarrollarse en cuanto se refiere al estudio de la tecnología SDN, así como la comunicación de los diferentes elementos mediante el protocolo OpenFlow y el estudio teórico de investigaciones sobre algoritmos para el balanceo de carga como: Round Robin, Least Connection, Random; y el equilibradores de carga dinámicos presentados en fuentes bibliográficas de confiabilidad como IEEExplore, Scopus, entre otros.

Para el desarrollo y aplicación de los algoritmos estudiados se procederá a la selección de software entre los diferentes controladores y virtualizadores de red existentes en el mercado como Ryu, ONOS, Floodlight, Opendaylight y GNS3, Mininet, Omnet, NS2 entre otros respectivamente, con base a la norma ISO/IEC/IEEE 29148. El estándar contiene disposiciones para los procesos y productos relaciones con la ingeniería, así como los requisitos para los sistemas, productos de software y servicios. Para elección del software se determinará el controlador de acuerdo a parámetros como lenguaje de desarrollo, versión de protocolo de comunicación Openflow soportado, interfaz gráfica, fuentes de información para el manejo de la herramienta, virtualización (Mininet, GNS3, Omnet) y plataformas soportadas, entre otros parámetros necesarios presentados conforme al desarrollo de este proyecto con la finalidad de levantar el controlador SDN y la recreación de escenarios con elementos de red a trabajar como son los Switches OpenFlow Virtuales, Servidores VoIP y Streaming de Video, y hosts que permita el desarrollo y evaluación del proyecto.

Por otra parte, de acuerdo con el modelo establecido para el análisis del rendimiento de redes en la UF1880 Gestión de redes telemáticas (García, 2014), una vez diagnosticado el problema y el propósito de dicho estudio de análisis se procederá a la determinación de métricas e indicadores que permiten valorar el estado de la red de estudio, para este proyecto se ha considerado adecuado los indicadores más importantes para el rendimiento de la red son: la latencia, dispersión de retardo (jitter), tasas de transferencia, utilización del ancho de banda y pérdidas o errores de paquetes en la recepción.

En la fase de desarrollo se hará uso de la metodología de desarrollo de algoritmos que consiste en el diseño del algoritmo, programación del algoritmo, compilación y ejecución del algoritmo; para finalmente llegar a la fase de pruebas de funcionamiento en la cual se verificará el balanceo de carga realizado por el algoritmo, así como depuraciones en caso de ser necesario. Cabe señalar que paralelamente en todas las fases llevará una documentación de lo realizado y adicionalmente se realizará la implementación de los algoritmos de balanceo de carga estáticos ya existentes para su ejecución para un posterior análisis comparativo entre los parámetros de medición determinados por el modelo UF1880 entre el algoritmo de balanceo estático y el desarrollo en el presente proyecto.

Finalmente se procederá a la integración de los servidores de VoIP y Streaming de Video en el escenario de estudio, donde se tomará como escenario generar tráfico dos clientes conectados en los extremos de la topología para que realicen peticiones a los servicios y mediante los generadores iPerf para tráfico (ICMP) y NTttcp con la finalidad de verificar que el enrutamiento de paquetes se lo realizará por diferentes rutas alternas permitiendo reducir la sobrecarga de enlaces debido a la intervención del algoritmo de balanceo de carga dinámico de múltiples rutas. Además, medir los parámetros estipulados en este proyecto mediante el uso de herramientas de rendimiento como iPerf, PRTG, entre otros, con el objetivo de determinar el nivel de rendimiento que se posee al implementar el algoritmo propuesto y compararlo con otros algoritmos estáticos existentes para su ejecución para poder concluir que el algoritmo desarrollado presenta mejoras en el rendimiento de la red.

1.5. Justificación

Cada vez es más la información que se maneja y sectores que se conectan a Internet; este último es uno de los componentes que a diario se hace uso tanto para consultas informativas, reproducciones multimedia o contenido de entretenimiento (juegos de video). Según un pronóstico realizado por las Naciones Unidas supone que en el 2019 la mitad de la población mundial tendrá acceso a las tecnologías a la información e internet (Tecno, 2018). Aquí surge una incertidumbre con el crecimiento de la red debido a que las redes tradicionales son poco eficientes en la administración y en el manejo de flujos de tráfico excesivos, en este punto es donde surge las nuevas tecnologías 5G, entre las cuales se encuentran NFV y SDN, que permitirá eliminar esta barrera, poner a prueba criterios técnicos y de ingeniería para solventar problemas latentes en las redes tradicionales.

Las redes SDN es una red altamente programable, da la posibilidad de estudiar nuevas técnicas y aplicarlas para determinar los modelos que son mucho más afectivos antes problemas presentadas en las redes como son: el congestionamiento o sobrecarga, caída de enlaces, retardo en las comunicaciones, el ahorro de energía entre otros aspectos redes (Recio, G. & Riocerezo, G., 2018). Por esta razón esta tecnología al igual que NFV son especialmente trabajadas y evaluadas para poder enriquecer conocimientos y establecer futuras investigaciones que aporten al desarrollo de redes inteligentes y automatizadas, y posteriormente a la inmersión de técnicas de machine learning. Este proyecto no es la excepción, el investigar técnicas como el balanceo de carga dinámicos permitirá mejorar el rendimiento de la red, así como la calidad en la prestación servicios; Además, que permite la inmersión en de tecnologías libres permitiendo reducir costos y equipamiento físico en comparación a una red típica, ya que en la actualidad, en el mercado existen equipos (hardware) de balanceo de carga, pero a un costo alto y que utilizan algoritmos de balanceo estáticos sin garantizar que sea compatible con los escenarios de red que se podría presentar de acuerdo con los flujos de información de la red.

Por otra parte, estudio realizado por (Mallik, A., & Hegde, S. , 2015) realiza la redistribución de tráfico ante congestiones repentinas que pueden ser causadas por picos de carga o fallas de enlaces. (Umme, Z. & Hanene B., 2017) propone un algoritmo de balanceo de carga basado en SDN para optimización de enlaces en Centros de Datos mediante el cálculo de las rutas más cortas y del costo de cada enlace; cuando se detecta una congestión, saturación o caída del enlace en el trayecto de menor costo hacia el destino, está ruta es sustituida por otro trayecto alternativo que tenga menor costo y flujo de tráfico menor. En la documentación de (Sayali, 2018) presenta un estudio de un balanceador de carga para encontrar el mejor camino en SDN hacia el destino mediante parámetros de sobrecarga de enlaces, recuento de saltos, pérdida de paquetes e implementación de controladores distribuidos en la red. (Jerome et al. , 2018) Utiliza el protocolo de capa de transporte Multi-Path TCP (MPTCP) junto con el alias de IP, que permite la formación de subflujos múltiples dependiendo de la disponibilidad de direcciones IP en ambos extremos y ayuda a desviar el tráfico en los subflujos a través de todas las rutas disponibles. Todas estas investigaciones presentadas hacen referencia a trabajar en técnicas en el plano de datos para mejorar el rendimiento en una red SDN haciendo énfasis al balanceo de carga.

Estos algoritmos fueron tomados en su momento como una alternativa para solucionar los problemas de congestión en la red tradicional de forma local (LAN) y en una red SDN para solventar problemas en la sobrecarga en enlaces; pero su mecanismo sigue presentando problemáticas en parámetros como disponibilidad de la red debido a que estos estudios presentados actúan una vez suscitadas las fallas o sobrecarga en enlaces, es decir, acción una vez presentado el problema en la red produciendo retardo en las comunicaciones y pérdida de paquetes; o son basados en parámetros poco efectivos como el parámetro de número de saltos que podría no ser aplicado a una topología en malla extendida. Además, todos estos estudios proponen la típica implementación del algoritmo de Dijkstra para el cálculo de rutas.

Frente a esta problemática y dificultades aún presentes es necesario desarrollar un balanceador de carga dinámico mediante la investigación de algoritmos de PathFinding para el cálculo de rutas óptimas en el enrutamiento de paquetes e implementar técnicas para obtener el estado actual de los enlaces de red (tiempo real) y redistribuir los tráficos entre todas las rutas alternas de forma mucho más eficiente; estimando mejoras en el rendimiento de la red en base a métricas como la latencia, dispersión de retardo (jitter), tasas de transferencia, utilización del ancho de banda entre otras, de tal manera que permitan evaluar el rendimiento en la red de datos y verificar que el algoritmo desarrollado presenta mejoras ante otros algoritmos existentes como por ejemplo Round Robin, Last Connections y Random. La contribución de este trabajo es un algoritmo de balanceo de carga de múltiples dinámico, que, a diferencia de otros métodos de balanceo, se puede aplicar a una red independientemente de estado, ya sea esta, antes o después de fallas en el plano de datos para garantizar disponibilidad en la red y además tener un reporte actualizado del estado del enlace (congestión). Esta investigación se encuentra ligada a la línea de conectividad e integración de sistemas de la Carrera de Ingeniería en Electrónica y Redes de Comunicación de la Universidad Técnica del Norte.

Capítulo 2. Fundamentación Teórica

1.1. Introducción

A principio de los años 70, la Agencia para Proyectos Avanzados del Proyecto de Defensa Norteamericano, por sus siglas DARPA, desarrolló un plan que dio inicio a la red de redes, más conocida como Internet, llamada por aquellos tiempos como ARPANET. Debido a distintos factores como fueron: La aparición de computadores personales, la mejora de las redes de telecomunicaciones, nuevos servicios multimedia, y la gran cantidad de información, en los años 90, Internet empezó a abrirse al mundo de manera pública (Serrano, 2015).

Desde inicios del año 2000 y hasta la actualidad, las redes siguen en crecimiento exponencial, son cada vez más grandes, y almacenan también mayor cantidad de datos; por lo que se requieren redes o servicios de redes más dinámicos y flexibles, y una gestión de red más eficiente para controlar, planificar y organizar los diferentes dispositivos para garantizar la operatividad de la misma. Siendo que las redes tradicionales, no permiten programabilidad, flexibilidad, o capacidad para probar nuevas soluciones o alternativas sin la interrupción de las mismas, se empiezan a profundizar y conceptualizar las Redes Definidas por Software (SDN) (Serrano, 2015).

1.2. Redes Definidas por Software

Las Redes Definidas por Software por sus siglas en ingles SDN. Es una de las tecnologías emergente mayormente aplicada en el área de la administración de una red de datos como un Data Center o Cloud Computing debido a que brinda: alta flexibilidad al controlar una red volviéndola totalmente programable; automatización en procesos de operación y manejo de una red empresarial. Además, una red SDN se caracteriza por centralizar el plano de control y las funciones de reenvío de paquetes en una red de transporte permitiendo el control dinámico, adaptable y reducir la utilización de recursos de TI que por consecuencia reduce los gastos generales de operación e infraestructura. (Cisco, 2018)

2.2.1. Ventajas de una Red SDN.

Entre los aspectos más sobresalientes que hace a una Red SDN sea el futuro en la administración y como la solución tecnológica en la implementación de una red de tráfico de datos según (Guerrero, 2017) y (Millán, 2014) son:

- **Disponibilidad**: Debido al desacoplamiento del plano de control de los equipos de hardware y el control centralizado de políticas de gestión de paquetes, hace que la disponibilidad al acceso a los servicios sea totalmente funcional en la mayoría de tiempo de operación, y en caso de fallos presentados en la red aplicar medidas correctivas debido a su programabilidad.
- **Seguridad:** La seguridad se potencializa en las redes SDN debido a que el controlador de la red aplica políticas de acceso para todos los equipos tanto conmutadores y enrutadores de la topología de red, permitiendo salvaguardar la integridad de la información digital para las empresas donde la información es sumamente crítica.
- **Escalabilidad:** Al ser una red flexible y de fácil configuración mediante programas de optimización, permite la posibilidad de la integración de nuevas aplicaciones y recursos de red rápidamente y sin complicaciones. A diferencia de las redes tradicionales que el aspecto de escalabilidad es mucho más complejo debido a la interoperabilidad entre diferentes equipos de fábrica.
- **Mantenimiento:** El manejo en el mantenimiento es más rápido y contundente al ser menos los equipos de red a ser monitoreados principalmente el controlador, al ser el mecanismo central del manejo de la red. Esto a su vez permite un ahorro en costos de mantenimiento para las empresas.

- **Automatización:** Al ser la administración centralizada en un punto de toda la red permite reajustar los recursos de la red de acuerdo con los cambios y necesidades que puedan presentarse en un determinado tiempo como cuellos de botella, sobrecarga de enlaces o puertos de atacados.
- **Virtualización:** Permite disponer de redes más flexibles ante la reconfiguración de equipos en mínimas cantidades de tiempo y evitar la configuración de cada uno de los equipos o nodos de una red como en redes tradicionales de datos.

2.2.2. Arquitectura de una red SDN

La red SDN se caracteriza por su desacoplamiento del plano de datos y el plano de control, donde este último controla la inteligencia de la red y se encuentra centralizada lógicamente o de acuerdo con criterios de administración. Además, establece estructuras de software que puedan ser interpretadas por otros sistemas mediante una Interfaz de Programación de Aplicaciones (API: Aplication Program Interface).

En las redes SDN, las API’s permiten la comunicación entre las diferentes capas de la arquitectura SDN para la asignación o ejecución de instrucciones que influyan en el comportamiento de la red de datos. Las API`s se denominan hacia el Norte y hacia el Sur por la dirección que toma desde el centro de la arquitectura que es el plano de control; la API hacia el Norte son instrucciones desde el controlador hacia la capa superior o de aplicación mientras que la API hacia el Sur, son instrucciones desde el controlador hacia la capa inferior o de datos de la arquitectura SDN. (NewNow, 2018)

La figura 1 presenta la estructura de la red SDN la cual es una arquitectura de alto nivel compuesta por 3 planos principales: Plano de Control, Plano de Datos y Plano de Aplicación. El **plano de control** que representa el centro de toda la red, según (Guerrero, 2017) es la sección donde se procesa la toma de decisiones de acuerdo con políticas de seguridad, enrutamiento de paquetes, acceso o denegación de tráfico en la red, estado actual de la red entre otros aspectos de libre manejo mediante lenguaje de programación como Java, Python o C++. El **plano de control** es el “cerebro” de la red que mediante las API hacia el Sur mantiene actualizada las tablas de reenvío dinámicamente de tráfico permitiendo una administración más flexible y ágil de la red.

El **plano de datos** es la capa inferior de la arquitectura y representa a los dispositivos de reenvío, físicos o virtuales (Switches OpenFlow y OpenvSwitches), que mediante el manejo de tablas de flujo anunciadas por el plano de control aplica las políticas para conmutar, descartar o modificar paquetes de flujos por sus puertos de salida.

Para el reenvío de flujos de tráfico el conmutador OpenFlow cumple la función de verificar que el flujo realice un Match[1] al menos en una de sus tablas de flujo anunciados por el plano de control, caso contrario, ante el desconocimiento de alguna política de ejecución, el plano de datos mediante el protocolo de comunicación OpenFlow consultará al plano de control de alguna política de tratamiento del paquete ya sea para enrutarlo o descartarlo inmediatamente por seguridad (Conde, 2018). En la **sección 2.3.1** se explica en detalle la interacción entre el plano de control y datos de la arquitectura SDN.

Por otra parte, según (Cevallos, 2018), el **plano de aplicación** es la capa más alta de esta arquitectura de red controlada por el plano de control y consiste en los servicios presentados a los usuarios de una red de datos, mismos que pueden estar alojados dentro de la red local de una empresa o servicios de computación en la nube bajo un esquema IaaS (Infraestructura como servicio) como OpenStack, vCloud, CloudStack entre otros. La comunicación del plano de control y el plano de aplicación se lo realiza mediante las API hacia el Norte.

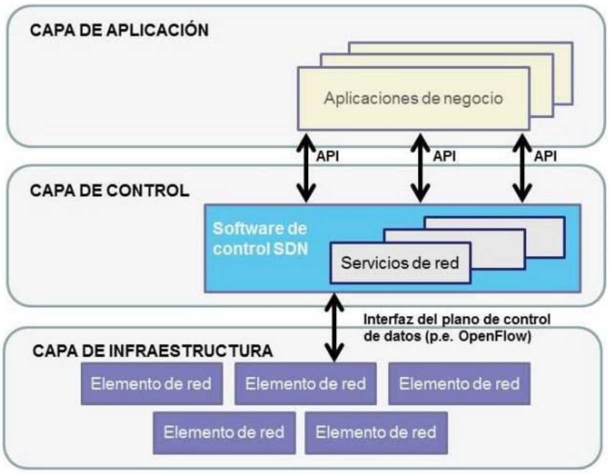


Figura 1 Arquitectura de las Redes Definidas por Software

Fuente: (Millán, 2014)

2.2.3. Elementos de una Red Definida por Software.

Al igual que una red tradicional que dispone de equipos dedicados para el enrutamiento, conmutación, seguridad, balanceo de carga o gestión de la red. Una red SDN dispone de una menor cantidad de equipamiento en hardware y software para el levantamiento de una red definida por software, ver Figura 2.

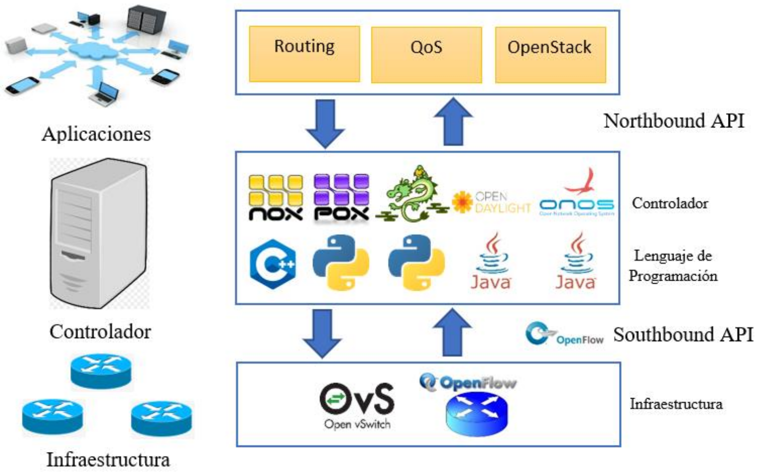


Figura 2 Elementos de la arquitectura SDN

Fuente: (Ramírez, M. & López, A., 2018)

A continuación, se presentan los elementos principales dentro de las SDN's, los cuales son:

2.2.3.1.Protocolo de Comunicación

El protocolo de comunicación establecido para la interacción entre el plano de control y plano de datos se denomina "OpenFlow", el cual está bajo la estandarización de la ONF (Open Networking Foundation), quien es la entidad a la vanguardia en promover el desarrollo de las redes SDN (Sandoval, 2018).

El protocolo OpenFlow según (Salinas, 2017), es el componente esencial para la comunicación de los equipos de la red con el controlador SDN, ya que es necesario disponer de un medio o canal para la interacción del plano de control y el plano de datos en una red SDN, debido a que en esta red se deslinda el plano de control de los equipos de conmutación y enrutamiento que usualmente en las redes tradicionales es integrado ambos planos en un mismo equipo o nodo dificultando la administración centralizada.

En las SDN, esté plano de control se centraliza y trabaja la red como un todo sin limitarse a una sección de la red en específico, permitiendo el control en la toma de decisiones como enrutamiento o conmutación de flujos en los equipos de hardware o software del plano de datos por parte del controlador SDN (Salinas, 2017).

Para mejorar y ofrecer mayores oportunidades en el manejo del plano de datos y tratamiento del tráfico de datos, OpenFlow ha venido evolucionado desde su aparición con la versión 1.0 en el año 2009 y nuevas versiones posteriormente con la salida en su versión 1.1, 1.2, 1.3 y 1.4. Sin embargo, las versiones más aplicadas en controladores SDN y Switches OpenFlow son la versión 1.1 y 1.3 debido a su funcionalidad y soporte en equipamiento actualmente en el mercado (Kuster, 2015). En el apartado **2.3.2** se estudia cada una de las versiones de este protocolo y cuáles son sus aportaciones a lo largo del tiempo.

2.2.3.2. Equipamiento de infraestructura

El equipamiento de red hace referencia a los componentes del plano de datos mismos que pueden ser de hardware o software como los conmutadores físicos OpenFlow o los conmutadores virtuales (vSwitch) mediante simuladores como Mininet, GNS3, Omnet entre otros; que permiten recrear escenarios de prueba e interconectarlos a un controlador SDN de software para controlar el tráfico de flujos y paquetes TCP/IP en base a los estudios que se deseen alcanzar (León, S. & Pino A., 2018). Estos equipos generalmente utilizan en su mecanismo de operación tablas de flujo establecidas por el controlador SDN para operar el tráfico de la red.

El conmutador virtual (OVS: Open Virtual Switch), cuenta con las mismas funciones que un switch convencional y soporta el protocolo de comunicación OpenFlow tanto en su versión 1.0 como 1.3. OVS no presenta ninguna complicación para ser utilizada en topologías de estudio simuladas para algún determinado fin; el hacer uso de este elemento se obtiene un ahorro al no adquirir equipos físicos de costos elevados o que no garantizan su interacción con los demás componentes de trabajo de una red real (Oyala, 2015).

2.2.3.3. Controlador

Otro de los componentes de la red SDN e indispensable ya que es el centro de procesamiento en la ejecución de políticas de red, es el controlador, el cual permite controlar los flujos de tráfico de los equipos de infraestructura en el plano de datos mediante el protocolo OpenFlow.

El controlador puede ser tanto un equipo físico dedicado al control de la red o como un programa de software virtualizado para cumplir las mismas funciones dentro de las redes SDN. Muchos de los software de controlador que se pueden encontrar, están basados en el kernel de Linux como por ejemplo ODL, ONOS, NOX/POX, RYU entre otros; y son mayormente utilizados en últimos estudios por la gran cantidad de información que proporcionan sus plataformas digitales para su manejo y levantamiento (Ampuño, A. & Chávez, M, 2015).

Un controlador, según (Conde, 2018), por lo general tiene una perspectiva completa de la topología de red que maneja. El usuario que lo administra puede destacar el estado de la red, puertos donde existe circulación de tráfico y poder generar tendencias o estadísticas para aplicar criterios de ingeniería de tráfico mediante líneas de código (programación) que se tratara a continuación.

2.2.3.4. Lenguaje de Programación

El lenguaje de programación es esencial para poder aplicar o modificar políticas de manejo de la red en el controlador. Este elemento viene implícitamente en el componente del controlador y es un lenguaje de alto nivel, el cual tiene un grado de complejidad manejarlo para aplicar instrucciones que el controlador ejecute en el entorno de trabajo de red (Cevallos, 2018).

El elemento que permite que se pueda manejar la red de forma dinámica mediante un lenguaje de programación es el protocolo OpenFlow o Southbound APIs (API hacia el sur), el cual convierte las líneas de código en instrucciones que los elementos de red pueden comprender y ejecutar en sus instancias.

Los lenguajes de programación que usualmente se encuentran depende del controlador que se implemente, algunos de los lenguajes más comunes son Java, Python o C++.

2.3. Protocolo de Comunicación OpenFlow

El protocolo OpenFlow como se ha venido detallando, es un elemento base para la creación y comunicación de los componentes de una red definida por software. Los desarrolladores de este protocolo desde sus inicios no se han conformado y han ido integrado mayores funciones en cada una de sus versiones con la finalidad de proporcionar en el campo de una red de transporte, mejores prestaciones en la calidad de servicio y administración de la red; volviéndose una red más automatizada a diferencia de las redes tradicionales que usualmente emplean los actuales proveedores de servicios y empresas con redes extendidas.

2.3.2. Funcionamiento de un conmutador y protocolo OpenFlow

En esta sección, se tratará la terminología y que función cumple cada uno de los elementos que se podrá encontrar en el siguiente esquema simplificado de un OpenFlow Switch y el protocolo de comunicación. En la Figura 3, se puede observar como el protocolo OpenFlow funciona como un medio de comunicación con el controlador y en especial lo hace mediante un canal seguro cifrado asimétrico basado en TLS para evitar el ingreso y comunicaciones de intrusos.

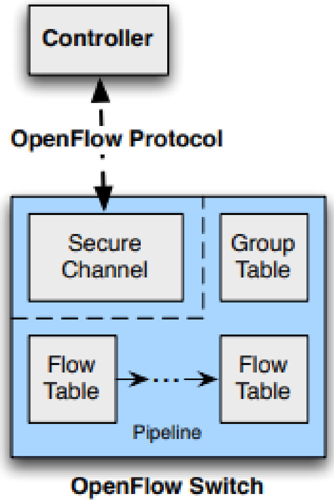


Figura 3 Comunicación conmutador y protocolo OpenFlow

Fuente: (Open Networking Foundation, 2011)

Como se puede apreciar en la Figura 3, el conmutador se compone de una tabla de flujo (**flow table**) que contiene un conjunto de entradas de flujos que permiten valorar los encabezados de los paquetes e identificar con cual tabla existe una coincidencia y un conjunto de una o más acciones para aplicar a los paquetes. Todo paquete que entra al equipo es comparado con las tablas de flujo, en el caso de que un paquete no tenga una coincidencia con ninguna tabla es reenviado al controlador mediante el protocolo OpenFlow por el canal seguro.

En este caso, el responsable de anunciar la adición o eliminación de las entradas de flujo para determinar el manejo de dicho paquete en el plano de datos es el controlador, tal como se muestra en la Figura 4 en la ruta C (Sánchez, A. & López, V., 2017). La entrada de un paquete por el puerto 2 ingresa a la función de emparejamiento con las tablas de flujo existentes en el dispositivo y ante la coincidencia pasa a la fase de acción donde el paquete se conmutará (ruta A) a otro puerto si existe una coincidencia; en caso contrario pasa su envío por el canal seguro al controlador (C) para determinar qué acciones ejecutar ante el paquete. Una vez que el controlador determina las acciones se las comunica al conmutador para modificar, eliminar o aumentar las tablas de flujo y decidir si conmutar el paquete a otro puerto o simplemente descartarlo (B).

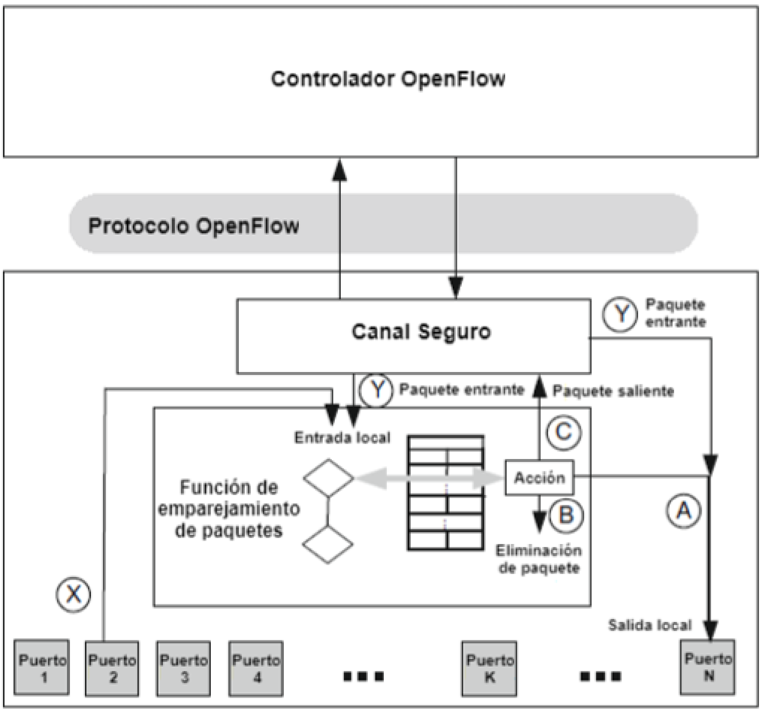


Figura 4 Mecanismo de comunicación entre Switch, Controlador y el Canal Seguro

Fuente: (Sánchez, A. & López, V., 2017)

La tabla de grupos o group table, como se muestra en la Figura 3, hacen referencia a un conjunto de acciones para reenvíos más complejos como inundaciones cuando se realiza la agregación de enlaces, direccionamientos rápidos o permitir el reenvío de múltiples flujos a un solo identificador. Las tablas de grupo contienen entradas al igual que las tablas de flujo, con la diferencia que hace relación a un grupo, donde cada grupo representa una variedad de acciones determinadas que se ejecutarán. Los paquetes que son enviados a los grupos pueden aplicarse uno o varios grupos de acciones dependiendo de la finalidad que se desee cumplir (Open Networking Foundation, 2011).

2.3.3. Evolución del protocolo OpenFlow.

A continuación, se presentan las versiones más destacadas del protocolo OpenFlow en el trabajo con las redes definidas por software y cuál es su aporte adicional con relación a su anterior versión.

2.3.3.1. OpenFlow v1.0

La función principal que cumple el protocolo OpenFlow en esta primera versión, por sus campos que maneja según (Open Networking Foundation, 2009), es el de conmutar paquetes de una interfaz a otra; adicionalmente permite la comunicación con el controlador para modificar, descartar o reenviar un paquete poniéndolo en cola y sólo trabaja con una tabla de flujo en su sistema para la comparación de los flujos entrantes.

En la primera versión, el protocolo consideró conveniente aplicar funciones tales como discriminar paquetes por dirección mediante la comparación del campo MAC, identificación de puertos de comunicación mediante el campo TCP-UDP y direccionamiento mediante el campo IPv4; estos campos presentados representan una parte del manejo de tabla de flujo del conmutador mediante el protocolo OpenFlow, esta tabla está compuesta por los campos Header Fields, Counters y Actions tal como se presentan en la Figura 5.

Header Fields	Counters	Actions

Figura 5 Componentes de tabla de flujo entrante

Fuente: (Open Networking Foundation, 2009)

El campo **Header Fields** permite realizar la comparación con los flujos entrantes mediante los campos IP origen-destino y máscara de subred y determinar qué proceso realizar con dicho paquete en el campo **Actions**; el campo **Counters** permite contabilizar el número de paquetes, el tamaño de cada flujo y el tiempo desde la última coincidencia del flujo con la finalidad de borrar flujos inactivos. Finalmente, el campo **Actions** determina la forma como debe ser procesado el paquete de acuerdo con su comparación de la cabecera; en el caso de que no exista una coincidencia el paquete es descartado y emite un mensaje de error.

2.3.3.2.*OpenFlow v1.1.*

En esta segunda versión (Open Networking Foundation, 2011), incorpora la capacidad de integrar el manejo de múltiples tablas de flujo con lo cual mejora el control del flujo de procesamiento del paquete por parte de los elementos de conmutación en el plano de datos. Además, se incrementa la función del manejo de etiquetas para brindar calidad de servicio con el multiprotocolo MPLS (ver Figura 9) y realizar acciones de insertar o eliminar una etiqueta de paquetes, permitiendo así reducir tiempos de retardo en el procesamiento de paquetes en las colas de una interfaz de salida del equipo de red.

Al incluir el manejo de múltiples tablas de flujo, aparece el mecanismo **Pipeline Processing,** el cual es un proceso que cumple el conmutador para encontrar una coincidencia con el paquete de entrada y aplicar las acciones o instrucciones pertinentes para cada tráfico.

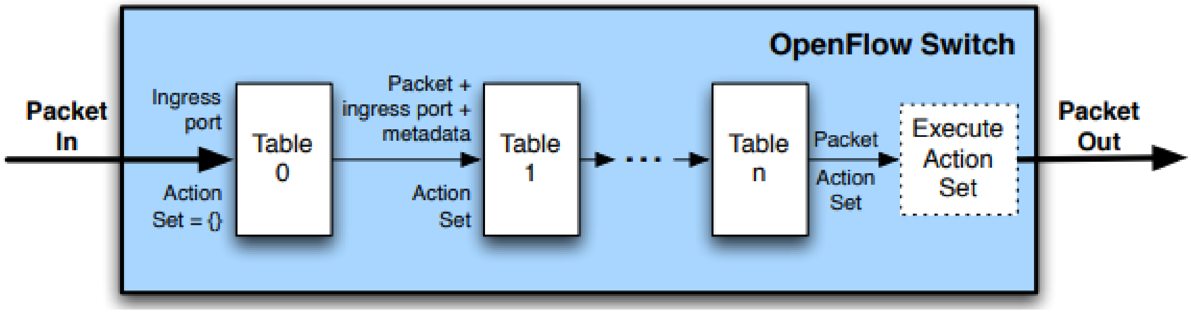


Figura 6 Proceso Pipeline para Match de paquetes

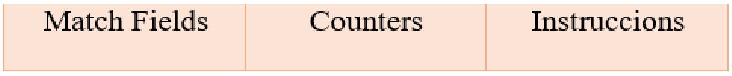
Fuente: (Open Networking Foundation, 2011)

La Figura 6 muestra un esquema simplificado de este proceso **Pipeline Processing**, en donde, **a**l ingresar un paquete al conmutador OpenFlow (OpenFlow Switch) se realiza la comparación con las tablas de flujo declaradas (Table 0, Table 1, Table n); al entrar el paquete, el proceso de comparación empieza con la primera tabla de la secuencia, en este caso “Table 0”; y en el caso de no encontrar una coincidencia en esta tabla pasa a la siguiente tabla según su orden hasta la tabla n, sin la posibilidad de retroceder. En el caso de existir una coincidencia, el flujo pasa al proceso de acción para procesarlo y aplicar las políticas requeridas; mientras que, si una vez cursado todas las tablas de flujo no encuentra una coincidencia, el paquete será enviado mediante el protocolo OpenFlow por un canal seguro al controlador para que este determine qué acciones aplicar a este tipo de tráfico en la red.

2.3.3.3.OpenFlow v1.2.

En el año 2011 con la llegada del protocolo IPv6 debido a la falta de direccionamiento IPv4 a nivel mundial, la fundación ONF vio necesario el integrar al campo **Match Fields** un campo de direccionamiento IPv6 para el manejo e interoperabilidad a futuro con el direccionamiento IPv6 en una red de datos. Cabe mencionar que en esta versión del protocolo OpenFlow, permite el reconocer controladores de backup distribuidos de forma estratégica para evitar caídas en la red, si el controlador principal, quedará inactivo por cuestiones de saturación o fallas en el sistema.

Los campos de las tablas de flujo manejados por la versión 1.1 y 1.2 son similares a la versión inicial con la diferencia que cambia su denominación por su funcionalidad especialmente el campo Header Fields y Actions por Match Fields e Instructions respectivamente, tal como se aprecia en la Figura 7.



Match Fields	Counters	Instruccions

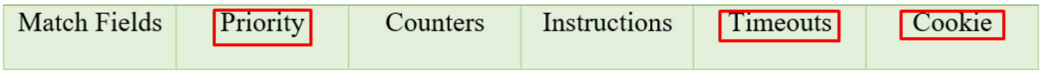
Figura 7 Campos de la tabla de flujo en su versión 1.1 y 1.2

Fuente: (Open Networking Foundation, 2011)

2.3.3.4. OpenFlow v1.3.

Está versión a diferencia de su antecesor versión 1.2, incrementa la posibilidad de manejar múltiples controladores desde diferentes puntos estratégicos para evitar el colapso por sobrecarga de procesamiento si se manejará una red compleja y extensa. Por otro lado, también presenta la posibilidad de limitar el flujo de paquetes para controlar las colas o entradas de flujo del plano de datos y evitar colapsos en las interfaces del equipo, esta función es conocida técnicamente como Meter Table (Open Networking Foundation, 2012).

Además, en esta versión se incrementa nuevos campos en la estructura de la tabla de flujo, los cuales son **Priority, Timeouts y Cookie,** tal como se presenta en la Figura 8.



Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Figura 8 Campos de la tabla de flujo versión 1.3

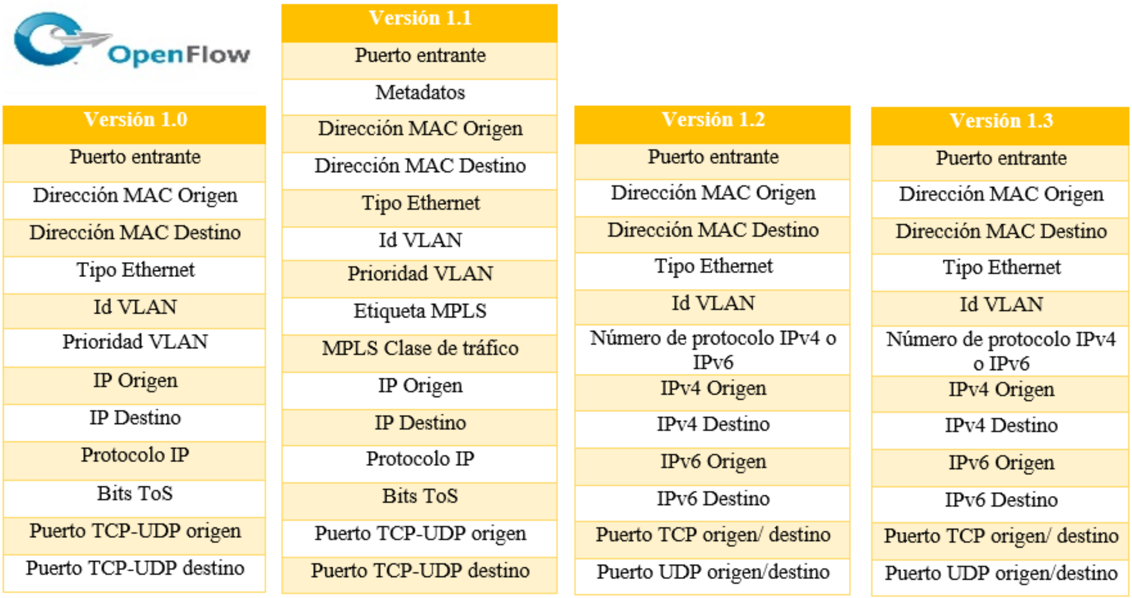
Fuente: (Open Networking Foundation, 2012)

La funcionalidad de los campos de la tabla de flujo de la versión 1.3 se detalla a continuación:

- **Match Fields:** Permite realizar un concepto de comparación con los campos de un paquete de entrada ya sea de dirección IP, puertos o cualquier campo de la cabecera.
- **Priority:** La prioridad es necesaria cuando un mismo paquete de datos es capturado por diferentes entradas de flujo provocando inconsistencias de procesamiento debido a que capturan los mismos paquetes y apliquen diferentes acciones para solucionar este conflicto se analiza este campo de Priority y se aplica las acciones al paquete de la entrada con mayor prioridad.
- **Counters:** Este campo lleva un registro de las acciones de las tablas de flujo del conmutador para que el controlador pueda tener una estadística de los paquetes que coincidieron con dicha tabla.

- **Instructions:** Este campo lleva las instrucciones para ejecutar el conjunto de acciones o el proceso de pipeline.
- **Timeouts:** Corresponde al tiempo de vida de un flujo antes de ser descartado o eliminado del conmutador debido a su inactividad o también puede definirse como el tiempo que una tabla podrá estar publicada antes de ser eliminada sin importar si pasa o no tráfico a través de está.
- **Cookie:** Es un valor numérico especificado por el controlador para filtrar estadísticas de flujo, modificar o eliminar flujo no utilizados al momento de procesar paquetes.

En la Figura 9, se presenta un esquema resumido de los subcampos perteneciente al segmento **Match Fields** de un flujo, y como el protocolo OpenFlow integra subcampos adicionales en cada una de sus versiones para el manejo de la red de datos SDN.



Versión 1.0	Versión 1.1	Versión 1.2	Versión 1.3
Puerto entrante	Puerto entrante	Puerto entrante	Puerto entrante
Dirección MAC Origen	Metadatos	Dirección MAC Origen	Dirección MAC Origen
Dirección MAC Destino	Dirección MAC Origen	Dirección MAC Destino	Dirección MAC Destino
Tipo Ethernet	Dirección MAC Destino	Tipo Ethernet	Tipo Ethernet
Id VLAN	Tipo Ethernet	Id VLAN	Id VLAN
Prioridad VLAN	Id VLAN	Número de protocolo IPv4 o IPv6	Número de protocolo IPv4 o IPv6
IP Origen	Prioridad VLAN	IPv4 Origen	IPv4 Origen
IP Destino	Etiqueta MPLS	IPv4 Destino	IPv4 Destino
Protocolo IP	MPLS Clase de tráfico	IPv6 Origen	IPv6 Origen
Bits ToS	IP Origen	IPv6 Destino	IPv6 Destino
Puerto TCP-UDP origen	IP Destino	Puerto TCP origen/ destino	Puerto TCP origen/ destino
Puerto TCP-UDP destino	Protocolo IP	Puerto UDP origen/destino	Puerto UDP origen/destino
	Bits ToS		
	Puerto TCP-UDP origen		
	Puerto TCP-UDP destino		

Figura 9 Campos de la cabecera Match Fields versión 1.0 hasta la versión 1.3 de OpenFlow

Fuente: Elaboración propia

2.3.3.5.OpenFlow v1.4.

Esta versión se ha extendido a la implementación y soporte de puertos ópticos y términos como potencia y frecuencia de la señal entre otros, ya no solo mediante tecnología Ethernet, sino que permite un nuevo medio de transmisión para mejorar la calidad de servicio y aumentar la velocidad en la transmisión de datos. Sin embargo, es una versión no muy utilizada por equipos y software comerciales en los controladores y switches de hardware y software.

Por otra parte, debido a la poca implementación de esta versión, el presente trabajo no aborda a detalle la misma; además que la versión 1.3 tiene los campos necesarios para el cumplimiento y desarrollo de esta investigación respecto al balanceo de carga de múltiples rutas.

2.3.4. Características del protocolo de comunicación OpenFlow 1.3

En esta sección se presentan temas relacionados al protocolo OpenFlow en su versión 1.3, debido a que es necesario conocer en detalle cada uno de los aspectos que influyen en esta versión del protocolo, como por ejemplo: los puertos que utiliza y mensajes que administra, para posteriormente comprender y conocer que pasa en la red con los mensajes que puedan presentarse en la parte práctica del proyecto.

2.3.4.1. Puertos OpenFlow

Los puertos OpenFlow permiten la conexión lógica entre los conmutadores de la red, muchas de las veces el número de puertos que posee un equipo no corresponde a que todos están habilitados para manejar este protocolo de comunicación. De acuerdo con (Open Networking Foundation, 2012) y (Kuster, 2015) OpenFlow posee 3 tipos de puertos: Físicos, Lógicos y Reservados.

Los **puertos físicos** corresponden a la interfaz de un equipo de hardware. Muchas veces el conmutador OpenFlow puede virtualizarse en el hardware del conmutador, es decir que un puerto físico OpenFlow puede representar una parte virtual de la interfaz del equipo de hardware.

Por otra parte, los **puertos lógicos** son puertos que no corresponden a una interfaz física del equipo, sino que son abstracciones similares a una interfaz loopback en un equipo de red tradicional. Estos puertos lógicos pueden encapsular y asignarse a varios puertos físicos e interactuar con el procesamiento OpenFlow como puertos físicos.

Finalmente, los **puertos reservados** son puertos lógicos definidos por el conmutador OpenFlow. Este tipo de puertos especifican acciones para reenvío genéricos al controlador, hacer una inundación de puertos utilizando métodos que no son de OpenFlow como los conmutadores ethernet tradicionales.

A continuación, se detallan los puertos reservados que debe soportar un conmutador OpenFlow:

- **All** son todos los puertos que el conmutador puede utilizar para reenviar un paquete en específico. En esta función se puede enviar una copia del paquete a los demás puertos a excepción del puerto de entrada del paquete y puertos configurados.
- **Controller** es un puerto que hace referencia al canal seguro de comunicación con el controlador. Puede ser un puerto de entrada o salida, al momento de cumplir la función de enviar un paquete lo encapsula y utiliza el mecanismo del protocolo OpenFlow.
- **Table** es el inicio del Pipeline Progress, este puerto tiene la función principal de salida del paquete de las listas de acciones y envía el paquete a la primera tabla de flujo para que el paquete sea procesado.
- **In-Port** es el puerto de entrada de paquete, puede usar como puerto de salida los paquetes que envía por su puerto de entrada.

- **Any** es un puerto asignado por defecto y es utilizado cuando no se especifica para un mensaje ninguno puerto en específico. Este puerto no puede ser utilizado como puerto de entrada, y tampoco como puerto de salida.

2.3.4.2. Mensajes del controlador al conmutador

Para (Kuster, 2015) son mensajes enviados por el controlador hacia al conmutador en el plano de datos; mismos que pueden necesitar o no de una respuesta por parte del conmutador. Los mensajes enviados por el controlador al conmutador son:

- **Features:** Es un mensaje petición/respuesta (reply/request) de información que solicita el controlador al conmutador sobre las características que detallen las capacidades del conmutador y deben ser contestadas.
- **Configuration:** Son solicitudes enviadas por el controlador sobre parámetros de configuración del equipo de conmutación y esté solo responde a las solicitudes del controlador. Se subdividen en 2 tipos de mensajes de configuración los cuales son **SET** que hace referencia a modificar la configuración del equipo y **GET** para obtención de información sin ejecutar cambios.
- **Modify-State:** Este tipo de mensajes es enviado por el controlador para administrar el estado de los conmutadores y sus características principales es adherir, eliminar o modificar las entradas de flujo y grupo en las tablas de OpenFlow.
- **Read-State:** Es un mensaje enviado por el controlador al conmutador, solicitando la lectura sobre la configuración y capacidades de los puertos y colas que maneja el conmutador en el plano de datos.
- **Packet-out:** Permite enviar mensajes de cualquier tipo encapsulados en un mensaje OpenFlow. El envio del mensaje puede ser entero o fragmentado; cuando el mensaje es fragmentado los datos son almacenado en buffer del conmutador hasta ser transmitido en su totalidad.

- **Role:** Este tipo de mensaje de petición/respuesta (reply/request) es útil para especificar el rol del canal OpenFlow cuando existe múltiples controladores en la red SDN.
- **Group-Mod:** Mediante este mensaje el controlador puede modificar o enviar peticiones de modificación de las tablas de grupo que maneja el conmutador.
- **Flow-Mod:** Es uno de los mensajes principales emitidos por el controlador que le permite modificar el estado del conmutador.
- **Barrier:** Son mensajes de petición/respuesta (reply/request) para asegurar que las dependencias o instrucciones que el controlador a ordenado al conmutador se hayan cumplido y recibir notificaciones de operaciones completadas.
- **Multipart:** Los mensajes petición/respuesta (reply/request) de este tipo permite al controlador solicitar información al conmutador sobre flujos individuales que este se encuentre trabajando.
- **Queue_Get_config:** Son mensajes de petición/respuesta (reply/request) que permite al controlador consultar el estado de las colas asociadas a varios puertos en el conmutador.
- **GetAsync:** Conjunto de mensajes de petición/respuesta por parte del controlador para establecer los mensajes asíncronos que debe enviar como también consultar al conmutador qué mensajes de este tipo enviará.

2.3.4.3.Mensajes Asincrónicos

Según (Serrano, 2015) los mensajes asíncronos son mensajes que se establecen entre dos elementos de forma diferida, es decir, que no existe coincidencia temporal. En una red SDN los mensajes asíncronos son enviados al controlador por parte del conmutador sin la necesidad que el controlador los haya solicitado. Entre los mensajes que pueden ser enviados por el conmutador son para denotar la llegada de un paquete, cambio de estado o falla del mismo.

Los tipos de mensajes que se clasifican como asincrónicos en la red SDN son los siguientes:

- **Packet-in:** Este mensaje cumple la función de dar la autoridad el conmutador al controlador de cómo tratar el paquete en el plano de datos y este decidir qué hacer el. En este caso de que un paquete no coincida con las tablas de flujo, el conmutador puede tomar dos decisiones; la primera es de enviar en su totalidad el paquete al controlador o segunda solo enviar una porción del mismo con el encabezado del paquete y el ID del buffer donde se almacena el resto del paquete en espera hasta que el controlador de la orden de procesamiento por parte del conmutador. Este último proceso se da solo en equipos que poseen un almacenamiento amplio, por lo general en todos los conmutadores hacen referencia al primer proceso.
- **Flow-Removed:** Es un mensaje que notifica al controlador sobre la eliminación de una entrada de flujo debido a su expiración del tiempo de espera en el conmutador.
- **Port-Status:** Permite conocer al controlador sobre el estado de un puerto del conmutador que pueden ser habilitado, bloqueado o inactivo.
- **Error:** Da a conocer a través de este mensaje la presencia de errores en la configuración del conmutador al controlador.

2.3.4.4.Mensajes Simétricos

Los mensajes simétricos no necesitan de un proceso de solicitud por parte del controlador o el conmutador y son enviados en cualquier dirección. Entre estos mensajes simétricos se encuentran tres tipos, los cuales son:

- **Hello:** Mensajes de saludo que se intercambian entre el conmutador y el controlador al iniciar conexión entre ambos elementos.
- **Echo:** Los mensajes reply/request de echo para asegurar la conexión entre el controlador y el conmutador; puede ser utilizado también para medir la latencia y ancho de banda de la conexión.

- **Experimenter:** Este tipo de mensaje ofrece funciones adicionales dentro del mensaje OpenFlow. Es considerado un área para futuras funciones de versiones OpenFlow.

En la Tabla 1, se presenta un resumen de los mensajes expuestos en la nomenclatura y su clasificación en cada una de las categorías según (Open Networking Foundation, 2012).

Tabla 1 Mensajes del protocolo OpenFlow v1.3

Tipo de Mensaje	Categoría	Subcategoría
HELLO	Mensaje Simétrico	N/A
ERROR	Mensaje Simétrico	N/A
ECHO_REQUEST	Mensaje Simétrico	N/A
ECHO_REPLY	Mensaje Simétrico	N/A
EXPERIMENTER	Mensaje Simétrico	N/A
FEATURES_REQUEST	Controlador-Conmutador	Configuración Switch
FEATURES_REPLY	Controlador-Conmutador	Configuración Switch
GET_CONFIG_REQUEST	Controlador-Conmutador	Configuración Switch
GET_CONFIG_REPLY	Controlador-Conmutador	Configuración Switch
SET_CONFIG	Controlador-Conmutador	Configuración Switch
PACKET_IN	Mensaje Asincrónico	Mensaje Asincrónico
FLOW_REMOVED	Mensaje Asincrónico	Mensaje Asincrónico
PORT_STATUS	Mensaje Asincrónico	Mensaje Asincrónico
PACKET_OUT	Controlador-Conmutador	Comandos desde el controlador
FLOW_MOD	Controlador-Conmutador	Comandos desde el controlador
GROUP_MOD	Controlador-Conmutador	Comandos desde el controlador
PORT_MOD	Controlador-Conmutador	Comandos desde el controlador
TABLE_MOD	Controlador-Conmutador	Comandos desde el controlador
MULTIPART_REQUEST	Controlador-Conmutador	Multipartes
MULTIPART_REPLY	Controlador-Conmutador	Multipartes
BARRIER_REQUEST	Controlador-Conmutador	Barrera
BARRIER_REPLY	Controlador-Conmutador	Barrera
QUEUE_GET_CONFIG_REQUEST	Controlador-Conmutador	Configuración de colas
QUEUE_GET_CONFIG_REPLY	Controlador-Conmutador	Configuración de colas
ROLE_REQUEST	Controlador-Conmutador	Cambio de rol del controlador
ROLE_REPLY	Controlador-Conmutador	Cambio de rol del controlador
GET_ASYNC_REQUEST	Controlador-Conmutador	Mensaje Asincrónico
GET_ASYNC_REPLY	Controlador-Conmutador	Mensaje Asincrónico

Fuente: (Open Networking Foundation, 2012)

2.4. Controladores SDN

Como ya se ha mencionado en apartados anteriores, el controlador es el cerebro de toda la red SDN y es el elemento principal para que toda una red pueda entrar en función al aplicar políticas de tratamiento de paquetes y poderlos conmutar o enrutar de cada uno de los nodos de infraestructura.

Entre los controladores que podemos encontrar están de código libre y licenciados, que para el desarrollo de este tema de tesis se ha optado por herramientas de software libre debido a que brindan libertad de modificación del código fuente para agregar nuevas funciones y comportamientos que el desarrollador desee incorporar a la red de datos, que, a diferencia de los licenciados, no permiten la modificación del código fuente limitando al desarrollador a funciones específicas únicamente. Por otra parte, cabe mencionar que el software libre permite un ahorro económico, brinda flexibilidad y facilidad en el manejo de la herramienta al momento de desarrollo de proyectos.

A continuación, se presenta a detalle y las estructuras que maneja cada uno de los controlares SDN de software libre, ya que es distinta para cada controlador ya que cada empresa desarrolladora maneja una estructura diferente de funcionamiento.

2.4.1. Herramientas de Software Libre.

El concepto de software libre según (Stallman, 2005), hace referencia a la libertad que tienen los usuarios al utilizar, modificar, distribuir o mejorar una herramienta sin la necesidad de solicitar autorización a una entidad u persona. La libertad de usar software libre se considera 3 clases de libertad que son: libertad para ejecutar el programa independientemente el objetivo; libertad para estudiar el programa, poderlo modificar a voluntad según las necesidades del usuario y finalmente la libertad de modificar el programa y dar a conocer a los demás para un bien en común.

Con estos antecedentes, la sección siguiente presenta los controladores RYU, ODL, NOX/POX, ONOS y Floodlight que son controladores de software libre que se pueden ejecutar para el desarrollo del presente trabajo.

2.4.1.1. Controlador RYU.

RYU es un controlador presentado y desarrollado por la empresa japonesa NTT. Este controlador es basado en el lenguaje de programación Python. Según (Guerrero, 2017), la característica principal que distingue a RYU de las demás herramientas, es que permite la creación de aplicaciones para la administración y control del tráfico de una red mediante el lenguaje de programación Python y posteriormente el controlador RYU las ejecutará; esta finalidad se logra al disponer de una API bien definida permitiendo a los desarrolladores la creación de aplicaciones. Hay que tener en consideración que el desarrollo de aplicaciones no corresponde al plano de superior de la arquitectura SDN, sino más bien a la ejecución de aplicaciones internas, propias del controlador.

Como se muestra en la Figura 10 donde se hace referencia a la arquitectura del controlador, existe un campo llamado RYU Applications, que hace énfasis el desarrollo de aplicaciones, donde los desarrolladores pueden aplicar criterios de tratamiento del tráfico entre dos nodos para la creación de la aplicación y que el controlador envíe las instrucciones mediante el protocolo OpenFlow a los equipos del nivel de infraestructura y estos a su vez las ejecute al tráfico circulante. El protocolo de comunicación OpenFlow puede ser empleado desde su versión 1.0 hasta la más reciente 1.4, pero no es el único para la administración de los equipos conmutadores, sino que adicionalmente puede trabajar con Netconf, Ofconfig, OVSDB, XFLOW y 3rd Party.

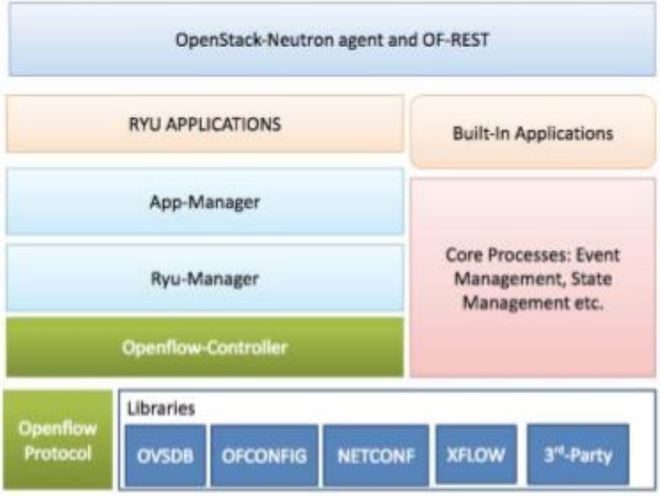


Figura 10 Arquitectura RYU

Fuente: (Sayans, 2018)

2.4.1.2. Controlador ODL (OpenDayLight).

ODL es uno de los controladores más comerciales actualmente para el despliegue de Redes Definidas por Software (SDN). El controlador ODL es de código libre basado en el kernel de Linux; maneja un lenguaje de programación en Java y su soporte es sobre Apache karaf, además utiliza la herramienta Maven para levantar el controlador y sus respectivas dependencias, ya que Maven permite la gestión y construcción de proyectos Java. (Sayans, 2018)

ODL tiene la capacidad de acoplarse a redes compuesta por diferentes fabricantes, es decir que puede integrarse a redes heterogéneas sin presentar complicaciones. Al disponer de una capa SAL (Abstracción de Servicio) brinda soporte multiprotocolo de comunicación con la capa inferior, además de que es considerado uno de los controladores con más apoyo financiero y con socios potenciales como Cisco, Ericsson y RedHat.

Entre otras características que se puede mencionar son: la funcionalidad de Clustering (redundancia); enrutamiento de paquetes basado en el algoritmo de la menor ruta (Dijkstra); posee una interfaz gráfica que permite al administrador el control y monitoreo de los equipos de red como conmutadores, estadísticas de enlaces entre otros. Cabe mencionar que este controlador soporta la versión de OpenFlow 1.0 hasta 1.3 para la comunicación con la cada de infraestructura de las redes SDN. (Ampuño, A. & Chávez, M, 2015)

Para una mejor abstracción de la operación del controlador se presenta en la Figura 11 la arquitectura que maneja OpenDayLight, en donde se puede observar que consta principalmente de tres niveles al igual que la arquitectura SDN; los mismos que son separados por interfaces que permiten la interacción entre las diferentes capas. A continuación, se presenta en detalle según (Álvarez, 2015), la función de cada uno de los elementos presentados en dicha arquitectura del controlador ODL, los cuales son:

- **Aplicaciones de red, orquestación y servicios:** Compuesta por aplicaciones de negocio como telefonía IP, web o aplicaciones para el control y administración del comportamiento de la red con la intervención del plano de control, es por ello que es necesario un interfaz para la comunicación constante entre ambos módulos.
- **Interfaz Northbound:** Brinda una variedad de servicios al controlador mediante el uso de API's para que las aplicaciones de la capa superior puedan explotar al máximo su capacidad en la gestión de la infraestructura de red.
- **Plataforma del controlador:** Proporciona múltiples servicios de red, entre los más comunes se encuentran: manejo de información de dispositivos; estadísticas de los nodos de la red; registro de los nodos disponibles y detalles de conexión; gestión en los flujos y políticas de la red e información de equipos finales tanto la dirección lógica como física entre otros servicios adicionales para mejorar las funcionalidades SDN.
- **Abstracción de servicios (SAL):** Representa una de las capas centrales en esta arquitectura del controlador, debido a que maneja la conexión entre las funciones de la red y los plugins del protocolo de la interfaz hacia el sur. Se caracteriza por cumplir un servicio independientemente del protocolo de comunicación que se utilice entre el controlador y los dispositivos de red.
- **Interfaz Southbound:** Hace referencia a los plugins utilizados con el fin de controlar la capa de infraestructura de la red. Estos plugins pueden variar entre OpenFlow, SNMP, NETCONF, OVSDB, entre otros.
- **Dispositivos de Red:** La última capa de esta arquitectura, que generalmente se encuentran los equipos o dispositivos de enrutamiento o conmutación de paquetes tanto físicos como virtuales, es decir, que se compone de toda la red a ser gestionada por el controlador OpenDayLight.

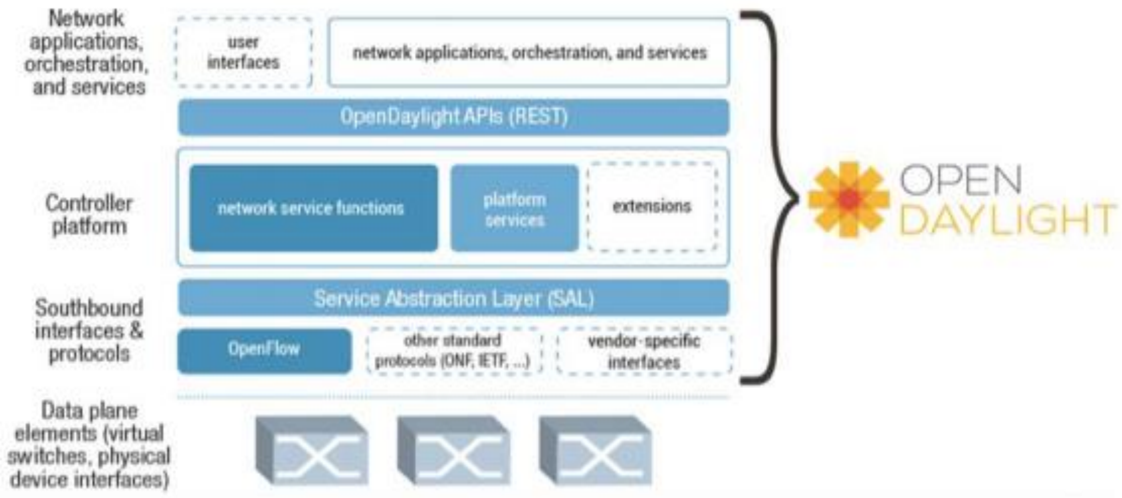


Figura 11 Arquitectura OpenDayLight

Fuente: (Kumbhare, 2018)

2.4.1.3. Controlador NOX/POX.

Es uno de los primeros controladores que tomo fuerza en el campo de las Redes Definidas por Software y especialmente es recodado por su desarrollo en lenguaje de programación C++. Este controlador quedó en la historia, debido a la empresa desarrolladora en ese entonces Nicira lanzó al mercado un nuevo controlador bajo la denominación de POX con mejores funcionalidades y un ambiente de desarrollo más amigable para el desarrollador; quedando el controlador NOX obsoleto e inactivo.

Según (Jeeva, 2016), al igual que el antecesor NOX, POX permite el desarrollo de aplicaciones mediante lenguaje de programación para el control de la red, con la diferencia que se basa en el código de desarrollo Python. POX al ser uno de los controladores más antiguos no dispone de una interfaz gráfica web para su administración; además cabe mencionar que apoya el trabajo con el protocolo OpenFlow en su versión 1.0 con la finalidad de crear entornos de trabajo para la experimentación e investigación sobre de redes SDN, en donde todos los elementos y actividades de la red se realizan como componentes individuales, pero sin descartar también la posibilidad de soportar este protocolo hasta su versión actual 1.4.

Para (Frómeta, D., et al , 2016), este controlador POX presenta una desventaja al ser uno de los controladores de primera generación para el desarrollo de redes SDN, debido a que no es sencilla la creación de aplicaciones sobre este controlador y porque se basan en un nivel de programación bajo la abstracción, es decir, que aumenta el nivel de complejidad al tener que el desarrollador enfrentarse a situaciones de comunicación entre el controlador y el conmutador que muy poco tiene que ver objetivo que es trabajar con aplicaciones para el tratamiento de los servicios o funciones que vaya a desempeñar la aplicación.

2.4.1.4. Controlador ONOS

ONOS es uno de los controladores más utilizados de código libre en el mercado, después de OpenDayLight. Este controlador para (Álvarez, 2015)**,** es usualmente implementado en redes de proveedores (WAN) y centro de datos, debido a la gran extensión de red a ser controlada o monitoreada brindando una menor tolerancia a fallos y permitiendo la distribución de la red hacia múltiples controladores.

ONOS presenta su integración en redes heterogenias sin importar si son equipos tradicionales o dispositivos OpenFlow. El controlador para su levantamiento se basa en el lenguaje de programación Java; es construido gracias a la plataforma Apache Karaf e instalación de dependencias por parte de Maven. Actualmente esta herramienta cuenta con el apoyo para su desarrollo e impulso por parte de empresas como AT&T, T-Mobile, Comcast, Samsung, Intel entre las más destacadas y Huawei para el despliegue de escenarios reales. (Open Network Operating System, 2018)

A diferencia de otros controladores, ONOS presenta algunas características importantes las cuales según (Conde, 2018) son:

- Posee una plataforma de documentación clasificada y actualizada que permite al desarrollador levantar la herramienta sin complicaciones, solo se necesita activar dos componentes para que el controlador pueda crear flujos dinámicos y poder la red tratar los paquetes.

- Presenta una interfaz gráfica superior a la de OpenDayLight en donde se visualiza la topología de la red y el estado actual de la misma.
- Posee una línea de comandos CLI que permite administrar las características de la red especialmente las políticas de control.
- Una API para la integración de aplicaciones externas por parte de la capa aplicación.

La arquitectura que maneja el controlador ONOS según (López, 2018), se divide en tres capas: Interfaz hacia el sur (Southbound), Núcleo (Core) e Interfaz hacia el Norte (Northbound), como se presenta en la Figura 12.

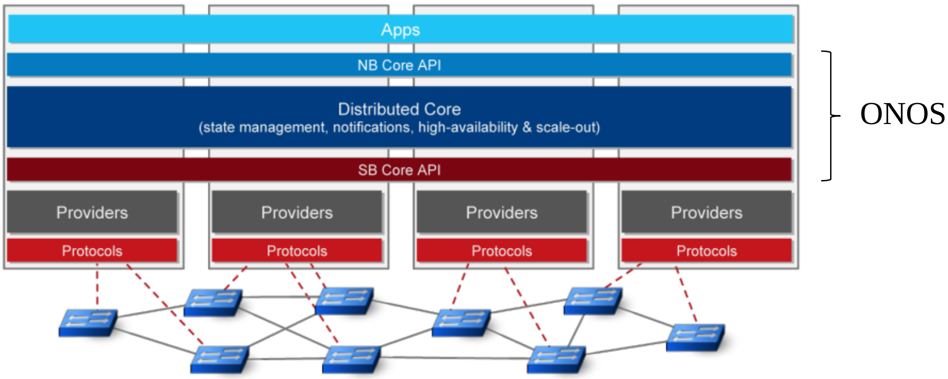


Figura 12 Arquitectura ONOS

Fuente: (López, 2018)

La **interfaz southbound (SB)** es el elemento encargado de la comunicación entre la capa de núcleo y los elementos de red de la capa inferior. Ofrece la interoperabilidad, integración o eliminación de equipos, aunque la red siga operando, un stack de protocolos de comunicación como, por ejemplo: OpenFlow, NET-CONF, SNMP entre otros; y además la integración de equipos virtuales sin la necesidad de disponer de equipos reales para una investigación.

Por otra parte, la capa de Core representa la parte central de la arquitectura permitiendo gestionar los nodos involucrados en la red como conmutadores, enrutadores y puntos de acceso. Permite el control de los equipos terminales como son los hosts de origen y destino de un tráfico determinado. Gestiona los enlaces entre nodos, equipos terminales, reglas de las tablas de flujo de los equipos y proporciona una visualización de la red general.

Finalmente**,** la interfaz northbound (NB) es la interfaz que permite la comunicación entre el núcleo y la capa aplicación o servicios. Es la característica esencial para el manejo de la red por parte de aplicaciones externas, es decir que las políticas son anunciadas por alguna aplicación en la capa superior y el controlador las ejecuta sin importar la lógica que esté presente.

A pesar de no ser el controlador que se encuentra a la vanguardia en el desarrollo de las redes SDN tal como lo es OpenDayLight, múltiples pruebas e investigaciones apuestan que ONOS ofrece mejores prestaciones en rendimiento y no se descarta que en un futuro esté controlador sea líder en el despliegue y desarrollo de Redes Definidas por Software (Sayans, 2018).

2.4.1.5.Controlador Floodlight.

Es uno de los controladores que aparece luego de su antecesor conocido como Beacon. Floodlight según (Kuster, 2015), es desarrollado mediante el lenguaje de programación Java y es considerado un controlador muy completo y con documentación actualizada para su levantamiento.

Este controlador es de código abierto y posee una comunidad de desarrolladores detrás que se denomina Big Switch Networks, que hacen de esta herramienta cada día más llamativa. Entre las características que se pueden destacar según (Sánchez, A. & López, V., 2017) se encuentran:

- Cuenta con módulos que hacen más fácil su manejo y ampliación.

- Fácil configuración de dependencias.
- Soporte de conmutadores virtuales y reales.
- Controla equipos tradicionales y equipos con OpenFlow.
- Permite la orquestación mediante la capa aplicación en la nube con OpenStack.

En la Figura 13, se presenta la arquitectura de operación del controlador Floodlight, el cual se basa especialmente en tres capas al igual que la arquitectura de una red SDN. Esta arquitectura consta de la capa Aplicación, Control y Datos que son separadas por interfaces de comunicación Northbound y Southbound. En la interfaz de southbound debido al stack de protocolos como OpenFlow admite la comunicación entre la capa del controlador y equipos de red.

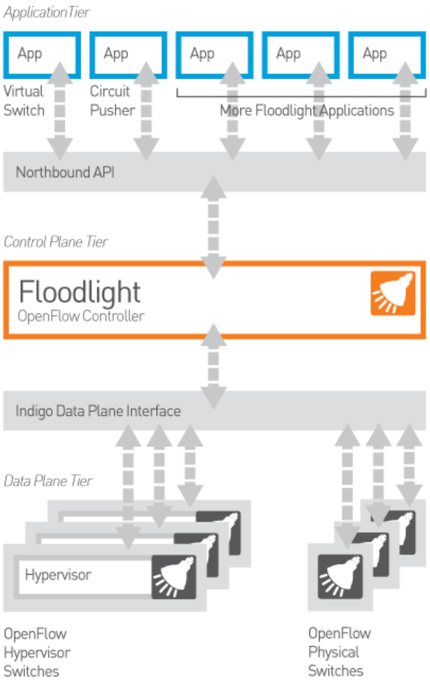


Figura 13 Arquitectura Floodlight

Fuente: (Project Floodlight, 2019)

2.5. Herramientas de simulación de topologías de red SDN

Para el desarrollo de una red SDN es esencial disponer de equipos virtuales que simulen la red de transporte (capa de infraestructura) es decir equipos conmutadores, enrutadores, equipos terminales entre otros. Para una investigación es conveniente simular una red para evitar gastos económicos innecesarios al elegir equipos físicos que probablemente no permitan cumplir con el objetivo de este trabajo. Además de que permite la recreación de escenarios de red en ambientes controlados con múltiples nodos similares a la de una red en producción para la obtención de resultados más realistas.

Entre los simuladores más utilizados para el despliegue de redes SDN son Mininet, GNS3, EstiNet, NS-3 y Omnet**++,** mismos que son explicados en destalle en las siguientes secciones.

2.5.1. Mininet

Según (Salinas, 2017) y (Guerrero, 2017), Mininet es una de las herramientas de software para la emulación de redes, permitiendo la creación de conmutadores, hosts, enrutadores y la posibilidad de crear controladores SDN, pero este último se descarta ya que este tipo de controlador que ofrece es muy simple y de baja capacidad.

Mininet es conocido por ser de código abierto sobre el kernel de Linux. Este emulador brinda un entorno de línea de comandos CLI para el manejo de los usuarios con la red virtualizada.

Entre otros aspectos importantes que se pueden tener con esta herramienta según (Serrano, 2015) son:

- La sencillez de manejo al solo ejecutar un comando se puede desplegar una red completamente rápida.
- Permite la personalización de escenarios de estudio mediante una interfaz gráfica o mediante lenguajes de programación con Python.
- Emulaciones de entornos reales donde los paquetes atraviesan interfaces reales ethernet, es decir que se puede obtener valores de velocidad del enlace y retardo.
- Correr o levantar programas reales como servidores y capturadores de tráfico.
- Conmutadores programables al implementar el protocolo OpenFlow 1.0 al 1.3.

- Es una de las herramientas en constante desarrollo por ende se encuentra fácilmente información sobre conflictos presentados en el emulador.
- Un host se comporta como una máquina real y controlarla mediante el ingreso por SSH.

2.5.2. GNS3.

Es una herramienta que permite emular, configurar y probar redes de todo tipo. En un inicio GNS3 solo soportaba el levantamiento de equipos de la marca Cisco para brindar una plataforma de entrenamiento para certificaciones de esta entidad, pero actualmente posee alrededor de 800 000 miembros que han trabajo y han expandido la gama de equipos que puede soportar, entre los cuales tenemos equipos Mikrotik, conmutadores Cumulis Linux, Docker, conmutadores virtuales y muchos más equipos basados en Linux.

La característica más llamativa es que posee la capacidad de simular controladores predeterminados en su conjunto de herramientas y especialmente trabajar mediante la comunicación entre conmutadores y controladores por OpenFlow en su versión 1.3 (GNS3, 2019).

Adicionalmente GNS3 presenta compatibilidad para su instalación en plataformas como Windows, MAC OS X, Linux entre otras. Pero, así como es considerado un emulador tan potencial presenta requerimientos de hardware muy elevados para poder soportar el programa y ejecutarse con normalidad, en el caso de un equipo con requerimientos básicos, el potencial de este emulador vería afectado con lentitud y colapso del equipo físico, debido a que esta herramienta ejecuta el sistema operativo real de los equipos de redes consumiendo gran cantidad de procesamiento (Telectrónika, 2018).

2.5.3. EstiNet

Es un emulador y simulador de redes con fines de estudio e investigación con su aparición comercial en el 2011. Posee una interfaz gráfica para construir y depurar simulaciones como también ver los resultados de la misma.

Ante investigaciones realizadas por (Calle, A., et al, 2018) asegura que EstiNet presenta mejores prestaciones en el desempeño y escalabilidad con relación a Mininet y NS-3. Además de ser una herramienta con posibilidades de simular elementos de redes inalámbricas y alámbricas, también permite la recreación de escenarios para levantar una red SDN mediante el protocolo de comunicación OpenFlow.

La única limitante que presenta este software es que solo puede levantarse en sistemas operativos de Linux para aprovechar la pila de protocolos TCP/IP que este posee y poder funcionar correctamente.

2.5.4. NS-3

Esta herramienta se enfoca en el trabajo de las capas 2 y 4 del modelo de referencia OSI. Soporta algunas formas de virtualización; especialmente este simulador es utilizado para generación de resultados y recolección de datos durante la ejecución de la simulación. NS-3 es completamente escrito en C++ facilitando la depuración, además se presenta su instalación en sistemas operativos Linux, OS X, FreeBSD, Solaris y Windows (Calle, A., et al, 2018).

NS-3 según (Wang, S. & Chao, C., 2017), no es considerado una de las mejores herramientas con potencial debido a su bajo estudio y por su poca documentación sobre la herramienta, lo cual imposibilita aprovechar al máximo su capacidad. Actualmente no soporta protocolos como OpenFlow v1.3; no permite integrar la red simulada con controladores reales, ni tampoco soporta el protocolo Spaning Tree para dar redundancia a las redes simuladas.

2.5.5. Omnet++

Es un simulador que empezó su aparición en el año 1997. Es una herramienta con menor esfuerzo que es aplicada para la recreación de redes móviles, inalámbricas, ATM y redes ópticas. A pesar de que presenta algunas aplicaciones, los módulos que dispone son simples y no completamente desarrollados, el cuál complica su uso y genera complicaciones a los desarrolladores que lo utilizan, debido a que deben modificar el código fuente o instalar complementos que son muy complicados de encontrar para cumplir con cierta finalidad. Este aspecto ha generado una línea de aprendizaje complicada que no es una tarea sencilla de realizar. (Calle, A., et al, 2018)

Adicionalmente, este emulador permite el soporte del protocolo de comunicación OpenFlow en sus especificaciones de la versión 1.3 (Cosmas, 2015), pero debido a su falta de complementos necesarios se ve afectada para ser una opción como una de las herramientas a ser utilizada en este trabajo de investigación.

2.6. Enrutamiento de paquetes TCP/IP

El enrutamiento hace referencia al encaminar paquetes de datos de un extremo a otro, es decir, desde un punto de origen hacía un punto destino tomando en cuenta criterios de selección de la mejor ruta hacia el destino mediante la identificación de la dirección IP del paquete. Este proceso cumple los equipos llamados routers en una red tradicional de datos, el cuál dicho equipo aplica técnicas para determinar por qué interfaz encaminar el paquete hacia el próximo nodo y así hasta llegar a su destino. (Oracle, 2010)

El enrutamiento de un paquete como se especifica es mediante la dirección IP, siendo este uno de los campos que posee todo paquete dentro de una red de datos. La dirección del paquete es dada por el protocolo IP que se encuentra en la capa 2 dentro del stack de protocolos de la arquitectura TCP/IP y capa 3 en el modelo de referencia OSI como se presenta en la Figura 14.

En la figura se presentan algunos de los campos necesarios para la comunicación entre los diferentes equipos de una red de datos. Mediante estos modelos se puede comprender de cómo está compuesto cada uno de los paquetes que circula por una red de transporte y especialmente para este caso en que capa se enfoca para el enrutamiento de paquetes.

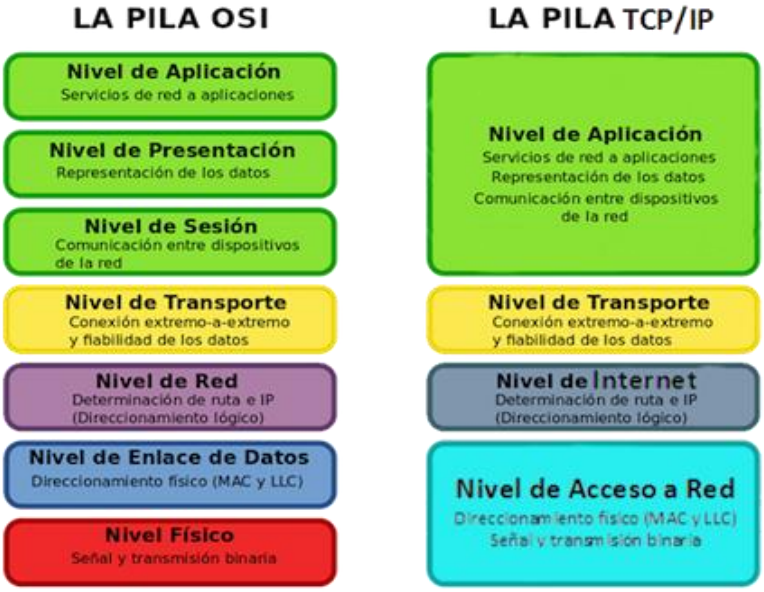


Figura 14 Modelo de referencia OSI y arquitectura TCP/IP

Fuente: (Master Móviles UA, 2019)

El enrutamiento de paquetes se integra de acuerdo a la tecnología que se maneje o a las necesidades de requerimiento de los servicios presentados en la red. Tanto en las redes tradiciones como las redes SDN, se poseen protocolos que permiten habilitar el enrutamiento de paquetes, en especial las redes tradicionales que poseen protocolos estandarizados a nivel mundial para integrar en las redes de datos de cualquier entidad u empresa. En la siguiente sección se presentan la variedad de protocolos de enrutamiento que hasta el día de hoy siguen vigentes en el campo de las telecomunicaciones.

2.6.1. Protocolos de enrutamiento en redes tradicionales.

Los protocolos de enrutamiento con los que se cuentas dentro de las redes tradicionales según (Rojas, 2015), son un número limitado que puede encontrar en los equipos de marcas reconocidas como Cisco, Mikrotik, HP, Huawei entre otros, lo cual limita a los administradores de redes a solo solventar sus problemas bajo las posibilidades que puede brindar un equipo.

En este caso los únicos protocolos que, por lo general, se utilizan en toda clase de red administrada por equipos propietarios son los siguientes:

- **RIP:** Para (Cisco, 2005), es entre los primeros protocolos de enrutamiento dinámico en implementarse y se basa en el vector distancia como criterio de enrutamiento. Utiliza un conteo de saltos como métrica para la selección de rutas, las rutas que excedan el número máximo 15 saltos son inalcanzables. Además de no admitir el reconocimiento de máscaras de red para la identificación de subredes.
- **OSPF:** Es el protocolo más utilizado en redes LAN. El protocolo se basa en el estado del enlace como criterio de enrutamiento. Utiliza el costo del enlace para la selección de rutas donde el costo sea mejor hacia un destino determinado es la ruta óptima para el reenvió de paquetes, por ello es llamado el protocolo de selección de la ruta más corta. Permite también el reconocimiento de subredes mediante la comparación de la máscara de red (Cisco, 2005).
- BGP: Es un protocolo que para se basa en el vector de ruta. Establece la comunicación entre dos routers autónomos mediante un previo establecimiento conexión entre ambos elementos para el intercambio de tablas de enrutamiento (Cisco, 2016). El objetivo principal es que determina la ruta más eficiente entre los dos nodos garantizando una correcta circulación de la información en una red de transporte (nube) evitando conflictos en el envío de datos.
- **IS-IS:** Este protocolo al igual que OSPF, se basa en la selección de rutas mediante el algoritmo de Dijkstra. Permite le manejo de redes extensas por ello es mayormente

implementado en ISP para disponer de una red más confiable y de rápida convergencia a diferencia de OSPF (NSRC, 2013).

- **EGRP:** El protocolo propietario de Cisco, basado en el enrutamiento por vector distancia, aunque también implementa la técnica de estado de enlace. Utiliza como métrica para selección de rutas el ancho de banda y la carga del enlace. Puede administrar una red de máximo 255 saltos entre routers (Cisco, 2005).

Muchos de estos protocolos ofrecen un cierto tipo de beneficios a los administradores de una red, pero estos mismos beneficios limitan al personal a solo disponer de únicas soluciones que se presentan en los equipos y más no poder modificar e interactuar constantemente con el equipo para resolver problemáticas que se pueden presentar en el instante.

Es por ello, que SDN cambie este esquema y presenta una herramienta totalmente programable y que se adapta a las necesidades de los administradores de red y para ello se debe conocer teóricamente el mecanismo de lección de rutas para poder aplicar correctamente en una red SDN y más aun con la aparición de nuevos algoritmos más efectivos que el tradicional Algoritmo de Dijkstra como son los algoritmos de PathFinding.

2.6.2. Algoritmos para el cálculo de rutas PathFinding.

El pathFinding según (Cuevas, 2013), forma parte del campo de la inteligencia artificial que permite la creación de procesos automatizados mediante la toma de decisiones por parte de los sistemas computacionales de una forma más rápida y eficiente. Esto conlleva igualmente a disponer de equipos con características de hardware altas en memoria y procesamiento para la ejecución del algoritmo en el entorno de desarrollo.

Los algoritmos de pathfinding se relacionan con este tema de trabajo al enfocarse especialmente en sistemas de navegación, con la finalidad de encontrar el mejor camino desde un punto de inicio hacia un punto de destino, teniendo en cuenta diversos criterios como la ruta más barata, más corta, más rápida y seleccionar el mejor camino (Geethu, 2015). Entre los algoritmos que se pueden encontrar hay mucha diversidad, pero en esta ocasión se presentan los más relacionados al área de las redes y su forma de operar para posteriormente aplicar uno de ellos en el desarrollo del algoritmo de balanceo de carga.

2.6.2.1. Algoritmo de A.*

Es uno de los algoritmos más simples y que mayormente es estudiado para el desarrollo de videojuegos al generar movimiento en personajes de videojuegos o robots de forma más reales (Cuevas, 2013). Es utilizado especialmente para el cálculo de rutas entre dos puntos en una plataforma cuadricular y seleccionar la más corta hacia el destino de una forma rápida para evitar la lentitud visual de los personajes en los juegos.

Este algoritmo se basa en el criterio del algoritmo de Dijkstra, pero de una forma más eficiente mediante el método heurístico dando soluciones óptimas a las que podría presentar el algoritmo de Dijkstra y con menor tiempo de procesamiento. La desventaja que presenta Dijkstra es que genera un proceso de visita a todos los nodos el cual puede generar en grafos (redes) más extensos procesos más lentos en la convergencia de la red (Chambi, 2013).

La técnica que utiliza este algoritmo para la selección de las rutas es mediante la siguiente función heurística que se expresa en la siguiente Ec 1.

$$F(N)= G(N)+H(N) \qquad \textit{(Ec. 1)}$$

Donde, G(N) representa al costo del enlace de un punto a otro y H(N) la distancia administrativa general entre el punto de partida y el punto de llegada en una red. Suponiendo que se tiene la red como se presenta en la Figura 15.

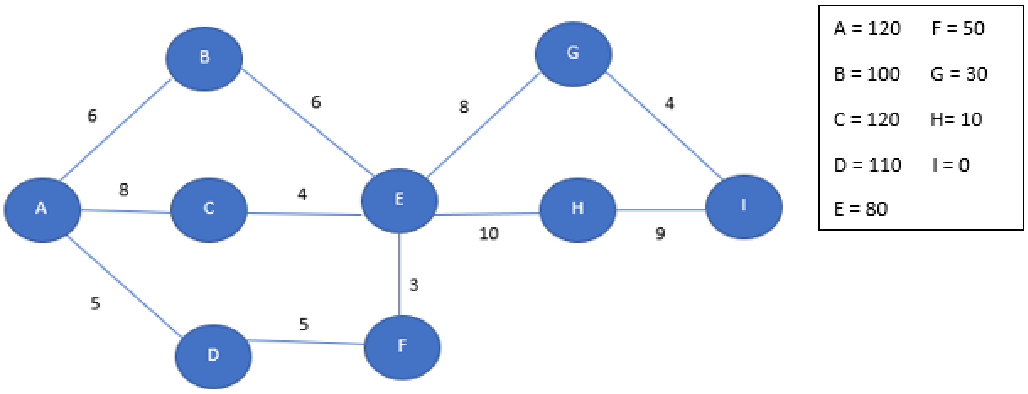


Figura 15 Grafo ejemplo para algoritmo A*

Fuente: Elaboración propia

Donde el recuadro representa el valor de la distancia administrativa H(N) considerado desde el nodo A hacia el nodo I, dicho valor es establecido bajo criterios de ingeniería y tráfico de datos. Los enlaces entre nodos son simbolizados por líneas azules entre los nodos y el valor de **costo**[1] que este enlace representa.

Entonces se aplica la función heurística Ec.1 del nodo A hacia al nodo I, obteniendo como resultado que el trazo A - B es el primer salto con un valor de 106 ($F(N)_1 = G(N) + H(N) = 100 + 6 = 106$), un valor menor que al trazo A - D o A - C. Para el nodo B existe un único enlace donde se toma dicho costo y la distancia administrativa de E - I = 80, dando como resultado un valor de $F(N)_2 = 86$. Posteriormente para el nodo E, el trazo más óptimo es hacia H con un valor de $F(N)_3 = 20$ y finalmente el salto de H - I para llegar a su destino con valor $F(N)_4 = 9$. Finalmente se suman estos resultados obtenidos para obtener el valor final desde A hacia I de $F(N)_T = F(N)_1 + F(N)_2 + F(N)_3 + F(N)_4 = 106 + 86 + 20 + 9 =$ **221**. Así es como este algoritmo trabaja para la detección de rutas óptimas.

A pesar de su gran efectividad en proceso de cálculo de rutas, el algoritmo A* también antes su aplicación en redes extensas y con niveles excesivos de tráfico puede presentar complicaciones por falta de procesamiento y memoria, volviendo al equipo más lento por el alto nivel que necesita en recursos para la ejecución del algoritmo, pero es de recalcar que es menor que la ejecución del algoritmo Dijkstra.

2.6.2.2. Algoritmo Breadth -First Search (BFS).

Es el algoritmo basado en la búsqueda en anchura de un grafo, es decir que este algoritmo realiza un descubrimiento de rutas mediante la visita de todos los nodos vecinos hasta completar el grafo con la condición de que, si un nodo ya ha sido visitado por otro trayecto, este no será toma en cuenta nuevamente (Garg, 2019).

En la siguiente Figura 16, se presenta una red con diversos nodos y que mediante el descubrimiento de nodos vecinos se crea un árbol jerárquico el cual determina el trayecto o ruta a tomar de un nodo inicial hacia el destino.

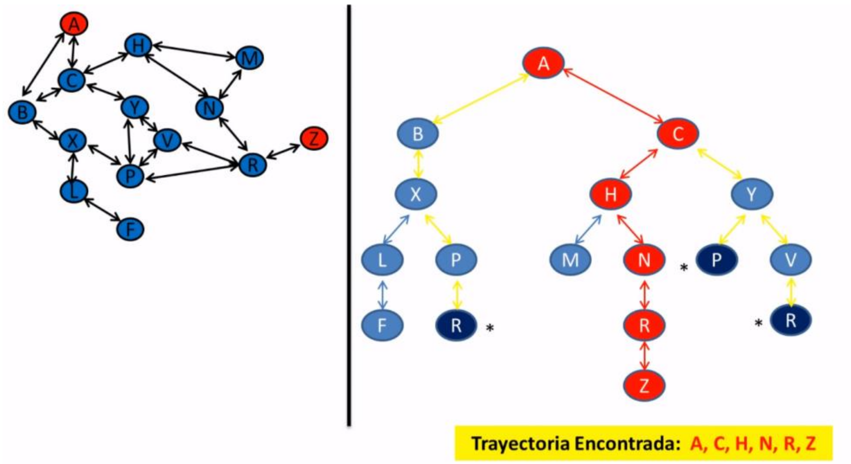


Figura 16 Descubrimiento de trayecto mediante BFS

Fuente: https://bit.ly/2JWtoF0

Como se presenta en la parte derecha de la Figura 16, las líneas rojas representan el trayecto desde el nodo A hacia el nodo Z pasando por C, H, N, R y Z. Las líneas amarillas presentan rutas alternas que podrían tomar llegar al nodo R y por ende hacia el nodo Z pero por cuestión de criterio del algoritmo no son tomadas como rutas principales pero que pueden ser consideradas como opción ante complicaciones.

2.6.2.3. Algoritmo Depth -First Search (DFS).

Es un algoritmo, que a diferencia al anterior el descubrimiento se realiza a fondo cuando se toma un nodo determinado hasta terminar su trayecto y retornar a un nodo anterior que disponga de más ramificaciones hacía vecinos por descubrir (Erickson, 2019).

En la Figura 17, se presenta un grafo donde se inicia el descubrimiento desde A hacia los nodos vecinos empezando por B (E, F) y descubriendo todas sus ramificaciones. Posteriormente pasar a C e igualmente estableces todos los vecinos presentes (G, I, J, H) y finalmente descubrir a D para finalizar el grafo completo. Es lo que distingue este algoritmo del presentado anteriormente, el DFS realiza el descubrimiento de forma vertical o profundidad y BFS lo realiza a lo ancho u horizontal.

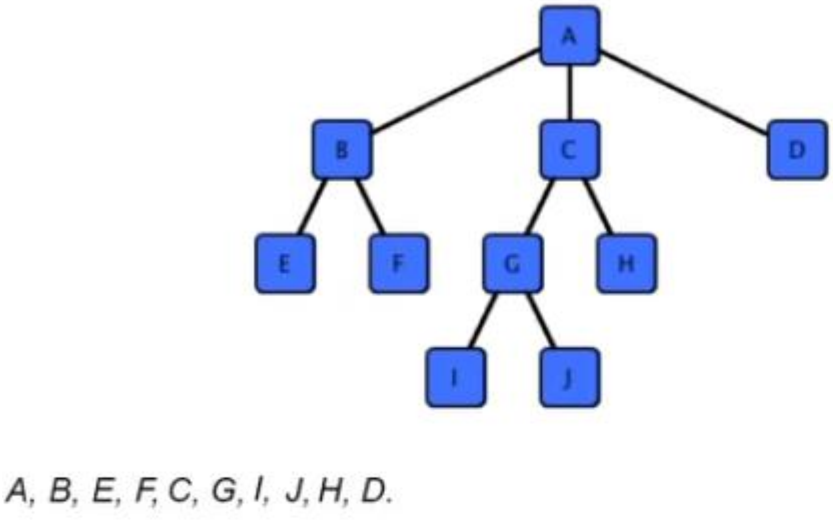


Figura 17 Descubrimiento de vecinos mediante DFS

Fuente: https://bit.ly/2WQByEd

Tanto para el algoritmo BFS y DFS presentan los trayectos hacia el destino, pero para dar valor a cada una de ellas es necesario de disponer de un método que permita obtener un valor cuantitativo que permita priorizar la ruta entra las demás. Es por lo cual se presenta la ecuación 2 para el cálculo del costo que se pueden presentar se estudia el protocolo de enrutamiento OSPF.

$$Costo\ del\ enlace = \frac{BW\ referencia}{BW} \qquad (Ec.\ 2)$$

Donde, BW referencia es el ancho de banda referencial que es 10^8 o 1Gbps y BW es el ancho de banda real del enlace que puede ser modificado por el administrador o mediante las interfaces físicas del equipo que intervienen en el enlace.

2.6.2.4.Algoritmo Digraph

Por otra parte, este tipo de algoritmo se base en la terminación de rutas mediante grafos. Esta en base al mecanismo utilizado por Dijkstra, DiGraph almacena nodos y bordes con datos y atributos.

Por definición, un grafo es una colección de nodos (vértices) junto con pares de nodos identificados (llamados aristas, enlaces, entre otros). En NetwokX, los nodos pueden ser cualquier objeto modificable, por ejemplo, una cadena de texto, una imagen, un objeto, otro grafo, un objeto de nodo personalizado, etc.

Dentro del algoritmo DiGraph, los bordes de una topología se representan como enlaces entre nodos con atributos “valor” permitiendo los bucles automáticos, pero no bordes múltiples.

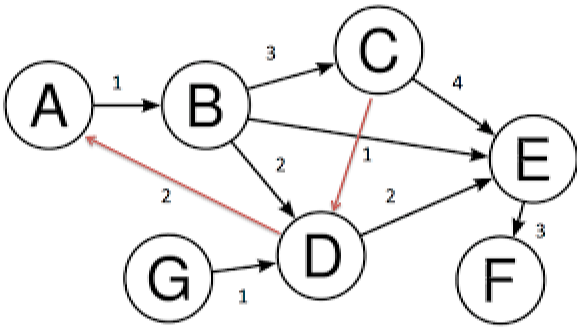


Figura. Descubrimiento de vecinos por Digraph

Fuente: (Stackoverflow, 2020)

En la Figura anterior, se puede observar como el algoritmo determina las rutas posibles mediante costos y saltos hacia los destinos. Por ejemplo del tramo desde A hacia E, es mediante los saltos B-C-E y B-D-E, teniendo como prioridad la ruta con menor costo de enlace es decir el enlace A-B-C-E**.**

2.7. Balanceadores de carga

Un balanceador de carga según (SoftSecurity, 2015), es proceso que lleva al equilibrar o distribuir la carga que puede estar receptando un único servidor o equipo de red por su interfaz hacia otros equipos para liberar la sobrecarga de procesamiento y así los tráficos puedan ser atendidos de manera inmediata y rápida evitando conflictos en la red y sobre todo evitar malestar en los usuarios de una red.

Los balanceadores de carga por lo general existen equipos comerciales especialmente para esta función y son implementados en la zona DMZ de la red para equilibrar los tráficos hacia los equipos que brinden algún tipo de servicio a la red por demanda y así disponer de una red con mayor rendimiento y disponibilidad.

A continuación, se presentan los balanceadores de carga existentes hasta el día de hoy en redes tradicionales como para las Redes Definidas por Software (SDN).

2.7.1. Balanceadores de carga en redes tradicionales.

Entre las mayores tareas de los balanceadores de carga en redes tradicionales está la de distribuir la carga de una zona en específico, especialmente en la zona desmilitarizada donde se encuentran los servidores debido a que el acceso a los servicios dentro de las redes es concurrente y en cierto punto excesivo. Es por ello que se hace uso de equipos de balanceo de carga bajo diferentes técnicas de funcionamiento para brindar mayor disponibilidad y rapidez de los servicios (Morató, 2016).

Como se presenta anteriormente se puede concluir que actualmente los balanceadores de carga son suministrados en un sector de las redes LAN sin tomar en cuenta que la saturación puede presentarse en las redes de transporte, pero es aplicado considerando que los equipos de las redes de transporte en redes tradicionales tienen una visualización y administración independiente de la red y no de forma general o centralizada, lo cual complica la realización de un balance tomando en cuenta una red completa. Es por esta razón que la solución tecnológica SDN cambia este mecanismo de trabajo permitiendo dar soluciones óptimas y efectivas.

En el siguiente apartado se presentan algunas de las técnicas u algoritmos como: Round Robin, Weighted Round Robin, Least Connections, Weighted Last Connections y Random; utilizados para equilibrar las cargas y que se basan generalmente en el tráfico saliente hacia una zona de servidores.

El primer algoritmo en abordar es el Round Robin, este algoritmo aplica la técnica de distribuir la carga saliente a una red de servidores de forma secuencial sin importar si alguno de los servidores se encuentra con gran cantidad de solicitudes de procesamiento o fuera de servicio por colapsos como se observa en la Figura 18. Por esto es recomendado su uso solo y sí los servidores son de las mismas características de hardware caso contrario existiría complicaciones en la red (IBM, 2017).

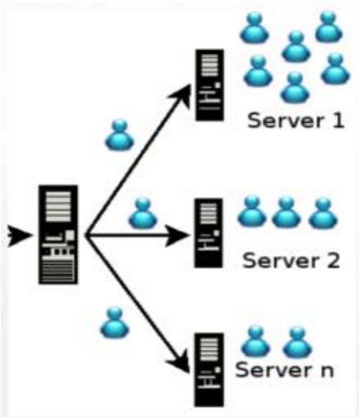


Figura 18 Técnica de distribución de carga Round Robin

Fuente: https://bit.ly/34A3U8k

Por otro lado Weighted Round Robin, misma que según (Acedo, 2016) es la técnica evolutiva de Round Robin según la cual consiste en distribuir la carga entrante a los servidores de forma ponderada, es decir que, si un servidor soporta mayor cantidad de solicitudes, el equipo balanceador enviará más peticiones por el enlace hacia este servidor y los de menos capacidad menor cantidad de solicitudes como se aprecia en la Figura 19.

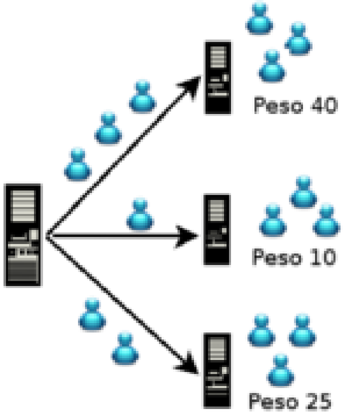


Figura 19 Técnica de distribución de carga Weighted Round Robin

Fuente: https://bit.ly/34A3U8k

Otro de los algoritmos presentes en el balanceo de carga es Least Connections; esta técnica consiste en distribuir la carga de acuerdo con el número de conexiones presentes en ese momento para cada servidor y ante una nueva solicitud será distribuida al servidor con menor número de conexión simultaneas como para el caso de la Figura 20, donde el Server 1 y Server n poseen menor conexiones simultaneas y el algoritmo Least Connections les asigna mayores solicitudes de servicio. Se relaciona con el algoritmo Round Robin en el sentido de que se aplica este algoritmo si los servidores poseen las mismas capacidades de soporte. (Kuster, 2015)

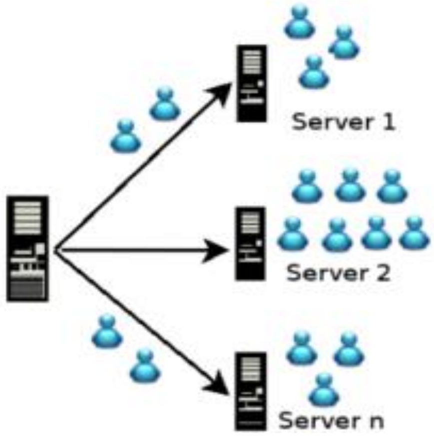


Figura 20 Técnica de distribución de carga Last Connections

Fuente: https://bit.ly/34A3U8k

Según (IBM, 2017) se tiene el siguiente el algoritmo **Weighted Least Connections** que es una combina la técnica de Last Connections y de Weighted Round Robin, en donde las peticiones se asignan a cada servidor dependiendo de las conexiones simultaneas en el instante y se pondera la cantidad de solicitudes a cada servidor de acuerdo a sus características de hardware para el rendimiento, ver Figura 21.

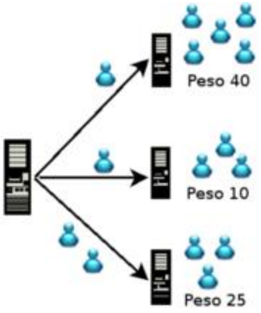


Figura 21 Técnica de distribución de carga Weighted Last Connections

Fuente: https://bit.ly/34A3U8k

Finalmente, el algoritmo Random el cual consiste en asignar un petición o solicitud al servidor de forma aleatoria tratando de hacer de una manera uniforme entre los servidores disponibles. Para hacer uso de este algoritmo es necesario que el conjunto de servidores sea de las mismas capacidades (Acedo, 2016).

2.7.2. Balanceadores de carga en redes SDN.

En las redes tradicionales, los enrutadores o nodos almacenan las tablas de enrutamiento, las tablas de enrutamiento solo contienen información de la red destino a alcanzar y el siguiente salto a realizar sin disponer de una vista de la red global. Por otro lado, la tecnología SDN tiene la capacidad de mostrar la vista global de la red y el controlador SDN puede descubrir todas las rutas entre cada nodo origen y destino. Al utilizar este mecanismo de trabajo de la red SDN, se puede evaluar la condición de cada ruta global, como se puede observar en la Figura 22.

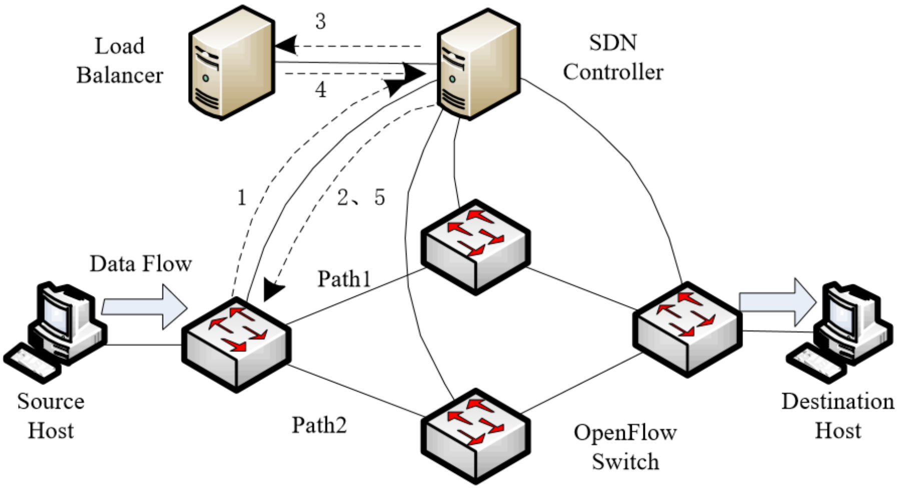


Figura 22 Arquitectura de la res SDN y el balanceador de carga

Fuente: (Ya-bin, X. & Chen-Xiao, C., 2016)

El servidor de balanceo de carga conjuntamente con el controlador SDN aplican los criterios de selección para identificar la ruta menos cargada y posteriormente el controlador asigna las tablas de flujo a los conmutadores OpenFlow para lograr la exitosa transmisión de datos evitando la sobrecarga de enlaces.

Los servidores de balanceo de carga en SDN se basan en algoritmos presentados en la sección 2.7.1 para equilibrar las cargas en la red. Pero aún existen complicaciones debido a que su mecanismo sigue siendo poco práctico para mejorar el rendimiento en la red. Frente a esta problemática y dificultades aún presentes es necesario desarrollar un balanceador de carga dinámico en base a algoritmos de la ruta más corta, que permitan obtener un estado de la red y redistribuir los tráficos de forma mucho más eficiente.

Entre algunos estudios sobre balanceadores de carga se tienen: el estudio realizado por (Mallik, A., & Hegde, S. , 2015) se presenta la redistribución de tráfico por trayectorias múltiples mediante el enrutamiento MP (Multipath) e integración de redes SDN, ante congestiones repentinas que pueden ser causadas por picos de carga o fallas de enlaces. En esta investigación no se garantiza la disponibilidad de la red en su totalidad, ya que este algoritmo desarrollado solo interviene cuando la red ya ha presentado complicaciones y lapsos de tiempo fuera de servicio.

En la Figura 23, el controlador cumple la función de la construcción de rutas mediante el MP, posteriormente monitorea en la red controlada para la detección de congestiones o fallas de enlaces para notificar al controlador y consecuentemente realizar la división del tráfico hacia otro enlace. Como se menciona anteriormente el algoritmo provoca tiempos fuera de los servicios en la red hacia ciertos destinos hasta que el controlador tome las decisiones de balancear la carga hacia otras rutas disponibles.

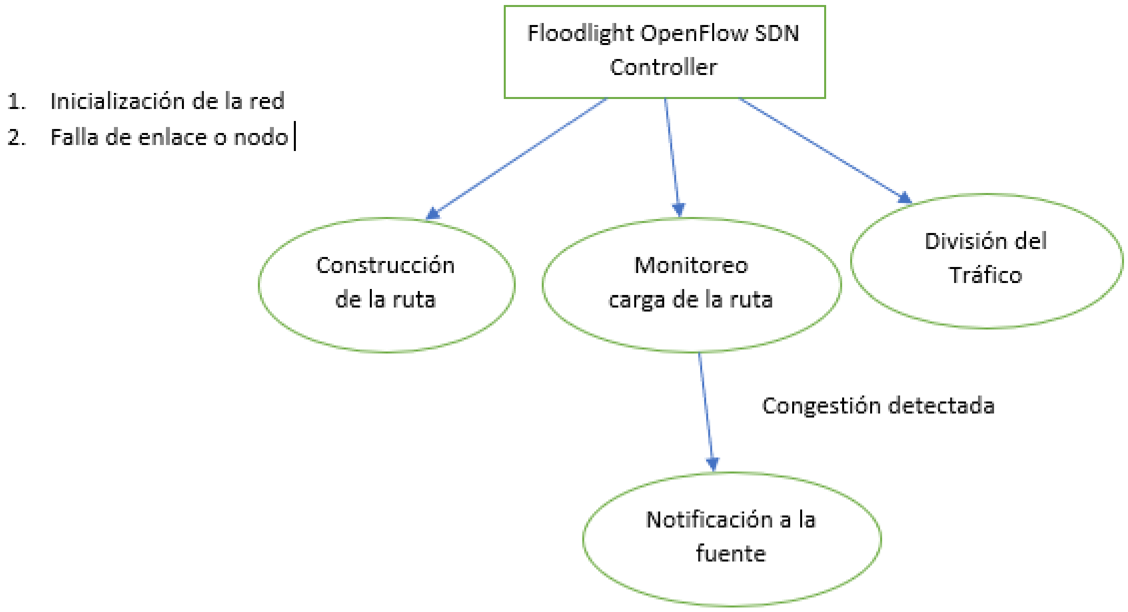


Figura 23 Diagrama de flujo del algoritmo de balanceo de carga por MP

Fuente: (Mallik, A., & Hegde, S. , 2015)

En la investigación de (Umme, Z. & Hanene B., 2017) se presenta un algoritmo de balanceo de carga basado en SDN para optimización de enlaces en Centros de Datos mediante el cálculo de las rutas más cortas y del costo de cada enlace; cuando se detecta una congestión, saturación o caída del enlace en el trayecto de menor costo hacia el destino, está ruta es sustituida por el siguiente trayecto calculado por el algoritmo de Dijkstra que tenga menor costo y flujo de tráfico menor. En la Figura 25 se presenta el mecanismo de trabajo de este algoritmo en la red SDN.

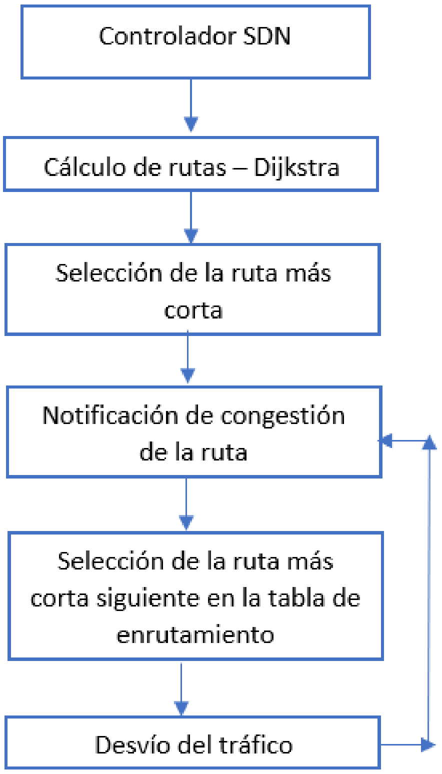


Figura 24 Diagrama de flujo del algoritmo de balanceo por Dijkstra

Fuente: (Umme, Z. & Hanene B., 2017)

En la documentación de (Sayali, 2018) presenta un estudio de un balanceador de carga para encontrar el mejor camino en SDN hacia el destino mediante parámetros de sobrecarga de enlaces, recuento de saltos, pérdida de paquetes e implementación de controladores distribuidos en la red. En este trabajo presentado el mecanismo de selección de rutas es poco eficiente ya que se basa en la detección de la ruta mediante el número de saltos, ver Figura 29, que en redes más extensas con múltiples nodos genera complicaciones ya que dispone de un límite de saltos, excedido este límite la ruta es considerada inalcanzable.

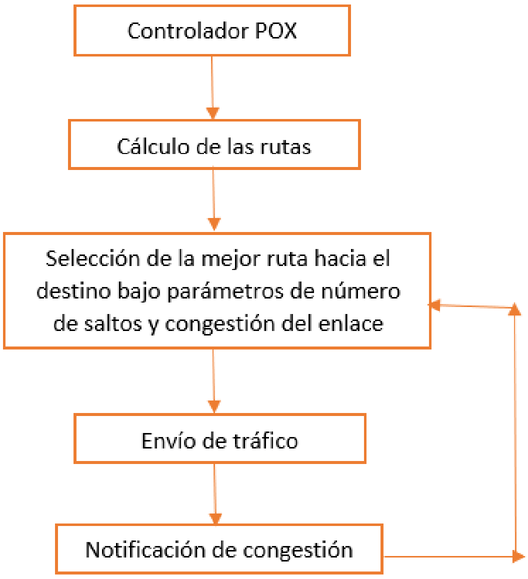


Figura 25 Diagrama de flujo para balanceo mediante número de saltos y congestión del enlace

Fuente: (Sayali, 2018)

Como última investigación presentada en el balanceo de carga en redes SDN es por (Jerome et al. , 2018) utiliza el protocolo de capa de transporte Multi-Path TCP (MPTCP) junto con IP, que permite la formación de subflujos múltiples dependiendo de la disponibilidad de direcciones IP en ambos extremos y ayuda a desviar el tráfico en los subflujos a través de todas las rutas disponibles.

Estos son algunos de los estudios realizados para balanceo de carga mediante la tecnología SDN, que, a diferencia del algoritmo a desarrollarse, es la contribución en un algoritmo de balanceo de carga de múltiples rutas dinámico que se puede aplicar a una red independientemente de estado, ya sea esta, antes o después de fallas en el plano de datos para garantizar disponibilidad en la red y aplicar técnicas diferentes como algoritmos de Pathfinding para el cálculo de rutas hacia los destinos.

2.8. Modelo de Rendimiento UF1880

Es de considerar que dentro de una investigación es importante la implementación de un modelo de referencia que permita obtener una perspectiva más clara y objetiva de lo se desea alcanzar o lograr con este proyecto de tesis. Es por ello que en el presente trabajo se presenta el modelo de rendimiento UF1880 debido a su relación con la línea de investigación, el cual es mejorar el rendimiento en el enrutamiento de paquetes TCP/IP de una red SDN.

Para el presente trabajo, el modelo UF1880 consta de 4 fases; comenzando por la **fase 0** que presenta la selección de las herramientas de software mediante la norma ISO/IEC/IEEE 29148, además del estudio y construcción de escenarios SDN de prueba o testbed.

En la **fase 1**, se procede al levantamiento de las herramientas de software seleccionadas, posteriormente la recreación de los escenarios de prueba mediante las herramientas de trabajo seleccionadas en la fase anterior y análisis del comportamiento de la red relacionado al enrutamiento de paquetes TCP/IP.

La fase 2, corresponde al análisis de las diferentes métricas o parámetros de evaluación del rendimiento que corresponda según el caso de estudio. Las métricas de rendimiento son muy diversas, entre las cuales se pueden mencionar:

- **Tasas de transferencia:** Conocido también como **throughput**, el cual consiste en la medición de la cantidad de paquetes o bits por unidad de tiempo [pps - bps], haciendo referencia también a la velocidad de transporte de datos en la red.
- **Ancho de banda utilizado:** El ancho de banda utilizado es medido en bits por segundo [bps] y representa la capacidad efectiva en relación con la capacidad nominal del canal, en otras palabras, la capacidad efectiva es parte utilizada del valor teórico total del canal.
- **Jitter:** Conocido también como dispersión de retardo, hace referencia a la variación en tiempos distintos de llegada de los paquetes y se debe a la interferencia en el canal o por congestión de este. La métrica de jitter es importante cuando se necesita tiempos fijos de tratamiento del tráfico especialmente en servicios que requieren fluidez.
- **Tiempo de respuesta:** Es el retardo dentro de la red a las solicitudes individuales. Se puede definir que es el intervalo entre el final de un envío de solicitud y el comienzo de la

respuesta correspondiente del sistema o como el intervalo entre el final de un envío de solicitud y el final del correspondiente. En el parámetro de tiempo de respuesta intervienen otras sub-parámetros necesarios para determinar el tiempo final, los cuales son: retardo de procesamiento, retardo de cola, retardo de transmisión y retardo de propagación, que en conjunto determinan el valor final de tiempo de respuesta a una solicitud.

- **Pérdida de Paquetes:** Como su nombre lo especifica es la cantidad de paquetes que necesitan ser reenviados por la red hacia los respectivos destinos debido a la saturación de enlaces o colas extensas en los equipos intermedios, que hacen que los equipos de direccionamiento descarten paquetes en el camino presentando demoras en la presentación de los servicios a los usuarios de una red. La pérdida de paquetes por lo general es medida por el número de paquetes extraviados en la red o también por su representación en porcentaje (%), este último es el número de paquetes perdidos con relación al total de paquetes enviados desde su origen hacia el destino por la cantidad de 100 como se muestra en la Ecuación 3.

$$\%\, paquetes\ perdidos = \frac{Número\ de\ paquetes\ perdidos}{Número\ total\ de\ paquetes\ enviados} \qquad (Ec.3)$$

La **fase 3** y la última presentada en el modelo UF1880, hace referencia a la identificación de herramientas para la evaluación de los parámetros o métricas, con la finalidad de obtener datos estadísticos sobre el rendimiento de la red SDN y efectuar mejoras mediante el desarrollo del algoritmo de balanceo de carga dinámico de múltiples rutas. Entre las herramientas que se pueden encontrar para el análisis de rendimiento son:

- **iPerf**. Es una de las herramientas más útiles para este proyecto debido a que se integra fácilmente con la red simulada por Mininet. Esta herramienta crea flujos TCP y UDP para la evaluación del rendimiento de la red mediante la valoración de paquetes perdidos, ancho de banda, tasas de transmisión entre los extremos. Además, dicha herramienta es de código abierto y soporta múltiples plataformas como: Windows, Linux, Unix entre otros. iPerf presenta un informe con las marcas de tiempo, la cantidad de datos transmitidos y el rendimiento medido en la red SDN, lo cual permite analizar resultados de rendimiento del antes y después de aplicar el algoritmo de balanceo de carga dinámico de múltiples rutas en las redes de estudio.

- **Ping**: Es uno de los comandos comúnmente utilizados para la conectividad entre extremos de una red. Los parámetros que evalúa son el tiempo de respuesta, el número de paquetes enviados y recibidos, así como la cantidad y el porcentaje de paquetes perdidos en la red. La integración del comando PING, se encuentra en casi todos los sistemas operativos tanto para MAC, Windows, Linux, Unix mediante su ejecución por comandos en el terminal de cada una de las plataformas mencionadas.
- **Wireshark:** Es un sniffer para la verificación de tráfico en la red. Permite la evaluación del rendimiento mediante gráficos estadísticos de paquetes trasmitidos, tiempos de transmisión, ancho de banda utilizado y la valoración del parámetro **jitter** para el servicio de telefonía IP, entre otros aspectos que pueden ser de suma importancia al momento de analizar y medir el rendimiento de las redes de datos.

Capítulo 3. Metodología para Medición del Rendimiento en la Red

En este capítulo se desarrolla las fases de modelo UF1880: Gestión de Redes Telemáticas para la detección de complicaciones del transporte de paquetes TCP/IP en las Redes Definidas por Software.

En la fase 1, se procede al levantamiento del controlador SDN y la emulación de los nodos de la red de transporte para el monitoreo de esta y determinar ante tráficos excesivos como la red se comporta para evitar que la red quede sin conectividad hacia determinadas redes de destino.

En la fase 2, se presentará los parámetros o métricas a ser evaluadas en la red para la observación de las mejoras ante la aplicación del algoritmo de balanceo de carga de múltiples rutas y cumplir con los objetivos del esté proyecto tesis.

Finalmente, en la fase 3 se presenta las herramientas de rendimiento en el mercado para la evaluación de los parámetros y obtener datos estadísticos sobre el rendimiento de la SDN con propósito de comprobar mejoras en la red de estudio al aplicar el algoritmo de balanceo a desarrollarse.

3.1. Situación actual de las redes SDN (Introducción)

Actualmente muchas de las redes SDN presentan complicaciones en el manejo de las tablas de flujo ante tráficos excesivos. En la red el dispositivo de conmutación reenvía el primer paquete de ese flujo al controlador SDN donde se asignará una ruta de reenvió relevante para el flujo. Una política de reenvío se define para cada flujo en las tablas de flujo del equipo SDN (Switch Openflow) (Ramírez, M. & López, A., 2018), por tal motivo existe una cantidad excesiva de paquetes que utilizan un flujo en particular lo cual provoca sobrecarga, pérdida de paquetes, retardo en las comunicaciones y otros aspectos que son cruciales sobre todo en servicios en tiempo real.

Teniendo en cuenta la red de trabajo que se presenta en la Figura 26, el trayecto que tiene el tráfico entre ambos hosts es señalada por las flechas, en donde, ante un tráfico determinado hacia un destino interviene la acción de las tablas de flujo de los conmutadores presentes en la ruta y por ende las acciones que ejecutará en el conmutador es de conmutar el tráfico entrante a otro puerto de salida hasta que el tráfico llegue a su destino.

Conociendo que esta es la ruta predeterminada para alcanzar el destino entre ambos hosts, en situaciones donde exista la excesiva cantidad de tráfico, la red puede colapsar y caer los enlaces dejando sin conectividad entre los extremos de la red hasta que el controlador SDN realice nuevos cálculos y encuentre una ruta alterna entre ambos destinos. Es por ello, la disponibilidad de la red se ve afectada al buscar la solución a la falla presentada después que esta se haya suscitado, produciendo intermitencias en los servicios y descontento por parte de los usuarios de la red.

El objetivo de este trabajo es evitar que la red quede sin conectividad y mejorar el rendimiento al balancear la carga de la red entre todas las rutas disponibles al destino y no sobrecargar a una sola tabla de flujo todas las peticiones de una red origen y destino.

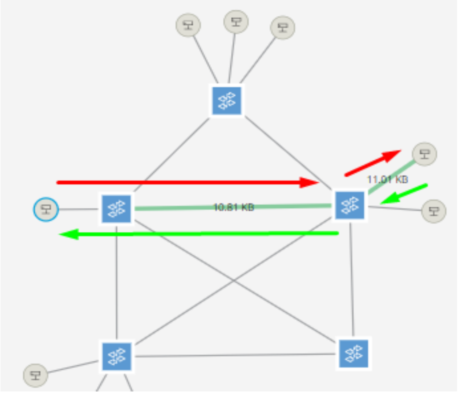


Figura 26 Topología de red SDN

Fuente: Elaboración propia

Pero antes de realizar el estudio del algoritmo de balanceo es necesario disponer de las herramientas necesarias para la recreación de escenarios de prueba o testbed de redes SDN y trabajar sobre estos escenarios. Por tal motivo en la sección 3.2 y 3.3 se realiza la selección del controlador y virtualizadores de una red SDN mediante la norma ISO/IEC/IEEE 29148.

3.2. Selección del controlador SDN

En esta sección se procede a la selección del controlador basando en la norma ISO/IEC/IEEE 29148:2018, mismo que presenta una guía para establecer parámetros de evaluación que especialmente serán ligados al desarrollador de este trabajo y los cuales debe cumplir el controlador entre los presentados.

La selección de realiza mediante la comparación de los requisitos funcionales y no funcionales, ya que mencionan aspectos técnicos necesarios para esta investigación y poder seleccionar un producto de software que se acople a los objetivos planteados en este tema de tesis. Además de presentar el propósito, alcance, funciones de la herramienta seleccionada con respecto a las necesidades de este proyecto de tesis.

3.2.1. Requisitos Funcionales del controlador

En la Tabla 2 se presentan los requisitos funcionales que el controlador debe cumplir para ser seleccionado el más adecuado entre los más presentados para el desarrollo de este proyecto. La calificación se valora en una ponderación de 1 a 3, donde 1 es la valoración baja y 3 la más alta para ser elegido. El cero hace referencia a que no cumple con la especificación de requerimiento de software (SRS) dentro de sus prestaciones. El controlador que posee un valor mayor dentro de la puntuación total es el más ideal para ser levantado y trabajar.

Tabla 2 Requerimientos funcionales para selección del controlador

Nro.	Requerimiento	Prioridad				
		RYU	POX	ODL	ONOS	Floodlight
SRS1	Soporte del protocolo de comunicación OpenFlow 1.3	3	3	3	3	3
SRS2	Controlador da soporte a conmutadores virtuales OVS	3	3	3	3	3
SRS3	Ejecución de scripts en lenguaje de programación de alto nivel	3	1	3	3	3
SRS4	Levantamiento sobre plataformas de software libre	3	3	3	2	2
SRS5	Código abierto para libre modificación y desarrollo.	3	3	3	3	3
SRS6	Alto rendimiento para investigación y desarrollo de pruebas.	3	1	2	2	1
SRS7	Soporte de multiprocesos	3	1	2	2	3
	TOTAL	21	15	19	18	18

Fuente: Elaboración propia

3.2.2. Requisitos no funcionales del controlador

Como su nombre lo especifica, estos requisitos hacen referencia al recurso que se encuentra disponible para el desarrollo y levantamiento del controlador. No necesariamente corresponde a la parte funcional o técnica que debe ofrecer la herramienta, pero es un punto esencial también para la selección. En la Tabla 3 se presentan los requerimientos necesarios para evaluar cada uno de los controladores.

Tabla 3 Requerimientos no funcionales para selección del controlador

Nro.	Requerimiento	Prioridad				
		RYU	POX	ODL	ONOS	Floodlight
SRS8	Lenguaje de programación	3	0	3	3	3
SRS9	Documentación disponible para el levantamiento del controlador	3	1	3	3	2
SRS10	Controlador de fácil instalación	3	1	3	3	2
SRS11	No contar con actualizaciones periódicas constantemente para evitar problemas en las herramientas	2	2	1	2	2
SRS12	Simplicidad en su uso	3	1	3	3	2
	TOTAL	15	6	13	14	11

Fuente: Elaboración propia

Como se presenta en las tablas de valoración el controlador con mejores prestaciones es el controlador RYU debido a que tiene librerías más abiertas con facilidad para ser utilizadas, y aunque su mayor inconveniente es que tiene gran compatibilidad con python 2.7 solamente, para la gran mayoría de los programadores ONOS es un ambiente muy cerrado; Así mismo, este controlador soporta todas las versiones de Openflow, desde la v1.0 hasta la v1.5, por tal motivo, para realizar este proyecto se considerará el controlador RYU, pues además dentro de su página web: osrg.github.io, incluye interfaces API muy completas y bastante documentadas, y componentes de software que simplifican desarrollo del control y administración de una red SDN (Betegón, 2018).

RYU es un controlador SDN de código abierto, escrito en lenguaje Python y desarrollado por la compañía japonesa NTT, este controlador admite todas las versiones del protocolo OpenFlow desarrolladas hasta la actualidad **(Programador, Click;)**.

Una vez que se ha seleccionado el controlador SDN, que para este caso es el controlador RYU, según la norma ISO/IEC/IEEE 29148 propone diferentes ítems de evaluación que hacen que la herramienta seleccionada desea la ideal para alcanzar los objetivos del proyecto a desarrollarse, mismos que se presentan a continuación (Yagues, 2015).

3.2.3. Componentes del controlador RYU

En esta sección se detallan los componentes útiles del directorio RYU / RYU para aplicaciones SDN, también se pueden implementar nuevos o modificar los ya exitsentes (Guerrero, 2017):

3.2.3.1. Ejecutables

Con "**bin / ryu-manager" se lo llama de manera automática.** Este es el ejecutante principal. También determina la clase base **AppManager**, que se utiliza para la administración de la aplicación.

3.2.3.2. Componentes de base

El archivo **"ryu.base.app_manager**" cumple una función muy importante, pues actúa como el centro de administración de otros componentes de la aplicación RYU. Algunas de las utilidades del archivo son cargar la aplicación RYU, proporcionar contextos a las aplicaciones RYU, aceptar la información que fue enviada desde la aplicación y completar el enrutamiento del mensaje dentro de una aplicación RYU (Guerrero, 2017).

Entre sus principales funciones están registrar aplicaciones, buscar y definir la clase base de RyuApp, cerrar sesión, y determinar las propiedades básicas de RyuApp, incluye nombres, observadores, eventos, sus controladores, y varias funciones básicas como start (), stop (), etc (Programador, Click;).

3.2.3.3.Controlador OpenFlow

En la carpeta controller hay archivos muy importantes como “ofp_events.py”, “ofp_handler.py”, “controller.py”, “dpset.py”, etc. En el componente “ryu.controller.controller” se gestiona el canal seguro conectado al conmutador OF, controla las conexiones de interruptores, genera y enruta la publicación de los "eventos" correspondientes para la activación de la lógica de procesamiento del componente suscrito al "evento". En el archivo “ryu.controller.dpset” se definen algunos mensajes del lado del conmutador y está planeado para ser reemplazado por ryu/topología. El archivo “ryu.controller.ofp_events”, para definir eventos de OpenFlow, En el componente “ryu.controller.ofp_handler”, se realiza la administración básica del OpenFlow, incluyendo la negociación (Guerrero, 2017).

3.2.3.4. Aplicaciones RYU

Este directorio incluye algunos archivos como: “ryu.app.simple_switch” que determina un conjunto de estructuras de datos de switch, de aprendizaje OpenFlow 1.0 L2. De igual manera, se tiene “ryu.topology” que determina realizar el cambio y la vinculación del módulo de descubrimiento y está planeado para reemplazar a ryu/controller/dpset.

3.2.3.5. Bibliotecas

En el controlador RYU se tienen algunas bibliotecas:

ryu.lib.packet, para realizar las implementaciones de descodificadores/codificadores de protocolos populares como es el caso de TCP/IP.

ryu.lib.ovs, Ovsdb es un biblioteca de interacción.

ryu.lib.of_config, es una biblioteca OF-Config implementación.

ryu.lib.netconf, es una biblioteca que incluye definiciones de NETCONF que son utilizadas por ryu/lib/of_config.

3.2.4. Alcance del controlador con RYU

El controlador Ryu está programado enteramente en Phyton y soporta las versiones de OpenFlow 1.0 a 1.5, además ha sido probado y certificado con varios switches que soportan el protocolo OpenFlow tales como Open vSwitch, HP, IBM y NEC.

A través de la invocación de un grupo de aplicaciones, Ryu permite administrar eventos de red, analizar solicitudes de cambio de cualquier tipo y reaccionar a los cambios de red con la instalación de flujos adicionales o complementarios, si fuera el caso. Ryu no se considera precisamente un controlador, es más bien una plataforma Component-based SDN; es decir, un ambiente de trabajo que permite implementar redes SDN mediante software (Pachés, 2020).

Para entender la arquitectura de Ryu es una estructura formada por diversos componentes que son las aplicaciones, que son muy variadas y que se pueden modificar para utilizarse o a su vez implementar o crear nuevas.

3.2.5. Funciones del Producto RYU

Algunas de las funcionalidades que destacan del controlador Ryu para utilizar en un proyecto son (Pachés, 2020):

- La capacidad de escuchar eventos asíncronos y observarlos.
- Análisis de paquetes entrantes y enviarlos en la red, así como explorar y distribuír prototipos.
- Capacidad de crear y enviar mensajes y la conectividad con conmutadores virtuales OpenFlow.
- Agilidad para crear infraestructuras SDN y flexibilidad en la API Northbound.

3.3. Selección del simulador de topologías de red SDN.

La selección de realiza mediante la comparación de los requisitos funcionales y no funcionales, ya que mencionan aspectos técnicos necesarios para esta investigación y poder seleccionar un producto de software que se acople a los objetivos planteados en este tema de tesis. Además de presentar el propósito, alcance, funciones de la herramienta seleccionada con respecto a las necesidades de este proyecto de tesis.

3.3.1. Requerimientos funcionales del simulador

Entre los requerimientos funcionales se encuentran en la Tabla 4, donde la calificación se valora en un ponderado de 1 a 3, donde 1 es la valoración baja y 3 la más alta para ser elegido. El cero hace referencia a que no cumple con dicha característica dentro de sus prestaciones. El simulador que posee un valor mayor dentro de la puntuación total es el más ideal para ser ejecutado.

Tabla 4 Requerimientos funcionales para selección del simulador

Nro.	Requerimiento	Prioridad				
		Mininet	GNS3	EstiNet	NS-3	Omnet++
SRS14	Soporte de conmutadores virtuales	3	3	3	3	3
SRS15	Soporte de protocolo OpenFlow v1.3	3	3	3	0	3
SRS16	Integración de servidores	3	3	3	2	1
SRS17	Soporte multiplataforma	3	3	1	3	3
SRS18	Bajo consumo de recursos de hardware	3	2	3	3	3
SRS19	Desempeño y escalabilidad	2	2	3	2	1
	TOTAL	17	16	16	13	14

Fuente: Elaboración propia

3.3.2. Requerimientos no funcionales del simulador

Como su nombre lo especifica, estos requisitos hacen referencia al recurso que se encuentre disponible para el desarrollo y levantamiento del controlador. No necesariamente corresponde a la parte funcional o técnica que debe ofrecer la herramienta, pero es un punto esencial también para la selección. En la tabla 5 se presenta requerimientos no funcionales.

Tabla 5 Requerimientos no funcionales para selección del simulador

Nro.	Requerimiento	Prioridad				
		Mininet	GNS3	EstiNet	NS-3	Omnet++
SRS20	Fuentes de documentación de la herramienta	3	3	3	2	1
SRS21	Interfaz gráfica para el manejo de la herramienta	3	3	3	2	1
SRS22	Fácil manejo y uso de la herramienta	3	3	3	2	1
SRS23	Integra todos los complementos necesarios en una sola instalación	3	3	3	2	1
	TOTAL	12	12	12	8	4

Fuente: Elaboración propia

De acuerdo con las evaluaciones de requerimientos, se presenta que la herramienta de simulación más acorde para el desarrollo del presente trabajo es Mininet, el cual permite el despliegue de una red de datos y conectarla al controlador SDN remoto con el objetivo de volver la red más automática y desarrollar el algoritmo de balanceo de carga propuesto para este proyecto.

Una vez que se ha seleccionado virtualizador de red SDN, que para este caso es Mininet, según la norma ISO/IEC/IEEE 29148 propone diferentes ítems de evaluación que hacen que la herramienta seleccionada desea la ideal para alcanzar los objetivos del proyecto a desarrollarse, mismos que se presentan a continuación:

3.3.3. Propósito con Mininet

El propósito principal que cumple el emulador de redes Mininet con respecto a este proyecto, que es de software libre que permite la creación de redes SDN que soportan el protocolo de comunicación OpenFlow v1.3 y que estos escenarios pueden brindar resultados similares a las redes en producción y sobre todo que soporta el equipo físico en cuestiones de rendimiento.

3.3.4. Alcance con Mininet

Permita la realización de pruebas similares a los equipos reales, implementar servidores de Streaming de Video y Telefonía IP tal como se presenta en el alcance de este trabajo y adicionalmente brinda la conectividad con controladores externos a la plataforma de simulación.

3.3.5. Funciones del Producto con Mininet

Entre las funciones que cumple Mininet con las funciones de este proyecto son:

- Recreación de topologías de redes simples y extensas.
- Comunicación de los equipos de infraestructura creados por el simulador con el controlador SDN.
- Habilitar OpenFlow en su versión 1.3 en los conmutadores.
- Integración a la topología un área de servidores.
- Captura de tráfico mediante sniffers.

3.4. Levantamiento del controlador y virtualizador SDN

En esta sección antes de proceder con los pasos para el montaje de las herramientas de trabajo para una red SDN, se presenta en la Figura 27 un esquema simplificado de las herramientas y equipos físicos a ser utilizados, los cuales consta de: un computador portátil Corei7 Octava Generación con SSD 128GB, 16GB RAM y Sistema Operativo Win10; software de virtualización Vmware Workstation versión 15 Pro; Sistema Operativo GNU/Linux Ubuntu 20.04, Controlador RYU y el emulador de infraestructura de red Mininet 2.1.1.

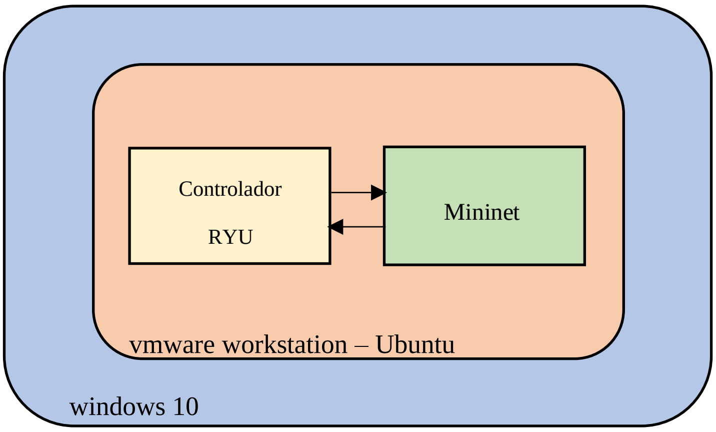


Figura 27 Esquema simplificado de equipos y herramientas

Fuente: Elaboración propia

Una vez comprendido la composición y esquema de los elementos de trabajos, se realiza el levantamiento del controlador RYU, para ello es necesario disponer de un sistema operativo de software libre en base al kernel de Linux, que para este caso es GNU/Linux Ubuntu 20.04, tener instalado Mininet y Python, e instalar los paquetes y complementos necesarios para el levantamiento de la red de trabajo.

Para la instalación del virtualizador de redes SDN de estudio, es necesario descargar la aplicación desde la página oficial de Mininet, misma que es gratuita y se instala directamente en nuestra máquina de trabajo ya sea de la distribución de Windows, Linux o MAC. En la Figura 28, se presenta el entorno de trabajo del emulador que consta de una barra con componentes de red entre otras herramientas necesarias para el desarrollo de este proyecto.

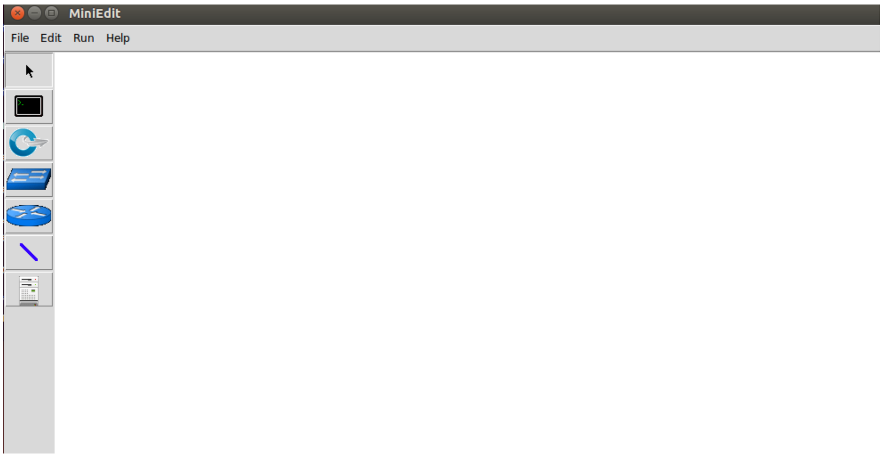


Figura 28 Entorno de trabajo de Mininet

Fuente: Elaboración propia

Con las herramientas en funcionamiento para la simulación de la red SDN, el esquema de trabajo de los equipos, se procede a la al dimensionamiento de escenarios de prueba o testbed en las siguientes subsecciones del apartado 3.5.

3.5. Dimensionamiento de una red SDN en un ambiente controlado

En este apartado se presenta la creación de escenarios de prueba con la finalidad de disponer de un ambiente controlado para el monitoreo de la red y determinar el comportamiento de una red SDN en referencia al enrutamiento de paquetes, métricas, rendimiento y balanceo de carga.

En este siguiente apartado se presentan las características y parámetros de dos tipos de escenarios de prueba que se evaluaran con el desarrollo de este trabajo.

3.5.1. Características y parámetros del primer escenario SDN.

El primer escenario es recreado por el emulador Mininet. Es una topología en forma similar a la de una estrella compuesta por cinco conmutadores virtuales (OpenvSwitch), cada conmutador dispone de múltiples conexiones mediante tecnología ethernet hacia los demás conmutadores presentados en la Figura 29. Cada conmutador maneja un segmento de red y un dispositivo terminal para la generación de tráfico entre los terminales con la finalidad de evaluar el comportamiento de la red SDN en relación al enrutamiento.

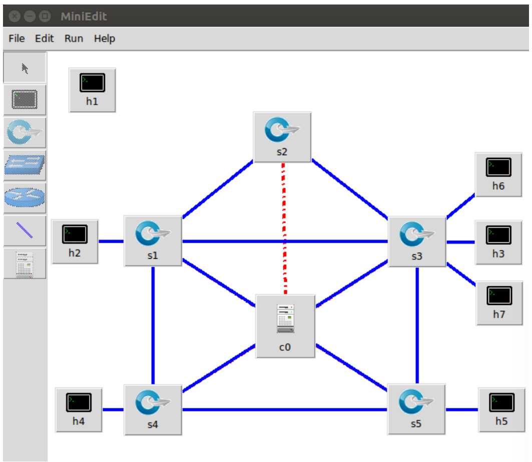


Figura 29 Escenario de pruebas 1 emulador por Mininet

Fuente: Elaboración propia

En esta topología de estudio los nodos se encuentran habilitado con el protocolo de comunicación OpenFlow versión 1.3, el cual permite la comunicación de los equipos de infraestructura virtualizados por Mininet con el controlador SDN-RYU mediante la dirección IP **127.0.0.1** por el puerto tcp 6633.

3.5.2. Características y parámetros del segundo escenario SDN.

En la Figura 30 se presenta el escenario de estudio 2, donde se propone una red más compleja con la intervención de 10 conmutadores virtuales OpenvSwitch, mismos que se encuentran interconectados por múltiples enlaces con la finalidad de presentar más rutas alternas hacia los destinos con la finalidad de más adelante equilibrar la carga de una forma más eficiente como se propone en este proyecto.

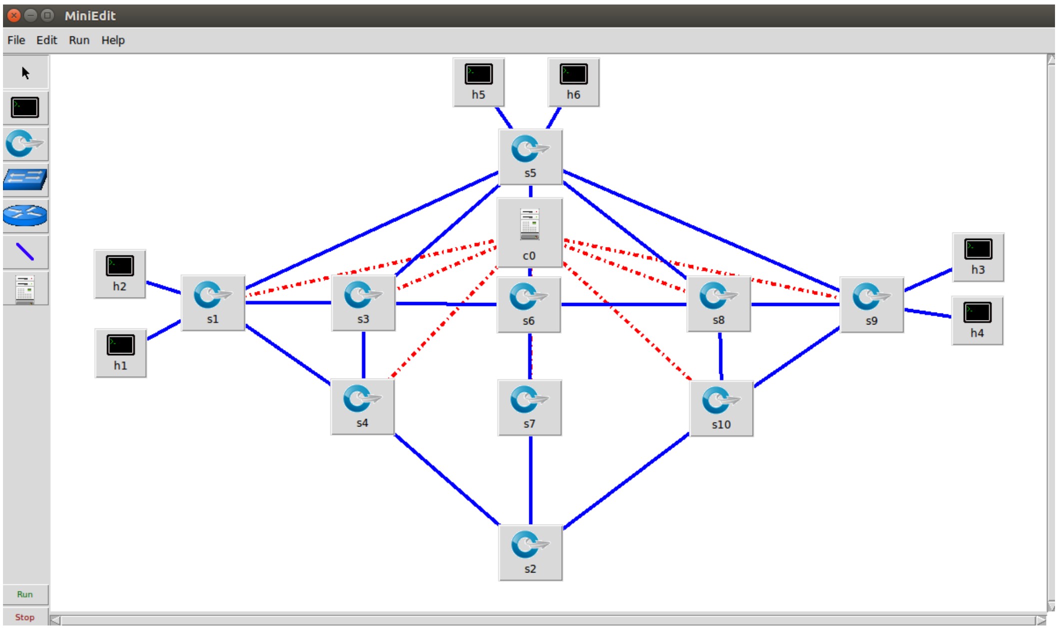


Figura 30 Segundo escensario de prueba virtualizado por Mininet y visualizado por RYU

Fuente: Elaboración propia

Una vez recreadas las redes de prueba se procede a realizar el análisis de los parámetros a ser evaluados en las respectivas redes de prueba como enrutamiento, métricas, rendimiento y balanceo de carga.

3.6. Análisis de enrutamiento de paquetes en redes SDN.

En esta sección se presenta los parámetros analizados en el establecimiento de rutas que toma un paquete hacia sus respectivos destinos. Este apartado tiene como objetivo el establecer las complicaciones que presenta este algoritmo de enrutamiento de paquetes en relación a la saturación de enlaces en la red de transporte SDN.

3.6.1. Primer escenario de prueba o testbed SDN

RYU integra aplicaciones que permite el levantamiento de la red SDN, como por ejemplo el descubrimiento de las redes mediante el número de saltos, misma que se encuentra bajo el seudónimo **Reactive Forwarding** en la sección de Applications del guide del controlador.

Una vez realizada la acción se procede a la construcción de la red sencilla con 5 nodos open vswitch mediante el virutalizador de redes SDN Mininet. En la Figura 31, se presenta la red de trabajo para verificar el manejo del algoritmo de direccionamiento; se realiza un ping desde el h1 a h3, y el controlador toma la ruta más corta, el cual, para este caso, es entre los nodos s3-s1 como se muestra el enlace de color rojo en la figura 31.

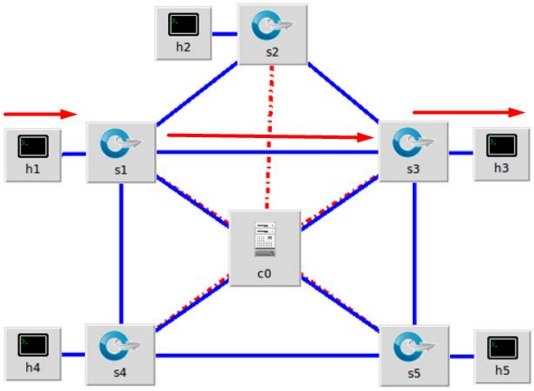


Figura 31 Primera ruta más corta mediante el número de saltos

Fuente: Elaboración propia

Posteriormente se realiza la suspensión del enlace entre el nodo s1-s3, para que, mediante el algoritmo de la ruta más corta, se determine la nueva ruta alterna en la tabla de enrutamiento entre los destinos h1-h3, y se procede a ejecutar el comando ping para generar tráfico a la nueva ruta que corresponde a los nodos s1-s4-s3 como se muestra de color naranja en la Figura 32.

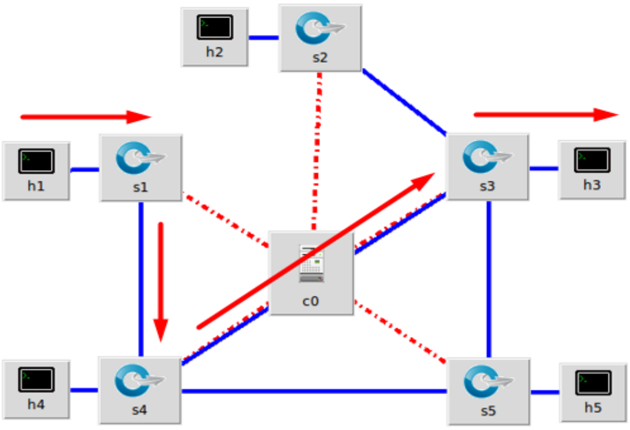


Figura 32 Segunda ruta más corta mediante el número de saltos

Fuente: Elaboración propia

La tercera ruta es determinada por el controlador mediante el cálculo del algoritmo de la ruta más corta es la perteneciente a los enlaces entre los nodos s1-s5-s3 permitiendo no perder la conexión entre los extremos de los hosts h1-h3.

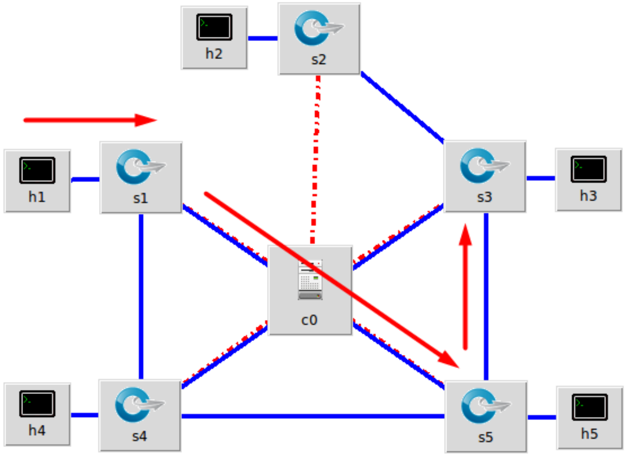


Figura 33 Tercera ruta más corta mediante el algoritmo de Dijkstra

Fuente: Elaboración Propia

Finalmente, la última ruta posible encontrada por el algoritmo de enrutamiento entre los extremos de h1-h3 es mediante el enlace disponible entre los nodos s1-s2-s3, que pertenece a la última instancia de la tabla de enrutamiento que maneja el controlador RYU.

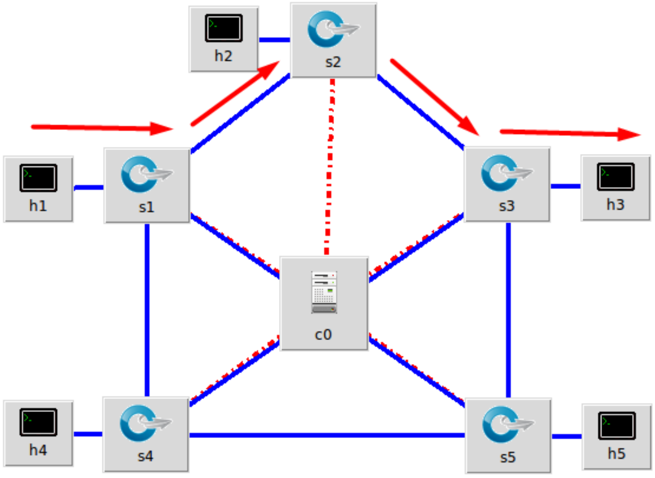


Figura 34 Última ruta más corta mediante el número de saltos

Fuente: Elaboración Propia

Como se muestra en las anteriores figuras, el controlador RYU implementa el algoritmo de número de saltos para la determinación de rutas hacia los destinos, pero como se presenta en el Capítulo 2 de este trabajo, existen algoritmos más eficientes como BFS, DFS o Digraph que utilizan menores recursos tanto en redes simples como extensas de los equipos para la determinación de rutas.

3.6.2. Segundo escenario de prueba o testbed SDN.

3.7. Indicadores o métricas de rendimiento.

La determinación de indicadores para la evaluación del desempeño de la red SDN es importante y necesaria, ya que esto depende del análisis que se vaya a realizar. Las métricas pueden variar ya que estas dependen de los servicios que se esté presenten en la red, por ejemplo, en un servicio de telefonía es necesario tener en cuenta las métricas de retardo, velocidad de transmisión y la cantidad de paquetes perdidos por segundo en equipos intermedios de la red (Jain, 1991).

Es por ello que el determinar las métricas a evaluar deben ser seleccionados de acuerdo a criterios de desempeño de la red que se desea obtener y posteriormente analizar para la obtención de resultados esperados para el presente proyecto.

3.7.1. Métricas medidas en el rendimiento de redes.

En el desarrollo de este proyecto, (Jain, 1991) afirma que para el estudio de una red de transporte se centra principalmente en el análisis en el reenvío de paquetes entre equipos intermedios de la red hasta llegar hacia el destino final y que dicha solicitud sea atendida correctamente. En base a esta afirmación las métricas de desempeño a analizar en este proyecto son las siguientes:

Tasas de transferencia.

Conocido también como throughput, el cual consiste en la medición de la cantidad de paquetes o bits por unidad de tiempo [pps - bps], haciendo referencia también a la velocidad de transporte de datos en la red. Esta métrica permite determinar la cantidad de paquetes que son tratados en los enlaces y evitar la saturación del enlace en relación con la capacidad nominal del enlace.

Ancho de banda utilizado.

El ancho de banda utilizado es medido en bits por segundo [bps] y representa la capacidad efectiva en relación con la capacidad nominal del canal, en otras palabras, la capacidad efectiva es parte utilizada del valor teórico total del canal, siempre la capacidad efectiva será una fracción de la capacidad nominal.

Jitter.

Conocido también como dispersión de retardo. Hace referencia a la variación en tiempos distintos de llegada de los paquetes y se debe a la interferencia en el canal o por congestión de este. La métrica de jitter es importante cuando se necesita tiempos fijos de tratamiento del tráfico especialmente en servicios que requieren fluidez. Este parámetro de análisis no tiene mucha relevancia para la medición del desempeño en la red, especialmente en el escenario 1, ya que este primer testbed no posee servicios en tiempo real que requieran fluidez como telefonía IP o streaming de video. Pero el análisis en el segundo escenario, es requerido, ya que se integra un servidor de VOIP y streaming de video para comprobar la efectividad del algoritmo de balanceo de carga.

Tiempo de respuesta.

Es el retardo dentro de la red a las solicitudes individuales. Se puede definir que es el intervalo entre el final de un envío de solicitud y el comienzo de la respuesta correspondiente del sistema o como el intervalo entre el final de un envío de solicitud y el final del correspondiente.

En el parámetro de tiempo de respuesta intervienen otras sub-parámetros necesarios para determinar el tiempo final, los cuales son: retardo de procesamiento, retardo de cola, retardo de transmisión y retardo de propagación, que en conjunto determinan el valor final de tiempo de respuesta a una solicitud.

Pérdida de Paquetes.

Como su nombre lo especifica es la cantidad de paquetes que necesitan ser reenviados por la red hacia los respectivos destinos debido a la saturación de enlaces o colas extensas en los equipos intermedios, que hacen que los equipos de direccionamiento descarten paquetes en el camino presentando demoras en la presentación de los servicios a los usuarios de una red.

La pérdida de paquetes por lo general es medida por el número de paquetes extraviados en la red o también por su representación en porcentaje (%), este último es el número de paquetes perdidos con relación al total de paquetes enviados desde su origen hacia el destino por la cantidad de 100 como se muestra en la Ecuación 3.

$$\%\, paquetes\ perdidos = \frac{Número\ de\ paquetes\ perdidos}{Número\ total\ de\ paquetes\ enviados} \qquad (Ec.3)$$

3.8. Herramientas para la medición de parámetros de la red SDN

En el mercado existe una variedad de herramientas para la medición del rendimiento en una red de datos, cada una de estas herramientas evalúa determinados parámetros de rendimiento y que a continuación se explica cada uno de ellos.

3.8.1. Medición de Rendimiento con iPERF

Es una de las herramientas más útiles para este proyecto debido a que se integra fácilmente con la red simulada por Mininet. Esta herramienta crea flujos TCP y UDP para la evaluación del rendimiento de la red mediante la valoración de paquetes perdidos, ancho de banda, tasas de transmisión entre los extremos. Además, dicha herramienta es de código abierto y soporta múltiples plataformas como Windows, Linux, Unix entre otros.

iPERF presenta un informe con las marcas de tiempo, la cantidad de datos transmitidos y el rendimiento medido en la red SDN, lo cual permite analizar resultados de rendimiento del antes y después de aplicar el algoritmo de balanceo de carga dinámico de múltiples rutas en las redes de estudio.

3.8.2. Medición de Rendimiento con PING.

Es uno de los comandos comúnmente utilizados para la conectividad entre extremos de una red. Los parámetros que evalúa son: el tiempo de respuesta, el número de paquetes enviados y recibidos, así como, la cantidad y el porcentaje de paquetes perdidos en la red de estudio.

La integración del comando PING, se encuentra en casi todos los sistemas operativos tanto para MAC, Windows, Linux, Unix, mediante su ejecución por comandos en el terminal de cada una de las plataformas mencionadas.

3.8.3. Medición de Rendimiento con Wireshark.

Es un sniffer para la verificación de tráfico en la red. Permite la evaluación del rendimiento mediante gráficos estadísticos de paquetes trasmitidos, tiempos de transmisión, ancho de banda utilizado y la valoración del parámetro jitter para el servicio de telefonía IP, entre otros aspectos que pueden ser de suma importancia al momento de analizar y medir el rendimiento de las redes de datos, que para este caso es la red SDN.

3.9. Monitoreo de la red mediante las herramientas de rendimiento.

En la presente sección se presentan las pruebas realizadas a ambos escenarios propuestos donde se hace énfasis a las métricas establecidas en este proyecto como son la **tasa de transferencia, ancho de banda, jitter, perdida de paquetes y tiempo de respuesta**, con la finalidad de evaluar el rendimiento de la red con configuraciones por defecto. Es necesario explicar que el controlador RYU de fábrica integra el balanceador de carga Round Robin y Random.

3.9.1. Tasas de Transferencia, ancho de banda, tiempo de respuesta.

En la evaluación de las tasas de transferencia del primer escenario se lo realiza con la ayuda de la herramienta iPerf que permite obtener valores de tiempo de respuesta, ancho de banda y tasas de transferencia.

En la generación de tráfico en la red SDN se realiza un ping de carga de 0.06MB semejante a la carga de una solicitud de página WEB básica, en donde se realiza un ping del host **h3 y h7** y la evaluación de rendimiento con iPerf entre **h6 y h2** como se muestra en la Figura 35.

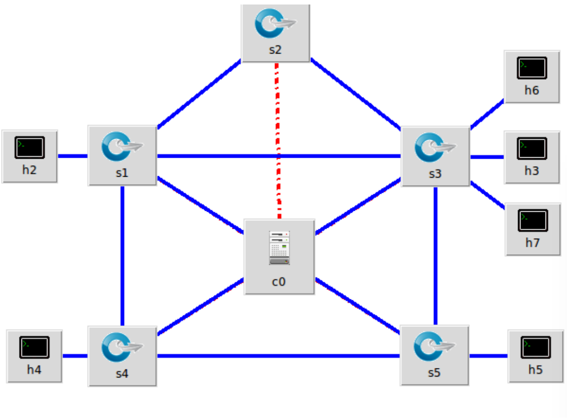
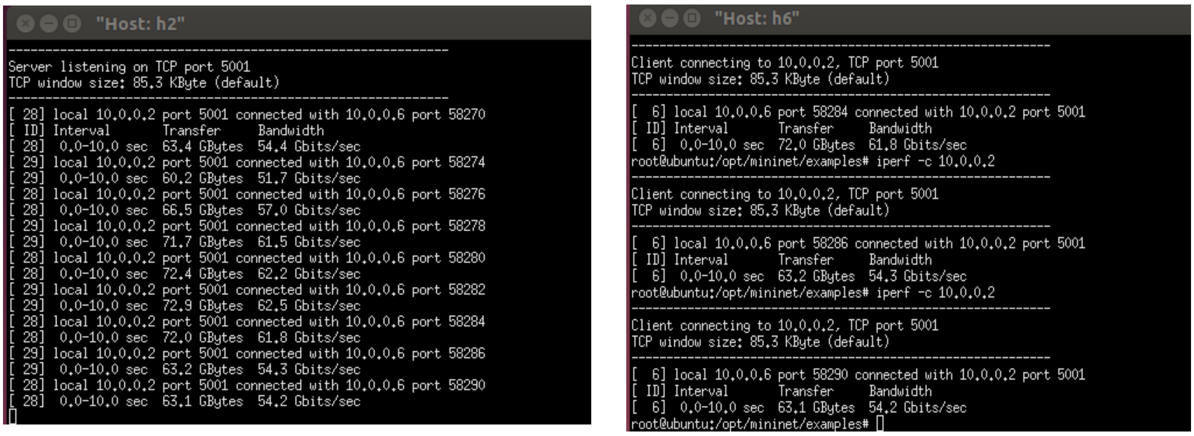


Figura 35 Testbed 1 de la red SDN

Fuente: Elaboración propia

Los valores presentados en la Figura 36 son el rendimiento en relación a las tasas de transferencia en varias pruebas, las mismas que dieron como resultado una mínima de 60.2GBs a la máxima de 72.9GBs; un ancho de banda de 57.0Gbs a 62.2 Gbs y entre 0.0-10.0 sec (segundos) en tiempo de respuesta para la ruta predeterminada por el controlador RYU siendo esta entre el conmutador virtual **s1 y s3.**



```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 28] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58270
[ ID] Interval       Transfer     Bandwidth
[ 28]  0.0-10.0 sec  63.4 GBytes  54.4 Gbits/sec
[ 29] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58274
[ 29]  0.0-10.0 sec  60.2 GBytes  51.7 Gbits/sec
[ 28] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58276
[ 28]  0.0-10.0 sec  66.5 GBytes  57.0 Gbits/sec
[ 29] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58278
[ 29]  0.0-10.0 sec  71.7 GBytes  61.5 Gbits/sec
[ 28] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58280
[ 28]  0.0-10.0 sec  72.4 GBytes  62.2 Gbits/sec
[ 29] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58282
[ 29]  0.0-10.0 sec  72.9 GBytes  62.5 Gbits/sec
[ 28] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58284
[ 28]  0.0-10.0 sec  72.0 GBytes  61.8 Gbits/sec
[ 29] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58286
[ 29]  0.0-10.0 sec  63.2 GBytes  54.3 Gbits/sec
[ 28] local 10.0.0.2 port 5001 connected with 10.0.0.6 port 58290
[ 28]  0.0-10.0 sec  63.1 GBytes  54.2 Gbits/sec
```

```
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.0.6 port 58284 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  72.0 GBytes  61.8 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 10.0.0.2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.0.6 port 58286 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  63.2 GBytes  54.3 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 10.0.0.2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.0.6 port 58290 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  63.1 GBytes  54.2 Gbits/sec
root@ubuntu:/opt/mininet/examples# 
```

Figura 36 Evaluación de rendimiento mediante iPERF en el testbed 1

Fuente: Elaboración propia

En la presentación de la interfaz web del controlador como se ve en la Figura 37, el enlace entre s1 y s3 se puede visualizar el tráfico que se relacionan a los resultados obtenidos por la herramienta de rendimiento iPERF.

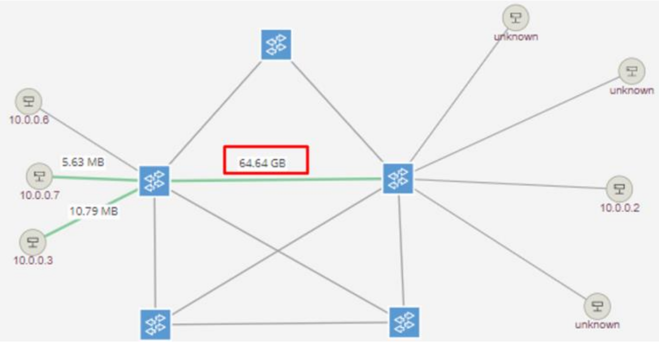


Figura 37 Transferencia de datos en enlace s1 y s3 del GUI RYU

Fuente: Elaboración propia

En el caso del segundo escenario donde se presenta la implementación de servicios de telefonía y video, se procede a realizar una llamada entre los host h2 y h3 mediante el servidor de voz en el host h5 con dirección IP 192.168.86.5, como se muestra en la Figura 38.

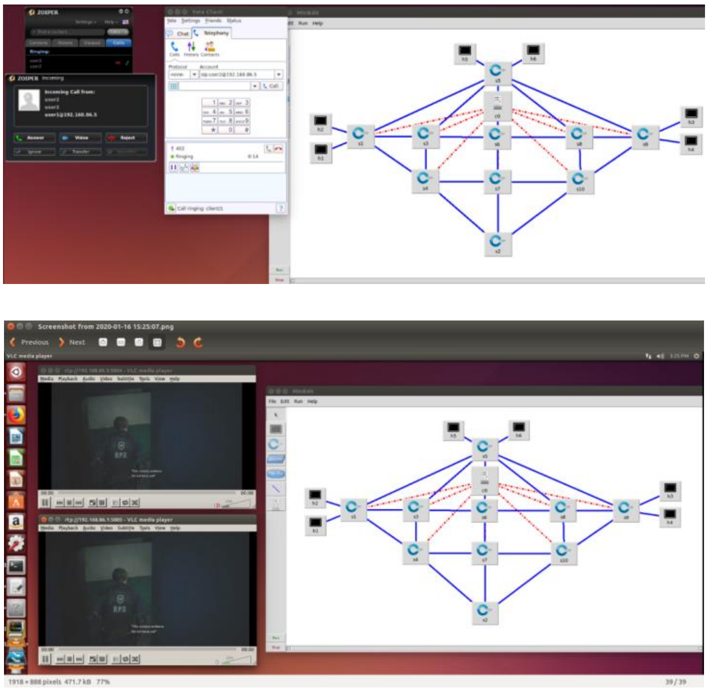
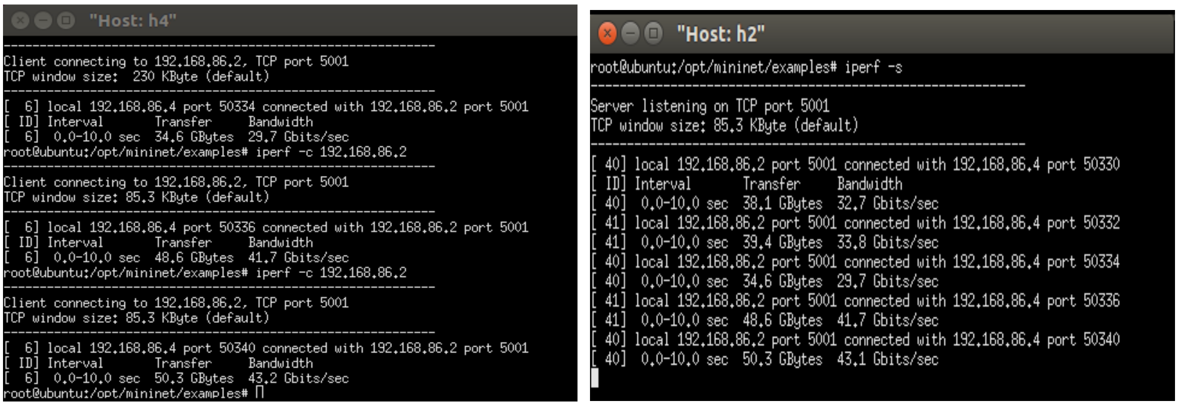


Figura 38 Realización de llamada y Streaming de video en escenario 2 SDN

Fuente: Elaboración propia

Donde secuencialmente, se procede a evaluar el rendimiento mediante la herramienta iPerf para conocer valores referentes a las métricas de tasas de transferencia, ancho de banda y tiempo de respuesta como se muestra en la Figura 39, donde se tiene valores de tasa de transferencia van de 34.6GBs a 50.3GBs, ancho de banda de 29.7Gbs a 43.1Gbs y un tiempo de respuesta entre 0 a 10 sec (segundos) para la ruta predeterminada **S1-S5-S9** como se observa en la Figura 40.



```
------------------------------------------------------------
Client connecting to 192.168.86.2, TCP port 5001
TCP window size:  230 KByte (default)
------------------------------------------------------------
[  6] local 192.168.86.4 port 50334 connected with 192.168.86.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  34.6 GBytes  29.7 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.86.2
------------------------------------------------------------
Client connecting to 192.168.86.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 192.168.86.4 port 50336 connected with 192.168.86.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  48.6 GBytes  41.7 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.86.2
------------------------------------------------------------
Client connecting to 192.168.86.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 192.168.86.4 port 50340 connected with 192.168.86.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  50.3 GBytes  43.2 Gbits/sec
root@ubuntu:/opt/mininet/examples# 
```

```
root@ubuntu:/opt/mininet/examples# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 40] local 192.168.86.2 port 5001 connected with 192.168.86.4 port 50330
[ ID] Interval       Transfer     Bandwidth
[ 40]  0.0-10.0 sec  38.1 GBytes  32.7 Gbits/sec
[ 41] local 192.168.86.2 port 5001 connected with 192.168.86.4 port 50332
[ 41]  0.0-10.0 sec  39.4 GBytes  33.8 Gbits/sec
[ 40] local 192.168.86.2 port 5001 connected with 192.168.86.4 port 50334
[ 40]  0.0-10.0 sec  34.6 GBytes  29.7 Gbits/sec
[ 41] local 192.168.86.2 port 5001 connected with 192.168.86.4 port 50336
[ 41]  0.0-10.0 sec  48.6 GBytes  41.7 Gbits/sec
[ 40] local 192.168.86.2 port 5001 connected with 192.168.86.4 port 50340
[ 40]  0.0-10.0 sec  50.3 GBytes  43.1 Gbits/sec
```

Figura 39 Evaluación de rendimiento mediante iPERF en el testbed 2

Fuente: Elaboración propia

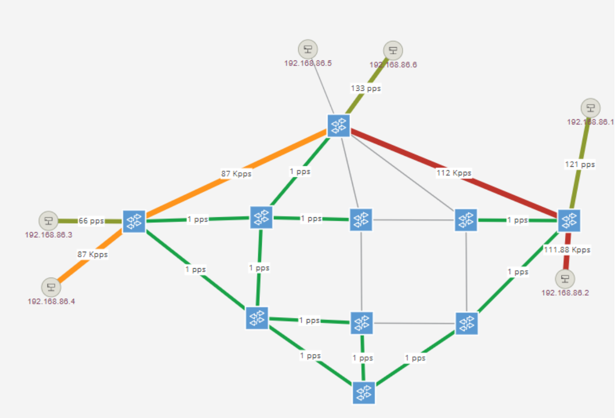


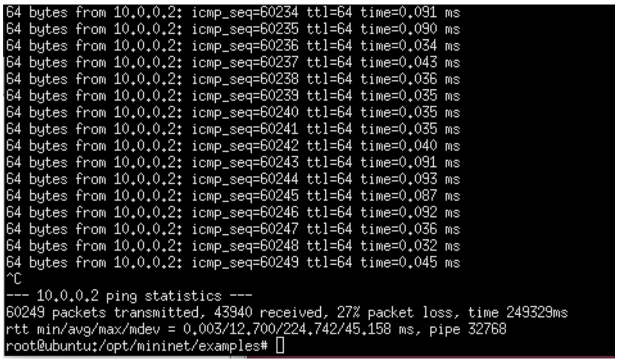
Figura 40 Transferencia de datos en enlace s1-s5-s9 del GUI RYU

Fuente: Elaboración propia

Con estos resultados, se puede decir que existirá un momento en donde el enlace se sature y genere complicaciones en la red como retraso e intermitencias en la presentación de servicios que se dispongan en la red debido a que el sistema utiliza una única ruta para llegar a los destinos.

3.9.2. Perdidas de Paquetes

Para la determinación de paquetes perdidos en la red se lo realiza mediante el uso de la herramienta de PING en el terminal de los hosts. Se realiza la ejecución del ping en el escenario 1, desde el host h3 al h2, el cual presenta una pérdida del 27% de paquetes perdidos en la red, es decir que, de 60249 paquetes enviados, 43940 fueron respondidos por parte del destinatario a las solicitudes que llegaron a este host, los demás paquetes son considerados perdidos o eliminador por la red. En la Figura 41, se presentan los resultados especificados anteriormente de la pérdida de paquetes en la red.

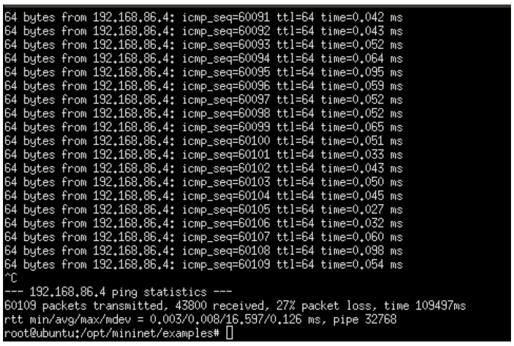


```
64 bytes from 10.0.0.2: icmp_seq=60234 ttl=64 time=0.091 ms
64 bytes from 10.0.0.2: icmp_seq=60235 ttl=64 time=0.090 ms
64 bytes from 10.0.0.2: icmp_seq=60236 ttl=64 time=0.034 ms
64 bytes from 10.0.0.2: icmp_seq=60237 ttl=64 time=0.043 ms
64 bytes from 10.0.0.2: icmp_seq=60238 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_seq=60239 ttl=64 time=0.035 ms
64 bytes from 10.0.0.2: icmp_seq=60240 ttl=64 time=0.035 ms
64 bytes from 10.0.0.2: icmp_seq=60241 ttl=64 time=0.035 ms
64 bytes from 10.0.0.2: icmp_seq=60242 ttl=64 time=0.040 ms
64 bytes from 10.0.0.2: icmp_seq=60243 ttl=64 time=0.091 ms
64 bytes from 10.0.0.2: icmp_seq=60244 ttl=64 time=0.093 ms
64 bytes from 10.0.0.2: icmp_seq=60245 ttl=64 time=0.087 ms
64 bytes from 10.0.0.2: icmp_seq=60246 ttl=64 time=0.092 ms
64 bytes from 10.0.0.2: icmp_seq=60247 ttl=64 time=0.036 ms
64 bytes from 10.0.0.2: icmp_seq=60248 ttl=64 time=0.032 ms
64 bytes from 10.0.0.2: icmp_seq=60249 ttl=64 time=0.045 ms
^C
--- 10.0.0.2 ping statistics ---
60249 packets transmitted, 43940 received, 27% packet loss, time 249329ms
rtt min/avg/max/mdev = 0.003/12.700/224.742/45.158 ms, pipe 32768
root@ubuntu:/opt/mininet/examples# 
```

Figura 41 Resultados de pérdidas de paquetes en la red SDN testbed 1

Fuente: Elaboración propia

De igual forma se procede hacer un ping desde el host h2 al h4 para el testbed 2, obteniendo los siguientes resultados en la perdida de paquetes. El número de paquetes enviados es de 60109, de los cuales 43800 fueron respondidos por el destino, el cual representa en porcentaje el 27% de paquetes perdidos en la red como se muestra en la Figura 42.



```
64 bytes from 192.168.86.4: icmp_seq=60091 ttl=64 time=0.042 ms
64 bytes from 192.168.86.4: icmp_seq=60092 ttl=64 time=0.043 ms
64 bytes from 192.168.86.4: icmp_seq=60093 ttl=64 time=0.052 ms
64 bytes from 192.168.86.4: icmp_seq=60094 ttl=64 time=0.064 ms
64 bytes from 192.168.86.4: icmp_seq=60095 ttl=64 time=0.095 ms
64 bytes from 192.168.86.4: icmp_seq=60096 ttl=64 time=0.059 ms
64 bytes from 192.168.86.4: icmp_seq=60097 ttl=64 time=0.052 ms
64 bytes from 192.168.86.4: icmp_seq=60098 ttl=64 time=0.052 ms
64 bytes from 192.168.86.4: icmp_seq=60099 ttl=64 time=0.065 ms
64 bytes from 192.168.86.4: icmp_seq=60100 ttl=64 time=0.051 ms
64 bytes from 192.168.86.4: icmp_seq=60101 ttl=64 time=0.033 ms
64 bytes from 192.168.86.4: icmp_seq=60102 ttl=64 time=0.043 ms
64 bytes from 192.168.86.4: icmp_seq=60103 ttl=64 time=0.050 ms
64 bytes from 192.168.86.4: icmp_seq=60104 ttl=64 time=0.045 ms
64 bytes from 192.168.86.4: icmp_seq=60105 ttl=64 time=0.027 ms
64 bytes from 192.168.86.4: icmp_seq=60106 ttl=64 time=0.032 ms
64 bytes from 192.168.86.4: icmp_seq=60107 ttl=64 time=0.060 ms
64 bytes from 192.168.86.4: icmp_seq=60108 ttl=64 time=0.098 ms
64 bytes from 192.168.86.4: icmp_seq=60109 ttl=64 time=0.054 ms
^C
--- 192.168.86.4 ping statistics ---
60109 packets transmitted, 43800 received, 27% packet loss, time 109497ms
rtt min/avg/max/mdev = 0.003/0.008/16.597/0.126 ms, pipe 32768
root@ubuntu:/opt/mininet/examples# 
```

Figura 42 Resultados de pérdidas de paquetes en la red SDN testbed 2

Fuente: Elaboración propia

3.9.3. Jitter

Para medir el nivel de jitter presentado en una llamada; se realiza la simulación del escenario de prueba 2 donde el servidor con IP 192.168.86.5 establece la señalización de la llamada de los hosts h2 y h3 con el servidor de voz y posteriormente los paquetes de voz RTP son enviados y recibidos por los terminales creando un canal directo entre ambos como se puede observar en la Figura 43.

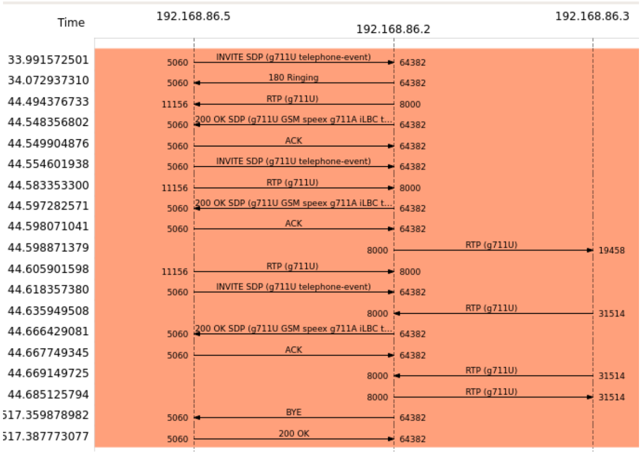
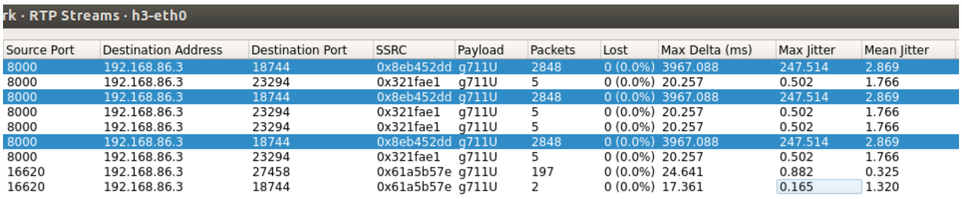


Figura 43 Secuencia de iniciación y finalización de llamada en la red SDN

Fuente: Elaboración propia

Posteriormente mediante la herramienta Wireshark, se pueden tener diferentes parámetros de medición de la llamada realizada y en especial al medir la métrica jitter, se obtiene un valor de 247.514 ms como se observa en la Figura 44, el cual está fuera del rango permitido que es de 150 ms para brindar calidad de la llamada al usuario. En base a esto se concluye que la llamada es afectada cuando existe un tráfico crítico en los enlaces de la red.



Source Port	Destination Address	Destination Port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter	Mean Jitter
8000	192.168.86.3	18744	0x8eb452dd	g711U	2848	0 (0.0%)	3967.088	247.514	2.869
8000	192.168.86.3	23294	0x321fae1	g711U	5	0 (0.0%)	20.257	0.502	1.766
8000	192.168.86.3	18744	0x8eb452dd	g711U	2848	0 (0.0%)	3967.088	247.514	2.869
8000	192.168.86.3	23294	0x321fae1	g711U	5	0 (0.0%)	20.257	0.502	1.766
8000	192.168.86.3	23294	0x321fae1	g711U	5	0 (0.0%)	20.257	0.502	1.766
8000	192.168.86.3	18744	0x8eb452dd	g711U	2848	0 (0.0%)	3967.088	247.514	2.869
8000	192.168.86.3	23294	0x321fae1	g711U	5	0 (0.0%)	20.257	0.502	1.766
16620	192.168.86.3	27458	0x61a5b57e	g711U	197	0 (0.0%)	24.641	0.882	0.325
16620	192.168.86.3	18744	0x61a5b57e	g711U	2	0 (0.0%)	17.361	0.165	1.320

Figura 44 Jitter presentado en la realización de una llamada en SDN

Fuente: Elaboración propia

A continuación, en la Figura 45 se presenta un diagrama de barras de los parámetros medidos en los diferentes escenarios, para posteriormente comparar estos resultados con los que se obtendrán después de aplicar el algoritmo de balanceo de carga de múltiples rutas dinámico en la red SDN, y comprobar que existe una mejora en el tratamiento del tráfico en la red al disminuir el uso de los recursos.

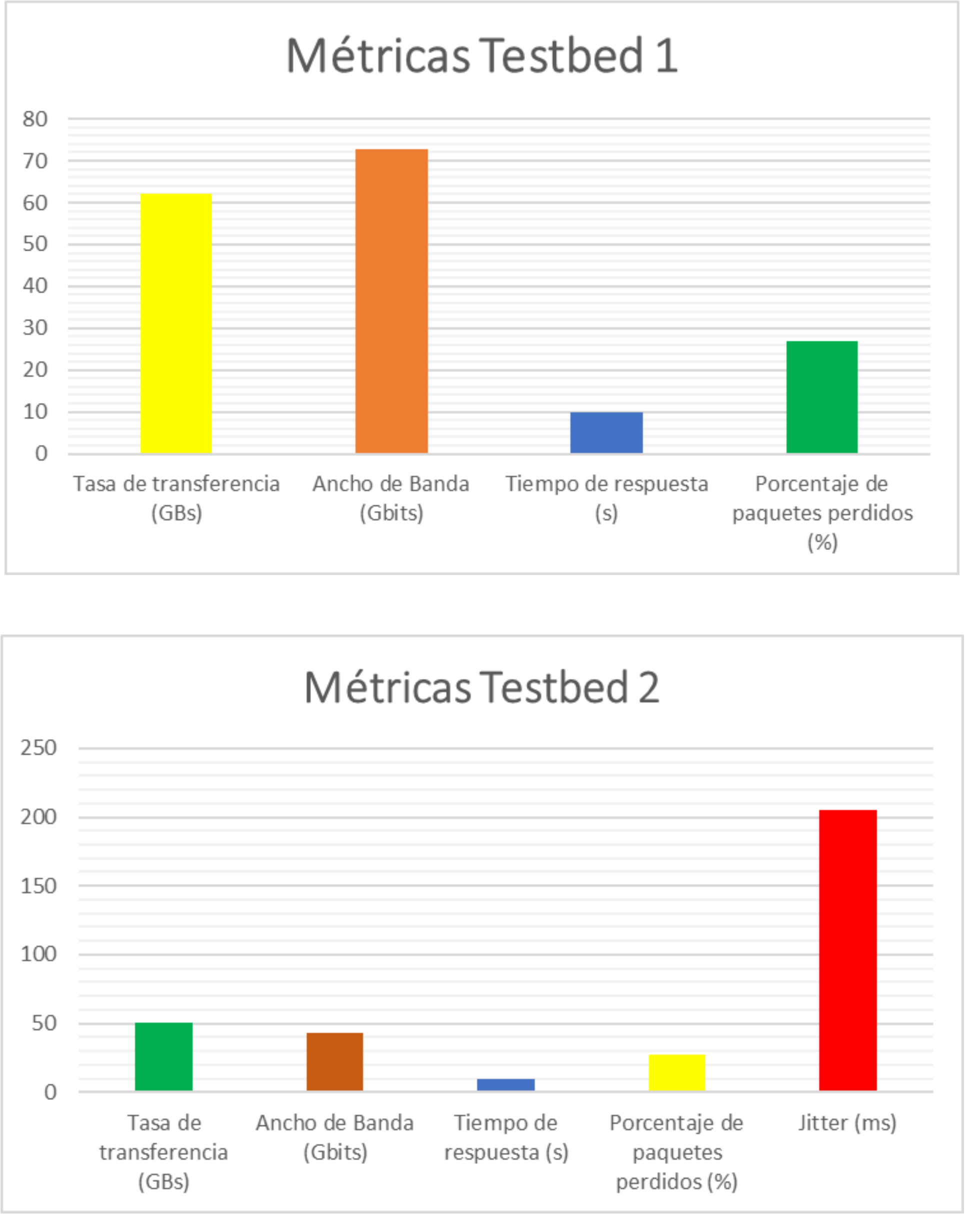


Figura 45 Diagrama de barras de las métricas de rendimiento de las redes SDN

Fuente: Elaboración propia

Capítulo 4. Desarrollo del algoritmo de balanceo de carga dinámico

4.1 Diseño del algoritmo de balanceo de carga

Los protocolos de enrutamiento mantienen tablas de enrutamiento dinámicas por medio de mensajes de actualización del enrutamiento, que contienen información acerca de los cambios sufridos en la red, y que indican al software del router que actualice la tabla de enrutamiento en consecuencia. Intentar utilizar el enrutamiento dinámico sobre situaciones que no lo requieren es una pérdida de ancho de banda, esfuerzo, y en consecuencia de dinero.

4.1.1 Criterios de diseño del algoritmo

Escalabilidad

La escalabilidad, en una red SDN es muy importante, pues la capacidad de adaptación y respuesta de un sistema con respecto a su rendimiento y a medida que aumenta significativamente el número de usuarios del mismo, al aplicar un modelo escalable horizontalmente, no existen limitaciones de crecimiento a priori, pues implica que la creación de nuevos nodos dentro de la red está cubiertos desde el algoritmo de enrutamiento dinámico.

Flexibilidad

Así es como, en el modo dinámico, los routers pueden descubrir cualquier información automáticamente y compartirla con otros routers a través de algoritmos de enrutamiento dinámicos. Esto es básicamente el lenguaje que utiliza un router para comunicarse con otros routers y cuyo objetivo es compartir información sobre la accesibilidad y el estado de las redes, determinando automáticamente la mejor ruta hacia un destino, si la ruta vigente está inaccesible.

Los dispositivos de la red “aprenden” de manera automática, por lo que la configuración manual de rutas pasa a un segundo plano, y por lo tanto, en caso de ocurrencias de fallo en la red, el algoritmo de enrutamiento por sí solo genera una nueva ruta ideal, sin modificar la configuración.

Factibilidad

Este algoritmo de enrutamiento es factible ya que permitirá mejorar el estado actual de la red, mejorar su funcionalidad y establecer conexiones con mejor prestación de servicios disminuyendo la pérdida de paquetes y el retraso de la señal. Este algoritmo permite establecer una mejor comunicación cuando se transmite la señal.

4.1.1.1 Enrutamiento de paquetes.

Para el caso de un enrutamiento dinámico, las tablas de enrutamiento dinámicas se mantienen actualizadas por los protocolos de enrutamiento utilizando mensajes de actualización del enrutamiento que incluyen información acerca de cualquier cambio sucedido en la red, lo que se indica al software del router para que actualice la tabla de enrutamiento en consecuencia. El enrutamiento dinámico aplicado sobre situaciones de red que no lo requieren, significaría una pérdida de ancho de banda, esfuerzo, y por supuesto de dinero.

4.1.1.2 Selección de rutas hacia el destino.

En un inicio se ha procedido a identificar todos los elementos de la red y todas las rutas posibles, compuestas por todos los enlaces disponibles dentro de la red; luego se realiza ruta por ruta, el cálculo de todos los parámetros de calidad de servicio cumpliendo con las restricciones impuestas por los servicios, finalmente, se normalizan los valores adquiridos y se realiza el cálculo del valor de la función objetivo. Entonces, las rutas resultantes cumplen con los mejores parámetros de calidad de servicio y permiten a su vez la transmisión de todos los servicios que los usuarios requieren.

4.1.1.3 Estado del enlace.

El estado del enlace recrea la topología de toda la red de manera precisa, denominado también “Primero la Ruta Libre Mas Corta” DiGraph. Utiliza una métrica basada en el ancho de banda, retardo, carga y confiabilidad, de cada uno de los enlaces posibles y disponibles para llegar a un destino determinado, en base a los conceptos mencionados el protocolo escoge una ruta por sobre otra. Esta clase de protocolos utilizan cierto tipo de publicaciones conocidas como Reconocimiento de estado de enlace (LSA), las mismas que se intercambian entre los routers, a través del estado de enlace cada router crea una base de datos de la topología completa de la red.

- Se busca la unión común de la topología completa de la red.

- Cada uno de los dispositivos realiza el cálculo de la ruta más corta hacia los otros routers.

- Por cada cambio (cada evento) en la topología de la red se activan las actualizaciones.

- Transmisión de las actualizaciones.

4.1.1.4 Mecanismo de balanceo de la carga en enlaces.

Cuando un router tiene disponible varias rutas hacia un mismo destino, y todas las rutas cuentan con métricas y distancias administrativas similares, en este caso, puede darse el balanceo de carga, ya sea balanceo por destino o balanceo por paquete.

La métrica es considerada como el análisis, y en la que el algoritmo del protocolo de enrutamiento dinámico se basa para definir y seleccionar una ruta por sobre otra, de este modo, el protocolo creará en el router la tabla de enrutamiento y publicará únicamente las mejores rutas. Un protocolo de enrutamiento utiliza métrica para determinar qué vía utilizar para transmitir un paquete a través de un Intercambio. La métrica que utilizan los protocolos de enrutamiento incluyen:

- Número de saltos: que es la cantidad de routers por los que debe pasar un paquete.

- Pulsos: el retraso que se da en un enlace de datos contabilizado por pulsos de reloj de PC.

- Coste: es un valor basado generalmente en el ancho de banda, el costo económico o cualquier otra medida.

- Ancho de banda: la capacidad de datos que posee un enlace.

- Carga: la cantidad de la actividad que existe en un recurso de red, como un router o un enlace.

- Fiabilidad: es la tasa de errores de bits de cada enlace de red.

- MTU: es la unidad máxima de transmisión y cuya máxima longitud de trama en octetos es la aceptada por todos los enlaces de la ruta.

Los protocolos de enrutamiento almacenan los resultados de los datos mencionados en una tabla de enrutamiento.

4.1.2 Diagrama de flujo del mecanismo de funcionamiento.

A continuación, se presenta el diagrama de flujo de funcionamiento del algoritmo del balanceador de carga propuesto:

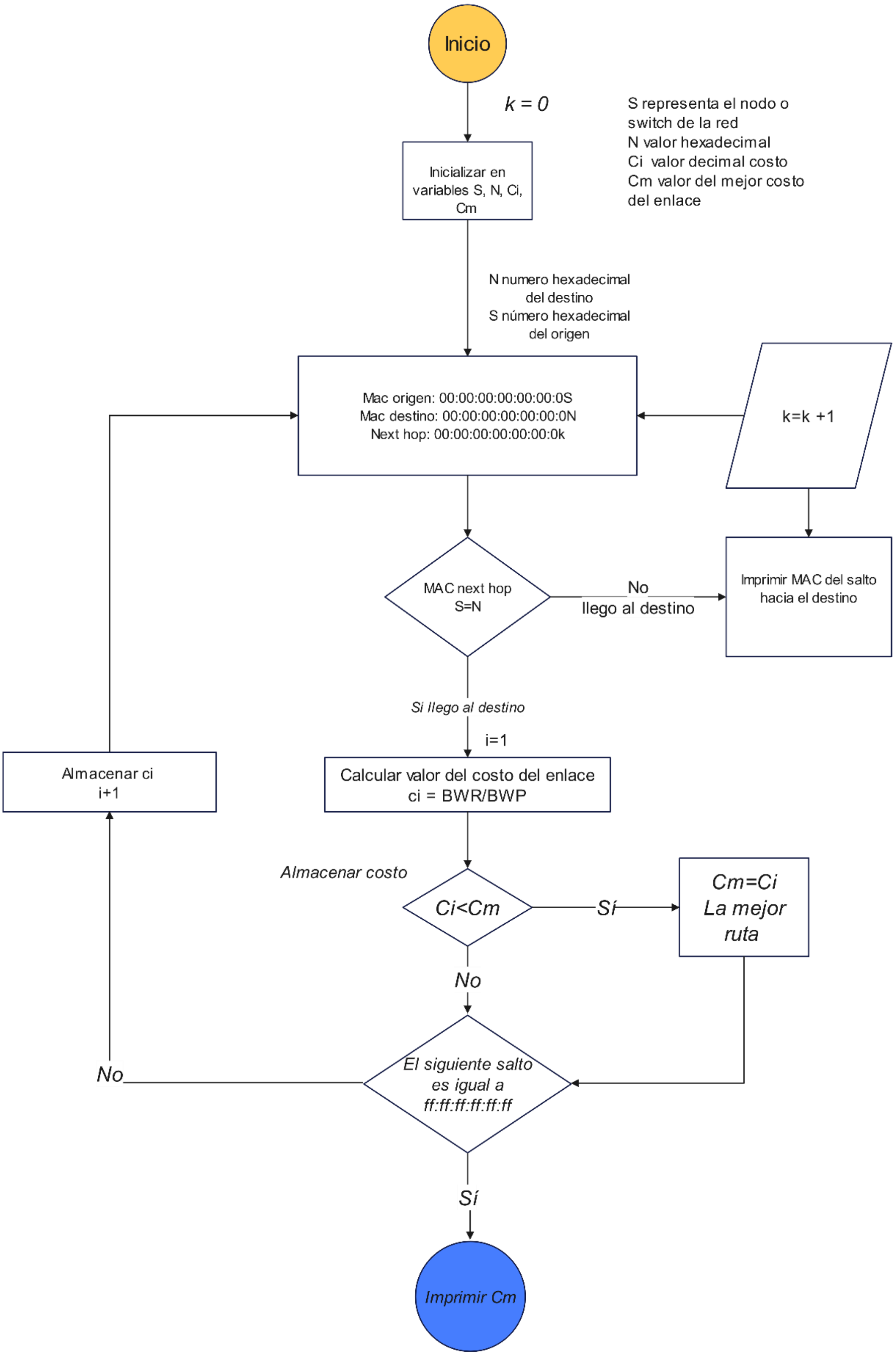


Figura 46 Flujograma del algoritmo del balanceador de cargas

Fuente: Elaboración propia

4.2 Programación del algoritmo

En la siguiente sección se detalla el código de desarrollo del algoritmo de balanceo de carga bajo una red SDN, es importante mencionar que el desarrollo del código lenguaje está realizado en Python.

4.2.1 Desarrollo de scripts en lenguaje de programación para RYU

En la presente sección, se ha creado la clase failOver (nombre utilizado para referir al proyecto en desarrollo), en la cual se inicializan los protocolos a utilizarse (en este caso se ha utilizado la versión 3 pero tiene soporte para las demás)

```
class failOver(app_manager.RyuApp):
  OFP_VERSIONS = [(ofproto_v1_0.OFP_VERSION),
          (ofproto_v1_2.OFP_VERSION),
          (ofproto_v1_3.OFP_VERSION),
          (ofproto_v1_4.OFP_VERSION),
          (ofproto_v1_5.OFP_VERSION)]

  _CONTEXTS = {'stplib': stplib.Stp}
```

Inmediatamente después se realiza la inicialización del constructor considerado (__init__) y con ellos las variables necesarias para el funcionamiento del proyecto. La función *self.net = nx.DiGraph()* (NetworkX developers, 2004-2022), es una de las funciones más importantes que se requiere en el proyecto, porque en éste punto se utiliza la función DiGraph para almacenar el grafo resultante de la topología creada.

```
  def __init__(self, *args, **kwargs):
    super(failOver, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self.net = nx.DiGraph()
    self.count = 0
```

```
    self.stp = kwargs['stplib']
```

El bloque siguiente se encarga de inicializar y preconfigurar el escenario de trabajo. Para que no existan problemas a la hora de ejecutar el algoritmo, se han vinculado las librerías simple_switch13 y simple_switch_stp_13, en ellas se han modificado y ajustado los parámetros en post de lo que se busca con su desarrollo, obteniendo los resultados satisfactorios.

```
    config = {dpid_lib.str_to_dpid('10:00:00:00:00:00:01'):
      {'bridge': {'priority': 0x8000}},
      dpid_lib.str_to_dpid('10:00:00:00:00:00:02'):
      {'bridge': {'priority': 0x9000}},
      dpid_lib.str_to_dpid('10:00:00:00:00:00:0f'):
      {'bridge': {'priority': 0xa000}}}

    self.stp.set_config(config)
```

En el bloque siguiente de código se requiere el método o función que se utiliza para modificar una topología determinada para que se pueda aplicar a diferentes variedades topológicas de las distintas redes a ser analizadas.

```
  def Manipulando_Topologias(self, ev):
    dp = ev.dp
    dpid_str = dpid_lib.dpid_to_str(dp.id)
    msg = 'Recibiendo Topologias MACs.'
    if dp.id in self.mac_to_port:
      self.delete_flow(dp)
      del self.mac_to_port[dp.id]
```

En este bloque de código, se determina el método o función que se utiliza para mostrar o visualizar el comportamiento de un puerto específico del switch en cuestión, permitiendo identificar si el puerto se encuentra como: “Disabled”, “Block”, “Listen”, “Learn”, forward”.

```
  def _port_state_change_handler(self, ev):
    dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
    of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
```

```
        stplib.PORT_STATE_BLOCK: 'BLOCK',
        stplib.PORT_STATE_LISTEN: 'LISTEN',
        stplib.PORT_STATE_LEARN: 'LEARN',
        stplib.PORT_STATE_FORWARD: 'FORWARD'}
```

A continuación, se ejecuta el método o función que es utilizado para determinar y capturar tanto el origen “src” como el destino “dst” de un determinado paquete de datos:

```
def Manipulando_Paquetes(self, ev):
   pkt = packet.Packet(msg.data)
   eth = pkt.get_protocols(ethernet.ethernet)[0]
   dst = eth.dst
   src = eth.src
   N = dst
   k = 0
  Cj= 0

   dpid = datapath.id
   self.mac_to_port.setdefault(dpid, {})

   self.logger.info("PAQUETE ENTRANDO", dpid, src, dst, in_port)
   self.mac_to_port[dpid][src] = in_port
```

A continuación, se puede ver el método o función utilizada para agregar un nuevo camino o ruta entre 2 elementos determinados de la red donde Next es el valor del siguiente salto hacia el destino.

```
      if src not in self.net:
       self.net.add_node(src)
       self.net.add_edge(dpid, src, {'port':in_port})
       self.net.add_edge(src, dpid)

    if dst in self.net:
       path = nx.bidirectional_shortest_path(self.net, src, dst)
       next = path[path.index(dpid)+1]
```

```
    S = next
    out_port = self.net[dpid][next]['port']
    pkt = self.net[dpid][next]['pkt']
    self.logger.info("dst encontrado, salida pkt to port %s", out_port)
    self.logger.info("ruta is : %s", str(path))
```

En este punto comienza el Balanceador de Cargas propuesto, y en el siguiente fragmento del código se realiza la iniciación de las variables a ser utilizadas previamente, luego se guardan los datos en las variables src, dst y el valor del dato.

```
if S=N
def failOVER(self, ev):
     msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    ci = Br/Bf
    in_port = msg.match['in_port']
if ci>cm
    cm = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
```

A continuación, se visualiza un fragmento de código que se utiliza para comprobar cuál es el tipo de protocolo de red con el cual está trabajando; es decir, IPV4 o IPV6.

```
    if eth.ethertype == ether_types.ETH_TYPE_LLDP or eth.ethertype == ether_types.ETH_TYPE_IPV6:
       return
    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})
```

Esta pequeña rutina que se muestra, lo que hace es simplemente validar que la dirección ARP no se encuentre en uso

```
if src not in self.mac_to_port[dpid]:
    self.mac_to_port[dpid][src] = in_port
```

En el siguiente fragmento del código se validan todos los valores y se comprueba que todos ellos sean los correctos. Adicionalmente, en este tramo se realiza la validación de origen y destino de los paquetes de datos

```
    self.logger.info("origen desconocido mac to port===================================")
    self.logger.info(self.mac_to_port)
else:

    self.logger.info("origen mac to port==============================================")
    self.logger.info(self.mac_to_port)
    if self.mac_to_port[dpid][src] != in_port and str(dst) == "ff:ff:ff:ff:ff:ff":
         actions = []
        match = parser.OFPMatch(eth_src=src,in_port = in_port)
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=prio,
                                    match=match, instructions=inst)

        datapath.send_msg(mod)
        self.logger.info("PUERTO INCORRECTO: %s, PAQUETE BLOQUEADO", in_port)

        return
```

En el siguiente bloque, se requiere de la función que realiza la impresión de la topología definida en la red

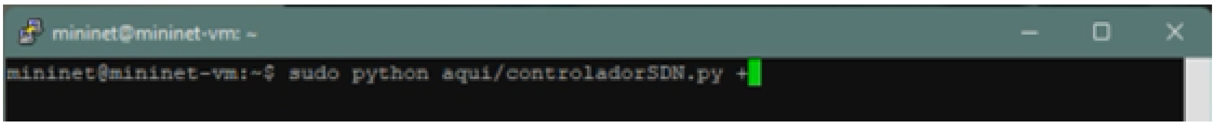
```
def imprimir_topologia(self, ev):
    switch_list = get_switch(self, None)
    switches = [i.dp.id for i in switch_list]
    link_list = get_link(self, None)
    links = [(link.src.dpid, link.dst.dpid, {'port':link.src.port_no}) for link in link_list]
    print ("Links:-----------------------------------------------------")
    print (links)
```

```
print ("Size: ", str(len(links)))
print ("----------------------------------------------------------------")
self.net.add_nodes_from(switches)
self.net.add_edges_from(links)
if len(switches) >= 20:
    print
("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++")
    return
```

4.3 Compilación del algoritmo y detección de errores de programación.

4.3.1 Compilación del controlador SDN. RYU

A continuación, se visualizan en la Figura 47 la compilación del controlador, para empezar, en la siguiente pantalla se puede visualizar el comando para la iniciación del controlador SDN:

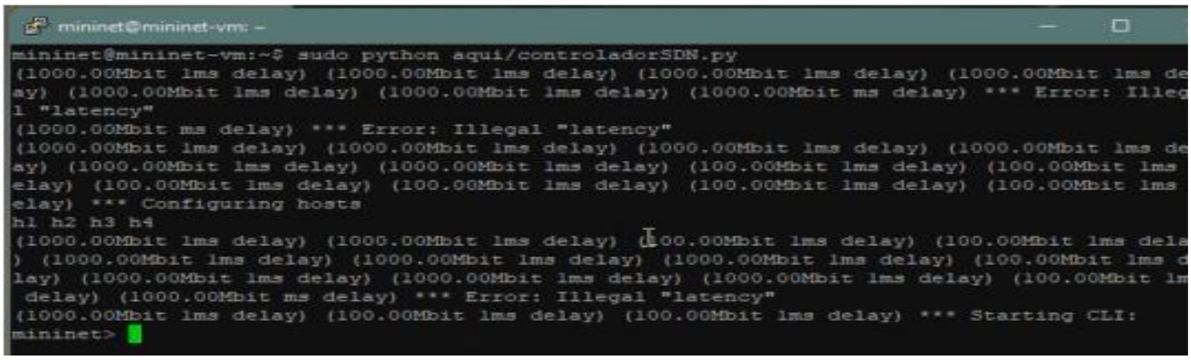


```
mininet@mininet-vm:~$ sudo python aqui/controladorSDN.py +
```

Figura 47 Comando de iniciación del controlador SDN

Fuente: Elaboración propia

Inmediatamente se produce la creación y la configuración tanto de los hosts desde el Host 1 hasta el Host 4 (h1 – h4) que son parte de la red que se analiza, así como de los switches, de ser el caso como se visualiza en la Figura 48:

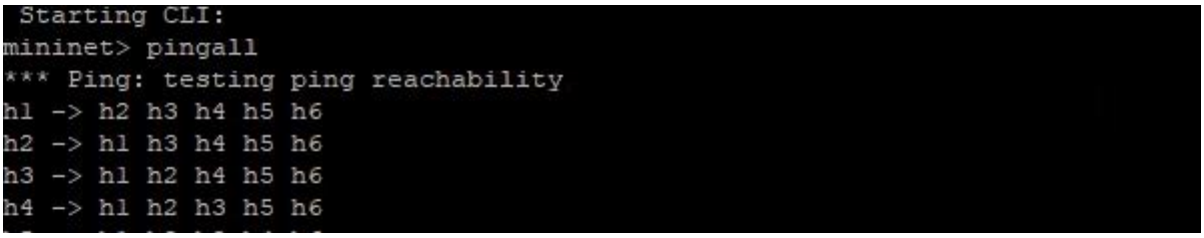


```
mininet@mininet-vm:~$ sudo python aqui/controladorSDN.py
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms de
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit ms delay) *** Error: Illeg
l "latency"
(1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms de
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms
elay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms
elay) *** Configuring hosts
h1 h2 h3 h4
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms dela
) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms d
lay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1m
 delay) (1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) *** Starting CLI:
mininet>
```

Figura 48 Compilación inicial del controlador SDN

Fuente: Elaboración propia

Al mismo tiempo se verifican los links que se han establecido tanto para los hosts como para los switches que forman parte de la red, utilizando el comendo ping para obtener respuesta desde todos estos elementos de red como se aprecia en la Figura 49:

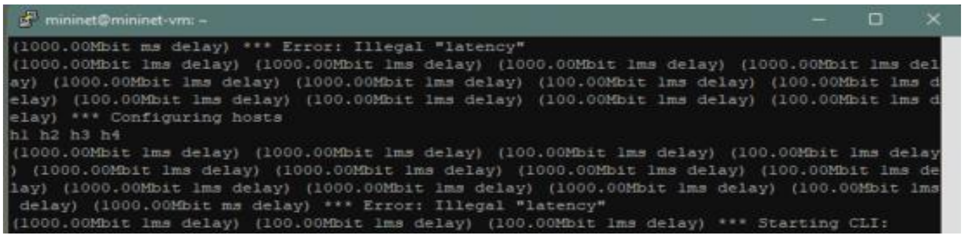


```
 Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6
h2 -> h1 h3 h4 h5 h6
h3 -> h1 h2 h4 h5 h6
h4 -> h1 h2 h3 h5 h6
```

Figura 49 Comando de testeo hacia los hosts

Fuente: Elaboración propia

Una vez recibida respuesta de que existe enlace y comunicación entre los hosts y switches, se realiza también la verificación del ancho de banda disponible entre cada uno de los hosts (ver Figura 50), esto con el objeto de ir definiendo cual es la ruta menos congestionada:



```
(1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) *** Configuring hosts
h1 h2 h3 h4
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay
) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms de
lay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms
 delay) (1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) *** Starting CLI:
```

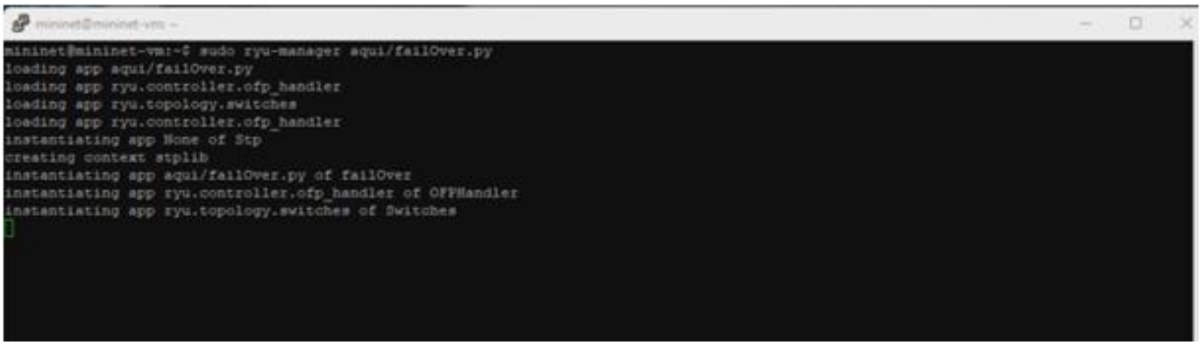
Figura 50 Respuesta del test hacia los hosts

Fuente: Elaboración propia

4.3.2 Compilación del balanceador de carga

A continuación, se visualizan algunas pantallas de la compilación del algoritmo del balanceador de cargas propuesto:

Primeramente, se produce la carga y la inicialización de todas las librerías mínimas requeridas para una buena operación del algoritmo del balanceador propuesto bajo el controlador RYU, como se observa en la Figura 51:

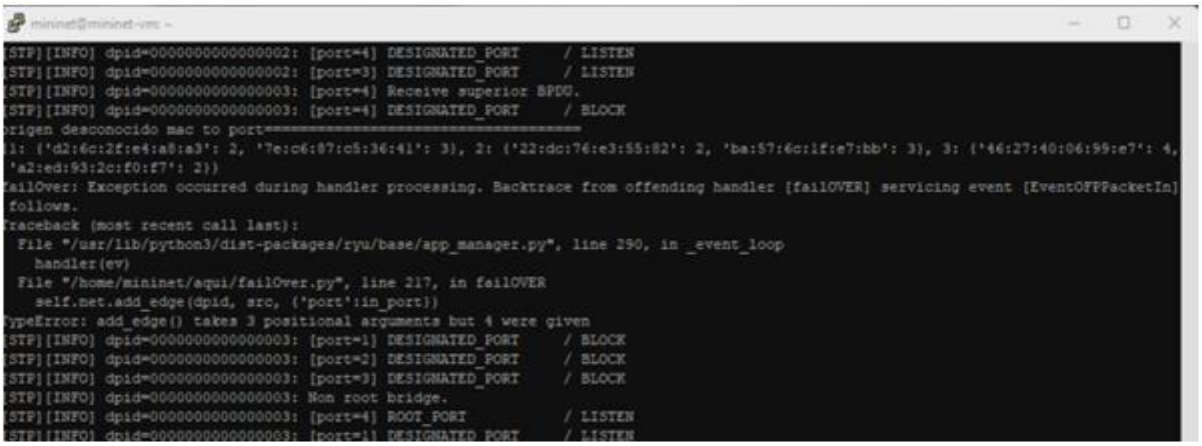


```
mininet@mininet-vm:~$ sudo ryu-manager aqui/failOver.py
loading app aqui/failOver.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of Stp
creating context stplib
instantiating app aqui/failOver.py of failOver
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
```

Figura 51 Comando de llamado del algoritmo failover, carga e iniciación de las librerías

Fuente: Elaboración propia

A continuación, se visualizan la pantalla de la Figura 52, de la compilación una vez realizada la creación de la clase failover, y la inicialización de los protocolos, constantes y variables que han sido previamente:

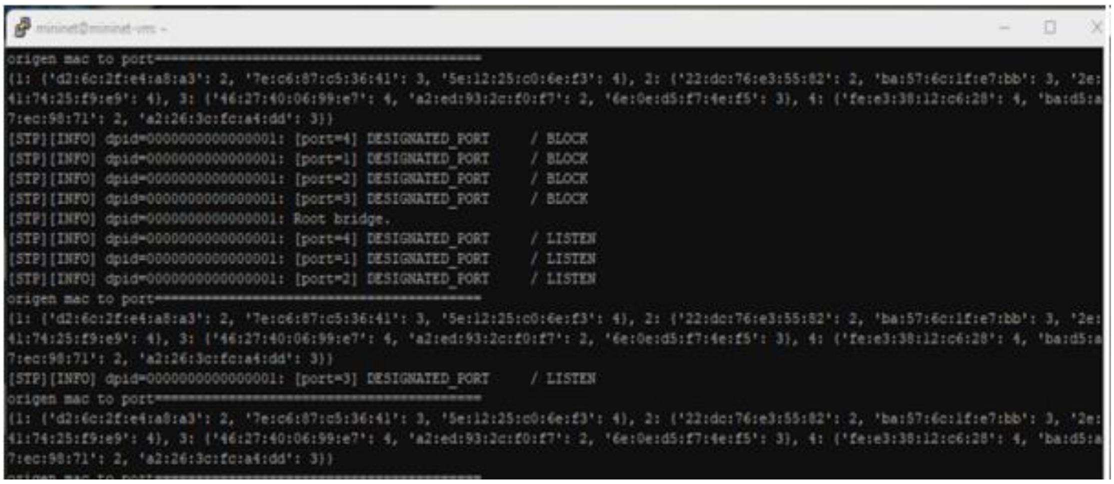


```
[STP][INFO] dpid=0000000000000002: [port=4] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000003: [port=4] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=4] DESIGNATED_PORT     / BLOCK
origen desconocido mac to port==================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3}, 3: {'46:27:40:06:99:e7': 4,
 'a2:ed:93:2c:f0:f7': 2}}
failOver: Exception occurred during handler processing. Backtrace from offending handler [failOVER] servicing event [EventOFPPacketIn]
 follows.
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/ryu/base/app_manager.py", line 290, in _event_loop
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 217, in failOVER
    self.net.add_edge(dpid, src, {'port':in_port})
TypeError: add_edge() takes 3 positional arguments but 4 were given
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000003: Non root bridge.
[STP][INFO] dpid=0000000000000003: [port=4] ROOT_PORT           / LISTEN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT     / LISTEN
```

Figura 52 Datos de iniciación del algoritmo failover

Fuente: Elaboración propia

En la siguiente ventana, se visualiza un extracto de la compilación y configuración de cada uno de los puertos, puesto que el mismo proceso se realiza de manera repetitiva, de igual manera se visualiza la validación del estado de la ruta desde el origen hasta el destino, repetitivamente hasta que se alcance el estado real de las rutas (ver Figura 53).

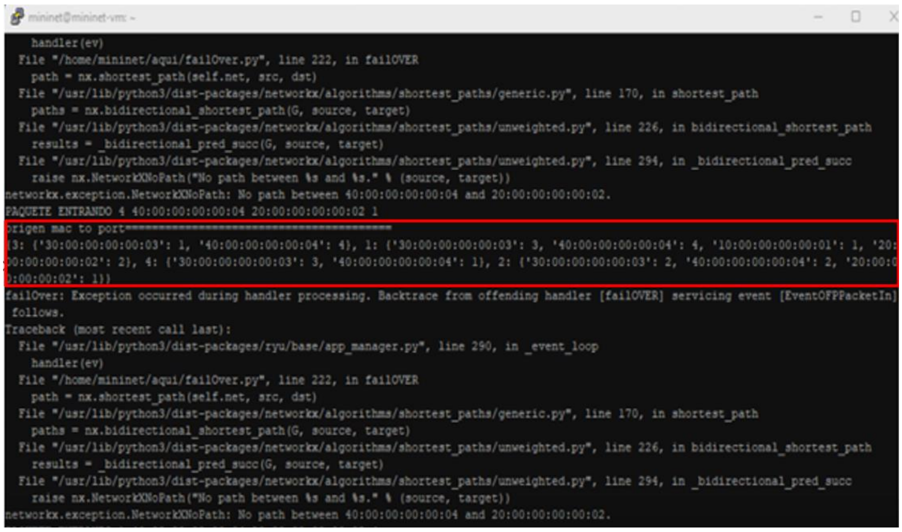


```
origen mac to port========================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
[STP][INFO] dpid=0000000000000001: [port=4] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000001: Root bridge.
[STP][INFO] dpid=0000000000000001: [port=4] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT     / LISTEN
origen mac to port========================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT     / LISTEN
origen mac to port========================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port========================================
```

Figura 53 Estado de la ruta desde origen al destino

Fuente: Elaboración propia

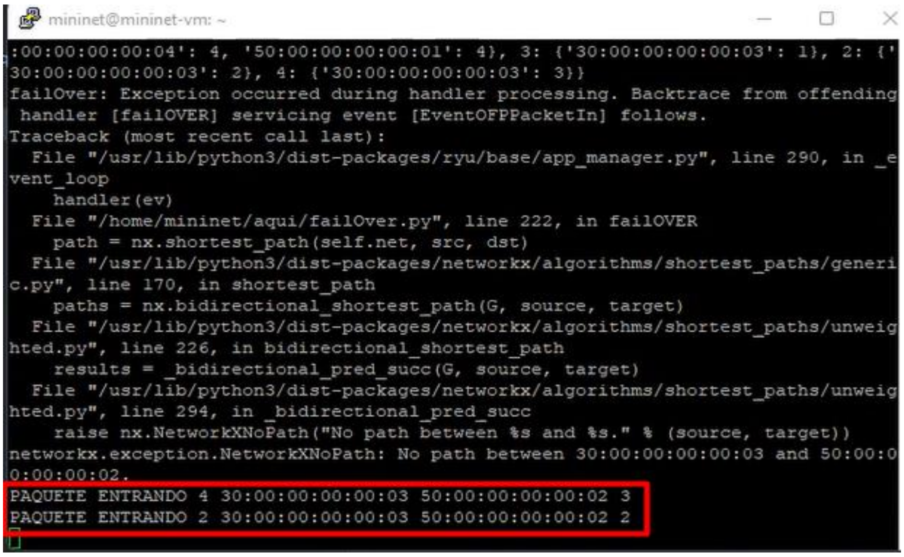
Posteriormente, se inicia el envío de paquetes llamando al algoritmo creado con la clase failover, de este modo se van enviando una serie de paquetes sobre los cuales el algoritmo propuesto procede a ir probando la mejor ruta, en las pantallas siguientes extraídas durante la compilación del algoritmo se visualizan por un lado la verificación del ingreso de paquetes en la Figura 54, y la confirmación del ingreso de paquetes en la Figura 55:



```
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
    path = nx.shortest_path(self.net, src, dst)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/generic.py", line 170, in shortest_path
    paths = nx.bidirectional_shortest_path(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 226, in bidirectional_shortest_path
    results = _bidirectional_pred_succ(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 294, in _bidirectional_pred_succ
    raise nx.NetworkXNoPath("No path between %s and %s." % (source, target))
networkx.exception.NetworkXNoPath: No path between 40:00:00:00:00:04 and 20:00:00:00:00:02.
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
rigen mac to port========================================
{3: {'30:00:00:00:00:03': 1, '40:00:00:00:00:04': 4}, 1: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 4, '10:00:00:00:00:01': 1, '20:
0:00:00:00:02': 2}, 4: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 1}, 2: {'30:00:00:00:00:03': 2, '40:00:00:00:00:04': 2, '20:00:0
:00:00:02': 1}}
failOver: Exception occurred during handler processing. Backtrace from offending handler [failOVER] servicing event [EventOFPPacketIn]
 follows.
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/ryu/base/app_manager.py", line 290, in _event_loop
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
    path = nx.shortest_path(self.net, src, dst)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/generic.py", line 170, in shortest_path
    paths = nx.bidirectional_shortest_path(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 226, in bidirectional_shortest_path
    results = _bidirectional_pred_succ(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 294, in _bidirectional_pred_succ
    raise nx.NetworkXNoPath("No path between %s and %s." % (source, target))
networkx.exception.NetworkXNoPath: No path between 40:00:00:00:00:04 and 20:00:00:00:00:02.
```

Figura 54 Verificación de ingreso de paquetes

Fuente: Elaboración propia

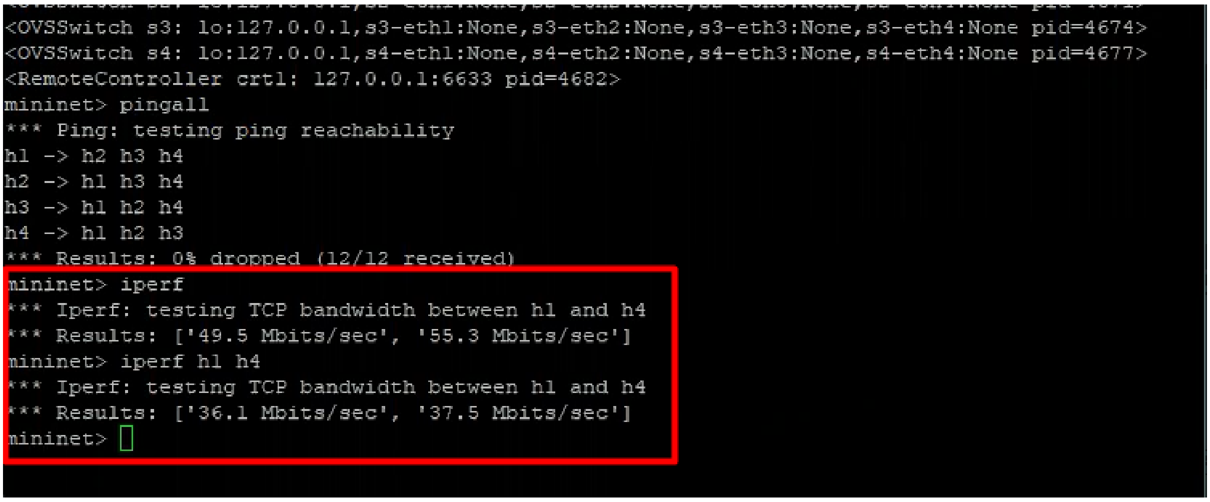


```
:00:00:00:00:04': 4, '50:00:00:00:00:01': 4}, 3: {'30:00:00:00:00:03': 1}, 2: {'
30:00:00:00:00:03': 2}, 4: {'30:00:00:00:00:03': 3}}
failOver: Exception occurred during handler processing. Backtrace from offending
 handler [failOVER] servicing event [EventOFPPacketIn] follows.
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/ryu/base/app_manager.py", line 290, in _e
vent_loop
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
    path = nx.shortest_path(self.net, src, dst)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/generi
c.py", line 170, in shortest_path
    paths = nx.bidirectional_shortest_path(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweig
hted.py", line 226, in bidirectional_shortest_path
    results = _bidirectional_pred_succ(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweig
hted.py", line 294, in _bidirectional_pred_succ
    raise nx.NetworkXNoPath("No path between %s and %s." % (source, target))
networkx.exception.NetworkXNoPath: No path between 30:00:00:00:00:03 and 50:00:0
0:00:00:02.
PAQUETE ENTRANDO 4 30:00:00:00:00:03 50:00:00:00:00:02 3
PAQUETE ENTRANDO 2 30:00:00:00:00:03 50:00:00:00:00:02 2
```

Figura 55 Confirmación de ingreso de paquetes

Fuente: Elaboración propia

Finalmente, en base al estado del ancho de banda disponible en cada una de las rutas desde el origen al destino, una vez aplicado el algoritmo propuesto, se realiza la selección y confirmación para el nuevo enrutamiento de paquetes a través de la ruta más óptima disponible (ver Figura 56):



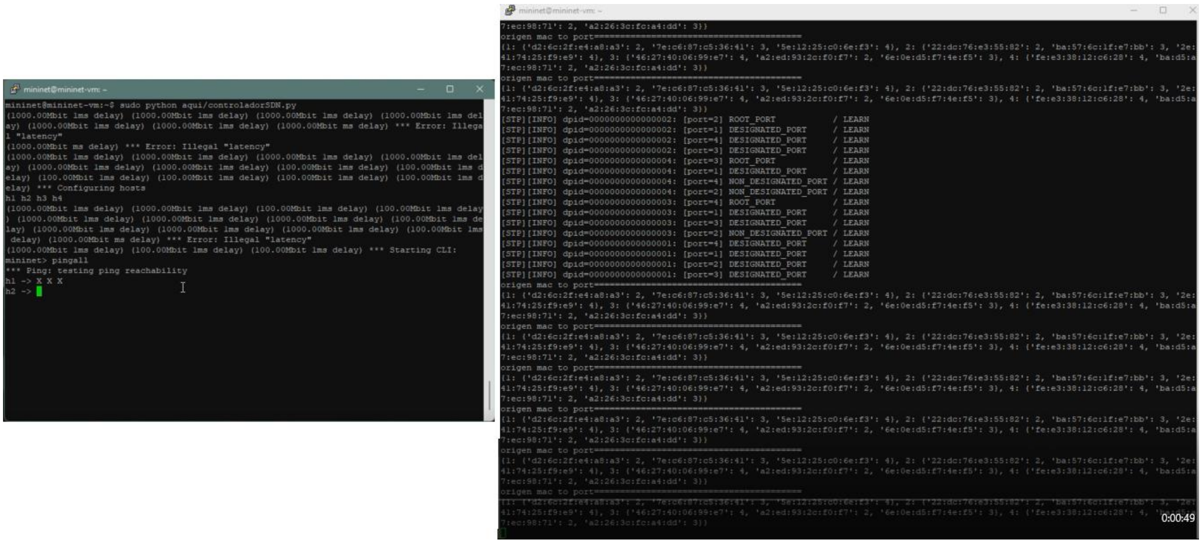
```
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None,s3-eth4:None pid=4674>
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None,s4-eth4:None pid=4677>
<RemoteController crtl: 127.0.0.1:6633 pid=4682>
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['49.5 Mbits/sec', '55.3 Mbits/sec']
mininet> iperf h1 h4
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['36.1 Mbits/sec', '37.5 Mbits/sec']
mininet> 
```

Figura 56 Confirmación de nuevo enrutamiento de paquetes

Fuente: Elaboración propia

4.4 Ejecución y verificación del algoritmo sobre el controlador SDN.

A continuación, se muestran las pantallas de ejecución del algoritmo del balanceador de cargas sobre el controlador SDN, básicamente lo que se quiere resaltar es el hecho de cómo se va sucediendo la compilación al mismo tiempo tanto con la ejecución del controlador RYU, así como la ejecución del algoritmo balanceador de carga propuesto, como se presentan en las Figuras 57 a la Figura 57:



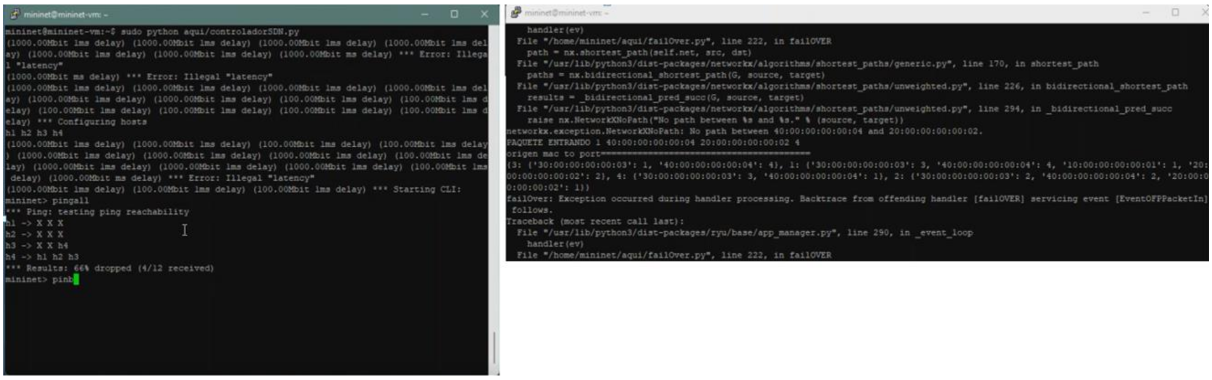
```
mininet@mininet-vm:~$ sudo python aqui/controladorSDN.py
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit ms delay) *** Error: Illega
l "latency"
(1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) *** Configuring hosts
h1 h2 h3 h4
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay
) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms de
lay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms
 delay) (1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) *** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> 
```

```
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT          / LEARN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000002: [port=4] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000004: [port=3] ROOT_PORT          / LEARN
[STP][INFO] dpid=0000000000000004: [port=1] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000004: [port=4] NON_DESIGNATED_PORT / LEARN
[STP][INFO] dpid=0000000000000004: [port=2] NON_DESIGNATED_PORT / LEARN
[STP][INFO] dpid=0000000000000003: [port=4] ROOT_PORT          / LEARN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / LEARN
[STP][INFO] dpid=0000000000000001: [port=4] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT    / LEARN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT    / LEARN
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
origen mac to port================================================
{1: {'d2:6c:2f:e4:a8:a3': 2, '7e:c6:87:c5:36:41': 3, '5e:12:25:c0:6e:f3': 4}, 2: {'22:dc:76:e3:55:82': 2, 'ba:57:6c:1f:e7:bb': 3, '2e:
41:74:25:f9:e9': 4}, 3: {'46:27:40:06:99:e7': 4, 'a2:ed:93:2c:f0:f7': 2, '6e:0e:d5:f7:4e:f5': 3}, 4: {'fe:e3:38:12:c6:28': 4, 'ba:d5:a
7:ec:98:71': 2, 'a2:26:3c:fc:a4:dd': 3}}
```

(a) (b)

Figura 57 (a) Ejecución sobre el controlador, (b) Inicio del algoritmo balanceador

Fuente: Elaboración propia



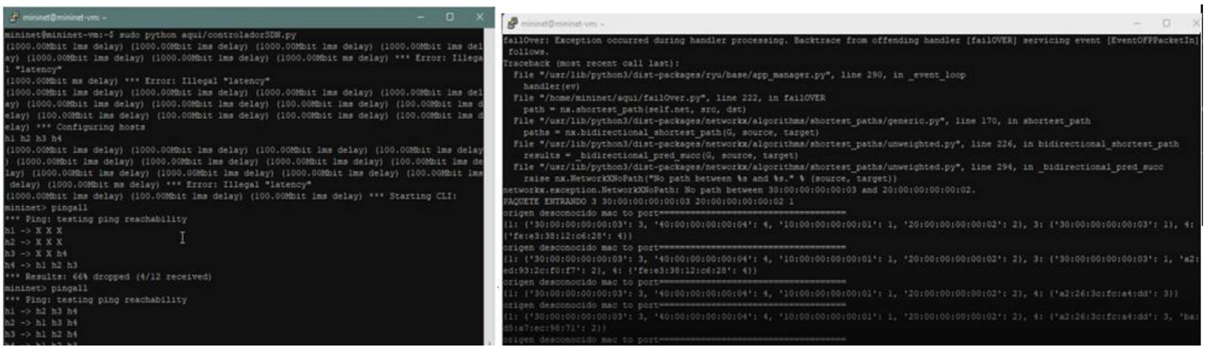
```
mininet@mininet-vm:~$ sudo python aqui/controladorSDN.py
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit ms delay) *** Error: Illega
l "latency"
(1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) *** Configuring hosts
h1 h2 h3 h4
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay
) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms de
lay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms
 delay) (1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) *** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X h4
h4 -> h1 h2 h3
*** Results: 66% dropped (4/12 received)
mininet> pinb
```

```
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
    path = nx.shortest_path(self.net, src, dst)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/generic.py", line 170, in shortest_path
    paths = nx.bidirectional_shortest_path(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 226, in bidirectional_shortest_path
    results = _bidirectional_pred_succ(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 294, in _bidirectional_pred_succ
    raise nx.NetworkXNoPath("No path between %s and %s." % (source, target))
networkx.exception.NetworkXNoPath: No path between 40:00:00:00:00:04 and 20:00:00:00:00:02.
PAQUETE ENTRANDO 1 40:00:00:00:00:04 20:00:00:00:00:02 4
origen mac to port================================================
{3: {'30:00:00:00:00:03': 1, '40:00:00:00:00:04': 4}, 1: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 4, '10:00:00:00:00:01': 1, '20:
00:00:00:00:02': 2}, 4: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 1}, 2: {'30:00:00:00:00:03': 2, '40:00:00:00:00:04': 2, '20:00:0
0:00:00:02': 1}}
failOver: Exception occurred during handler processing. Backtrace from offending handler [failOVER] servicing event [EventOFPPacketIn]
 follows.
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/ryu/base/app_manager.py", line 290, in _event_loop
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
```

(a) (b)

Figura 58 (a) Testeo ping sobre el controlador, (b) Verificación de ingreso de paquetes

Fuente: Elaboración propia



(a)

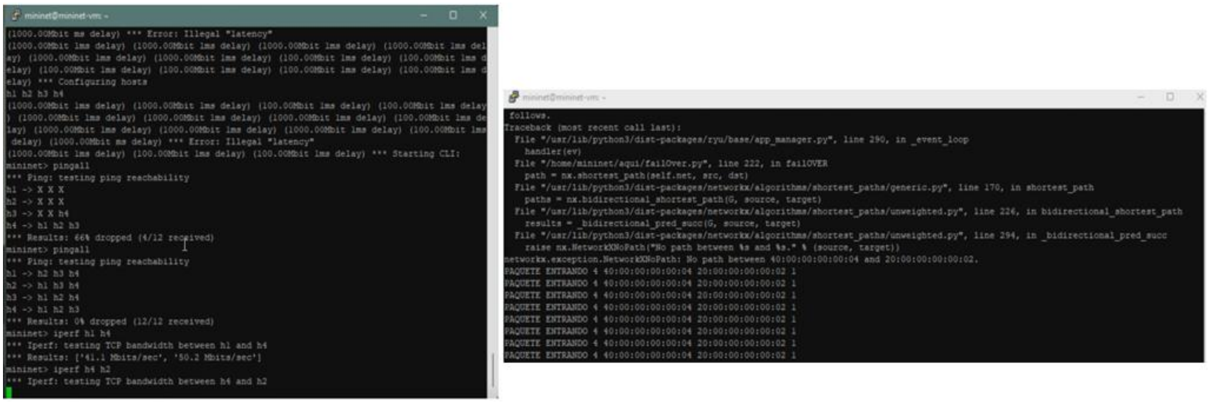
```
mininet@mininet-vm:~$ sudo python aqui/controladorSDN.py
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit ms delay) *** Error: Illega
l "latency"
(1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) *** Configuring hosts
h1 h2 h3 h4
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay
) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms de
lay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms
 delay) (1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) *** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X h4
h4 -> h1 h2 h3
*** Results: 66% dropped (4/12 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
```

(b)

```
failOver: Exception occurred during handler processing. Backtrace from offending handler [failOVER] servicing event [EventOFPPacketIn]
 follows.
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/ryu/base/app_manager.py", line 290, in _event_loop
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
    path = nx.shortest_path(self.net, src, dst)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/generic.py", line 170, in shortest_path
    paths = nx.bidirectional_shortest_path(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 226, in bidirectional_shortest_path
    results = _bidirectional_pred_succ(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 294, in _bidirectional_pred_succ
    raise nx.NetworkXNoPath("No path between %s and %s." % (source, target))
networkx.exception.NetworkXNoPath: No path between 30:00:00:00:00:03 and 20:00:00:00:00:02.
PAQUETE ENTRANDO 3 30:00:00:00:00:03 20:00:00:00:00:02 1
origen desconocido mac to port====================================
{1: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 4, '10:00:00:00:00:01': 1, '20:00:00:00:00:02': 2}, 3: {'30:00:00:00:00:03': 1}, 4:
{'fe:e3:38:12:c6:28': 4}}
origen desconocido mac to port====================================
{1: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 4, '10:00:00:00:00:01': 1, '20:00:00:00:00:02': 2}, 3: {'30:00:00:00:00:03': 1, 'a2:
ed:93:2c:f0:f7': 2}, 4: {'fe:e3:38:12:c6:28': 4}}
origen desconocido mac to port====================================
{1: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 4, '10:00:00:00:00:01': 1, '20:00:00:00:00:02': 2}, 4: {'a2:26:3c:fc:a4:dd': 3}}
origen desconocido mac to port====================================
{1: {'30:00:00:00:00:03': 3, '40:00:00:00:00:04': 4, '10:00:00:00:00:01': 1, '20:00:00:00:00:02': 2}, 4: {'a2:26:3c:fc:a4:dd': 3, 'ba:
d5:a7:ec:98:71': 2}}
origen desconocido mac to port====================================
```

Figura 59 (a) Testeo ping hacia los hosts, (b) Verificación de ingreso de paquetes a los hosts

Fuente: Elaboración propia



(a)

```
(1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms del
ay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms d
elay) *** Configuring hosts
h1 h2 h3 h4
(1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay
) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms de
lay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (1000.00Mbit 1ms delay) (100.00Mbit 1ms
 delay) (1000.00Mbit ms delay) *** Error: Illegal "latency"
(1000.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) *** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X h4
h4 -> h1 h2 h3
*** Results: 66% dropped (4/12 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> iperf h1 h4
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['41.1 Mbits/sec', '50.2 Mbits/sec']
mininet> iperf h4 h2
*** Iperf: testing TCP bandwidth between h4 and h2
```

(b)

```
 follows.
Traceback (most recent call last):
  File "/usr/lib/python3/dist-packages/ryu/base/app_manager.py", line 290, in _event_loop
    handler(ev)
  File "/home/mininet/aqui/failOver.py", line 222, in failOVER
    path = nx.shortest_path(self.net, src, dst)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/generic.py", line 170, in shortest_path
    paths = nx.bidirectional_shortest_path(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 226, in bidirectional_shortest_path
    results = _bidirectional_pred_succ(G, source, target)
  File "/usr/lib/python3/dist-packages/networkx/algorithms/shortest_paths/unweighted.py", line 294, in _bidirectional_pred_succ
    raise nx.NetworkXNoPath("No path between %s and %s." % (source, target))
networkx.exception.NetworkXNoPath: No path between 40:00:00:00:00:04 and 20:00:00:00:00:02.
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
PAQUETE ENTRANDO 4 40:00:00:00:00:04 20:00:00:00:00:02 1
```

Figura 60 (a) Respuesta del test hacia los hosts, (b) Confirmación de ingreso de paquetes

Fuente: Elaboración propia

Capítulo 5. Pruebas de Funcionamiento y Medición de rendimiento

En el presente capítulo, se detalla las pruebas realizadas en los testbed aplicando el algoritmo desarrollado para la selección y enrutamiento de paquetes dentro de la red SDN. Como primer punto se tratará el tema referente a la selección de rutas hacia los destinos y como el algoritmo interactúa con la red. Posteriormente se realiza un análisis de los parámetros de rendimiento propuestos: jitter, latencia, pérdida de paquetes, anchos de banda y tasas de transferencia de los enlaces de ambos escenarios de estudio para finalmente realizar una comparativa entre los resultados obtenidos anteriormente con los resultados aplicando el algoritmo desarrollado.

5.1 Determinación de rutas hacia los destinos.

En este apartado se presenta como el algoritmo desarrollado realiza el estudio para la determinación de rutas hacia el destino. Para este caso se realiza la documentación del primer testbed compuesto por 5 OpenvSwitch y 5 hosts y el controlador RYU, como se observa en el Figura 61.

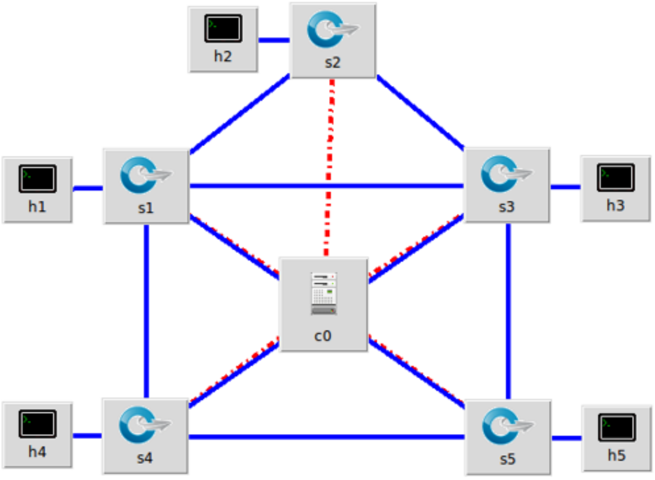


Figura 61 Topología testbed 1 SDN

Fuente: Elaboración propia

Para este caso en particular se procede a realizar el análisis de enrutamiento de paquetes, considerando cada ruta posible desde el host h1 al host h3. En este sentido se configura la red SDN de Mininet con anchos de banda determinados como se presenta en la siguiente Tabla 7 para el estudio de como interactuaría el algoritmo ya en ejecución.

Tabla 6 Anchos de banda y retardo de enlaces del h1 al h3

Enlace	Ancho de Banda (Mbits)	Jitter (ms)
S1-S3	800	10
S1-S5	700	20
S5-S3	700	20
S1-S2	600	50
S2-S3	600	50
S1-S4	500	70
S4-S3	500	70
S4-S5	300	100

Fuente: Elaboración propia

Considerando los valores presentados anteriormente se determina que las rutas a trazarse dentro de la red SDN del host h1 al host h3 son las que se muestran en la siguiente Tabla 8.

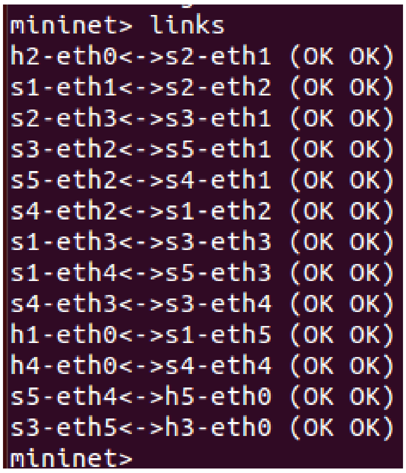
Tabla 7 Tabla de prioridad de enrutamiento

Enlace	Prioridad
h1-S1-S3-h3	Primera
h1-S1-S5-S3-h3	Segunda
h1-S1-S2-S3-h3	Tercera
h1-S1-S4-S3-h3	Cuarta
h1-S1-S4-S5-S3-h3	Quinta

Fuente: Elaboración propia

Una vez establecido las rutas teóricamente, se presenta los resultados de enrutamiento de los paquetes dentro de la red SDN emulada en Mininet y en controlador SDN RYU. En la

Figura 62 se presentan los enlaces establecidos con el levantamiento de la red, siendo s la nomenclatura del switch, eth el puerto de conexión y h el host.



```
mininet> links
h2-eth0<->s2-eth1 (OK OK)
s1-eth1<->s2-eth2 (OK OK)
s2-eth3<->s3-eth1 (OK OK)
s3-eth2<->s5-eth1 (OK OK)
s5-eth2<->s4-eth1 (OK OK)
s4-eth2<->s1-eth2 (OK OK)
s1-eth3<->s3-eth3 (OK OK)
s1-eth4<->s5-eth3 (OK OK)
s4-eth3<->s3-eth4 (OK OK)
h1-eth0<->s1-eth5 (OK OK)
h4-eth0<->s4-eth4 (OK OK)
s5-eth4<->h5-eth0 (OK OK)
s3-eth5<->h3-eth0 (OK OK)
mininet>
```

Figura 62 Enlaces levantados por el emulador Mininet

Fuente: Elaboración propia

Se procede a simular tráfico mediante la herramienta de iperf desde el host h1 al host h3 para determina la ruta primera tomada por el algoritmo, esperando obtener como resultado la ruta h1-S1-S3-h3 como se observa en la Figura 63.

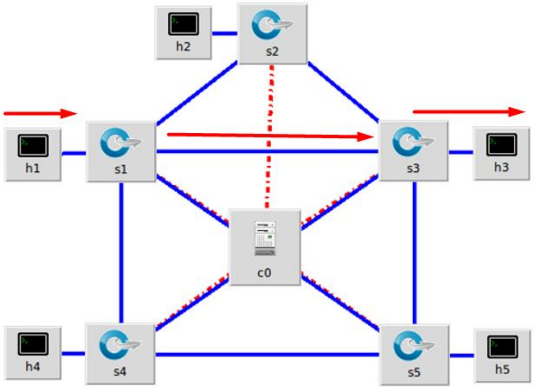
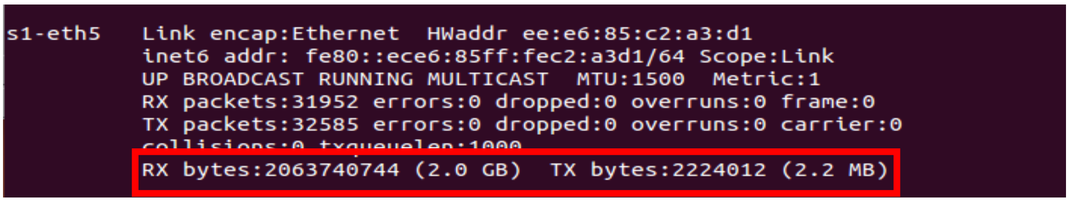


Figura 63 Primera ruta h1-s1-s3-h3

Fuente: Elaboración propia

En la Figura 63, se presenta la interfaz s1-eth5 que existe tráfico existente de 2.0GB que se encuentra conectado al h1-eth0 según los enlaces presentados en la Figura 64.

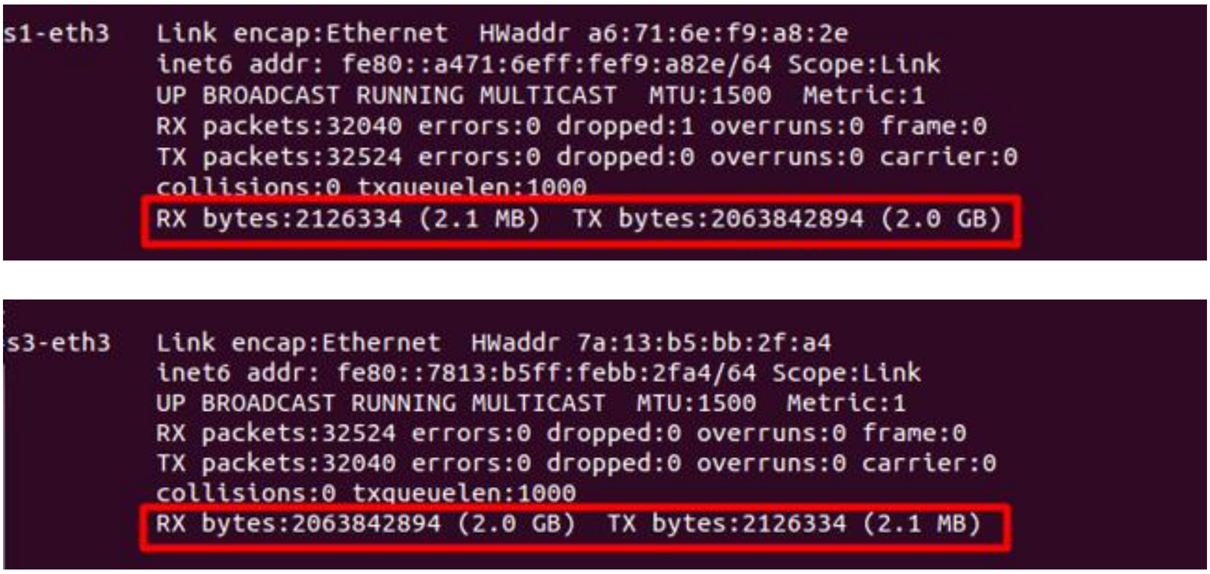


```
s1-eth5   Link encap:Ethernet  HWaddr ee:e6:85:c2:a3:d1
          inet6 addr: fe80::ece6:85ff:fec2:a3d1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31952 errors:0 dropped:0 overruns:0 frame:0
          TX packets:32585 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2063740744 (2.0 GB)  TX bytes:2224012 (2.2 MB)
```

Figura 64 Tráfico de h1 a s1

Fuente: Elaboración propia

Posteriormente dicho tráfico se conmuta por el puerto s1-eth3 misma que se encuentra conectado el enlace hacia el switch s3-eth3 como se puede observar en las imágenes de la Figura 65



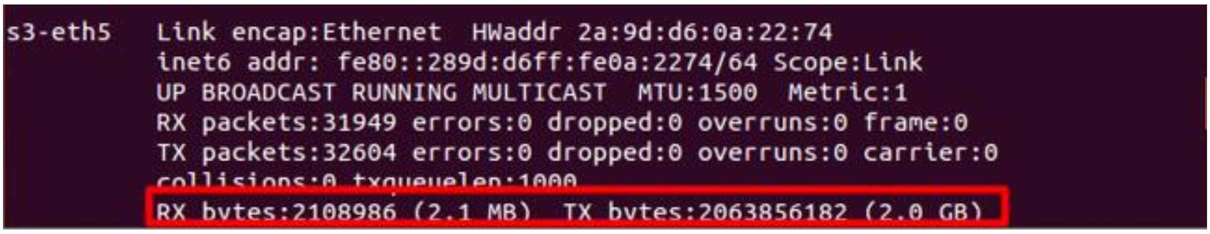
```
s1-eth3   Link encap:Ethernet  HWaddr a6:71:6e:f9:a8:2e
          inet6 addr: fe80::a471:6eff:fef9:a82e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:32040 errors:0 dropped:1 overruns:0 frame:0
          TX packets:32524 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2126334 (2.1 MB)  TX bytes:2063842894 (2.0 GB)
```

```
s3-eth3   Link encap:Ethernet  HWaddr 7a:13:b5:bb:2f:a4
          inet6 addr: fe80::7813:b5ff:febb:2fa4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:32524 errors:0 dropped:0 overruns:0 frame:0
          TX packets:32040 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2063842894 (2.0 GB)  TX bytes:2126334 (2.1 MB)
```

Figura 65 Tráfico presentado de s1 a s3

Fuente: Elaboración propia

Finalmente llega a su destinatario a través del puerto s3-eht5 (ver Figura 66) que se encuentra conectado hacia el host 3 mediante el puerto h3-eh0



```
s3-eth5   Link encap:Ethernet  HWaddr 2a:9d:d6:0a:22:74
          inet6 addr: fe80::289d:d6ff:fe0a:2274/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31949 errors:0 dropped:0 overruns:0 frame:0
          TX packets:32604 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2108986 (2.1 MB)  TX bytes:2063856182 (2.0 GB)
```

Figura 66 Tráfico presentado de s3 a h3

Fuente: Elaboración propia

Se presenta la topología eliminando el enlace directo hacia el destino s1-s3 y verificar como el algoritmo en baje al ancho de banda y el jitter procede a determinar la siguiente ruta **h1-s1-s2-s3-h3,** como se presenta en la siguiente Figura 67.

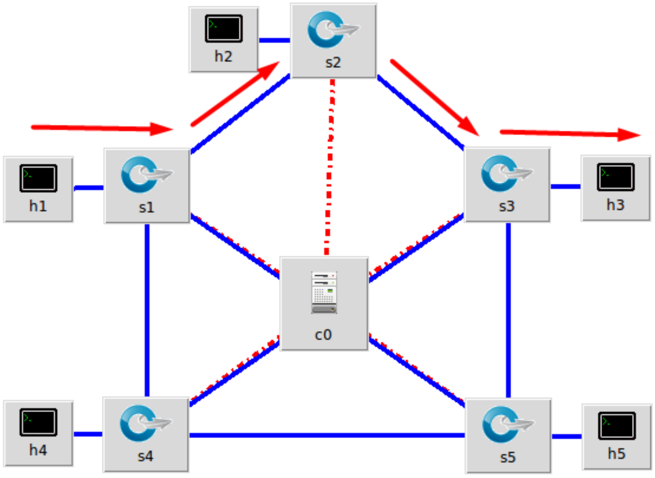
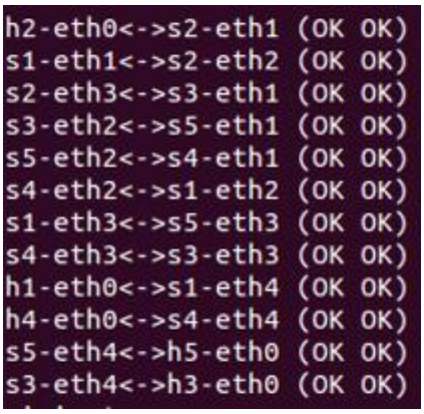


Figura 67 Segunda ruta h1-s1-s2-s3-h3

Fuente: Elaboración propia

La Figura 68, se presenta los nuevos enlaces de la red simulada en Mininet eliminando la conexión de s1 a s3.

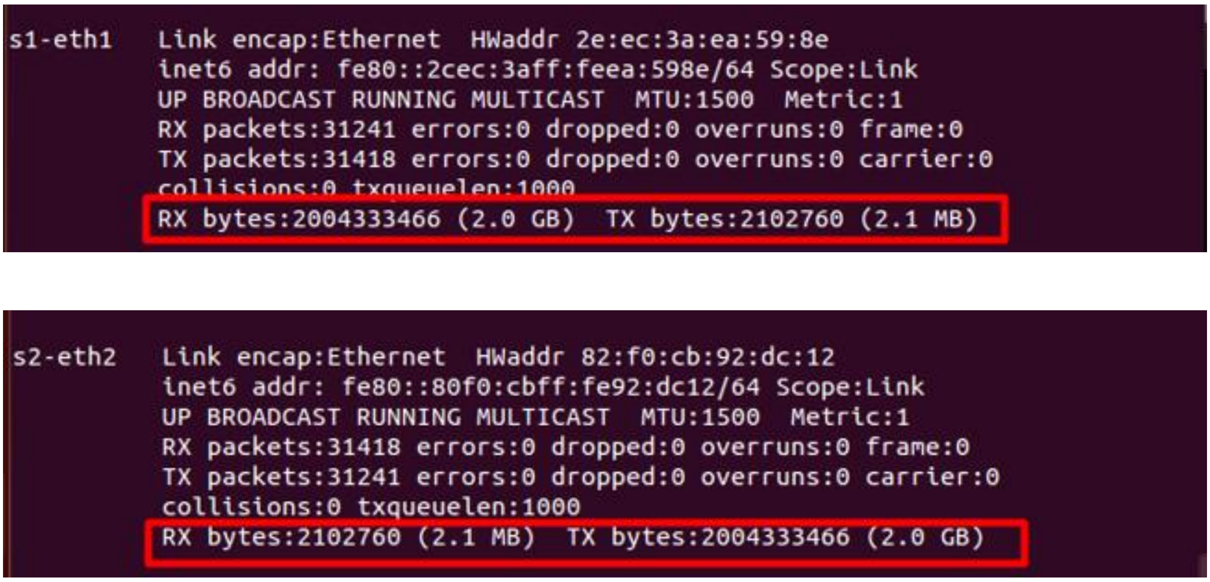


```
h2-eth0<->s2-eth1 (OK OK)
s1-eth1<->s2-eth2 (OK OK)
s2-eth3<->s3-eth1 (OK OK)
s3-eth2<->s5-eth1 (OK OK)
s5-eth2<->s4-eth1 (OK OK)
s4-eth2<->s1-eth2 (OK OK)
s1-eth3<->s5-eth3 (OK OK)
s4-eth3<->s3-eth3 (OK OK)
h1-eth0<->s1-eth4 (OK OK)
h4-eth0<->s4-eth4 (OK OK)
s5-eth4<->h5-eth0 (OK OK)
s3-eth4<->h3-eth0 (OK OK)
```

Figura 68 Enlaces levantados por el emulador Mininet sin s1 a s3

Fuente: Elaboración propia

Se verifica el tráfico presentado en el s1, como se presenta en la Figura 69, donde s1-eht1 se encuentra conectado hacia s2-eth2 y presenta un tráfico de 2.0GB



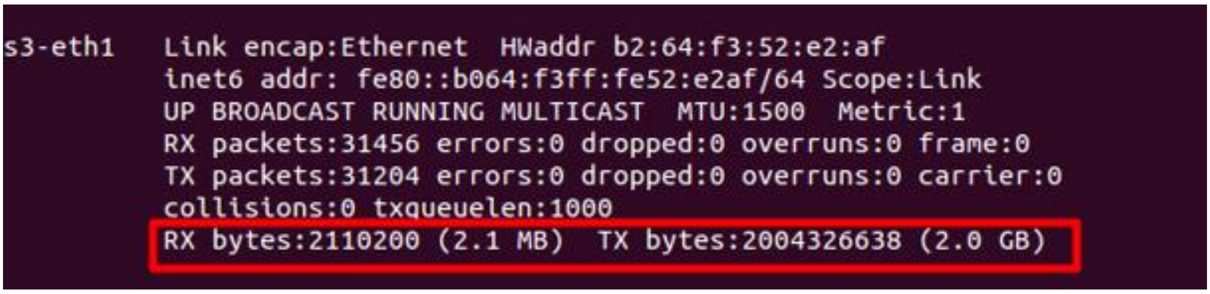
```
s1-eth1   Link encap:Ethernet  HWaddr 2e:ec:3a:ea:59:8e
          inet6 addr: fe80::2cec:3aff:feea:598e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31241 errors:0 dropped:0 overruns:0 frame:0
          TX packets:31418 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2004333466 (2.0 GB)  TX bytes:2102760 (2.1 MB)
```

```
s2-eth2   Link encap:Ethernet  HWaddr 82:f0:cb:92:dc:12
          inet6 addr: fe80::80f0:cbff:fe92:dc12/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31418 errors:0 dropped:0 overruns:0 frame:0
          TX packets:31241 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2102760 (2.1 MB)  TX bytes:2004333466 (2.0 GB)
```

Figura 69 Tráfico presentado en s1-eth1 y s2-eth2

Fuente: Elaboración propia

Seguido del salto de s2 hacia s3 mediante la interfaz s2-eth3 hacia s3-eth1 para poder llegar hacia el destino h3 como se aprecia en la Figura 70.



```
s3-eth1   Link encap:Ethernet  HWaddr b2:64:f3:52:e2:af
          inet6 addr: fe80::b064:f3ff:fe52:e2af/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31456 errors:0 dropped:0 overruns:0 frame:0
          TX packets:31204 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2110200 (2.1 MB)  TX bytes:2004326638 (2.0 GB)
```

Figura 70 Tráfico de s3 hacia s2

Fuente: Elaboración propia

Para la siguiente ruta se presenta en la Figura 71 donde el tramo es desde h1-s1-s5-s3-h3 debido a que es la tercera ruta con prioridad como se presenta en la Tabla 8

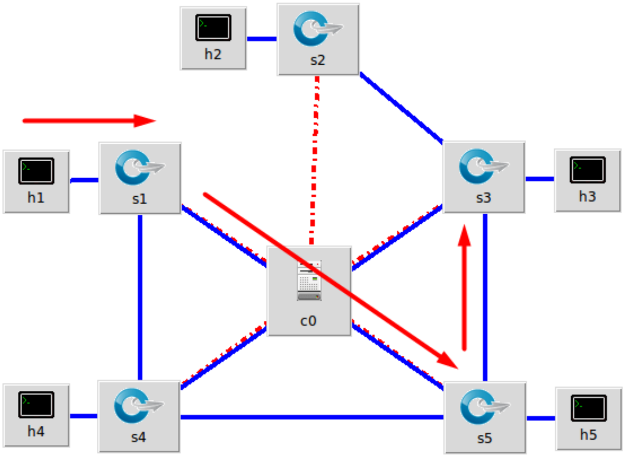
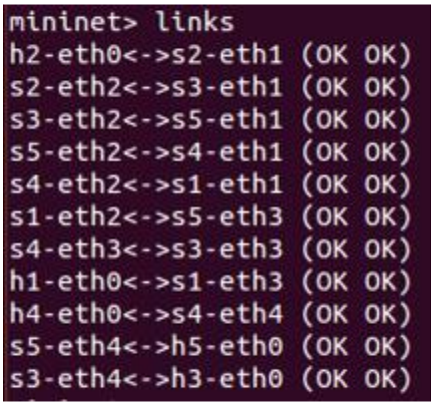


Figura 71 Tercera ruta h1-s1-s5-s3-h3

Fuente: Elaboración propia

De la misma forma mediante la herramienta de iperf realizamos un testeo de ancho de banda desde el host h1 hacia h3 para verificar mediante ipconfig las el tráfico presente en las interfaces de los switches y determinar la ruta. Como primera captura de tráfico se presenta el listado de interfaces levantadas en Mininet ver Figura 72

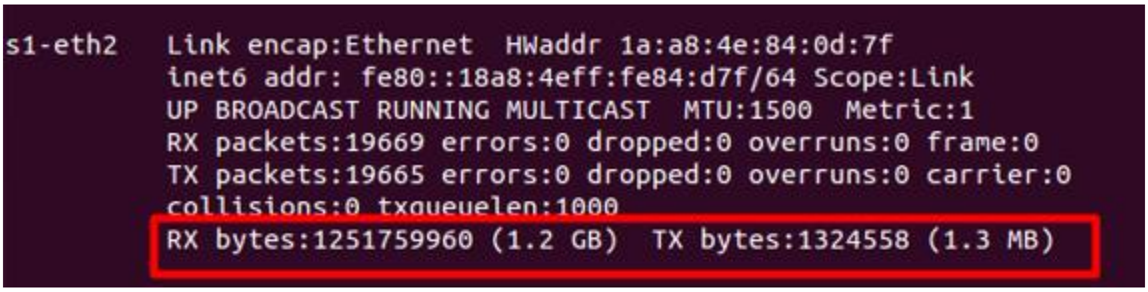


```
mininet> links
h2-eth0<->s2-eth1 (OK OK)
s2-eth2<->s3-eth1 (OK OK)
s3-eth2<->s5-eth1 (OK OK)
s5-eth2<->s4-eth1 (OK OK)
s4-eth2<->s1-eth1 (OK OK)
s1-eth2<->s5-eth3 (OK OK)
s4-eth3<->s3-eth3 (OK OK)
h1-eth0<->s1-eth3 (OK OK)
h4-eth0<->s4-eth4 (OK OK)
s5-eth4<->h5-eth0 (OK OK)
s3-eth4<->h3-eth0 (OK OK)
```

Figura 72 Enlaces levantados por el emulador Mininet sin s1-s3 y s1-s2

Fuente: Elaboración propia

Se procede realizar la captura del tráfico presenta en la interfaz del switch s1 por eth2 enlace conectado al switch s5 por la interfaz eth3 como se presenta en la Figura 73.

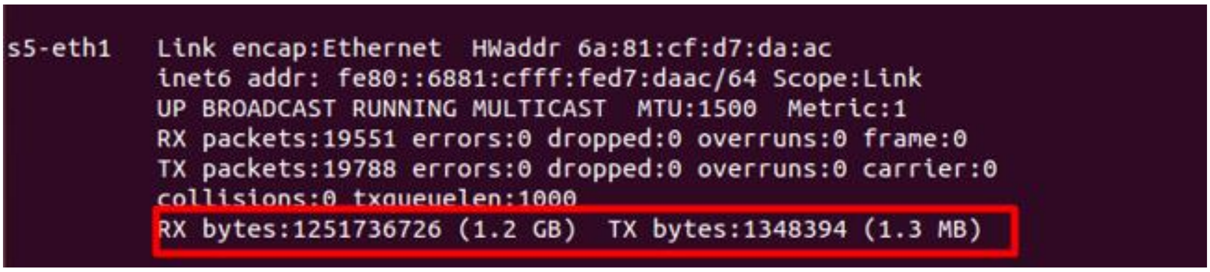


```
s1-eth2   Link encap:Ethernet  HWaddr 1a:a8:4e:84:0d:7f
          inet6 addr: fe80::18a8:4eff:fe84:d7f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:19669 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19665 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1251759960 (1.2 GB)  TX bytes:1324558 (1.3 MB)
```

Figura 73 Tráfico de s1 hacia s5

Fuente: Elaboración propia

Mediante la Figura 74, se verifica el siguiente salto de esta ruta que es s5 por la interfaz eth1 conectado hacia el puerto eth2 del switch s3, llegando así al destino el host h3



```
s5-eth1   Link encap:Ethernet  HWaddr 6a:81:cf:d7:da:ac
          inet6 addr: fe80::6881:cfff:fed7:daac/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:19551 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19788 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1251736726 (1.2 GB)  TX bytes:1348394 (1.3 MB)
```

Figura 74 Tráfico de s5 hacia s3

Fuente: Elaboración propia

La siguiente ruta presentada por la Tabla 8 del presente documento es como se muestra en la Figura 75, donde la ruta trazada es desde el h1-s1-s4-s3-h3.

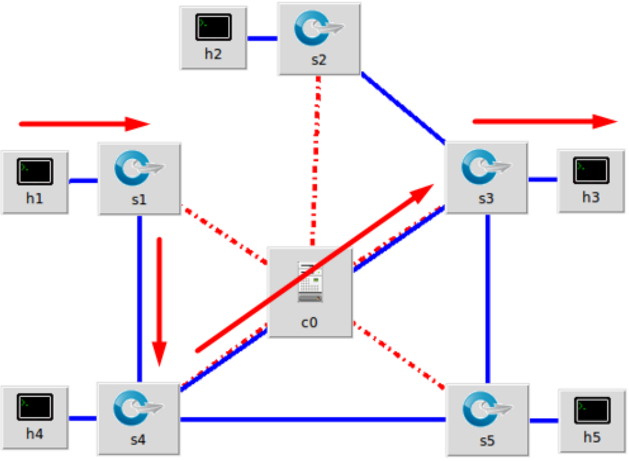
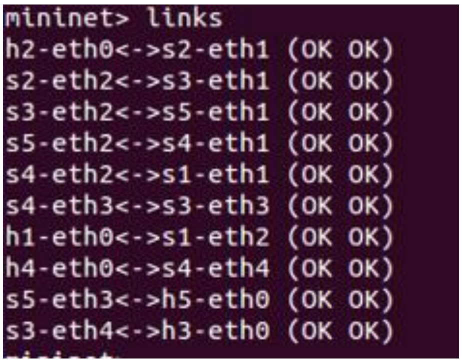


Figura 75 Cuarta ruta h1-s1-s4-s3-h3

Fuente: Elaboración propia

En la Figura 76, se presenta los links levantados el emulador de Mininet sin consideras las conexiones anteriores de s1-s3, s1-s2, s1-s5.

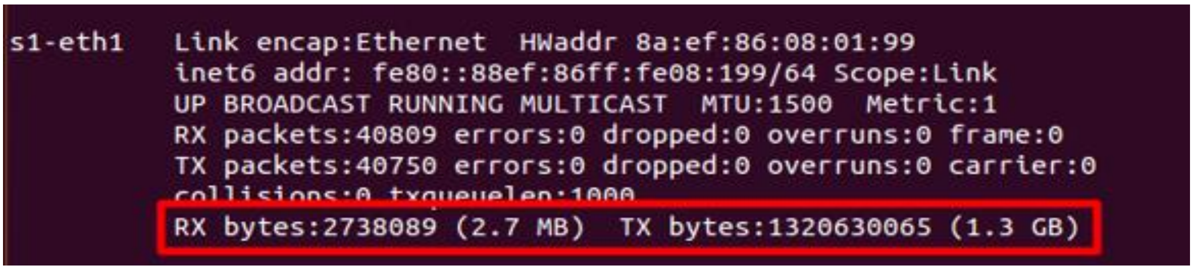


```
mininet> links
h2-eth0<->s2-eth1 (OK OK)
s2-eth2<->s3-eth1 (OK OK)
s3-eth2<->s5-eth1 (OK OK)
s5-eth2<->s4-eth1 (OK OK)
s4-eth2<->s1-eth1 (OK OK)
s4-eth3<->s3-eth3 (OK OK)
h1-eth0<->s1-eth2 (OK OK)
h4-eth0<->s4-eth4 (OK OK)
s5-eth3<->h5-eth0 (OK OK)
s3-eth4<->h3-eth0 (OK OK)
```

Figura 76 Enlaces levantados por el emulador Mininet sin s1-s3, s1-s2, s1-s5

Fuente: Elaboración propia

Se verifica mediante ipconfig el tráfico presentado por los puertos de los switches virtuales. Primeramente, por el switch s1 como se muestra en la Figura 77, donde el tráfico se conmuta s1 por el puerto eth1 mismo se encuentra conectado al switch s4 por el puerto eth2

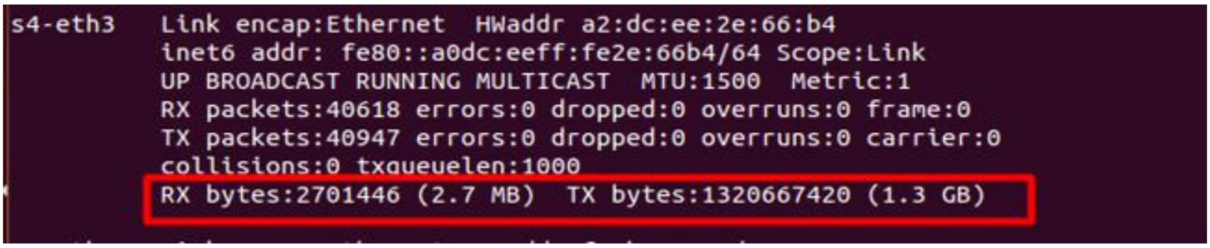


```
s1-eth1   Link encap:Ethernet  HWaddr 8a:ef:86:08:01:99
          inet6 addr: fe80::88ef:86ff:fe08:199/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:40809 errors:0 dropped:0 overruns:0 frame:0
          TX packets:40750 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2738089 (2.7 MB)  TX bytes:1320630065 (1.3 GB)
```

Figura 77 Tráfico de s1 hacia s4

Fuente: Elaboración propia

Seguido de la conmutación del switch s4 hacia el switch s3 como se presenta en la Figura 78, donde s4 se conecta mediante el puerto eth3 hacia s3 por el puerto eth3, llegando así hacia el destino h3.



```
s4-eth3   Link encap:Ethernet  HWaddr a2:dc:ee:2e:66:b4
          inet6 addr: fe80::a0dc:eeff:fe2e:66b4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:40618 errors:0 dropped:0 overruns:0 frame:0
          TX packets:40947 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2701446 (2.7 MB)  TX bytes:1320667420 (1.3 GB)
```

Figura 78 Tráfico de s4 hacia s3

Fuente: Elaboración propia

Finalmente se presenta la última ruta trazada dentro de la red SDN, como se muestra en la Figura 79.

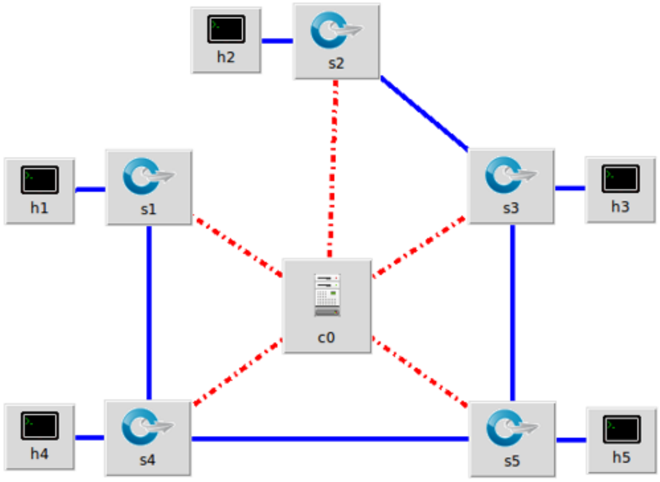
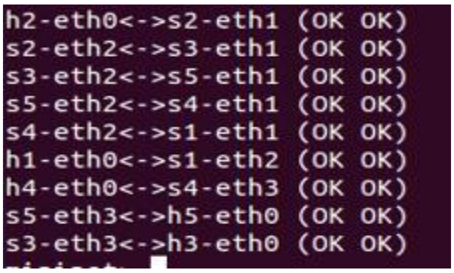


Figura 79 Quinta ruta h1-s1-s4-s5-s3-h3

Fuente: Elaboración propia

En la siguiente Figura 80, se presenta los links levantados por el emulador descartando las rutas anteriores y como se genera los nuevos enlaces.

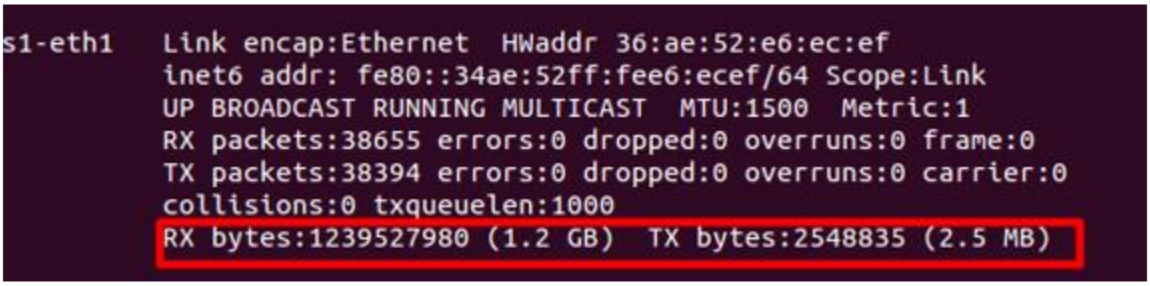


```
h2-eth0<->s2-eth1 (OK OK)
s2-eth2<->s3-eth1 (OK OK)
s3-eth2<->s5-eth1 (OK OK)
s5-eth2<->s4-eth1 (OK OK)
s4-eth2<->s1-eth1 (OK OK)
h1-eth0<->s1-eth2 (OK OK)
h4-eth0<->s4-eth3 (OK OK)
s5-eth3<->h5-eth0 (OK OK)
s3-eth3<->h3-eth0 (OK OK)
```

Figura 80 Enlaces levantados por el emulador Mininet sin s1-s3, s1-s2, s1-s5, s4-s3

Fuente: Elaboración propia

Se visualiza el tráfico presentado por la herramienta iperf desde s1 hacia s4 mediante la interfaz s1-eth1 hacia s4-eht2 ver Figura 81.

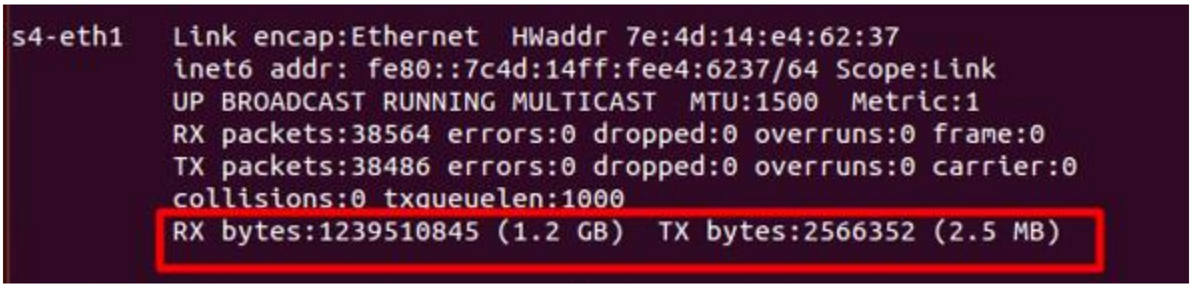


```
s1-eth1   Link encap:Ethernet  HWaddr 36:ae:52:e6:ec:ef
          inet6 addr: fe80::34ae:52ff:fee6:ecef/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38655 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38394 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1239527980 (1.2 GB)  TX bytes:2548835 (2.5 MB)
```

Figura 81 Tráfico de s1 hacia s4

Fuente: Elaboración propia

Posteriormente el salto presentado desde s4 hacia s5 mediante el puerto s4-eth1 (ver Figura 82) mismo que se encuentra conectado hacia s5-eth2



```
s4-eth1   Link encap:Ethernet  HWaddr 7e:4d:14:e4:62:37
          inet6 addr: fe80::7c4d:14ff:fee4:6237/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38564 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38486 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1239510845 (1.2 GB)  TX bytes:2566352 (2.5 MB)
```

Figura 82 Tráfico de s4 hacia s5

Fuente: Elaboración propia

Se tiene como ultimo salto desde s5 (ver Figura 83) por el puerto eth1 hacia s3 por el puerto eth3 teniendo en cuenta los enlaces establecidos por Mininet en la Figura 80

```
s5-eth1   Link encap:Ethernet  HWaddr f2:ed:63:a0:91:00
          inet6 addr: fe80::f0ed:63ff:fea0:9100/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38472 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38578 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1239493046 (1.2 GB)  TX bytes:2584443 (2.5 MB)
```

Figura 83 Tráfico de s5 hacia s3

Fuente: Elaboración propia

5.2 Medición de parámetros de rendimiento aplicando el algoritmo desarrollado

En el siguiente apartado se documenta los resultados obtenidos con la aplicación del algoritmo desarrollado, analizando los parámetros de rendimiento como tasas de transferencia, anchos de banda, tiempos de respuesta, pérdida de paquetes y jitter para posteriormente realizar un comparativa con los resultados obtenidos anteriormente en la red SDN sin aplicar el algoritmo.

5.2.1 Tasas de Transferencia

Se realiza las pruebas respectivas en escenario de prueba 1 correspondiente al testbed1 donde se compone de 5 switches con múltiples conexiones y 5 hosts como se muestra en la Figura 84.

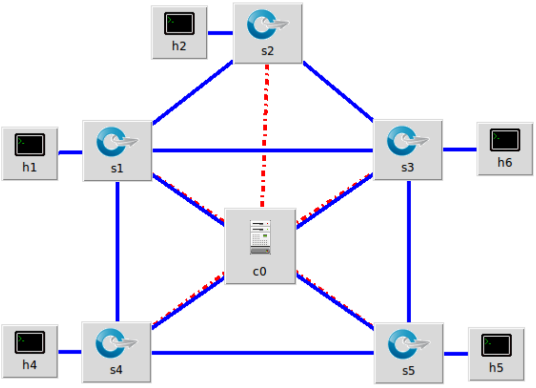
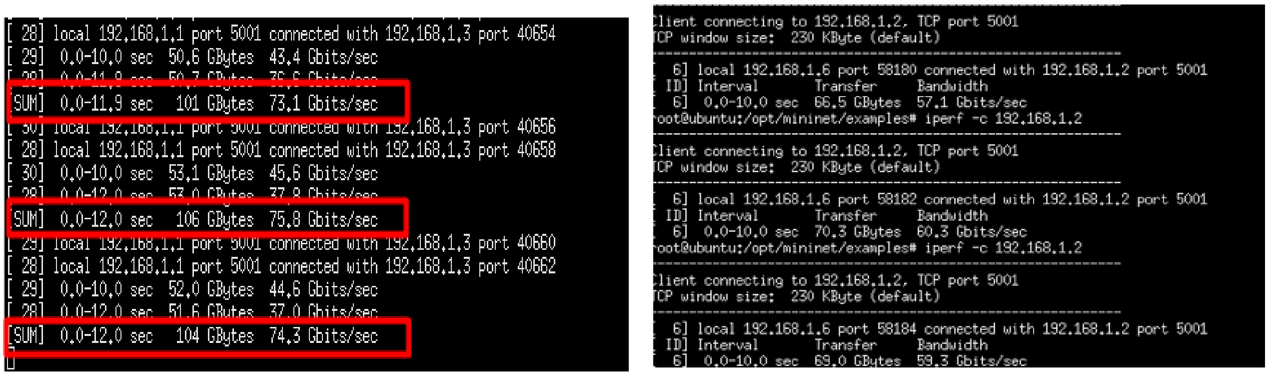


Figura 84 Testbed numero 1

Fuente: Elaboración propia

Mediante la herramienta de **iperf** se realiza una prueba de tráfico entre los terminales h1 y h6 con la finalidad de obtener el ancho de banda y tasas de transferencia, como se aprecia en la Figura 85, donde se obtiene un máximo de ancho de banda de 101Gbs a 106Gb y una tasa de transferencia de 73.1Gbs a 75.8Gbs debido a la capacidad del enlace que se emula dentro de la herramienta de Mininet.



```
[ 28] local 192.168.1.1 port 5001 connected with 192.168.1.3 port 40654
[ 29]  0.0-10.0 sec  50.6 GBytes  43.4 Gbits/sec
[ 28]  0.0-11.9 sec  50.7 GBytes  36.6 Gbits/sec
[SUM]  0.0-11.9 sec   101 GBytes  73.1 Gbits/sec
[ 30] local 192.168.1.1 port 5001 connected with 192.168.1.3 port 40656
[ 28] local 192.168.1.1 port 5001 connected with 192.168.1.3 port 40658
[ 30]  0.0-10.0 sec  53.1 GBytes  45.6 Gbits/sec
[ 28]  0.0-12.0 sec  53.0 GBytes  37.8 Gbits/sec
[SUM]  0.0-12.0 sec   106 GBytes  75.8 Gbits/sec
[ 29] local 192.168.1.1 port 5001 connected with 192.168.1.3 port 40660
[ 28] local 192.168.1.1 port 5001 connected with 192.168.1.3 port 40662
[ 29]  0.0-10.0 sec  52.0 GBytes  44.6 Gbits/sec
[ 28]  0.0-12.0 sec  51.6 GBytes  37.0 Gbits/sec
[SUM]  0.0-12.0 sec   104 GBytes  74.3 Gbits/sec
```

```
------------------------------------------------------------
Client connecting to 192.168.1.2, TCP port 5001
TCP window size:  230 KByte (default)
------------------------------------------------------------
[  6] local 192.168.1.6 port 58180 connected with 192.168.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  66.5 GBytes  57.1 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.1.2
------------------------------------------------------------
Client connecting to 192.168.1.2, TCP port 5001
TCP window size:  230 KByte (default)
------------------------------------------------------------
[  6] local 192.168.1.6 port 58182 connected with 192.168.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  70.3 GBytes  60.3 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.1.2
------------------------------------------------------------
Client connecting to 192.168.1.2, TCP port 5001
TCP window size:  230 KByte (default)
------------------------------------------------------------
[  6] local 192.168.1.6 port 58184 connected with 192.168.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  69.0 GBytes  59.3 Gbits/sec
```

Figura 85 Tasas de transferencia y anchos de banda testbed 1

Fuente: Elaboración propia

Mediante el análisis de la norma RFC 2544 se realiza la valoración de los resultados obtenidos en le medición del rendimiento por tasas de transferencia y anchos de banda del testbed 1 propuesto, obteniendo como resultado la siguiente Figura 86; donde el eje Y representa el tamaño de los paquetes enviados y eje X el tamaño del paquete recibido desde el host 1 hacia el host 6 de la topología propuesta.

Tabla 8 Tasas de Transferencia TestBed1

Datagrama Enviado Bytes	Tasas tranferencia Mbps
85	120
171	241
254	357
340	478
420	588
498	699
561	787

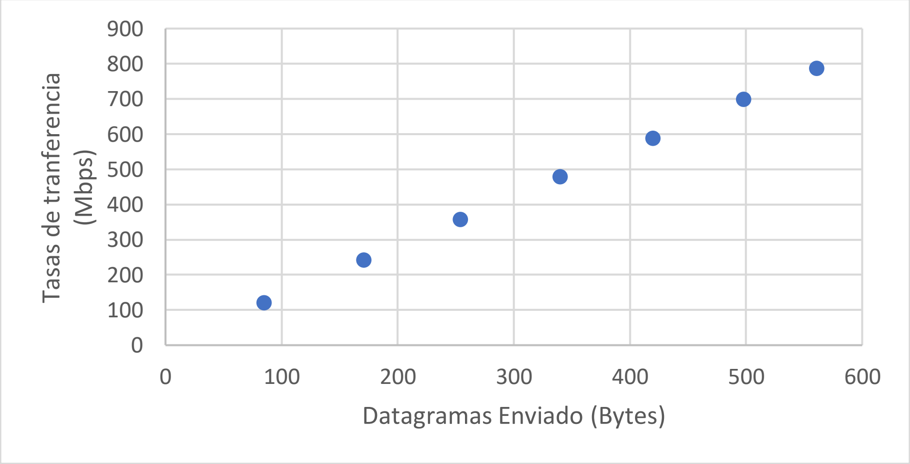


Figura 86 Gráfica de tendencia de las tasas de transferencia del testbed1

Fuente: Elaboración propia

De la misma forma se realiza la prueba para el segundo escenario de prueba donde se obtiene servidores de voz y video para pruebas de calidad como se presenta en la Figura 87.

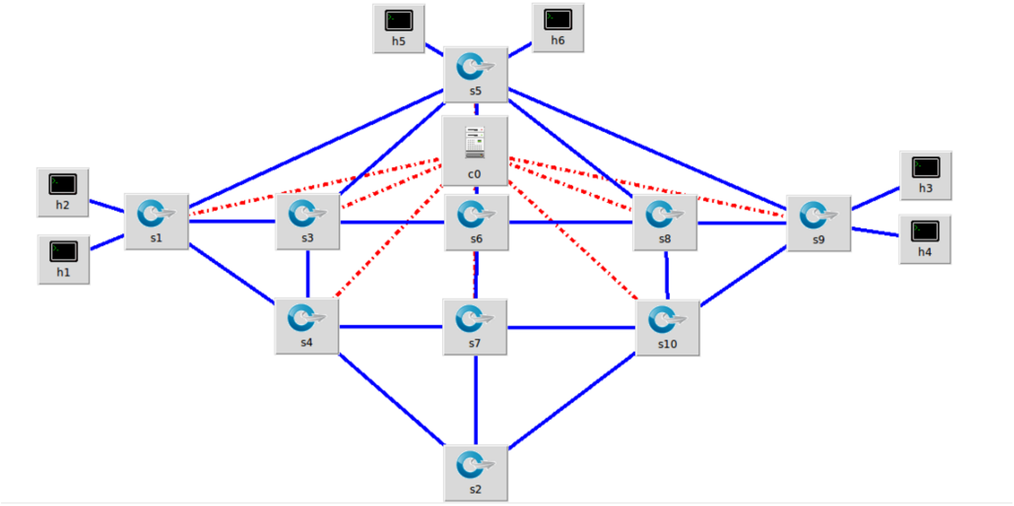
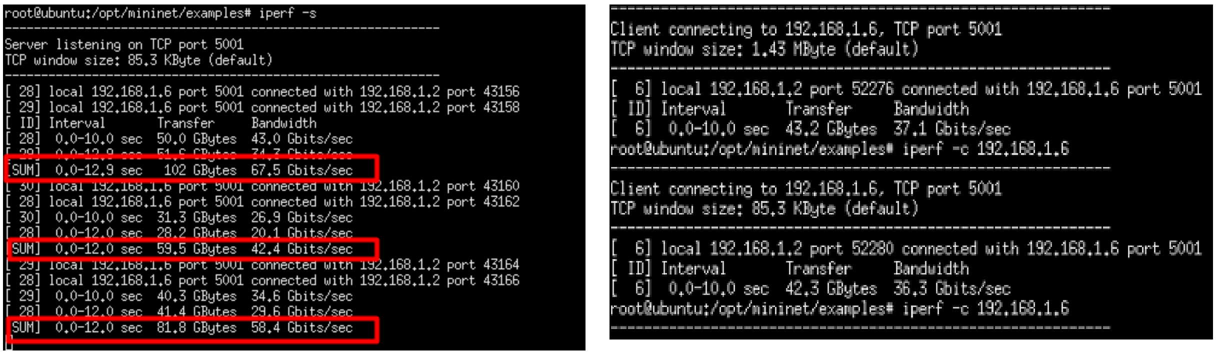


Figura 87 Testbed numero 2

Fuente: Elaboración propia

Se realiza como anteriormente la prueba de la herramienta de iperf para la obtener los valores de tasas de transferencia y anchos de banda desde el host h2 hacia el host h6, siendo este último donde se aloja el servicio de voz y video. Obteniendo como resultado tasas de transferencia desde 59.5 Gbs a 102 Gbs y anchos de banda de 42.4 Gbs a 67.5Gbs. cómo se presenta en la Figura 88



```
root@ubuntu:/opt/mininet/examples# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 43156
[ 29] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 43158
[ ID] Interval       Transfer     Bandwidth
[ 28]  0.0-10.0 sec  50.0 GBytes  43.0 Gbits/sec
[ 29]  0.0-12.9 sec  51.6 GBytes  34.3 Gbits/sec
[SUM]  0.0-12.9 sec   102 GBytes  67.5 Gbits/sec
[ 30] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 43160
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 43162
[ 30]  0.0-10.0 sec  31.3 GBytes  26.9 Gbits/sec
[ 28]  0.0-12.0 sec  28.2 GBytes  20.1 Gbits/sec
[SUM]  0.0-12.0 sec  59.5 GBytes  42.4 Gbits/sec
[ 29] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 43164
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 43166
[ 29]  0.0-10.0 sec  40.3 GBytes  34.6 Gbits/sec
[ 28]  0.0-12.0 sec  41.4 GBytes  29.6 Gbits/sec
[SUM]  0.0-12.0 sec  81.8 GBytes  58.4 Gbits/sec
```

```
------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
TCP window size: 1.43 MByte (default)
------------------------------------------------------------
[  6] local 192.168.1.2 port 52276 connected with 192.168.1.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  43.2 GBytes  37.1 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.1.6
------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 192.168.1.2 port 52280 connected with 192.168.1.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  42.3 GBytes  36.3 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.1.6
------------------------------------------------------------
```

Figura 88 Tasas de transferencia y anchos de banda testbed 2

Fuente: Elaboración propia

Se realiza el análisis de la norma RFC 2544 para la valoración de los resultados obtenidos en le medición del rendimiento por tasas de transferencia del testbed 2 propuesto, obteniendo como resultado en la Tabla 10 y la tendencia de la pendiente en la Figura 89; donde el eje Y representa el tamaño de los paquetes enviados y eje X el tamaño del paquete recibido desde el host 1 hacia el host 6 de la topología propuesta.

Tabla 9 Tasas de Transferencia TestBed2

Datagrama Enviado	**Tasas tranferencia**
Bytes	**Mbps**
127	179
212	298
296	413
381	535
460	637
540	758
644	904

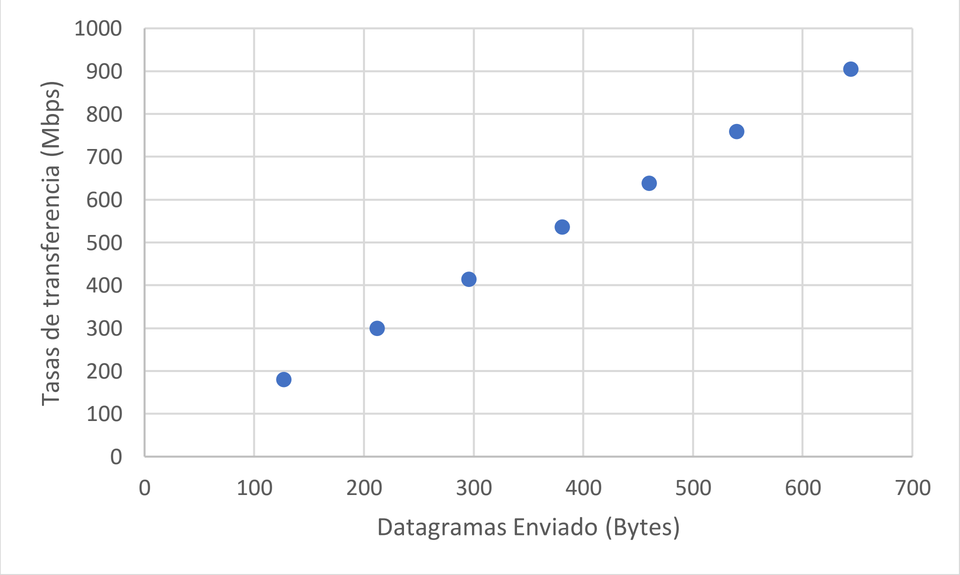
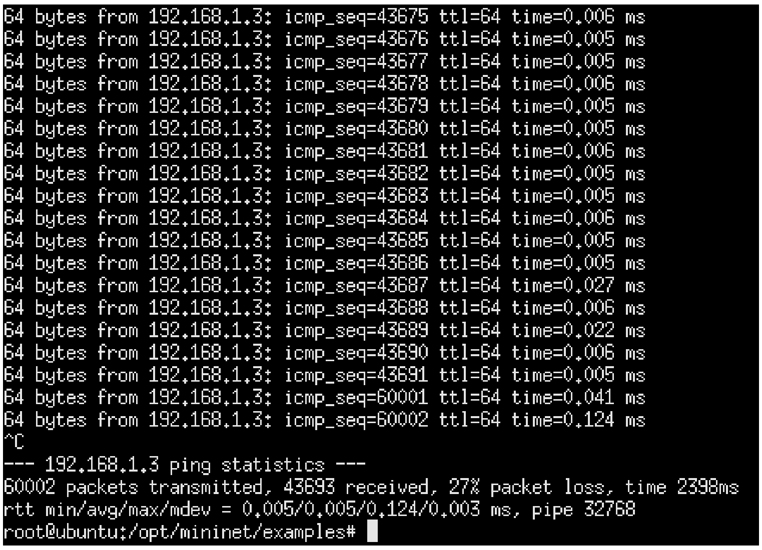


Figura 89 Gráfica de tendencia de las tasas de transferencia del testbed2

Fuente: Elaboración propia

5.2.2 Perdidas de paquetes

Se realiza mediante la herramienta de ping una cantidad determina de paquetes enviados y de esta forma verificar cuantos paquetes se perdieron en el proceso. Según la Figura 90 se puede aprecias que de 60000 paquetes enviados se recibieron 43693, teniendo como resultado el 27% de paquetes perdidos al momento de su ejecución.



```
64 bytes from 192.168.1.3: icmp_seq=43675 ttl=64 time=0.006 ms
64 bytes from 192.168.1.3: icmp_seq=43676 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43677 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43678 ttl=64 time=0.006 ms
64 bytes from 192.168.1.3: icmp_seq=43679 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43680 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43681 ttl=64 time=0.006 ms
64 bytes from 192.168.1.3: icmp_seq=43682 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43683 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43684 ttl=64 time=0.006 ms
64 bytes from 192.168.1.3: icmp_seq=43685 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43686 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=43687 ttl=64 time=0.027 ms
64 bytes from 192.168.1.3: icmp_seq=43688 ttl=64 time=0.006 ms
64 bytes from 192.168.1.3: icmp_seq=43689 ttl=64 time=0.022 ms
64 bytes from 192.168.1.3: icmp_seq=43690 ttl=64 time=0.006 ms
64 bytes from 192.168.1.3: icmp_seq=43691 ttl=64 time=0.005 ms
64 bytes from 192.168.1.3: icmp_seq=60001 ttl=64 time=0.041 ms
64 bytes from 192.168.1.3: icmp_seq=60002 ttl=64 time=0.124 ms
^C
--- 192.168.1.3 ping statistics ---
60002 packets transmitted, 43693 received, 27% packet loss, time 2398ms
rtt min/avg/max/mdev = 0.005/0.005/0.124/0.003 ms, pipe 32768
root@ubuntu:/opt/mininet/examples# 
```

Figura 90 Perdida de paquetes en el testbed 1

Fuente: Elaboración propia

Mediante la norma RFC2544 para el análisis del rendimiento de la red de evaluación se sometió a la topología testbed1, compuesta por 5 OpenVswitch, 5 hosts y el enlace generado entre el h1 a h3. En la Tabla 11 se muestran los valores de prueba realizados y resultados obtenidos. En la Figura 91 se muestra la tendencia de la recta en relación de paquetes enviados y paquetes perdidos.

Tabla 10 Valoración de paquetes perdidos en el Testbed1

Paquetes transmitidos	Paquetes recibidos	Paquetes Perdidos
10000	8039	1961
20000	16876	3124

30000	25314	4686
40000	33751	6249
50000	42189	7811
60000	43692	16308
65538	43693	21845

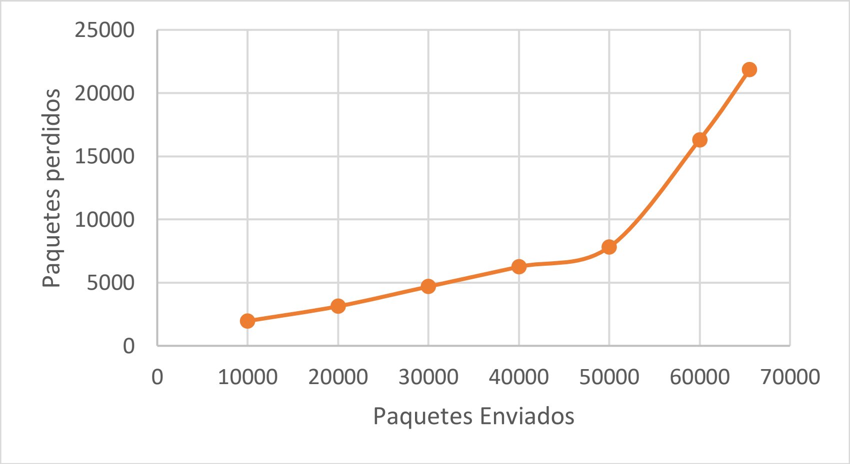
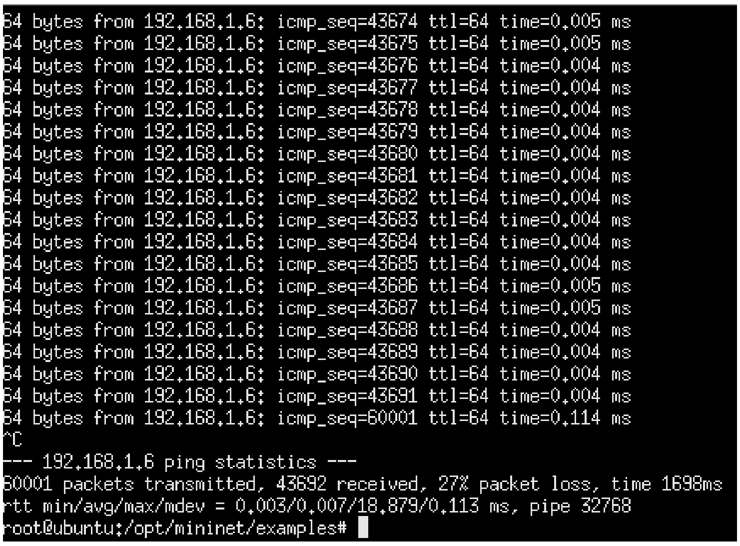


Figura 91 Tendencia de la recta en la perdida de paquetes del Testbed1

Fuente: Elaboración propia

De la misma forma procedemos a realizar las pruebas respectivas con el segundo escenario de prueba, donde los resultados se presentan en la Figura 92, siendo que de 60000 paquetes enviados 43692 fueron respondidos siendo equivalente a un 27% de paquetes perdidos. Hay que considerar que a pesar de que el porcentaje es similar antes de aplicar el algoritmo desarrollado, el tiempo de respuesta es menor de 1698ms a comparación de 109497ms ver Figura 42.



```
64 bytes from 192.168.1.6: icmp_seq=43674 ttl=64 time=0.005 ms
64 bytes from 192.168.1.6: icmp_seq=43675 ttl=64 time=0.005 ms
64 bytes from 192.168.1.6: icmp_seq=43676 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43677 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43678 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43679 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43680 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43681 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43682 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43683 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43684 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43685 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43686 ttl=64 time=0.005 ms
64 bytes from 192.168.1.6: icmp_seq=43687 ttl=64 time=0.005 ms
64 bytes from 192.168.1.6: icmp_seq=43688 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43689 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43690 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43691 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=60001 ttl=64 time=0.114 ms
^C
--- 192.168.1.6 ping statistics ---
60001 packets transmitted, 43692 received, 27% packet loss, time 1698ms
rtt min/avg/max/mdev = 0.003/0.007/18.879/0.113 ms, pipe 32768
root@ubuntu:/opt/mininet/examples#
```

Figura 92 Perdida de paquetes en el testbed 2

Fuente: Elaboración propia

Mediante la norma RFC2544 para el análisis del rendimiento de la red de evaluación se sometió a la topología testbed2, compuesta varios nodos OpenVswitch, servidores y el enlace generado entre el h1 a h6. En la Tabla 12 se muestran los valores de prueba realizados y resultados obtenidos. En la Figura 93 se muestra la tendencia de la recta en relación de paquetes enviados y paquetes perdidos.

Tabla 11 Valoración de paquetes perdidos en el Testbed2

Paquetes transmitidos	Paquetes recibidos	Paquetes Perdidos
10000	8439	1561
20000	17875	2125
30000	26314	3686
40000	34751	5249
50000	42189	7811
60000	43692	16308
65538	45694	19844

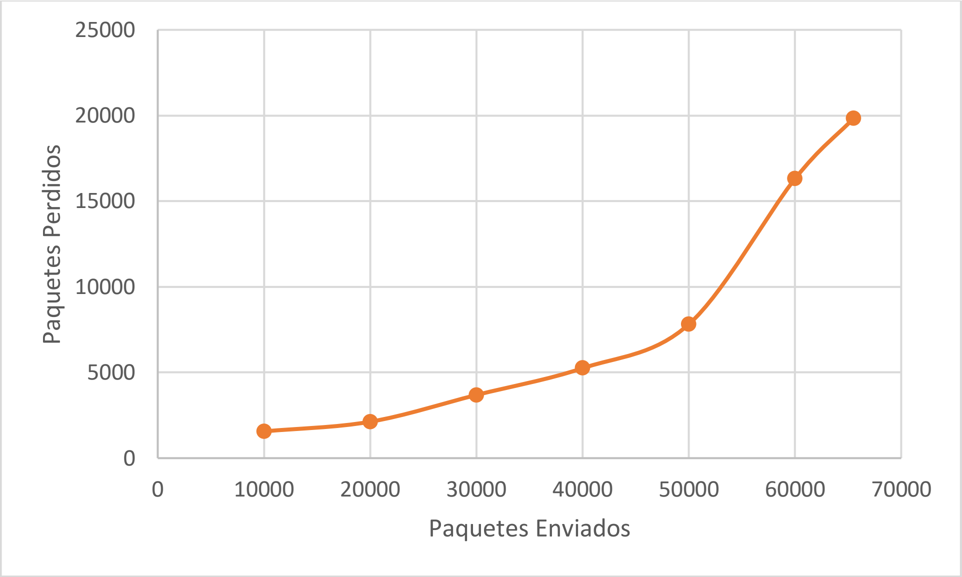


Figura 93 Tendencia de la recta en la perdida de paquetes del Testbed2

Fuente: Elaboración propia

5.2.3 Jitter

Para medir este parámetro se hace referencia al testbed 2 donde se integra los servidores de voz y video. Para este estudio se realiza una llamada mediante el servidor Elastix alojado en el host 6 configurado con la ip 192.168.1.6 dentro de la red SDN emulada Mininet. En la Figura 94 se presenta la secuencia de inicialización entre los terminales y el servidor; posteriormente la conexión de llamada entre los dos terminales hasta ser finalizada.

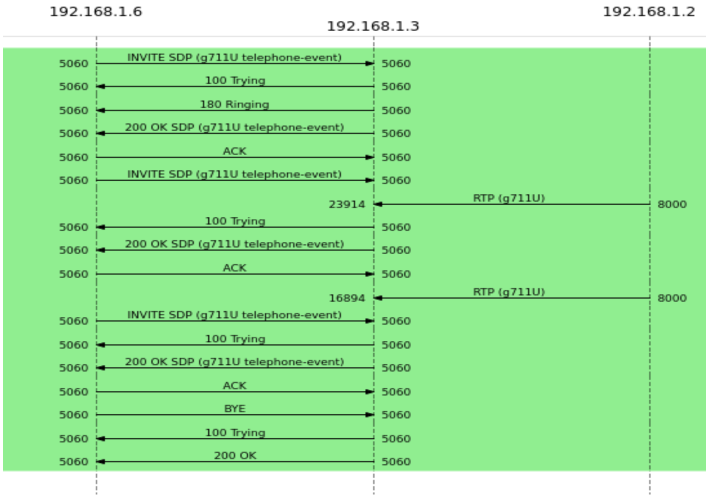
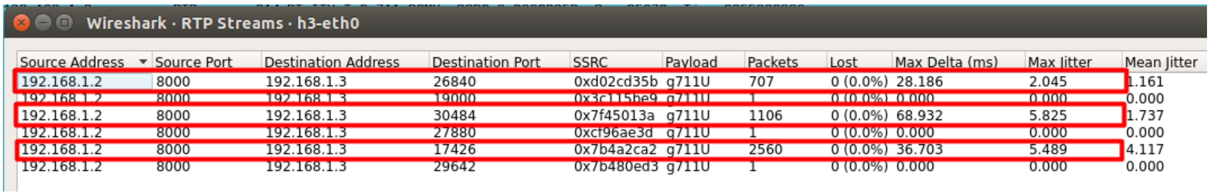


Figura 94 Secuencia de llamada entre host h2 y host h3

Fuente: Elaboración propia

Se realiza el análisis mediante Wireshark la calidad de la llamada y como esta se lleva a cabo. Mediante esta herramienta se filtra los paquetes RTP entre el host h2 y host h3 en donde con la ayuda de RTP Streams se puede observar la cantidad de paquetes que se pueden perder en una llamada y el máximo de jitter que experimento la misma. En este caso la llamada de prueba obtuvo una máximo de 68.932ms y sin pérdida de paquetes del 0% como se presenta en la Figura 95.



Source Address	Source Port	Destination Address	Destination Port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter	Mean Jitter
192.168.1.2	8000	192.168.1.3	26840	0xd02cd35b	g711U	707	0 (0.0%)	28.186	2.045	1.161
192.168.1.2	8000	192.168.1.3	19000	0x3c115be9	g711U	1	0 (0.0%)	0.000	0.000	0.000
192.168.1.2	8000	192.168.1.3	30484	0x7f45013a	g711U	1106	0 (0.0%)	68.932	5.825	1.737
192.168.1.2	8000	192.168.1.3	27880	0xcf96ae3d	g711U	1	0 (0.0%)	0.000	0.000	0.000
192.168.1.2	8000	192.168.1.3	17426	0x7b4a2ca2	g711U	2560	0 (0.0%)	36.703	5.489	4.117
192.168.1.2	8000	192.168.1.3	29642	0x7b480ed3	g711U	1	0 (0.0%)	0.000	0.000	0.000

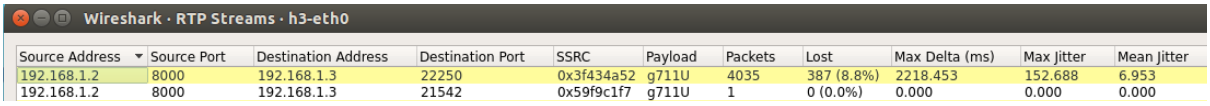
Figura 95 Análisis de jitter y perdida de paquetes en llamada

Fuente: Propia

5.3 Calidad del Servicio

Para el estudio de los servicios prestados en el presente documento se obtuvo mejores resultados en la prestación de la calidad de voz y video, debido a que el tiempo de retardo (jitter) mejoro a comparación del anterior estudio.

En donde el valor de jitter de la llamada aplicado el algoritmo es de 152.688ms como se muestra en la Figura 96 y el sin aplicar el algoritmo es de 247.514ms donde la calidad de la llamada se ve afectada. Cabe mencionar que el escenario simulado es saturando con la herramienta iperf del host h2 a h3 el enlace donde se transporta la llamada obteniendo estos resultados, mejores que el anterior.

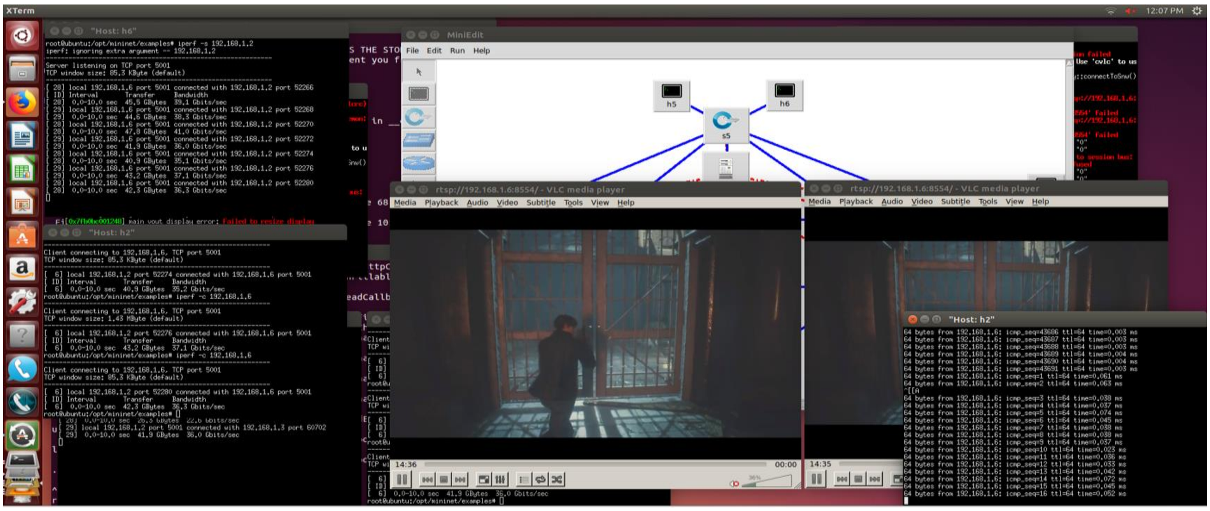


Source Address	Source Port	Destination Address	Destination Port	SSRC	Payload	Packets	Lost	Max Delta (ms)	Max Jitter	Mean Jitter
192.168.1.2	8000	192.168.1.3	22250	0x3f434a52	g711U	4035	387 (8.8%)	2218.453	152.688	6.953
192.168.1.2	8000	192.168.1.3	21542	0x59f9c1f7	g711U	1	0 (0.0%)	0.000	0.000	0.000

Figura 96 Retardo de la llamada en enlace saturado

Fuente: Elaboración propia

De igual manera para el servicio de video streaming se realiza la saturación del canal, pero este no presenta pérdida de calidad de video, ni de pines al servidor de video, como se muestra en la Figura 9, donde se ejecuta la función iperf, ping y la transmisión de video.



```
root@ubuntu:/opt/mininet/examples# iperf -s 192.168.1.2
iperf: ignoring extra argument -- 192.168.1.2
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52266
[ ID] Interval       Transfer     Bandwidth
[ 28]  0.0-10.0 sec  45.5 GBytes  39.1 Gbits/sec
[ 29] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52268
[ 29]  0.0-10.0 sec  44.6 GBytes  38.3 Gbits/sec
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52270
[ 28]  0.0-10.0 sec  47.8 GBytes  41.0 Gbits/sec
[ 29] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52272
[ 29]  0.0-10.0 sec  41.9 GBytes  36.0 Gbits/sec
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52274
[ 28]  0.0-10.0 sec  40.9 GBytes  35.1 Gbits/sec
[ 29] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52276
[ 29]  0.0-10.0 sec  43.2 GBytes  37.1 Gbits/sec
[ 28] local 192.168.1.6 port 5001 connected with 192.168.1.2 port 52280
[ 28]  0.0-10.0 sec  42.3 GBytes  36.3 Gbits/sec
```

```
------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 192.168.1.2 port 52274 connected with 192.168.1.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  40.9 GBytes  35.2 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.1.6
------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
TCP window size: 1.43 MByte (default)
------------------------------------------------------------
[  6] local 192.168.1.2 port 52276 connected with 192.168.1.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  43.2 GBytes  37.1 Gbits/sec
root@ubuntu:/opt/mininet/examples# iperf -c 192.168.1.6
------------------------------------------------------------
Client connecting to 192.168.1.6, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 192.168.1.2 port 52280 connected with 192.168.1.6 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.0 sec  42.3 GBytes  36.3 Gbits/sec
root@ubuntu:/opt/mininet/examples# 
```

```
[ 28]  0.0-10.0 sec  26.3 GBytes  22.6 Gbits/sec
[ 29] local 192.168.1.2 port 5001 connected with 192.168.1.3 port 60702
[ 29]  0.0-10.0 sec  41.9 GBytes  36.0 Gbits/sec
```

```
64 bytes from 192.168.1.6: icmp_seq=43686 ttl=64 time=0.003 ms
64 bytes from 192.168.1.6: icmp_seq=43687 ttl=64 time=0.003 ms
64 bytes from 192.168.1.6: icmp_seq=43688 ttl=64 time=0.003 ms
64 bytes from 192.168.1.6: icmp_seq=43689 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43690 ttl=64 time=0.004 ms
64 bytes from 192.168.1.6: icmp_seq=43691 ttl=64 time=0.003 ms
64 bytes from 192.168.1.6: icmp_seq=1 ttl=64 time=0.061 ms
64 bytes from 192.168.1.6: icmp_seq=2 ttl=64 time=0.063 ms
^[[A
64 bytes from 192.168.1.6: icmp_seq=3 ttl=64 time=0.038 ms
64 bytes from 192.168.1.6: icmp_seq=4 ttl=64 time=0.037 ms
64 bytes from 192.168.1.6: icmp_seq=5 ttl=64 time=0.074 ms
64 bytes from 192.168.1.6: icmp_seq=6 ttl=64 time=0.045 ms
64 bytes from 192.168.1.6: icmp_seq=7 ttl=64 time=0.038 ms
64 bytes from 192.168.1.6: icmp_seq=8 ttl=64 time=0.038 ms
64 bytes from 192.168.1.6: icmp_seq=9 ttl=64 time=0.037 ms
64 bytes from 192.168.1.6: icmp_seq=10 ttl=64 time=0.023 ms
64 bytes from 192.168.1.6: icmp_seq=11 ttl=64 time=0.036 ms
64 bytes from 192.168.1.6: icmp_seq=12 ttl=64 time=0.033 ms
64 bytes from 192.168.1.6: icmp_seq=13 ttl=64 time=0.042 ms
64 bytes from 192.168.1.6: icmp_seq=14 ttl=64 time=0.072 ms
64 bytes from 192.168.1.6: icmp_seq=15 ttl=64 time=0.045 ms
64 bytes from 192.168.1.6: icmp_seq=16 ttl=64 time=0.052 ms
```

Figura 97 Streaming de video aplicando el algoritmo

Fuente: Elaboración propia

5.4 Análisis de mejoras presentadas mediante el algoritmo de balanceo de carga

En la siguiente sección, se presentan cuadros de porcentajes de los parámetros de rendimiento evaluados como son el jitter, tasas de transferencia, anchos de banda, pérdida de paquetes. Estos parámetros se los considera antes y después de ser aplicado el algoritmo desarrollado.

En la siguiente Figura 98, se visualiza los valores obtenidos antes de ser aplicado el algoritmo, misma que se encuentra en la sección final del capítulo 3.

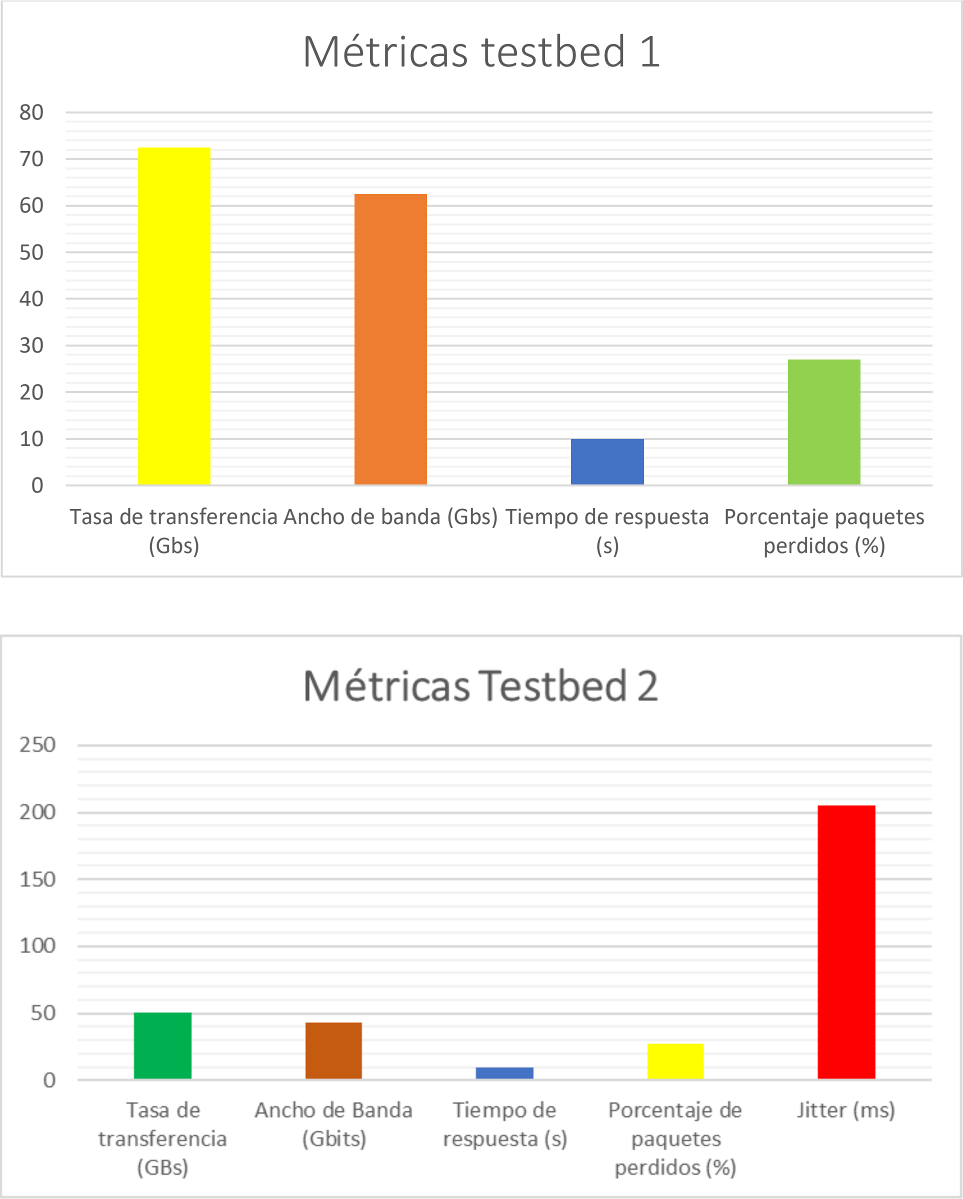


Figura 98 Diagrama de barras de las métricas de rendimiento sin aplicar el algoritmo

Fuente: Elaboración propia

Posteriormente se realiza el siguiente diagrama de barras de los resultados obtenidos aplicando el algoritmo desarrollo en los mismos escenarios de prueba o testbed. En la Figura 99 se presenta los parámetros de medición de rendimiento en las topologías emuladas por Mininet

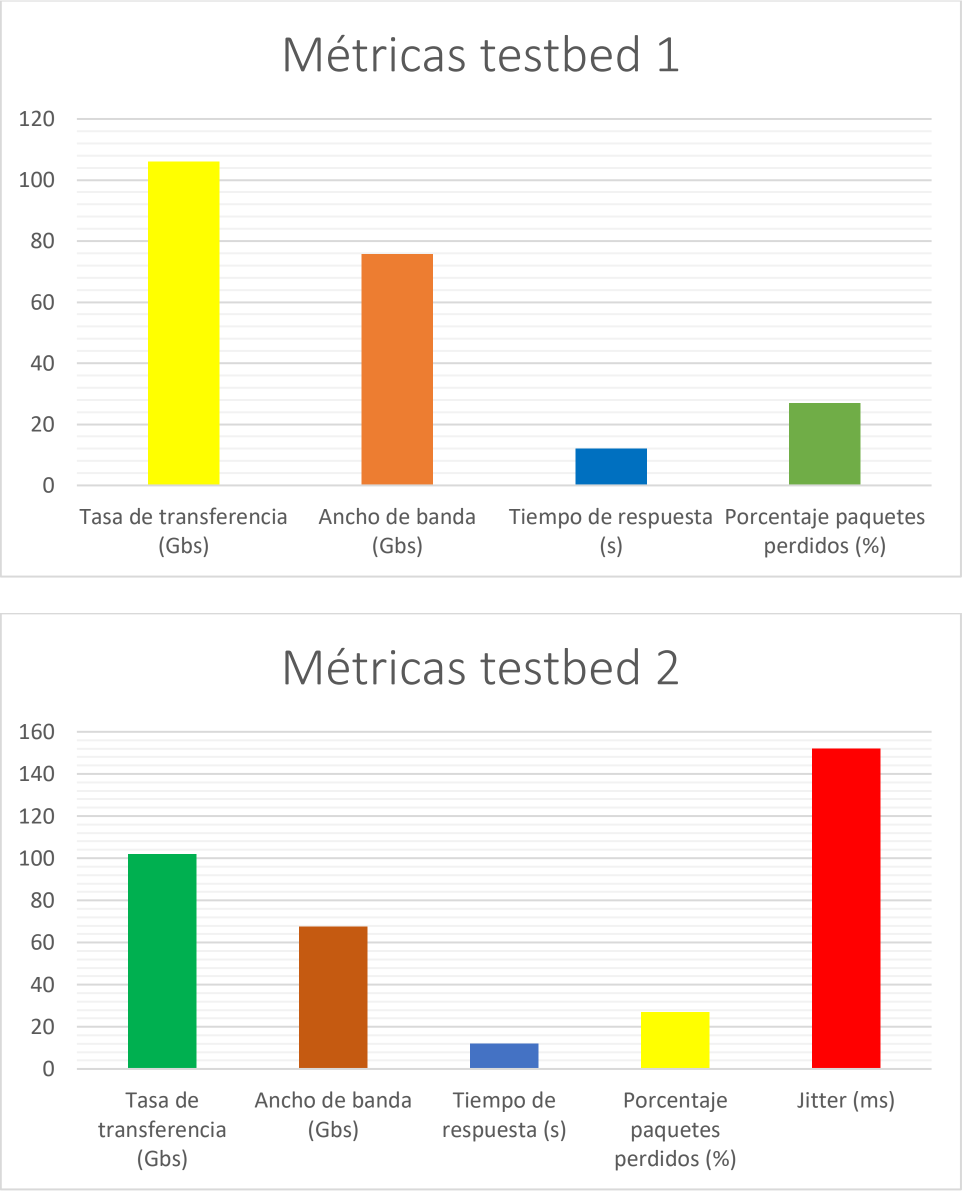


Figura 99 Diagrama de barras de las métricas de rendimiento aplicando el algoritmo

Fuente: Elaboración propia

Considerando los parámetros de rendimiento estudiados, se puede evidenciar que el algoritmo obtiene mejores resultados esperados, como mayor ancho de banda, mejores tasas de transferencia, un tiempo de respuesta casi similar, porcentaje de perdida de paquetes en el mismo nivel y valores de retardo en la llamada mucho menor al obtenido en los primeros resultados de estudio.

Finalmente se puede concluir que se obtiene mejor rendimiento en la red y con enrutamiento de paquetes mucho más optimo en la saturación de enlaces, mejorando la estabilidad de las redes SDN.

Conclusiones

- El algoritmo de balanceo de carga dinámico desarrollado, logró distribuir los flujos de tráfico por múltiples rutas de una red SDN, donde se obtuvo mejoras en el rendimiento de la red SDN y resultados como mayor anchos de banda, mejor latencia y menor retardo (jitter) en los escenarios o testbeds presentados para el desarrollo de este tema de tesis.

- A pesar que existe una gran variedad de estudios sobre las redes SDN y la ingeniería de tráfico, estos estudios son recientes y hasta el momento no existe una red completamente SDN en producción para el manejo de flujos de tráfico y calidad de servicio, debido a que es necesario de equipos de alto rendimiento de procesamiento como el controlador para el manejo de datos.

- Existe una amplia diversas de herramientas para el manejo y estudio de las redes SDN que pueden ajustarse a las necesidades del usuario, como en el presente estudio, que se implemento el emulador mininet y el controlador ryu, ambas herramientas de software libre haciéndolas adecuadas para la recreación de ambientes de prueba.

- Mediante el uso del modelo UF1880 se estableció parámetros de medición de rendimiento de las redes para de esta forma evaluar resultados obtenidos antes y después de aplicar el algoritmo desarrollado en los diferentes escenarios de prueba o testbeds propuestos.

- El desarrollo del algoritmo de balanceo de carga se lo realizo mediante un script en lenguaje de programación python debido a que la base de desarrollo del controlador ryu es completamente en este lenguaje.

- Las métricas tomadas en estudio para el mecanismo de balanceo del controlador son mediante los valores de anchos de banda y latencia del enlace entre los nodos OpenvSwitch hacia los destinatarios para mejorar la distribución de flujos en la red SDN.

- Mediante la aplicación del algoritmo desarrollado en el controlador ryu, se obtuvo como resultado mejores anchos de banda, tasas de transferencia más amplios y fluctuación de retardo menor en lo que corresponde al análisis de calidad de VoIP, cabe mencionar que dicho algoritmo ira seleccionando la mejor ruta dependiendo de la calidad del enlace hacia el destinatario y el algoritmo de enrutamiento implementado.

- Se puede concluir que el algoritmo propuesto presenta mejoras ante otros algoritmos tradicionales, por el simple motivo que se acopla a las necesidades que el usuario presente en ese momento para el desarrollo de la red, teniendo como desventaja que cada modificación siempre debe realizarse en el código de programación del controlador.

Recomendaciones

- Para el estudio el estudio de las redes SDN, es indispensable disponer de un equipo robusto, para este estudio se usó un equipo laptop con memoria RAM de 16GB y 128 GB de disco sólido.

- El uso de las herramientas de Mininet y RYU demandan de bastante procesamiento ya que el emulador genera equipos virtualizados haciendo uso los recursos físicos y más aún al momento de levantar servicios como servidores VOIP y Streaming de Video en la red SDN.

- Para la comunicación del controlador y la red simulada se debe verificar la dirección IP del controlador y el puerto de comunicación que es 6653 TCP y este programarlo dentro del emulador Mininet.

- Es necesario estudiar el lenguaje de programación que cada controlador maneja dentro de las redes SDN para posteriormente realizar modificaciones o nuevas aplicaciones para la operación de la red.

- El estudio de las redes SDN es un tanto compleja debido a la escases de información presentada y estudios aún en desarrollo para el tratamiento y uso de herramientas como controladores SDN.

- Incentivar más al estudio de las redes SDN debido a que son tecnologías de nueva generación para en un futuro tener sistemas más centralizados y con mejor gestión de la información o datos.

Referencia Bibliográfica

Acedo, J. (7 de Septiembre de 2016). *Algoritmos de balanceo de carga*. Obtenido de http://programacion.jias.es/2016/09/algoritmos-de-balanceo-de-carga/

Álvarez, R. (2015). *Estudio de las redes definidas por software mediante el desarrollo de escenarios virtuales basados en el controlador OpenDayLight*. Obtenido de http://oa.upm.es/42968/1/TFM_RAUL_ALVAREZ_PINILLA.pdf

Ampuño, A. & Chávez, M. (2015). *Diseño y Simulación de una Red de DataCenters basada en Topología FAT-TREE en un ambiente de redes defindas por software (SDN)*. Obtenido de https://dspace.ups.edu.ec/bitstream/123456789/10405/1/UPS-GT001452.pdf

Betegón, M. (junio de 2018). *Estudio de técnicas de Ingeniería de Tráfico basadas en SDN.* Obtenido de https://repositorio.unican.es/xmlui/bitstream/handle/10902/14193/409476.pdf?sequence=1&isAllowed=y

Calle, A., et al. (2018). *Comparación de Parámetros para una Selección Apropiada de Herramientas de Simulación de Redes*. Obtenido de https://scielo.conicyt.cl/pdf/infotec/v29n6/0718-0764-infotec-29-06-00253.pdf

Cevallos, L. (2018). *IMPLEMENTACIÓN DE REDES DEFINIDAS POR SOFTWARE (SDN) SOBRE REDES IEEE 802.11 MEDIANTE MININET WI-FI.* Obtenido de http://dspace.espoch.edu.ec/handle/123456789/9144

Chambi, V. (2013). *Inteligencia Artificial Juegos* . Obtenido de http://www.revistasbolivianas.org.bo/pdf/rits/n1/n1a17.pdf

Cisco. (2016). *BGP Best Path Selection Algorithm*. Obtenido de https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13753-25.html

Cisco. (2005). *Guía de diseño de OSPF*. Obtenido de https://www.cisco.com/c/es_mx/support/docs/ip/open-shortest-path-first-ospf/7039-1.html

Cisco. (2005). *IGRP Metric*. Obtenido de https://www.cisco.com/c/en/us/support/docs/ip/interior-gateway-routing-protocol-igrp/13678-3.html

Cisco. (2018). *Meraki Documentation*. Obtenido de https://documentation.meraki.com/zGeneral_Administration/Tools_and_Troubleshooting/Troubleshooting_Client_Speed_using_iPerf

Cisco. (2018). *Redes Definidas por Software*. Obtenido de https://www.cisco.com/c/es_ec/solutions/software-defined-networking/overview.html

Conde, J. (2018). *Extensión de la funcionalidad de la aplicación VPLS para el controlador ONOS*. Obtenido de http://oa.upm.es/52004/1/PFC_JAVIER_CONDE_DIAZ_2018.pdf

Cosmas, J. (2015). *OpenFlow 1.3 Extension for OMNeT++*. Obtenido de https://www.researchgate.net/publication/291357317_OpenFlow_13_Extension_for_OMNeT

Cuevas, D. (Diciembre de 2013). *Análisis de Algoritmos Pathfinding*. Obtenido de http://opac.pucv.cl/pucv_txt/txt-5000/UCE5372_01.pdf

Cuevas, D. (2013). *Análisis de Algoritmos Pathfinding.* Valparaíso: Pontificia Universidad Católica de Valparaíso.

Erickson, J. (2019). *Depth-First Search*. Obtenido de http://jeffe.cs.illinois.edu/teaching/algorithms/book/06-dfs.pdf

Fernández, D. (Mayo de 2015). *RedIRIS.* Obtenido de http://www.rediris.es/difusion/eventos/ponencias/?id=frc2015-frc-ses-infrII-a5b3c1.pdf

Frómeta, D., et al . (2016). *Desarrollo de aplicaciones SDN*. Obtenido de http://www.revistatonoetecsa.cu/articulo/desarrollo-de-aplicaciones-sdn

García, C. (2014). *UF1880 - Gestión de redes telemáticas.* ic editorial.

Garg, P. (2019). *Breadth First Search*. Obtenido de https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

Geethu, E. (2015). *Heurística basada en dirección para Pathfinding en videojuegos*. Obtenido de https://www.sciencedirect.com/science/article/pii/S1877050915004743

GNS3. (2019). *Getting Started with GNS3*. Obtenido de https://docs.gns3.com/1PvtRW5eAb8RJZ11maEYD9_aLY8kkdhgaMB0wPCz8a38/index.html

Guerrero, G. (2017). *Desarrollo de una plataforma para evaluar calidad de servicios (QoS) en redes definidas por software (SDN)*. Obtenido de http://dspace.espoch.edu.ec/handle/123456789/8437

IBM. (2017). *Algorithms for making load-balancing decisions*. Obtenido de https://www.ibm.com/support/knowledgecenter/SS9H2Y_7.6.0/com.ibm.dp.doc/lbg_algorithms.html

Irizar, C. & Calderón, C. (2017). PROPUESTA PARA LA EVALUACIÓN DE PARÁMETROS DE QOS EN SDN. *Telemática* , 12-24.

Jain, R. (1991). *Art of Computer Systems Performance Analysis Techniques For Experimental Design Measurements Simulation And Modeling.* Wiley Computer Publishing.

Jeeva, R. (Septiembre de 2016). *OpenFlow-based control plane for Information-Centric Networking*. Obtenido de https://www.scss.tcd.ie/Stefan.Weber/PDFs/Jeeva%20Rajendran%20-%20MSc%20NDS%202016.pdf

Jerome et al. . (19 de Abril de 2018). *SDN-based load balancing for multi-path TCP*. Obtenido de https://ieeexplore.ieee.org/document/8406943

Kumbhare, A. (Septiembre de 2018). *OpenDaylight Current and Future Use Cases*. Obtenido de https://wiki.opendaylight.org/images/1/1b/ONS_EU18_OpenDaylight_v1.0.pdf

Kuster, C. (2015). *Balanceo de carga en redes IPv4, un enfoque de Redes Definidas por Software.* Obtenido de Universidad ORT: https://bibliotecas.ort.edu.uy/bibid/81983/file/2108

León, S. & Pino A. (2018). *Diseño de una red utilizando SDN (software defined network) para la empresa Calital Cía. Ltda. en la ciudad de Guayaquil*. Obtenido de http://repositorio.ug.edu.ec/handle/redug/33139

Lopez, B. (2005). *Algoritmo A*.* Nuevo laredo: Instituto tecnológico de Nuevo Laredo.

López, J. (Octubre de 2018). *Desarrollo de una interfaz entre el controlador SDN ONOS y Net2Plan para la optimización de redes*. Obtenido de http://repositorio.upct.es/bitstream/handle/10317/7567/tfm-lop-des.pdf?sequence=1&isAllowed=y

Mallik, A., & Hegde, S. . (2015). *A novel proposal to effectively combine multipath data forwarding for data center networks with congestion control and load balancing using Software-Defined Networking Approach.* Obtenido de https://ieeexplore.ieee.org/abstract/document/6996178

Master Móviles UA. (2019). *Protocolos de comunicación en red*. Obtenido de Master Móviles UA: https://mastermoviles.gitbook.io/tecnologias2/protocolos-de-comunicacion-en-red

Millán, R. (2014). *SDN: el futuro de las redes inteligentes*. Obtenido de https://www.ramonmillan.com/tutoriales/sdnredesinteligentes.php

Morató, D. (2016). *Balanceadores*. Obtenido de https://www.tlm.unavarra.es/~daniel/docencia/rng/rng15_16/slides/Tema1-06-Balanceadores.pdf

Neghabi, A., et al. (2018). *IEEExplore.* Obtenido de https://ieeexplore.ieee.org/document/8306964

NewNow. (20 de Febrero de 2018). *SDN: Los muchos significados de las redes definidas por software definidas por software*. Obtenido de https://www.thenewnow.es/tecnologia/los-muchos-significados-las-redes-definidas-software/

NSRC. (2013). *Comparación de IS-IS y OSPF*. Obtenido de https://www.nsrc.org/workshops/2015/walc/routing/raw-attachment/wiki/Agenda/04-ISIS-vs-OSPF.pdf

Open Network Operating System. (2018). *Controlador ONOS*. Obtenido de https://onosproject.org/members/

Open Networking Foundation. (05 de Diciembre de 2011). *OpenFlow 1.2*. Obtenido de https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf

Open Networking Foundation. (2 de Junio de 2017). *OpenFlow Switch Specification*. Obtenido de https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf

Open Networking Foundation. (2 de Junio de 2009). *OpenFlow versión 1.0*. Obtenido de https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf

Open Networking Foundation. (28 de Febrero de 2011). *OpenFlow versión 1.1*. Obtenido de https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf

Open Networking Foundation. (25 de Junio de 2012). *OpenFlow versión 1.3*. Obtenido de https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf

Open Networking Foundation. (14 de Octubre de 2013). *OpenFlow versión 1.4*. Obtenido de https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf

Oracle. (2010). *Reenvío de paquetes y rutas en redes IPv4*. Obtenido de https://docs.oracle.com/cd/E19957-01/820-2981/6nei0r10a/index.html

Oyala, M. (2015). *Diseño e implementación de una aplicación para balanceo de carga para una Red Definida por Software (SDN)*. Obtenido de https://bibdigital.epn.edu.ec/browse?type=author&value=Olaya+Yandun%2C+Marlon+Esteban

Pachés, J. A. (2020). *Estudio del controlador SDN Ryu sobre una Raspberry-Pi Model 4*. Valencia: Universitat Politécnica de Valencia.

Programador, Click;. (s.f.). *Instalación e implementación de POX del controlador de red SDN*. Obtenido de Programador click: https://programmerclick.com/article/63291436821/

Programador, Click;. (s.f.). *Notas realcionadas con el controlador SDN (3) RYU*. Obtenido de Programador click: https://programmerclick.com/article/5015731774/

Project Floodlight. (2019). *Floodlight*. Obtenido de http://www.projectfloodlight.org/floodlight/

Ramírez, M. & López, A. (2018). Redes de datos definidas por software-SDN, arquitectura, componentes y funcionamiento. *Journal de Ciencia e Ingeniería* , 55-61.

Recio, G. & Riocerezo, G. (2018 de febrero de 2018). *nae*. Obtenido de nae doing ahead: https://nae.global/5g-salto-internet-de-las-cosas/

Rojas, D. (2015). *ESTUDIO COMPARATIVO BASADO EN LA SIMULACION DE ESCENARIOS ENTRE LOS PROTOCOLOS DE ENCAMINAMIENTO SAODV Y AODV UTILIZADOS EN REDES AD-HOC*. Obtenido de http://repositorio.espe.edu.ec/jspui/bitstream/21000/11247/1/T-ESPE-049422.pdf

Salinas, G. (2017). *ESTUDIO DE REDES DEFINIDAS POR SOFTWARE E IMPLEMENTACIÓN DE ESCENARIOS VIRTUALES DE PRUEBA*. Obtenido de https://www.dit.upm.es/~posgrado/doc/TFM/TFMs2016-2017/TFM_Gabriela_Salinas_Jardon_2017.pdf

Sánchez, A. & López, V. (Octubre de 2017). *Estudio y Despliegue de Redes Definidas por Software*. Obtenido de http://tauja.ujaen.es/bitstream/10953.1/6704/1/TFG_Angela_Maria_Sanchez_Valdepeas_Lopez.pdf

Sandoval, C. (2018). *Implementación de un clúster-controlador de SDN basado en un framework de software libre para la infraestructura Cloud de la facultad de ingeniería en Ciencias Aplicadas*. Obtenido de http://repositorio.utn.edu.ec/handle/123456789/7986

Sayali, P. (2018). *Load Balancing Approach for Finding Best Path in SDN.*

Sayans, P. (Junio de 2018). *Universidad Politécnica de Madrid.* Obtenido de http://oa.upm.es/51526/1/TFG_PABLO_SAYAUS_COBOS.pdf

Serrano, D. (2015). *Redes Definidas por Software (SDN): OpenFlow*. Obtenido de https://riunet.upv.es/bitstream/handle/10251/62801/SERRANO%20-%20Redes%20Definidas%20por%20Software%20%28SDN%29%3A%20OpenFlow.pdf?sequence=3&isAllowed=y

SoftSecurity. (2015). *BALANCEO DE CARGAS*. Obtenido de http://www.softsecuritycorp.com/index.php/soluciones-y-productos/proteccion-de-redes-y-aplicaciones/balanceo-de-cargas

Stallman, R. (12 de Enero de 2005). *Software libre para una sociedad libre*. Obtenido de https://www.gnu.org/philosophy/fsfs/free_software.es.pdf

Tecno, D. (07 de Diciembre de 2018). *ONU: Unos 4.000 millones de personas usan internet.* Obtenido de https://www.eluniverso.com/larevista/2018/12/07/nota/7087710/onu-4000-millones-personas-usan-internet

Telectrónika. (Abril de 2018). *GNS3 Guía Introductoria: Características y Requerimientos Mínimos*. Obtenido de https://telectronika.com/articulos/ti/que-es-gns3/

Umme, Z. & Hanene B. (23 de Noviembre de 2017). *Dynamic Load Balancing in SDN-Based Data Center Networks.* Obtenido de https://ieeexplore.ieee.org/document/8117206

Wang, S. & Chao, C. (2017). *EstiNet OpenFlow Network Simulator and Emulator*. Obtenido de https://ieeexplore.ieee.org/document/6588659

Wireshark. (2020). *About Wireshark*. Obtenido de https://www.wireshark.org/#learnWS

Xataka. (9 de Julio de 2019). *Xataka Basics*. Obtenido de https://www.xataka.com/basics/que-son-el-ping-y-la-latencia-y-por-que-no-solo-importa-la-velocidad-en-tu-conexion

Ya-bin, X. & Chen-Xiao, C. (2016). Research on Load Balance Method in SDN. *International Journal of Grid and Distributed Computing* , 25-26.

Yagues, P. (2015). *Programación de redes SDN mediante el controlador POX.* Cartagena: Universidad Politécnica de Cratagena.

Anexos

MANUAL DE USUARIO

ALGORITMO DE BALANCEO DE CARGA DE MÚLTIPLES RUTAS EN UNA RED SDN

El siguiente documento pretende indicar un breve procedimiento, para que cualquier usuario con conocimientos mínimos de redes y sistemas TIC, pueda utilizar el algoritmo de optimización de enrutamiento en cualquier Red Definida por Software (SDN) que se requiera.

Para este propósito es importante mencionar que el algoritmo de balanceo de carga desarrollado ha utilizado los siguientes elementos básicos:

El controlador RYU, que es una infraestructura que sienta su base en componentes para redes SDN, y cuyos componentes proveen una API bien definida que facilita la creación de nuevas aplicaciones de control por parte de los desarrolladores; entre otras cosas, soporta protocolos para la administración de dispositivos como el OpenFlow.

El emulador MiniNet, que se utiliza para desplegar redes sobre los recursos muy limitados que posee un simple y sencillo ordenador físico o virtual. En base al Kernel de Linux y otros recursos adicionales, MiniNet puede emular algunos elementos de la red SDN como son: el controlador, los switches OpenFlow y los hosts.

El Protocolo OpenFlow una red puede gestionarse como un todo y permite al propio servidor decirle a los switches de la red a dónde enviar los paquetes; así las decisiones sobre enrutamiento de paquetes están centralizada y la red se puede programar independientemente de los switches.

A continuación, se procede a indicar los requerimientos mínimos, el proceso de instalación y ejecución de los diferentes componentes paso a paso, hasta la aplicación del algoritmo de balanceo de carga desarrollado.

Requerimientos mínimos:

Se requiere una PC con las siguientes características mínimas para que realice las veces de controlador RYU:

- Procesador Intel Core 2 Duo de 2.93Ghz.
- Memoria RAM 2 Gb.
- Gráficos Intel G41 x86/MMX/SE2
- Almacenamiento 500 Gb
- Sistema Operativo GNU/Linux Ubuntu 20.04

Paso 1: Descarga e Instalación del controlador RYU

El sitio oficial de RYU es el siguiente: *https://ryu-sdn.org/*

Se descarga el controlador RYU del siguiente enlace: *https://github.com/osrg/ryu*

Una vez descargado RYU, se continúa con la instalación; para lo cual existen dos maneras:

La primera y más sencilla, desde el administrador, ejecutando

```
$ pip install ryu
```

La segunda, desde el código fuente, ejecutando

```
$ git clone git://github.com/osrg/ryu.git
$ cd ryu
$ python ./setup.py install
```

La instalación puede tardar varios minutos dependiendo del servicio de internet disponible, pues los archivos que se requieren para la instalación son de tamaño moderado.

Al finalizar la instalación, es posible verificar el funcionamiento ejecutando una aplicación con Ryu:

```
$ cd ryu
$ ryu-manager –verbose ryu.app.example_switch_13.py
```

Si se muestra en la pantalla mensajes como los siguientes, entonces es indicación de una instalación exitosa:

```
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
```

Paso 2: Instalación del emulador MiniNet

Se puede encontrar una guía de instalación en el siguiente enlace: https://github.com/mininet/mininet/blob/master/INSTALL

Se procede con la instalación de Git: sudo apt-get install git

Existen tres opciones o alternativas para realizar la instalación del emulador MiniNet que son las siguientes: 1) Utilizando una imagen VM pre-creada, 2) Instalación sobre Ubuntu package, y 3) Instalación de origen desde la fuente.

1) La más sencilla, utilizando imagen VM pre-creada.

La manera más sencilla de instalar MiniNet es iniciar una de las imágenes de máquina virtual pre-creada desde http://mininet.org/

Luego bootear la imagen VM, realizar el log in, y seguir las instrucciones del sitio web de MiniNet.

Una ventaja de éste método es que no se mezcla con el sistema operativo actual, ni lo daña de ninguna manera. Aunque una sola instancia de MiniNet puede simular redes con múltiples controladores, únicamente una instancia de MiniNet puede ser ejecutada al mismo tiempo, y requiere acceso root en la máquina a ejecutarse. Por lo tanto, si se dispone de multiusuario, se debería considerar ejecutar MiniNet en una VM.

2) La segunda más sencilla, sobre Ubuntu package.

Para instalar MiniNet por sí solo (por ejemplo´mn´ y la API de Python) sobre Ubuntu 16.04+, se ejecuta la siguiente instrucción:

```
sudo apt-get install mininet
```

Nota: Esta es una manera anterior de instalar MiniNet, la cual puede no soportar Python3. Si se requiere la última versión de MiniNet se debe considerar hacerlo como se describe en la siguiente sección.

3) Instalación de origen desde la fuente

Si está en otro directorio, primero:

```
cd ~  # si está en otro directorio
```

El comando para descargar MiniNet desde el código fuente es:

```
git clone git://github.com/mininet/mininet
```

Si se quiere ejecutar la última versión etiquetada, se utiliza la siguiente instrucción:

```
cd mininet
git tag  # lista versiones disponibles
```

Y luego:

```
git checkout <release tag>  # en donde release tag es la version a revisarse
```

Instalación default recomendada que incluye todos los componentes:

```
cd ..
mininet/util/install.sh -a  # instalación default recomendada, incluye todos los componentes
```

Normalmente, Python viene instalado de manera predeterminada; sin embargo, podría ser necesario instalar algunos módulos de Python compatibles, para éste propósito se ejecuta lo siguiente:

```
cd ~/ryu
sudo apt-get install python-dev python-pip python-setuptools
sudo pip install .
```

Así, se ejecutará automáticamente setup.py, que buscará los módulos de Python faltantes. Aunque el script instalará los módulos relevantes, se pueden ejecutar las siguientes instrucciones para evitar la flata de módulos más adelante:

```
sudo pip install webob
sudo pip install eventlet
sudo pip install paramiko
sudo pip install routes
```

Paso 3. Iniciando el servicio de controlador en MiniNet

Se arranca MiniNet para iniciar el controlador SDN con la siguiente instrucción:

```
sudo python aquí/controladorSDN.py +
```

Se procede a crear y configurar los hosts y switches (topología) conforme a la red a la cual se le va a aplicar el algoritmo de optimización de rutas, en éste ejemplo 3 hosts y un switch:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

Se realiza el test entre hosts y switchs; así como el ancho de banda disponible en las rutas:

```
pingall
iperf h1…hn
```

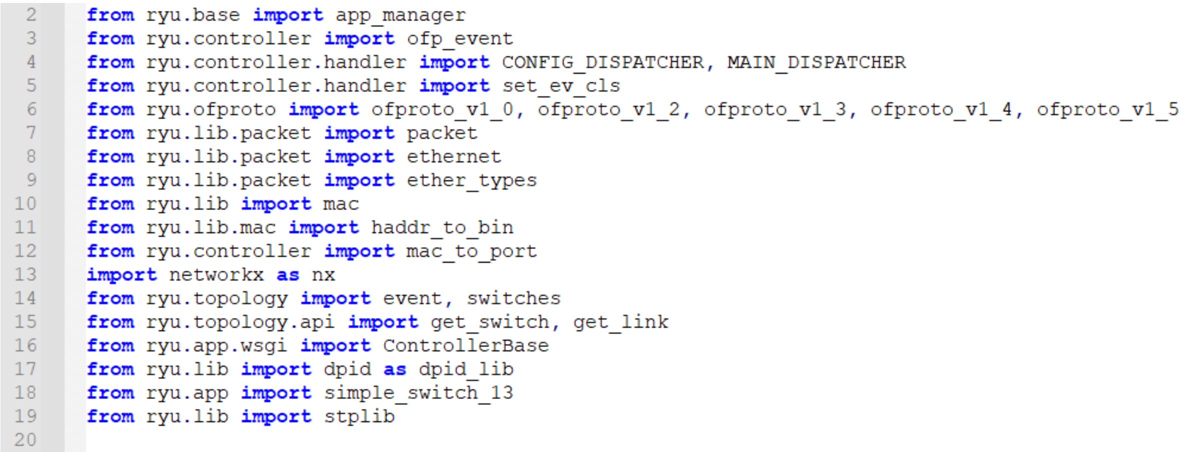
Paso 4. Iniciando el servicio de optimización de enrutamiento

Al mismo tiempo se abre otra ventana de terminal para ejecutar RYU, y se ejecuta la aplicación creada <<failover>> y carga de librerías:

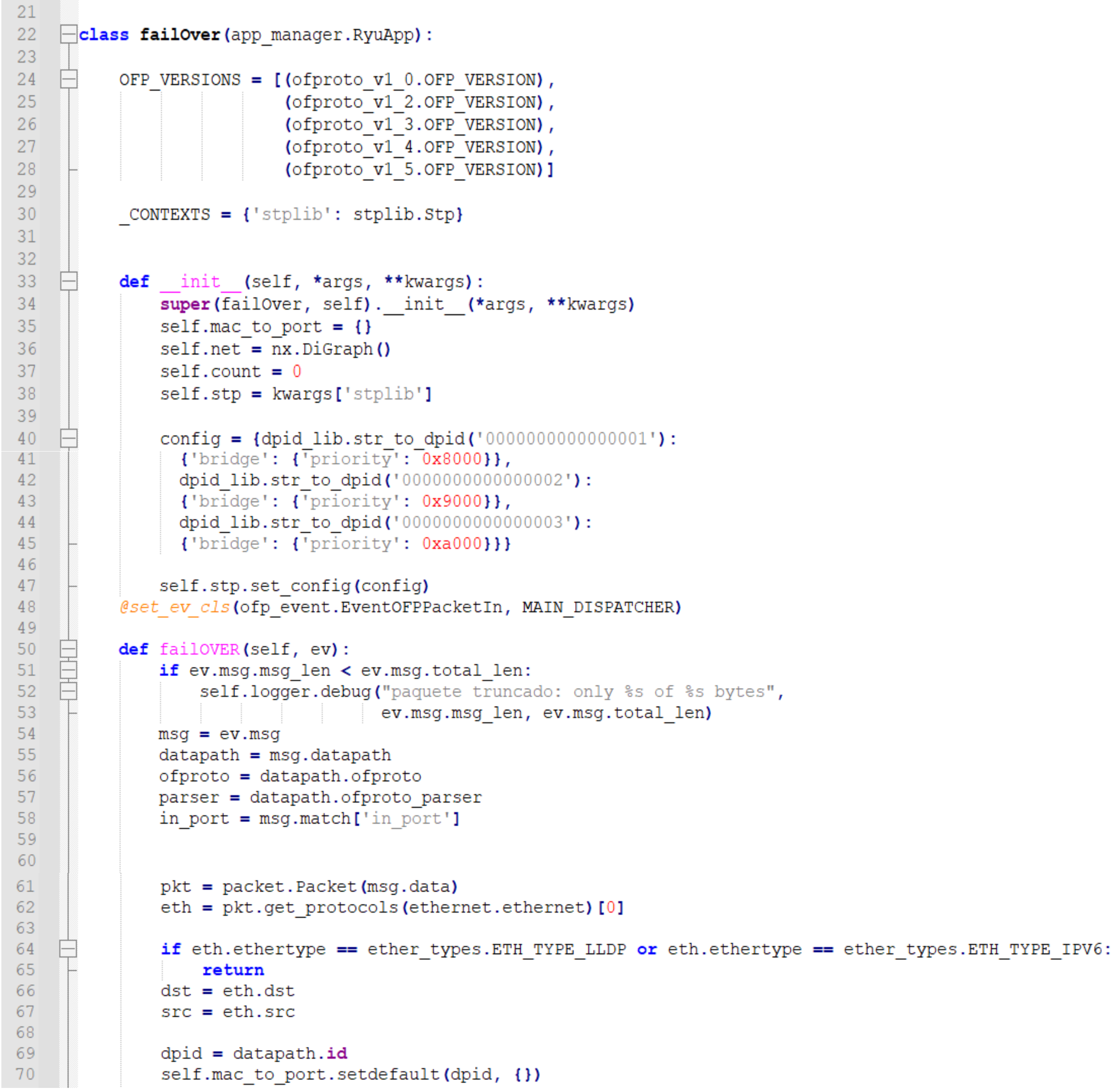
```
sudo ryu-manager aquí/failover.py
```

Finalmente se puede ir verificando los resultados del algoritmo desarrollado para encontrar las rutas más óptimas para el envío de paquetes entre uno y otro host.

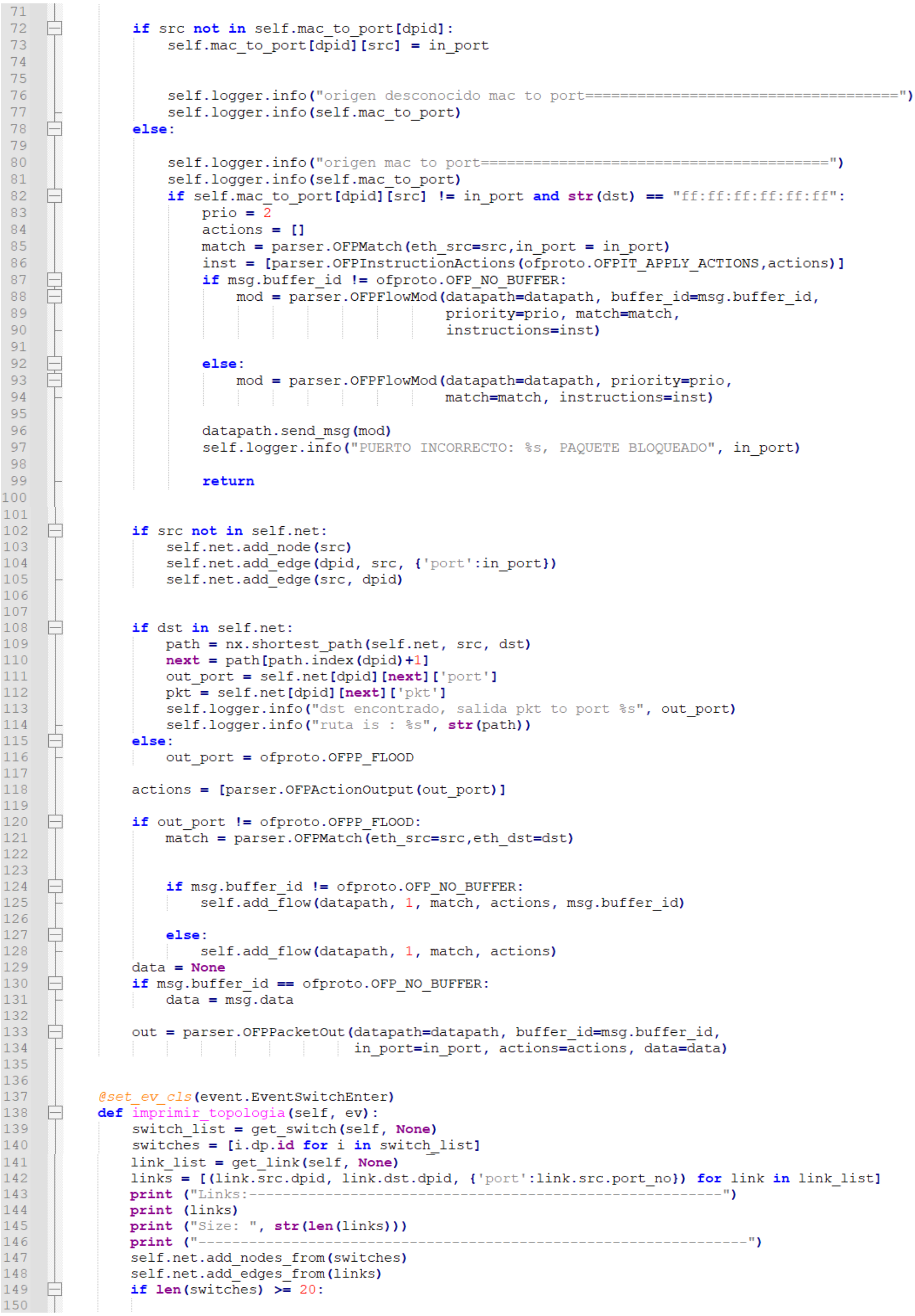
CÓDIGO DEL ALGORITMO DE BALANCEO DE CARGA DINÁMICO DE MÚLTIPLES RUTAS



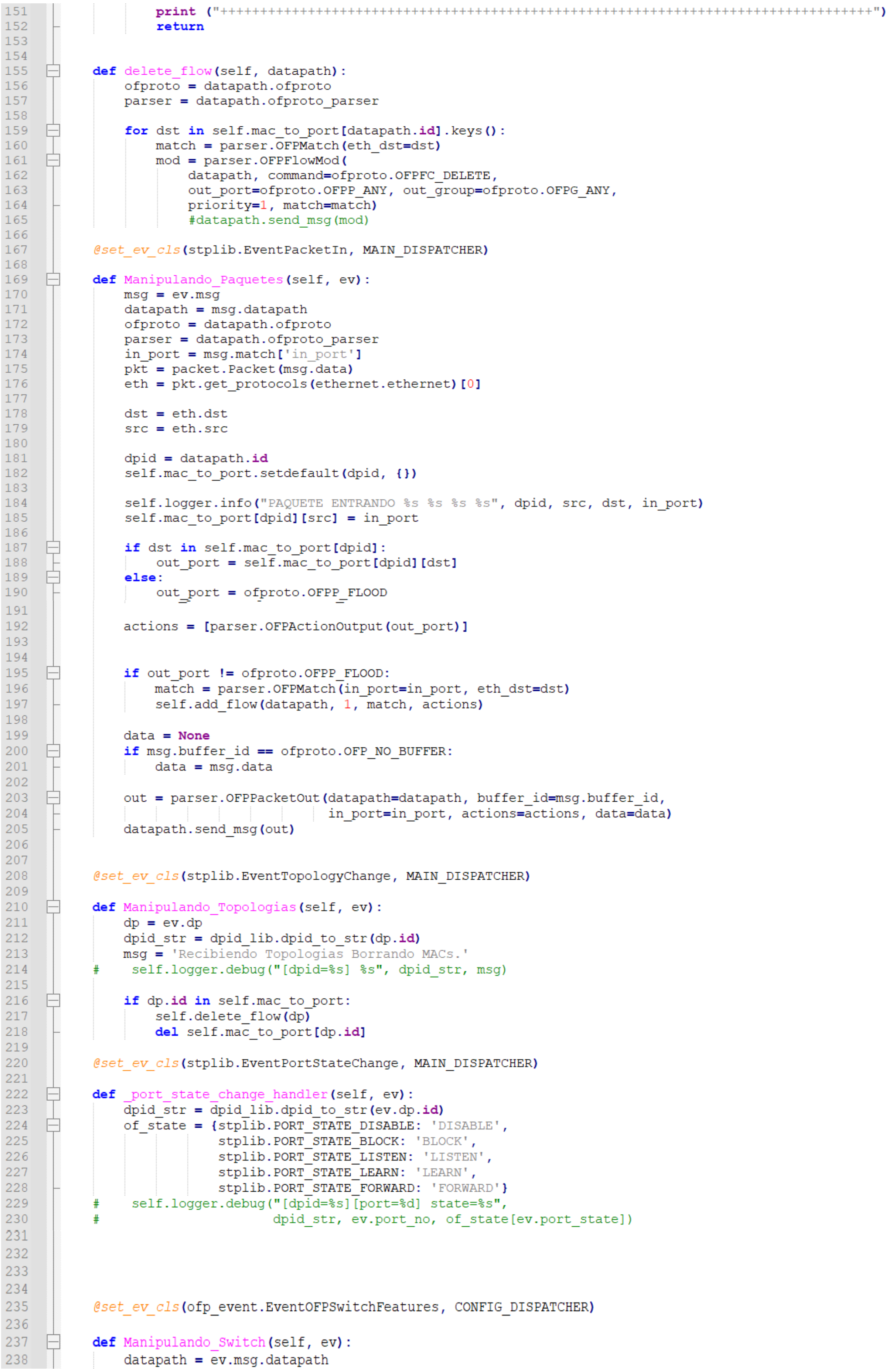
```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0, ofproto_v1_2, ofproto_v1_3, ofproto_v1_4, ofproto_v1_5
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.lib import mac
from ryu.lib.mac import haddr_to_bin
from ryu.controller import mac_to_port
import networkx as nx
from ryu.topology import event, switches
from ryu.topology.api import get_switch, get_link
from ryu.app.wsgi import ControllerBase
from ryu.lib import dpid as dpid_lib
from ryu.app import simple_switch_13
from ryu.lib import stplib

```



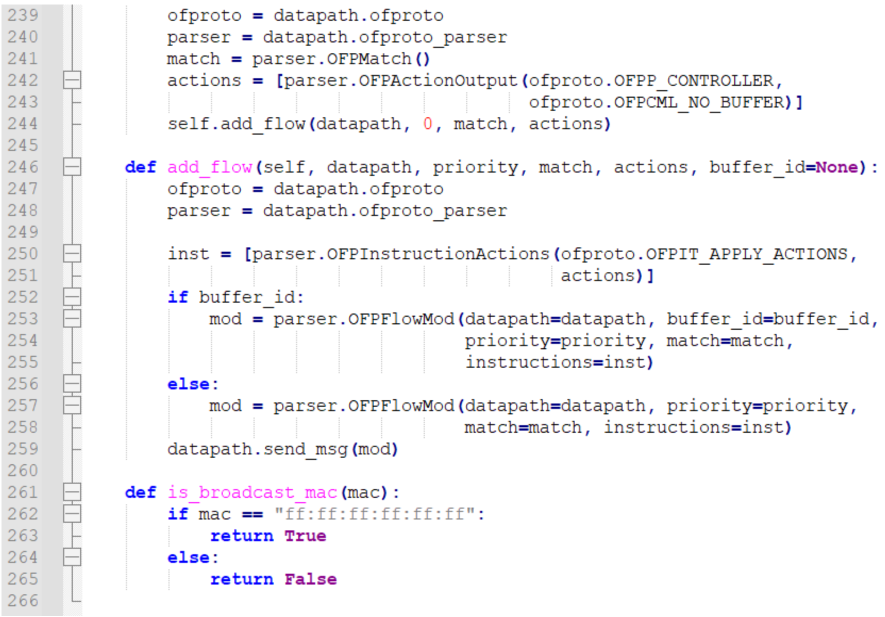
```python

class failOver(app_manager.RyuApp):

    OFP_VERSIONS = [(ofproto_v1_0.OFP_VERSION),
                    (ofproto_v1_2.OFP_VERSION),
                    (ofproto_v1_3.OFP_VERSION),
                    (ofproto_v1_4.OFP_VERSION),
                    (ofproto_v1_5.OFP_VERSION)]

    _CONTEXTS = {'stplib': stplib.Stp}


    def __init__(self, *args, **kwargs):
        super(failOver, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.net = nx.DiGraph()
        self.count = 0
        self.stp = kwargs['stplib']

        config = {dpid_lib.str_to_dpid('0000000000000001'):
          {'bridge': {'priority': 0x8000}},
          dpid_lib.str_to_dpid('0000000000000002'):
          {'bridge': {'priority': 0x9000}},
          dpid_lib.str_to_dpid('0000000000000003'):
          {'bridge': {'priority': 0xa000}}}

        self.stp.set_config(config)
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)

    def failOVER(self, ev):
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("paquete truncado: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']


        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        if eth.ethertype == ether_types.ETH_TYPE_LLDP or eth.ethertype == ether_types.ETH_TYPE_IPV6:
            return
        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})
```



```

        if src not in self.mac_to_port[dpid]:
            self.mac_to_port[dpid][src] = in_port


            self.logger.info("origen desconocido mac to port====================================")
            self.logger.info(self.mac_to_port)
        else:

            self.logger.info("origen mac to port========================================")
            self.logger.info(self.mac_to_port)
            if self.mac_to_port[dpid][src] != in_port and str(dst) == "ff:ff:ff:ff:ff:ff":
                prio = 2
                actions = []
                match = parser.OFPMatch(eth_src=src,in_port = in_port)
                inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
                if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                    mod = parser.OFPFlowMod(datapath=datapath, buffer_id=msg.buffer_id,
                                            priority=prio, match=match,
                                            instructions=inst)

                else:
                    mod = parser.OFPFlowMod(datapath=datapath, priority=prio,
                                            match=match, instructions=inst)

                datapath.send_msg(mod)
                self.logger.info("PUERTO INCORRECTO: %s, PAQUETE BLOQUEADO", in_port)

                return


        if src not in self.net:
            self.net.add_node(src)
            self.net.add_edge(dpid, src, {'port':in_port})
            self.net.add_edge(src, dpid)


        if dst in self.net:
            path = nx.shortest_path(self.net, src, dst)
            next = path[path.index(dpid)+1]
            out_port = self.net[dpid][next]['port']
            pkt = self.net[dpid][next]['pkt']
            self.logger.info("dst encontrado, salida pkt to port %s", out_port)
            self.logger.info("ruta is : %s", str(path))
        else:
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(eth_src=src,eth_dst=dst)


            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                self.add_flow(datapath, 1, match, actions, msg.buffer_id)

            else:
                self.add_flow(datapath, 1, match, actions)
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)


    @set_ev_cls(event.EventSwitchEnter)
    def imprimir_topologia(self, ev):
        switch_list = get_switch(self, None)
        switches = [i.dp.id for i in switch_list]
        link_list = get_link(self, None)
        links = [(link.src.dpid, link.dst.dpid, {'port':link.src.port_no}) for link in link_list]
        print ("Links:-------------------------------------------------------")
        print (links)
        print ("Size: ", str(len(links)))
        print ("---------------------------------------------------------------")
        self.net.add_nodes_from(switches)
        self.net.add_edges_from(links)
        if len(switches) >= 20:

```



```
        print ("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")
        return


def delete_flow(self, datapath):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    for dst in self.mac_to_port[datapath.id].keys():
        match = parser.OFPMatch(eth_dst=dst)
        mod = parser.OFPFlowMod(
            datapath, command=ofproto.OFPFC_DELETE,
            out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY,
            priority=1, match=match)
            #datapath.send_msg(mod)

@set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)

def Manipulando_Paquetes(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("PAQUETE ENTRANDO %s %s %s %s", dpid, src, dst, in_port)
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]


    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)


@set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)

def Manipulando_Topologias(self, ev):
    dp = ev.dp
    dpid_str = dpid_lib.dpid_to_str(dp.id)
    msg = 'Recibiendo Topologias Borrando MACs.'
#    self.logger.debug("[dpid=%s] %s", dpid_str, msg)

    if dp.id in self.mac_to_port:
        self.delete_flow(dp)
        del self.mac_to_port[dp.id]

@set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)

def _port_state_change_handler(self, ev):
    dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
    of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                stplib.PORT_STATE_BLOCK: 'BLOCK',
                stplib.PORT_STATE_LISTEN: 'LISTEN',
                stplib.PORT_STATE_LEARN: 'LEARN',
                stplib.PORT_STATE_FORWARD: 'FORWARD'}
#    self.logger.debug("[dpid=%s][port=%d] state=%s",
#                      dpid_str, ev.port_no, of_state[ev.port_state])




@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)

def Manipulando_Switch(self, ev):
    datapath = ev.msg.datapath
```



```
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)

    def is_broadcast_mac(mac):
        if mac == "ff:ff:ff:ff:ff:ff":
            return True
        else:
            return False

```

CÓDIGO DEL EMULADOR DE MININET ESCARIO DE PRUEBA

Desarrollo de scripts en lenguaje de programación para mininet

En las siguientes líneas de código, se pone en consideración el desarrollo de los escenarios de estudio, es decir, la creación de los diferentes equipos, conexiones y parámetros que rige la arquitectura de red SDN a valorar.

Las líneas presentadas a continuación son librerías que se pueden importan con ayuda de repositorios. Estas líneas de código, permiten al desarrollador inicializar todos los parámetros para el levantamiento de equipos virtualizados como son: Openvswitch, enlaces de comunicación entre los equipos de la red (giga-ethernet, fast-ethernet), equipos terminales y la comunicación entre los nodos y el controlador SDN RYU.

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController, OVSKernelSwitch, UserSwitch
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.link import TCLink, Intf, Link
from subprocess import call
from mininet.link import TCIntf
from mininet.util import custom
from mininet.topo import Topo
```

Una vez realizada la importación de las librerías, se continúa con la iniciación de la variable *red,* con el propósito de declarar la sincronización del emulador de red Mininet con el controlador SDN remoto RYU, esto debido a que, por defecto Mininet puede crear un controlador SDN para funciones de prueba. Esta función de sincronización de Mininet y RYU se presenta en la siguiente línea de código.

```
red1 = Mininet( controller=RemoteController, link=TCLink, switch=OVSKernelSwitch )
```

Posteriormente, se procede a la creación de los equipos terminales "*host - h*", con su respectiva dirección MAC y direccionamiento IP, que para este caso en particular, se consideran las redes 192.168.1.X/24. Estos parámetros eventualmente permitirán la comunicación y pruebas de conectividad de la red SDN, la cual se aplicará el algoritmo de balanceo de carga.

```
h1 = red1.addHost( 'h1', mac = '10:00:00:00:00:01', ip='192.168.1.1/24')
h2 = red1.addHost( 'h2', mac = '20:00:00:00:00:02', ip='192.168.1.2/24')
h3 = red1.addHost( 'h3', mac = '30:00:00:00:00:03', ip='192.168.1.3/24')
h4 = red1.addHost( 'h4', mac = '40:00:00:00:00:04', ip='192.168.1.4/24')
```

De la misma forma, en las líneas siguientes se procede a la inicialización de los equipos de red, es decir los equipos de red que permitirán el enrutamiento de los paquetes dentro de la red SDN, para estos equipos se considera la variable *s* (switch)

```
s1 = red1.addSwitch('s1')
s2 = red1.addSwitch('s2')
s3 = red1.addSwitch('s3')
s4 = red1.addSwitch('s4')
```

A continuación, se procede a declarar la comunicación del controlador SDN y la red emulada por Mininet, mediante la siguiente línea de código, en la cual se considera la IP local 127.0.0.1, puesto que tanto el emulador y el controlador se ejecutan dentro de la misma máquina virtual, solamente se toma en cuenta el puerto de comunicación entre RYU y Mininet que es el puerto 6633

```
ctr1 = red1.addController('crt1', controller = RemoteController, defaultIP ='127.0.0.1',port=6633)
```

Para la creación de los enlaces entre los equipos de red se procede a declarar las siguientes líneas de código, en las que se especifica el puerto de red de cada uno de los equipos terminales o hosts y cada uno de los OpenvSwitches que se utilizarán para la conexión física y de igual manera para la conexión entre todo los equipos que forman parte de la red.

En estas líneas además se establecen la capacidad de ancho de banda y el retardo que existe en cada uno de los posibles enlaces, con el objetivo de brindar toda la información al algoritmo que posteriormente determinará la detección de la ruta más óptima y el balanceo de carga.

```
red1.addLink(h1, s1, intfName1='h1-eth0', intfName2='s1-eth1',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(h2, s2, intfName1='h2-eth0', intfName2='s2-eth1',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(h3, s3, intfName1='h3-eth0', intfName2='s3-eth1',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(h4, s4, intfName1='h4-eth0', intfName2='s4-eth1',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(s1, s2, intfName1='s1-eth2', intfName2='s2-eth2',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(s2, s3, intfName1='s2-eth3', intfName2='s3-eth2',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(s3, s4, intfName1='s3-eth3', intfName2='s4-eth2',bw=1000, delay='1ms',max_queue_size=1000)
red1.addLink(s1, s3, intfName1='s1-eth3', intfName2='s3-eth4',bw=100, delay='1ms',max_queue_size=1000)
red1.addLink(s1, s4, intfName1='s1-eth4', intfName2='s4-eth3',bw=100, delay='1ms',max_queue_size=1000)
red1.addLink(s4, s2, intfName1='s4-eth4', intfName2='s2-eth4',bw=100, delay='1ms',max_queue_size=1000)
```

Posteriormente, se procede a la construcción e inicialización de la Red y los Switches para realizar el arranque global de lo que se va a considerar como el ambiente o escenario de trabajo sobre el que se aplicará el algoritmo. Las variables *red1, s1, s2, s3, s4* corresponden a la red y a los equipos a los que va a dar inicio dentro del emulador Mininet y el comando *CLI* para desplegar una ventana terminal para el control de los equipos.

```
red1.build()
ctr1.start
s1.start([ctr1])
s2.start([ctr1])
s3.start([ctr1])
s4.start([ctr1])
CLI( red1 )
```