UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Implementación SDN de Arquitecturas Spine & Leaf Multidimensionales

TRABAJO FIN DE GRADO

Diciembre - 2024

AUTOR: Tomas Ambrazavicius

DIRECTORES: Salvador Alcaraz Carrasco

Pedro Juan Roig Roig

AGRADECIMIENTOS

Agradezco a mi familia que estuvieron en mi lado a lo largo del camino.

Agradezco a mis compañeros, principalmente a Amine, Fernando, Fran por las innumerables horas que lo hemos pasado en la biblioteca, por todos los exámenes finales que nos hemos ayudado mutuamente. Sin ellos la historia pudo haber sido distinta.

Agradezco a aquellos profesores que apretaban, pero no ahogaban.

Agradezco a mis tutores Salvador y Pedro por darme la oportunidad de desarrollar este trabajo y por su ayuda.

Y no puedo olvidar a Potter, mi fiel compañero de cuatro patas, un majestuoso lobo que con su presencia noble ha sido mi apoyo incondicional.

"Auribus tenere lupum"

(Coge al lobo por las orejas)

–Publius Terentius-

RESUMEN

Este Trabajo de Fin de Grado consiste en modelar una topología de switches Spine-Leaf multidimensionales en un entorno SDN (Software Defined Network), mediante el emulador Mininet-WiFi. El objetivo principal es explorar y analizar esta topología, ampliamente utilizada en centros de datos modernos, para ello utilizando el lenguaje de programación Python. El proyecto desarrolla la configuración, implementación y pruebas de rendimiento automatizadas.

Como el entorno es SDN, requerimos de un controlador, inicialmente se utilizó "ONOS", pero debido a problemas técnicos, se migro a "OpenDaylight". Posteriormente se implementó un algoritmo de prevención de bucles, que elimino la dependencia con el controlador. El trabajo se realizó en un entorno Linux, inicialmente con una máquina virtual, posteriormente se migro a una partición.

Para analizar los parámetros de rendimiento de la topología, se desarrolló distintos scripts de testeo que automatizan las pruebas de rendimiento, permiten personalizar la topología y la prueba en cuestión. Para ello se creó distintos scripts auxiliares que se encargar de interactuar con los distintos entornos, como el emulador Mininet-WiFi, el protocolo OpenFlw, la entrada de datos mediante los argumentos de consola y la salida de datos en forma de gráfica. Dichos scripts auxiliares son utilizados por los scripts de testeo.

Durante las pruebas en las distintas configuraciones de Spine-Leaf, se dedujo que los controladores usaban el algoritmo Spanning Tree Protocol (STP) para topologías con bucles, dicho algoritmo consiste en bloquear los enlaces redundantes. Para aprovechar la topología Spine-Leaf se implementó el algoritmo Equal-Cost Multi-Path (ECMP), permitiendo aprovechar los enlaces redundantes, y por ello, mejorar el ancho de banda y latencia. Se realizo la implementación del algoritmo ECMP mediante un script auxiliar que automatizada la definición de los flujos necesarios mediante comandos del protocolo OpenFlow, de esta manera podemos emular la topología Spine-Leaf sin controlador.

Al implementar algoritmo ECMP, se enfocó en pruebas de ancho de banda con distintos tipos de flujos, revelando que en flujo único no se obtenía mejoras, pero en flujo múltiple y distribuido incrementaba el ancho de banda, pero en cambio mostraba cierta inestabilidad en los resultados, debido a que el algoritmo ECMP designada los flujos de manera aleatoria a los enlaces. Lo que podía provocar con pocos flujos, saturación de enlaces. Al aumentar el número de flujos, el tráfico se distribuiría de manera más uniforme, incrementando la estabilidad.

Al experimental con ECMP, se descubrió la utilización de cada capa de la topología, el número de switches en la capa Spine definía el ancho de banda máximo, mientras que los switches de la capa Leaf permitieran acceso a un mayor número de dispositivos a la red.

Este trabajo demuestra las mejoras que introduce el algoritmo ECMP teniendo como referente el algoritmo STP, utilizado por los controladores. También demuestra la utilidad del entorno SDN, ya que permite implementar algoritmos que no están en los switches convencionales, ya que son sistemas cerrados.

Palabras clave: SDN, OpenFlow, Mininet, Mininet-WiFi, Spine-Leaf, ECMP, Python, automatización, latencia, ancho de banda, flujos de datos, controlador SDN, OpenDaylight.

ABSTRACT

This Final Degree Project involves modeling a multidimensional Spine-Leaf switch topology in a Software Defined Network (SDN) environment, using the Mininet-WiFi emulator. The main objective is to explore and analyze this topology, widely used in modern data centers, utilizing the Python programming language. The project develops automated configuration, implementation, and performance testing.

As the environment is SDN-based, a controller is required. Initially, "ONOS" was used, but due to technical issues, it was migrated to "OpenDaylight". The work was carried out in a Linux environment, initially using a virtual machine, and later migrating to a partition to optimize computational resources.

To analyze the topology's performance parameters, various testing scripts were developed that automate performance tests and allow customization of both the topology and the specific test. For this purpose, several auxiliary scripts were created to interact with different environments, such as the Mininet-WiFi emulator, the OpenFlow protocol, data input through console arguments, and data output in the form of graphs. These auxiliary scripts are utilized by the main testing scripts.

During tests on various Spine-Leaf configurations, it was deduced that the controllers used the Spanning Tree Protocol (STP) algorithm for topologies with loops, which blocks redundant links. To leverage the Spine-Leaf topology, the Equal-Cost Multi-Path (ECMP) algorithm was implemented, allowing the use of redundant links, thereby improving bandwidth and latency. The ECMP implementation was achieved through an auxiliary script that automated the definition of necessary flows using OpenFlow protocol commands, enabling the emulation of the Spine-Leaf topology without a controller, as the script acted as one.

Upon implementing ECMP, the focus shifted to bandwidth tests with different types of flows, revealing that single-flow tests showed no improvements, but multiple and distributed flows increased bandwidth. However, this also demonstrated some instability in the results, as the ECMP algorithm assigned flows to links randomly.

Experimenting with ECMP revealed the utilization of each layer in the topology: the number of switches in the Spine layer defined the maximum bandwidth, while the switches in the Leaf layer allowed access to a greater number of devices in the network.

This work demonstrates the improvements introduced by the ECMP algorithm, using the STP algorithm (employed by controllers) as a reference. It also showcases the utility of the SDN environment, as it allows the implementation of algorithms that are not present in conventional switches due to their closed-system nature.

Keywords: SDN, OpenFlow, Mininet, Mininet-WiFi, Spine-Leaf, ECMP, Python, automation, latency, bandwidth, data flows, SDN controller, OpenDaylight.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ABREVIATURAS

- ARP (Address Resolution Protocol, Protocolo de resolución de direcciones)
- BW (Bandwidth, Ancho de banda)
- CLI (Command Line Interface, Interfaz de línea de comandos)
- ECMP (Equal Cost Multi-Path, Múltiples rutas de igual costo)
- IoT (Internet of Things, Internet de las Cosas)
- IP (Internet Protocol, Protocolo de Internet)
- ISO (Disk Image File, Archivo de imagen de disco)
- MAC (Media Access Control, Control de acceso al medio)
- mn (Mininet)
- ODL (Controlador SDN OpenDayLight)
- ONOS (Open Network Operating System, Sistema Operativo de Red Abierta)
- OSI (Open Systems Interconnection, Modelo de interconexión de sistemas abiertos)
- .py (Extensión de los archivos de Python)
- SDN (Software Defined Networks, Redes definidas por software)
- STP (Spanning Tree Protocol, Protocolo de Árbol de expansión)
- TCP (Transmission Control Protocol, Protocolo de control de transmisiones)
- UDP (User Datagram Protocol, Protocolo de datagramas de usuario)
- UTP (Unshielded twisted pair, par trenzado no blindado)

1 INTRODUCCIÓN

1.1 PRESENTACIÓN DEL TEMA Y JUSTIFICACIÓN DE SU RELEVANCIA.

La adaptación masiva a los dispositivos IoT (Internet de las cosas) ha promocionado un cambio en la computación remota. Los dispositivos IoT, tienen limitaciones de potencia de cómputo y memoria, por ello tienen la necesidad de enviar los datos a servidores externos. Por ello es necesario establecer centros de datos al borde de la red, obteniendo unas latencias y ancho de bandas aceptables, donde la topología de red es fundamental.

En este contexto, la topología Spine-Leaf es una opción ideal ya que permite obtener unas tasas de latencias menores y un ancho de banda grande entre nodos. De esta manera permite procesar y almacenar los datos generados por los dispositivos IoT.

La implementación de las arquitecturas SDN (Software-Defined Networking) en la topología Spine-Leaf multidimensionales presenta una evolución en los centros de datos. Tradicionalmente las bases de datos utilizaban una topología de tres capas, que eran la capa de acceso, la capa de agregación y la capa de núcleo. Este diseño esta optimizado para el tráfico de cliente a servidor (tráfico vertical), pero presenta inconvenientes como la escalabilidad y la eficiencia en el tráfico de servidor a servidor (tráfico horizontal).

En cambio, la topología Spine-Leaf solo está formado por dos capas, la capa de acceso llamada “leaf” y la capa “Spine”, obteniendo una comunicación entre dispositivos más corta, ya que hay menos enlaces entre ellos. Obteniendo una latencia más baja y un ancho de banda mayor.

La implementación del entorno SDN permite una gestión centralizada de todos los switches que forma parte de la topología Spine-Leaf. Esto es fundamental ya que nos permite introducir programación a los componentes de la red, esto abre un mundo de posibilidades, como la implementación del algoritmo ECMP.

1.2 OBJETIVOS

El objetivo de este TFG es modelar la topología Spine-Leaf en un ambiente SDN (software defined network), mediante el emulador Mininet-WiFi y el uso del lenguaje de programación Python. Una vez conseguido, se estudiará la implementación de ciertos algoritmos para observar las mejoras producidas con respecto al modelo original.

Los objetivos concretos del proyecto son:

- Implementar la topología Spine-Leaf en entorno SDN, para ello utilizar el emulador Mininet-WiFi.
- Explorar la configuración de la topología. Generación automatizada de topologías Spine-Leaf de distintas configuraciones.
- El uso de containers que está íntimamente ligado a SDN. Explorar la implementación de los diferentes nodos a base de containers, en especial, Dockers.
- Implementar las medidas de rendimiento oportunas para obtener una visión de la red, desde el punto de vista de diferentes parámetros tales como latencia extremo a extremo, ancho de banda, flujos, etc.

- Implementar algoritmos que permitan mejorar los rendimientos de la topología Spine-Leaf.

1.3 ESTRUCTURA DE LA MEMORIA

El documento estará dividido en las siguientes secciones:

1. **Introducción**: Descripción general de TFG.
2. **Marco teórico**: Se explicará toda la teoría que es necesaria para la comprensión de las distintas pruebas.
3. **Herramientas**: Explicación del entorno y de las distintas herramientas utilizadas en el proyecto, también el uso de cada una y su instalación.
4. **Scripts implementados**: Sección que consiste en explicar los distintos scripts de Python que fueron hechos para testear las topologías.
5. **Marco practico**: Con las herramientas y los scripts, emulamos las distintas configuraciones de la topología y explicamos los resultados.
6. **Conclusiones**: Descripción general de la conclusión del proyecto.
7. **Anexo**: Código completo de los scripts.

2 MARCO TEÓRICO

2.1 REDES DEFINIDAS POR SOFTWARE

El software-defined networking (SDN) o redes definidas por software, es un concepto de red que permite gestionar y controlar de forma centralizada los distintos componentes (Switch, router, cortafuegos, etc) que forma parte de la red. [1] Es decir, desde un servidor llamado controlador SDN, se controlan a los distintos dispositivos que forma parte de la red.

Para controlar los dispositivos se emplean protocolos abiertos como **OpenFlow**, que nos permite acceder a los dispositivos de red, que no sería posible si fuese firmware propietario.

2.1.1 Arquitectura

SDN define una arquitectura que permite una gestión de los dispositivos de red de manera estrictamente basada en software.

Con este objetivo, la arquitectura se divide en tres planos:

- **Plano de Control**: Este plano se encargado de **determinar la ruta** que deben seguir los paquetes de datos para llegar a su destino. Es responsable de supervisar el tráfico de datos y por ello ha de llevar a cabo todos los análisis necesarios. Un ejemplo de su funcionamiento es la creación de tablas de enrutamiento, que son esenciales para la transmisión de datos.

  El plano de control opera generalmente en la Capa 3 (la capa de Red) y superiores en el modelo OSI (Modelo de interconexión de sistemas abiertos).

  - **Controlador SDN**: Para controlar a los distintos componentes de la red, se emplea el controlador SDN que se considera el núcleo de una red SDN y se encuentra en un servidor central, es una aplicación que se encarga de administrar, controlar la red, monitorización, etc.

- **Plano de Datos**: Este plano se encuentra en todos los dispositivos de red (routers, switches, cortafuegos, etc.) y su principal función es el **reenvío de paquetes**. Debido a esta función simplificada, los dispositivos requieren menos potencia de cálculo, lo que resulta en un firmware menos complejo y un costo más bajo.

  El plano de datos opera en los niveles más bajo de las capas OSI, generalmente en la Capa 2 (la capa de enlace de datos) y la Capa 3 (la capa de red).

- **Plano de aplicación**: Este plano es exclusivo de SDN, se encuentra en el controlador SDN, contiene todas las aplicaciones del controlador SDN, que tienen diferentes propósitos como algoritmos de enrutamiento. administración de red, seguridad de red o implementaciones de política.

  Opera en la capa 7 (Capa de aplicación) del modelo OSI. [2]

En la Figura 1, podemos ver cómo se relacionan las distintas capas del modelo SDN.

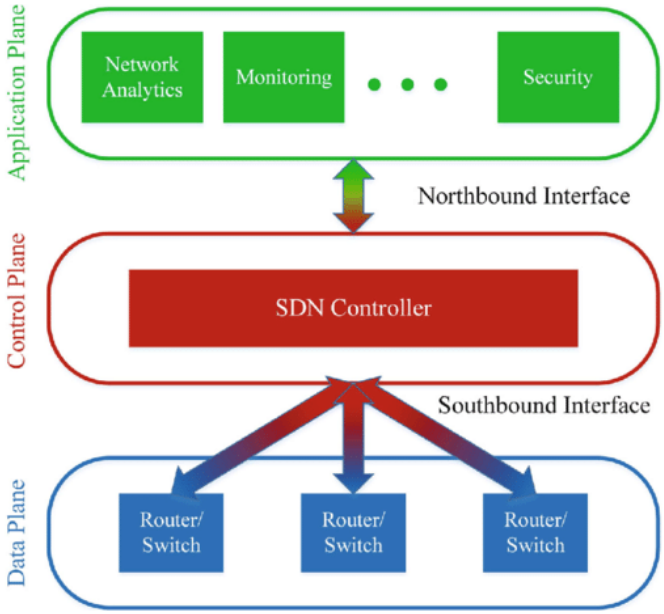


Figura 1: Distintos planos de SDN [3]

En la Figura 2, podemos observar donde se ubican los distintos planos de cada tipo de arquitectura, en la arquitectura tradicional tenemos dos planos, el plano de datos y control ubicados en los switches, en cambio en la arquitectura SDN, tenemos tres planos, solamente el plano de datos se encuentra en los switches, en cambio el plano de control y aplicación están en el controlador SDN. [4]

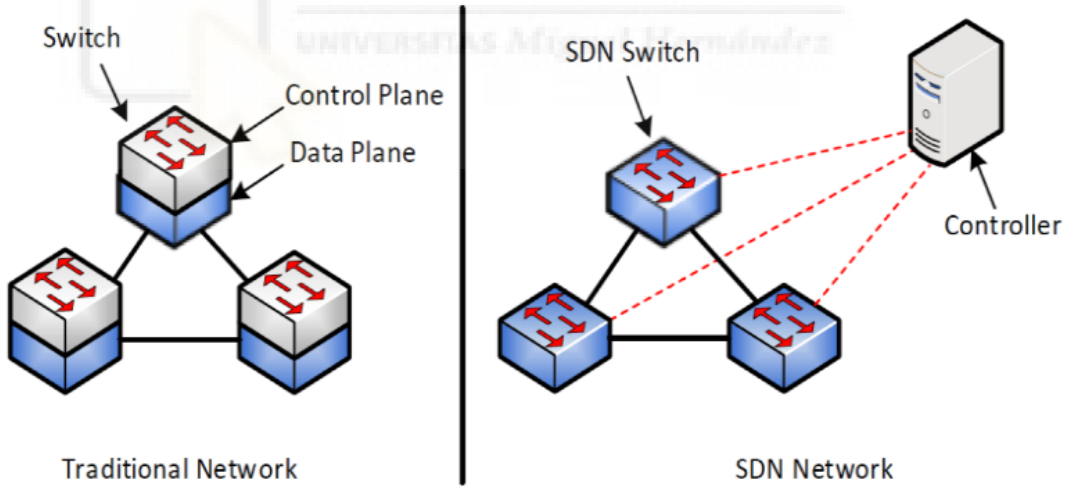


Figura 2: Arquitectura tradicional vs SDN [5]

2.1.2 Ventajas

Las redes SDN ofrecen varias ventajas sobre las arquitecturas de redes tradicionales, que incluyen: [6]

- **Control centralizado de la red**: SDN ofrece un controlador único que centraliza la gestión de la red. Esto facilita la definición y aplicación de políticas de red de manera más detallada, mejorando la seguridad, el rendimiento y la fiabilidad de la red.

- **Red programable**: Los dispositivos de red en un entorno SDN son programables y pueden ser reconfigurados en tiempo real para adaptarse a las necesidades cambiantes de la red. Esto permite una rápida adaptación de la red a los patrones de tráfico cambiantes, mejorando el rendimiento y la eficiencia de la red.
- **Ahorro de costos**: SDN permite el uso de hardware estándar para construir una red, lo que reduce el costo del hardware de la red. Además, la centralización del control de la red puede disminuir la necesidad de gestión manual de la red, generando ahorros en costos de mano de obra y mantenimiento.
- **Mejora de la seguridad de la red**: El control centralizado de la red en SDN facilita la detección y respuesta a las amenazas de seguridad. La implementación de políticas y reglas de red permite controles de seguridad que pueden mitigar los riesgos de seguridad.
- **Escalabilidad**: SDN facilita la expansión de la red para satisfacer las demandas cambiantes del tráfico. Con la capacidad de controlar la red mediante programación, los administradores pueden ajustar rápidamente la red para manejar más tráfico sin necesidad de intervención manual.
- **Simplificación de la gestión de la red**: SDN puede simplificar la gestión de la red al abstraer el hardware de red subyacente y presentar una vista lógica de la red a los administradores. Esto facilita la administración y la resolución de problemas de la red, mejorando el tiempo de actividad y la fiabilidad de la red

2.1.3 Inconvenientes

- **Complejidad**: Las redes SDN pueden presentar una mayor complejidad en comparación con las redes tradicionales debido a su sofisticación tecnológica y la necesidad de habilidades especializadas para su gestión.

- **Dependencia del controlador**: En una red SDN, el controlador centralizado es un componente esencial. Si esta falla, podría provocar la caída de toda la red. Por lo tanto, es crucial que las organizaciones garanticen la alta disponibilidad del controlador y tengan un plan de respaldo y recuperación ante desastres.

- **Compatibilidad**: Es posible que algunos dispositivos de red antiguos no sean compatibles con SDN. Esto podría requerir que las organizaciones reemplacen o actualicen estos dispositivos para maximizar los beneficios de SDN.

- **Seguridad**: Aunque SDN puede mejorar la seguridad de la red, también puede presentar nuevos riesgos de seguridad. Un punto de control único podría ser un blanco atractivo para los atacantes, y la capacidad de programar la red podría permitirles manipular el tráfico.

- **Rendimiento**: El control centralizado de la red en SDN puede introducir latencia, lo que podría afectar el rendimiento de la red en ciertas situaciones. Además, la sobrecarga del controlador SDN podría impactar el rendimiento de la red a medida que esta se expande.

2.2 OPENFLOW

OpenFlow es un protocolo de comunicación abierto y emergente que permite a un servidor determinar el camino de reenvío de paquetes en una red de switches. Para determinar el camino del paquete se introducen las tablas de flujo en los switches. También tiene la posibilidad de monitoreo y diagnóstico.[7]

Figura 3: Logo de OpenFlow [8]

Este protocolo es uno de los primeros estándares de las redes definidas por software (SDN) y define la comunicación entre el plano de control (controlador SDN) y el plano de datos que son los dispositivos de red, Figura 4.

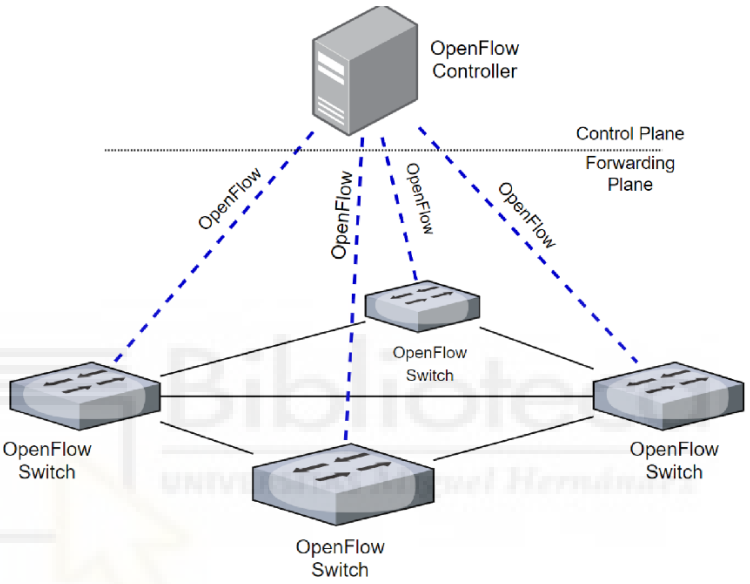


Figura 4: Diferenciación entre el plano de datos y control [8]

El funcionamiento de OpenFlow se basa en la comunicación bidireccional entre un Switch y su controlador. El controlador puede enviar comandos al switch y el switch puede enviar paquetes al controlador. El controlador gestiona al switch agregando, actualizando o eliminando entradas de flujo en las tablas de flujo del switch en concreto.

Estas **tablas de flujo** son componentes fundamentales del switch y almacenan las entradas de flujo que detallan cómo manejar diferentes tipos de tráfico de red.

Campos de la cabecera	Contadores	Acciones	Prioridad
If ingress port == 2		Drop packet	32768
if IP_addr == 129.79.1.1		re-write to 10.0.1.1, forward port 3	32768

Tabla 1: Flujo simplificado

Los componentes principales de la tabla de flujo son:

- **Campos de coincidencia (Match Fields)**: Estos son los campos de la cabecera del paquete que se utilizan para comparar con los paquetes entrantes. Algunos de los campos de coincidencia son: puerto, IP, Mac (Media Access Control), etc.
- **Contadores:** Realizan un seguimiento de los paquetes y bytes para cada flujo.
- **Acciones:** Dictan qué hacer con los paquetes que coinciden.
- **Prioridad:** En el caso que haya varias coincidencias en la tabla de flujos, se selecciona la entra con mayor prioridad.

Cuando un paquete llega a un switch, se verifica en las tablas de flujo. Si no hay coincidencia, se consulta al controlador SDN. Esto permite una gestión dinámica y flexible del tráfico de la red, lo que resulta en una mayor efectividad en el uso de los recursos de la red en comparación con una red convencional.

OpenFlow está diseñado para abordar la movilidad de máquinas virtuales (VM), redes con misiones críticas o redes NGN móviles (Red de siguiente generación).

En resumen, OpenFlow permite que las decisiones que implican el movimiento de paquetes estén centralizadas, lo que significa que la red puede ser programada independientemente de los switches. Esto otorga una flexibilidad, permitiendo programar y gestionar la red de manera centralizada. [9] [10] [11]

2.2.1 Versiones

Las distintas versiones que existen son las siguientes:[12]

1. **OpenFlow 1.0**: Esta fue la primera versión estándar del protocolo. Introdujo el concepto básico de tablas de flujo y la capacidad de modificar el comportamiento de los switches mediante reglas definidas por el controlador.
2. **OpenFlow 1.1**: Esta versión añadió soporte para múltiples tablas de flujo y grupos de acciones, permitiendo una mayor flexibilidad en la gestión del tráfico.
3. **OpenFlow 1.2**: Esta versión introdujo el soporte para extensiones experimentales y mejoró la capacidad de manipulación de paquetes.
4. **OpenFlow 1.3**: Es una de las versiones más utilizadas. Añadió soporte para contadores de flujo, tablas de metadatos y mejoras en la gestión de grupos.
5. **OpenFlow 1.4**: Esta versión mejoró la capacidad de monitoreo y diagnóstico, y añadió soporte para la sincronización de relojes entre el controlador y los switches.
6. **OpenFlow 1.5**: Es la versión más reciente y estable. Introdujo mejoras en la gestión de grupos y tablas, así como nuevas capacidades para la manipulación de paquetes.

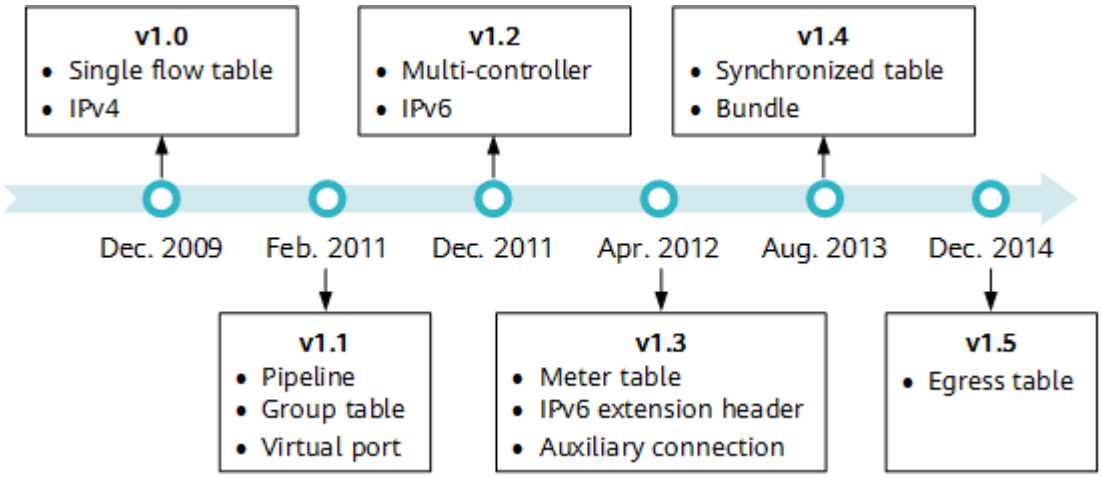


Figura 5: Evolución de las versiones OpenFlow [7]

Cabe mencionar que las más utilizadas son la versión 1.0 y 1.3. En nuestro caso utilizaremos la versión "1.3".

2.3 TOPOLOGÍA SPINE-LEAF

Una **arquitectura "Spine-Leaf"** es una topología de red utilizada en centros de datos de tamaño pequeño y medio. Está formado por la capa superior llamada "Spine" y la capa inferior "Leaf". La particularidad se encuentra en las conexiones de los switches entre capas, están conectados de tal manera que cada switch "Spine", interconecta con todos los switches "Leaf", lo que significa que cada dispositivo, esta únicamente a 4 enlaces con cualquiera otro, 2 si están en el mismo switch.

En la Figura 6 visualizamos una topología formada por 2 switches Spine y 4 switches Leaf.

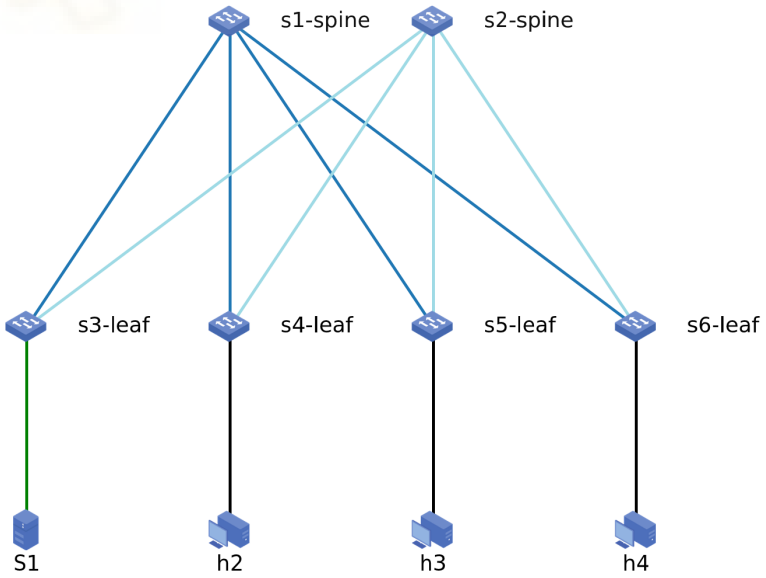


Figura 6: Topología Spine-Leaf

La arquitectura se compone de los siguientes elementos:

- **Columna vertebral ("Spine")**: Esta es la capa superior de la arquitectura. Consiste en **switches de alta capacidad** que se conectan entre sí formando una malla

completa con los switches Leaf. Estos switches son responsables de enrutar el tráfico entre las ramificaciones (capa Leaf).

- **Ramificaciones ("Leaf")**: La capa inferior está formada por **switches de acceso**. Estos switches se conectan directamente a los hosts. Cada switch "Leaf" se conecta con todos los switches "Spine", creando una conexión directa entre los servidores y la columna vertebral.

La arquitectura "Spine-Leaf" ofrece varias ventajas:

1. **Latencia baja y predecible**: Al garantizar que el tráfico tenga siempre el mismo número de saltos hasta su próximo destino, se reduce la latencia. Esto es crucial para aplicaciones de alto rendimiento.
2. **Mejor escalabilidad**: A diferencia de las redes tradicionales de tres capas, donde se agregan más switches para escalar, la arquitectura "Spine-Leaf" permite un **escalamiento horizontal** sin aumentar la infraestructura.
3. **Utilización del algoritmo ECMP**: En lugar de depender del Spanning Tree Protocol (STP), que a menudo limita el rendimiento, las topologías "Spine-Leaf" permiten la utilización de protocolos como **ECMP** para equilibrar el tráfico de carga en todas las rutas disponibles.
4. **Reducción de costos**: Permite aumentar las conexiones a cada switch, de tal manera que permite desplegar menos switches en la topología soportando las mismas cargas de trabajo.

En esta arquitectura es ideal para centros de datos modernos con aplicaciones distribuidas y tráfico este-oeste creciente. [13], [14]

2.4 Algoritmos para Prevención de Bucles

En redes con enlaces redundantes (varios caminos que llevan al mismo destino) producen bucles. Este fenómeno genera situaciones indeseadas, como la duplicación exponencial de las tramas hasta colapsar la red.

Las situaciones indeseadas son las siguientes:

- **Tormenta de Broadcast**: Al enviar una trama broadcast (a todos los hosts de la red), dicha trama empieza a duplicarse, porque los switches lo reenvían por todos los puertos, menos por el puerto recibido. La trama llega a los switches adyacentes, estos hacen la misma acción, de tal forma que la trama broadcast volverá al switch inicial en algún punto, y el switch repetirá la misma acción, este proceso se repetirá de manera indefinida, solo si hay bucles.
- **Inestabilidad de la tabla MAC**: Los switches generan la tabla MAC mediante las tramas que reciben, a cada trama que recibe, asigna la MAC al puerto correspondiente. Al recibir la misma trama con un puerto distinto, provoca una actualización de la tabla MAC.
- **Tramas Unicast Duplicadas**: Las tramas unicast (destinado a un único host) pueden duplicarse al atravesar un switch que no tiene su MAC en su tabla de direcciones MAC, por lo tanto, envía la trama por todos los puertos, a excepción del puerto recibido.

Para entender mejor **la tormenta de broadcast,** se explicara con un ejemplo: En la Figura 7 podemos observar como el host 1 inicia el mensaje broadcast[1], por ello comenzamos en el host 1, dicho host genera la trama numerado como el número 1, si seguimos los números, observamos que volvemos al switch inicial, número 13, y desde allí se repite el proceso, hasta que incremente de forma exponencial el número de tramas de broadcast y sature toda la red.[15]

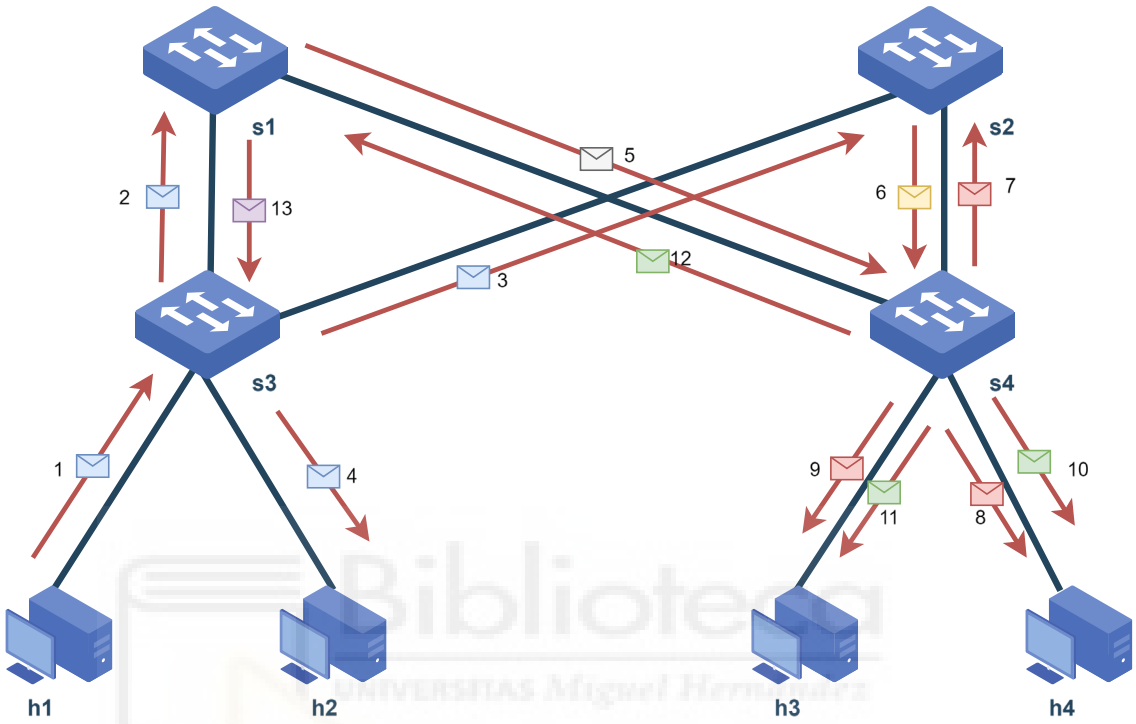


Figura 7: Tormenta de broadcast

La tormenta de **tramas Unicast Duplicadas**, es parecida ya que el switch no conoce el destino de la trama y por lo tanto lo reenvía por todos los puertos excluyendo el puerto recibid, el mismo comportamiento que broadcast.

Para evitar tales situaciones, se implementan protocolos y algoritmos que se encargan de detectar los bucles y anularlos. Hay muchos de ellos, pero se explicará el **Spanning Tree Protocol (STP)** y **Equal-Cost Multi-Path (ECMP).**

2.4.1 STP

El **Spanning Tree Protocol (STP)** es un protocolo de capa 2 diseñado para prevenir bucles en redes que contienen enlaces redundantes. Es el más conocido, ya que fue uno de los primeros en implementarse y de lo más utilizados.

El funcionamiento del algoritmo es sencillo, bloquear enlaces redundantes que puedan provocar bucles, dejando los enlaces mínimos necesarios para interconectar toda la red.

El funcionamiento del algoritmo STP se visualizará en la Figura 8.

[1] Una trama Broadcast es aquel que está destinado a todos los hosts que forma parte de la red

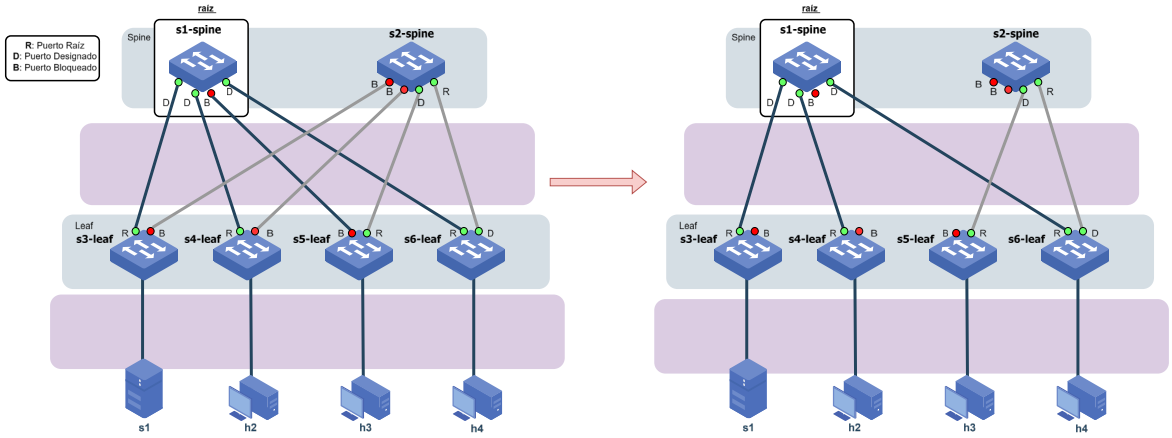


Figura 8: Funcionamiento del algoritmo STP

En el caso del algoritmo STP, tiene un costo, puede limitar el rendimiento de la red ya que bloquea todos los enlaces redundantes. Por lo tanto, este algoritmo no es el más conveniente para la topología Spine-Leaf. [16]

Hay que aclarar que en el entorno de SDN, no se utiliza el protocolo de Árbol de expansión, si no, su algoritmo, ya que, en el protocolo, cada switch se tiene que comunicar con los demás, por ello el protocolo tiene un tiempo de convergencia, en SDN, como todos los switches están comunicados con el controlador SDN, no hay comunicación entre switches, si no, que cada switch se comunica con el servidor. Por lo tanto, el tiempo de convergencia es nulo en SDN, porque todo esta centralizado.

2.4.2 ECMP

El **Equal-Cost Multi-Path (ECMP)** es un algoritmo de enrutamiento que permite balancear la carga de tráfico a través de múltiples rutas que tienen el mismo costo o métrica.[17]

En el contexto de la arquitectura Spine-Leaf, ECMP permite que el tráfico se distribuya de manera eficiente a través de todas las rutas disponibles, lo que mejora el rendimiento y la fiabilidad de la red. Es muy conveniente el algoritmo en topología Spine-leaf porque está diseñada de tal manera que los múltiples caminos que interconectan dos hosts cualesquiera tengan el mismo costo, esto es ideal para el algoritmo.

A continuación, se detalla el funcionamiento del algoritmo ECMP concretamente en una topología Spine-Leaf.

2.4.2.1 Identificación de los caminos

En el caso de la topología Spine-Leaf, este paso es muy sencillo ya que las distintas rutas que se dispone tienen el mismo costo, y como es una topología simétrica, siempre son los mismos caminos. No es necesario de implementar un algoritmo de búsqueda de caminos.

2.4.2.2 Selección de los caminos

Para distribuir los flujos de datos a los enlaces disponibles, se utiliza una función hash.

Una función hash toma unos valores de entrada, y proporciona una salida con otros valores, de una longitud fija. Normalmente la salida es un valor más corto que la entrada. Con la Figura 9 podemos visualizar el funcionamiento.

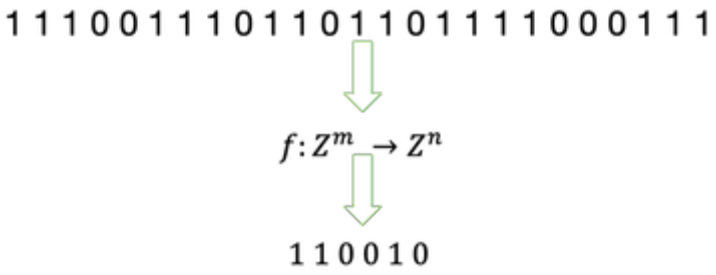


Figura 9: Funcionamiento de un hash [18]

Si a la función hash le proporcionamos los mismos valores, tendremos la misma salida.

Los valores que toma del paquete pueden ser:

- 3-Tuple: Dirección IP de origen, Dirección IP de destino, Protocolo
- 5-Tuple: Dirección IP de origen, Dirección IP de destino, Protocolo, Puerto de origen, Puerto de destino.
- 6-Tuple: Dirección IP de origen, Dirección IP de destino, Protocolo, Puerto de origen, Puerto de destino, VLAN.

En nuestro caso, se utiliza un hash simétrico de 6-Tuple. Un hash simétrico es aquel que ordena los parámetros como la IP de origen y destino, de tal manera que, al aplicar hash en el origen o destino, se obtiene el mismo resultado, y por ello el camino de ida y vuelta es el mismo.[19]

Una vez obtenido la salida del hash, se procede a elegir el enlace, para ello se aplica la operación:

$$H \ mod \ n \qquad (1)$$

Siendo H: el valor de hash.

Siendo n: el número de enlaces, equivalente en Spine-Leaf al número de rutas.

La operación modulo nos proporciona el resto de la división, siendo los resultados posibles desde el valor de 0 hasta $n - 1$. Con el resultado del módulo, se asignarán los flujos a los enlaces. Si el número de enlaces es 6, la operación nos proporcionara un resultado entre 0 a 5. De esta manera asignamos los flujos a los enlaces. [18]

Un ejemplo con el caso de un hash 110010 y un total de 6 enlaces para elegir:

$$110010 \ mod \ 6 = 0 \qquad (2)$$

El resultado indica que el flujo será asignado al enlace 1.

2.4.2.3 Factor estadístico

En el apartado anterior, se explicó como el algoritmo elije los posibles caminos, como es el hash el encargado de elegir, la toma decisión se vuelve pseudoaleatorio, esto se vuelve un problema o no, depende de la situación.

Si en la red tenemos pocos flujos de gran tamaño, es un problema, ya que el algoritmo puede asignar varios flujos a la misma ruta, y saturarla.

Pero si tenemos muchos flujos, esto cambia, la asignación se vuelve más uniforme.

Este comportamiento es debido a la ley de los grandes números, que es **un teorema fundamental de la teoría de la probabilidad**, lo que demuestra es que, al tener un número elevado de sucesos, la asignación tiende a ser una constante, es decir, misma posibilidad en todas las posibles opciones. [20]

Si observamos la Figura 10, la posibilidad de que salga una cara en concreto tiene una posibilidad de $\frac{1}{6}$ o 16´66%, como son sucesos independientes, es decir, la posibilidad de que en el próximo lanzamiento no influye el anterior, al aumentar el número de lanzamientos observamos que se aproxima a la posibilidad teórica.

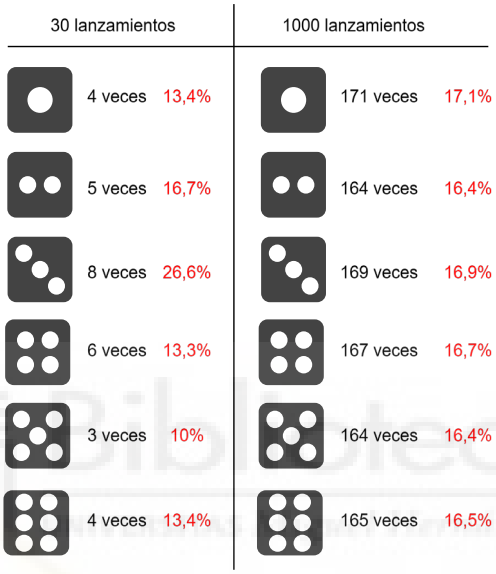


Figura 10: La ley de los grandes números [21]

Este comportamiento estadístico es clave, ya que nos permite implementar un algoritmo sencillo que es capaz de aprovechar la topología Spine-Leaf.

2.5 LATENCIA

La latencia de red es el tiempo de ida y vuelta que tarda en recorrer el paquete desde un punto hasta otro. Cuando este parámetro es más pequeño, es mejor.[22]

Cualquier elemento pasivo/activo introduce latencia, algunos ejemplos son: distancia física entre los equipos, la congestión en la red, la calidad de los cables y el equipo de red, y la velocidad del procesamiento de los datos.[23]

Es importante mantener este parámetro en niveles óptimos para la experiencia del usuario. Se considera una buena latencia menor de 30 ms.[24]

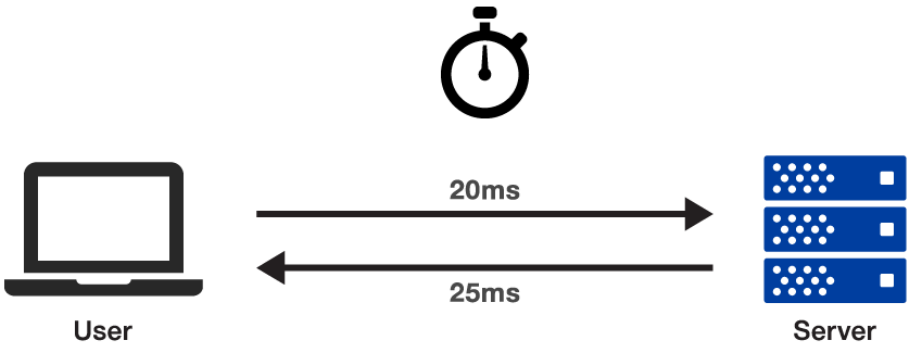


Figura 11: Medición de la Latencia [22]

Para determinar la latencia, utilizaremos “Ping”. Dicha herramienta se explicará en el apartado 3.6.

2.6 ANCHO DE BANDA

El **ancho de banda** es un término que se utiliza para describir la capacidad de transmisión de datos de una red o conexión a Internet. Se mide en bits por segundo (bps o bits/s) y expresa la cantidad máxima de datos que pueden ser transmitidos en un segundo.

Es decir, el ancho de banda es como una autopista: cuanto más ancho sea el camino (es decir, mayor sea el ancho de banda), más datos pueden viajar a la vez, lo que resulta en una transferencia de datos más rápida. Sin embargo, si el ancho de banda es limitado, los datos tendrán que “hacer cola” y la transferencia será más lenta.[25]

Es por eso por lo que un ancho de banda más alto generalmente se traduce en una mejor experiencia de usuario.

2.7 FLUJO

Un flujo en redes se refiere a un conjunto de paquetes que comparten ciertas características y que son tratados de manera uniforme por los componentes de la red. Es decir, son reglas que los switches de capa 3 utilizan para decidir qué hacer con los paquetes de red que reciben.

Dichas características pueden ser:

- Dirección IP de origen y destino.
- Puerto de origen y destino.
- Protocolo de transporte (por ejemplo, TCP o UDP).

El flujo puede ser visto como una “conversación” entre dos dispositivos de la red. Los componentes de red, como los switches de capa 3 y Reuters, utilizan estas características para identificar y gestionar los flujos, aplicando políticas específicas como enrutamiento, calidad de servicio (QoS) o seguridad.[26]

En OpenFlow, un flujo se define mediante una entrada en la tabla de flujos del switch, que especifica cómo deben ser tratados los paquetes que coinciden con ciertos criterios.

3 HERRAMIENTAS

Este apartado consistirá en la descripción de las herramientas y el entorno utilizados para la realización del proyecto.
Al utilizar Mininet como emulador, nos condiciona a trabajar en el entorno Linux.

3.1 UBUNTU

Ubuntu es un sistema operativo basado en Linux de código abierto que se utiliza en muchos ámbitos, tiene una gran popularidad en el sector de los servidores. Ofrece opciones de automatización y capacidad de programación para centros de datos, campus y redes de área amplia. [27]

Ubuntu ofrece versiones LTS, que significa que durante 5 años recibirán actualizaciones, utilizaremos estas versiones, principalmente la versión Ubuntu 20.04 LTS.

3.1.1 Instalación

Para ello accedemos a la página oficial de Ubuntu para descargar la ISO (Archivo de imagen de disco).[28]

Una vez Descargada la imagen, tenemos dos opciones de Instalación.

3.1.1.1 Máquina virtual

La instalación en una máquina virtual puede ser el método más sencillo y cómodo, pero **no lo recomiendo** si el propósito es la emulación, en el apartado 5.1.1 , se explicará el error que se obtuvo en las emulaciones. También se debe tener en cuenta el consumo de recursos computacionales, es mucho mayor que en la opción de la instalación en una partición.

Para la instalación en una máquina virtual, tenemos dos opciones comerciales populares:

- **Virtual box**: La más popular, pero en emulaciones con Mininet-WiFi, se obtiene resultados nefastos.
- **VMware**: La recomendada, una opción consolidada en el mercado.

Una vez instalado la máquina virtual, montamos la imagen, en este paso tenemos dos opciones:

- Montar una **imagen limpia** de Ubuntu: Con la ISO descargada del paso anterior, la montamos en la máquina virtual, de este modo, ya tenemos el entorno Linux preparado, nos faltaría instalar todas las herramientas faltantes, como Mininet-WiFi (emulador) o controlador SDN.
- Montar una **imagen preparada:** Tenemos la posibilidad de acceder a las páginas web oficiales de las distintas herramientas como Mininet-WiFi o los controladores SDN y descárganos una imagen preparada, con las herramientas ya instalados. Es una opción cómoda para empezar.[29]

3.1.1.2 Partición

La instalación en una partición es la opción más recomendable para nuestras emulaciones.

En este caso vamos a necesitar la ayuda de una herramienta para montar la imagen ISO en un pendrive u otro dispositivo de almacenamiento para la posterior instalación.

Una herramienta popular y sencilla es Rufus, se puede descargar la versión portable o el instalador, cualquiera opción es válida. [30]

Una vez instalado Rufus, montamos la imagen ISO en un pendrive e instalamos el Ubuntu en el dispositivo o partición deseada.

3.1.2 Comandos

Para poder trabajar en un entorno Linux, muchas veces recorrimos a los comandos de consola, en este apartado describiremos los comandos más usados en el proyecto.

Comando	Descripción
ls	Lista el contenido de una carpeta
cd {directorio}	Cambia al directorio especificado
cd ..	Cambia al directorio padre del directorio actual
nano	Editor de texto
sudo	Ejecuta un comando como superusuario
sudo -i	Inicia una nueva sesión como superusuario
sudo apt-get update	Actualiza la lista de paquetes disponibles
sudo apt-get upgrade	Actualiza todos los paquetes instalados
git clone {dirección web}	Descargar un repositorio
tar -xzf archivo.tar.gz	Descomprimir archivo
sudo pip3 install {librería}	Instalar librería deseada de Python
sudo rm -r	Eliminación de un directorio

Tabla 2: Comandos Linux más utilizados

3.2 MININET

Mininet es un emulador de red que permite el despliegue de redes SDN sobre los recursos de un ordenador o una máquina virtual. Los hosts de Mininet pueden ejecutar cualquier software que este instalado en el sistema Linux. Los switches soportan el protocolo OpenFlow.[31]

Figura 12: Logo de Mininet-Wifi [32]

En conclusión, los hosts virtuales, switches virtuales, enlaces y controladores de SDN, aunque sean emulaciones, se comportan como reales. Por lo tanto es posible crear redes muy similares a unas reales.[32]

En nuestro caso utilizaremos el Mininet-WiFi, es el mismo emulador, pero incluye novedades como estaciones wifi.[33]

3.2.1 Instalación

Para la instalación, tendremos que ejecutar siguientes comandos de Linux:[34]

```
sudo apt-get install git
git clone https://github.com/intrig-unicamp/mininet-wifi
cd mininet-wifi
sudo util/install.sh -Wlnfv
```

3.2.2 Comandos más utilizados

Se procederá a explicar los comandos más utilizados en este proyecto.

Crear una topología simple, muy útil para comprobar el funcionamiento de Mininet-WiFi:

```
sudo mn
```

Crea una topología de malla de 3x3 switch, con un controlador especifico que accede con la dirección IP proporcionada:

```
sudo mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --topo torus,3,3
```

Crea una topología personalizada desde un archivo Python, con un controlador deseado, tenemos que proporcionar la dirección del archivo de la topología:

```
sudo mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --custom
~/spine_leaf.py --topo mytopo
```

Una vez dentro de la emulación, podemos comprobar la conectividad de todos los hosts con la herramienta ping:

```
mininet-wifi>pingall
```

El indicativo “mininet-wifi>” delante de los comandos significa que estamos ejecutándolos en el entorno Mininet-WiFi.

Comprueba el ancho de banda entre hosts:

```
mininet-wifi>iperf h1 h2
```

Comando para salir de la emulación:

```
mininet-wifi>exit
```

Formatea a la configuración inicial de Mininet, muy recomendado después de salir de una emulación y lanzar otra, evita errores:

```
sudo mn -c
```

3.2.3 Ejemplo de utilización

El orden de los comandos para una emulación de Mininet-WiFi:

1. Creación de la emulación, el más sencillo es:

```
sudo mn
```

2. Dentro de la emulación, procedemos a hacer las pruebas deseadas, la más sencilla es:

```
mininet-wifi>pingall
```

3. Salimos de la emulación:

```
mininet-wifi>exit
```

4. Formateamos Mininet-Wifi:

```
sudo mn -c
```

3.3 LENGUAJE DE PROGRAMACIÓN PYTHON

Como el emulador está basado en Python, este será nuestro lenguaje de programación utilizado.

Es un lenguaje de programación que en los últimos años ha tomado fuerzas, por los siguientes puntos[35]:

- Felicidad de Aprendizaje: Se volvió conocido, por una parte, por ser un lenguaje accesible para personas de todo tipo de profesión, teniendo una curva de aprendizaje suave.

- Amplia Comunidad: Detrás de Python, cuenta con una comunidad enorme, lo que le permite tener todas tipos de librerías.

- Sintaxis: Cuenta con una sintaxis sencilla y legible, lo que facilita el trabajo en todos los sentidos.

También tiene puntos débiles, como la velocidad, es más lento que otros lenguajes de nivel más bajo como "C". En este proyecto no tenemos dicha necesitad.

3.3.1 Librerías

Con la Tabla 3, se nombrará las librerías más utilizadas en el proyecto y se explicará su funcionamiento.

Librería	Descripción
pexpect	Controla entornos interactivos basadas en terminal, utilizado para interactuar con Mininet-WiFi.
sys	Proporciona acceso a funcionalidades del sistema, como la entrada y salida estándar de datos. Utilizado para la lectura del terminal.
subprocess	Crea y controla nuevos procesos de Python. Utilizado para ejecutar el algoritmo ECMP entre otros usos.
time	Funciones para trabajar con el tiempo. Utilizado como una función aleatoria para generar archivos con distintos nombres.
re	Función de búsqueda, entre otras funcionalidades. Utilizado para la extracción de las medidas en los archivos generados.
os	Utiliza funcionalidades del sistema operativo. Se ha utilizado para la manipulación de las variables de entorno y otros usos.
pandas	Manipulación y análisis de datos con estructuras. Se ha utilizado su estructura.
matplotlib.pyplot	Creación de gráficos.
matplotlib.image	Permite trabaja con imágenes. Utilizado en la visualización de la topología.
matplotlib.lines	Crea y manipula líneas en los gráficos. Se ha utiliza en la visualización de la topología.
argparse	Permite la utilización de los argumentos pasados a un script desde la línea de comandos.
numpy	Soporte para arrays de tamaños grandes, junto con funciones matemáticas. Se ha utilizado para obtener la media.

Tabla 3: Librerías más utilizadas

Algunas de las librerías mencionadas en la Tabla 3, se deben instalarse. La instalación se ejecuta mediante el siguiente comando (en Linux):

```
sudo pip3 install “el nombre de la librería a instalar”
```

3.4 CONTROLADOR SDN

Es el cerebro de nuestra arquitectura, existe de dos tipos, los comerciales y lo de código abierto, en nuestro caso adoptaremos por los de código abierto. El motivo es la posibilidad que nos ofrecen de emular con Mininet-WiFi. En la actualidad están tomando fuerza los comerciales, esto es debido por el motivo de la dificultad de captar ingresos en el mercado de los controladores.[36]

3.4.1 OpenDayLight

OpenDayLight (ODL) es un controlador de redes definidas por software (SDN) de código abierto, desarrollado por una comunidad activa y respaldado por la Fundación Linux. Es una plataforma flexible y escalable para la gestión y control de redes, permitiendo a los administradores configurar y monitorear la infraestructura de red de manera programable.[37]

Figura 13: Logo del OpenDayLight

3.4.1.1 Instalación

Para la instalación de dicho controlador, utilizaremos como guía el video proporcionado por la universidad politécnica Salesiana, Ecuador. [38]

La instalación del controlador se realizará en el Ubuntu 20.04 LTS.

Procedemos a instalar los requisitos para el entorno:

```
sudo apt remove default-jre-headless

sudo add-apt-repository universe

sudo apt install openjdk-8-jre-headless -y

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

Ahora tenemos que clonar el repositorio, las distintas versiones del controlador se encuentran en el repositorio oficial de opendaylight.[39]

En nuestro caso, optamos por la versión 0.3.0:

```
wget
https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/opendaylight/integration/distribution-karaf/0.3.0-Lithium/distribution-karaf-0.3.0-Lithium.tar.gz

tar -xvf distribution-karaf-0.3.0-Lithium.tar.gz
```

Una vez clonado y descomprimido, entramos al directorio y ejecutamos el controlador:

```
cd distribution-karaf-0.3.0-Lithium/
sudo ./bin/karaf
```

Si la instalación tiene éxito, tenemos que visualizar la Figura 14.

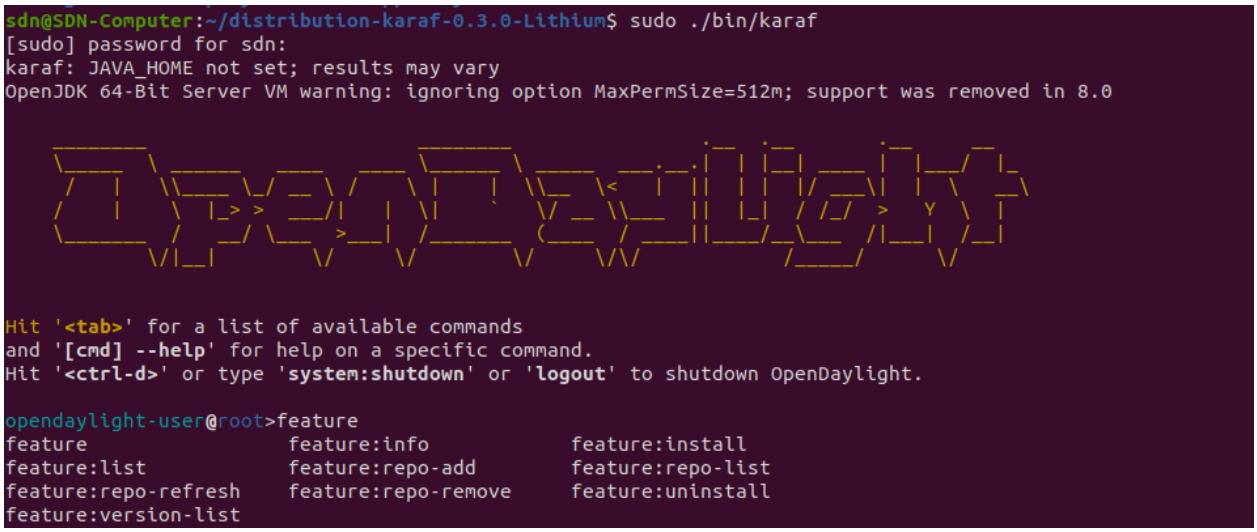


Figura 14: Entorno CLI del OpenDayLight

Para un buen funcionamiento tenemos que instalar las siguientes aplicaciones al controlador:

```
feature:install odl-restconf-all odl-openflowplugin-all odl-l2switch-all odl-mdsal-all odl-
yangtools-common odl-dlux-all
```

Procedemos a instalar un servidor web, para la interfaz web, en nuestro caso es Apache:

```
sudo apt install apache2
systemctl start apache2
sudo systemctl status apache2
```

3.4.1.2 Interfaz web

Para acceder a la interfaz web tenemos que acceder al navegador web para proporcionar el siguiente enlace: http://localhost:8181/index.html

Siendo el usuario/contraseña: admin/admin

3.4.1.3 Ejecución

Después de la instalación, para encender el controlador procedemos a iniciar el interfaz web y controlador:

```
systemctl start apache2
cd ~/distribution-karaf-0.3.0-Lithium
sudo ./bin/karaf
```

En otra pestaña del terminal ejecutamos el emulador Minnet-WiFi:

```
sudo mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --custom
spine_leaf.py --topo mytopo
```

El comando anterior ejecuta una topología personalizada desde un archivo Python, en el apartado 3.2.2 , están los distintos ejemplos de los comandos Mininet-WiFi.

3.4.2 Controlador ONOS

A continuación, se explicará el controlador ONOS, que fue el primero controlador en ser usado para este proyecto, pero por desgracia sufrió un ataque de ransomware. Antes del dicho ataque, fue unos los más importantes y avanzadas, pero después del ataque no se recuperó. Como aún se puede encontrar el archivo descargable para la instalación, se conserva el apartado.

3.4.2.1 Instalación

Para la instalación de este controlador, seguiremos las instrucciones Wikipedia de ONOS.[40]

Primer paso será preparar el entorno para su posterior instalación.[41]

Instalamos java:

```
sudo apt install openjdk-11-jdk
```

Este es un paso Opcional, es para mejorar el rendimiento del controlador, para ello tenemos que aplicar los siguientes comandos:

```
sudo su
cat >> /etc/environment <<EOL
JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
JRE_HOME=/usr/lib/jvm/java-11-openjdk-amd64/jre
EOL
```

Y por último Curl:

```
sudo apt-get install curl
```

Una vez teniendo los requisitos instalados, procedemos a instalar el controlador SDN Para ello creamos/accedemos a la carpeta opt.

```
sudo mkdir /opt
cd /opt
```

Descargamos la versión deseada de ONOS, para obtener el enlace del archivo ONOS con la versión deseada consultamos la siguiente página web: [42]

```
sudo wget -c https://repo1.maven.org/maven2/org/onosproject/onos-releases/$ONOS_VERSION/onos-$ONOS_VERSION.tar.gz
```

Ejemplo de la versión 2.7.0:

```
sudo wget -c https://repo1.maven.org/maven2/org/onosproject/onos-releases/2.7.0/onos-2.7.0.tar.gz
```

Descomprimimos el archivo (ejemplo para la versión 2.7.0):

```
sudo tar xzf onos-2.7.0.tar.gz
```

Cambiamos el nombre de la carpeta:

```
sudo mv onos-2.7.0 onos
```

Arrancamos ONOS:

```
/opt/onos/bin/onos-service start
```

3.4.2.2 Encender CLI

Para encender el entorno de consola, tenemos que introducir siguiente comando:

```
cd onos
./tools/test/bin/onos localhost
```

Si obtenemos un error al ejecutarlo, la alternativa es:

```
ssh -p 8101 karaf@localhost
```

La contraseña es: "karaf"

La ejecución correcta la podemos ver en la Figura 15.

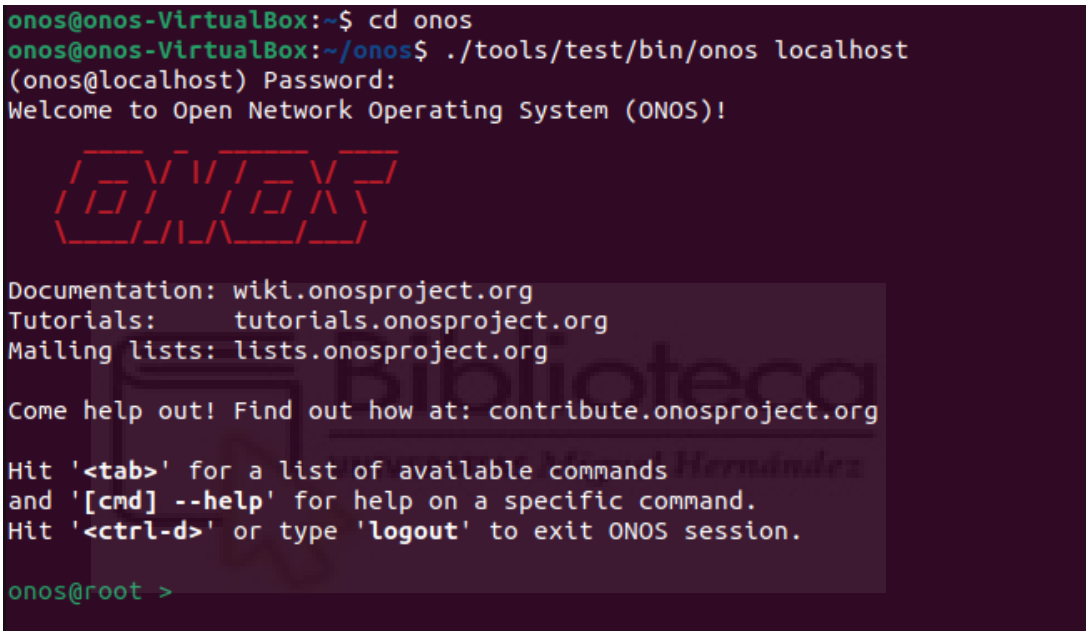


Figura 15: Entorno CLI de ONOS

3.4.2.3 Entorno web

Para acceder al entorno web tenemos que introducir en el navegador el siguiente enlace:

http://localhost:8181/onos/ui

Siendo el usuario/contraseña: onos/rocks

La interfaz web tiene la apariencia de la Figura 16, en la imagen se accedió a la interfaz mediante la IP, ambos modos son válidos.

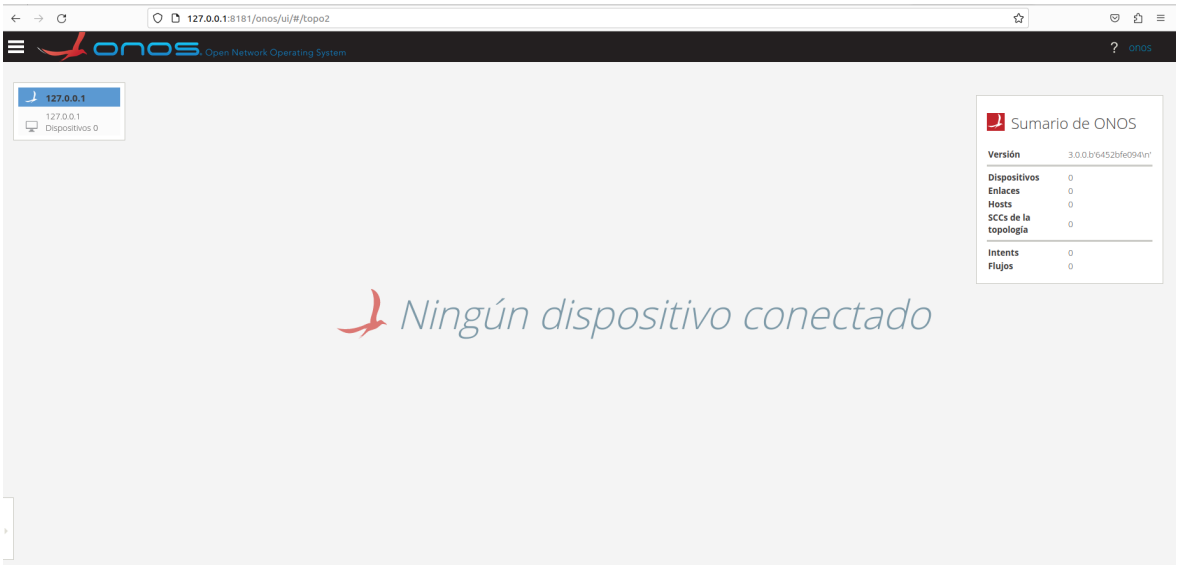


Figura 16: Entorno GUI de ONOS

3.4.2.4 Creación de Proceso

Ahora nos disponemos a crear el proceso de inicio, que servirá para inicial en el mismo instante que el sistema operativo.

Tenemos que introducir los siguientes comandos para una versión de Ubuntu superior a 16: [43]

```
sudo cp /opt/onos/init/onos.initd /etc/init.d/onos

sudo cp /opt/onos/init/onos.service /etc/systemd/system/

sudo systemctl daemon-reload

sudo systemctl enable onos
```

3.4.2.5 Comandos

Los comandos que veremos solo tendrán sentido dentro de la consola de ONOS.

Visualizar las aplicaciones que están activas en el controlador:

```
onos> apps -a -s
```

La opción "-a" significa solo visualizar las aplicaciones activas.

A continuación, omitiremos "onos>", pero seguiremos dentro de la consola de ONOS.

Si ejecutamos una topología con bucles (varios caminos que llegan al mismo punto) en Mininet-WiFi, podemos comprobar con el comando "pingall", no existe conectividad entre los hosts, el motivo es básicamente que en nuestro controlador SDN no tenemos activado ninguna aplicación que evite los bucles.

Para activar las aplicaciones necesarias para redes con bucles, ejecutamos:

```
app activate org.onosproject.fwd
app activate org.onosproject.openflow
```

Ahora el ping funcionaria.

También podemos desactivar las aplicaciones mientras el controlador este operativo.

```
app deactivate fwd
```

Está habilitada la abreviación de “org.onosproject.fwd” a “fwd”:

Disponemos de multitud de comandos, para visualizar la lista de todos los comandos disponibles:

```
help onos
```

También es posible ver la lista de los dispositivos/host/enlaces/flujos:

```
devices
host
links
flows
```

Dada una topología de red, ONOS calcula todas las rutas más cortas entre dos nodos cualesquiera. Esto es especialmente útil para que las aplicaciones obtengan información de ruta de acceso para la instalación de flujo o para algún otro uso. El comando paths toma dos argumentos, ambos son dispositivos. Para facilitarle las cosas, ONOS proporciona autocompletado CLI simplemente presionando la tecla <TAB>.

```
paths <TAB>
```

3.5 CONTAINERNET

Es una herramienta que permite utilizar contenedores con el emulador Mininet-WiFi. [44]

Este apartado será una mención ya que por falta de recursos computacionales no se ha podido continuar por este camino, pero el código para crear una topología Spine-Leaf con contenedores está disponible en el apartado 7.2, y es totalmente funcionar.

3.5.1 Instalación

Para la instalación, consultemos la página oficial de containernet.[44]

Los comandos para la instalación son los siguientes:

```
sudo apt-get install ansible
git clone https://github.com/containernet/containernet.git
sudo ansible-playbook -i "localhost," -c local containernet/ansible/install.yml
```

3.6 PING

Ping es una utilidad de diagnóstico de red que mide la latencia, que es el tiempo que tarda un paquete de datos en viajar desde un origen a un destino y regresar. Se utiliza principalmente para medir la latencia y verificar la conectividad entre dos dispositivos en una red.[45]

La latencia esta explicada en el apartado 2.5.

3.6.1 Funcionamiento

El funcionamiento interno es el siguiente: [45]

1. Envía un paquete ICMP (Internet Control Message Protocol) {Echo Request} al destino.
2. El destino responde con un paquete ICMP {Echo Reply}.
3. Mide el tiempo transcurrido entre el envío y la recepción del paquete.
4. Este tiempo de ida y vuelta (RTT - Round Trip Time) se reporta como la latencia.

3.6.2 Parámetros

Los parámetros que se ha utilizado en el proyecto son los siguientes:[45]

Parámetro	Descripción
-c [count]	Número de paquetes a enviar
-i [interval]	Intervalo entre paquetes en segundos

Tabla 4: Parámetros del Ping

3.6.3 Ejemplo de uso

El comando que hemos utilizado en los scripts es el siguiente:

```
ping 10.0.0.1 -c 10 -i 0.2
```

Siendo un número total de 10 paquetes, con un intervalo por paquete de 0.2 segundos, este intervalo es el más pequeño que nos permite el Ping en Mininet-WiFi.

3.7 IPERF

Iperf es una herramienta de comandos utilizada para medir el rendimiento de la red, especialmente el ancho de banda entre dos dispositivos.[46]

3.7.1 Funcionamiento

Iperf opera en modo cliente-servidor. Un extremo actúa como servidor y el otro como cliente. El cliente envía datos al servidor durante un período determinado. La herramienta mide la cantidad de datos transferidos y calcula el ancho de banda. Dicha prueba se ejecuta en TCP (Transmission Control Protocol), pero también puede ejecutarse en UDP (User Datagram Protocol).

La medición del ancho de banda se calcula enviando una serie de paquetes entre el cliente y el servidor, mide el tiempo que toman en transferir dichos paquetes, calcula el ancho de banda dividiendo los datos totales transferidos entre el tiempo transcurrido. [47]

3.7.2 Parámetros

Los Parámetros utilizados en el proyecto son:[48]

Parámetro	Descripción
-s	Inicia Iperf en modo servidor
-c [host]	Inicia Iperf en modo cliente, conectándose al servidor especificado
-t [tiempo]	Duración de la prueba en segundos (por defecto 10 segundos)
-P [número]	Número de flujos paralelos
-b [ancho de banda]	Limita el ancho de banda de la prueba
-i	Informe cada cierto periodo en segundo

Tabla 5: Parámetros de la herramienta Iperf

3.7.3 Ejemplo de uso

Comandos en el host que actuara como servidor:

```
iperf -s
```

En este caso el host1 con la IP 10.0.0.1 actuara como servidor.

Comandos en el host que actuara de cliente:

```
iperf -c 10.0.0.1 -t 5 -b 20M -P 5 -i 1
```

Con el comando anterior, iniciamos una prueba de ancho de banda hacia el servidor que es el host 1, la prueba durara 5 segundos, con un ancho de banda máximo de 20 Mbits/s de cada flujo, y en total serán 5 flujos, por lo tanto, el ancho total será de 100 Mbits/s. Por el terminal visualizaremos un informe sobre distintos parámetros de la prueba.

En la Figura 27, podemos observar el funcionamiento del Iperf desde varios clientes, con un tiempo de 7 segundos y 2 flujos paralelos de cada cliente.

4 SCRIPTS IMPLEMENTADOS

En este apartado consistirá en explicar el funcionamiento e implementación de los scripts que fueron desarrollados para testear las distintas configuraciones de la topología Spine-Leaf.

Pero antes de ello, se explicará cómo los distintos scripts implementados interactúan entre ellos. Fueron creados scripts auxiliares para interactuar en los distintos contornos como el emulador Mininet-WiFi o con el protocolo OpenFlow, en la Figura 17 lo podemos observar. En caso del protocolo OpenFlow, podemos interactuar con un controlador SDN cualquiera o mediante el script ECMP que define los flujos necesarios para el buen funcionamiento de la topología Spine-Leaf. Los scripts implementados cuentan con la ventaja de trabajar con el controlador que queramos o ninguno.

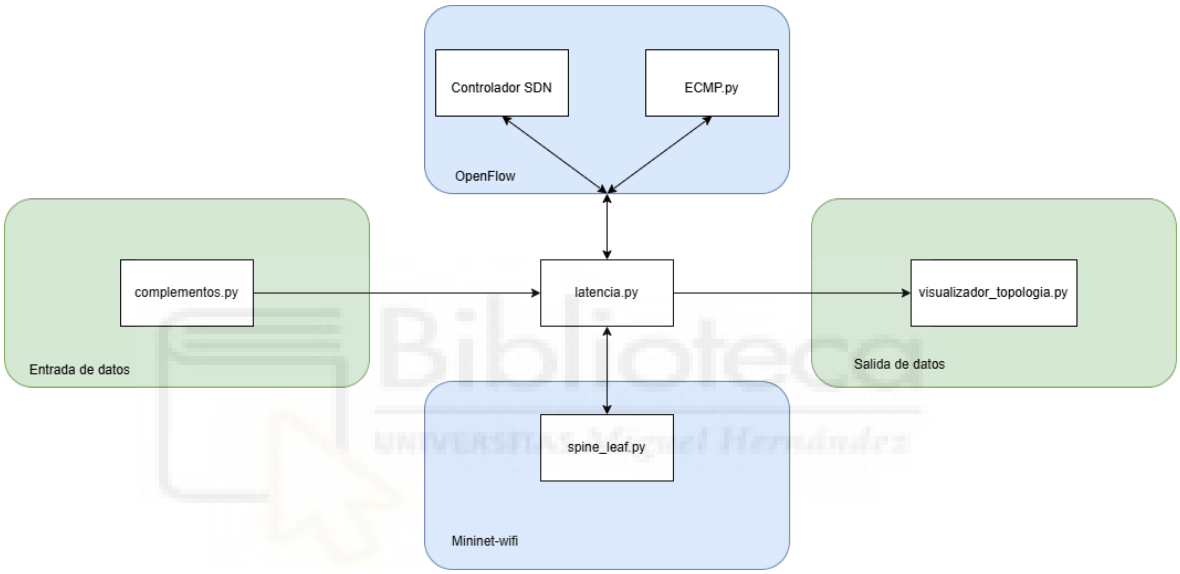


Figura 17: Comportamiento entre los scripts

En la Figura 17 podemos observar cómo interactúa "latencia.py" con los scripts auxiliares. Los demás scripts de testeo tendrían el mismo diagrama que la Figura 17, pero cambiando la posición de "latencia.py" por el suyo:

- "ancho_banda_flujo_unico.py"
- "ancho_banda_flujo_unico_N_configuraciones.py"
- "ancho_banda_flujo_multiple.py".
- "ancho_banda_flujo_multiple_N_configuraciones.py"
- "ancho_banda_flujo_distribuido.py".
- "ancho_banda_flujo_distribuido_N_configuraciones.py"
- "latencia_bajo_carga_flujo_distribuido.py"

Para interactuar entre ellos, se han utilizado dos maneras distintas, la primera por llamada de la función, lo más sencilla y conocida, pero no es posible hacerlo con el script "spine_leaf.py" porque es una clase de Mininet-WiFi, por ello se utilizó variables de entorno.

Las variables de entorno son valores que se definen en el sistema operativo y que los programas/scripts pueden usar. Las variables de entorno serán las siguientes:

1. CAPA_SWITCH: Numero de switch de cada capa de la red.
2. NUM_HOST: Numero de dispositivos totales contando con el servidor.
3. USE_ECMP: Variable booleana del uso del algoritmo ECMP.
4. HOSTS_ADYACENTES: Variable booleana de los hosts adyacentes al servidor, si esta activada, bloquea los hosts adyacentes al servidor.

La sintaxis es la siguiente:

```
import os

# Definir una variable de entorno
os.environ['MI_VARIABLE'] = 'mi valor'

# Para acceder a a la variable
print(os.getenv('MI_VARIABLE'))
```

Cabe mencionar que, al ser creadas por nuestro script de Python, solo los procesos hijos del script padre tendrán acceso a ellos.

4.1 Creación de la Topología

Este script se encarga básicamente de la implementación de la topología Spine-Leaf al emulador Mininet-WiFi. El script se llamará “spine_leaf.py”, es un componente fundamental que interactúa con el emulador Mininet-WiFi. El código se encuentra en el apartado 7.1.

Sera utilizado por los scripts de testeo, entonces de manera directa no lo llamaremos, si no, los scripts lo harán. Este script es altamente flexible y permite adaptar la topología Spine-Leaf según las necesidades específicas, mediante las variables de entorno, que son declaradas por los scripts de testeo:

1. Número de switches en cada capa: Se proporciona como una lista, donde la longitud de la lista determina el número de capas, y cada componente especifica el número de switches en esa capa particular. En nuestro caso solo utilizaremos 2 capas, la capa Spine y Leaf, pero el script es capaz de generar más capas.

2. Número de hosts: Se especifica como un número entero, indicando la cantidad total de hosts en la topología. La distribución de los hosts con los switches es uniforme, es decir, división entera.

3. Hosts adyacentes al servidor: Este parámetro permite desactivar los enlaces de los hosts adyacentes al switch que se encuentra el servidor. Esto es útil a la hora de querer medir el ancho de banda total de la topología, de esta manera obligamos que todo el tráfico pase por los distintos switches Spine-Leaf y podamos tomar una medida más fiable.

Dichos parámetros lo podemos cambiar manualmente en el script “spine_leaf.py”, o mediante los argumentos en los scripts de testeo. Por lo tanto, tenemos dos maneras de iniciar la topología, de la forma manual escribiendo nosotros mismos en el terminal mediante comandos, o ejecutando un script de testeo que de forma automatizada llamara al

script nombrado. En el apartado 4.1.2, se explicará el uso de la manera manual, y posteriormente se verá como los distintos scripts de testeo lo utilizan.

Las líneas que permiten leer a las variables de entorno son las siguientes[2]:

```
capa_switch = list(map(int, os.getenv('CAPA_SWITCH','4,4').split(',')))
num_host = int(os.getenv('NUM_HOST', '8'))
hosts_adyacentes = int(os.getenv('HOSTS_ADYACENTES', '0'))
```

Este código lee las variables de entorno llamadas “CAPA_SWITCH” , “NUM_HOST” y 'HOSTS_ADYACENTES', que deben ser declaradas por los scripts de testeo, si no están declaradas, otorga las valores predeterminados que en este caso son [4,4] , 8 y 0. Una vez obtenidos los valores de las variables de entorno, hay que tener en cuenta que dichas variables son cadena de caracteres, en el caso de los host, se convierte en entero, pero en el caso de las capas de switches es más complicado, tenemos que separar la cadena utilizando como símbolo de separación ‘,’ , de esta forma obtenemos una lista , y por ultimo con la función map convertimos todos los elementos de la lista en enteros. La variable de los hosts adyacentes lo convertimos en entero, pero lo tratamos como variable booleana.

Después de determinar las variables, el script se encarga de desarrollar la topología, para evitar errores, asignar en la última capa, un numero de switches de tal forma que la división con el número de host tenga el resto nulo, es decir, si en la capa Leaf tenemos 4 switches, debemos asignar {4,8,16,32…} host.

Si utilizamos los valores determinados de este caso, la topología seria como en la Figura 18.

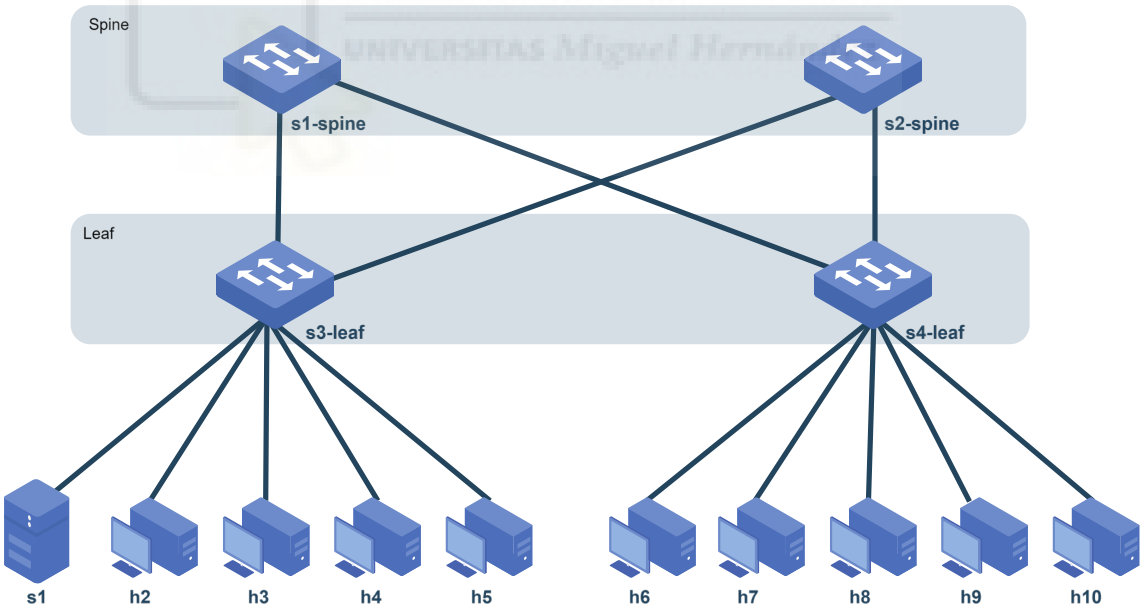


Figura 18: Topología Spine-Leaf

4.1.1 Enlaces

Para poder emular con los recursos del hardware disponible, limitaremos el ancho de banda entre los switches a 100 Mbytes/s por enlace. De esta forma conseguimos la posibilidad de emular utilizando menos recursos computacionales.

[2] Las líneas de código fueron extraídas del script spine_leaf.py que esta adjunto en el apartado Anexos

De forma predeterminada, los enlaces en Mininet-WiFi no tienen retardo, para visualizar los distintos cambios de latencia al circular por los enlaces, los enlaces que conectan los hosts con los switches tendrán una latencia de 5 μs , en cambio los enlaces que interconectan a los switches de las distintas capas tendrán una latencia de 0.5 ms.

En la realidad, un cable UTP (par trenzado no blindado) categoría 5e suele producir una latencia de 5 ns/m [49], pero el entorno de trabajo nos introduce una latencia de 0.250ms, se explicara en el apartado 5.1.2 , para poder visualizar en las gráficas el uso de los enlaces, se opta por introducir estas latencias.

De esta manera, el ejemplo de topología se nos queda como en la Figura 19, observamos que los enlaces entre switches están limitados a 100 Mbits/s con una latencia de 0.5 ms, los enlaces entre switches y hosts no tienen limitación en ancho de banda, pero con una latencia de 5 ns.

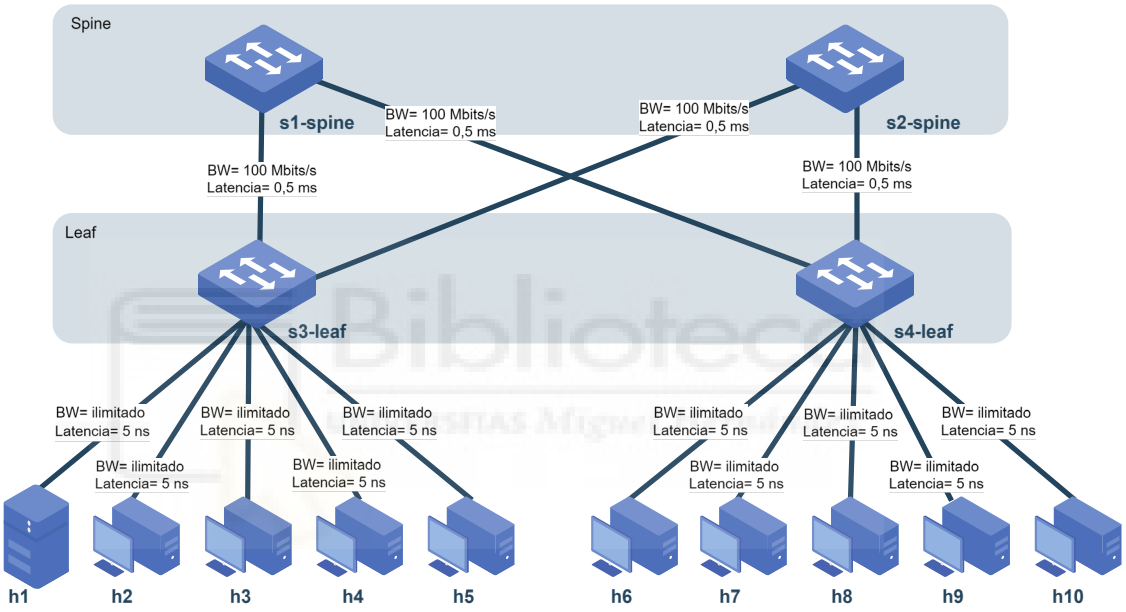


Figura 19: Topología Spine-Leaf con los parámetros

En la Figura 20 observamos como la señal desde el punto inicial, se atenúa y se retrasa hasta llegar al punto final. La latencia es un parámetro clave ya que influye directamente a la calidad del servicio. En el tema de la atenuación, no se tendrá en cuenta, ya que las distancias en los data centers no son tan grandes.

Figura 20: Retardo de un cable UTP [49]

4.1.2 Ejecución manual

Se procederá a explicar cómo usar dicho script de forma manual, para ello debemos tener el entorno de trabajo instalado correctamente, básicamente el Ubuntu con el Mininet-WiFi. También es necesario el código del script, que se encuentra en el apartado 7.1

La única manera de ejecutar el script directamente es escribiendo el comando del Mininet-WiFi en el terminal de Linux, ya que lo único que hace este script es crear una topología personalizada en el emulador.

Para ejecutar la topología personalizada, tenemos que extraer el comando necesario, pero dicho comando no se encuentra en "spine_leaf.py", porque serán los scripts de testeo que llamarán a este script, lo podemos visualizar en la Figura 17.

Entonces extraeremos la línea de código del script "latencia.py", está disponible en el apartado 7.6, pero cualquiera otro script de testeo posee la misma línea:

```
comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --
custom spine_leaf.py --topo mytopo'
```

Para ejecutarlo en el terminar, tenemos que eliminar el código que está en negro, eliminamos "-E" porque no tenemos las variables de entornos declaradas, aplicando las modificaciones, obtenemos:

```
sudo mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --custom
spine_leaf.py --topo mytopo
```

Antes de ejecutarlo necesitamos implementar un algoritmo anti-bucles para que funcione la red, ya que Spine-Leaf es una topología con bucles, para ello tenemos dos maneras de hacerlo:

1. Controlador SDN: Activar algún controlador SDN como ONOS, ryu, nox, opendaylight, etc. Y activar la aplicación del dicho controlador que evite los bucles.
2. Script ECMP.py: Este script incorpora los flujos necesarios para el correcto funcionamiento en una topología Spine-Leaf.

4.1.2.1 Procedimiento

Para evitar los bucles se utilizará el script "ECMP.py", los distintos pasos para su ejecución son:

1. Abrimos el terminal.
2. Accedemos al directorio que se encuentran todos los scripts, mediante el siguiente comando "cd":

   ```
   cd Simulaciones
   ```

   Se debe aclarar que no son simulaciones en realidad, ya que Mininet-WiFi es un emulador, el nombre del directorio es erróneo.
3. Antes de ejecutar el comando tenemos que comprobar que las valores por defectos de las variables de entorno coincidan en los scripts de "ECMP.py" y "spine_leaf.py".
   En la Figura 21 podemos observar las variables de entorno del script "spine_leaf.py", que es que crea la topología personalizada.

```
13      # Leer los valores de las variables de entorno
14      capa_switch = list(map(int, os.getenv('CAPA_SWITCH', '4,4').split(',')))
15      num_host = int(os.getenv('NUM_HOST', '4'))
16      hosts_adyacentes = int(os.getenv('HOSTS_ADYACENTES', '0'))
```

Figura 21: Variables de entorno del "spine_leaf.py"

En la Figura 22 observamos lo mismo, pero del script "ECMP.py", como podemos observar, los valores por defecto coinciden y, por tanto, funcionaria correctamente.

```
64    # Leer los valores de las variables de entorno
65    capa_switch = list(map(int, os.getenv('CAPA_SWITCH', '4,4').split(',')))
66    num_host = int(os.getenv('NUM_HOST', '4'))
67    use_ecmp = os.getenv('USE_ECMP', '0') == '1' #busca una variable de entorno llamada 'USE_ECMP'
```

Figura 22: Variables de entorno del "ECMP.py"

Si queremos emular otra configuración de Spine-Leaf, tenemos que cambiar los valores por defectos de ambos scripts, esto es debido que estamos ejecutando dichos scripts manualmente.

4. Procedemos a ejecutar la emulación en Mininet-WiFi:

```
sudo mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --custom
spine_leaf.py --topo mytopo
```

Si todo está instalado correctamente, obtendremos la respuesta del terminal que se visualiza en la Figura 23:

```
sdn@SDN-USER:~$ cd Simulaciones/
sdn@SDN-USER:~/Simulaciones$ sudo  mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --custom spine_leaf.py --topo mytopo
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1-spine s2-spine s3-spine s4-spine s5-leaf s6-leaf s7-leaf s8-leaf
*** Adding links:
(0.005ms delay) (0.005ms delay) (h1, s5-leaf) (100.00Mbit 0.005ms delay) (100.00Mbit 0.005ms delay) (h2, s6-leaf) (100.00Mbit 0.005ms delay) (
100.00Mbit 0.005ms delay) (h3, s7-leaf) (100.00Mbit 0.005ms delay) (100.00Mbit 0.005ms delay) (h4, s8-leaf) (100.00Mbit 0.5ms delay) (100.00Mb
it 0.5ms delay) (s5-leaf, s1-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s5-leaf, s2-spine) (100.00Mbit 0.5ms delay) (100.00Mbit
 0.5ms delay) (s5-leaf, s3-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s5-leaf, s4-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0
.5ms delay) (s6-leaf, s1-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s6-leaf, s2-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5
ms delay) (s6-leaf, s3-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s6-leaf, s4-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms
 delay) (s7-leaf, s1-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s7-leaf, s2-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms d
elay) (s7-leaf, s3-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s7-leaf, s4-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms del
ay) (s8-leaf, s1-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s8-leaf, s2-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay
) (s8-leaf, s3-spine) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (s8-leaf, s4-spine)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 8 switches
s1-spine s2-spine s3-spine s4-spine s5-leaf s6-leaf s7-leaf s8-leaf ...(100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms del
ay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0
.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.
00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (0.005ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.0
0Mbit 0.5ms delay) (100.00Mbit 0.005ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms del
ay) (100.00Mbit 0.005ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit
 0.005ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay) (100.00Mbit 0.5ms delay)
*** Starting CLI:
mininet-wifi>
```

Figura 23: Ejecución del script "spine_leaf.py"

5. Abrimos otra pestaña del terminal y ejecutamos el algoritmo ECMP mediante el comando:

```
sudo python ECMP.py
```

Una vez ejecutado, podemos visualizar como los distintos flujos son definidos, Figura 24.

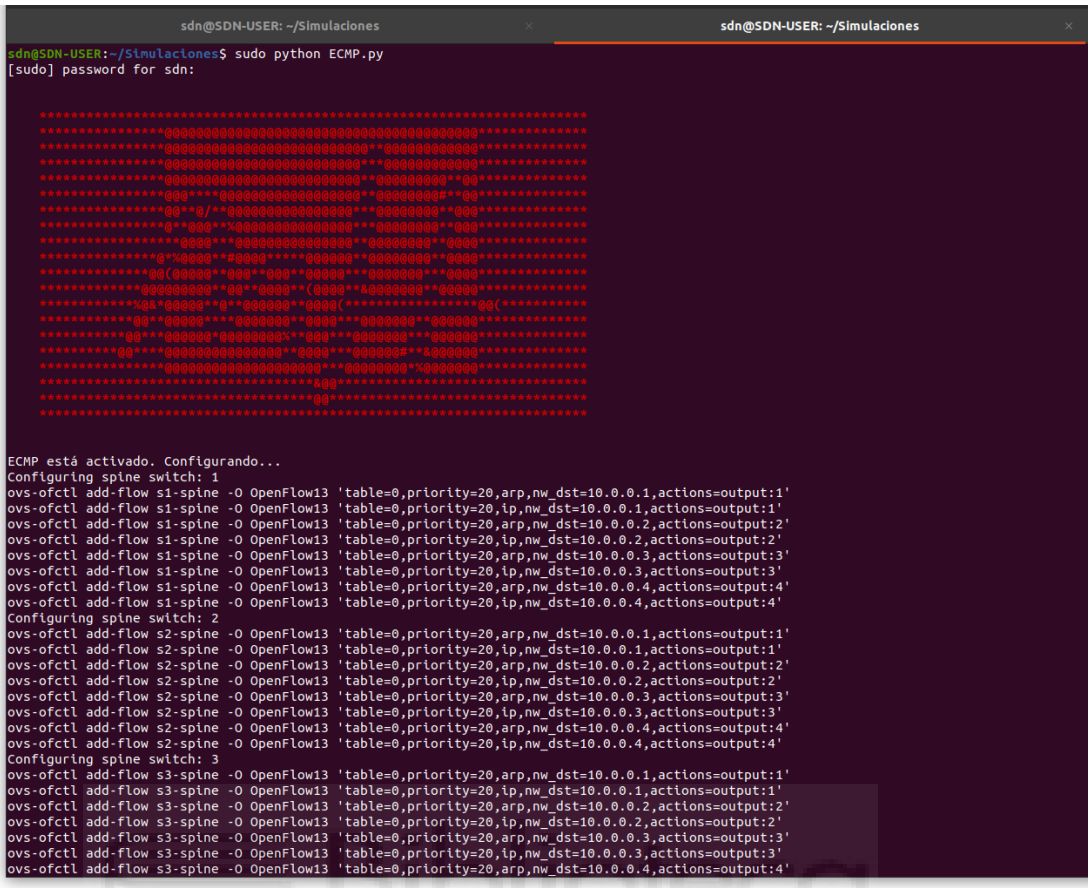


Figura 24: Ejecución del script "ECMP.py"

6. En el emulador Mininet-WiFi procedemos a hacer dos pruebas nativas
    a. Pingall: Comprueba la conectividad entre todos los dispositivos
    b. iperf: Mide el ancho de banda entre dos dispositivos

    Mediante los siguientes comandos:

```
pingall
iperf
```

    Si tenemos conectividad, tendríamos que obtener el resultado de la Figura 25.

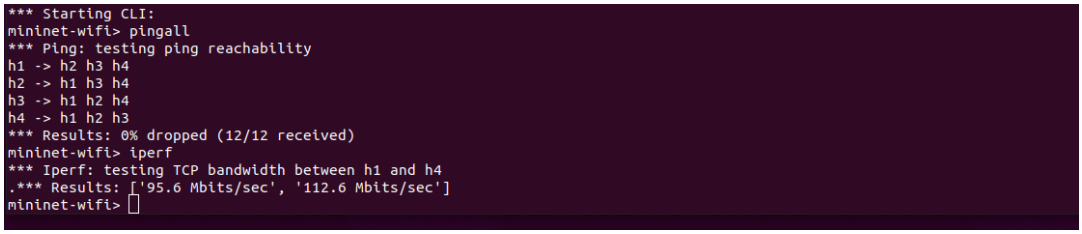


Figura 25: Ejecución del comando iperf y pingall

7. Para hacer las pruebas más profundas, procedemos a abrir los terminales de los hosts en el emulador Mininet-WiFi, con el siguiente comando:

```
xterm h1 h2 h3 h4
```

Este comando nos abrirá las consolas de los dispositivos h1, h2, h3, h4. El resultado se observa en la Figura 26.

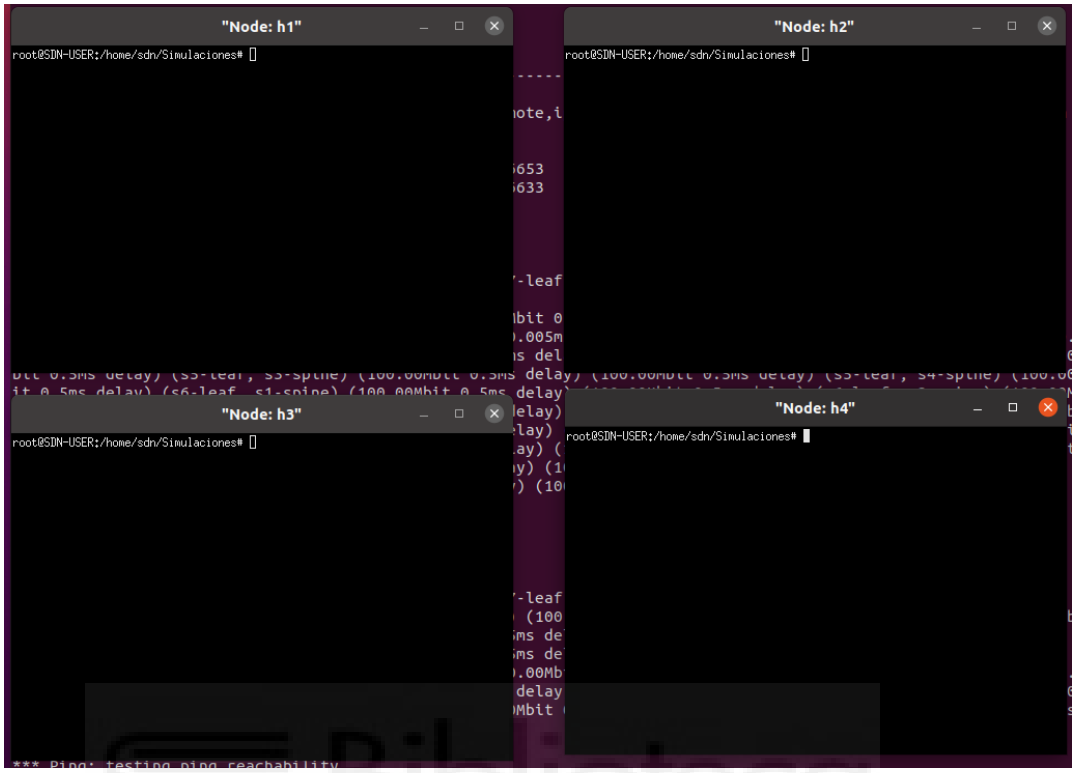


Figura 26: Terminales de los hosts en el emulador

8. Procedemos a hacer una prueba de ancho de banda con flujo distribuido, siendo h1 el servidor del comando Iperf, y los demás clientes.

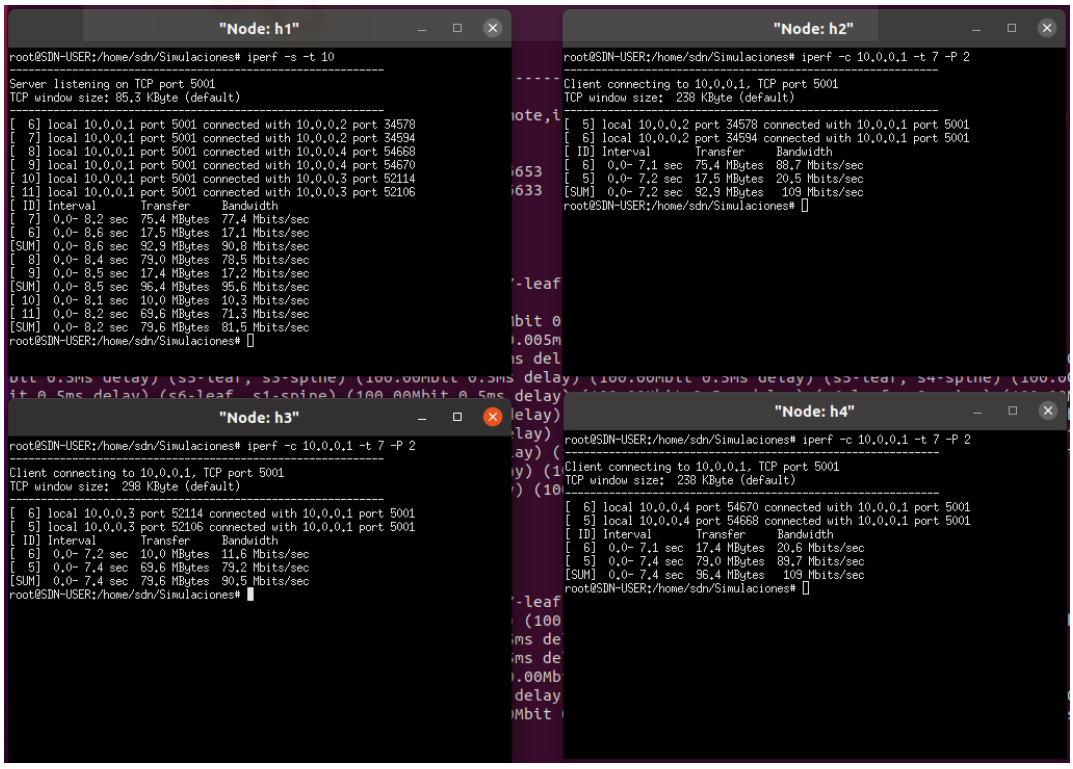


Figura 27: Flujo múltiple manual

9. Para salir del emulador, utilizamos el siguiente comando en la consola Mininet-WiFi, dicho comando también cierra todos los terminales de xterm:

```
exit
```

En la Figura 28 contemplamos el cierre correcto del emulador.

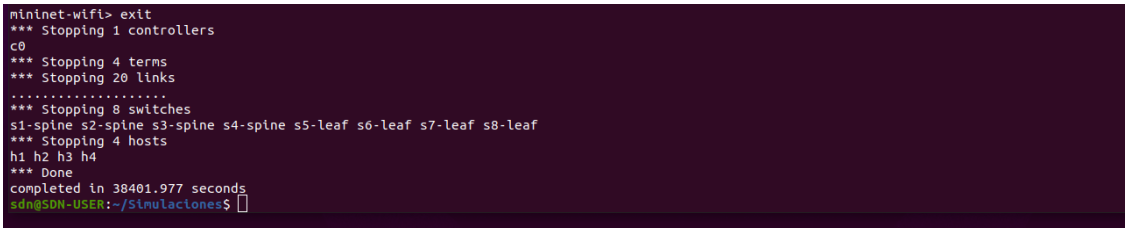


Figura 28: Cierre del emulador Mininet-WiFi

10. Se debe de limpiar el entorno de emulación para evitar errores:

```
sudo mn -c
```

En la Figura 29 se encuentra la limpieza del entorno, este paso es muy importante, ya que nos permite volver a emular correctamente, si saltamos este paso, existe una gran probabilidad que surja un error.

Figura 29: Limpieza del entorno de emulación

4.2 ALGORITMO ECMP

Este script define los flujos necesarios del algoritmo ECMP para la topología Spine-Leaf, es capaz de adaptarse a cualquiera configuración de la topología. Este script es llamado "ECMP.py", el código se encuentra en el apartado 7.3.

Dichos flujos los definimos con el protocolo OpenFlow. Al implementar múltiples rutas para el tráfico proporcionamos mejoras al rendimiento y la resistencia en la red en caso de caídas de enlaces.

Además, permite emular Spine-Leaf sin tener que utilizar un controlador SDN, ya que las topologías con bucles, se necesita de algún algoritmo que los evite, dicho problema esta explicado en el apartado 2.4.

Para poner en marcha el script, es necesario que anteriormente se inicie la emulación en el Mininet-WiFi.

Existe la posibilidad de utilizar controlador SDN con el script "ECMP.py", pero dependerá el funcionamiento del controlador, en el caso del controlador "OpenDayLight", si activamos el script con el controlador activo, dicho script sobrescribirá los flujos y funcionaria correctamente, pero este comportamiento dependerá del controlador.

4.2.1 Funcionamiento

Lo primero de todo es obtener el número de switches de cada capa y los hosts, dichos datos lo obtendremos con las variables de entorno que serán declaradas desde otro script de testeo, con estos datos podemos calcular el número de puertos de cada switch, con ello ya podemos determinar los flujos necesarios para cada switch.

Dichos flujos se definirán de forma automatiza mediante la consola, los comandos para crear los flujos son los siguientes:

```
cmd = f"ovs-ofctl add-flow s{sw}-spine -O OpenFlow13
'table=0,priority=20,{tipo_paquete},nw_dst=10.0.0.{ip},actions=output:{puerto}'"

cmd = f"ovs-ofctl add-group s{sw}-leaf -O OpenFlow13 'group_id=50,type=select,{buckets}'"

cmd = f"ovs-ofctl add-flow s{sw}-leaf -O OpenFlow13
'table=0,priority=10,{tipo_paquete},in_port={puerto},actions=group:50'"

cmd = f"ovs-ofctl add-flow s{sw}-leaf -O OpenFlow13
'table=0,priority=20,{tipo_paquete},nw_dst=10.0.0.{ip},actions=output:{puerto}'"
```

Para entenderlo mejor, explicaremos el ejemplo de dos flujos que viajan desde h4 al s1, con la ayuda de la Figura 30:

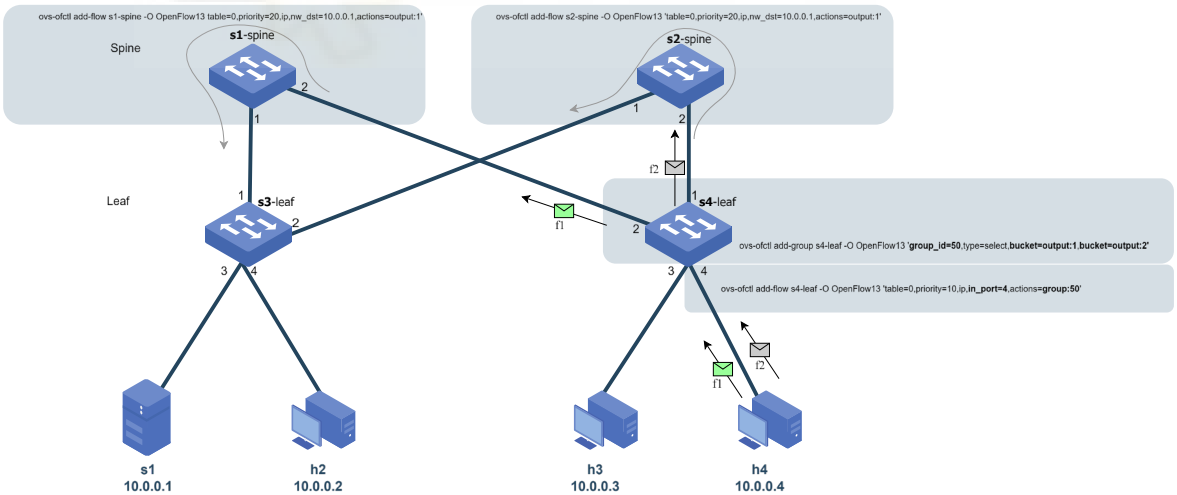


Figura 30: Funcionamiento ECMP mediante OpenFlow

En la Figura 30, visualizamos que el host 4 envía dos flujos a la IP 10.0.0.1, que concretamente es el servidor 1, los flujos en el s4-leaf son recibos por el siguiente comando:

```
ovs-ofctl add-flow s4-leaf -O OpenFlow13 'table=0,priority=10,ip,in_port=4,actions=group:50'
```

Dicho comando asigna los flujos entrantes del puerto 4 al grupo 50.

A continuación, asignamos las salidas del s4-leaf, mediante el siguiente comando:

```
ovs-ofctl add-group s4-leaf -O OpenFlow13 'group_id=50,type=select,bucket=output:1,bucket=output:2'
```

Este comando convierte al grupo 50, en uno de tipo "select", en el switch s4-leaf. Configura un mecanismo de balanceo de carga en los switches Leaf, permitiendo distribuir el tráfico entre múltiples puertos de salida. Dichos puertos los asignamos mediante la siguiente expresión, "bucket=output:1,bucket=output:2", podemos observar cuales son los puertos con la ayuda de la Figura 30.

Cuando se envía un paquete a este grupo, el switch utiliza una función de hash para seleccionar el puerto(bucket) entre cual enviar el paquete, en este caso existen 2 puertos disponibles que son el puerto 1 y el 2, en la Figura 30 lo podemos visualizar fijándose en el s4-leaf.[50]

Ahora los flujos son recibidos por los switches Spine, este los tratara con el siguiente comando:

```
ovs-ofctl add-flow s1-spine -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.1,actions=output:1'
```

Básicamente lo que hace es a aquellos paquetes que tienen como destino la IP 10.0.0.1 reenviarlo por el puerto 1, y con la misma lógica con todos los puertos, es decir, a cada puerto del switch Spine se asignara un rango de IP. Como en Mininet-WiFi por defecto se asignan las IP de forma ordenada, se calcula que IP se encuentran en cada switch Leaf, mediante el número de switches Leaf y el número de host.

Entonces, el script "ECMP.py" definiría siguientes flujos para la topología de Figura 30:

```
ECMP está activado. Configurando...
Configuring switch: 1
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.1,actions=output:1'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.1,actions=output:1'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.2,actions=output:1'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.2,actions=output:1'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.3,actions=output:2'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.3,actions=output:2'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.4,actions=output:2'
ovs-ofctl add-flow s1 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.4,actions=output:2'
Configuring switch: 2
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.1,actions=output:1'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.1,actions=output:1'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.2,actions=output:1'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.2,actions=output:1'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.3,actions=output:2'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.3,actions=output:2'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.4,actions=output:2'
ovs-ofctl add-flow s2 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.4,actions=output:2'
Configuring switch: 3
ovs-ofctl add-group s3 -O OpenFlow13 'group_id=50,type=select,bucket=output:1,bucket=output:2'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=10,arp,in_port=3,actions=group:50'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=10,ip,in_port=3,actions=group:50'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=10,arp,in_port=4,actions=group:50'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=10,ip,in_port=4,actions=group:50'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.1,actions=output:3'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.1,actions=output:3'
```

```
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.2,actions=output:4'
ovs-ofctl add-flow s3 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.2,actions=output:4'
Configuring switch: 4
ovs-ofctl add-group s4 -O OpenFlow13 'group_id=50,type=select,bucket=output:1,bucket=output:2'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=10,arp,in_port=3,actions=group:50'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=10,ip,in_port=3,actions=group:50'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=10,arp,in_port=4,actions=group:50'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=10,ip,in_port=4,actions=group:50'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.3,actions=output:3'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.3,actions=output:3'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=20,arp,nw_dst=10.0.0.4,actions=output:4'
ovs-ofctl add-flow s4 -O OpenFlow13 'table=0,priority=20,ip,nw_dst=10.0.0.4,actions=output:4'
```

Si analizamos los parámetros de los comandos del script “ECMP.py”, todos los parámetros son conocidos menos:

1. **Prioridad:** que es un valor numérico (0-65535 en OpenFlow 1.3) que determina el orden en que se evalúan las reglas de flujo en un switch. Las reglas con mayor prioridad se aplican primero cuando coinciden con un paquete entrante.
2. **Tabla:** En OpenFlow, un switch puede tener múltiples tablas de flujo, numeradas secuencialmente empezando desde 0. La tabla 0 es la tabla de flujo principal y típicamente la primera en ser consultada cuando un paquete ingresa al switch.

Se puede observar que le han definidos el doble de flujos, uno para el protocolo IP y otro para ARP (Address Resolution Protocol), el protocolo ARP se utiliza para mapear direcciones IP a direcciones MAC (Media Access Control) en redes locales. Cuando un dispositivo necesita enviar datos a una IP en su red local, pero desconoce la MAC correspondiente, emite un paquete ARP de solicitud (broadcast[3]) para identificar que dispositivo posee esa IP. El dispositivo con esa IP responde con un paquete ARP de respuesta (unicast[4]) que contiene su dirección MAC. [51]

El funcionamiento interno del script se detalla en el siguiente diagrama de flujo, en la Figura 31.

[3] Una respuesta broadcast es aquel que es dirigido a todos los dispositivos de la red.
[4] Una respuesta unicast es aquella que es dirigida solamente a un dispositivo en concreto.

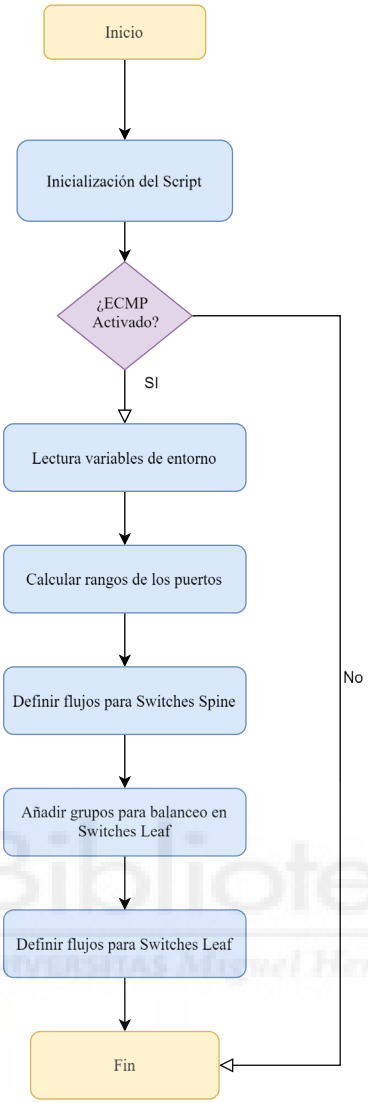


Figura 31: Diagrama de flujo del script "ECMP.py"

Para programar el algoritmo para Spine-Leaf se ha utiliza como guía de inicio, el proyecto de GitHub que implementaba para Fattree network con dos configuraciones específicas.[52]

También se ha utilizado otras fuentes para entender el funcionamiento del protocolo OpenFlow. [53], [54].

4.2.2 Ejecución manual

A lo largo del proyecto, se utilizará este algoritmo mediante el argumento "--ECMP" al ejecutar algún script de testeo, pero también es posible de forma manual.

Se procede a explicar los pasos necesarios para su ejecución de manera manual:

1. Mediante el terminar nos ubicamos en la carpeta de los scripts y ejecutamos Mininet-WiFi.

```
cd Simulaciones
sudo mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 --custom
spine_leaf.py --topo mytopo
```

En la Figura 32 observamos la correcta ejecución del emulador Mininet-WiFi.

Figura 32: Ejecución del emulador

2. Abrimos otro terminal y ejecutamos el script:

```
sudo python ECMP.py
```

En la Figura 33 observamos el correcto funcionamiento del script “ECMP.py”. Es necesario haber copiado el código del script anteriormente, disponible en el apartado 7.3.

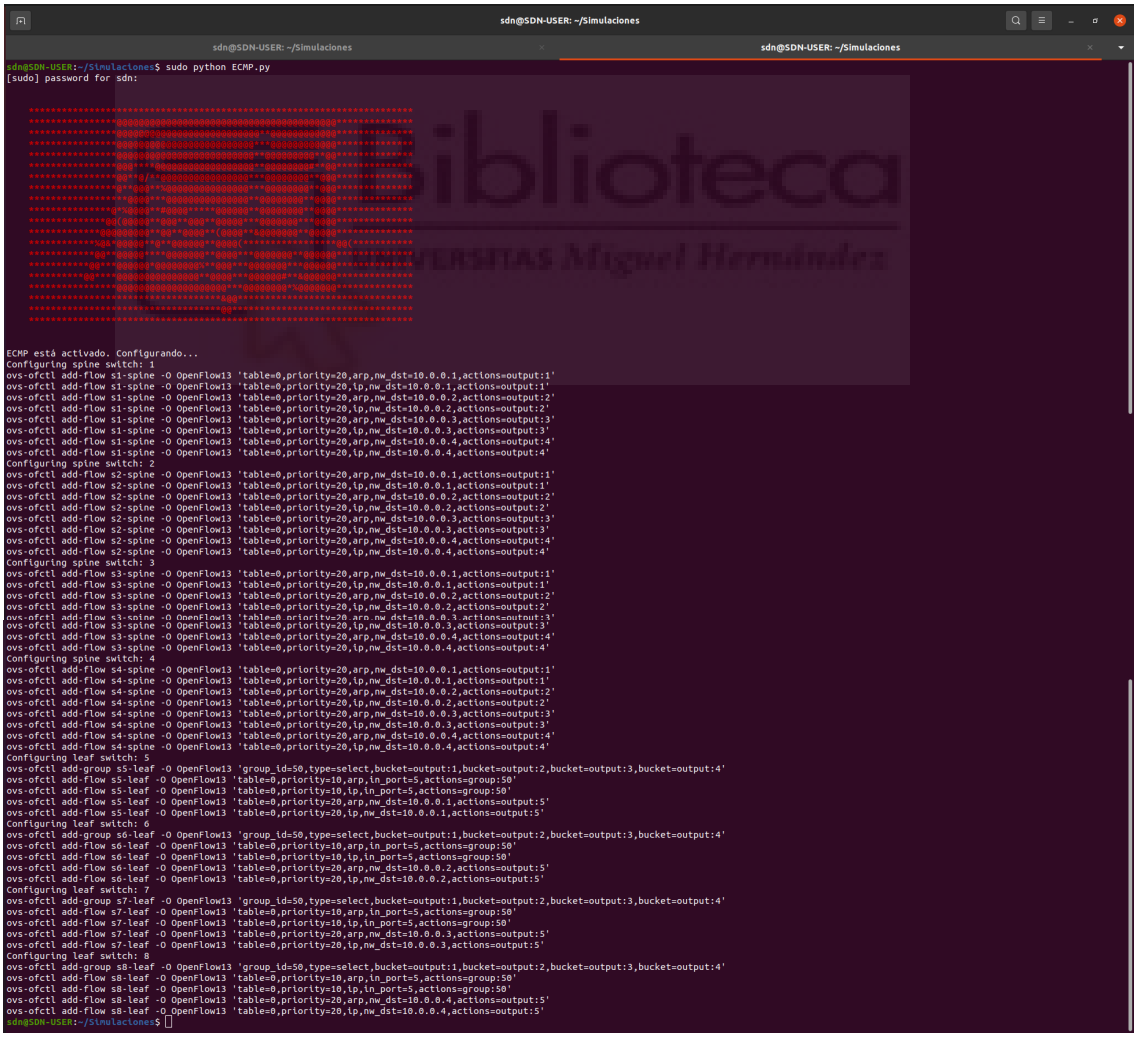


Figura 33: Salida de consola del “ECMP.py”

3. Ejecutamos “pingall” en el Mininet-WiFi para comprobar si la ejecución del algoritmo es correcta:

```
pingall
```

En la Figura 34, observamos que todos los hosts tienen conectividad entre ellos. Si no hubiese conectividad, en vez del número de host, saldría una “x”.

Figura 34: Comprobación del funcionamiento mediante pingall

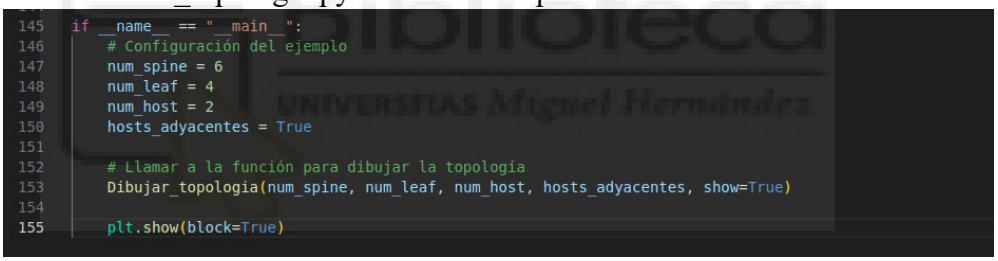
4.3 VISUALIZACIÓN DE LA TOPOLOGÍA

Este script representa gráficamente la topología Spine-Leaf a emular. Para ello, lee las variables de entorno necesarias para graficar la configuración de Spine-Leaf deseada. Muestra la imagen y lo guarda en el directorio que se ubica el script. Este script es llamado “visualizador_topologia.py” y el código se encuentra en el apartado 7.4.

4.3.1 Ejecución manual

Este script será utilizado por los scripts de testeo mediante el argumento “--g”, pero también podemos utilizarlo de forma manual, es este apartado se explicará cómo hacerlo:

1. Mediante un editor modificamos las variables del script “visualizador_topologia.py” a la deseada para mostrar:



```
if __name__ == "__main__":
    # Configuración del ejemplo
    num_spine = 6
    num_leaf = 4
    num_host = 2
    hosts_adyacentes = True

    # Llamar a la función para dibujar la topología
    Dibujar_topologia(num_spine, num_leaf, num_host, hosts_adyacentes, show=True)

    plt.show(block=True)
```

Figura 35: Variables del script “visualizador_topologia.py”

2. Para visualizar correctamente, se debe descargar los mismos iconos o distintos, pero deben llamarse de la siguiente manera:
    a. pc.png
    b. switch.png
    c. servidor.png

    El enlace de la descarga de cada imagen se encuentra en el código fuente, apartado 7.4. Es importante usar el formado “.png”, ya que el formado “.svg” no funciona en el script.

3. Accedemos al directorio de los scripts y lo ejecutamos:

```
cd Simulaciones/
sudo python3 visualizador_topologia.py
```

En la Figura 36, observamos la ejecución de los comandos.

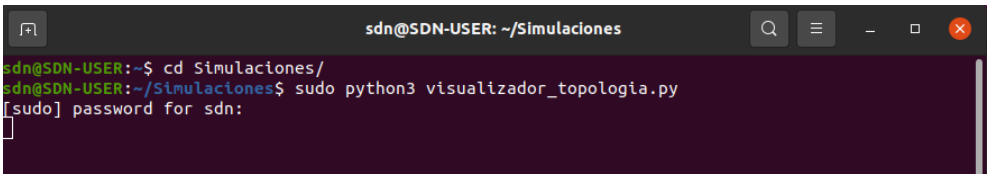


Figura 36: Ejecución de script "visualizador_topologia.py"

El resultado debe ser el que se muestra en la Figura 37.

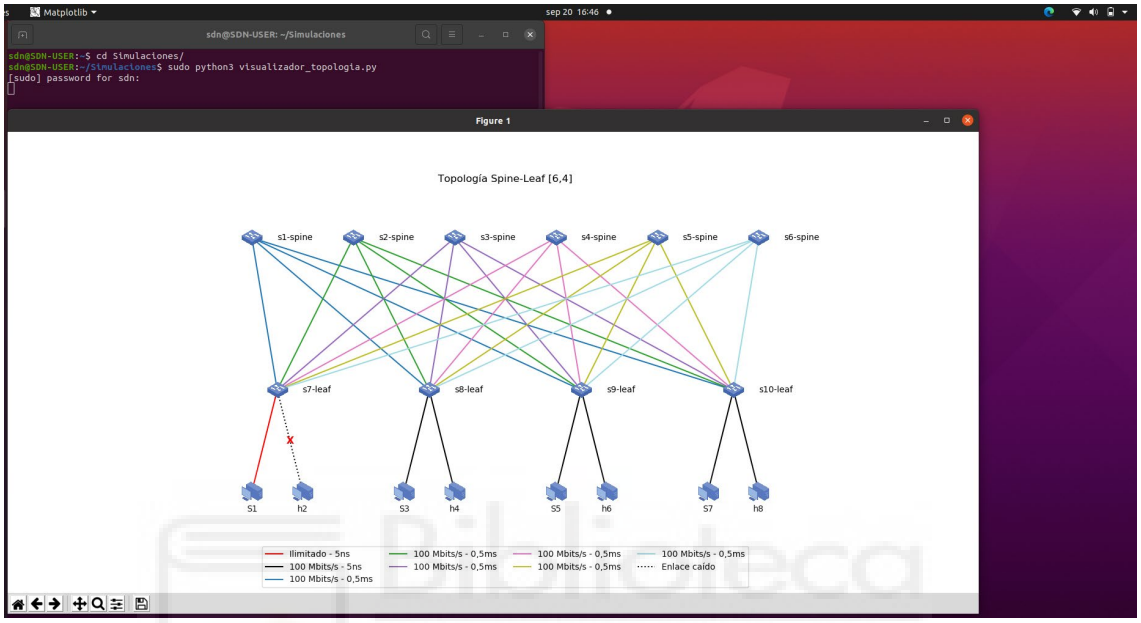


Figura 37: Resultado del script "visualizador_topologia.py"

La imagen se guarda automáticamente en el directorio del script.

4.4 COMPLEMENTOS

Este script se encarga de la lectura de los argumentos que le proporcionamos por el terminal. Y también muestra el logo de UMH por la salida del terminal. El script se nombrará como "complementos.py". El código se encuentra en el apartado 7.5.

El código es bastante repetitivo, explicaremos un fragmento:

```
    parser.add_argument('--s', '--capa_switch', type=str, default='2,2',
                        help='Capas de switches separadas por comas (por defecto: 2,2)')

    parser.add_argument('--h', '--n_host', type=int, default=10,
                        help='Número de hosts (por defecto: 10)')
```

El primer argumento, "--s" o "--capa_switch", permite al usuario especificar las capas de switches, esperando una cadena de texto con valores separados por comas (por defecto '2,2').

El segundo argumento, "--h" o "--n_host", permite al usuario definir el número de hosts en la red, esperando un valor entero (por defecto 10).

Estos argumentos permiten personalizar la emulación sin tener que modificar el código fuente.

4.5 LATENCIA

El script llamado “latencia.py” se encarga de medir las latencias de todos los hosts que forman parte de la topología. Dicha latencia esta vista desde el host 1, que lo trataremos como servidor, por lo tanto, todos los hosts, lanzaran una prueba de “ping” hacia el servidor de forma ordenada, del dispositivo más cercano al más lejano.

Cada vez que se realizada el ping, se guarda el resultado. Este procedimiento se realiza con todos los dispositivos. Una vez recopilados todos los datos necesarios, se procede a graficar los resultados.

Para lanzar los pings se utilizará la siguiente línea de código en bucle:

```
child.sendline(f'h{ i + 1} ping h1 -c 10 -i 0,2')
```

Para configurar los parámetros de la topología y de la emulación, lo hacemos mediante los argumentos de consola, dichos parámetros para este script son:

Parámetro	Forma corta	Forma larga	Tipo	Valor por defecto	Descripción
Capas de switches	--s	--capa_switch	str	'2,2'	Numero de switches por capa separadas por comas
Número de hosts	--h	--n_host	int	8	Número de hosts
ECMP	--e	--ECMP	bool	False	Activar algoritmo ECMP
Visualización	--g	--graficar	bool	False	Visualización de la topología
Hosts adyacentes	--a	--hosts_adyacentes	bool	False	Desactivar hosts adyacentes del switch que se encuentra h1
Ayuda	--help	--help	bool	-	Muestra el mensaje de ayuda

Tabla 6: Argumentos del script “latencia.py”

Breve explicación de cada argumento de la Tabla 6:

- El argumento “--capa_switch”: Asigna el número de switches en cada capa, el primer valor se trata de la capa Spine y el siguiente de la capa Leaf.
- El argumento “--n_host”: Número total de dispositivos en la emulación, incluido el servidor(h1).
- El argumento “--ECMP”: Activación del algoritmo ECMP.
- El argumento “--graficar”: Grafica la topología Spine-Leaf una vez terminada la emulación.
- El argumento “--hosts_adyacentes”: Desactiva aquellos host adyacentes al servidor(h1), se explicará su utilidad en el apartado 4.8.
- El argumento “--help”: Proporciona información sobre los argumentos disponibles del script.

A continuación, se mostrará el diagrama de flujo para entender el funcionamiento interno del script, Figura 38.

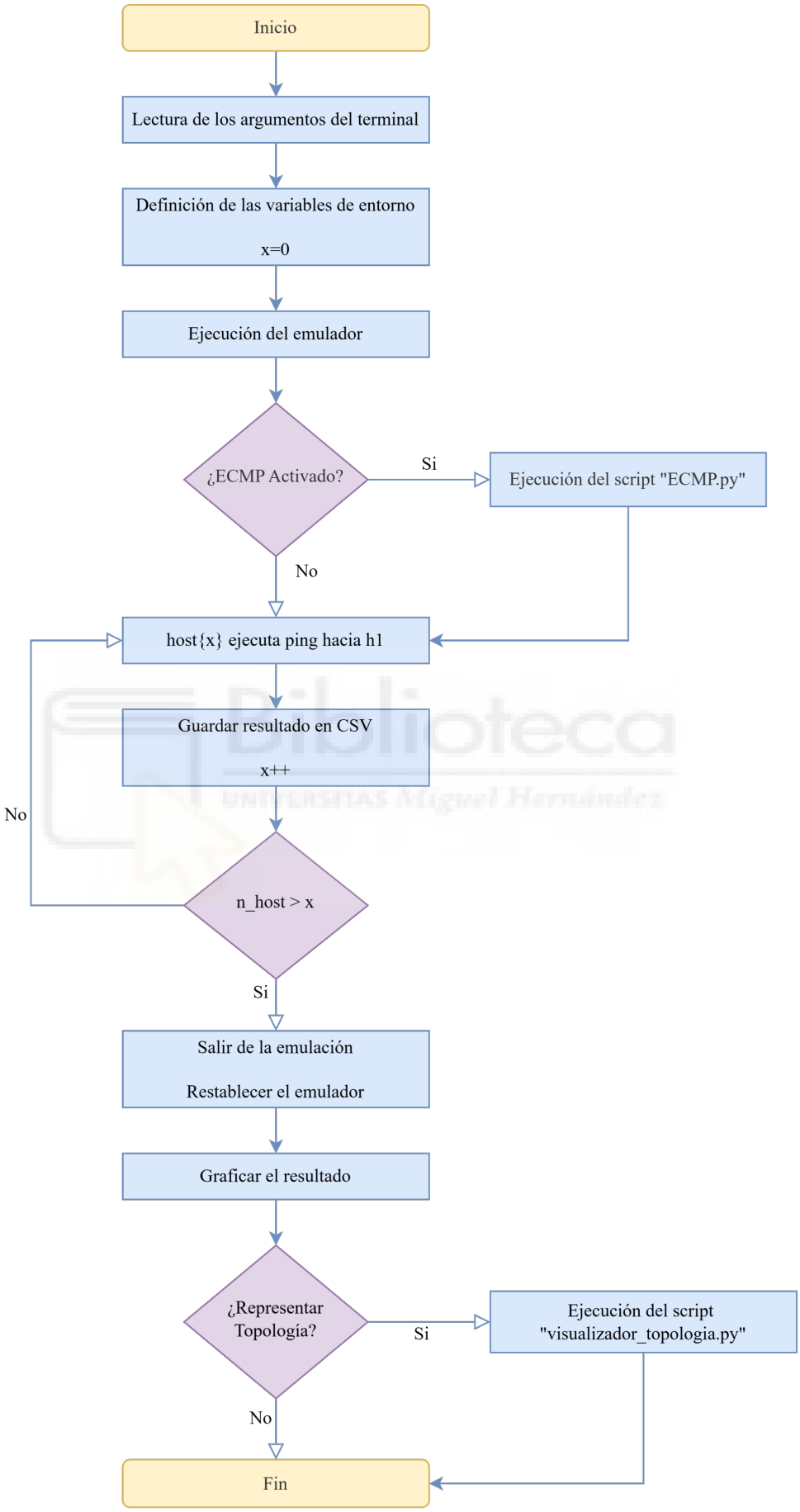


Figura 38: Diagrama de flujo del script "latencia.py"

4.5.1 Ejecución con argumentos

Antes de la ejecución se debe de crear el script, el código se encuentra en el apartado 7.6, se aconseja crear todos los scripts o al menos los scripts auxiliares, apartado 4, ya que existen dependencias entre ellos, todos los códigos se encuentran en el apartado 7.

1. Accedemos al directorio de los scripts y ejecutamos la ayuda para ver los distintos argumentos disponibles:

```
cd Simulaciones/
sudo python3 latencia.py --help
```

En la Figura 39, observamos los distintos argumentos que disponemos para la personalización de la topología Spine-Leaf y de la emulación.

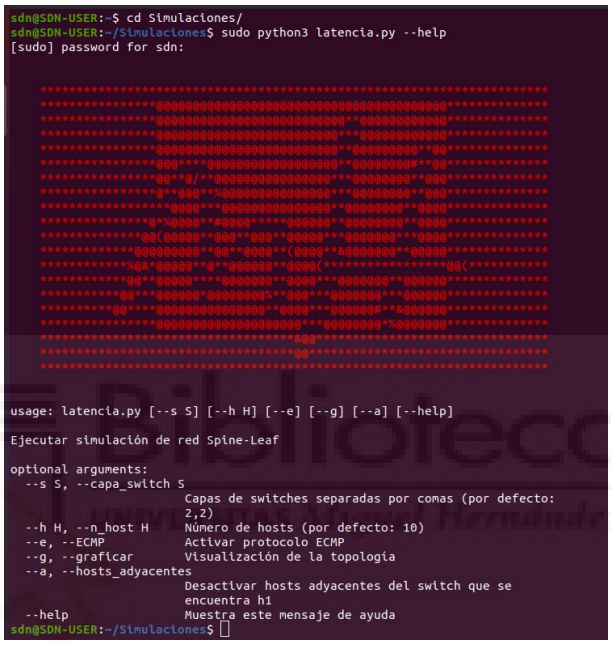


Figura 39: Argumento de ayuda en el script "latencia.py"

2. Ejecutamos al script indicando el número de host y el número de switch de cada capa, como no estamos utilizando controlador SDN, utilizamos el algoritmo ECMP:

```
sudo python3 latencia.py --h 10 --s 4,2 --ECMP
```

En el comando del terminal anterior, vemos que el argumento "--h" indica el número de host y "--s" indica el número de switches de cada capa.
Cabe destacar que podemos ejecutarlo sin argumentos, en ese caso, utiliza los valores por defecto (sin el controlador, hay que incluir --ECMP para disponer de conectividad).

Figura 40: Ejecución del script "latencia.py"

3. Una vez finalizado la ejecución de la emulación, el script mostrara una gráfica con los resultados obtenidos, Figura 41.

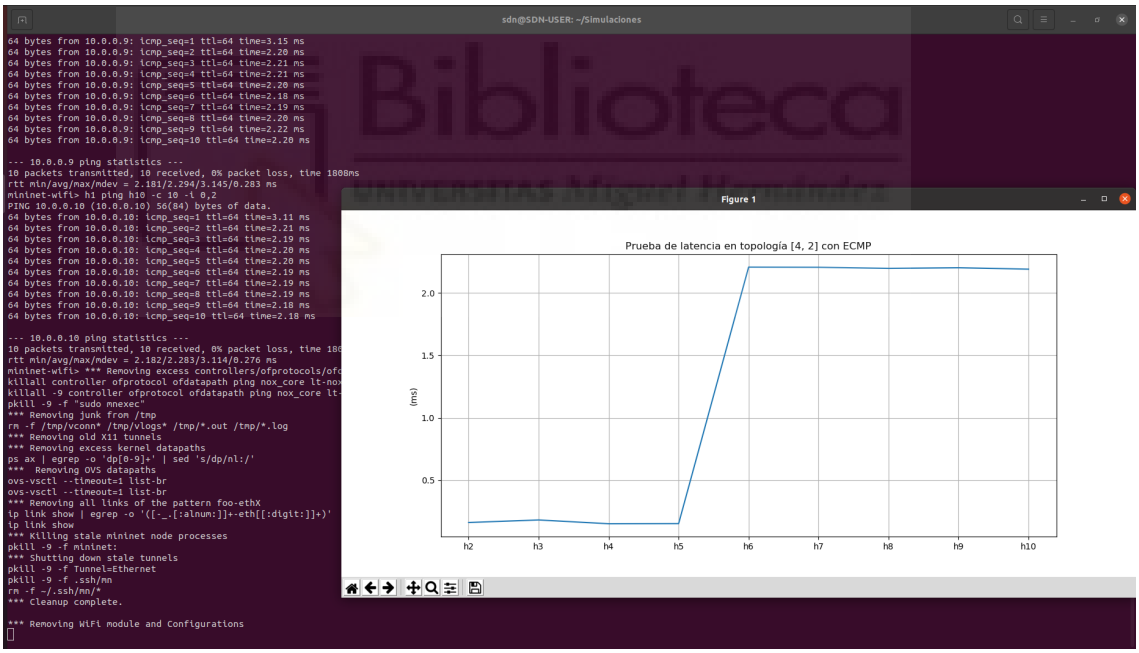


Figura 41: Resultado del script "latencia.py"

También disponemos la posibilidad de visualizar la topología con el argumento "--g" y eliminar los enlaces adyacentes al servidor con "--a":

```
sudo python3 latencia.py --h 10 --s 4,2 -ECMP --a --g
```

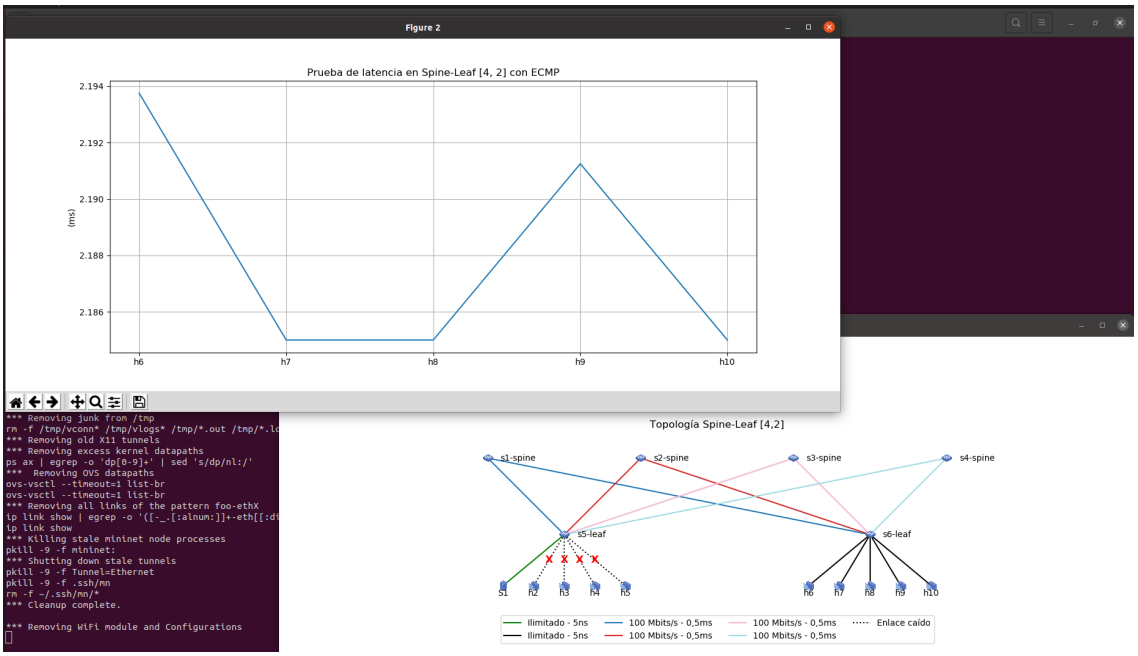


Figura 42: Resultado del script "latencia.py" con argumentos extras

En la Figura 42, observamos el resultado, si nos fijamos en el título de la gráfica, podemos ver que nos especifica el uso del algoritmo ECMP. En la topología mostrada, al estar activado el argumento "--a", los enlaces adyacentes al servidor están desactivados.

Aunque parezca que la variación de la latencia es muy grande en la Figura 42, no es así, como hemos desactivado los hosts adyacentes, los restantes tienen una latencia muy parecida, si nos fijamos en el 'eje Y', la variación es muy pequeña.

4.6 FLUJO ÚNICO

Este script se encarga de testear el ancho de banda entre el primer host (servidor) y último host. Para ello utilizaremos la herramienta Iperf para medir el ancho de banda. Dicho script se llamará "ancho_banda_flujo_unico.py" y el código se encuentra en el apartado 7.7.

Para este objetivo, tenemos que asignar el servidor Iperf al host 1. Para ello empleamos el siguiente comando:

```
ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")
```

Observamos que tiene el argumento "-t", esto es el tiempo que actuara como servidor, esto es necesario para poder controlar los tiempos en el script. Procuremos que dicho tiempo sea mayor que los tiempos en el cliente, esto para garantizar la integridad de los resultados.

El fragmentó "`&>log/S{k}.log`" indica que la salida del comando será redirigida al archivo log correspondiente. El "`&`" final, indica que se ejecuta en segundo plano, lo que nos permite hacer otras operaciones al mismo tiempo.

Una vez que ya tenemos el servidor asignado, desde el ultimo host de la topología, haremos la prueba, la línea de código que ejecuta el comando en el cliente es la siguiente:

```
ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {size_Mbits}M -t {tiempo_prueba}
&>log/h{n_host}.log&')
```

En este caso tenemos el parámetro extra del tamaño, esto es básicamente para poder graficar el resultado en una gráfica.

Para configurar los parámetros de la topología y de la emulación, lo hacemos mediante los argumentos, que son los siguientes:

Parámetro	Forma corta	Forma larga	Tipo	Valor por defecto	Descripción
Capas de switches	--s	--capa_switch	str	'2,2'	Numero de switches por capa separadas por comas
Número de hosts	--h	--n_host	int	8	Número de hosts
ECMP	--e	--ECMP	bool	False	Activar algoritmo ECMP
Visualización	--g	--graficar	bool	False	Visualización de la topología
Hosts adyacentes	--a	--hosts_adyacentes	bool	False	Desactivar hosts adyacentes del switch que se encuentra h1
Tiempo de prueba	--t	--tiempo_prueba	int	5	Tiempo de prueba en segundos
Carga de tráfico máxima	--c	--carga_trafico_max	int	1000	Carga de tráfico máxima en Mbps
Número de pasos	--p	--pasos	int	4	Número de iteraciones del bucle (precisión de la gráfica)
Ayuda	--help	--help	bool	-	Muestra el mensaje de ayuda

Tabla 7: Argumentos del script "ancho_banda_flujo_unico.py"

Breve explicación de los nuevos argumentos disponibles:

- El argumento "--tiempo_prueba": Es el tiempo asignado al comando Iperf de los clientes, es decir, es el tiempo que tarda en medir el ancho de banda, el tiempo del servidor será un 50% mayor que el tempo en los clientes, para evitar que se cierre antes que los clientes acaben su sesión.
- El argumento "--carga_trafico_max": Como define su nombre, es la carga máxima que se introduce a la topología.
- El argumento "--pasos": Define la cantidad de iteraciones que se tendrá al generar tráfico, mientras el valor de "pasos" es mayor, la prueba será más precisa porque habrá más puntos de muestreo.

En la Figura 43, nos muestra el funcionamiento interno del script gracias al diagrama de flujo.

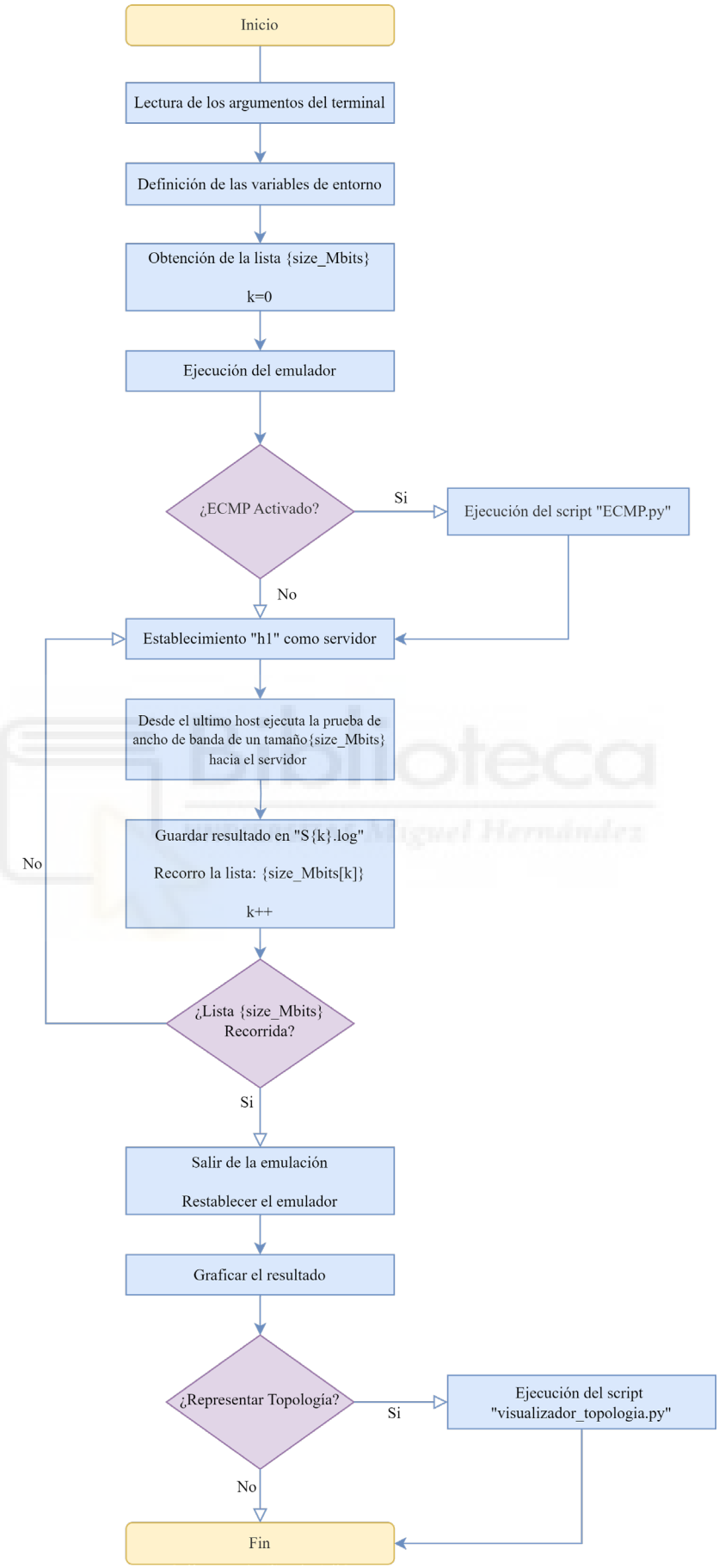


Figura 43: Diagrama de flujo del script "ancho_banda_flujo_unico.py"

4.6.1 Varias configuraciones

Este es otro script que permite realizar la prueba del ancho de banda en flujo único en varias configuraciones de la topología Spine-Leaf. Este script será llamado "ancho_banda_flujo_unico_N_configuraciones.py".

Lo único realmente novedoso es la siguiente modificación a la variable "capa_switch":

```
capas_switch=[2,4],[2,8],[4,4],[8,4]
```

En este caso observamos que alberca una lista de listas (tupla en Python), que son las diferentes configuraciones que se utilizan para cada emulación, el script está pensado para 4 configuraciones distintas. El código se adaptó para las distintas emulaciones, de tal manera que se muestre todos los resultados en un único gráfico.

En el tema de argumentos, el único que cambia es el de "--capas_switch" que en esta ocasión se debe introducir una lista de configuraciones de los switches, en el apartado 4.6.3, se muestra un ejemplo.

4.6.2 Ejecución con argumentos

Primero se explicará el uso del script "ancho_banda_flujo_unico.py", y en el apartado 4.6.3, se procederá a explicar el script "ancho_banda_flujo_unico_N_configuraciones.py".

1. Accedemos al directorio de los scripts y ejecutamos el script, activamos el algoritmo ECMP para tener conectividad:

   ```
   cd Simulaciones/
   sudo python3 ancho_banda_flujo_unico.py --h 8 --s 4,2 --ECMP
   ```

   Si todo está correcto, nos debe mostrar la Figura 44.

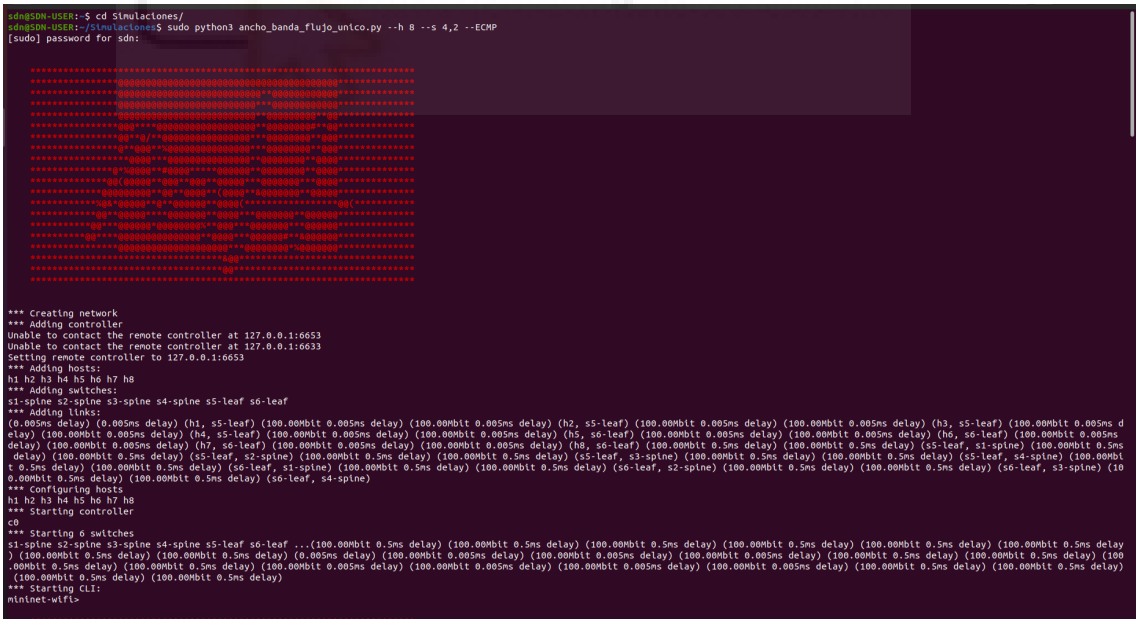


Figura 44: Ejecución del script "ancho_banda_flujo_unico.py"

2. Una vez finalizado la ejecución de la emulación, el script mostrara una gráfica con los resultados obtenidos, Figura 45, en dicha figura, podemos observar la consola del terminal detrás de la gráfica, allí en color rojo muestran los resultados de cada iteración.

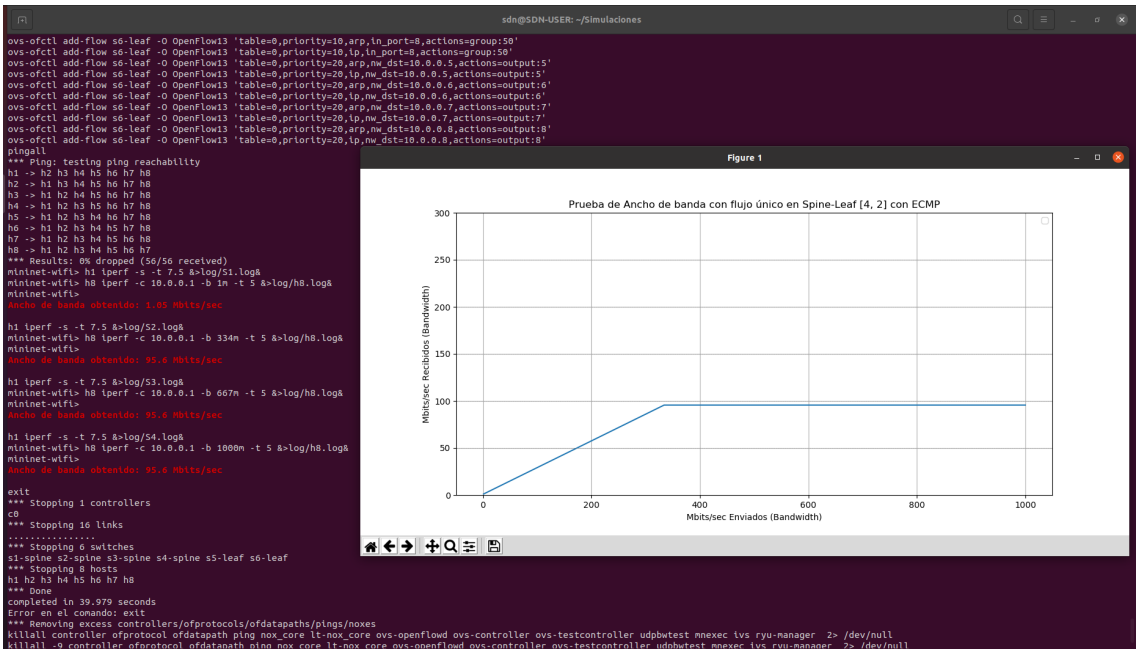


Figura 45: Resultado del script "ancho_banda_flujo_unico.py"

3. También disponemos la posibilidad de añadir más parámetros:

```
sudo python3 ancho_banda_flujo_unico.py --h 8 --s 4,2 --ECMP --t 2 --c 200 --p 8
```

En este caso se ha disminuido el tiempo de cada medición ancho de banda de 5 a 2 segundos, la carga máxima se ha disminuido a 200Mbts/s y se ha aumentado el número de iteraciones a 8.

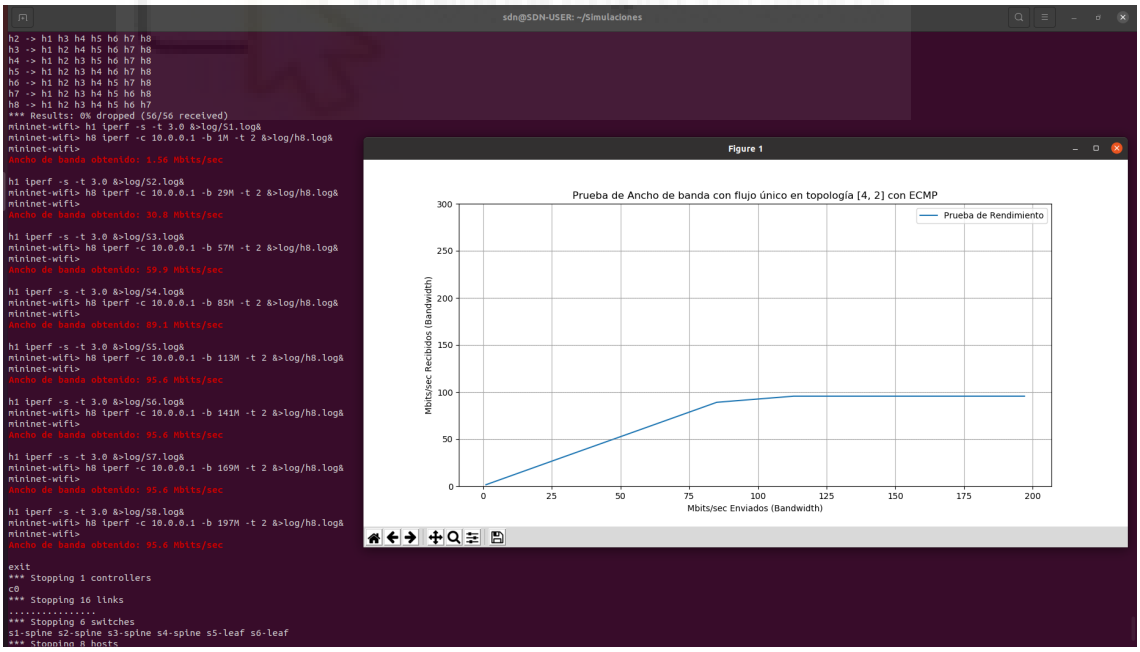


Figura 46: Resultado del script "ancho_banda_flujo_unico.py" con argumentos

La Figura 46 presenta el resultado de la ejecución del script "ancho_banda_flujo_unico.py" con argumentos.

4.6.3 Ejecución de N configuraciones con argumentos

Ahora se explicará la ejecución del script que se encarga de ejecutar 4 pruebas de ancho de banda con flujo único a la vez y graficar en una única grafica. Dicho script se llamará "ancho_banda_flujo_unico_N_configuraciones.py" y el código se encuentra en el apartado 7.8.

Al no utilizar un controlador SDN, podemos utilizar el parámetro ECMP:

```
sudo python3 ancho_banda_flujo_unico_N_configuraciones.py --ECMP
```

El resultado del script lo podemos visualizar en la Figura 47, como ya se ha mencionado, obtenemos los resultados de las 4 configuraciones.

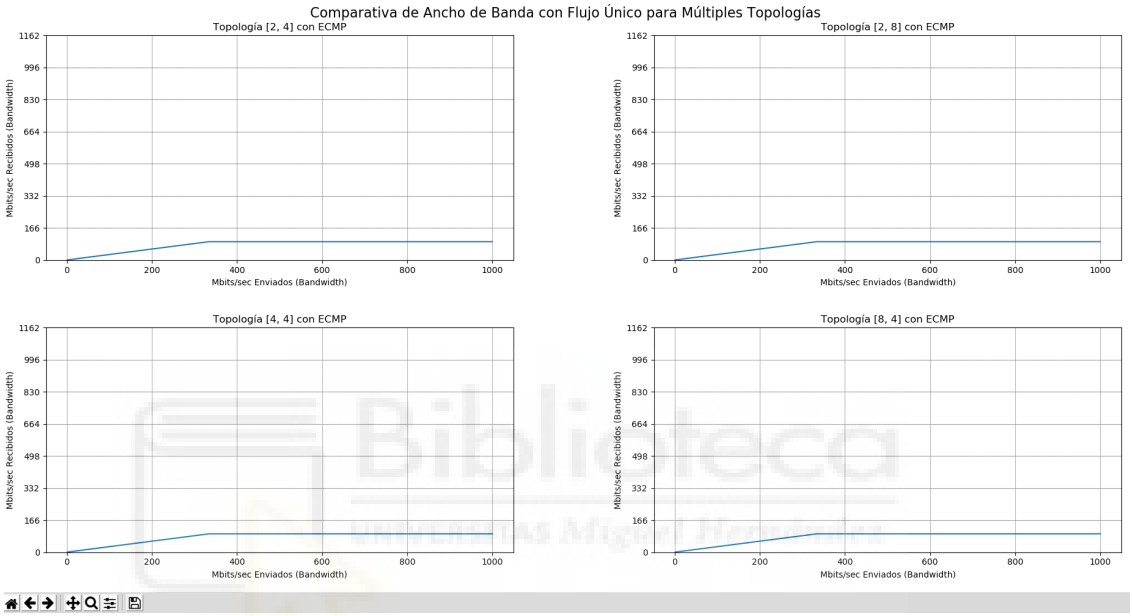


Figura 47: Resultado del script "ancho_banda_flujo_unico_N_configuraciones.py"

A continuación, se muestra un ejemplo con distintos argumentos que se pueden configurar, como el número de host, las configuraciones de topología, la carga de trabajo máxima, el número de iteraciones sobre el tráfico máximo, el tiempo de la prueba, y que nos muestre las topologías, con el siguiente comando:

```
sudo python3 ancho_banda_flujo_unico_N_configuraciones.py --h 8 --s 2,2 4,2 4,4 8,4 --ECMP --c
400 --p 6 --t 2 --g
```

El resultado de la personalización con los argumentos, podemos verla en la Figura 48.

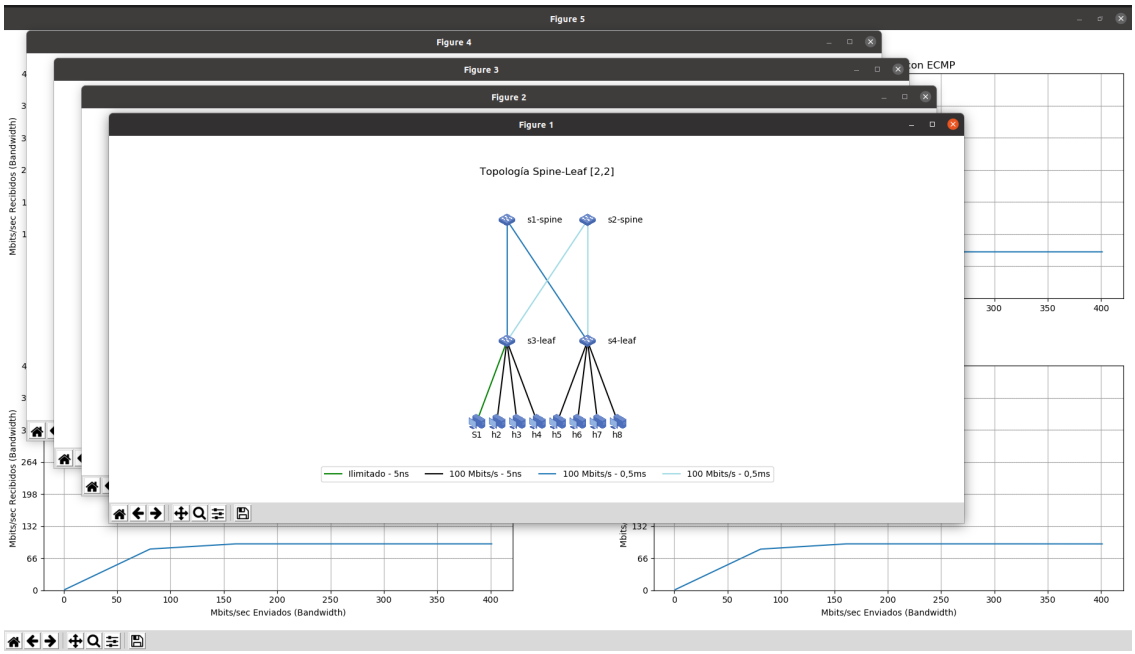


Figura 48: Resultado del script "ancho_banda_flujo_unico_N_configuraciones.py" con argumentos

En la Figura 48, al tener el argumento "--g", nos ha dibujado las 4 configuraciones de Spine-Leaf.

4.7 FLUJO MÚLTIPLE

Es parecido en el flujo único, pero su diferencia primordial es que envía N flujos paralelos simultáneos al servidor Iperf. De esta manera podemos aprovechar el algoritmo ECMP. El script es llamado "ancho_banda_flujo_multiple.py" y el código se encuentra en el apartado 7.9.

El script tiene ciertas diferencias con el de flujo único, pero la más importante es la siguiente:

```
ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {size_Mbits}M -t {tiempo_prueba} -P
{flujos_paralelos}  &>log/h{n_host}.log&')
```

Si nos fijamos, tiene un nuevo argumento, que es "-P", que indica el número de flujos que vamos a ejecutar hacia la IP correspondiente, en nuestro caso es 10.0.0.1 que corresponde al servidor.

Para configurar los parámetros de la topología y de la emulación, lo hacemos mediante los argumentos, que son los siguientes:

Parámetro	Forma corta	Forma larga	Tipo	Valor por defecto	Descripción
Capas de switches	--s	--capa_switch	str	'2,2'	Numero de switches por capa separadas por comas
Número de hosts	--h	--n_host	int	8	Número de hosts
ECMP	--e	--ECMP	bool	False	Activar algoritmo ECMP
Visualización	--g	--graficar	bool	False	Visualización de la topología
Hosts adyacentes	--a	--hosts_adyacentes	bool	False	Desactivar hosts adyacentes del switch que se encuentra h1
Tiempo de prueba	--t	--tiempo_prueba	int	5	Tiempo de prueba en segundos
Carga de tráfico máxima	--c	--carga_trafico_max	int	1000	Carga de tráfico máxima en Mbps
Número de pasos	--p	--pasos	int	4	Número de iteraciones del bucle (precisión de la gráfica)
Numero de flujos	--P	--flujos_paralelos	int	5	Numero de flujo en paralelo
Ayuda	--help	--help	bool	-	Muestra el mensaje de ayuda

Tabla 8: Argumentos del script "ancho_banda_flujo_multiple.py"

Breve explicación de los nuevos argumentos disponibles:

- El argumento "--flujos_paralelos": Es el número de flujos paralelos que enviamos hacia el servidor.

En la Figura 49 observamos el diagrama del flujo, en el podemos entender el funcionamiento interno del script.

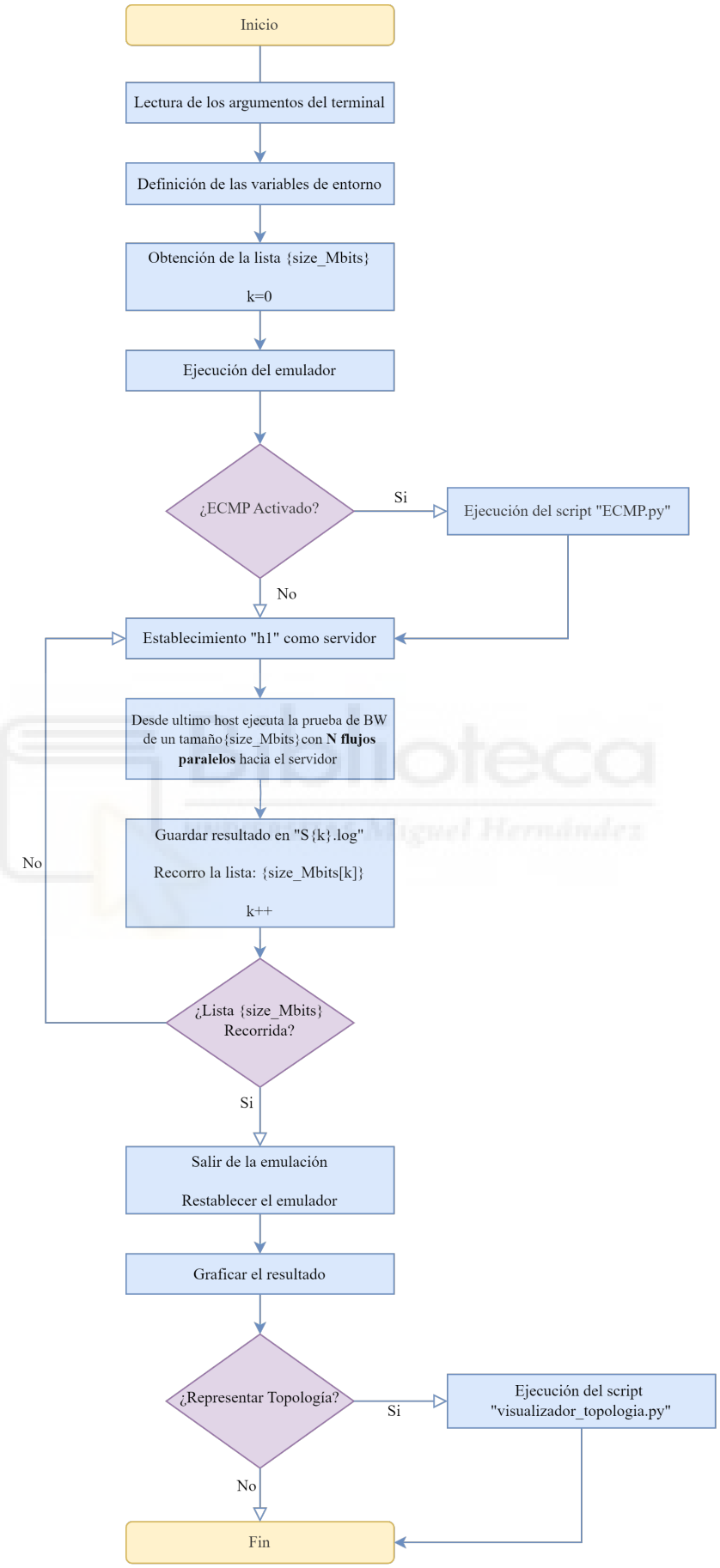


Figura 49: Diagrama de flujo del script "ancho_banda_flujo_multiple.py"

4.7.1 Varias Configuraciones

Este script se llamará “ancho_banda_flujo_multiple_N_configuraciones.py”, lo que hace es emular 4 distintas configuraciones de Spine-Leaf y lo representa en una única gráfica, lo mismo que en el apartado 4.6.1, que se trataba de flujo único en varias configuraciones, pero esta vez con flujo múltiple. El código se encuentra en el apartado 7.10.

4.7.2 Ejecución con argumentos

En este apartado se explica la utilización del script “ancho_banda_flujo_multiple.py”.

1. Accedemos al directorio de los scripts y ejecutamos la ayuda del script:

```
cd Simulaciones/
sudo python3 ancho_banda_flujo_multiple.py --help
```

Una vez ejecutado el comando, se debe visualizar la Figura 50.

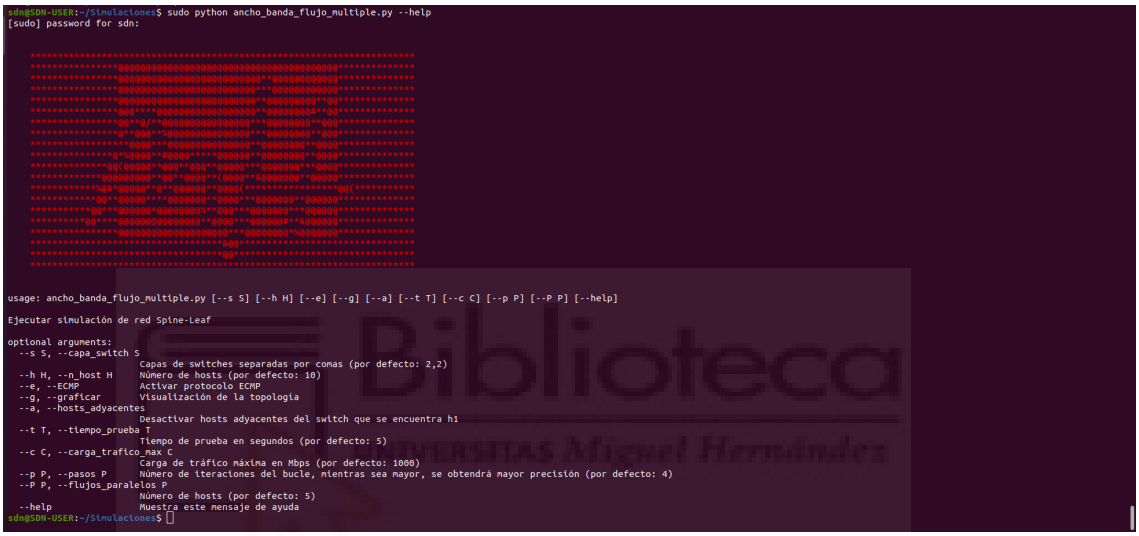


Figura 50: Ejecución del script “ancho_banda_flujo_multiple.py”

2. Ejecutamos el script con los parámetros mínimos, como no tenemos controlador SDN, utilizamos ECMP:

```
sudo python3 ancho_banda_flujo_multiple.py --ECMP
```

Si todo esta correctamente instalado y configurado, deberíamos obtener el resultado de la Figura 51.

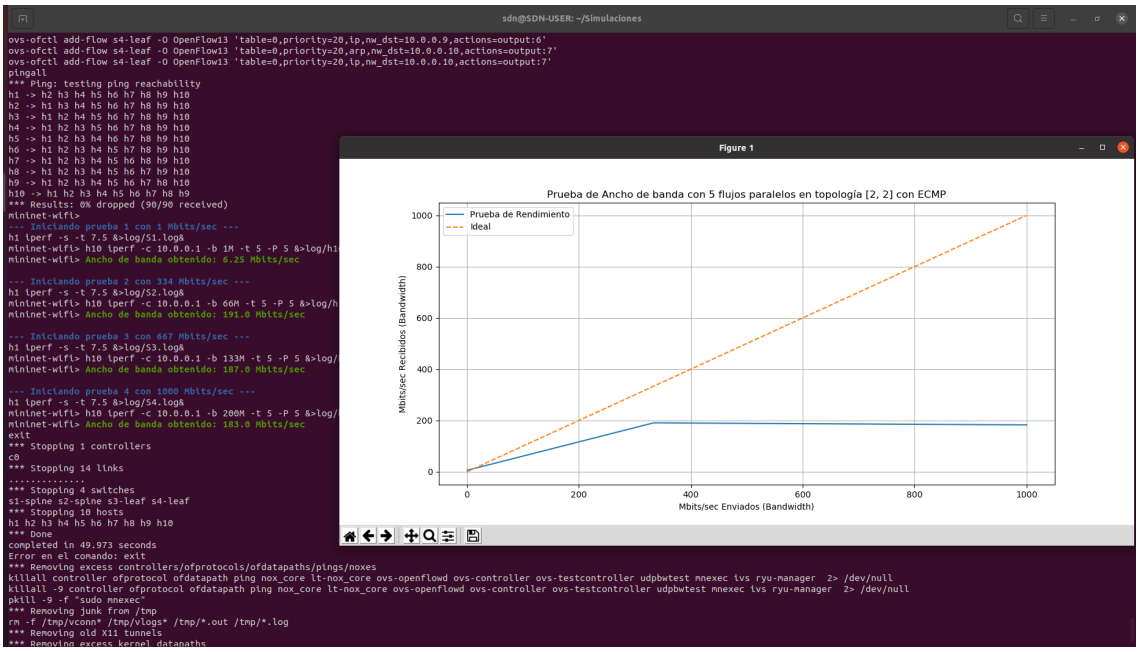


Figura 51: Resultado del script "ancho_banda_flujo_multiple.py"

3. En este ejemplo ejecutamos el script con casi todos los parámetros disponibles:

```
sudo python ancho_banda_flujo_multiple.py --s 4,4 --h 8 --ECMP --g --t 2 --c 800 --p 5 --P 10
```

El resultado se encuentra en la Figura 52, podemos observar en el título, que informa sobre el número de flujos paralelos.

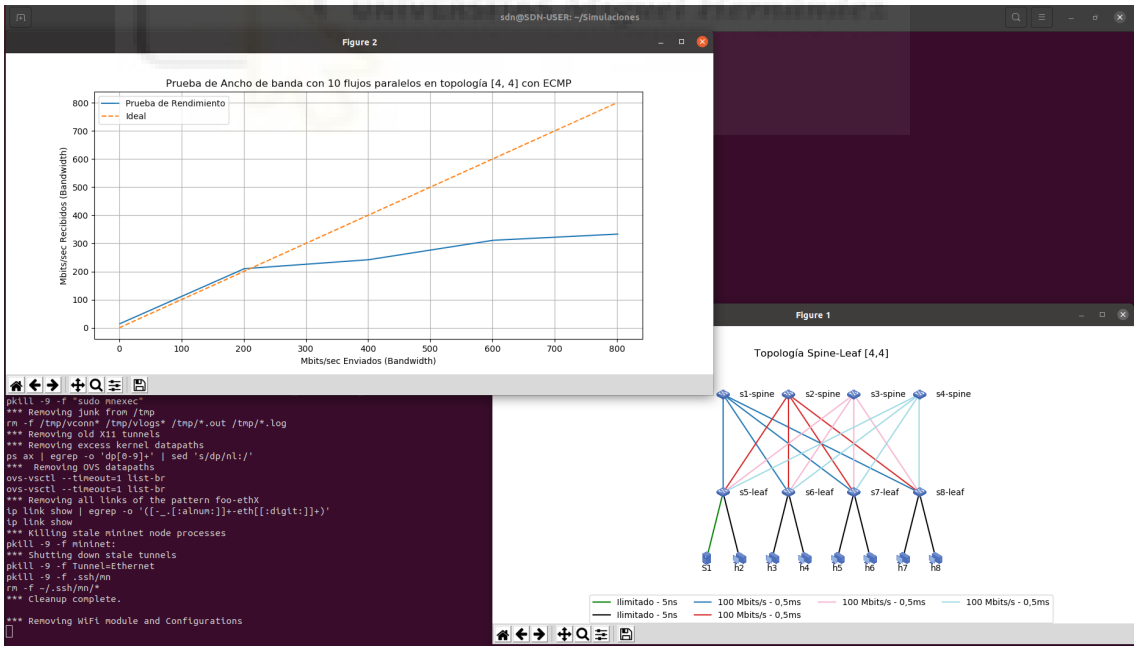


Figura 52: Resultado del script "ancho_banda_flujo_multiple.py" con argumentos

Ignoramos el parámetro "--a" porque en esta prueba no hace ningún efecto, ya que generamos el tráfico solo desde el ultimo host hasta el servidor, aunque los hosts adyacentes estén bloqueados(h2), no surgiría ningún cambio en el resultado.

4.7.3 Ejecución de N configuraciones con argumentos

Ahora se explicará el uso del script "ancho_banda_flujo_multiple_N_configuraciones.py". Para ejecutarlo, con los parámetros mínimos sin controlador:

```
sudo python3 ancho_banda_flujo_multiple_N_configuraciones.py --ECMP
```

El resultado del script lo podemos visualizar en la Figura 53.

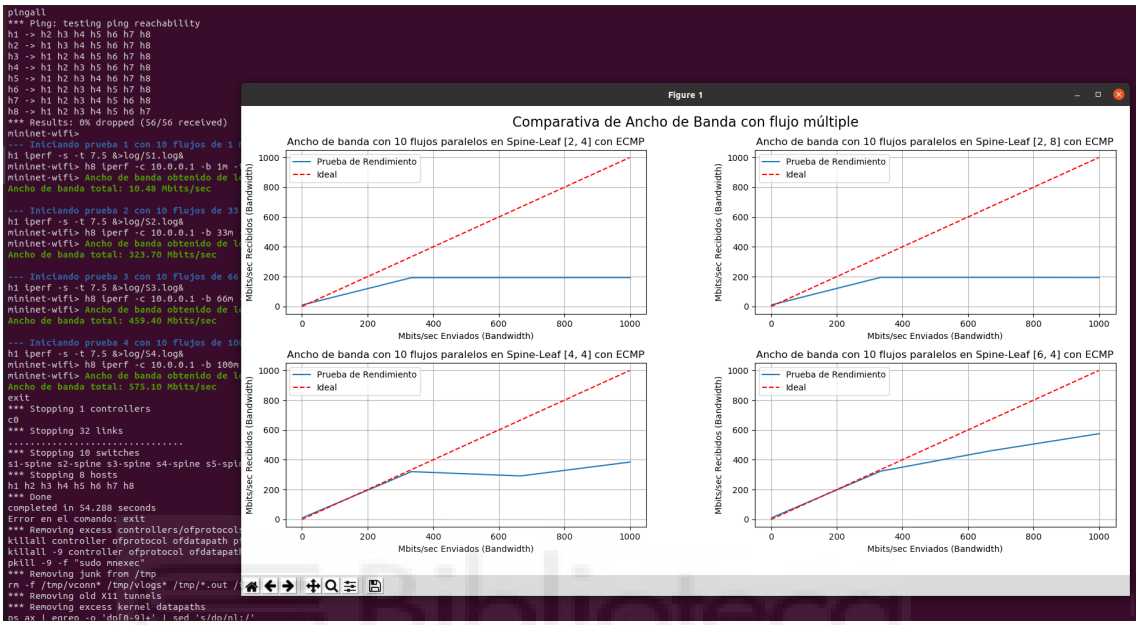


Figura 53: Resultado del script "ancho_banda_flujo_multiple_N_configuraciones.py"

Ahora se mostrará un ejemplo de ejecución más personalizada:

```
sudo python3 ancho_banda_flujo_multiple_N_configuraciones.py --h 8 --s 4,4 4,8 8,4 10,4 --ECMP
--c 800 --p 6 --t 2 --g
```

El resultado lo podemos observar en la Figura 54.

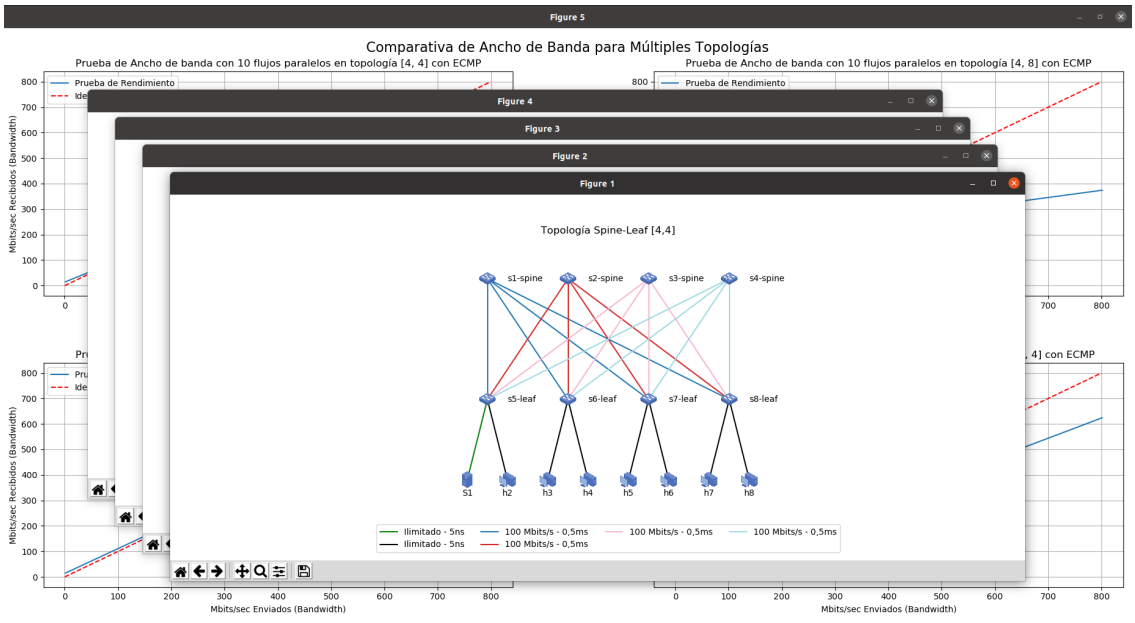


Figura 54: Resultado del script "ancho_banda_flujo_multiple_N_configuraciones.py" con argumentos

4.8 FLUJO DISTRIBUIDO

Este script es parecido que el anterior, pero la diferencia fundamental, es que el servidor Iperf recibe pruebas de ancho de bandas de todos los dispositivos que forma parte de la topología, al mismo tiempo. De esta manera se distribuye el tráfico de forma uniforme por toda la topología. El script se llamará "ancho_banda_flujo_distribuido.py" y el código está en el apartado 7.11.

El código es parecido al flujo único, pero ahora todos los hosts lanzan una prueba de ancho de banda hacia el servidor.

En el tema de argumentos, disponemos de los mismos que en el flujo único, Tabla 9.

Parámetro	Forma corta	Forma larga	Tipo	Valor por defecto	Descripción
Capas de switches	--s	--capa_switch	str	'2,2'	Numero de switches por capa separadas por comas
Número de hosts	--h	--n_host	int	8	Número de hosts
ECMP	--e	--ECMP	bool	False	Activar algoritmo ECMP
Visualización	--g	--graficar	bool	False	Visualización de la topología
Hosts adyacentes	--a	--hosts_adyacentes	bool	False	Desactivar hosts adyacentes del switch que se encuentra h1
Tiempo de prueba	--t	--tiempo_prueba	int	5	Tiempo de prueba en segundos
Carga de tráfico máxima	--c	--carga_trafico_max	int	1000	Carga de tráfico máxima en Mbps
Número de pasos	--p	--pasos	int	4	Número de iteraciones del bucle (precisión de la gráfica)
Ayuda	--help	--help	bool	-	Muestra el mensaje de ayuda

Tabla 9: Argumentos del script "ancho_banda_flujo_distribuido.py"

Cabe destacar el siguiente argumento:

- El argumento "--hosts_adyacentes": Desactiva aquellos host adyacentes al servidor(h1), la principal utilización es al medir el ancho de banda en flujo distribuido, porque dichos hosts tienen el camino directo al servidor sin pasar por los switches Spine, con lo que interfieren en el resultado, proporcionando un valor más alto de lo que la topología soporta en el ámbito del ancho de banda.

La Figura 55 muestra el diagrama de flujo del script.

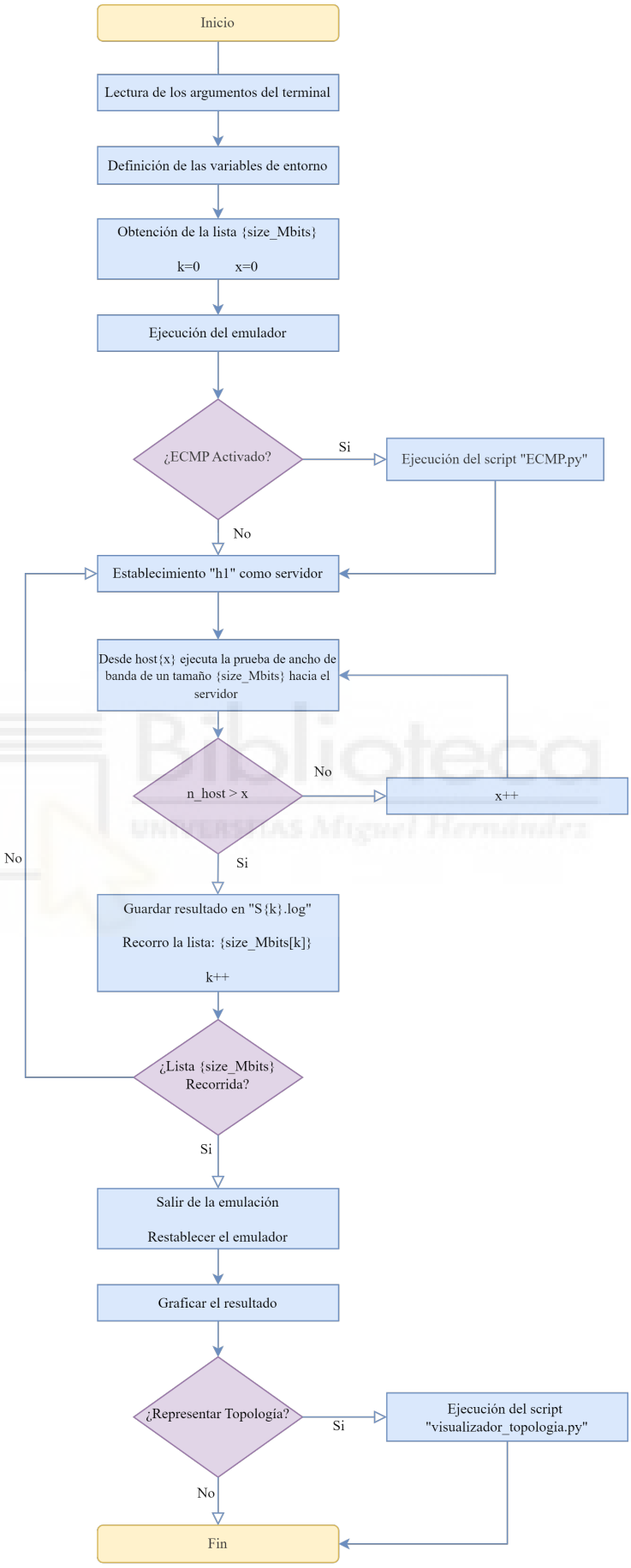


Figura 55: Diagrama de flujo del script "ancho_banda_flujo_distribuido.py"

4.8.1 Varias Configuraciones

Lo mismo que en los apartados de varias configuraciones anteriores, pero esta vez serán emulaciones de flujos distribuidos en 4 distintas configuraciones de Spine-Leaf. Dicho script se llamará "ancho_banda_flujo_distribuido_N_configuraciones.py" y el código está disponible en el apartado 7.12.

4.8.2 Ejecución con Argumentos

Comenzamos con el script "ancho_banda_flujo_distribuido.py".

Para ejecutarlo, con los parámetros mínimos sin controlador:

```
sudo python3 ancho_banda_flujo_distribuido.py --ECMP --a
```

En los parámetros mínimos, lo novedoso es el parámetro "--a", que bloquea los dispositivos adyacentes al servidor, esto es para que no interfieran en la medida del ancho de banda de la topología. Si no bloqueamos dichos dispositivos, dispondrán de un camino directo sin tener que utilizar la topología Spine-Leaf, esto interferiría en la medida ya que nos daría valores muchos más altos de que realmente la topología soporta.

El resultado del script lo podemos visualizar en la Figura 56.

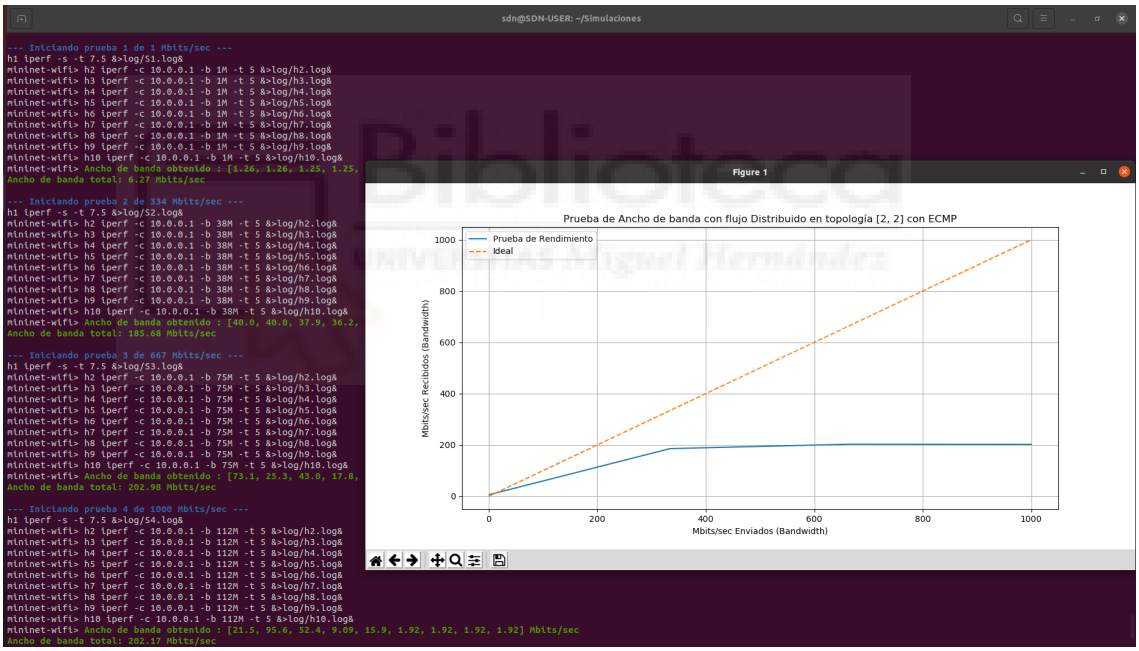


Figura 56: Resultado del script "ancho_banda_flujo_distribuido.py"

Ejemplo de una ejecución más personalizada:

```
sudo python3 ancho_banda_flujo_distribuido.py --ECMP --a --h 16 --s 4,4 --c 800 --p 6 --t 4 --
g
```

El resultado de la ejecución del script lo podemos consultar en la Figura 57.

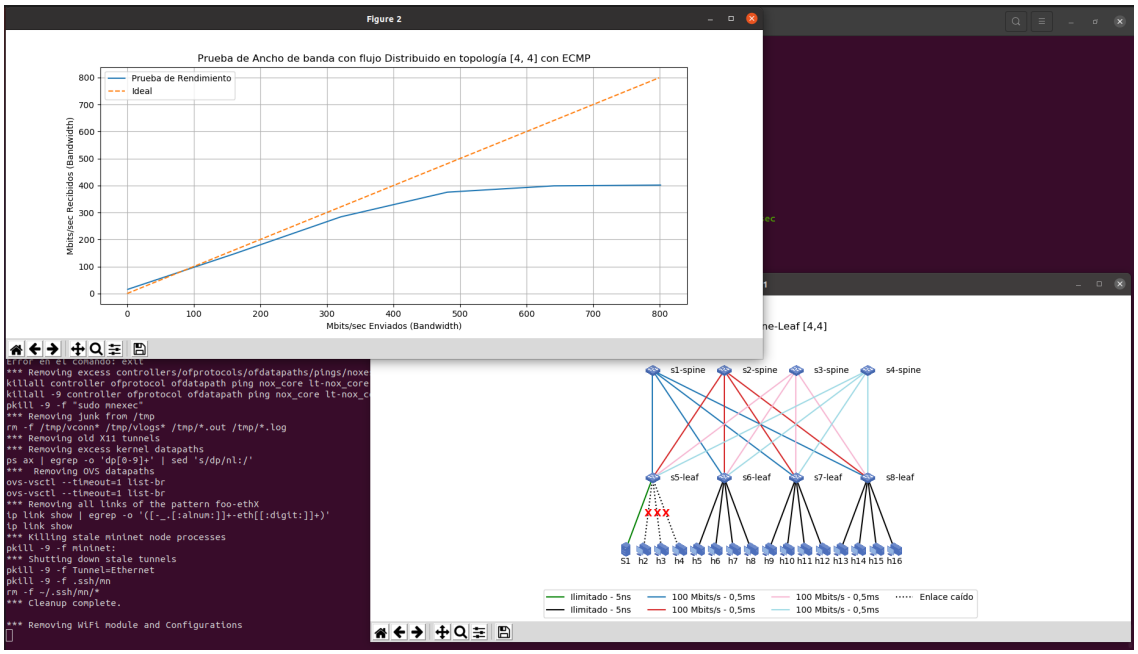


Figura 57: Resultado del script "ancho_banda_flujo_distribuido.py" con argumentos

4.8.3 Ejecución de N configuración con argumentos

Ahora mostraremos la ejecución "ancho_banda_flujo_distribuido_N_configuraciones.py".

Para ejecutarlo sin controlador SDN, es necesario utilizar dos parámetros:

```
sudo python3 ancho_banda_flujo_distribuido_N_configuraciones.py --ECMP --a
```

En la Figura 58, visualizamos el resultado de la ejecucion.

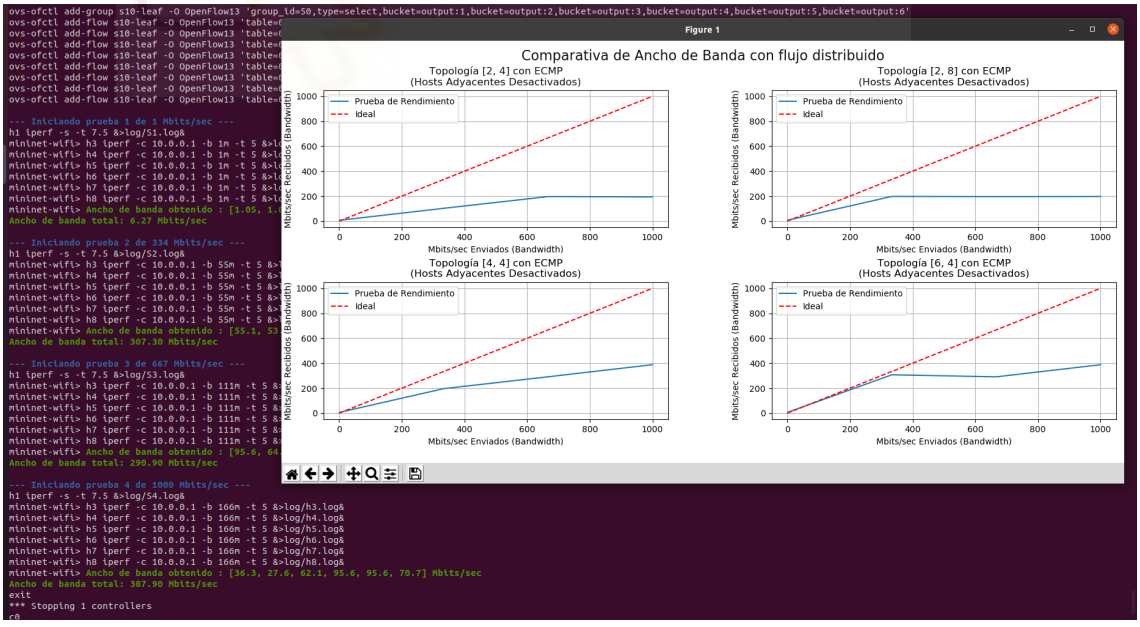


Figura 58: Resultado del script "ancho_banda_flujo_distribuido_N_configuraciones.py"

Ahora procedemos a una ejecución más personalizada:

```
sudo python3 ancho_banda_flujo_distribuido_N_configuraciones.py --ECMP --a --h 16 --s 4,4 4,8 8,4 10,4 --c 800 --p 6 --t 4 --g
```

En la Figura 59 observamos el resultado de la ejecución del script.

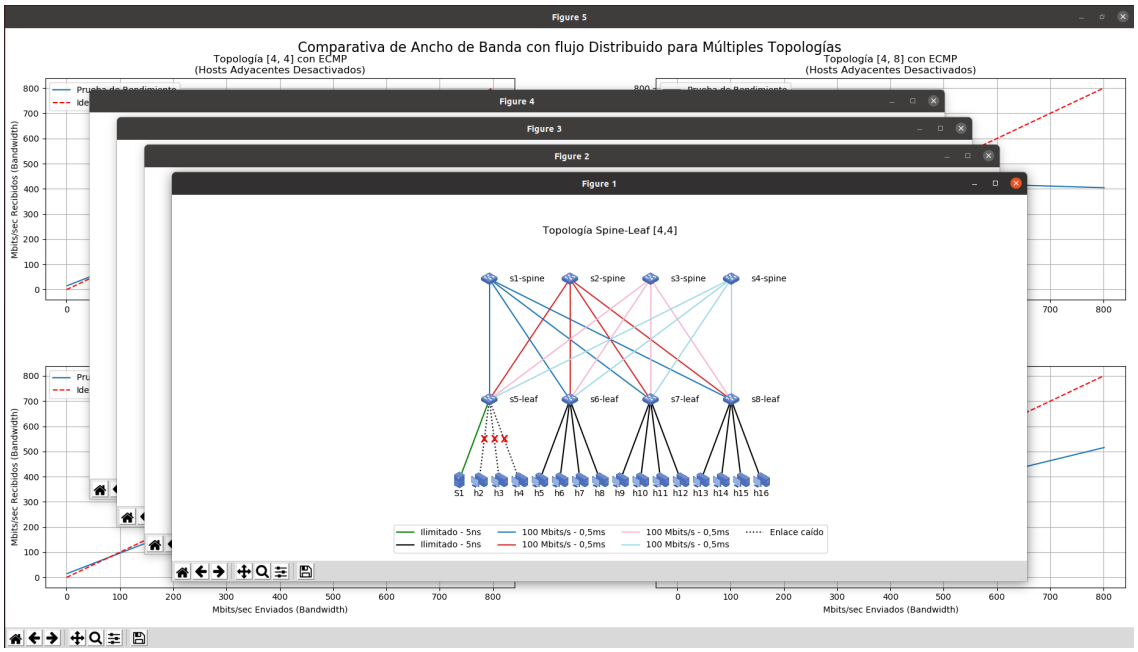


Figura 59: Resultado del script "ancho_banda_flujo_distribuido_N_configuraciones.py" con argumentos

4.9 LATENCIA BAJO CARGA

Combinaremos las pruebas de latencia y ancho de banda con flujo distribuido en una única prueba, con el motivo de poder observar cómo se comporta la latencia ante la presencia de cargas de trabajos. También tiene la capacidad de introducir flujos paralelos en cada host.

En el tema de código, es la mezcla de ambos scripts, pero con modificaciones para su implantación en un único script.

El script es llamado "latencia_bajo_carga_flujo_distribuido.py" y su código esta disponible en el apartado 7.13.

Para configurar los parámetros de la topología y de la emulación, lo hacemos mediante los argumentos, que son los siguientes:

Parámetro	Forma corta	Forma larga	Tipo	Valor por defecto	Descripción
Capas de switches	--s	--capa_switch	str	'2,2'	Numero de switches por capa separadas por comas
Número de hosts	--h	--n_host	int	8	Número de hosts
ECMP	--e	--ECMP	bool	False	Activar algoritmo ECMP
Visualización	--g	--graficar	bool	False	Visualización de la topología
Hosts adyacentes	--a	--hosts_adyacentes	bool	False	Desactivar hosts adyacentes del switch que se encuentra h1
Tiempo de prueba	--t	--tiempo_prueba	int	5	Tiempo de prueba en segundos
Carga de tráfico máxima	--c	--carga_trafico_max	int	1000	Carga de tráfico máxima en Mbps
Número de pasos	--p	--pasos	int	4	Número de iteraciones del bucle (precisión de la gráfica)
Numero de flujos	--P	--flujos_paralelos	int	5	Numero de flujo en paralelo
Numero de ping	--i	--ping	int	10	Numero de iteraciones del ping
Ayuda	--help	--help	bool	-	Muestra el mensaje de ayuda

Tabla 10: Argumentos del script "latencia_bajo_carga.py"

Breve explicación de los nuevos argumentos disponibles:

- El argumento "--ping": Es la cantidad de paquetes del comando ping que se enviará hacia el servidor, mientas mayor sea, más tardará la prueba y más preciso será.

En la Figura 60, podemos consultar el diagrama del flujo.

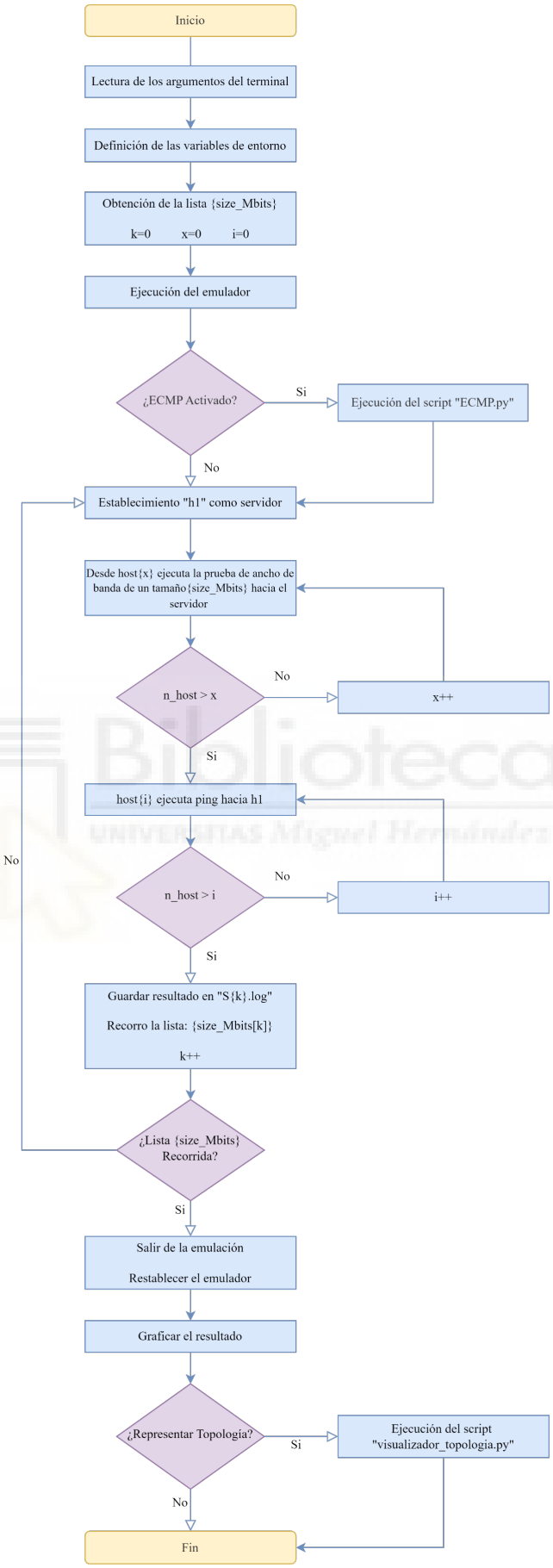


Figura 60: Diagrama de flujo del script "latencia_bajo_carga_flujo_distribuido.py"

4.9.1 Ejecución con argumentos

Para ejecutarlo sin controlador SDN, solo es necesario ECMP:

```
sudo python3 latencia_bajo_carga_flujo_distribuido.py --ECMP
```

En la Figura 61, observamos el resultado de la ejecución, esta vez la gráfica es distinta, ya que está en escala logarítmica.

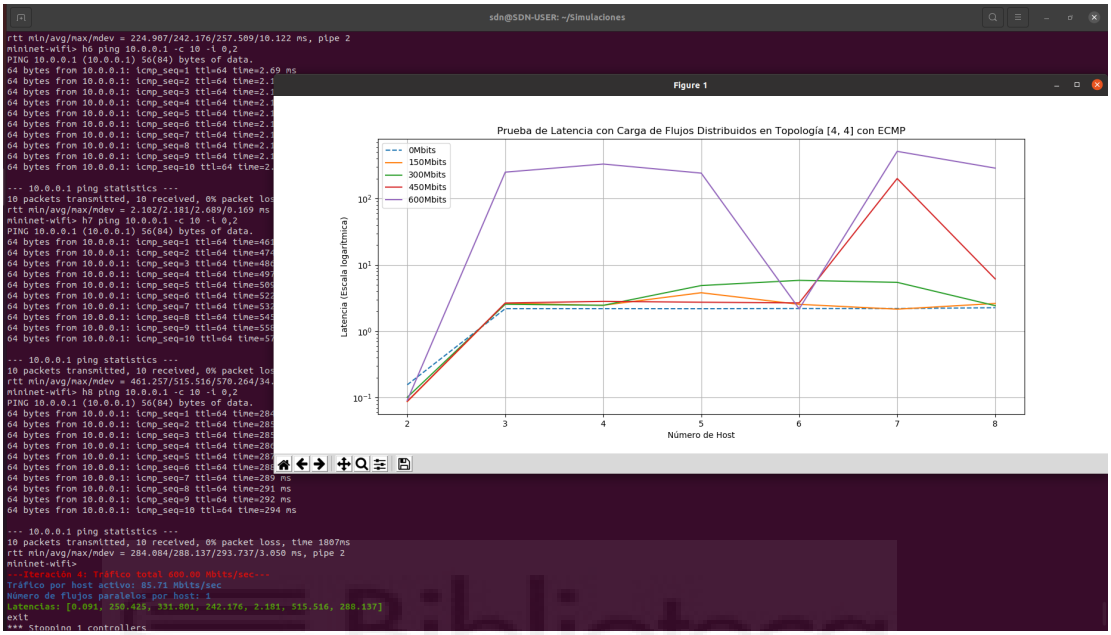


Figura 61: Resultado del script "latencia_bajo_carga_flujo_distribuido.py"

Ejemplo de una ejecución más personalizada:

```
sudo python3 latencia_bajo_carga_flujo_distribuido.py --ECMP --s 6,4 --h 16 --g --c 700 --p 5
--i 5 --P 4
```

El resultado de la ejecución se encuentra en la Figura 62.

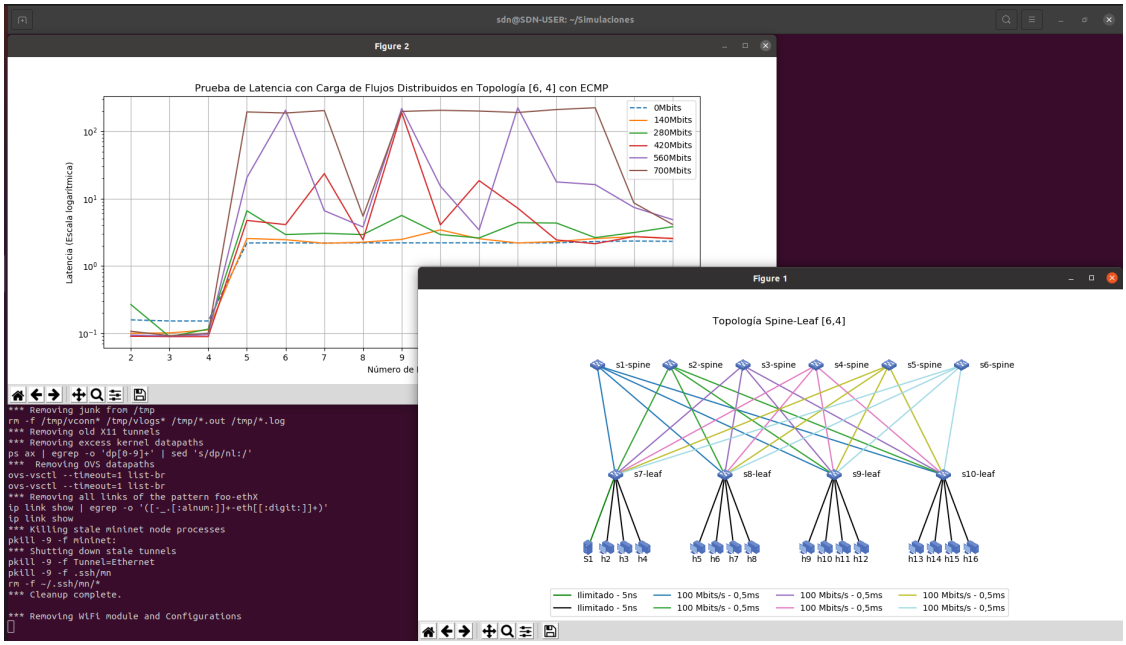


Figura 62: Resultado del "latencia_bajo_carga_flujo_distribuido.py" con argumento

5 MARCO PRACTICO

En este apartado se explicará los resultados obtenidos, los problemas y sus soluciones.

El entorno de trabajo para todas las pruebas con el algoritmo STP (Protocolo de Árbol de expansión) en SDN es el siguiente:

- Sistema operativo: Ubuntu 20.04 LTS
- Lenguaje de programación: Python
- Emulador: Mininet-WiFi
- Controlador: OpenDaylight

Y en el caso del algoritmo ECMP, es entorno es:

- Sistema operativo: Ubuntu 20.04 LTS
- Lenguaje de programación: Python
- Emulador: Mininet-WiFi
- Controlador: ninguno, no es necesario porque las rutas son definidas por el script "ECMP.py".

Cabe destacar, que todo el entorno esta explicado en el apartado 3, 'Herramientas'.

5.1 LATENCIA

En este apartado estudiaremos el comportamiento de la latencia en las distintas configuraciones de la topología Spine-Leaf. Antes de ellos, contemplaremos los problemas que surgieron y sus soluciones.

5.1.1 Problema con VirtualBox

En las primeras emulaciones, me tope con resultados inestables de los pings. Podemos verlo en la Figura 63.

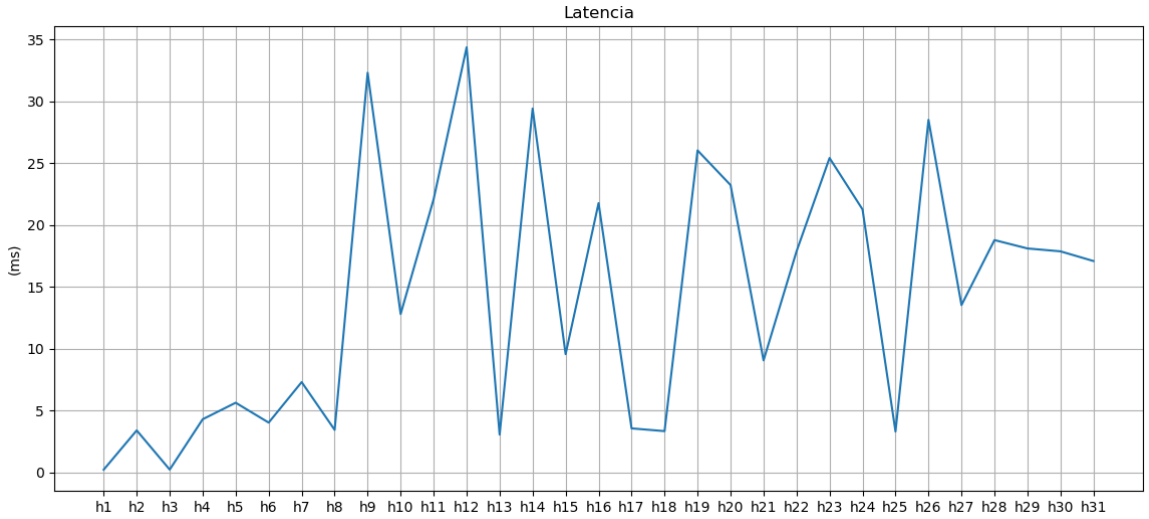


Figura 63: Grafica de la emulación de la latencia en VirtualBox

Después de probar varias soluciones, encontré en el foro de GitHub [55] la solución al problema. El problema estaba en el VirtualBox, una vez que exporté la máquina virtual a VMware, el ping se estabilizó en todos los hosts, Figura 64.

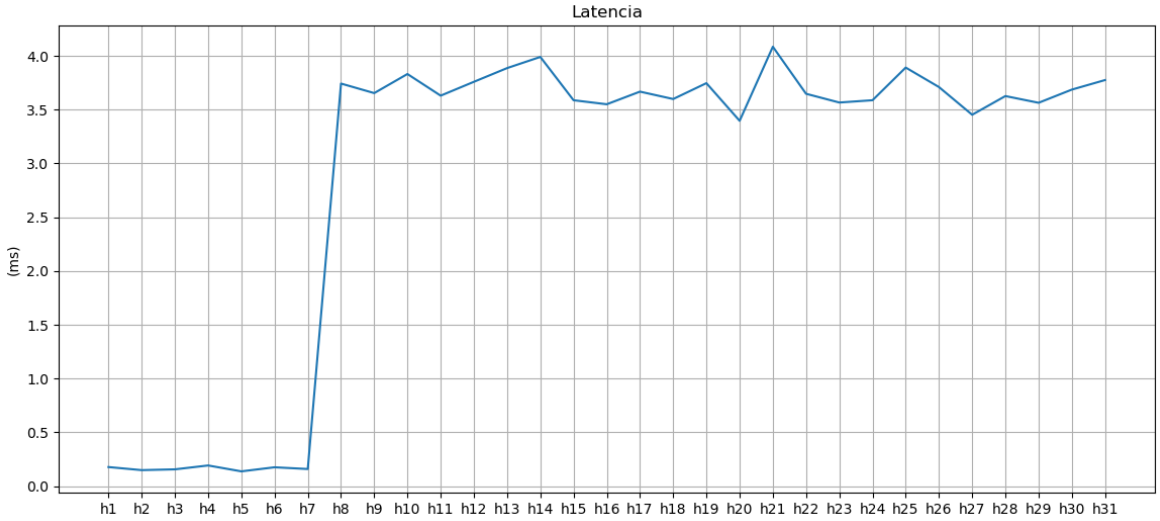


Figura 64: Grafica de la emulación de la latencia en VMware.

Como podemos observar la diferencia de los resultados de la Figura 63 e Figura 64, el cambio es drástico en los resultados, como solución final, trabajaremos en una partición con el Ubuntu 20.04 LTS por el tema de los recursos computacionales.

5.1.2 Latencia del emulador

Si emulamos en Mininet-WiFi una topología sin introducir latencia en los enlaces, el resultado podemos observarlo en la Figura 65.

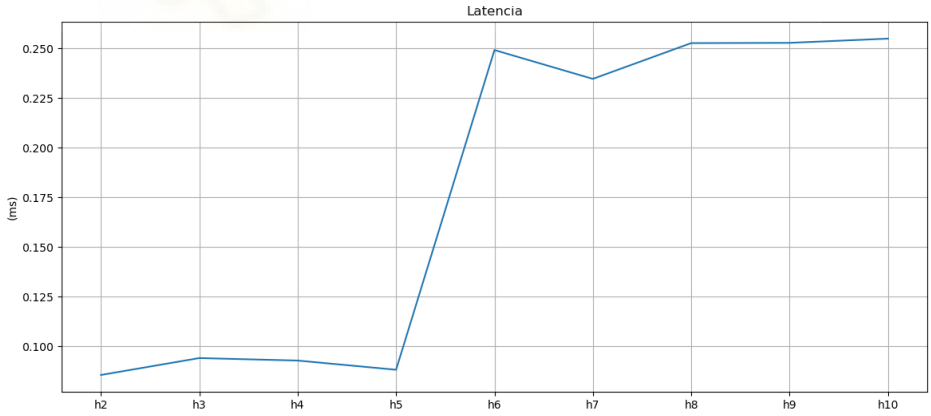


Figura 65: Latencia sin introducir retardo en los enlaces

Esta latencia esta vista del host 1, es decir, desde todos los hosts que forma parte de la red harán los pings al servidor(h1). Desde este punto, al host 1 lo tratamos como el servidor de la red.

Los hosts que están en el mismo switch (h2, h3, h4, h5) que el servidor (h1), tienen una latencia aproximada de 0.9 ms, porque al estar conectado en el mismo switch tienen un camino directo, en los demás switches, como tienen que recorrer la topología para llegar al

destino, están separados por 4 enlaces[5], por ello tienen una latencia de 0.250 ms. Para entenderlo más claramente, podemos visualizar la topología en cuestión, en la Figura 77.

Cabe resaltar que dicha latencia esta data en el entorno de trabajo descrito anteriormente, no la podemos disminuir. Al calcular la latencia, se tendrá en cuenta la latencia del entorno.

5.1.3 Emulación con algoritmo STP

Comenzaremos a testear con el algoritmo del controlador OpenDaylight, posteriormente en el apartado, averiguaremos que se trata del algoritmo del protocolo de Árbol de expansión (STP). Se ha tenido que averiguar de forma práctica porque en la documentación oficial de los controladores SDN, no mencionaban el algoritmo utilizado para evitar los bucles.

Comenzaremos con la topología Spine-Leaf [2,2][6] ,en cada capa tendrá 2 switches, y con un total de 20 host. La topología en concreto se muestra en la Figura 66.

La ejecución de los Scripts está explicada en el apartado 4, 'Scripts implementados'.

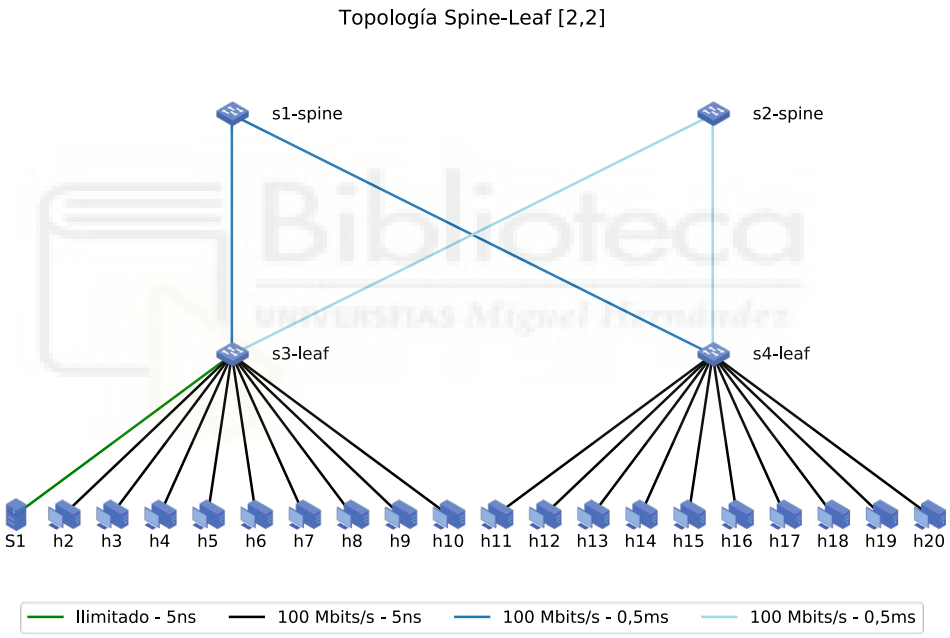


Figura 66: Topología Spine-Leaf [2,2] con 20 host

En la Figura 67 observamos el resultado de la prueba, podemos visualizar que los hosts que están en el mismo switch que el s1, tienen una latencia cercana 0.100 ms, en cambio los hosts del switch s4, tienen una latencia cercana a 2,5 ms, eso es debido que la latencia mide el tiempo de ida y de vuelta, de la siguiente manera (latencia de los hosts del s4-leaf):

$$Latencia = T_{Ida} + T_{Vuelta} \qquad (3)$$

$$T_{Ida} = L_{h20-s4\_leaf} + L_{s4\_leaf-s1\_spine} + L_{s1\_spine-s3\_leaf} + L_{s3\_leaf-s1} + L_E \qquad (4)$$

[5] Una característica de la topología Spine-Leaf, es que cualquiera host se comunica con otro host, con 4 enlaces, si se encuentran en distintos switches.
[6] La nomenclatura [2,4] significa que en la capa Spine hay definidos 2 switches y 4 en la capa Leaf.

Siendo L_{h20-s4_leaf} :la latencia del enlace que conecta el host 20 con el switch s4-leaf

Siendo L_E :la latencia del entorno

$$T_{Ida} = 5\mu s + 0.5ms + 0.5ms + 5\mu s + 0.250ms \quad (5)$$

$$T_{Ida} = 1.26ms \quad (6)$$

Como la ida y vuelta es el mismo camino, entonces:

$$Latencia = 2 \cdot T_{Ida} = 2.52\ ms \quad (7)$$

Si comparamos los resultados de la emulación, que lo podemos consultar en Figura 67 y teóricos calculados anteriormente, coinciden en gran medida. Esto indica que la emulación está operando correctamente.

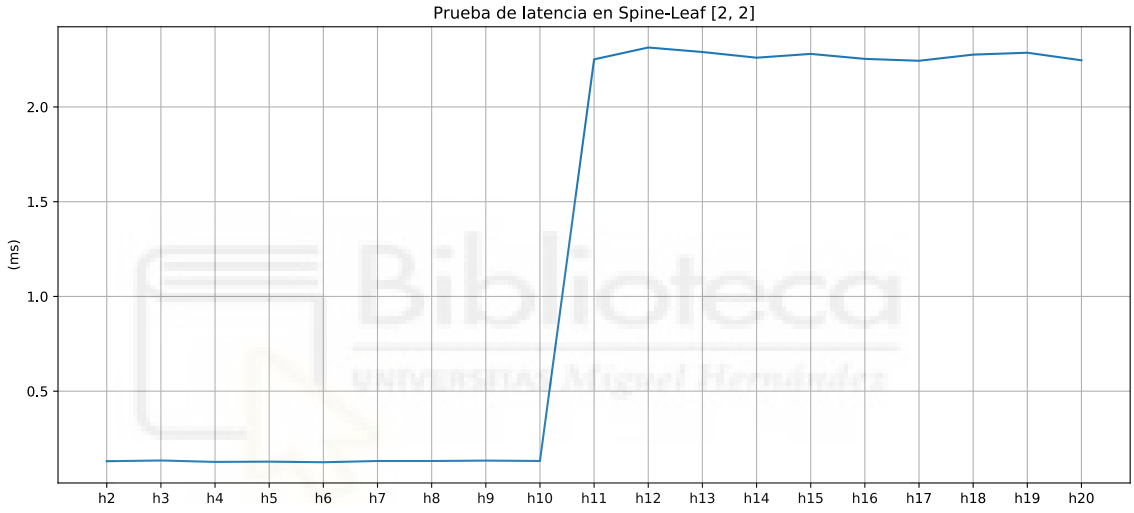


Figura 67: Latencia en Spine-Leaf [2,2]

Vamos a probar con una configuración más grande en la capa de Leaf, agregando 2 switches más en la capa Leaf, y así obtenemos una configuración Spine-Leaf [2,4] que es la que se muestra en la Figura 68.

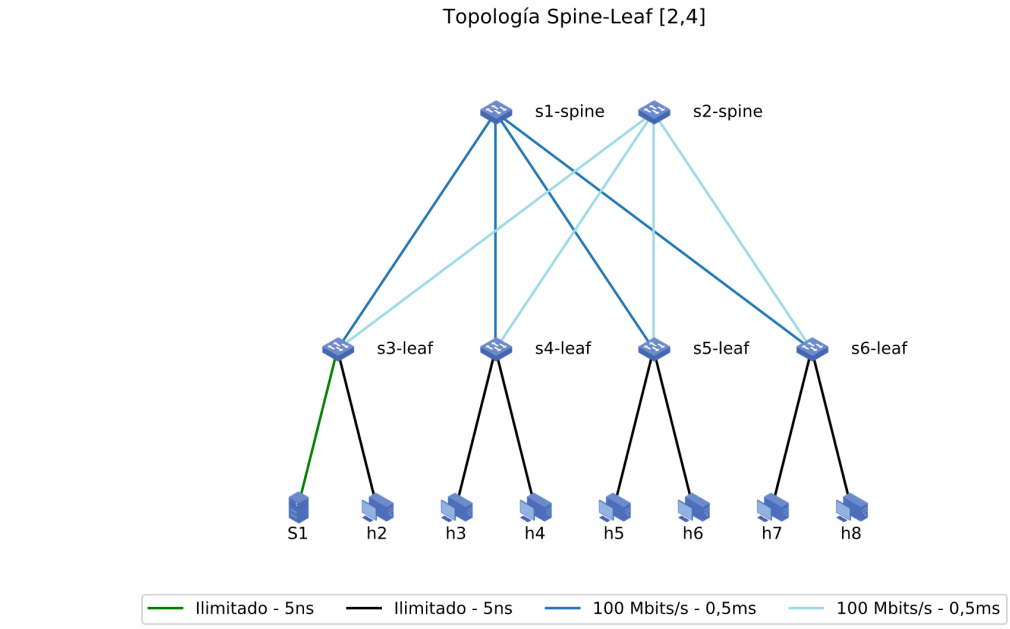


Figura 68: Topología Spine-Leaf [2,4] con 8 host

En los resultados de la Figura 69 observamos que la latencia ha aumentado en los hosts que están conectados al switch 5, eso significa que los paquetes tienen que atravesar más enlaces hasta llegar al destino y volver.

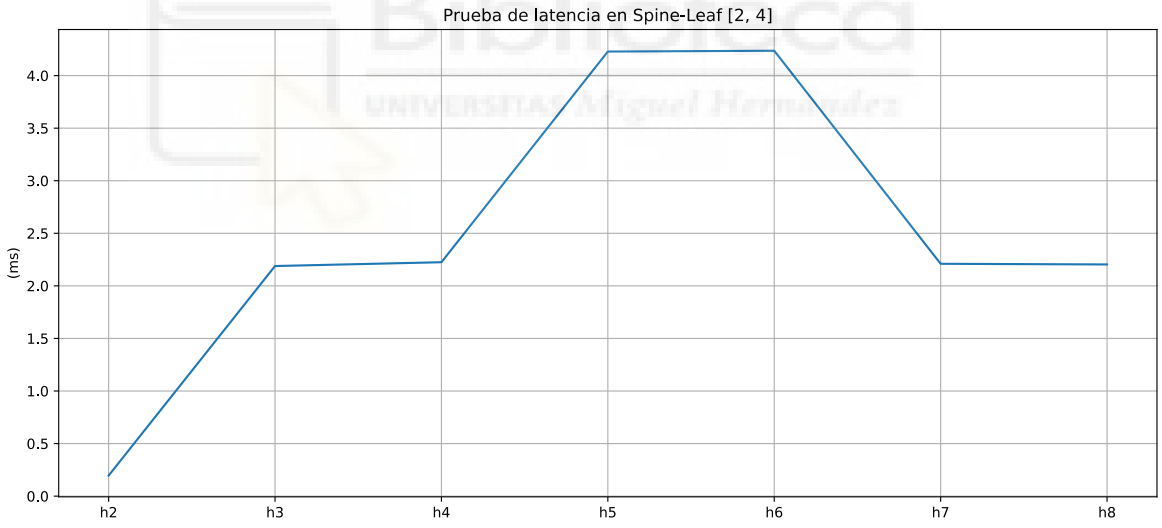


Figura 69: Latencia en Spine-Leaf [2,4]

Para entender el resultado, simularemos que utilice el algoritmo de STP, siendo el switch s1 la raíz, Figura 70.

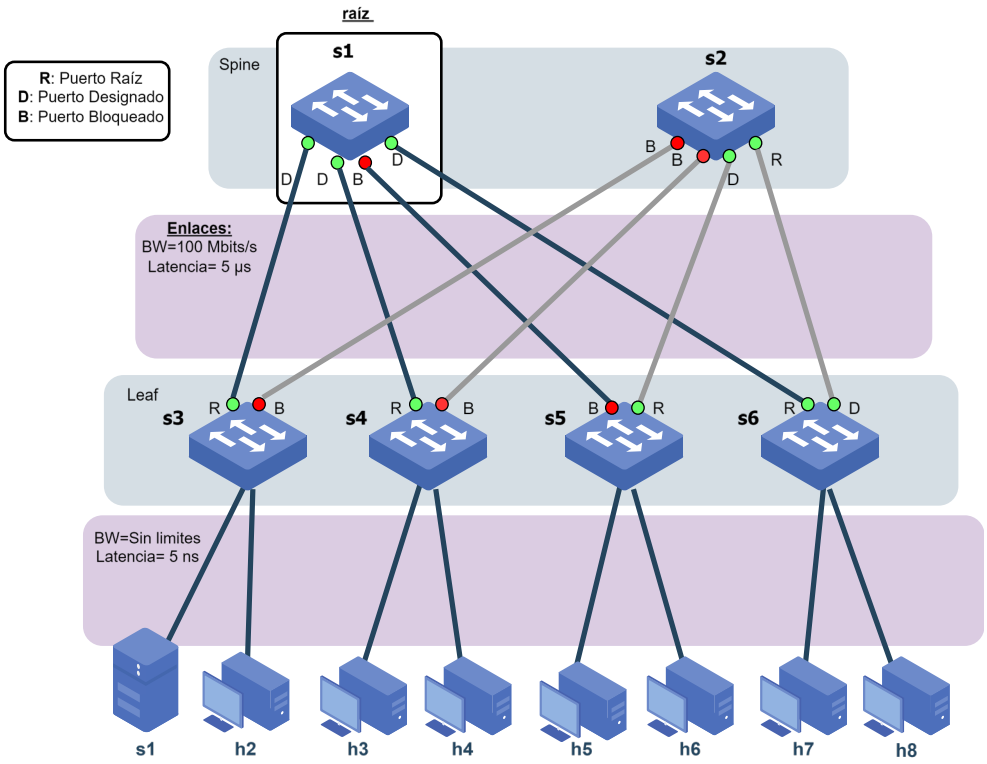


Figura 70: STP de Spine-Leaf [2,4]

Podemos observar que concuerdan los resultados con la emulación del algoritmo STP, ya que los hosts 5 y 6 tienen que recorrer "s5-Leaf – s2-Spine – s6-Leaf – s1-Spine – s3-Leaf" para llegar al servidor 1, en cambio los demás hosts solo tienen que recorrer 2 switches. De allí viene el incremento de la latencia.

Con los resultados anteriores podemos asumir que el controlador OpenDaylight introduce el algoritmo STP u similares, mediante las aplicaciones de tráfico activadas. En la documentación oficial no data sobre el algoritmo STP, pero asumiremos que sí.

Probaremos la latencia esta vez en una configuración de la topología más grande, en este caso, 10 switches Spine y 10 switches Leaf, con los mismos hosts. Dicha topología lo representamos en la Figura 71.

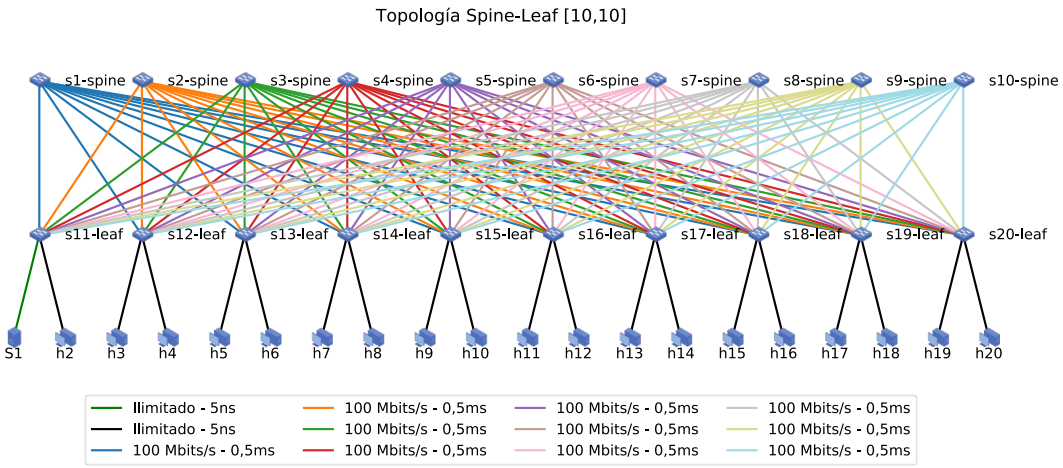


Figura 71: Topología Spine-Leaf [10,10] con 20 host

Si observamos el resultado de la emulación de la Figura 72 , en hosts concretos tienen una latencia mayor que en la configuración [2,4], lo podemos ver en la Figura 69. Esto debido que, al ser la configuración más grande, dependiendo del switch que están conectados los hosts, los paquetes tendrán que recorrer más o menos enlaces.

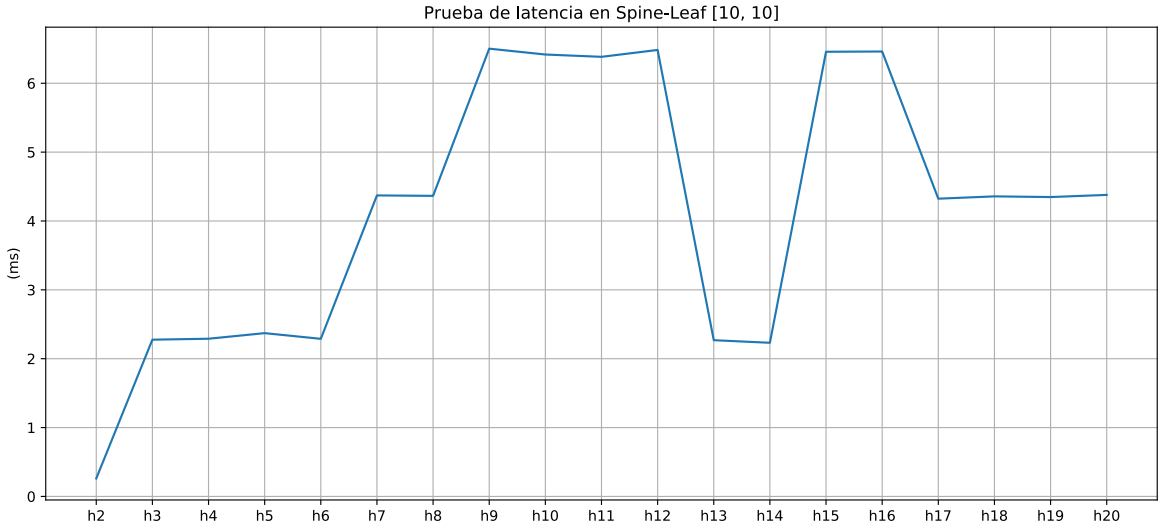


Figura 72: Latencia en Spine-Leaf [10,10]

5.1.4 Emulación con ECMP

Testearemos ahora con el algoritmo de ECMP, con las mismas configuraciones de topologías que en el apartado anterior.

Comenzamos con una configuración de Spine-Leaf [2,2] que lo podemos observar en la Figura 66. El resultado de la latencia de dicha configuración está en la Figura 73.

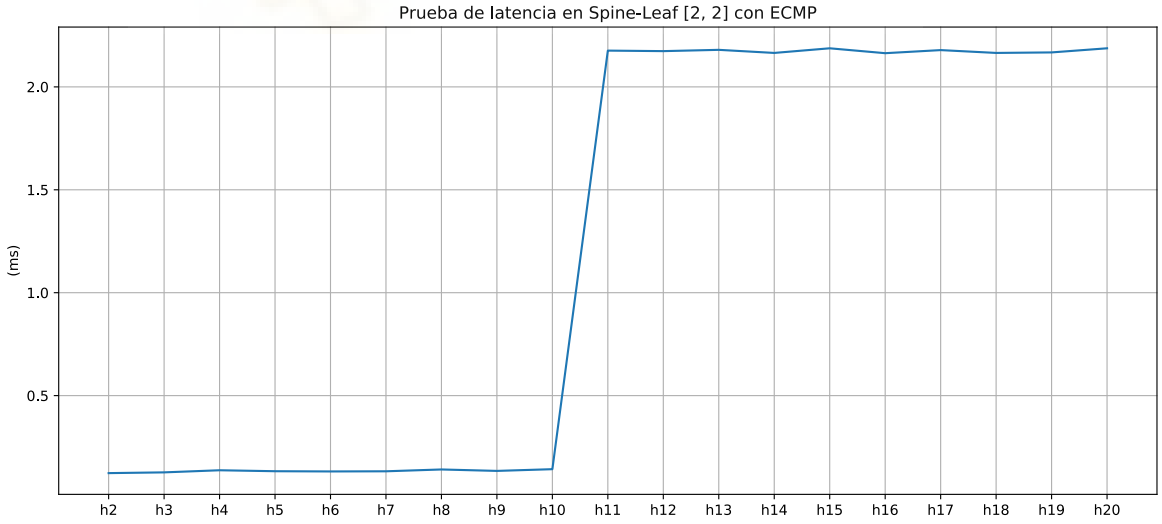


Figura 73: Latencia en Spine-Leaf [2,2] con ECMP

Si comparamos los resultados del algoritmo STP, que se encuentra en la Figura 67 con la Figura 73, que se trata de los resultados de ECMP. Los resultados son prácticamente idénticos, por ello, en una configuración pequeña, el algoritmo ECMP no presenta mejoras en latencia.

Ahora aumentaremos un poco de tamaño la configuración, concretamente, Spine-Leaf [2,4], Figura 68.

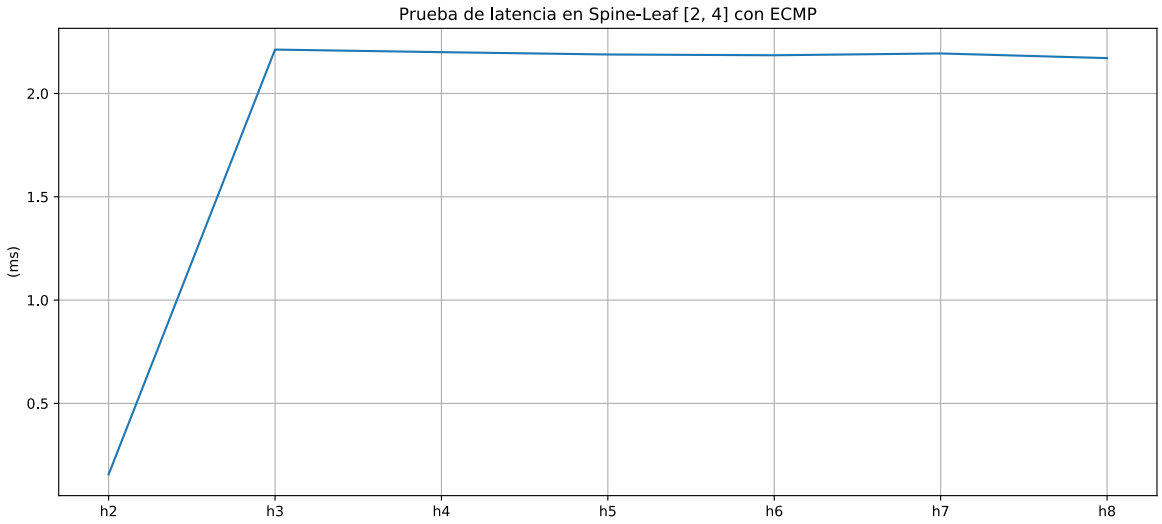


Figura 74: Latencia en Spine-Leaf [2,4] con ECMP

En los resultados de la Figura 74, podemos ver que tiene el mismo resultado que en la configuración anterior, Figura 73, pero si observamos el resultado de la misma configuración con el algoritmo STP, Figura 69, defieren los resultados, ya que se observa un incremento de latencia por parte del algoritmo STP.

Y, por último, emularemos la configuración más grande, Spine-Leaf [10,10], Figura 71.

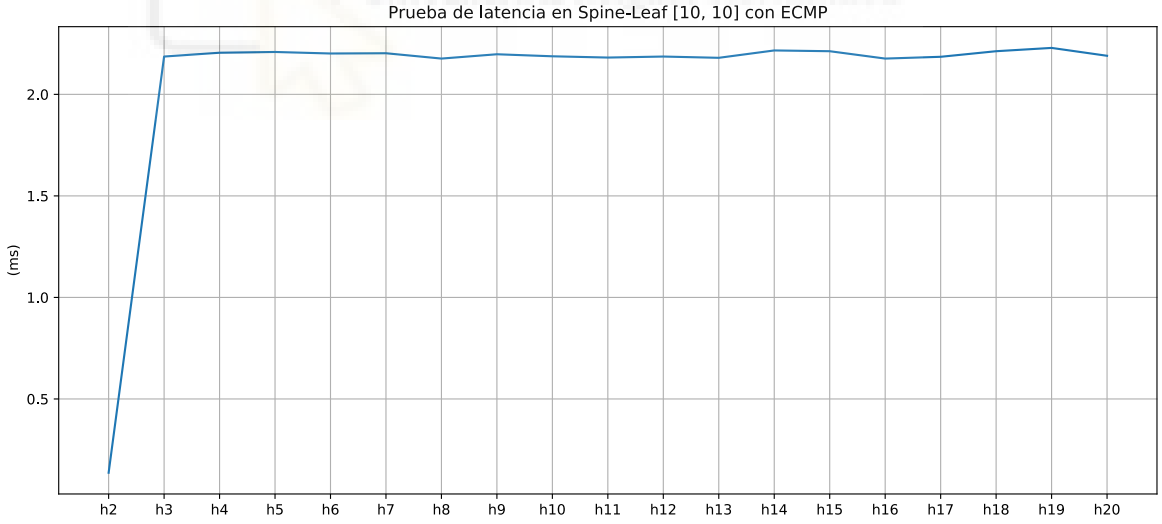


Figura 75: Latencia en Spine-Leaf [10,10] con ECMP

Observando la Figura 75, obtenemos los mismos resultados que en casos anterior con el algoritmo ECMP. Dichos resultados defieren con los de algoritmo STP, ya que en ellas la latencia aumenta a la medida que las topologías crecen.

5.1.5 Conclusión

La conclusión es sencilla, en el algoritmo de STP, al aumentar el tamaño de la topología, se aumenta la latencia, en cambio en ECMP, se mantiene.

El motivo básicamente en el algoritmo STP, es que trata de bloquear los enlaces redundantes para evitar los bucles, pero al bloquear enlaces, puede provocar que el camino hacia el destino incremente, por ello atraviese más enlaces, y de allí proviene el aumento de latencia.

En cambio, el algoritmo ECMP no bloquea los enlaces, en cambio utiliza todas las rutas disponibles de la misma longitud, concretamente en Spine-Leaf, todas las rutas que conectan dos dispositivos cualesquiera están separa a 4 enlaces (hosts en distintos switches). Esta característica permite tener la misma latencia en toda la topología.

La parte negativa es la cantidad de puertos que se utilizan los switches Leaf, ya que deben de estar conectados con todos los switches de capa superior (Spine), entonces si en la capa Spine tenemos 16 switches, en los switches Leaf se utilizaran 16 puertos para interconectar los switches Spine. Esto es un inconveniente para la escalabilidad de la topología.

5.2 Flujo Único

En este apartado mediremos el ancho de banda con un único flujo, quiere decir, que desde el ultimo host de la topología, haremos una prueba de ancho de banda con Iperf hacia el primer host llamado s1, que actuara como servidor.

En la Figura 76 observamos el flujo único generado por h4, teniendo como destino s1. Observamos que recorre el camino "s4-leaf – s2-spine – s3-leaf", pero también existe el camino "s4-leaf – s1-spine – s3-leaf", cualquiera de ellos es válido.

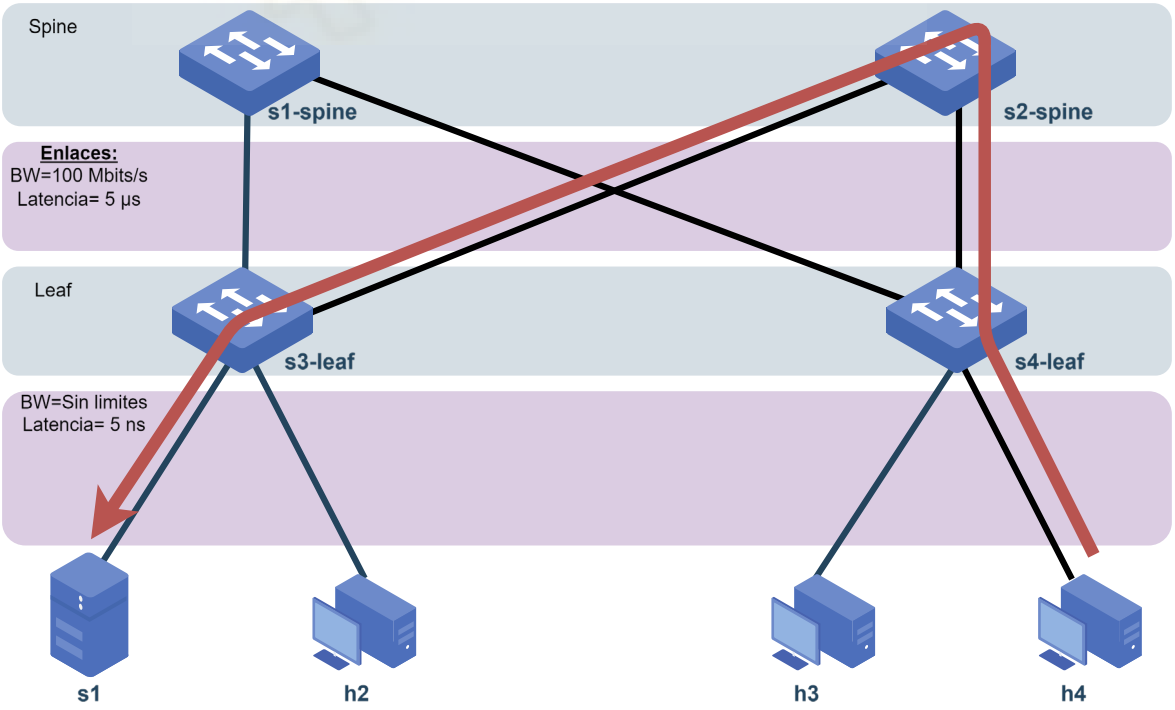


Figura 76: Flujo único

5.2.1 Problema con Mininet-WiFi

Uno de los problemas que encontraremos es con la herramienta Iperf en el emulador Mininet-WiFi, si nos fijamos en la Figura 80, observamos que la gráfica no llega a tocar la meta de los 100 Mbits/s en flujo único, realmente se debería alcanzar el meta, ya que el ancho de banda en los enlaces entre switches es de 100Mbits/s.

En la Figura 45 podemos ver el fenómeno descrito, fijándose en la consola del terminal, específicamente en los resultados de color rojo, en las pruebas de ancho de banda menores a los 100 Mbits/s obtenemos resultados ligeramente superiores que los especificados con el argumento "-b" del comando Iperf, pero al especificar una carga de 100 Mbits/s, obtenemos resultados inferiores.

Al parecer es un error del Mininet desconocido [56] , en nuestro caso no es un problema grave, simplemente lo vamos a tener en cuenta en los resultados.

5.2.2 Emulación con algoritmo STP

Comenzaremos con una configuración pequeña de Spine-Leaf que será [2,2] con 10 host, utilizando el algoritmo de STP. Dicha configuración la podemos visualizar en la Figura 77.

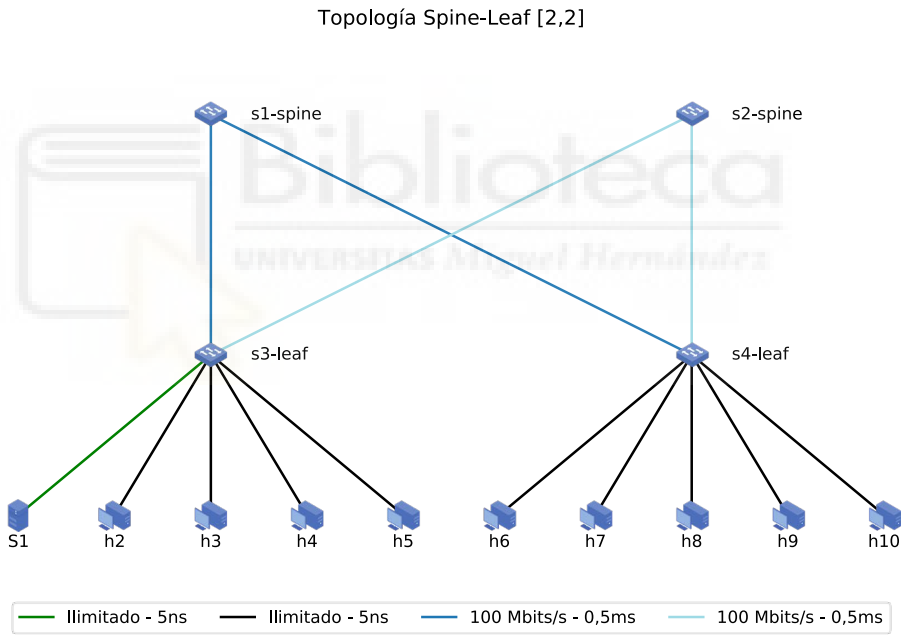


Figura 77: Spine-Leaf [2,2] con 10 host

El resultado lo podemos ver en la Figura 78, el ancho máximo es cercano de los 100 Mbits/seg, eso es debido porque los enlaces entre los switches están limitados a 100 Mbits/seg, como podemos observar en la leyenda de la Figura 77. Al ser flujo único, solo puede tomar un único camino, por ello, el ancho de banda será el del enlace con el ancho de banda menor, que forme parte de la ruta. Si nos fijamos determinadamente, podemos ver que no llega a los 100Mbits/s, en el apartado 5.2.1 explica el motivo.

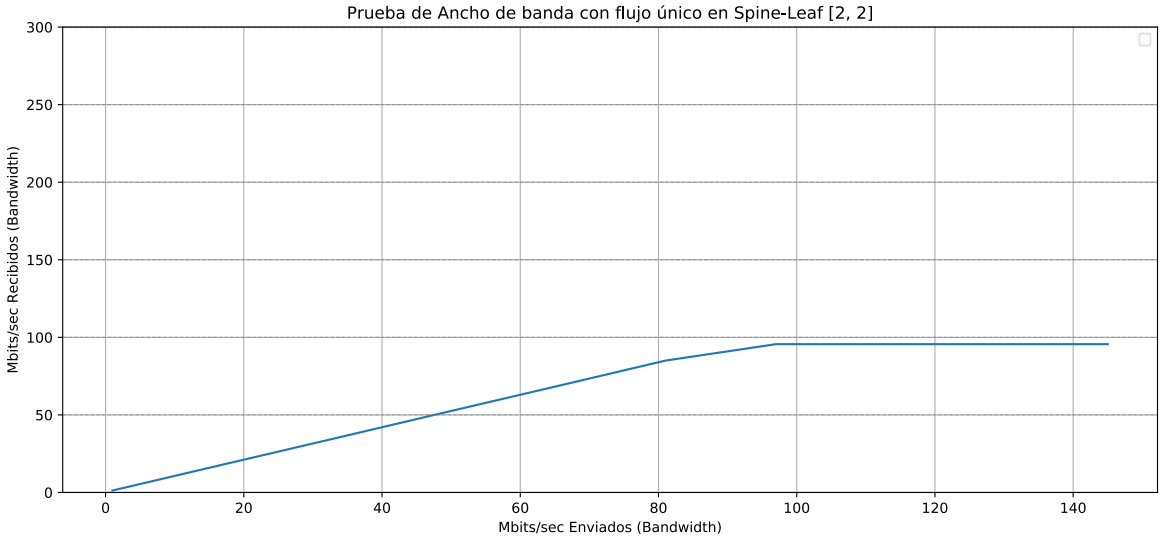


Figura 78: Flujo único en Spine-Leaf [2,2]

Ahora emulamos varias configuraciones a la vez, Figura 79 , para poder ver las distintas diferencias en los anchos de banda. En todas las configuraciones se utilizó 8 host.

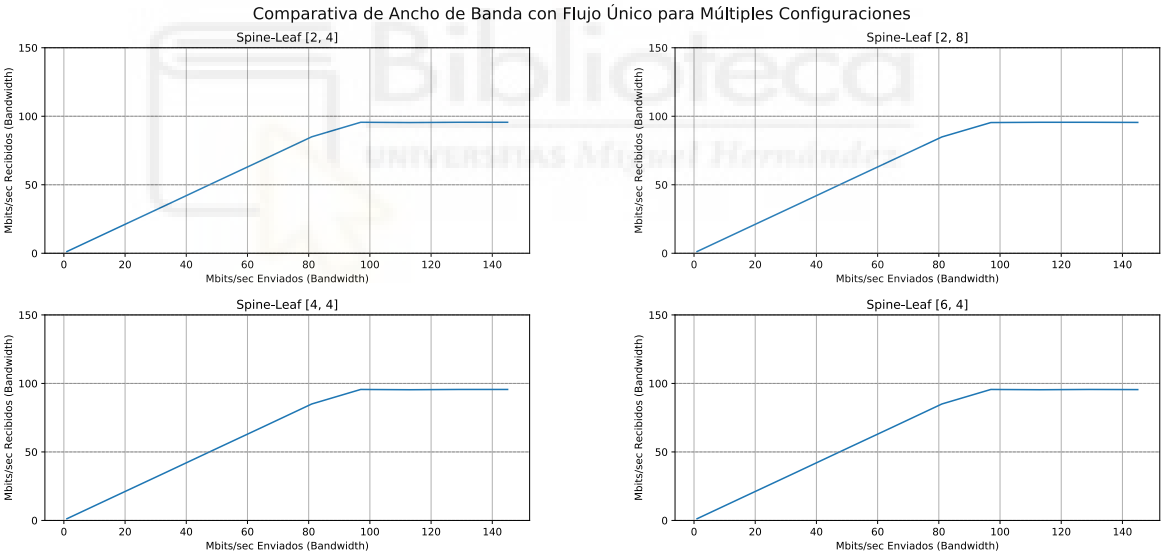


Figura 79: Flujo único en varias configuraciones

Como podemos ver, no hay incrementos en los anchos de bandas, al aumentar el número de switches en la topología, por el mismo motivo que explicado anteriormente, el camino que toma el flujo es único, entonces solo recorrerá un único camino.

5.2.3 Emulación con ECMP

Ahora lo mismo que anteriormente, pero con el algoritmo ECMP, comenzamos con la configuración Spine-Leaf [2,2] que se muestra en la Figura 77.

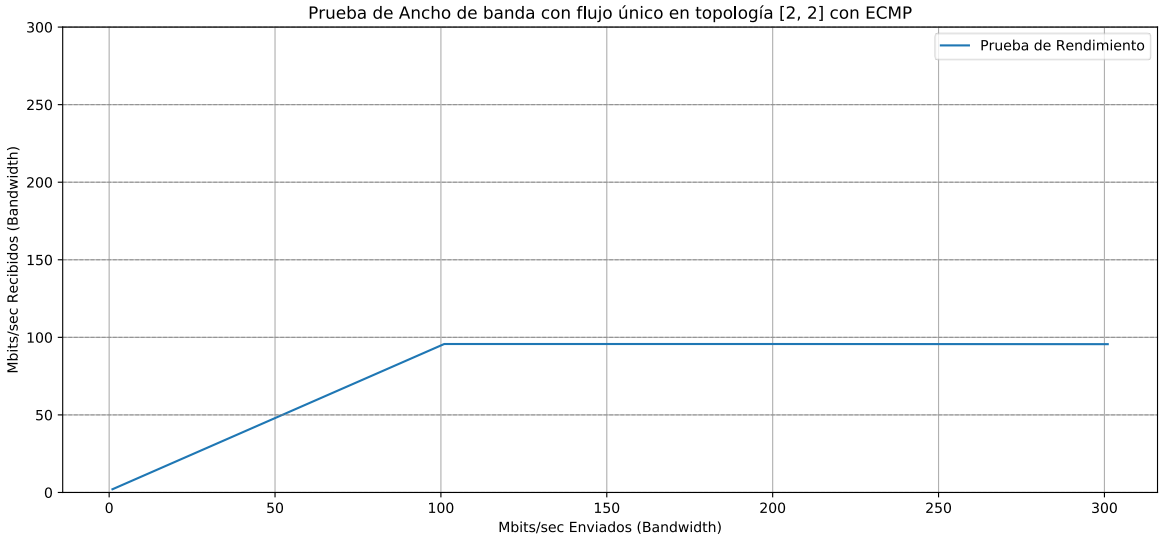


Figura 80: Flujo único en Spine-Leaf [2,2] con ECMP

Si analizamos el resultado del ECMP, Figura 80, podemos darnos cuenta de que es el mismo que en algoritmo STP, Figura 78.

Para descartar la posibilidad de que el motivo sea en el número de switches, extrapolamos la prueba a más configuraciones, Figura 81. En todas las configuraciones se utilizó 8 host.

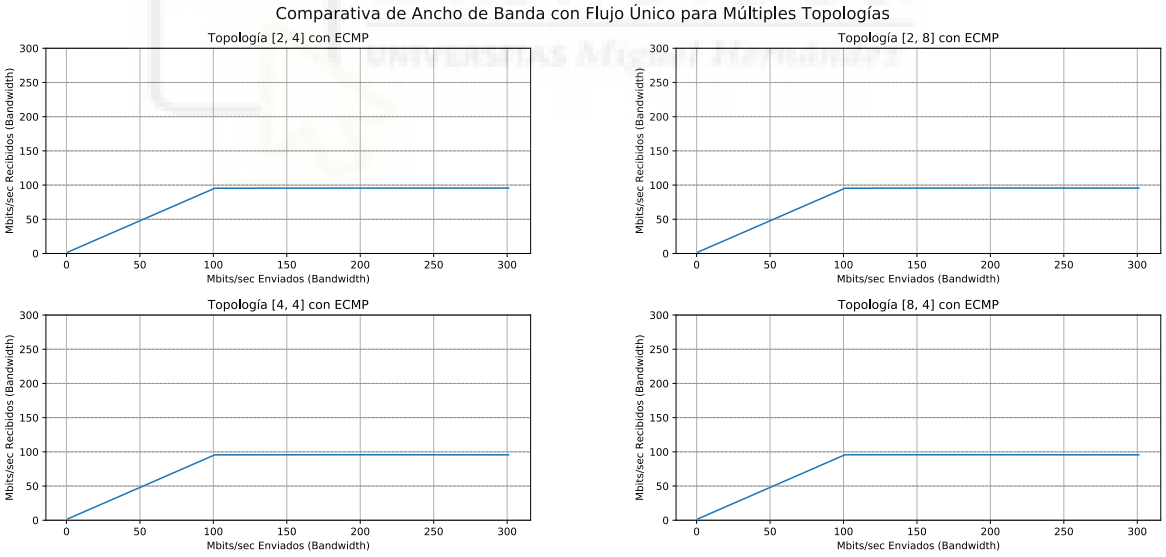


Figura 81: Flujo único en varias configuraciones con ECMP

El resultado lo podemos observar en la Figura 81, visualizamos que, en todas las configuraciones, el límite son 100 Mbits/s. Esto es debido que los enlaces entre los switches están limitados a 100 Mbits/s y es un único flujo, por lo tanto, solo puede recorrer un camino único, igual que en algoritmo STP.

5.2.4 Conclusión

Los resultados son claros, con un único flujo, ambos algoritmos tienen el mismo rendimiento, es verdad que en ECMP, tiene más caminos posibles, pero solo puedo recorrer un único camino a la vez.

Con esta prueba probamos que con flujo único no somos capaces de aprovechar la topología Spine-Leaf.

5.3 Flujo Múltiple

Mediremos el ancho de bando con múltiples flujos paralelos, desde el ultimo host, que será el cliente, hacia el servidor que será el host 1, lo mismo que con flujo único pero ahora con múltiples flujos paralelos.

En la Figura 82, observamos el flujo múltiple en acción, desde h4, generamos 2 flujos paralelos hacia el servidor, en esta ocasión los flujos recorren caminos distintos, pero es posible que ambos utilizan el mismo enlace.

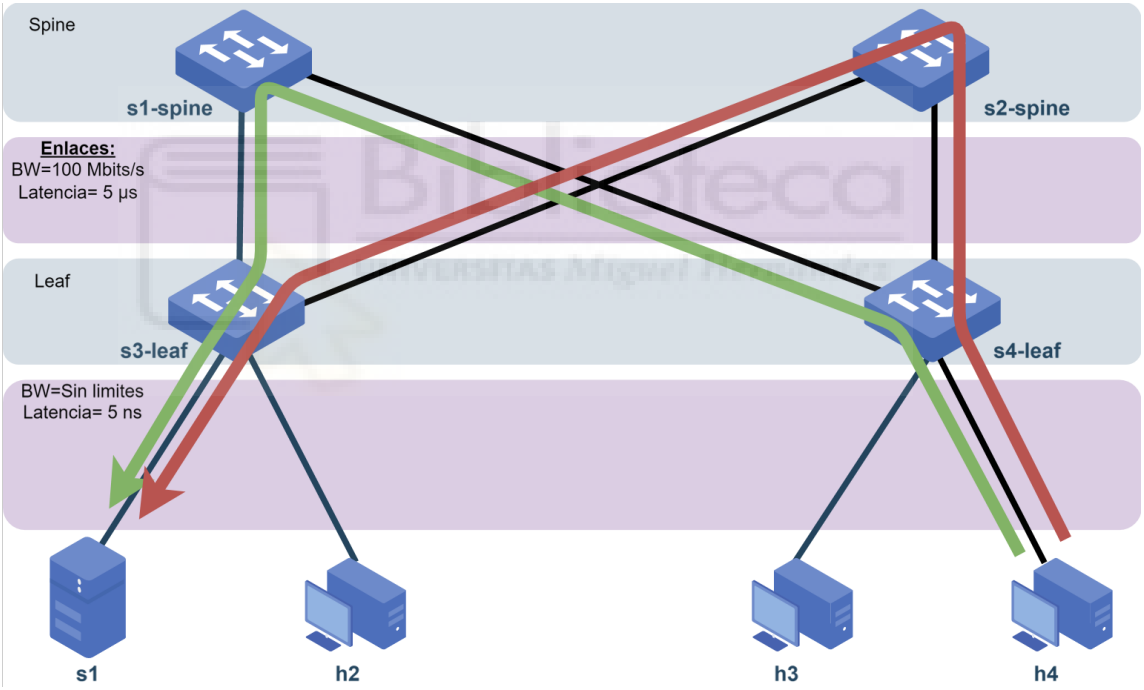


Figura 82: Flujo múltiple

5.3.1 Máximo teórico

El máximo teórico en el término de ancho de banda está definido por los switches Spine que existen en la topología.

Para explicarlo, observaremos la Figura 83, imaginamos que el h4 lanza distintos flujos hacia el servidor (s1), al tener 2 switches Spine, tiene 2 caminos distintos para llegar al servidor, porque todos los switches Spine están conectados con todos los switches Leaf.

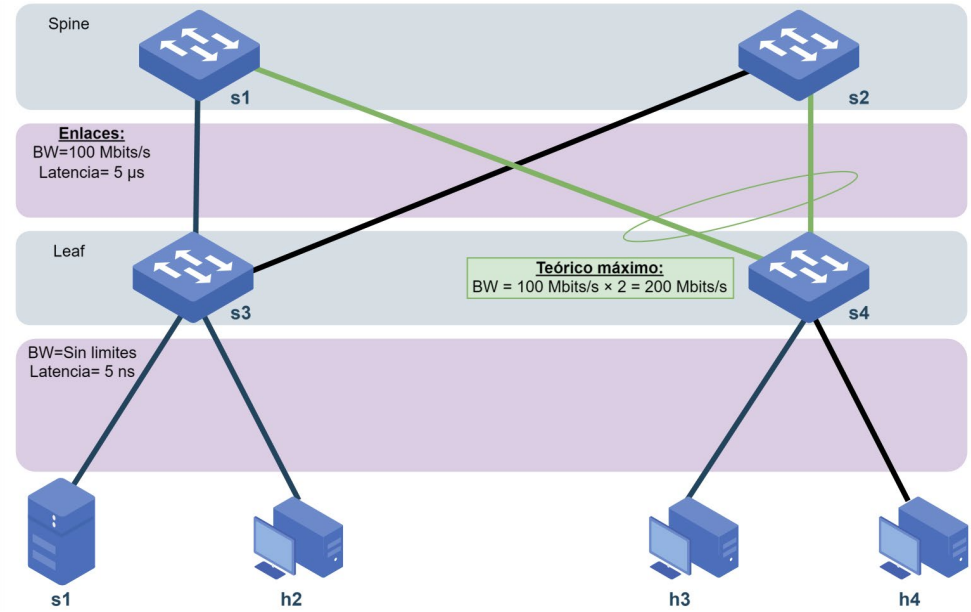


Figura 83: Máximo teórico

En nuestras topologías, todos los enlaces entre los switches están limitados a 100 Mbits/s, en cambio los enlaces entre hosts y switches no tienen limite, en el lado izquierdo de la Figura 83 esta especificado.

La cantidad de enlaces que están conectados con el switch Leaf, define el ancho de banda máximo. Por ello, se puedo afirmar que dicha configuración, Figura 83, presenta un ancho de banda máximo de 200 Mbits/s entre el servidor y h4, ya que sumamos los anchos de banda de los enlaces correspondientes. Si tuviéramos una configuración de 6 switches Spine, entonces tendríamos un máximo teórico de 600 Mbits/s.

Esta propiedad se descubre de forma experimental, pero la teoría lo colabora. A lo largo del apartado 5.3.4 , en las distintas emulaciones, descubriremos la dependencia del ancho de banda con los switches Spine.

5.3.2 Inconveniente con ECMP

En este caso se explicará un inconveniente con ECMP, se trata del comando "select" del protocolo OpenFlow, es la se encarga de seleccionar los caminos para los distintos flujos paralelos.

Dicho comando selecciona los caminos mediante una función hash, asigna de forma aleatoria los flujos a los distintos enlaces (rutas) que dispone, por probabilidad puede ocurrir que se asignen muchos flujos a un único enlace y por tanto saturarlo, dejando otros enlaces libres. Por ello al utilizar un mayor número de flujos, distribuimos los flujos de forma más uniforme a los distintos enlaces, y por ello, obtenemos un ancho de banda mayor. Al tener un número mayor de flujos, el tráfico total se divide en flujos más pequeños, y al solapar en un enlace, la perdida en ancho de banda es menor, ya que el tamaño del flujo es menor.

Por ello, no es posible de alcanzar el ancho de banda máximo teórico de la topología Spine-Leaf con un numero de flujos moderado.

En el apartado 2.4.2.3 se explica el factor estadístico y como disminuirlo.

Para entenderlo mejor el problema, se explica un caso práctico:

Testeando en una topología Spine-Leaf [4,4] (4 switches en cada capa) con 4 host y con 4 flujos paralelos (del h4 al h1) de 100 Mbits/s cada uno, con esta configuración deberíamos obtener un rendimiento máximo teórico de 400 Mbits/s.

La salida de la consola es la siguiente:

```
root@SDN-USER:/home/sdn/Simulaciones# iperf -c 10.0.0.1 -b 100m -P 4
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size:  170 KByte (default)
------------------------------------------------------------
[  5] local 10.0.0.4 port 53362 connected with 10.0.0.1 port 5001
[  6] local 10.0.0.4 port 53364 connected with 10.0.0.1 port 5001
[  7] local 10.0.0.4 port 53370 connected with 10.0.0.1 port 5001
[  8] local 10.0.0.4 port 53376 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec   119 MBytes   100 Mbits/sec
[  8]  0.0-10.0 sec   119 MBytes   100 Mbits/sec
[  7]  0.0-10.5 sec   101 MBytes  80.9 Mbits/sec
[  6]  0.0-10.6 sec  31.4 MBytes  24.8 Mbits/sec
[SUM]  0.0-10.6 sec   371 MBytes   294 Mbits/sec
```

El comando “select” asigna los flujos 3 y 4 al mismo enlace, y de allí la suma total cerca de los 100 Mbits/s (máximo del enlace), y por ello, obtenemos el valor máximo practico de 294 Mbits/s de los 400 Mbits/s.

Pero si hacemos otra vez la misma prueba:

```
root@SDN-USER:/home/sdn/Simulaciones# iperf -c 10.0.0.1 -b 100m -P 4
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size:  170 KByte (default)
------------------------------------------------------------
[  8] local 10.0.0.4 port 53208 connected with 10.0.0.1 port 5001
[  5] local 10.0.0.4 port 53180 connected with 10.0.0.1 port 5001
[  6] local 10.0.0.4 port 53200 connected with 10.0.0.1 port 5001
[  7] local 10.0.0.4 port 53186 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  7]  0.0-10.1 sec   117 MBytes  97.9 Mbits/sec
[  6]  0.0-10.1 sec  47.6 MBytes  39.6 Mbits/sec
[  8]  0.0-10.1 sec  39.4 MBytes  32.7 Mbits/sec
[  5]  0.0-10.6 sec  46.4 MBytes  36.6 Mbits/sec
[SUM]  0.0-10.6 sec   251 MBytes   198 Mbits/sec
```

Esta vez obtenemos un ancho de banda menor, con las mismas condiciones que en el caso anterior.

Aunque también se ha obtenido el máximo teórico con las mismas condiciones:

```
root@SDN-USER:/home/sdn/Simulaciones# iperf -c 10.0.0.1 -b 100m -P 4
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size:  170 KByte (default)
```

```
------------------------------------------------------------
[  8] local 10.0.0.4 port 59446 connected with 10.0.0.1 port 5001
[  6] local 10.0.0.4 port 59436 connected with 10.0.0.1 port 5001
[  5] local 10.0.0.4 port 59434 connected with 10.0.0.1 port 5001
[  7] local 10.0.0.4 port 59444 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  8]  0.0-10.0 sec   119 MBytes   100 Mbits/sec
[  6]  0.0-10.0 sec   119 MBytes   100 Mbits/sec
[  7]  0.0-10.0 sec   119 MBytes   100 Mbits/sec
[  5]  0.0-10.1 sec   119 MBytes  98.5 Mbits/sec
[SUM]  0.0-10.1 sec   477 MBytes   394 Mbits/sec
```

Este factor aleatorio, se minimiza al utilizar muchos flujos paralelos, como se ha comentado anteriormente.

5.3.3 Emulación con algoritmo STP

Primero emularemos una topología que se muestra en la Figura 84 con el algoritmo STP, es casi la misma que en flujo único, pero esta vez tiene 2 switches más en la capa Leaf y tiene 4 host en vez de los 10 host en el caso de Figura 77. El número de host no tiene ningún efecto en la topología porque solo interactúan el servidor y el ultimo host que en este caso es h4.

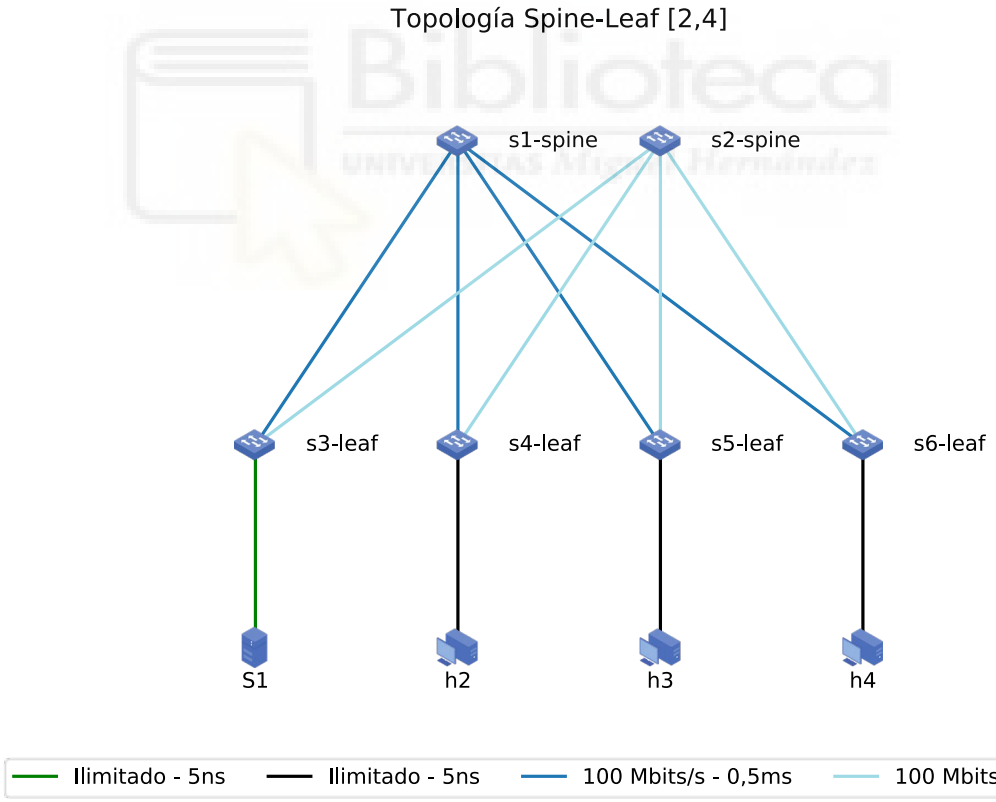


Figura 84: Topología Spine-Leaf [2,4] con 4 host

Si observamos el resultado de la prueba con 8 flujos, en la Figura 85, obtenemos el mismo resultado que en flujo único, Figura 79 (La primera grafica de la izquierda). Este fenómeno lo podemos explicar aplicando el algoritmo STP en la misma configuración, Figura 70 del apartado 5.1.3 , si nos fijamos en el switch 6 de la Figura 70, el puerto que puede utilizar

para enviar/recibir hacia el servidor, es el puerto raíz, entonces solo puede utilizar un enlace, y por ello, el ancho de banda máximo conseguido siempre será 100 Mbits/s. ya que los 8 flujos tienen que compartir el mismo enlace.

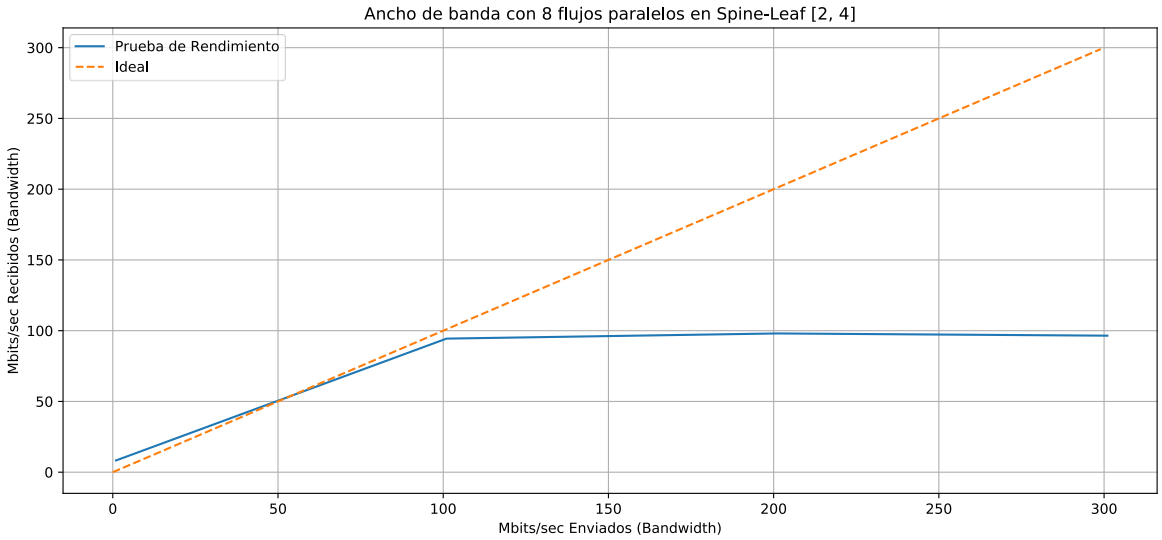


Figura 85: Flujo múltiple en Spine-Leaf [2,4]

Para confirmar la teoría, extrapolamos la prueba a más configuraciones, obteniendo el resultado de la Figura 86, en todas las configuraciones se utilizó 8 host.

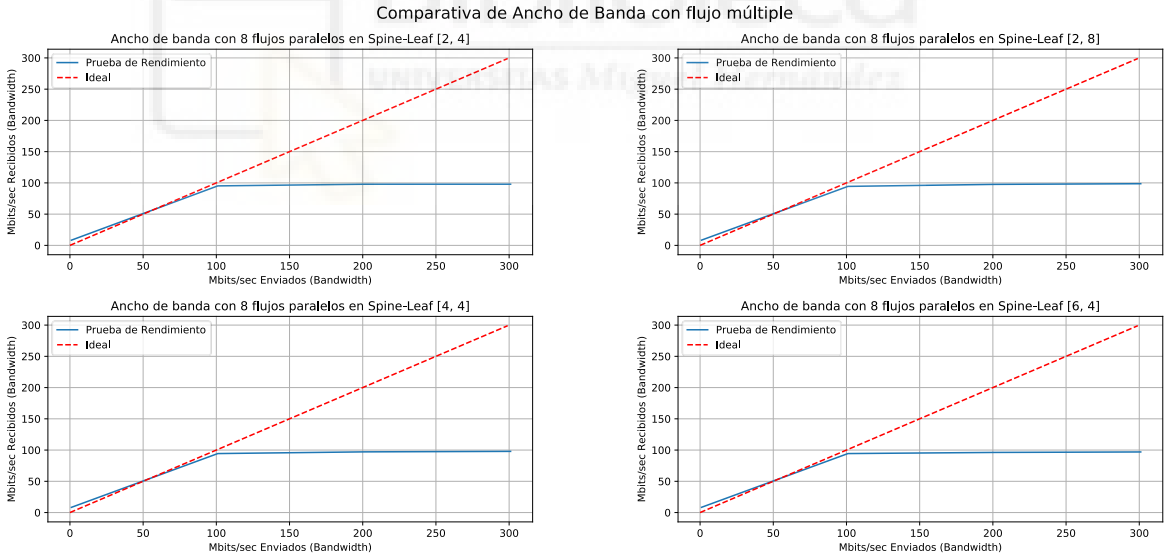


Figura 86: Flujo múltiple a varias configuraciones

El resultado de la Figura 86, es el esperado, con el algoritmo STP, no es posible de obtener un ancho de banda mayor que el del enlace con el menor ancho de banda.

5.3.4 Emulación con ECMP

Comenzamos a aplicar el algoritmo ECMP en pruebas de ancho de banda con flujo múltiple, en una configuración Spine-Leaf [2,4], Figura 84.

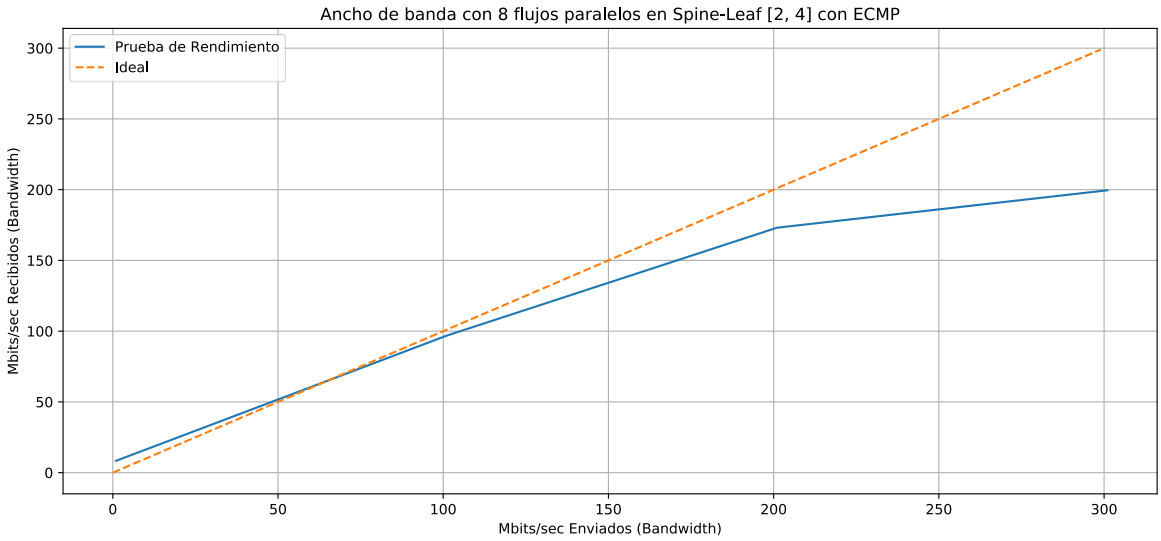


Figura 87: Flujo múltiple en Spine-Leaf [2,4] con ECMP

El resultado de la Figura 87, es distinto que en el caso del algoritmo STP, Figura 85, en este caso es mayor. Teniendo en cuenta que tenemos 2 switches Spine, el ancho de banda máximo teórico son de 200Mbits/s, pero no es capaz de alcanzarlo con una carga de 200Mbits/s, en el gráfico de la Figura 87, alcanza el máximo teórico, pero con una carga de 300 Mbits/s. El concepto de máximo teórico esta explicado en el apartado 5.3.1.

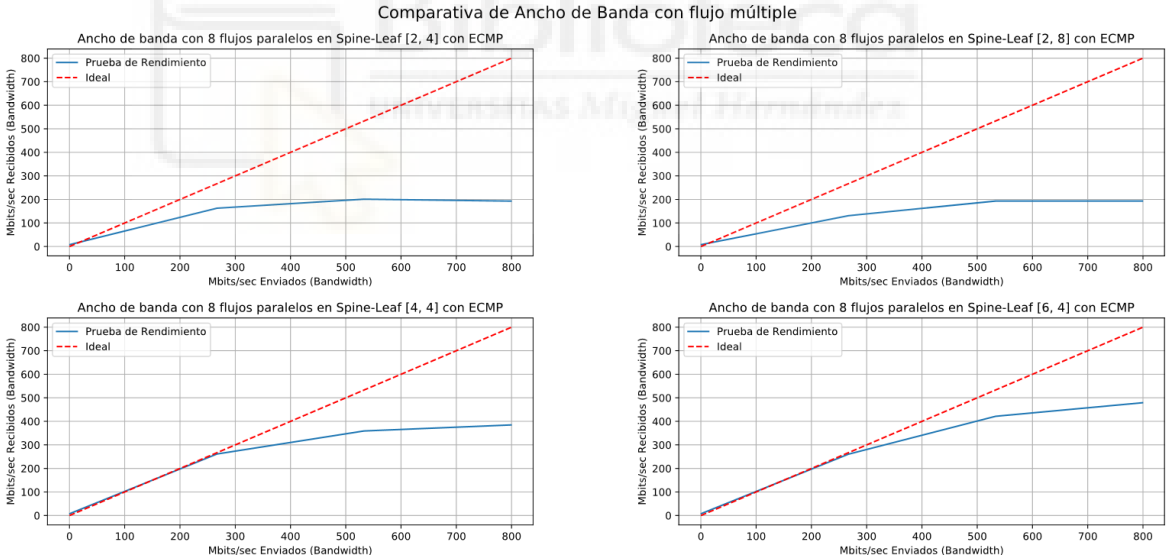


Figura 88: Flujo múltiple en varias configuraciones con ECMP

Extrapolamos la prueba a más configuraciones, Figura 88, lo primero que observamos es que el ancho de banda máximo es definido por el número de switches Spine, dicho fenómeno esta explicado en el apartado 5.3.1.

En los casos de 2 y 4 switches Spine, hemos logrado a llegar al máximo teórico, pero en el caso de 6 switches Spine, no fue así, porque solo alcanzo los 500Mbits/s y el máximo teórico son 600 Mbits/s. El motivo de este resultado esta explicado en el apartado 5.3.2, resumiéndolo, el algoritmo ECMP no es capaz de aprovechar de todos los enlaces disponibles con pocos flujos, en configuraciones de switches Spine más grandes.

Para disminuir el efecto estadístico, haremos pruebas con un mayor número de flujos, comenzando con la configuración de la Figura 89.

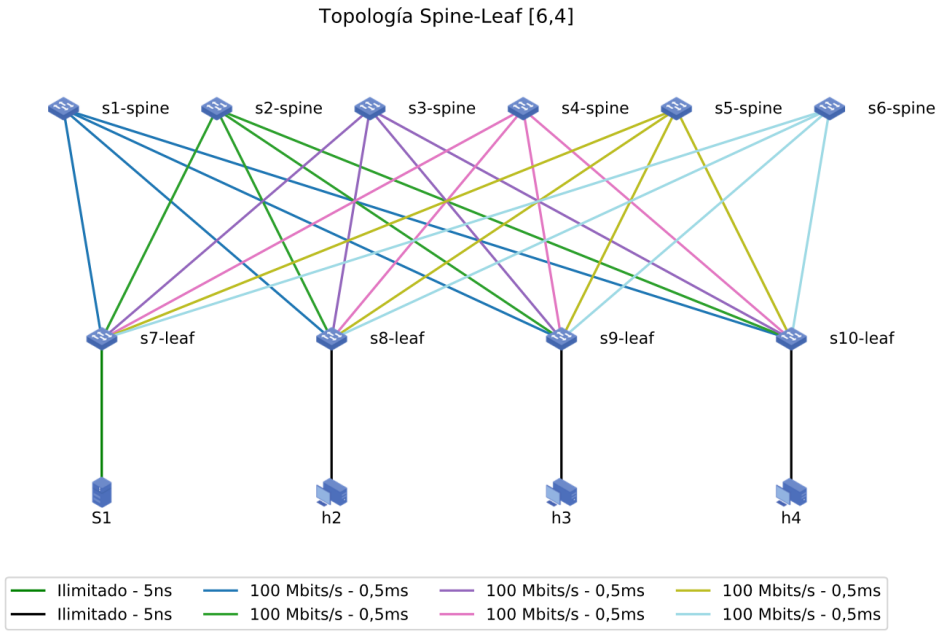


Figura 89: Topología Spine-Leaf [6,4] con 4 host

Si observamos la Figura 90, tenemos el resultado de la configuración Spine-Leaf [6,4] con 32 flujos, si lo comparamos con los 8 flujos de la Figura 88 (la gráfica abajo derecha), visualizamos que presenta ciertas mejoras en el ámbito de la estabilidad y ancho de banda.

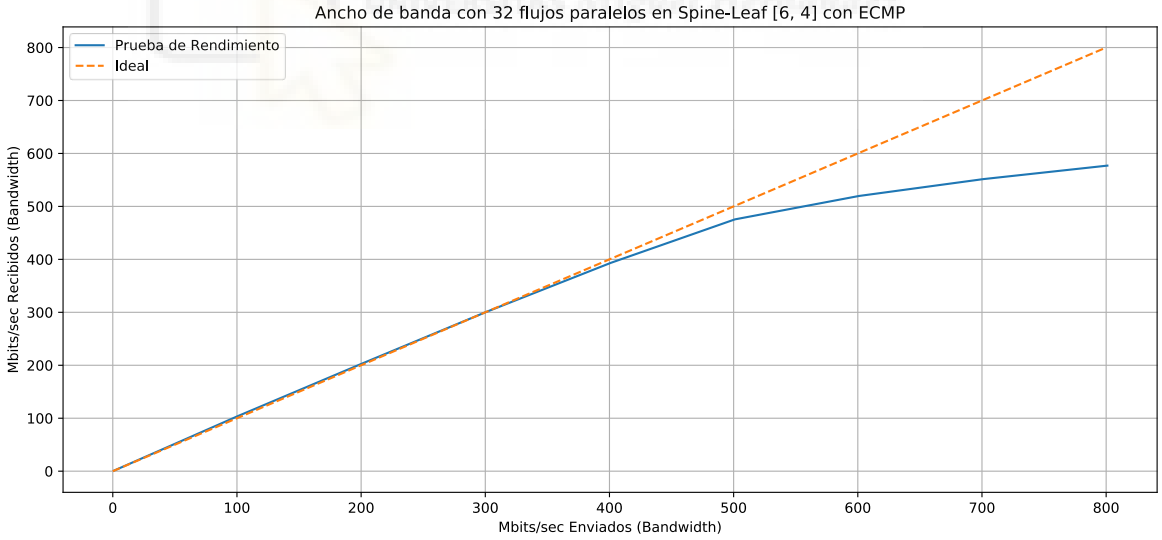


Figura 90: Flujo múltiple en Spine-Leaf [6,4] con ECMP con 32 flujos

Por lo tanto, se puede concluir, que al aumentar el número de flujos se obtiene un aumento de rendimiento, comparando el incremento de 8 a 32 flujos, obtenemos mejora en dos aspectos:

- La dispersión teniendo como referente el caso ideal (línea discontinua naranja) es nula hasta los 400 Mbits/s y posterior es pequeña hasta los 550 Mbits/s. Cabe

aclarar el caso ideal se trata de la transmisión de una cantidad de tráfico y la recepción de la misma cantidad en 1 segundo.

- El ancho de banda máxima aumento de 490 Mbits/s a 577 Mbits/s.

Con la misma configuración, aumentaremos el número de flujos hasta los 500, Figura 91.

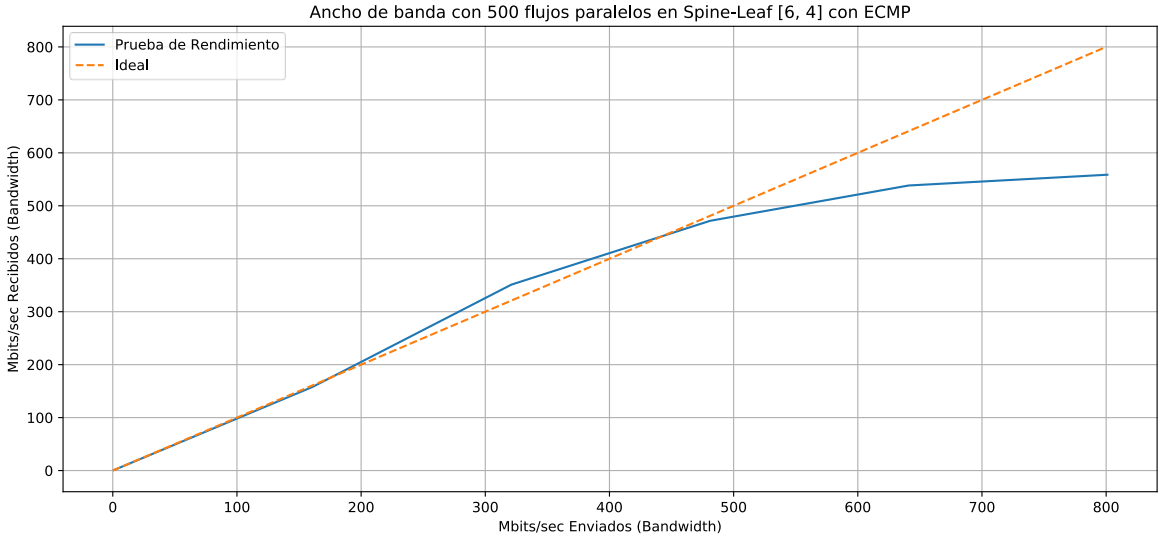


Figura 91: Flujo múltiple en Spine-Leaf [6,4] con ECMP con 500 flujos

En el caso de 500 flujos, Figura 91, no hemos aumentado ningún parámetro de rendimiento.

5.3.5 Conclusión

En el algoritmo STP, está claro, no es capaz de aprovechar la topología Spine-Leaf porque bloquea aquellos enlaces que provocan bucles, de tal manera, dejando solo aquellos necesarios para comunicarse con el switch, de esta manera, a costa de disminuir el ancho de banda.

En ECMP, es capaz de aprovechar de todos los enlaces redundantes en topologías pequeñas, y por ello incrementar el ancho de banda varias veces en comparación del algoritmo STP.

Al aumentar el tamaño de las topologías, ECMP no es capaz de balancear correctamente los flujos a los enlaces, por ello, no es capaz de aprovechar totalmente de la topología, pero la perdida de rendimiento no es preocupante.

Para aumentar el rendimiento en topologías más grande, se incrementó el número de flujos, se observa que con 32 flujos se obtiene mejoras frente a los 8 flujos, pero al aumentar a 500 flujos, se obtiene el mismo resultado que con 32 flujos. Estos resultados corresponden a una configuración [6,4], una topología con más cantidad de switches Spine, necesitara más flujos para aprovechar todo el rendimiento posible.

5.4 FLUJO DISTRIBUIDO

En esta prueba intervienen todos los hosts de la topología, desde cada uno de los hosts ejecutaremos una prueba de ancho de banda hacia el servidor.

En la Figura 92 podemos observar el flujo distribuido, que es generado por los hosts que no son adyacentes al servidor, debido a que queremos obligar al tráfico fluir por la topología, para ello no conectamos ningún host adyacente con el servidor, ya que aquellos que están conectados en el mismo switch que el servidor, tienen un camino directo y por lo tanto obtendríamos resultados mayores de lo que realmente.

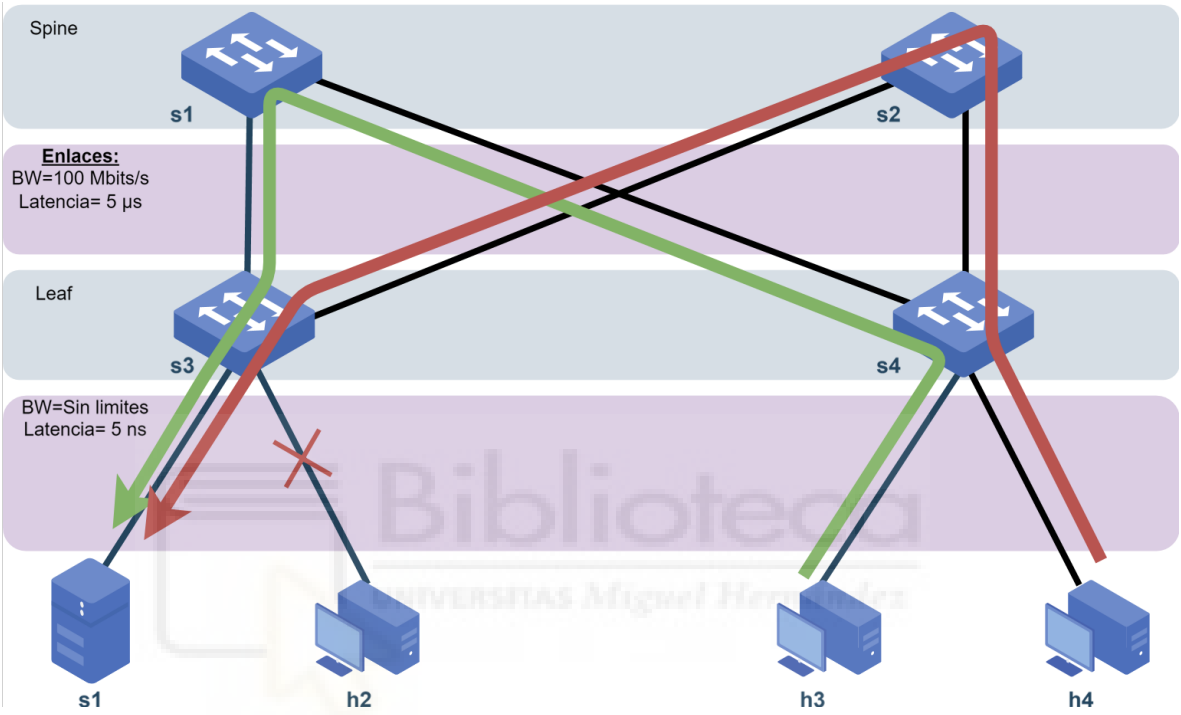


Figura 92: Flujo distribuido

5.4.1 Emulación con algoritmo STP

Vamos a comenzar con nuestra configuración típica de Spine-Leaf [2,4] con el algoritmo STP, Figura 93 ,pero ahora con el detalle de que tendremos los hosts adyacentes al servidor (h1) bloqueados.

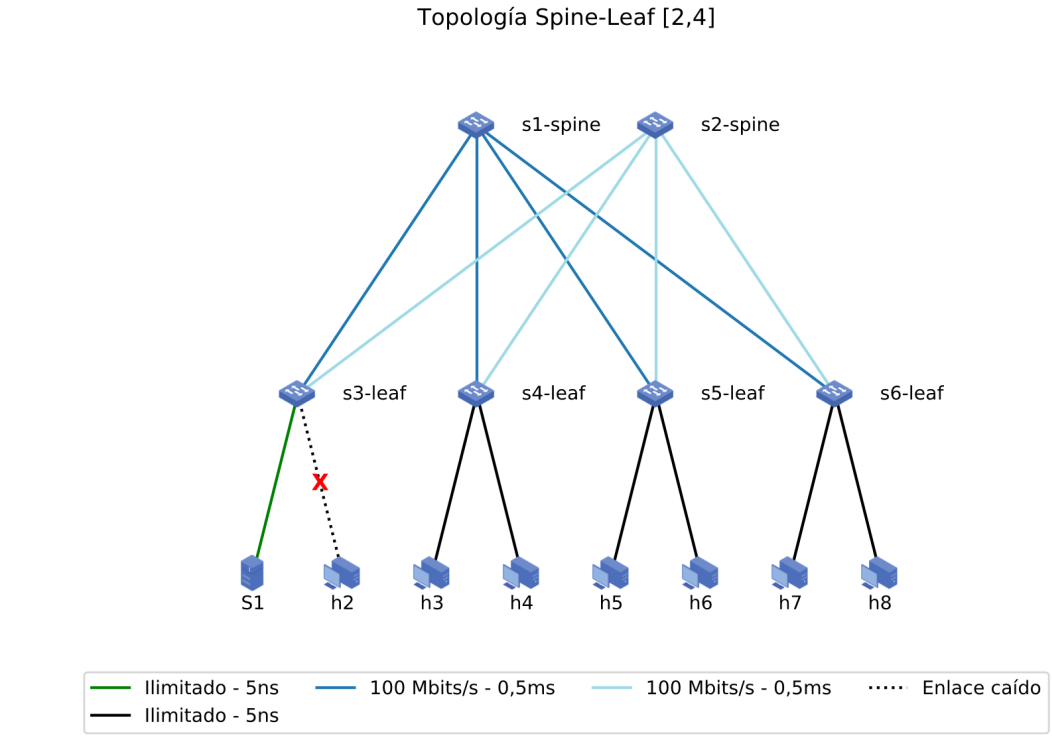


Figura 93: Topología Spine-Leaf [2,4] con 8 host

El resultado del ancho de banda con flujos distribuidos con algoritmo STP, se encuentra en la Figura 94, como podemos observar es el mismo resultado que en flujo único, Figura 79, y múltiple Figura 85 . El resultado es el mismo que en casos anterior porque solo hay un único enlace no bloqueado hacia el switch s3-leaf, en él se encuentra el servidor, Figura 70.

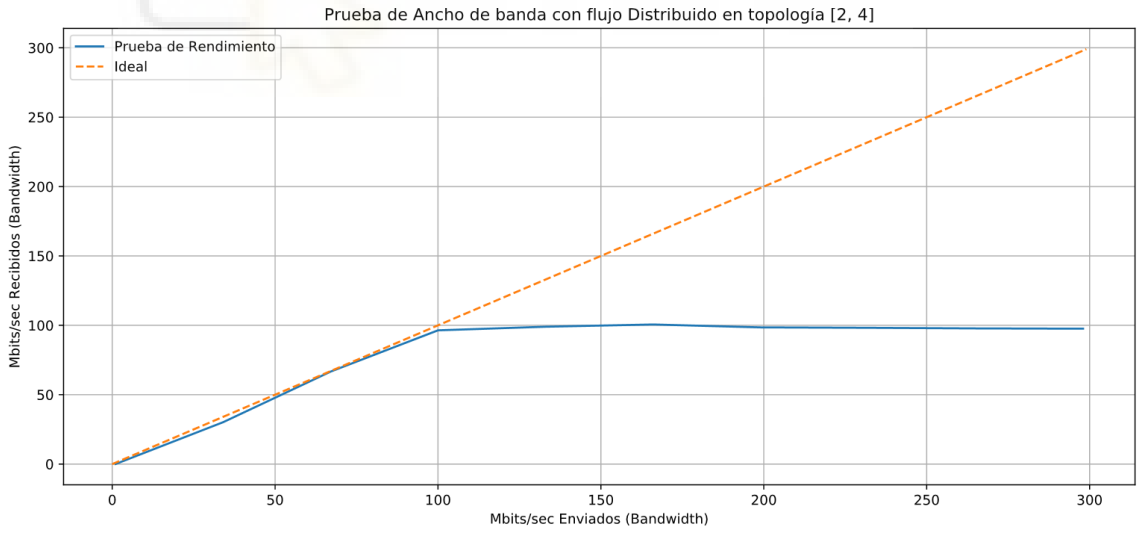


Figura 94: Flujo distribuido en Spine-Leaf [2,4]

Ahora con el mismo número de host probamos con más configuraciones, Figura 95.

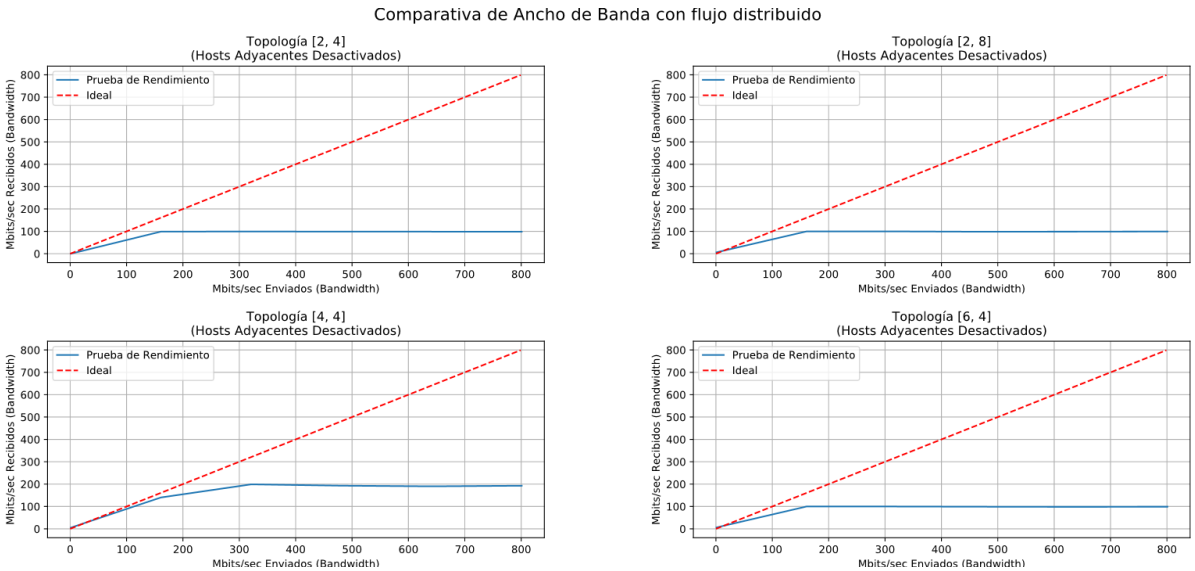


Figura 95: Flujo distribuido en varias configuraciones

El resultado de la Figura 95, es curioso, en la configuración Spine-Leaf [4,4], obtenemos un ancho de banda de 200 Mbits/s (los enlaces entre switches tienen un BW de 100Mbits/s), eso significa que el switch que está el servidor (switch 5, porque los 4 primeros son Spine) tiene 2 enlaces no bloqueados, esto puede ocurrir porque un enlace es el raíz y los demás designados, por lo tanto puede darse el caso, Figura 70. En la Figura 105 podemos consultar la configuración [4,4].

Con el controlador ONOS (los anteriores son con el controlador OpenDaylight) se obtiene resultados distintos en algún caso, en la configuración [2,4], Figura 96, observamos que el ancho de banda es de 200 Mbits/s, aumento el ancho de banda en la misma configuración utilizando otro controlador, eso es debido que elije los enlaces a bloquear distinto o la switch raíz es otro, pero al final, es lo mismo, bloquear enlaces para evitar los bucles.

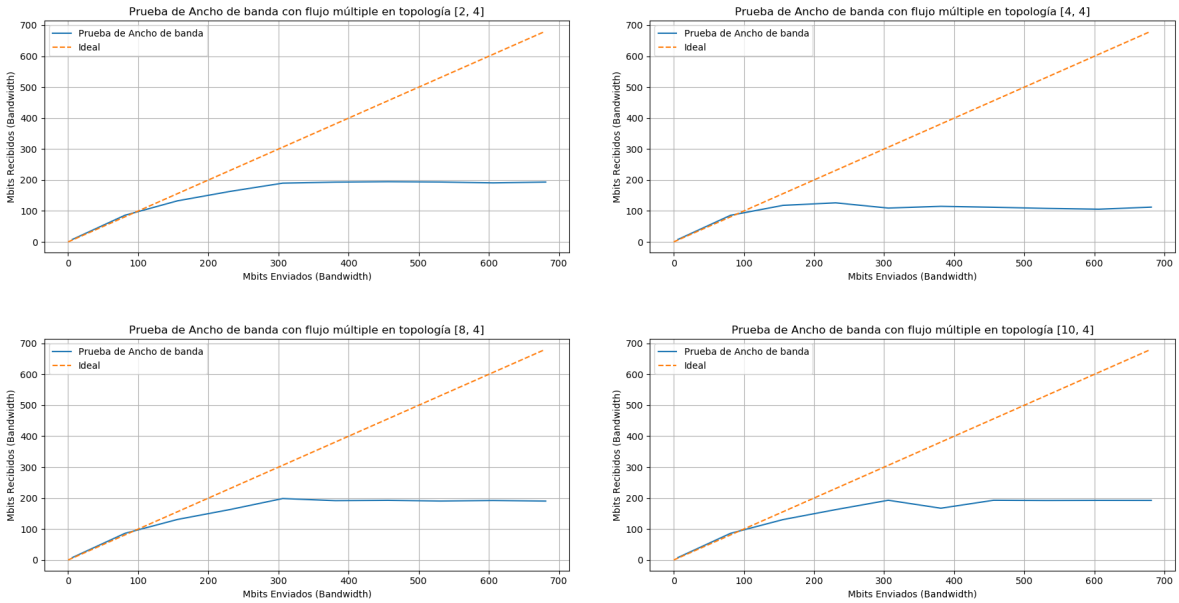


Figura 96: Flujo múltiple a varias configuraciones con ONOS

Dependiendo del número de switch de las capas, el ancho de banda varía entre 100 o 200 Mbits/s, esto es un problema, porque no aprovechamos la topología Spine-Leaf. Lo mismo que en flujo múltiple.

5.4.2 Emulación con ECMP

Con la misma configuración inicial que en los demás casos, Spine-Leaf [2,4] con 8 host, Figura 93, obtenemos el resultado de la Figura 97.

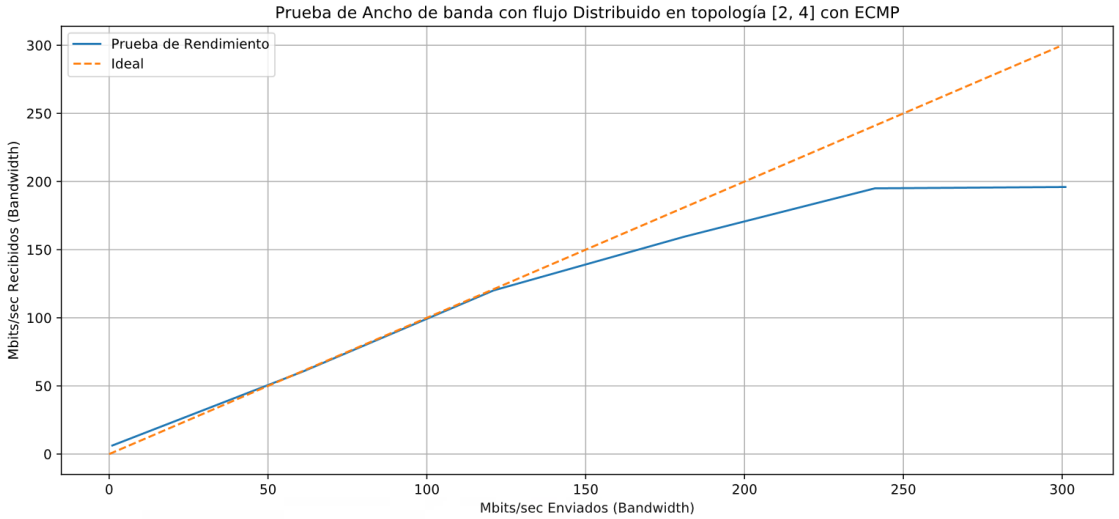


Figura 97: Flujo distribuido en Spine-Leaf [2,4] con ECMP

El resultado es mejor que con algoritmo STP, Figura 94, pero muy similar que en flujo múltiple con ECMP, Figura 87, esto seguramente sea debido porque, aunque utilicen distintos enlaces de switch Leaf al switch Spine, los flujos acaben en el mismo switch Spine, y dicho switch solo tendrá un único enlace conectado al switch Leaf destino. Lo explico con la Figura 98.

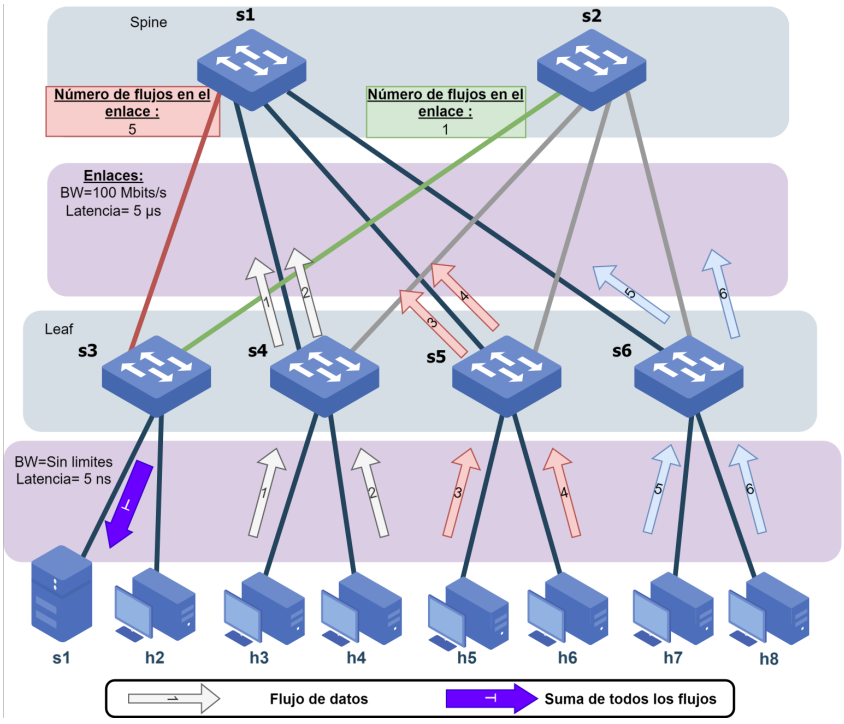


Figura 98: Saturación de enlaces

Podemos observar en la Figura 98, que todos los hosts envían flujos hacia el servidor, en cada switch Leaf (s4, s5, s6) tienen que elegir uno de los dos enlaces que conectan con los switches Spine, si muchos eligen el mismo switch Spine, lo que provoca que dicho switch se sature, ya que solo dispone de un único enlace hacia el servidor, dicho enlace se saturaría.

Si suponemos que los flujos son de 35 Mbits/s cada uno, y como muestra la Figura 98, 5 flujos eligieron el switch 1 y 1 flujo eligió el switch 2, entonces en el primer switch tenemos una carga total de 175 Mbits/s ($5 \times 35 Mbits/s$) y en el segundo switch tendría una carga de 35 Mbits/s. Como los enlaces entre switches tienen un límite de 100 Mbits/s, tendríamos una pérdida de 75Mbits/s en ancho de banda, en el enlace que comunica el switch 1 con el switch 3. El total de ancho de banda seria la suma de los 100Mbits/s del switch 1 y los 35Mbits/s del switch 2, en total 135Mbits/s de los 200 Mbits/s que realmente deberíamos obtener.

Ahora extrapolamos a más configuraciones, pero con 32 host, Figura 99, hay que aclarar que en este tipo de prueba sí que importa el número de host, no por la carga total de tráfico, ya que este se distribuye de forma uniforme entre los hosts activos, si no, por el número de flujos, ya que cada host activo genera un flujo (excluyendo el servidor).

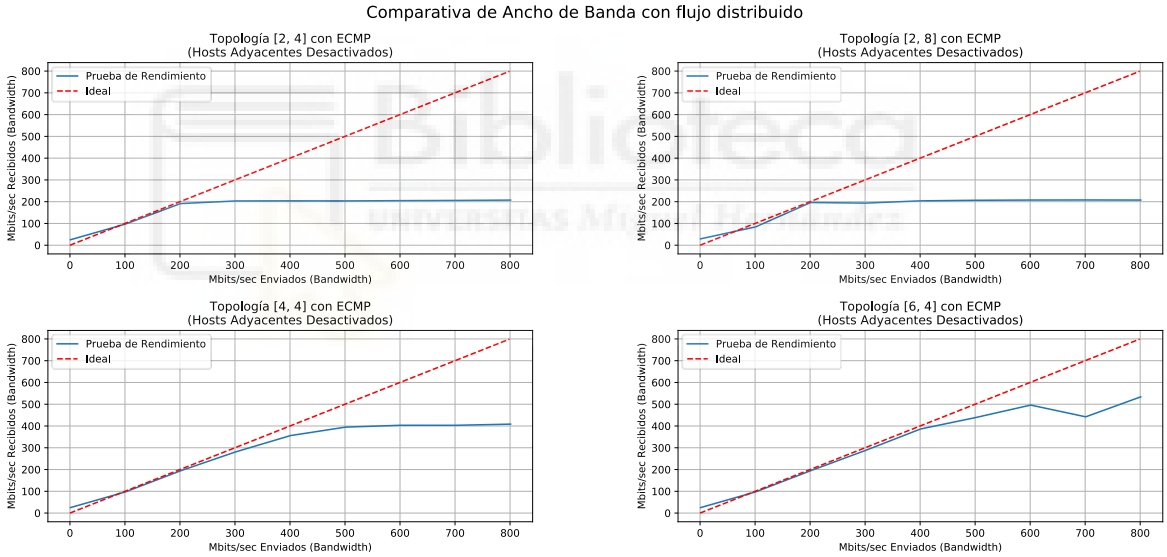


Figura 99: Flujo distribuido en varias configuraciones con ECMP

En los resultados de la Figura 99, comparando con el flujo múltiple, Figura 88, son resultados parecidos.

Si observamos las otras graficas de la Figura 99, podemos ver que al tener más switches Spine, es más difícil de llegar al límite teórico, lo mismo que pasaba con flujo múltiple, apartado 5.3.4.

5.4.3 Conclusión

En el algoritmo de STP, dependiendo de la configuración de Spine-Leaf, obtenemos más ancho de banda que en flujo múltiple, pero este aumento es solo casual, ya que dependerá de cómo el algoritmo STP bloqueará los enlaces.

En ECMP, obtenemos resultados parecidos que, en flujo múltiple, esto se debe por el motivo de que, en flujo múltiple, el tráfico viajaba desde un cliente hacia el servidor y se saturan los enlaces que interconectan los switches Leaf origen y los switches Spine, pero en flujo distribuido se satura los enlaces salientes de los switches Spine hacia el switch Leaf destino. La pérdida de ancho de banda en ambos casos de flujos defiere ligeramente, por ello los resultados son parecidos.

El tráfico se genera desde distintos orígenes, por ello, esta prueba refleja con mayor fidelidad las condiciones del mundo real.

5.5 LATENCIA CON CARGA DE FLUJOS DISTRIBUIDOS

En este caso combinamos la prueba de latencia, con la de flujo múltiple y flujo distribuido, para tener la posibilidad de visualizar el comportamiento de la latencia dependiendo de la carga de trabajo existente en la red emulada.

5.5.1 Emulación con algoritmo STP

En este caso comenzamos con la configuración Spine-Leaf [2,4] con el algoritmo STP, la Figura 100, la que hemos utilizado en pruebas anteriores.

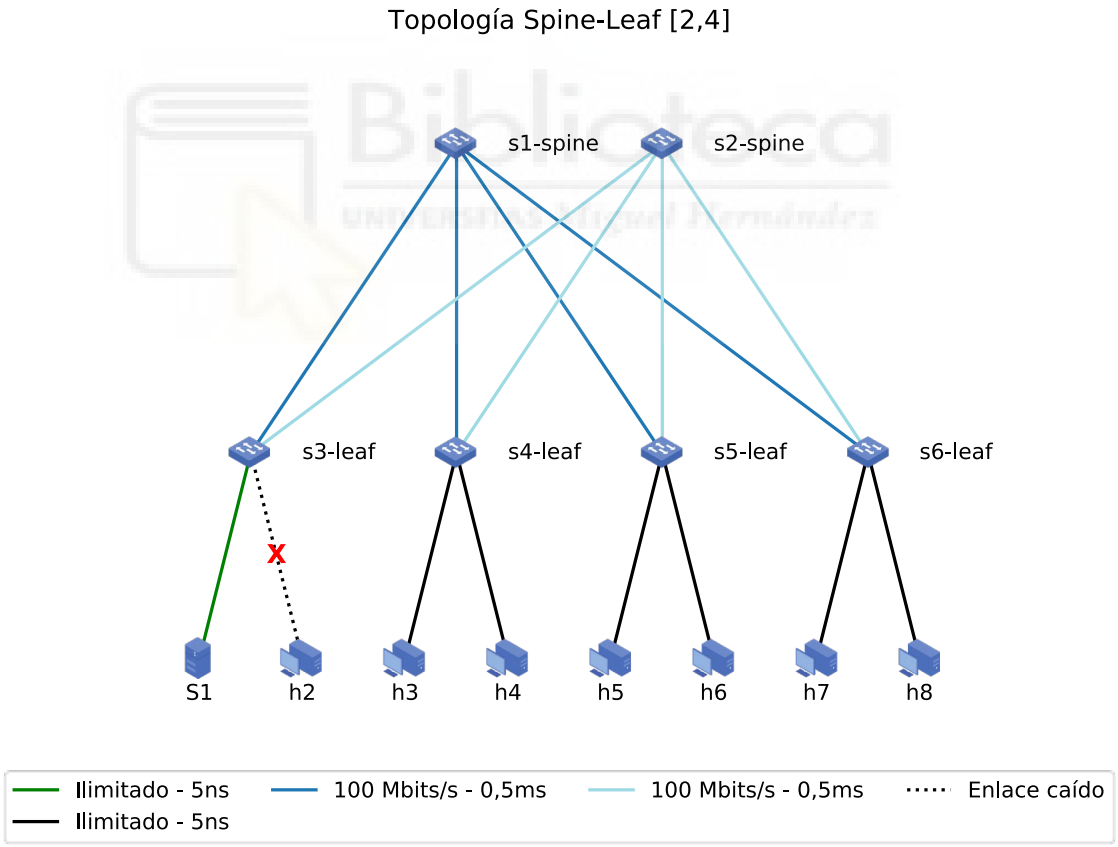


Figura 100: Topología Spine-Leaf [2,4] con 8 host

En la Figura 101, podemos observar el resultado de la latencia en presciencia de distintas cargas de trabajo, en primer lugar, visualizamos que hasta los 90 Mbits/s, la latencia se mantiene muy cerca con el caso de la carga nula.

Pero una vez que pasamos los 100 Mbits/s, la red se satura, y la latencia aumenta de forma exponencial, por ello el eje Y, está en escala logarítmica.

Por lo tanto, sufrimos el mismo problema de la limitación de ancho de banda por el algoritmo STP. El ancho de banda está limitado a 100 o 200Mbits, dependiendo de la configuración.

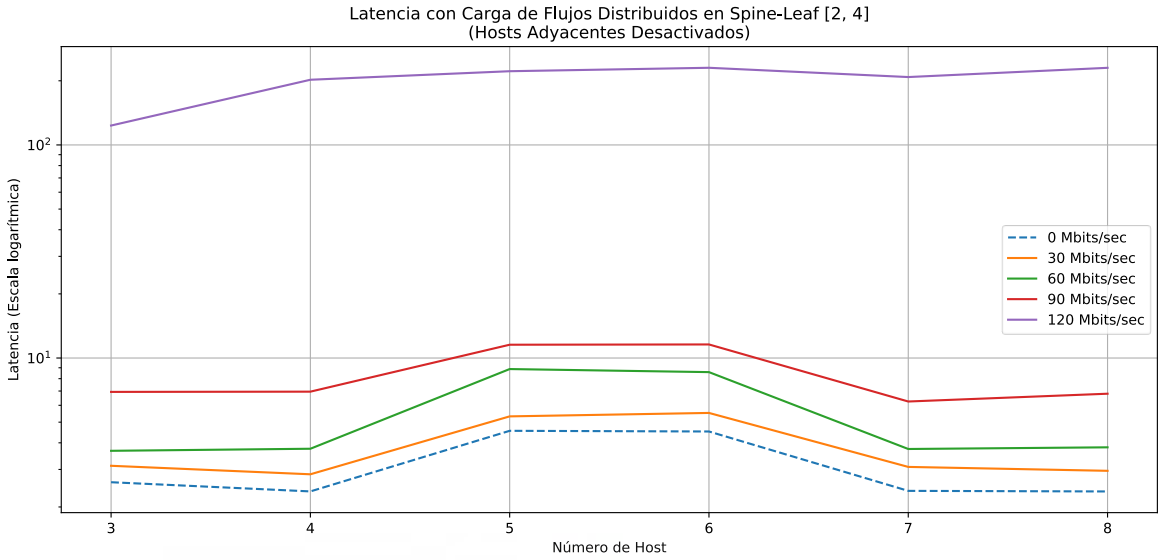


Figura 101: Latencia bajo carga en Spine-Leaf [2,4]

Para dimensionar el incremento de la latencia por saturación en los resultados de la Figura 101, podemos consultar la Tabla 11, que muestra los valores medios de la latencia en cada host dependiendo de la carga de trabajo.

Carga de Trabajo (Mbits/s)	h2	h3	h4	h5	h6	h7	h8
0	0.301	2.255	2.205	4.349	4.316	2.274	2.284
30	0.114	2.247	2.422	4.825	4.95	3.813	2.738
60	0.093	6.298	3.052	8.609	8.996	4.436	3.019
90	0.088	4.551	5.422	8.355	10.304	4.428	4.544
120	0.088	120.323	152.009	236.361	166.465	165.183	168.199

Tabla 11: Valores medios de los pings en bajo carga de trabajo

5.5.2 Emulación con ECMP

En este caso probaremos el algoritmo ECMP, comenzando con la misma configuración que en el caso del algoritmo STP.

Comenzamos a realizar las pruebas a la configuración Spine-Leaf [2,4] con host 8, Figura 100, en realidad son solo 6 hosts activos, ya que el servidor y h2 no generan tráfico.

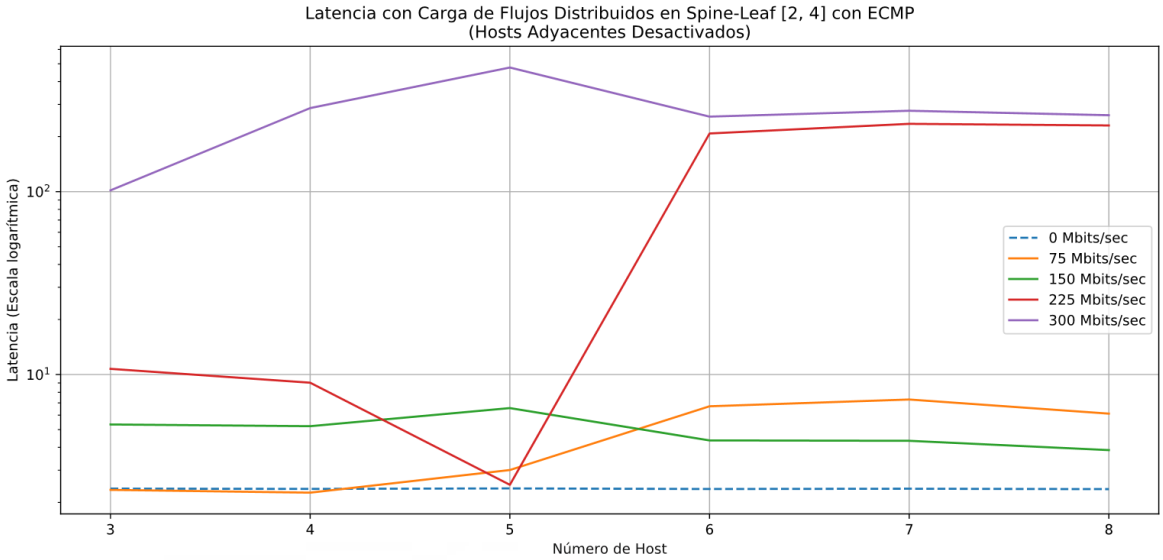


Figura 102: Latencia bajo carga en Spine-Leaf [2,4] con ECMP

En la Figura 102, observamos que el máximo teórico está próximo a 200Mbits/s, y al superar ese umbral, la red se satura. Existe cierto ruido en la gráfica por los resultados de la carga de trabaja de 225 Mbits/s, en ese caso el Ping es capaz de encontrar caminos con menos latencia en los hosts {3,4,5}.

Al hacer más emulaciones, obtenemos la mayor parte de las pruebas resultados con ruido, como en la Figura 103. La latencia no es estable, el Ping es capaz de encontrar un camino con menos carga de tráfico, ya que recordemos que la distribución del tráfico en los enlaces no es totalmente equilibrada, apartado 5.3.2.

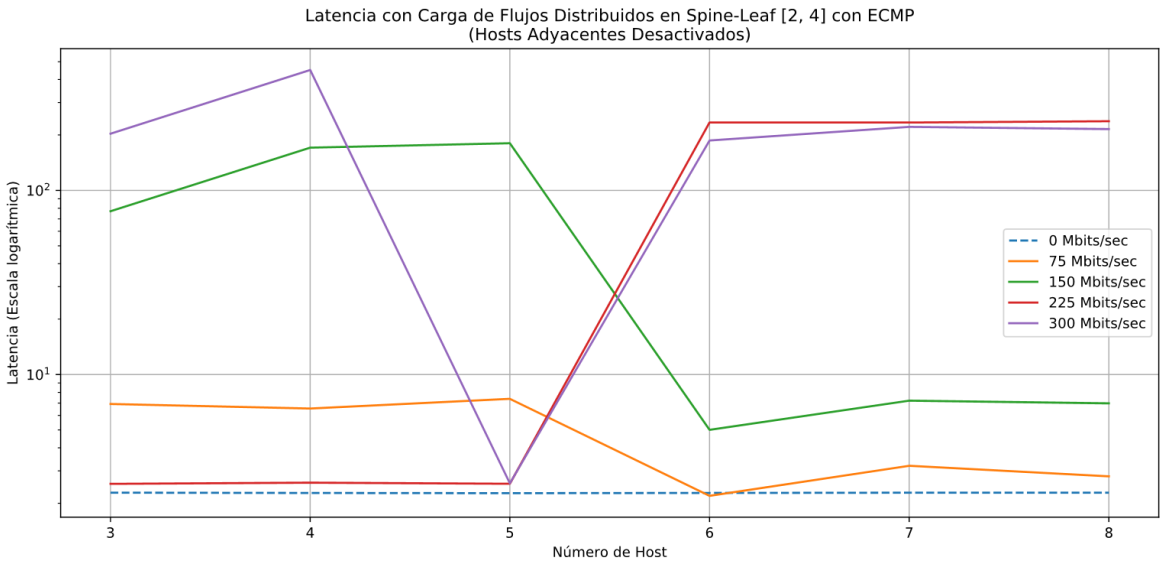


Figura 103: Latencia bajo carga en Spine-Leaf [2,4] con ECMP

Para intentar distribuir el tráfico de forma más uniforme sobre la red, utilizamos los flujos paralelos en la misma configuración, concretamente 24 flujos, Figura 104.

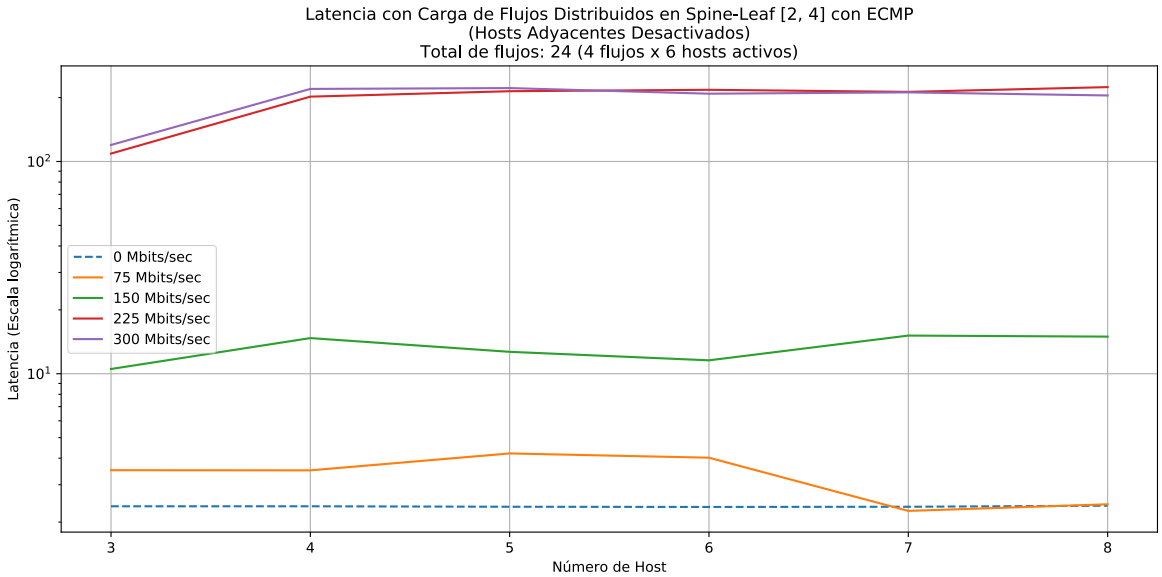


Figura 104: Latencia bajo carga en Spine-Leaf [2,4] con ECMP con 24 flujos

En la Figura 104, observamos un resultado más limpio, ya que la distribución del tráfico es más lineal sobre la red.

Vamos a aumentar el ancho de banda teórico máximo, con una configuración Spine-Leaf [4,4], que ahora seria de 400Mbits/s, la topología en cuestión seria la Figura 105.

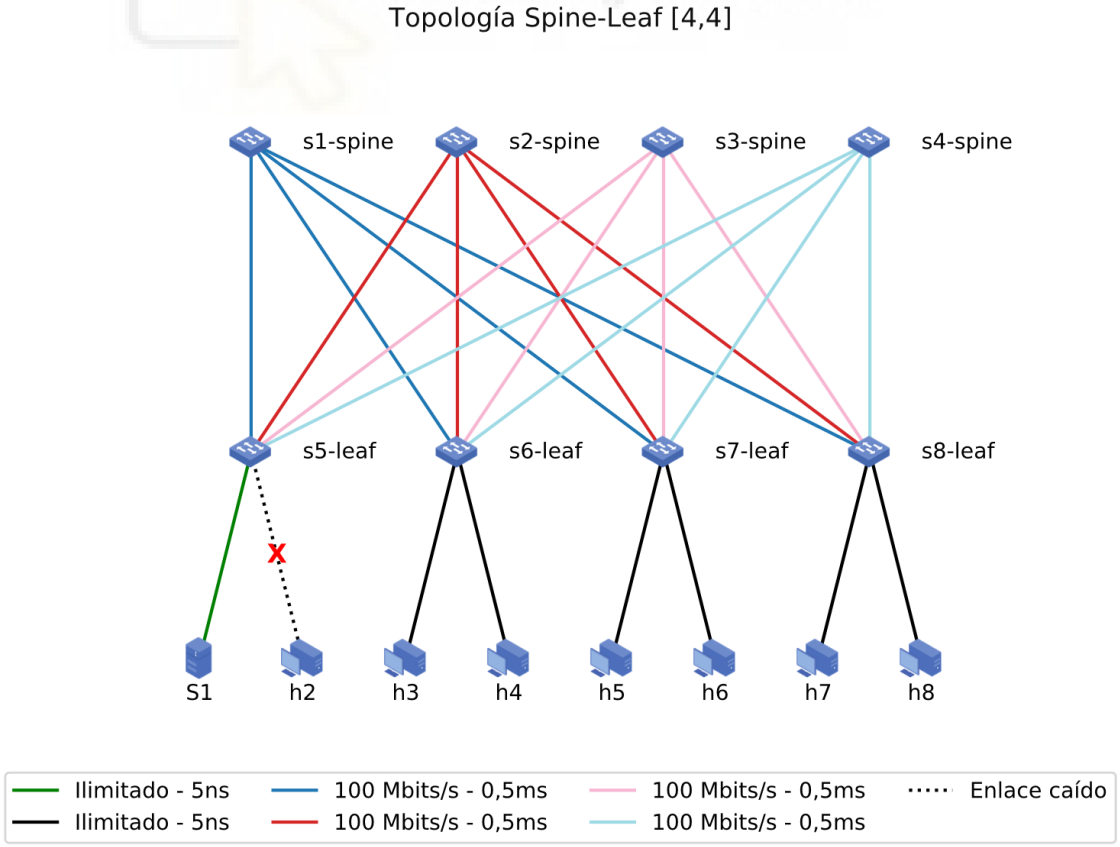


Figura 105: Topología Spine-Leaf [4,4] con host 8

El resultado de la configuración [4,4], la visualizamos en la Figura 106, se han utilizado 48 flujos, 8 flujos por host activo. Como el máximo teórico son 400 Mbits/s, al estar por debajo de ese límite, la latencia es cercano al caso de carga nula, pero al superar el límite, la red se satura.

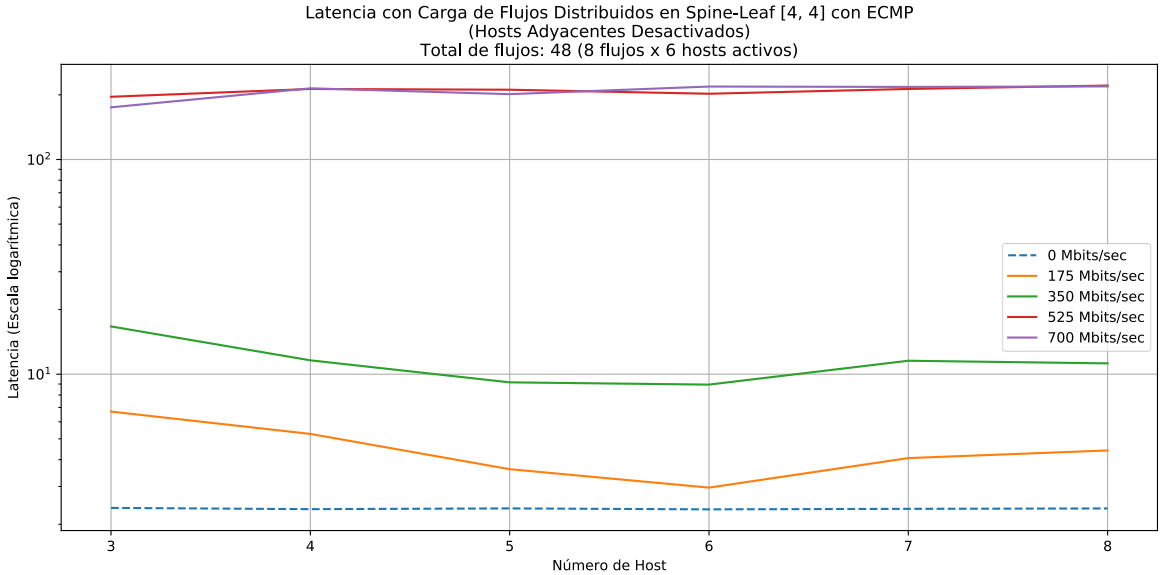


Figura 106: Latencia bajo carga en Spine-Leaf [4,4] con ECMP con 48 flujos

5.5.3 Conclusión

Con el algoritmo STP, no hay sorpresa, hasta llegar el límite de los 100Mbits/s de tráfico, la latencia es pequeña, pero una vez superada el límite, la red se satura. Como el algoritmo STP tumba los enlaces redundantes, para evitar bucles, se nos simplifica la topología, por consecuencia obtenemos unos resultados limpios ya que el tráfico no tiene alternativa al elegir camino, ya que solo existe uno.

En el algoritmo de ECMP, se complica todo, ya que disponemos de muchos caminos alternativos para cada host, los resultados que obtenemos con una cantidad pequeña de flujos no son limpios, debido que el algoritmo ECMP no distribuye los flujos de forma uniforme. Esto provoca que haya enlaces que estén saturados y otros no, de tal manera, que el Ping es capaz de evitar el tráfico en algunos de los casos, provocando graficas con resultados inestables.

Al aumentar el número de flujos en ECMP, obtenemos resultados más limpios, ya que el tráfico se distribuye de forma más uniforme.

El algoritmo ECMP es capaz de soportar mayores cargas de trabajo, comparándolo con el algoritmo STP.

6 CONCLUSIONES

En el camino de la realización de este TFG, he tenido la oportunidad de trabajar en un entorno Linux, conociendo sus partes buenas y malas, aprender el lenguaje de programación Python, y con ello explorar el mundo de la automatización.

El primer acercamiento fue la instalación del Lubuntu en una máquina virtual, para luego instalar el controlador ONOS, después de varios intentos, conseguí instalármelo y configurarlo correctamente.

Una vez obtenido el entorno de trabajo, se procedió con la creación de scripts de Python, para automatizar las distintas pruebas planteadas.

Se comenzó con la script de creación de la topología "Spine-Leaf", dicho script es capaz de crear la topología en el emulador Mininet-WiFi, con cualquier número de host, switch y numero de capas. Este script es el núcleo de todos los demás, ya que los demás de forma indirecta, lo llamaran.

Una vez hecha la topología "Spine-Leaf", se comenzó con la creación del script que media la latencia de los hosts en la topología. Una vez creado el script, se obtenía resultados inestables. Después de muchos intentos, la solución final fue instalar el Ubuntu en una partición del disco. En los enlaces se introdujo una latencia para poder visualizar la cantidad de enlaces que atravesaba los paquetes.

El próximo parámetro de testear fue el ancho de banda, para ello se creó un script que media el ancho de banda con único flujo. Para poder visualizar y comparar resultados se introdujeron límites de ancho de banda en los enlaces.

Sobre este momento, el controlador ONOS, dejo de funcionar por una actualización de Ubuntu, y poco después sufrió un ataque de ransomware, por ello se cambió de controlador a OpenDaylight.

Posteriormente cree un script que media el ancho de banda, pero con múltiples flujos, una vez hecho el script y corregido, los resultados que se obtenía no eran los esperados, el ancho de banda era menor del que se esperaba, se hizo la misma prueba manualmente, y se obtenía el mismo resultado. Investigando llegue a la conclusión que el controlador OpenDaylight utilizaba el algoritmo STP u similares, que bloqueaba los enlaces redundantes.

Para este problema opte por implementar al algoritmo ECMP, que permitía utilizar los enlaces redundantes, con ello se incrementó el ancho de banda.

Al hacer pruebas, se llegó a la conclusión de que el ancho de banda dependía de forma directa con el número de switches Spine, ha dicho propiedad se ha referido con la expresión "máximo teórico" en este proyecto.

Se implemento otro script que testeaba el ancho de banda con flujo distribuido, básicamente, enviar una prueba de ancho de banda desde todos los hosts hacia el servidor, en el mismo instante de tiempo.

Se descubrió que el algoritmo ECMP no funcionaba bien con pocos flujos. Esto debido que el algoritmo ECMP no es capaz de distribuir correctamente los flujos sobre los enlaces, ya

que lo hace de una forma aleatoria. Por ello, satura un conjunto de enlaces y deja otro conjunto con una carga menor, por ello los resultados se vuelven inestables, ya que en cada emulación determina los flujos a cada enlace distinto. Posterior se descubrió que, con una cantidad mayor de flujos, el factor aleatorio disminuía debido a la ley de los grandes números, que es un teorema fundamental de la teoría de la probabilidad.

Y por último se juntó las pruebas de latencia con el ancho de banda múltiple y distribuido, en el algoritmo STP se obtuvo lo esperado, pero en cambio con ECMP, los resultados eran muy inestables con pocos flujos, al aumentar los flujos, se solucionó. Ya que la herramienta Ping, era capaz de encontrar algunas veces, caminos con menos tráfico.

7 ANEXOS: CÓDIGO FUENTE

En este apartado, se proporcionará los códigos completos, para la posibilidad de probar las distintas emulaciones que se ha realizado en este proyecto.

- Si resulta que, al momento de copiar el código, se nos ha elimina las tabulaciones de los códigos, se puede utilizar un navegador web para abrir el archivo y copiar el código con el formado correcto.
- Se debe nombrar los scripts con el mismo nombre que indico, para evitar errores.
- Tener en cuenta que se debe tener las distintas librerías instaladas.
- En este trabajo mencionaremos y trabajaremos con "Mininet-WiFi", pero el "mininet" es totalmente funcional, con el único inconveniente que, en los scripts, se tendría que cambiar "mininet-wifi" por "mininet" mediante la herramienta de búsqueda (CTRL+F).
- Todos los scripts deben pertenecer en el mismo directorio.

7.1 CREACIÓN DE LA TOPOLOGÍA SPINE-LEAF

Este archivo se debe llamarse "spine_leaf.py".

```
import os
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
from mininet.link import TCLink

class MyTopo(Topo):
    def build(self):
        """
        Crea una topología con capas de switches y un número variable de hosts en la última
capa.
        """
        # Leer los valores de las variables de entorno
        capa_switch = list(map(int, os.getenv('CAPA_SWITCH', '4,4').split(',')))
        num_host = int(os.getenv('NUM_HOST', '8'))
        hosts_adyacentes = int(os.getenv('HOSTS_ADYACENTES', '0'))

        self.num_spine_switches=capa_switch[0] #Obtengo el numero de switchs spine
        self.num_leaf_switches=capa_switch[1] #Obtengo el numero de switchs leaf

        self.spine_ports_range= range(1, 1 + self.num_spine_switches)  #Puertos del switch
leaf, que conectan con switch spine
        self.leaf_ports_range = range(1, 1 + self.num_leaf_switches)

        #Puertos del switch leaf, que conectan con los hosts
        self.host_ports_range=range(self.num_spine_switches+1, 1 + self.num_spine_switches +
int(num_host/self.num_leaf_switches))

        # Variables para llevar la cuenta de switches y hosts
        contador_switch = 0
        contador_host = 0
        switch_final_capa_anterior = 0

        # Bucle para crear las capas de switches
        for i in range(len(capa_switch)):
            # Calcular el índice del último switch de la capa actual
            switch_final_capa_anterior = sum(capa_switch[:i])

            # Crear los switches de la capa actual
            for j in range(capa_switch[i]):
                # Comprueba si el switch es de la capa spine o leaf
```

```
                if i == 0:  # la capa spine es la primera capa
                    switch_nombre = 's%s-spine' % (contador_switch + 1)
                else:  #todas las demás capas son capas leaf
                    switch_nombre = 's%s-leaf' % (contador_switch + 1)
                self.addSwitch(switch_nombre)
                contador_switch += 1

                # Interconexiones entre switches (excepto en la primera capa)
                if i > 0:
                    for k in range(switch_final_capa_anterior - capa_switch[i - 1],
switch_final_capa_anterior):
                        self.addLink(switch_nombre, self.switches()[k], cls=TCLink, bw=100,
delay='0.5ms')

        # Bucle para crear los hosts en la última capa
        for switch in range(capa_switch[-1]):
            for host_switch in range(int(num_host / capa_switch[-1])):
                host_nombre = 'h%s' % (contador_host + 1)
                host = self.addHost(host_nombre)
                contador_host += 1

                # Enlaces entre hosts y switches
                if (switch == 0 and host_switch== 0) :
                    # El primer host se conectan sin límite de ancho de banda
                    self.addLink(host, self.switches()[-capa_switch[-1]+switch], cls=TCLink,
delay='0.005ms')

                elif(hosts_adyacentes==1 and host_switch!=0 and switch==0):
                    #Desactivamos los enlaces adyacentes de los host en el switch, a excepción
del h1
                    self.addLink(host, self.switches()[-capa_switch[-1]+switch], cls=TCLink,
bw=1, delay='0.005ms')

                else:
                    self.addLink(host, self.switches()[-capa_switch[-1]+switch], cls=TCLink,
delay='0.005ms')

def mytopo():
    return MyTopo()

topos = {'mytopo': mytopo}
```

7.2 CREACIÓN DE LA TOPOLOGÍA LEAF-SPINE CON CONTENEDORES

Este archivo se debe llamarse “Leaf_Spine_containernet.py”

```
from mininet.net import Containernet
from mininet.node import Controller, OVSKernelSwitch, RemoteController
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import info, setLogLevel


setLogLevel('info')
net = Containernet(controller=RemoteController, switch=OVSKernelSwitch) #remote controller

info('*** Adding controller\n')
net.addController('c0', controller=RemoteController, ip= '172.17.0.1', port= 6653)
#----------------------------------------------------------------------
s=2
l=2
n=4
#----------------------------------------------------------------------
contador_host=0
switch_Spine=[]
switch_leaf=[]
host=[]
#------------------------------capa Spine------------------------------
info('*** Adding switches\n')
for i in range(s):# Creo todos los Switch Spine
    switch_Spine.append(net.addSwitch('s%s-Spine' % (i + 1), protocols= "OpenFlow13"))
#------------------------------capa leaf-------------------------------
for j in range(l):# Creo todos los Switch leaf
    switch_leaf.append(net.addSwitch('s%s-leaf' % (j + s + 1), protocols= "OpenFlow13"))
#Almaceno en la lista leaf
    for k in range(s):#Recorro todos los switch spine
          net.addLink( switch_leaf[j],switch_Spine[k])#Conecto todos los switch spine con el
switch leaf

    #------------------------------capa host/servidores-----------------------
    info('*** Adding docker containers\n')
    for h in range(int(n/l)):#Dividimos los host por switch leaf
        d = net.addDocker('d%s' % (contador_host + 1), ip='10.0.0.%s' % (contador_host + 1),
dimage="ubuntu:trusty")
        host.append(d)
        info('*** Creating links\n')
        net.addLink(host[contador_host], switch_leaf[j])
        contador_host+=1

#----------------------------------------------------------------------
info('*** Starting network\n')
net.start()
info('*** Testing connectivity\n')
#net.ping([host[0], host[1],])
#net.iperf([host[0],host[1]])
info('*** Running CLI\n')
CLI(net)
info('*** Stopping network')
net.stop()
```

7.3 ALGORITMO ECMP

Este archivo se debe llamarse “ECMP.py”.

```
import os
import complementos

def Definicion_flujos(capa_switch,num_host):

    num_spine_switches = capa_switch[0]  # Obtengo el número de switches spine
    num_leaf_switches = capa_switch[1]  # Obtengo el número de switches leaf

    leaf_ports_range = range(1, 1 + num_spine_switches)  # Puertos del switch leaf, que
conectan con switch spine
    spine_ports_range = range(1, 1 + num_leaf_switches)

    # Puertos del switch leaf, que conectan con los hosts
    host_ports_range = range(num_spine_switches + 1, 1 + num_spine_switches + int(num_host /
num_leaf_switches))

    """
    Instalar entradas de flujo proactivas para los switches
    """

    switch_leaf_num_port_host = num_host / num_leaf_switches  # Número de hosts que toca por
switch leaf
    ip = 0

    # Switches Spine
    for sw in range(1, 1 + num_spine_switches):
        print(f"Configurando switch spine: {sw}")
        ip = 0
        # Añade flujos para cada switch spine
        for puerto in spine_ports_range:  # Recorro todos los puertos
            for i in range(int(switch_leaf_num_port_host)):  # Asigno todas las IP
correspondientes por puerto
                ip += 1
                for tipo_paquete in ['arp', 'ip']:
                    cmd = f"ovs-ofctl add-flow s{sw}-spine -O OpenFlow13
'table=0,priority=20,{tipo_paquete},nw_dst=10.0.0.{ip},actions=output:{puerto}'"
                    print(cmd)
                    os.system(cmd)

    ip = 0
    # Switches Leaf
    for sw in range(1 + num_spine_switches, num_leaf_switches + num_spine_switches + 1):
        print(f"Configurando switch leaf: {sw}")

        # Añade un grupo para cada switch leaf
        buckets = ','.join([f"bucket=output:{puerto}" for puerto in leaf_ports_range])
        cmd = f"ovs-ofctl add-group s{sw}-leaf -O OpenFlow13
'group_id=50,type=select,{buckets}'"
        print(cmd)
        os.system(cmd)

        # Añade flujos para cada switch leaf
        for puerto in host_ports_range:
            for tipo_paquete in ['arp', 'ip']:
                cmd = f"ovs-ofctl add-flow s{sw}-leaf -O OpenFlow13
'table=0,priority=10,{tipo_paquete},in_port={puerto},actions=group:50'"
                print(cmd)
                os.system(cmd)

        # Añade flujos específicos para cada dirección IP de destino
        for puerto in host_ports_range:
            ip += 1
            for tipo_paquete in ['arp', 'ip']:
                cmd = f"ovs-ofctl add-flow s{sw}-leaf -O OpenFlow13
'table=0,priority=20,{tipo_paquete},nw_dst=10.0.0.{ip},actions=output:{puerto}'"
                print(cmd)
```

```
                os.system(cmd)


def main():
    complementos.logo()
    # Leer los valores de las variables de entorno
    capa_switch = list(map(int, os.getenv('CAPA_SWITCH', '4,4').split(',')))
    num_host = int(os.getenv('NUM_HOST', '8'))
    use_ecmp = os.getenv('USE_ECMP', '0') == '1' #busca una variable de entorno llamada
'USE_ECMP',Si la variable existe, devuelve su valor. Si la variable no existe, devuelve '0'
    if not use_ecmp:
        print("ECMP no está activado. Saliendo...")
        return
    print("ECMP está activado. Configurando...")

    Definicion_flujos(capa_switch,num_host)



if __name__ == "__main__":
    os.environ['USE_ECMP'] = '1'# Definimos la variable de entorno
    main()
```

7.4 VISUALIZACIÓN DE LA TOPOLOGÍA

Este archivo se debe llamarse “visualizador_topologia.py”.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import matplotlib.lines as mlines
import time

def Dibujar_topologia(num_spine, num_leaf, num_host, hosts_adyacentes=False, show=True,
save=True):

    # Rutas de los iconos https://www.svgrepo.com/collection/servers-isometric-icons/
    #Descargar en formado png
    host_icono_ruta= "pc.png" #https://www.svgrepo.com/svg/474392/pc
    switch_icono_ruta = "switch.png" #https://www.svgrepo.com/svg/474396/switch
    server_icono_ruta = "server.png" #https://www.svgrepo.com/svg/474394/server

    # Crear la figura y el eje
    fig, ax = plt.subplots(figsize=(14, 6))

    # Cargar imágenes
    icono_host = mpimg.imread(host_icono_ruta)
    icono_switch = mpimg.imread(switch_icono_ruta)
    icono_server = mpimg.imread(server_icono_ruta)
    # Espacio entre nodos
    espacio_maximo = max(num_spine, num_leaf)
    espacio_spine = 2
    espacio_leaf = espacio_spine * espacio_maximo / num_leaf
    espacio_host = 2

    if num_host >= 4:  # Ajustar espaciado si hay muchos hosts
        espacio_spine = 6
        espacio_leaf = espacio_spine * espacio_maximo / num_leaf
        espacio_host = 6
    elif (num_spine or num_leaf) >14:
        espacio_spine = 2.5
        espacio_leaf = espacio_spine * espacio_maximo / num_leaf
        espacio_host = 2.5

    # Calcular el ancho total de la topología basado en la capa más ancha
    ancho_total = max(num_spine * espacio_spine, num_leaf * espacio_leaf)

    # Centrando las capas
```

```
    desplazamiento_spine = (ancho_total - (num_spine - 1) * espacio_spine) / 2
    desplazamiento_leaf = (ancho_total - (num_leaf - 1) * espacio_leaf) / 2

    # Coordenadas en Y para spine, leaf y host
    y_spine = 6
    y_leaf = 3
    y_host = 1
    if (num_spine or num_leaf) >14:
        y_spine = 9
        y_leaf = 3
        y_host = 1

    # Tamaño de los iconos
    size_icono = 0.4

    # Colores para los enlaces de Spine
    colores_spine = plt.get_cmap('tab20', num_spine)

    # Dibujar conexiones entre spines y leaves
    for i in range(num_spine):
        x_spine = i * espacio_spine + desplazamiento_spine
        color = colores_spine(i)
        for j in range(num_leaf):
            x_leaf = j * espacio_leaf + desplazamiento_leaf
            ax.plot([x_spine, x_leaf], [y_spine, y_leaf], color=color, zorder=1)

    # Dibujar conexiones host-leaf
    indice_host = 1
    for i in range(num_leaf):
        x_leaf = i * espacio_leaf + desplazamiento_leaf
        for j in range(num_host):
            x_host = x_leaf + (j - (num_host - 1) / 2) * espacio_host / num_host
            color = 'green' if indice_host == 1 else 'black'
            link = ax.plot([x_leaf, x_host], [y_leaf, y_host], color=color, zorder=1)[0]

            # Marcar enlaces caídos si hosts_adyacentes es True
            if hosts_adyacentes and i == 0 and j > 0:
                mid_x = (x_leaf + x_host) / 2
                mid_y = (y_leaf + y_host) / 2
                ax.text(mid_x, mid_y, 'X', color='red', fontsize=12, ha='center', va='center',
fontweight='bold')
                link.set_linestyle(':')  # Línea  para enlaces caídos

            indice_host += 1

    # Dibujar nodos spine
    for i in range(num_spine):
        x_spine = i * espacio_spine + desplazamiento_spine
        ax.imshow(icono_switch, extent=(x_spine - size_icono/2, x_spine + size_icono/2,
y_spine - size_icono/2, y_spine + size_icono/2), zorder=2)
        ax.text(x_spine + 0.5, y_spine, f's{i+1}-spine', va='center', ha='left', zorder=3)

    # Dibujar nodos leaf
    for i in range(num_leaf):
        x_leaf = i * espacio_leaf + desplazamiento_leaf
        ax.imshow(icono_switch, extent=(x_leaf - size_icono/2, x_leaf + size_icono/2, y_leaf -
size_icono/2, y_leaf + size_icono/2), zorder=2)
        ax.text(x_leaf + 0.5, y_leaf, f's{i+1+num_spine}-leaf', va='center', ha='left',
zorder=3)

    # Dibujar nodos host
    indice_host = 1
    for i in range(num_leaf):
        x_leaf = i * espacio_leaf + desplazamiento_leaf
        for j in range(num_host):
            x_host = x_leaf + (j - (num_host - 1) / 2) * espacio_host / num_host
            if j == 0 and i == 0:  # Para el servidor
                ax.imshow(icono_server, extent=(x_host - size_icono/2, x_host + size_icono/2,
y_host - size_icono/2, y_host + size_icono/2), zorder=2)
                ax.text(x_host, y_host - 0.4, f'S{indice_host}', ha='center', zorder=3)
            else:  # Para los demás hosts
```

```
                ax.imshow(icono_host, extent=(x_host - size_icono/2, x_host + size_icono/2,
y_host - size_icono/2, y_host + size_icono/2), zorder=2)
                ax.text(x_host, y_host - 0.4, f'h{indice_host}', ha='center', zorder=3)
            indice_host += 1

    # Añadir la leyenda
    etiqueta_ancho_banda = "Ilimitado"
    leyendas = [
        mlines.Line2D([], [], color='green', label='Ilimitado - 5ns'),
        mlines.Line2D([], [], color='black', label=f'{etiqueta_ancho_banda} - 5ns')
    ]


    for i in range(num_spine):
        color = colores_spine(i)
        leyendas.append(mlines.Line2D([], [], color=color, label='100 Mbits/s - 0,5ms'))
    if hosts_adyacentes:
        leyendas.append(mlines.Line2D([], [], color='black', linestyle=':', label='Enlace
caído'))

    ax.legend(handles=leyendas, loc='upper center', bbox_to_anchor=(0.5, 0), ncol=4)

    # Título del gráfico
    titulo = f"Topología Spine-Leaf [{num_spine},{num_leaf}]"
    ax.set_title(titulo)

    # Ajustar los límites del gráfico
    ax.set_xlim(-1, ancho_total + 1)
    ax.set_ylim(0, 7)
    if (num_spine or num_leaf) >14:
        ax.set_ylim(0, 10)

    # Ocultar ejes
    ax.axis('off')

    # Mostrar y/o guardar gráfico
    if show:
        plt.show(block=False)

    if save:
        timestamp = time.strftime("%d-%m-%Y-%H%M%S")
        nombre_archivo = f"Visualizacion_topologia [{num_spine},{num_leaf}]_{timestamp}"
        if hosts_adyacentes:
            nombre_archivo += "_hosts_adyacentes"
        fig.savefig(f"{nombre_archivo}.svg", format='svg', bbox_inches='tight',
pad_inches=0.2, dpi=600)

if __name__ == "__main__":
    # Configuración del ejemplo
    num_spine = 6
    num_leaf = 4
    num_host = 2
    hosts_adyacentes = True

    # Llamar a la función para dibujar la topología
    Dibujar_topologia(num_spine, num_leaf, num_host, hosts_adyacentes, show=True)

    plt.show(block=True)
```

7.5 COMPLEMENTOS

Este archivo se debe llamarse “complementos.py”.

```
import argparse
import os
import sys

def parametros():
    #Argumentos comunes en todos los scripts
    parser = argparse.ArgumentParser(description='Ejecutar la emulacion de red Spine-Leaf',
add_help=False)

    parser.add_argument('--s', '--capa_switch', type=str, default='2,2',
                        help='Capas de switches separadas por comas (por defecto: 2,2)')

    parser.add_argument('--h', '--n_host', type=int, default=10,
                        help='Número de hosts (por defecto: 10)')

    parser.add_argument('--e', '--ECMP', action='store_true',
                        help='Activar algoritmo ECMP')

    parser.add_argument('--g', '--graficar', action='store_true',
                        help='Visualización de la topología')

    parser.add_argument('--a', '--hosts_adyacentes', action='store_true',
                        help='Desactivar hosts adyacentes del switch que se encuentra h1')

    # Obtener el nombre del script que está llamando a esta función
    nombre_script = os.path.basename(sys.argv[0])

    #añadimos argumentos adicionales si el script es:
    if nombre_script != 'latencia.py':
        parser.add_argument('--t', '--tiempo_prueba', type=int, default=5,
                            help='Tiempo de prueba en segundos (por defecto: 5)')

        parser.add_argument('--c', '--carga_trafico_max', type=int, default=1000,
                            help='Carga de tráfico máxima en Mbps (por defecto: 1000)')

        parser.add_argument('--p', '--pasos', type=int, default=4,
                    help='Número de iteraciones del bucle, mientras sea mayor, se obtendrá
mayor precisión (por defecto: 4)')

        if nombre_script == 'ancho_banda_flujo_multiple.py':
            parser.add_argument('--P', '--flujos_paralelos', type=int, default=10,
                        help='Número de flujos paralelos producidos (por defecto: 10)')

    parser.add_argument('--help', action='help', default=argparse.SUPPRESS,
                    help='Muestra este mensaje de ayuda')

    args = parser.parse_args() # Almacena los argumentos en la variable 'args'

    capa_switch = list(map(int, args.s.split(','))) # Convierte la cadena de números separados
por comas en una lista de enteros
    n_host = args.h
    use_ecmp = args.e
    representar_topologia=args.g
    hosts_adyacentes=args.a
    resultado = [capa_switch, n_host, use_ecmp,representar_topologia,hosts_adyacentes]

    if nombre_script == 'latencia.py':
        # No añadir valores adicionales, solo se devolverán los valores básicos
        pass
    elif nombre_script == 'ancho_banda_flujo_multiple.py':
        resultado.extend([args.t, args.c, args.p, args.P])
    else:
        resultado.extend([args.t, args.c, args.p])

    return tuple(resultado) # Una tupla es una colección de elementos inmutables
```

```
def parametros_latencia_bajo_carga():
    #Argumentos comunes en todos los scripts
    parser = argparse.ArgumentParser(description='Ejecutar simulación de red Spine-Leaf',
add_help=False)

    parser.add_argument('--s', '--capa_switch', type=str, default='4,4',
                        help='Capas de switches separadas por comas (por defecto: 4,4)')

    parser.add_argument('--h', '--n_host', type=int, default=8,
                        help='Número de hosts (por defecto: 8)')

    parser.add_argument('--e', '--ECMP', action='store_true',
                        help='Activar algoritmo ECMP')

    parser.add_argument('--g', '--graficar', action='store_true',
                    help='Visualización de la topología')

    parser.add_argument('--a', '--hosts_adyacentes', action='store_true',
                    help='Desactivar hosts adyacentes del switch que se encuentra h1')

    parser.add_argument('--c', '--carga_trafico_max', type=int, default=600,
                        help='Carga de tráfico máxima en Mbps (por defecto: 600)')

    parser.add_argument('--p', '--pasos', type=int, default=4,
                    help='Número de iteraciones del bucle, mientras sea mayor, se obtendrá
mayor precisión (por defecto: 4)')


    parser.add_argument('--i', '--ping', type=int, default=10,
                        help='Numero de iteraciones del ping (por defecto: 10)')


    parser.add_argument('--help', action='help', default=argparse.SUPPRESS,
                    help='Muestra este mensaje de ayuda')

    parser.add_argument('--P', '--flujos_paralelos', type=int, default=1,
                        help='Número de flujos paralelos producidos (por defecto: 1)')


    args = parser.parse_args() # Almacena los argumentos en la variable 'args'

    capa_switch = list(map(int, args.s.split(','))) # Convierte la cadena de números separados
por comas en una lista de enteros
    n_host = args.h
    use_ecmp = args.e
    representar_topologia=args.g
    hosts_adyacentes=args.a
    carga_maxima=args.c
    num_iteraciones=args.p
    n_ping=args.i
    flujos_paralelos=args.P
    resultado = [capa_switch, n_host,
use_ecmp,representar_topologia,hosts_adyacentes,carga_maxima,num_iteraciones,n_ping,flujos_par
alelos]

    return tuple(resultado) # Una tupla es una colección de elementos inmutables

def parametros_N_configuraciones():
    parser = argparse.ArgumentParser(description='Ejecutar simulación de red Spine-Leaf',
add_help=False)

    parser.add_argument('--s', '--capa_switch', type=str, nargs=4, default=['2,4', '2,8',
'4,4', '6,4'],
                            help='Vector de 4 configuraciones de capas de switches (por
defecto: 2,4 2,8 4,4 6,4)')

    parser.add_argument('--h', '--n_host', type=int, default=8,
                        help='Número de hosts (por defecto: 8)')

    parser.add_argument('--e', '--ECMP', action='store_true',
                        help='Activar algoritmo ECMP')
```

```
    parser.add_argument('--g', '--graficar', action='store_true',
                help='Visualización de la topología')

    parser.add_argument('--a', '--hosts_adyacentes', action='store_true',
                    help='Desactivar hosts adyacentes del switch que se encuentra h1')

    parser.add_argument('--t', '--tiempo_prueba', type=int, default=5,
                        help='Tiempo de prueba en segundos (por defecto: 5)')

    parser.add_argument('--c', '--carga_trafico_max', type=int, default=1000,
                        help='Carga de tráfico máxima en Mbps (por defecto: 1000)')

    parser.add_argument('--p', '--pasos', type=int, default=4,
                    help='Número de iteraciones del bucle, mientras sea mayor, se obtendrá
mayor precisión (por defecto: 4)')

    nombre_script = os.path.basename(sys.argv[0])

    if nombre_script == 'ancho_banda_flujo_multiple_N_configuraciones.py':
        parser.add_argument('--P', '--flujos_paralelos', type=int, default=10,
                    help='Número de flujos paralelos producidos (por defecto: 10)')

    parser.add_argument('--help', action='help', default=argparse.SUPPRESS,
                    help='Muestra este mensaje de ayuda')
    args = parser.parse_args()

    capa_switch = [list(map(int, x.split(','))) for x in args.s] #
    #capa_switch = [[int(y) for y in x.split(',')] for x in args.s]
    n_host = args.h
    use_ecmp = args.e
    graficar=args.g
    hosts_adyacentes=args.a
    resultado = [capa_switch, n_host, use_ecmp, graficar, hosts_adyacentes, args.t, args.c,
args.p]

    if nombre_script == 'ancho_banda_flujo_multiple_N_configuraciones.py':
        resultado.extend([args.P])
    return resultado

""" Generador de arte ASCII:
https://www.ascii-art-generator.org/es.html
https://es.rakko.tools/tools/68/
https://manytools.org/hacker-tools/convert-images-to-ascii-art/go/
"""
def logo():
    print("\033[1;31m")# Cambia el color del texto a rojo
    print("""
    **********************************************************************
    ****************@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@**************
    ****************@@@@@@@@@@@@@@@@@@@@@@@@@@**@@@@@@@@@@@@**************
    ****************@@@@@@@@@@@@@@@@@@@@@@@@@***@@@@@@@@@@@@**************
    ****************@@@@@@@@@@@@@@@@@@@@@@@@@**@@@@@@@@@**@@**************
    ****************@@@****@@@@@@@@@@@@@@@@@@**@@@@@@@@#**@@**************
    ****************@@**@/**@@@@@@@@@@@@@@@@***@@@@@@@@**@@@**************
    ****************@**@@@**%@@@@@@@@@@@@@@@***@@@@@@@@**@@@**************
    ******************@@@@***@@@@@@@@@@@@@@@**@@@@@@@@**@@@@**************
    ***************@*%@@@@**#@@@@*****@@@@@@**@@@@@@@@**@@@@**************
    **************@@(@@@@@**@@@**@@@**@@@@@***@@@@@@@***@@@@**************
    *************@@@@@@@@@**@@**@@@@**(@@@@**&@@@@@@@**@@@@@**************
    ************%@&*@@@@@**@**@@@@@@**@@@@(*****************@@(***********
    ************@@**@@@@@****@@@@@@@**@@@@***@@@@@@@**@@@@@@**************
    ***********@@***@@@@@@*@@@@@@@@%**@@@***@@@@@@@***@@@@@@**************
    **********@@****@@@@@@@@@@@@@@@**@@@@***@@@@@@#**&@@@@@@**************
    ****************@@@@@@@@@@@@@@@@@@@@***@@@@@@@@*%@@@@@@@**************
    ***********************************&@@********************************
    ***********************************@@*********************************
    **********************************************************************
            """)
    print("\033[0m")  # Restablece el color del texto
```

7.6 LATENCIA

Este archivo se debe llamarse “latencia.py”.

```
import pexpect
import csv
import sys
import subprocess
import time
import os

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

import complementos
import visualizador_topologia

def Definicion_Topologia(capa_switch, n_host,hosts_adyacentes):
    # Definir las variables de entorno en Python
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch)) # La función map aplica una
función a todos los elementos de una lista. En este caso, convierte todos los elementos a str
    #La función join de una cadena de texto toma una lista de cadenas de texto y las une en
una sola cadena, utilizando la cadena original como separador.
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'

    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    # Ejecutar el comando
    return ejecutar_comando(comando, 'mininet-wifi>')

def ejecutar_comando(comando, expect):
    child = pexpect.spawn(comando)
    child.logfile_read = sys.stdout.buffer #Leemos el buffet, para poder visualizar la
inicializacion de mininet
    child.expect(expect,timeout=500)
    return child

def pings(child, n_host,hosts_adyacentes_count):
    # Crear el directorio 'log' si no existe
    if not os.path.exists('log'):
        os.makedirs('log')

    for i in range(hosts_adyacentes_count + 1, n_host):
        child.logfile_read = sys.stdout.buffer # Configuramos la salida del proceso hijo
        child.sendline(f'h{i + 1} ping h1 -c 10 -i 0,2')# Enviamos el comando de ping al
mininet
        child.expect('mininet-wifi>', timeout=500) # Esperamos la respuesta del comando

        output = child.before.decode('utf-8')# Decodificamos la salida y cerramos el buffer
        sys.stdout.buffer.close

        pings = output.split('\r\n')[2:-2] #separo cada ping en subcadenas, los 2 primeros y
ultimos ping los ingnoro

        # Guardamos los resultados en un archivo CSV en el directorio 'log'
        ruta_csv = os.path.join('log', f'host1_ping_host{i+1}.csv')
        with open(ruta_csv, 'w') as archivo_csv:
            writer = csv.writer(archivo_csv)
            writer.writerow(['pings', 'time'])
            for ping in pings:
                # Extraemos el tiempo de ping si está disponible
                if len(ping.split('=')[-1]) > 1:
                    tiempo_ping = ping.split('=')[-1].split()[0]
                    writer.writerow([ping, tiempo_ping])

def graficar_pings(n_host,hosts_adyacentes_count, capa_switch,use_ecmp):
    lista_pings = pd.Series()
    for i in range(hosts_adyacentes_count + 1, n_host):
```

```
        # Leer el archivo CSV desde el directorio 'log'
        ping = pd.read_csv(os.path.join('log', f'host1_ping_host{i+1}.csv'))
        ping["time"] = pd.to_numeric(ping["time"], errors="coerce")
        ping_media = np.mean(ping["time"][2:10]) # media
        lista_pings[f"h{i + 1}"] = ping_media

    plt.figure(figsize=(14, 6))
    plt.plot(lista_pings)
    plt.title(f"Latencia {capa_switch}")
    if use_ecmp:
        plt.title(f"Prueba de latencia en Spine-Leaf {capa_switch} con ECMP")
    else:
        plt.title(f"Prueba de latencia en Spine-Leaf {capa_switch}")
    plt.ylabel("(ms)")
    plt.grid(True)
    timestamp = time.strftime("%d-%m-%Y-%H%M%S")# Obtiene el tiempo actual y lo formatea como
una cadena

    # Guardar la figura en el directorio 'log'
    plt.savefig(f"Latencia_{capa_switch}_{timestamp}.svg", format='svg', dpi=300,
bbox_inches='tight')
    plt.show()

def calcular_hosts_adyacentes(n_host, capa_switch):
    hosts_por_leaf = n_host // capa_switch[1]
    return hosts_por_leaf - 1  # Excluyendo el primer host

def ECMP(capa_switch, n_host):
    print("Ejecutando algoritmo ECMP...")
    ecmp_env = os.environ.copy()  # Copia el entorno de variables de entorno actual
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch)) # Añade la variable CAPA_SWITCH
al entorno nuevo
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True) # Ejecuta
el script ECMP.py con el entorno modificado

def main():
    complementos.logo()
    capa_switch, n_host ,use_ecmp,representar_topologia,hosts_adyacentes=
complementos.parametros()#Obtengo los valores desde los parametros

    child=Definicion_Topologia(capa_switch,n_host,hosts_adyacentes)#Declaro las variables de
entorno y ejecuto la emulacion
    time.sleep(3) #pausamos 3 segundos, para que la topologia se conecte al controlador

    hosts_adyacentes_count = calcular_hosts_adyacentes(n_host, capa_switch) if
hosts_adyacentes else 0
    print(type(hosts_adyacentes_count), hosts_adyacentes_count)

    if use_ecmp:
        ECMP(capa_switch, n_host)

    if(hosts_adyacentes==0):#Para evitar que tarde mucho la prueba, porque hay host sin
conectividad
        child.sendline('pingall')
        child.expect('mininet-wifi>', timeout=500)

    pings(child, n_host,hosts_adyacentes_count)

    if representar_topologia:
        visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]),hosts_adyacentes)

    child.sendline('exit')
    subprocess.run('sudo mn -c', shell=True)
    graficar_pings(n_host,hosts_adyacentes_count,capa_switch,use_ecmp)

if __name__ == "__main__":
    main()
```

7.7 ANCHO DE BANDA CON FLUJO ÚNICO

Este archivo se debe llamarse “ancho_banda_flujo_unico.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")

# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp,hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'

    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado, posteriormente
permite interactuar con ejecutar_comando()
    child.logfile_read = sys.stdout.buffer  # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if(hosts_adyacentes==0):#Para evitar que tarde mucho la prueba, porque hay host sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts
    return child

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy() #copia de las variables de entorno actuales del sistema
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

# Función para generar tráfico de red usando iperf con flujo único
def generador_trafico(child, size_Mbits, tiempo_prueba, k, n_host):
    os.makedirs('log', exist_ok=True)  # Crea directorio para logs si no existe
    # Inicia servidor iperf en h1
    ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")
    # Inicia cliente iperf en el último host
    ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {size_Mbits}m -t {tiempo_prueba}
&>log/h{n_host}.log&')
    time.sleep(tiempo_prueba*1.5)  # Espera a que termine la prueba

# Función para obtener el ancho de banda de los logs de iperf
def obtener_ancho_banda(k):
    ancho_banda = 0
```

```
    with open(f'log/S{k}.log', 'r') as archivo:
        for linea in archivo:
            coincidencia = re.search(r'\d+(\.\d+)? Mbits/sec', linea)
            if coincidencia:
                ancho_banda = float(coincidencia.group().split()[0])
                break  # Tomamos solo el último valor
    return ancho_banda

# Función para graficar los resultados de las pruebas de ancho de banda
def graficar(lista_ancho_banda, capa_switch, use_ecmp):
    plt.figure(figsize=(14,6))
    plt.plot(lista_ancho_banda)

    plt.grid(True)
    plt.xlabel("Mbits/sec Enviados (Bandwidth)")
    plt.ylabel("Mbits/sec Recibidos (Bandwidth)")
    plt.ylim(0, 300)  # Establece el límite del eje Y a 300 Mbits/sec

    # Añade líneas horizontales para mejorar la legibilidad
    for y in range(0, 301, 50):
        plt.axhline(y=y, color='gray', linestyle='--', linewidth=0.5, alpha=0.7)

    titulo = f"Prueba de Ancho de banda con flujo único en Spine-Leaf {capa_switch}"
    if use_ecmp:
        titulo += " con ECMP"
    plt.title(titulo)
    plt.legend()

    # Ajusta los ticks del eje Y
    plt.yticks(range(0, 301, 50))

    timestamp = time.strftime("%d-%m-%Y-%H%M%S")

    plt.savefig(f"Ancho_banda_flujo_unico{str(capa_switch)}_{timestamp}.svg", format='svg',
dpi=300, bbox_inches='tight')
    plt.show()

def main():
    lista_ancho_banda = pd.Series([])
    complementos.logo()  # Muestra el logo

    # Obtiene los parámetros de la prueba
    capa_switch, n_host, use_ecmp, representar_topologia, hosts_adyacentes, tiempo_prueba,
carga_trafico_max, num_iteraciones = complementos.parametros()

    # Define y configura la topología de red
    child = Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes)
    paso = carga_trafico_max // (num_iteraciones - 1)

    # Realiza las pruebas de ancho de banda
    for k, trafico in enumerate(range(1, carga_trafico_max + num_iteraciones, paso), 1): #La
función enumerate() devuelve dos valores
        # en cada iteración, el indice(1,2,3...) y el valor del elemento
        generador_trafico(child, trafico, tiempo_prueba, k, n_host)
        ancho_banda = obtener_ancho_banda(k)
        print("\033[1;31m")
        print(f"Ancho de banda obtenido: {ancho_banda} Mbits/sec")
        print("\033[0m")
        lista_ancho_banda[int(trafico)] = ancho_banda

    # Representa la topología si se solicita
    if representar_topologia:
        visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]), hosts_adyacentes)

    # Cierra Mininet
    ejecutar_comando(child, 'exit')
    child.close()

    # Limpia cualquier proceso de Mininet que pudiera quedar
    subprocess.run('sudo mn -c', shell=True)
```

```
    # Grafica los resultados
    graficar(lista_ancho_banda, capa_switch, use_ecmp)

    print("El script se completó con éxito.")

if __name__ == "__main__":
    main()
```

7.8 ANCHO DE BANDA CON FLUJO ÚNICO N CONFIGURACIONES

Este archivo se debe llamarse “ancho_banda_flujo_unico_N_configuraciones.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")

# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp,hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'

    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado, posteriormente
permite interactuar con ejecutar_comando()
    child.logfile_read = sys.stdout.buffer  # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if(hosts_adyacentes==0):#Para evitar que tarde mucho la prueba, porque hay host sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts
    return child

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy()
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

# Función para generar tráfico de red usando iperf con flujo único
```

```
def generador_trafico(child, size_Mbits, tiempo_prueba, k, n_host):
    os.makedirs('log', exist_ok=True)  # Crea directorio para logs si no existe
    # Inicia servidor iperf en h1
    ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")
    # Inicia cliente iperf en el último host
    ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {size_Mbits}m -t {tiempo_prueba}
&>log/h{n_host}.log&')
    time.sleep(tiempo_prueba*1.5)  # Espera a que termine la prueba

# Función para obtener el ancho de banda de los logs de iperf
def obtener_ancho_banda(k):
    ancho_banda = 0
    with open(f'log/S{k}.log', 'r') as archivo:
        for linea in archivo:
            coincidencia = re.search(r'\d+(\.\d+)? Mbits/sec', linea)
            if coincidencia:
                ancho_banda = float(coincidencia.group().split()[0])
                break  # Tomamos solo el último valor
    return ancho_banda

# Función  para graficar los resultados de las pruebas de ancho de banda
def graficar(todas_listas_ancho_banda, todos_capas_switch, carga_trafico_max, use_ecmp,
hosts_adyacentes):
    fig, axs = plt.subplots(2, 2, figsize=(16,8))
    axs = axs.ravel()  # Convierte el array 2D de ejes en 1D para facilitar la iteración

    for i, (lista_ancho_banda, capa_switch) in enumerate(zip(todas_listas_ancho_banda,
todos_capas_switch)):
        ax = axs[i]
        ax.plot(lista_ancho_banda)

        ax.set_xlabel("Mbits/sec Enviados (Bandwidth)")
        ax.set_ylabel("Mbits/sec Recibidos (Bandwidth)")
        ax.set_ylim(0, carga_trafico_max)

        # Ajustar las líneas horizontales y ticks según carga_trafico_max
        distancia_lineas = max(50, carga_trafico_max // 6)  # Asegurar al menos 6 líneas
        for y in range(0, int(carga_trafico_max) + distancia_lineas, distancia_lineas):
            ax.axhline(y=y, color='gray', linestyle='--', linewidth=0.5, alpha=0.7)

        titulo = f"Spine-Leaf {capa_switch}"
        if use_ecmp:
            titulo += " con ECMP"
        if hosts_adyacentes:
            titulo += "\n(Hosts Adyacentes Desactivados)"
        ax.set_title(titulo)
        ax.grid(True)
        ax.set_yticks(range(0, int(carga_trafico_max) + distancia_lineas, distancia_lineas))

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    fig.suptitle("Comparativa de Ancho de Banda con Flujo Único para Múltiples
Configuraciones", fontsize=16, y=0.98)
    plt.subplots_adjust(hspace=0.3, wspace=0.3)

    timestamp = time.strftime("%d-%m-%Y-%H%M%S")
    plt.savefig(f"Ancho_banda_flujo_unico_comparativa_{timestamp}.svg", format='svg', dpi=300,
bbox_inches='tight')

    plt.show()


def main():
    complementos.logo()  # Muestra el logo

    # Obtenemos los valores de las variables para las múltiples configuraciones
    capas_switch, n_host, use_ecmp, representar_topologia,hosts_adyacentes, tiempo_prueba,
carga_trafico_max, num_iteraciones = complementos.parametros_N_configuraciones()

    todas_listas_ancho_banda = []

    # Iteramos sobre las configuraciones de capas_switch
```

```
    for i, capa_switch in enumerate(capas_switch):
        print(f"\nIniciando simulación {i+1} de {len(capas_switch)}")

        lista_ancho_banda = pd.Series([])

        # Define y configura la topología de red
        child = Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes)
        paso = carga_trafico_max // (num_iteraciones - 1)

        # Realiza las pruebas de ancho de banda
        for k, trafico in enumerate(range(1, carga_trafico_max + num_iteraciones, paso), 1):
            generador_trafico(child, trafico, tiempo_prueba, k, n_host)
            bandwidth = obtener_ancho_banda(k)
            print("\033[1;31m")
            print(f"Ancho de banda obtenido: {bandwidth} Mbits/sec")
            print("\033[0m")
            lista_ancho_banda[int(trafico)] = bandwidth

        # Representa la topología si se solicita
        if representar_topologia:
            visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]),hosts_adyacentes)
        # Cierra Mininet
        ejecutar_comando(child, 'exit')
        child.close()

        # Limpia cualquier proceso de Mininet que pudiera quedar
        subprocess.run('sudo mn -c', shell=True)

        todas_listas_ancho_banda.append(lista_ancho_banda)

        print(f"Simulación {i+1} completada.")

    # Grafica los resultados de todas las simulaciones juntas
    graficar(todas_listas_ancho_banda, capas_switch, carga_trafico_max,
use_ecmp,hosts_adyacentes)

    print("Todas las simulaciones se completaron con éxito.")

if __name__ == "__main__":
    main()
```

7.9 ANCHO DE BANDA CON FLUJO MÚLTIPLE

Este archivo se debe llamarse “ancho_banda_flujo_multiple.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")
```

```
# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'
    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado, posteriormente
permite interactuar con ejecutar_comando()
    child.logfile_read = sys.stdout.buffer # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if(hosts_adyacentes==0):  # Para evitar que tarde mucho la prueba, porque hay host sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts

    return child

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy() #copia de las variables de entorno actuales del sistema
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

# Función para generar tráfico de red usando iperf con flujos paralelos
def generador_trafico(child, size_Mbits, tiempo_prueba, k, n_host, flujos_paralelos):
    os.makedirs('log', exist_ok=True)  # Crea directorio para logs si no existe
    # Inicia servidor iperf en h1
    ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")

    # Calcula el ancho de banda por flujo
    size_Mbits_por_flujo = float(f"{size_Mbits / flujos_paralelos:.1f}")

    # Inicia cliente iperf en el último host con flujos paralelos
    if k==1:
        ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {1}m -t {tiempo_prueba} -P
{1} &>log/h{n_host}.log&') #Realizamos con el trafico minimo para graficar.
    else:
        ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {size_Mbits_por_flujo}m -t
{tiempo_prueba} -P {flujos_paralelos} &>log/h{n_host}.log&')

    time.sleep(tiempo_prueba*2)  # Espera a que termine la prueba

# Función para obtener el ancho de banda de los logs de iperf
def obtener_ancho_banda(k):
    ancho_banda = []
    with open(f'log/S{k}.log', 'r') as archivo:
        for linea in archivo:
            if "SUM" not in linea:  # Ignoramos líneas con "SUM"
                coincidencia = re.search(r'(\d+(?:\.\d+)?)\s*(M|K)bits/sec', linea)
                if coincidencia:
                    valor = float(coincidencia.group(1))
                    unidad = coincidencia.group(2)
                    # Convertimos Kbits a Mbits si es necesario
                    if unidad == 'M':
                        bandwidth = valor
                    else:
                        bandwidth = valor/1000

                    ancho_banda.append(bandwidth)
    return ancho_banda
```

```
# Función para graficar los resultados de las pruebas de ancho de banda
def graficar(lista_ancho_banda, capa_switch, carga_trafico_max, use_ecmp, flujos_paralelos):
    plt.figure(figsize=(14,6))
    plt.plot(lista_ancho_banda, label="Prueba de Rendimiento")
    x = range(int(carga_trafico_max))
    plt.plot(x, x, label="Ideal", linestyle='--')

    plt.grid(True)
    plt.xlabel("Mbits/sec Enviados (Bandwidth)")
    plt.ylabel("Mbits/sec Recibidos (Bandwidth)")

    titulo = f"Ancho de banda con {flujos_paralelos} flujos paralelos en Spine-Leaf
{capa_switch}"
    if use_ecmp:
        titulo += " con ECMP"
    plt.title(titulo)
    plt.legend()

    timestamp = time.strftime("%d-%m-%Y-%H%M%S")
    plt.savefig(f"Ancho_banda_{flujos_paralelos}_flujos_{str(capa_switch)}_{timestamp}.svg",
format='svg',  dpi=300, bbox_inches='tight')
    plt.show()

def main():
    lista_ancho_banda = pd.Series([])
    complementos.logo()  # Muestra el logo

    # Obtiene los parámetros de la prueba, incluyendo flujos_paralelos
    capa_switch, n_host, use_ecmp, representar_topologia, hosts_adyacentes, tiempo_prueba,
carga_trafico_max, num_iteraciones, flujos_paralelos = complementos.parametros()

    # Define y configura la topología de red
    child = Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes)
    paso = carga_trafico_max // (num_iteraciones - 1)

    # Realiza las pruebas de ancho de banda
    for k, trafico in enumerate(range(1, carga_trafico_max + num_iteraciones, paso), 1):
        print(f"\n\033[1;34m--- Iniciando prueba {k} con {flujos_paralelos} flujos de {trafico
/ flujos_paralelos:.1f} Mbits/sec  -- Trafico total: {trafico} Mbits/sec---\033[0m")
        generador_trafico(child, trafico, tiempo_prueba, k, n_host, flujos_paralelos)
        bandwidth = obtener_ancho_banda(k)
        print(f"\033[1;32mAncho de banda obtenido de los flujos: {bandwidth}
Mbits/sec\033[0m")
        lista_ancho_banda[int(trafico)] = sum(bandwidth)
        print(f"\033[1;32mAncho de banda total: {lista_ancho_banda[int(trafico)]:.2f}
Mbits/sec\033[0m")

    # Representa la topología si se solicita
    if representar_topologia:
        visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]), hosts_adyacentes)

    # Cierra Mininet
    ejecutar_comando(child, 'exit')
    child.close()

    # Limpia cualquier proceso de Mininet que pudiera quedar
    subprocess.run('sudo mn -c', shell=True)

    # Grafica los resultados
    graficar(lista_ancho_banda, capa_switch, carga_trafico_max, use_ecmp, flujos_paralelos)

    print("El script se completó con éxito.")

if __name__ == "__main__":
    main()
```

7.10 ANCHO DE BANDA CON FLUJO MÚLTIPLE EN VARIAS CONFIGURACIONES

Este archivo se debe llamarse “ancho_banda_flujo_multiple_N_configuraciones.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")

# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'
    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado
    child.logfile_read = sys.stdout.buffer   # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if not hosts_adyacentes:  # Para evitar que tarde mucho la prueba cuando hay hosts sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts
    return child

# Función  para generar tráfico de red usando iperf con flujos paralelos
def generador_trafico(child, size_Mbits, tiempo_prueba, k, n_host, flujos_paralelos):
    os.makedirs('log', exist_ok=True)  # Crea directorio para logs si no existe
    # Inicia servidor iperf en h1
    ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")

    # Calcula el ancho de banda por flujo
    size_Mbits_por_flujo = float(f"{size_Mbits / flujos_paralelos:.1f}")

    # Inicia cliente iperf en el último host con flujos paralelos
    if k==1:
        ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {1}m -t {tiempo_prueba} -P
{1} &>log/h{n_host}.log&') #Realizamos con el trafico minimo para graficar.
    else:
        ejecutar_comando(child, f'h{n_host} iperf -c 10.0.0.1 -b {size_Mbits_por_flujo}m -t
{tiempo_prueba} -P {flujos_paralelos} &>log/h{n_host}.log&')

    time.sleep(tiempo_prueba*2)  # Espera a que termine la prueba

# Función para obtener el ancho de banda de los logs de iperf
def obtener_ancho_banda(k):
    ancho_banda = []
    with open(f'log/S{k}.log', 'r') as archivo:
```

```
        for linea in archivo:
            if "SUM" not in linea:  # Ignoramos líneas con "SUM"
                coincidencia = re.search(r'(\d+(?:\.\d+)?)\s*(M|K)bits/sec', linea)
                if coincidencia:
                    valor = float(coincidencia.group(1))
                    unidad = coincidencia.group(2)
                    # Convertimos Kbits a Mbits si es necesario
                    if unidad == 'M':
                        bandwidth = valor
                    else:
                        bandwidth = valor/1000

                    ancho_banda.append(bandwidth)
    return ancho_banda

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy()
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'

    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

def graficar(todas_listas_ancho_banda, todos_capas_switch, carga_trafico_max, use_ecmp,
hosts_adyacentes,flujos_paralelos):
    fig, axs = plt.subplots(2, 2, figsize=(16, 8))
    axs = axs.ravel()  # Convierte el array 2D de ejes en 1D para facilitar la iteración

    for i, (lista_ancho_banda, capa_switch) in enumerate(zip(todas_listas_ancho_banda,
todos_capas_switch)):
        ax = axs[i]
        ax.plot(lista_ancho_banda, label="Prueba de Rendimiento")

        x = range(int(carga_trafico_max))
        ax.plot(x, x, label="Ideal", linestyle='--', color='red')

        ax.set_xlabel("Mbits/sec Enviados (Bandwidth)")
        ax.set_ylabel("Mbits/sec Recibidos (Bandwidth)")
        titulo = f"Ancho de banda con {flujos_paralelos} flujos paralelos en Spine-Leaf
{capa_switch}"
        if use_ecmp:
            titulo += " con ECMP"
        if hosts_adyacentes:
            titulo += "\n(Hosts Adyacentes Desactivados)"
        ax.set_title(titulo)
        ax.grid(True)
        ax.legend()

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Ajusta el layout para dejar espacio para el
título principal
    fig.suptitle("Comparativa de Ancho de Banda con flujo múltiple", fontsize=16, y=0.98)

    # Ajusta el espacio entre subplots
    plt.subplots_adjust(hspace=0.3, wspace=0.3)

    timestamp = time.strftime("%d-%m-%Y-%H%M%S")
    plt.savefig(f"Comparativa_ancho_banda_flujo_multiple_{timestamp}.svg", format='svg',
dpi=300, bbox_inches='tight')

    plt.show()
def main():
    complementos.logo()  # Muestra el logo

    # Obtenemos los valores de las variables
    capas_switch, n_host, use_ecmp, representar_topologia,hosts_adyacentes, tiempo_prueba,
carga_trafico_max, num_iteraciones,flujos_paralelos =
complementos.parametros_N_configuraciones()

    todas_listas_ancho_banda = []
```

```
    # Iteramos sobre las configuraciones de capas_switch
    for i, capa_switch in enumerate(capas_switch):
        print(f"\nIniciando simulación {i+1} de {len(capas_switch)}")

        lista_ancho_banda = pd.Series([])

        # Define y configura la topología de red
        child = Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes)
        paso = carga_trafico_max // (num_iteraciones - 1)

        # Realiza las pruebas de ancho de banda
        for k, trafico in enumerate(range(1, carga_trafico_max + num_iteraciones, paso), 1):
            print(f"\n\033[1;34m--- Iniciando prueba {k} con {flujos_paralelos} flujos de
{trafico / flujos_paralelos:.1f} Mbits/sec  -- Trafico total: {trafico} Mbits/sec---\033[0m")
            generador_trafico(child, trafico, tiempo_prueba, k, n_host, flujos_paralelos)
            bandwidth = obtener_ancho_banda(k)
            print(f"\033[1;32mAncho de banda obtenido de los flujos: {bandwidth}
Mbits/sec\033[0m")
            lista_ancho_banda[int(trafico)] = sum(bandwidth)
            print(f"\033[1;32mAncho de banda total: {lista_ancho_banda[int(trafico)]:.2f}
Mbits/sec\033[0m")

        # Representa la topología si se solicita
        if representar_topologia:
            visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]),hosts_adyacentes)

        # Cierra Mininet
        ejecutar_comando(child, 'exit')
        child.close()

        # Limpia cualquier proceso de Mininet que pudiera quedar
        subprocess.run('sudo mn -c', shell=True)

        todas_listas_ancho_banda.append(lista_ancho_banda)

        print(f"Simulación {i+1} completada.")

    # Grafica los resultados de todas las simulaciones en subplots
    graficar(todas_listas_ancho_banda, capas_switch, carga_trafico_max, use_ecmp,
hosts_adyacentes,flujos_paralelos)

    print("Todas las simulaciones se completaron con éxito.")

if __name__ == "__main__":
    main()
```

7.11 Ancho de Banda con Flujo Distribuido

Este archivo se debe llamarse “ancho_banda_flujo_distribuido.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
```

```
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")

# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp,hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'

    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado, posteriormente
permite interactuar con ejecutar_comando()
    child.logfile_read = sys.stdout.buffer  # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if(hosts_adyacentes==0):#Para evitar que tarde mucho la prueba, porque hay host sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts
    return child

# Función para generar tráfico de red usando iperf
def generador_trafico(child, carga_total, tiempo_prueba, k, n_host,hosts_adyacentes_count):
    os.makedirs('log', exist_ok=True)  # Crea directorio para logs si no existe
    hosts_activos = n_host - hosts_adyacentes_count - 1  # Excluyendo h1 y los hosts
adyacentes
    mbits_por_host = float(f"{carga_total / hosts_activos:.1f}")
    # Inicia servidor iperf en h1
    ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")
    # Inicia clientes iperf en los demás hosts
    for i in range(2+hosts_adyacentes_count, n_host+1):
        ejecutar_comando(child, f'h{i} iperf -c 10.0.0.1 -b {mbits_por_host}m -t
{tiempo_prueba} &>log/h{i}.log&')
    time.sleep(tiempo_prueba*2)  # Espera a que termine la prueba

# Función para obtener el ancho de banda de los logs de iperf
def obtener_ancho_banda(k):
    anchos_banda = []
    with open(f'log/S{k}.log', 'r') as file:
        for linea in file:
            if "SUM" not in linea:  # Ignoramos líneas con "SUM"
                coincidencia = re.search(r'(\d+(?:\.\d+)?)\s*(M|K)bits/sec', linea)
                if coincidencia:
                    valor = float(coincidencia.group(1))
                    unidad = coincidencia.group(2)
                    # Convertimos Kbits a Mbits si es necesario
                    if unidad == 'M':
                        bandwidth = valor
                    else:
                        bandwidth = valor/1000

                    anchos_banda.append(bandwidth)
    return anchos_banda

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy()
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

def calcular_hosts_adyacentes(n_host, capa_switch):
```

```
    hosts_por_leaf = n_host // capa_switch[1]
    return hosts_por_leaf - 1  # Excluyendo el primer host

# Función para graficar los resultados de las pruebas de ancho de banda
def graficar(lista_ancho_banda, capa_switch, carga_trafico_max, use_ecmp):
    plt.figure(figsize=(14,6))
    plt.plot(lista_ancho_banda, label="Prueba de Rendimiento")

    x = range(int(carga_trafico_max))
    plt.plot(x, x, label="Ideal", linestyle='--')

    plt.grid(True)
    plt.xlabel("Mbits/sec Enviados (Bandwidth)")
    plt.ylabel("Mbits/sec Recibidos (Bandwidth)")
    titulo = f"Prueba de Ancho de banda con flujo Distribuido en topología {capa_switch}"
    if use_ecmp:
        titulo += " con ECMP"
    plt.title(titulo)
    plt.legend()
    timestamp = time.strftime("%d-%m-%Y-%H%M%S")

    plt.savefig(f"Ancho_banda_flujo_distribuido{str(capa_switch)}_{timestamp}.svg",
format='svg',  dpi=300, bbox_inches='tight')
    plt.show()

def main():
    lista_ancho_banda = pd.Series([])
    complementos.logo()  # Muestra el logo

    # Obtiene los parámetros de la prueba
    capa_switch, n_host, use_ecmp, representar_topologia,hosts_adyacentes, tiempo_prueba,
carga_trafico_max, num_iteraciones = complementos.parametros()

    # Define y configura la topología de red
    child = Definicion_Topologia(capa_switch, n_host, use_ecmp,hosts_adyacentes)
    paso = carga_trafico_max // (num_iteraciones - 1)
    hosts_adyacentes_count = calcular_hosts_adyacentes(n_host, capa_switch) if
hosts_adyacentes else 0
    # Realiza las pruebas de ancho de banda
    for k, trafico in enumerate(range(1, carga_trafico_max + num_iteraciones, paso), 1):
        print(f"\n\033[1;34m--- Iniciando prueba {k} de {trafico} Mbits/sec ---\033[0m")
        generador_trafico(child, trafico, tiempo_prueba, k, n_host,hosts_adyacentes_count)
        anchos_banda = obtener_ancho_banda(k)
        print(f"\033[1;32mAncho de banda obtenido : {anchos_banda} Mbits/sec\033[0m")
        lista_ancho_banda[int(trafico)] = sum(anchos_banda)
        print(f"\033[1;32mAncho de banda total: {lista_ancho_banda[int(trafico)]:.2f}
Mbits/sec\033[0m")

    # Representa la topología si se solicita
    if representar_topologia:
        visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]),hosts_adyacentes)

    # Cierra Mininet
    ejecutar_comando(child, 'exit')
    child.close()

    # Limpia cualquier proceso de Mininet que pudiera quedar
    subprocess.run('sudo mn -c', shell=True)

    # Grafica los resultados
    graficar(lista_ancho_banda, capa_switch, carga_trafico_max, use_ecmp)

    print("El script se completó con éxito.")

if __name__ == "__main__":
    main()
```

7.12 ANCHO DE BANDA CON FLUJO DISTRIBUIDO EN VARIAS CONFIGURACIONES

Este archivo se debe llamarse “ancho_banda_flujo_distribuido_N_configuraciones.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")

# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'

    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado
    child.logfile_read = sys.stdout.buffer   # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if not hosts_adyacentes:  # Para evitar que tarde mucho la prueba cuando hay hosts sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts
    return child

# Función para generar tráfico de red usando iperf
def generador_trafico(child, carga_total, tiempo_prueba, k, n_host,hosts_adyacentes_count):
    os.makedirs('log', exist_ok=True)  # Crea directorio para logs si no existe
    hosts_activos = n_host - hosts_adyacentes_count - 1  # Excluyendo h1 y los hosts
adyacentes
    mbits_por_host = float(f"{carga_total / hosts_activos:.1f}")
    # Inicia servidor iperf en h1
    ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.5} &>log/S{k}.log&")
    # Inicia clientes iperf en los demás hosts
    for i in range(2+hosts_adyacentes_count, n_host+1):
        ejecutar_comando(child, f'h{i} iperf -c 10.0.0.1 -b {mbits_por_host}m -t
{tiempo_prueba} &>log/h{i}.log&')
    time.sleep(tiempo_prueba*2)  # Espera a que termine la prueba

# Función para obtener el ancho de banda de los logs de iperf
def obtener_ancho_banda(k):
    anchos_banda = []
    with open(f'log/S{k}.log', 'r') as file:
        for linea in file:
            if "SUM" not in linea:  # Ignoramos líneas con "SUM"
                coincidencia = re.search(r'(\d+(?:\.\d+)?)\s*(M|K)bits/sec', linea)
```

```
                if coincidencia:
                    valor = float(coincidencia.group(1))
                    unidad = coincidencia.group(2)
                    # Convertimos Kbits a Mbits si es necesario
                    if unidad == 'M':
                        bandwidth = valor
                    else:
                        bandwidth = valor/1000

                    anchos_banda.append(bandwidth)
    return anchos_banda

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy()
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

def calcular_hosts_adyacentes(n_host, capa_switch):
    hosts_por_leaf = n_host // capa_switch[1]
    return hosts_por_leaf - 1  # Excluyendo el primer host

def graficar(todas_listas_ancho_banda, todos_capas_switch, carga_trafico_max, use_ecmp,
hosts_adyacentes):
    fig, axs = plt.subplots(2, 2, figsize=(16, 8))
    axs = axs.ravel()  # Convierte el array 2D de ejes en 1D para facilitar la iteración

    for i, (lista_ancho_banda, capa_switch) in enumerate(zip(todas_listas_ancho_banda,
todos_capas_switch)):
        ax = axs[i]
        ax.plot(lista_ancho_banda, label="Prueba de Rendimiento")

        x = range(int(carga_trafico_max))
        ax.plot(x, x, label="Ideal", linestyle='--', color='red')

        ax.set_xlabel("Mbits/sec Enviados (Bandwidth)")
        ax.set_ylabel("Mbits/sec Recibidos (Bandwidth)")
        titulo = f"Topología {capa_switch}"
        if use_ecmp:
            titulo += " con ECMP"
        if hosts_adyacentes:
            titulo += "\n(Hosts Adyacentes Desactivados)"
        ax.set_title(titulo)
        ax.grid(True)
        ax.legend()

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Ajusta el layout para dejar espacio para el
título principal
    fig.suptitle("Comparativa de Ancho de Banda con flujo distribuido", fontsize=16, y=0.98)

    # Ajusta el espacio entre subplots
    plt.subplots_adjust(hspace=0.4, wspace=0.3)

    timestamp = time.strftime("%d-%m-%Y-%H%M%S")
    plt.savefig(f"Ancho_banda_flujo_distribuido_comparativa_{timestamp}.svg", format='svg',
dpi=300, bbox_inches='tight')

    plt.show()


def main():
    complementos.logo()  # Muestra el logo

    # Obtenemos los valores de las variables
    capas_switch, n_host, use_ecmp, representar_topologia,hosts_adyacentes, tiempo_prueba,
carga_trafico_max, num_iteraciones = complementos.parametros_N_configuraciones()

    todas_listas_ancho_banda = []
```

```
    # Iteramos sobre las configuraciones de capas_switch
    for i, capa_switch in enumerate(capas_switch):
        print(f"\nIniciando simulación {i+1} de {len(capas_switch)}")

        lista_ancho_banda = pd.Series([])

        # Define y configura la topología de red
        child = Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes)
        paso = carga_trafico_max // (num_iteraciones - 1)
        hosts_adyacentes_count = calcular_hosts_adyacentes(n_host, capa_switch) if
hosts_adyacentes else 0
        # Realiza las pruebas de ancho de banda
        for k, trafico in enumerate(range(1, carga_trafico_max + num_iteraciones, paso), 1):
            print(f"\n\033[1;34m--- Iniciando prueba {k} de {trafico} Mbits/sec ---\033[0m")
            generador_trafico(child, trafico, tiempo_prueba, k, n_host,hosts_adyacentes_count)
            anchos_banda = obtener_ancho_banda(k)
            print(f"\033[1;32mAncho de banda obtenido : {anchos_banda} Mbits/sec\033[0m")
            lista_ancho_banda[int(trafico)] = sum(anchos_banda)
            print(f"\033[1;32mAncho de banda total: {lista_ancho_banda[int(trafico)]:.2f}
Mbits/sec\033[0m")

        # Representa la topología si se solicita
        if representar_topologia:
            visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]),hosts_adyacentes)

        # Cierra Mininet
        ejecutar_comando(child, 'exit')
        child.close()

        # Limpia cualquier proceso de Mininet que pudiera quedar
        subprocess.run('sudo mn -c', shell=True)

        todas_listas_ancho_banda.append(lista_ancho_banda)

        print(f"Emulacion {i+1} completada.")

    # Grafica los resultados de todas las simulaciones en subplots
    graficar(todas_listas_ancho_banda, capas_switch, carga_trafico_max, use_ecmp,
hosts_adyacentes)

    print("Todas las simulaciones se completaron con éxito.")

if __name__ == "__main__":
    main()
```

7.13 Latencia Bajo Carga de Flujos Distribuidos

Este archivo se debe llamarse “latencia_bajo_carga_flujo_distribuido.py”.

```
import pexpect
import sys
import subprocess
import time
import re
import os
import pandas as pd
import matplotlib.pyplot as plt

import complementos
import visualizador_topologia

# Función para ejecutar comandos en Mininet
def ejecutar_comando(child, comando, cadena_esperada='mininet-wifi>', timeout=300):
    child.sendline(comando)  # Envía el comando a Mininet
    try:
        child.expect(cadena_esperada, timeout=timeout)  # Espera la respuesta de Mininet
    except (pexpect.EOF, pexpect.TIMEOUT): # EOF --> el proceso ha terminado inesperadamente
```

```
        # Si hay un error (EOF o timeout), imprime un mensaje
        print(f"Error en el comando: {comando}")

# Función para definir y configurar la topología de red
def Definicion_Topologia(capa_switch, n_host, use_ecmp,hosts_adyacentes):
    # Configura variables de entorno para la topología
    os.environ['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    os.environ['NUM_HOST'] = str(n_host)
    os.environ['HOSTS_ADYACENTES'] = '1' if hosts_adyacentes else '0'

    # Comando para iniciar Mininet con la topología personalizada
    comando = 'sudo -E mn --controller remote,ip=127.0.0.1 --switch ovs,protocols=OpenFlow13 -
-custom spine_leaf.py --topo mytopo'
    child = pexpect.spawn(comando) # Inicia Mininet como un proceso separado, posteriormente
permite interactuar con ejecutar_comando()
    child.logfile_read = sys.stdout.buffer  # Redirige la salida de Mininet a stdout
    child.expect('mininet-wifi>', timeout=300)
    time.sleep(3)  # Espera para asegurar que la red esté lista

    if use_ecmp:
        ECMP(capa_switch, n_host)  # Configura ECMP si es necesario

    if(hosts_adyacentes==0):#Para evitar que tarde mucho la prueba, porque hay host sin
conectividad
        ejecutar_comando(child, 'pingall')  # Verifica conectividad entre todos los hosts
    return child

def medir_latencia(child, n_host,hosts_adyacentes_count,n_ping=10):
    latencias = []
    for i in range(2+hosts_adyacentes_count, n_host + 1):
        comando = f"h{i} ping 10.0.0.1 -c {n_ping} -i 0,2"
        child.sendline(comando)
        child.expect('mininet-wifi>')
        output = child.before.decode('utf-8')
        coincidencia = re.search(r'rtt min/avg/max/mdev =
(\d+\.\d+)/(\d+\.\d+)/(\d+\.\d+)/(\d+\.\d+)', output)
        if coincidencia:
            latencia_promedio = float(coincidencia.group(2))
            latencias.append(latencia_promedio)
        else:
            latencias.append(0)  # Si no se pudo medir, se agrega 0
    return latencias

def obtener_ancho_banda(k):
    anchos_banda = []
    with open(f'log/S{k}.log', 'r') as file:
        for linea in file:
            if "SUM" not in linea:  # Ignoramos líneas con "SUM"
                coincidencia = re.search(r'(\d+(?:\.\d+)?)\s*(M|K)bits/sec', linea)
                if coincidencia:
                    valor = float(coincidencia.group(1))
                    unidad = coincidencia.group(2)
                    # Convertimos Kbits a Mbits si es necesario
                    if unidad == 'M':
                        bandwidth = valor
                    else:
                        bandwidth = valor/1000

                    anchos_banda.append(bandwidth)
    return anchos_banda

# Función para generar tráfico y medir latencia
def generador_trafico(child, carga_total, tiempo_prueba, k, n_host, n_ping, flujos_paralelos,
hosts_adyacentes_count):
    os.makedirs('log', exist_ok=True)

    hosts_activos = n_host - hosts_adyacentes_count - 1  # Excluyendo h1 y los hosts
adyacentes

    if carga_total > 0:
        ejecutar_comando(child, f"h1 iperf -s -t {tiempo_prueba*1.2} &>log/S{k}.log&")
```

```
        mbits_por_host = int(carga_total / hosts_activos)
        mbits_por_flujo = float(f"{mbits_por_host / flujos_paralelos:.1f}")  # Divide el ancho
de banda por flujo

        for i in range(2+hosts_adyacentes_count, n_host+1):
            ejecutar_comando(child, f'h{i} iperf -c 10.0.0.1 -b {mbits_por_flujo}m -t
{tiempo_prueba} -P {flujos_paralelos} &>log/h{i}.log&')

    # Medir latencia
    latencias = medir_latencia(child, n_host, hosts_adyacentes_count, n_ping)

    if carga_total > 0:
        # Esperar a que termine la prueba de iperf
        time.sleep(tiempo_prueba)

    return latencias

# Función para configurar ECMP (Equal-Cost Multi-Path)
def ECMP(capa_switch, n_host):
    ecmp_env = os.environ.copy()
    ecmp_env['CAPA_SWITCH'] = ','.join(map(str, capa_switch))
    ecmp_env['NUM_HOST'] = str(n_host)
    ecmp_env['USE_ECMP'] = '1'
    subprocess.run(['sudo', '-E', 'python3', 'ECMP.py'], env=ecmp_env, check=True)

def calcular_hosts_adyacentes(n_host, capa_switch):
    hosts_por_leaf = n_host // capa_switch[1]
    return hosts_por_leaf - 1  # Excluyendo el primer host

# Función para graficar ancho de banda y latencia
def Graficar(Latencia_obtenida, Titulo_graficas, capa_switch, hosts_adyacentes,
n_host,use_ecmp,flujos_paralelos,hosts_activos):
    plt.figure(figsize=(14,6))
    titulo=f"Latencia con Carga de Flujos Distribuidos en Spine-Leaf {capa_switch}"
    if use_ecmp:
        titulo += " con ECMP"
    if hosts_adyacentes:
        titulo += "\n(Hosts Adyacentes Desactivados)"
    if flujos_paralelos>1:
        titulo += f"\nTotal de flujos: {flujos_paralelos*hosts_activos} ({flujos_paralelos}
flujos x {hosts_activos} hosts activos)"

    plt.title(titulo)
    plt.grid(True)

    hosts_adyacentes_count = calcular_hosts_adyacentes(n_host, capa_switch) if
hosts_adyacentes else 0
    n_hosts_activos = len(Latencia_obtenida[0])
    eje_x = range(2 + hosts_adyacentes_count, 2 + hosts_adyacentes_count + n_hosts_activos)

    for i, latencias in enumerate(Latencia_obtenida):
        if i == 0:
            plt.plot(eje_x, latencias, label=f'{Titulo_graficas[i]} Mbits/sec',
linestyle='dashed')
        else:
            plt.plot(eje_x, latencias, label=f'{Titulo_graficas[i]} Mbits/sec')

    plt.yscale('log')
    plt.ylabel('Latencia (Escala logarítmica)')
    plt.xlabel('Número de Host')
    plt.xticks(eje_x)
    plt.legend()
    timestamp = time.strftime("%d-%m-%Y-%H%M%S")

    plt.savefig(f"Prueba_latencia_bajo_carga{str(capa_switch)}_{timestamp}.svg", format='svg',
dpi=300, bbox_inches='tight')
    plt.show()

def main():
    complementos.logo()
```

```
    capa_switch, n_host, use_ecmp, representar_topologia, hosts_adyacentes, carga_trafico_max,
num_iteraciones, n_ping, flujos_paralelos = complementos.parametros_latencia_bajo_carga()
    tiempo_prueba = 0.2 * n_ping * n_host

    child = Definicion_Topologia(capa_switch, n_host, use_ecmp, hosts_adyacentes)

    Latencia_obtenida = []
    Titulo_graficas = []

    hosts_adyacentes_count = calcular_hosts_adyacentes(n_host, capa_switch) if
hosts_adyacentes else 0
    hosts_activos = n_host - hosts_adyacentes_count - 1  # Excluyendo h1 y los hosts
adyacentes

    for k in range(num_iteraciones + 1):
        if k == 0:
            tamaño_mbits_total = 0  # Primera iteración sin carga
        else:
            tamaño_mbits_total = (k * carga_trafico_max) / num_iteraciones

        latencias = generador_trafico(child, tamaño_mbits_total, tiempo_prueba, k, n_host,
n_ping, flujos_paralelos, hosts_adyacentes_count)

        if k == 0:
            print("\033[1;31mIteración 0: Tráfico 0 Mbits/sec (carga nula)\033[0m")
        else:
            print(f"\n\033[1;31m---Iteración {k}: Tráfico total {tamaño_mbits_total:.2f}
Mbits/sec---\033[0m")
            if tamaño_mbits_total > 0:
                print(f"\033[1;34mTráfico por host activo: {tamaño_mbits_total /
hosts_activos:.2f} Mbits/sec\033[0m")
                print(f"\033[1;34mNúmero de flujos paralelos por host:
{flujos_paralelos}\033[0m")
                ancho_banda = obtener_ancho_banda(k)
                print(f"\033[1;32mAncho de banda obtenido de los flujos: {ancho_banda}
Mbits/sec\033[0m")
                print(f"\033[1;32mAncho de banda total: {sum(ancho_banda):.2f}
Mbits/sec\033[0m")

        print(f"\033[1;32mLatencias: {latencias}\033[0m")

        Latencia_obtenida.append(latencias)
        Titulo_graficas.append(f"{tamaño_mbits_total:.0f}")

    if representar_topologia:
        visualizador_topologia.Dibujar_topologia(capa_switch[0], capa_switch[1],
int(n_host/capa_switch[1]), hosts_adyacentes)

    ejecutar_comando(child, 'exit')
    child.close()
    subprocess.run('sudo mn -c', shell=True)

    Graficar(Latencia_obtenida, Titulo_graficas, capa_switch, hosts_adyacentes,
n_host,use_ecmp,flujos_paralelos,hosts_activos)

    print("El script se completó con éxito.")

if __name__ == "__main__":
    main()
```

8 BIBLIOGRAFÍA

[1] «SDN (Software Defined Networking)», IONOS Digital Guide. Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://www.ionos.es/digitalguide/servidores/know-how/software-defined-network/

[2] «Software Defined Networking (SDN): Benefits and Challenges of Network Virtualization - javatpoint», www.javatpoint.com. Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://www.javatpoint.com/software-defined-networking-sdn-benefits-and-challenges-of-network-virtualization

[3] A. Haggag, «Network Optimization for Improved Performance and Speed for SDN and Security Analysis of SDN Vulnerabilities», *Int. J. Comput. Netw. Commun. Secur.*, vol. 7, pp. 83-90, may 2019.

[4] «¿Qué es el plano de control? | Plano de control vs. plano de datos», Cloudflare. Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://www.cloudflare.com/es-es/learning/network-layer/what-is-the-control-plane/

[5] object Object, «Scalable ReliableControllerPlacementinSoftwareDefinedNetworking», Accedido: 5 de noviembre de 2024. [En línea]. Disponible en: https://core.ac.uk/reader/355870830

[6] «Redes Definidas por Software (SDN): Tipos, Ventajas y Aplicaciones | Comunidad FS», Knowledge. Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://community.fs.com/es/article/software-defined-networking-sdn-types-advantages-and-applications.html

[7] G. Yangyang, «What Is OpenFlow? How Does It Relate to SDN?», Huawei. Accedido: 20 de julio de 2024. [En línea]. Disponible en: https://info.support.huawei.com/info-finder/encyclopedia/en/OpenFlow.html

[8] A. Al-Ani, M. Anbar, S. U. A. Laghari, y A. Al-Ani, «Mechanism to prevent the abuse of IPv6 fragmentation in OpenFlow networks», *PLOS ONE*, vol. 15, p. e0232574, may 2020, doi: 10.1371/journal.pone.0232574.

[9] «¿Qué es OpenFlow? Definición y relación con SDN | Reef Recovery». Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://reefrecovery.org/es/qu%c3%a9-es-openflow-definici%c3%b3n-y-relaci%c3%b3n-con-sdn/

[10] D. Blandón, «OPENFLOW: EL PROTOCOLO DEL FUTURO* Openflow: The future protocol», mar. 2013.

[11] E. Collado, «OpenFlow y SDN», Eduardo Collado. Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://www.eduardocollado.com/2023/10/12/openflow-y-sdn/

[12] «VERSIONES - OPENFLOW Y EL MODELO SDN - Análisis de seguridad en redes SDN (Redes definidas por». Accedido: 20 de julio de 2024. [En línea]. Disponible en: https://1library.co/article/versiones-openflow-modelo-an%C3%A1lisis-seguridad-redes-redes-definidas.y81g6drz

[13] «¿Qué es la arquitectura “spine-leaf”?», HPE Aruba Networking. Accedido: 8 de marzo de 2024. [En línea]. Disponible en: //www.arubanetworks.com/es/faq/que-es-la-arquitectura-spine-leaf/

[14] «¿Qué es la arquitectura spine-leaf y cómo diseñarla? | Comunidad FS», Knowledge. Accedido: 12 de octubre de 2024. [En línea]. Disponible en: https://community.fs.com/es/article/leaf-spine-with-fs-com-switches.html

[15] A. Walton, «Propósito de STP » CCNA desde Cero», CCNA desde Cero. Accedido: 13 de octubre de 2024. [En línea]. Disponible en: https://ccnadesdecero.es/proposito-stp/

[16] «STP Part I | CCNA Blog». Accedido: 9 de marzo de 2024. [En línea]. Disponible en: https://www.ccnablog.com/stp-part-i/
[17] Daniel, «Equal Cost Multi-Path (ECMP) Explanation & Configuration», Study CCNA. Accedido: 9 de marzo de 2024. [En línea]. Disponible en: https://study-ccna.com/ecmp-equal-cost-multi-path/
[18] D. Singh, «Flow Distribution Across ECMP Paths». Accedido: 3 de noviembre de 2024. [En línea]. Disponible en: https://dipsingh.github.io/Flow-Distribution-Across-ECMP/
[19] «Using OpenFlow — Open vSwitch 3.4.0 documentation». Accedido: 12 de noviembre de 2024. [En línea]. Disponible en: https://docs.openvswitch.org/en/stable/faq/openflow/
[20] P. N. Roldán, «Ley de los grandes números», Economipedia. Accedido: 3 de noviembre de 2024. [En línea]. Disponible en: https://economipedia.com/definiciones/ley-los-grandes-numeros.html
[21] J. S. Partre, «literaturaconciencia: EL SIGNO DE LOS CUATRO (SIR ARTHUR CONAN DOYLE) Y LA LEY DE LOS GRANDES NÚMEROS», literaturaconciencia. Accedido: 3 de noviembre de 2024. [En línea]. Disponible en: https://literaturaconciencia.blogspot.com/2018/06/el-signo-de-los-cuatro-sir-arthur-conan.html
[22] A. Jenifa, «Troubleshooting Network Latency with Wireshark», Geekflare. Accedido: 25 de junio de 2024. [En línea]. Disponible en: https://geekflare.com/troubleshooting-network-latency-wireshark/
[23] S. Vidal, «¿Qué es la Latencia en una Conexión? ▷». Accedido: 25 de junio de 2024. [En línea]. Disponible en: https://tecnobits.com/que-es-la-latencia-en-una-conexion/
[24] M. with by F. VI VDESIGN &., «Network Latency: how latency, bandwidth & packet drops impact your speed», Scaleway. Accedido: 25 de junio de 2024. [En línea]. Disponible en: https://www.scaleway.com/en/blog/understanding-network-latency/
[25] «What is Bandwidth? - Definition and Details». Accedido: 20 de julio de 2024. [En línea]. Disponible en: https://www.paessler.com/it-explained/bandwidth
[26] M. Hayes, «What is a Network Traffic Flow?», Bits 'n Bytes. Accedido: 20 de julio de 2024. [En línea]. Disponible en: https://mattjhayes.com/2018/09/26/what-is-a-network-traffic-flow/
[27] «Enterprise Open Source and Linux», Ubuntu. Accedido: 12 de marzo de 2024. [En línea]. Disponible en: https://ubuntu.com/
[28] «Ubuntu 20.04.6 LTS (Focal Fossa)». Accedido: 12 de marzo de 2024. [En línea]. Disponible en: https://releases.ubuntu.com/focal/
[29] «Get Started», Mininet-WiFi. Accedido: 12 de marzo de 2024. [En línea]. Disponible en: https://mininet-wifi.github.io/get-started/
[30] «Rufus - Cree unidades USB arrancables fácilmente». Accedido: 12 de marzo de 2024. [En línea]. Disponible en: https://rufus.ie/es/
[31] «Mininet Overview - Mininet». Accedido: 12 de marzo de 2024. [En línea]. Disponible en: https://mininet.org/overview/
[32] L. M. Amaya Fariño, J. F. Arroyo Pizarro, M. Jaramillo Infante, A. R. Tumbaco Reyes, y B. M. Mendoza Morán, «SDN Redes definidas por Software usando MiniNet», *Rev. Científica Tecnológica UPSE*, vol. 9, n.º 1, pp. 48-56, jun. 2022, doi: 10.26423/rctu.v9i1.489.
[33] «Mininet-WiFi», Mininet-WiFi. Accedido: 12 de marzo de 2024. [En línea]. Disponible en: https://mininet-wifi.github.io/

[34] *intrig-unicamp/mininet-wifi*. (10 de octubre de 2024). Python. INTRIG. Accedido: 12 de octubre de 2024. [En línea]. Disponible en: https://github.com/intrig-unicamp/mininet-wifi
[35] R. Maldonado, «Ventajas y desventajas de Python | KeepCoding Bootcamps». Accedido: 15 de abril de 2024. [En línea]. Disponible en: https://keepcoding.io/blog/ventajas-y-desventajas-de-python/
[36] R. Chua, «Where Have All the SDN Controllers Gone?», SDxCentral. Accedido: 17 de abril de 2024. [En línea]. Disponible en: https://www.sdxcentral.com/articles/analysis/where-have-all-the-sdn-controllers-gone/2019/03/
[37] «Introducción — Documentación de OpenDaylight Documentación de Scandium». Accedido: 21 de julio de 2024. [En línea]. Disponible en: https://docs.opendaylight.org/en/latest/getting-started-guide/introduction.html
[38] César AA, *Instalación de opendaylight en Ubuntu integrado con mininet*, (12 de julio de 2021). Accedido: 13 de marzo de 2024. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=Sd-7TFBl3iE
[39] «Index of /repositories/opendaylight.release/org/opendaylight/integration/distribution-karaf». Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/opendaylight/integration/distribution-karaf/
[40] «ONOS - ONOS - Wiki». Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://wiki.onosproject.org/
[41] «Requirements - ONOS - Wiki». Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://wiki.onosproject.org/display/ONOS/Requirements
[42] «Downloads - ONOS - Wiki». Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://wiki.onosproject.org/display/ONOS/Downloads
[43] «Running ONOS as a service - ONOS - Wiki». Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://wiki.onosproject.org/display/ONOS/Running+ONOS+as+a+service
[44] «Containernet», Containernet. Accedido: 14 de marzo de 2024. [En línea]. Disponible en: https://containernet.github.io/
[45] G. Jevtic, «Linux ping Command with Examples | phoenixNAP KB», Knowledge Base by phoenixNAP. Accedido: 21 de julio de 2024. [En línea]. Disponible en: https://phoenixnap.com/kb/linux-ping-command-examples
[46] «iPerf - The TCP, UDP and SCTP network bandwidth measurement tool». Accedido: 21 de julio de 2024. [En línea]. Disponible en: https://iperf.fr/
[47] «Iperf Limitations Measuring Available Network Bandwidth». Accedido: 21 de julio de 2024. [En línea]. Disponible en: https://www.thousandeyes.com/blog/caveats-of-traditional-network-tools-iperf
[48] «iPerf - iPerf3 and iPerf2 user documentation». Accedido: 21 de julio de 2024. [En línea]. Disponible en: https://iperf.fr/iperf-doc.php
[49] «Network Cable Propagation Delay», Fluke Networks. Accedido: 28 de marzo de 2024. [En línea]. Disponible en: https://www.flukenetworks.com/knowledge-base/dtx-cableanalyzer/propagation-delay
[50] «How to Work with Fast-Failover OpenFlow Groups - Floodlight Controller - Confluence». Accedido: 14 de septiembre de 2024. [En línea]. Disponible en: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/7995427/How+to+Work+with+Fast-Failover+OpenFlow+Groups#HowtoWorkwithFast-FailoverOpenFlowGroups-TheSELECTGroup

[51] A. Jenifa, «Protocolo de resolución de direcciones (ARP): Hoja de ruta hacia la traducción de direcciones de red | Geekflare», Geekflare Spain. Accedido: 15 de septiembre de 2024. [En línea]. Disponible en: https://geekflare.com/es/address-resolution-protocol/

[52] Huangmc, *Huangmachi/ECMP*. (12 de septiembre de 2024). Python. Accedido: 23 de septiembre de 2024. [En línea]. Disponible en: https://github.com/Huangmachi/ECMP

[53] «learn-sdn-with-ryu/ryu_part8.md at master · knetsolutions/learn-sdn-with-ryu», GitHub. Accedido: 23 de septiembre de 2024. [En línea]. Disponible en: https://github.com/knetsolutions/learn-sdn-with-ryu/blob/master/ryu_part8.md

[54] 262588213843476, «Load balance(Multipath) Application on RYU», Gist. Accedido: 23 de septiembre de 2024. [En línea]. Disponible en: https://gist.github.com/Yiyiyimu/a042e2b8218bfe8614ade2bd14d8d8f4

[55] «ping/iperf3 got unstable measurement results · Issue #1138 · mininet/mininet», GitHub. Accedido: 3 de abril de 2024. [En línea]. Disponible en: https://github.com/mininet/mininet/issues/1138

[56] M. Sandri, «Answer to “iperf Server and Client Differences”», Stack Overflow. Accedido: 8 de octubre de 2024. [En línea]. Disponible en: https://stackoverflow.com/a/33380951

[57] B. Freke, «iperf Server and Client Differences», Stack Overflow. Accedido: 8 de octubre de 2024. [En línea]. Disponible en: https://stackoverflow.com/q/33121740