ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Estudio de técnicas de Ingeniería de Tráfico basadas en SDN
(Study of SDN Traffic Engineering techniques)

Para acceder al Título de

Graduado en
Ingeniería de Tecnologías de Telecomunicación

Autor: Miguel Betegón García

Junio - 2018

AGRADECIMIENTOS

Quisiera agradecer a mi director del trabajo, *Alberto Eloy García*, todo el apoyo brindado durante la realización del proyecto. Por su inestimable ayuda y enseñanzas, sin las cuales hubiera sido más que difícil la realización de este trabajo. Ante todas las adversidades que se han presentado, siempre ha estado dispuesto a dedicarme su tiempo y conocimiento. He aprendido mucho de tí, muchas gracias *Alberto*.

RESUMEN

Las redes definidas por software o *Software-Defined Network (SDN)* abarcan varios tipos de tecnologías de red para conseguir que la misma sea ágil y flexible. El objetivo de las redes SDN es que los ingenieros y administradores de red respondan rápidamente a los cambiantes requisitos comerciales.

En una red definida por software, el administrador puede dar forma al tráfico desde una consola de control centralizada sin tener que tocar individualmente cada switch y puede proporcionar servicios donde se necesiten en la red, sin importar a que dispositivos está conectado un servidor o un componente hardware.

Esto se consigue a través de técnicas de Ingeniería de Tráfico, y es de lo que trata este trabajo. De la posibilidad de aplicar dichas técnicas en redes SDN para comprobar que efectivamente las redes definidas por software son eficaces y que tienen aplicaciones ventajosas hoy en día.

Se desarrollan dos casos de uso reales que incorporan técnicas de Ingeniería de Tráfico en redes SDN y una vez implementados, se realiza un test para comprobar su funcionamiento y determinar su validez en función de los resultados. Estos casos de uso son en realidad aplicaciones que funcionan en una capa superior a OpenFlow.

Palabras Clave— Redes definidas por software, ingeniería de tráfico, Mininet, Ryu, OpenFlow.

ABSTRACT

Software-Defined Networks, *SDN*, cover several kinds of network technology aimed at making the network agile and flexible. The goal of SDN is to allow network engineers and administrators to respond quickly for changing business requirements.

In a Software-Defined Network, a network administrator can shape traffic from a centralized control console without having to touch individual switches, and can deliver services to wherever they are needed in the network, without regard to what specific devices a server or other hardware components are connected to.

This can be achieved through Traffic Engineering techniques, what this work is all about. The possibility of applying these techniques in SDN networks to verify that Software-Defined Networks are effective and that they have advantageous applications nowadays.

Two real use cases that incorporate Traffic Engineering techniques in SDN networks are developed in the project. Once they are implemented, a test is carried out to check its operation and determine its validity based on the given results. These use cases are actually applications sitting in top of the OpenFlow layer.

Keywords— Software-Defined Network, Traffic Engineering, Mininet, Ryu, OpenFlow.

Índice

Índice de Figuras

Capítulo 1

Introducción

Debido a la creciente demanda en las redes, en estos años se ha visto una evolución en el mercado de las telecomunicaciones con clara tendencia hacia la implantación de redes definidas por software (*SDN*).

Las redes SDN surgieron a principios de 2010 por necesidad, y es que, muchas de las redes de entonces, que se siguen utilizando en el presente, fueron diseñadas como aplicaciones cliente-servidor, que se ejecutan en una infraestructura no virtualizada, tal como *Rohit Mehra* y *Brad Casemore* ya describieron en [1], en su previsión Sobre SDN publicada en 2016 :

"Virtualization, cloud, mobility, and now the Internet of Things (IoT) have exposed the limitations of traditional network architectures and operational models"

Por ahora, SDN ha crecido más allá de su adolescencia y euforia prematura y se ha establecido como un producto conocido. No es una próxima novedad en el horizonte de la creación de redes, sino una realidad que la mayoría empresas y proveedores de servicios de todo el mundo ya han adoptado.

1.1. Motivación

Si bien existen diversos proyectos sobre el campo de investigación que plantean las soluciones SDN, la mayoría son mera aplicaciones de tutoriales, como es el caso del trabajo de fin de grado realizado por *Rubén Isa Hidalgo* y presentado en octubre de 2016, *Implementación de un laboratorio virtual para aprendizaje de SDN* [2], del que surge este proyecto.

El mismo, tiene como propósito general exponer dos casos de uso real de las

redes definidas por software, aplicando para ello técnicas de ingeniería de tráfico. Desde el despliegue de la red en un emulador virtual, hasta el uso de un controlador que se encargará de tomar las decisiones en la red.

Surge así, la necesidad de ir un paso más allá en el mundo de SDN, construyendo aplicaciones reales que se puedan aplicar en el día de hoy, siguiendo la línea abierta por los trabajos anteriormente desarrollados.

1.2. Objetivos

Partiendo de la experiencia obtenida a partir del trabajo anterior [2], se pretende utilizar Mininet como software para la docencia, e implementar varios ejemplos reales de redes.

El objetivo fundamental es obtener un entorno de laboratorio práctico en el que poder visualizar la red, implementar las redes y se crear scripts que apliquen técnicas de ingeniería de tráfico.

Como resultado, la configuración final deberá permitir analizar el comportamiento de las diferentes técnicas de ingeniería de tráfico, para su aprendizaje por parte del usuario.

1.3. Estructura del proyecto

Este documento está organizado en 4 capítulos mediante los cuales se da el mayor enfoque posible, tanto desde el punto de vista práctico como teórico, de todos los aspectos tratados a lo largo del trabajo.

Tras este breve capítulo de **introducción**, el segundo capítulo expone los **Conceptos teóricos**, donde se hace una revisión del estado del arte de tecnología SDN, de cada uno de los componentes, tecnologías y demás herramientas empleadas. Define que es la ingeniería de tráfico, sus escenarios existentes y cómo mejorar dichos escenarios con SDN. Así, en el siguiente apartado del capítulo dos se explican las técnicas de routing multicamino, los algoritmos que se emplean para ello y cómo se hace el cálculo del coste por camino.

La aplicación práctica de los algoritmos anteriores es descrita en el capítulo tres, dónde también se define el escenario de red y cómo conectar los switches que

forman parte de la red, así como el controlador correspondiente.

Por último, se discuten las principales conclusiones y se proponen ideas para extender la aplicación de este proyecto.

Capítulo 2

Conceptos teóricos

Antes de comenzar con la parte práctica del trabajo, cabe explicar el estado del arte en el que se encuentran las redes definidas por software. Está arquitectura de red nació hace ya 8 años, consiguiendo avances y mayor adopción desde entonces, así como los principales elementos que forman parte de este trabajo.

2.1. SDN (Software Defined Network)

SDN es un paradigma dentro de las redes de comunicaciones que permite eliminar muchas de las limitaciones de las infraestructuras de red actuales. El objetivo de SDN es separar el plano de datos del plano de control, centralizando el estado de la red y la capacidad de toma de decisiones. Así la programación en el plano de control (Controlador SDN), simplifica la operación en el plano de datos (Dispositivos de red SDN) y permite que la infraestructura subyacente sea abstraída para que aplicaciones y servicios puedan tratar a la red como una entidad lógica o virtual.

2.1.1. Estado del arte

Al separar la lógica de control de los dispositivos de red, SDN permite la programabilidad de la misma, su administración simplificada y autónoma y por lo tanto, abre mercados y oportunidades para los operadores, la red y los proveedores de servicios.

Los conceptos y tecnologías relacionadas con SDN se han desarrollado hasta el punto que el siguiente paso es crucial, ya que todos los participantes (fabricantes de equipos de red, proveedores de software y empresas, entre otros) están evaluando las estrategias apropiadas para su organización en términos de una transición rentable desde las redes tradicionales a una solución SDN.

La *Open Networking Foundation* define una arquitectura de alto nivel para SDN con tres capas o planos principales, como se muestra en la figura 2.1, creada con la información proporcionada en [3]. En esos planos se encuentran los elementos básicos de SDN: dispositivos SDN, controlador SDN y aplicaciones.

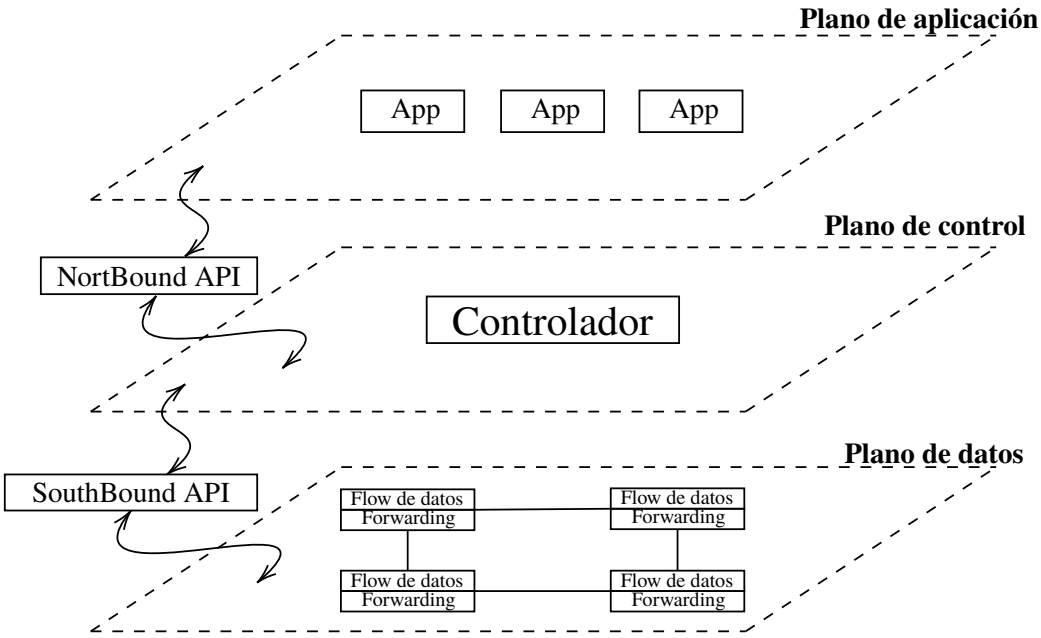

Figura 2.1: Arquitectura de alto nivel SDN.

Los dispositivos SDN contienen componentes para decidir qué hacer con el tráfico entrante. El controlador SDN programa los dispositivos de red y presenta una abstracción de la infraestructura de red subyacente a las aplicaciones SDN.

Por otra parte, el controlador permite que una aplicación SDN defina los flujos de tráfico y las rutas en los dispositivos de red. De esta forma, satisface sus necesidades y responde a los requisitos dinámicos de los usuarios y las condiciones del tráfico y la red.

Esta comunicación entre capas es posible gracias a las *SouthBound* y *NorthBound* APIs.

Por un lado, las *Southbound APIs* son usadas para la comunicación entre el controlador SDN y los elementos de red (switches, routers, etc). Estas pueden ser de código abierto como es el caso de OpenFlow o propietarias. OpenFlow fue la primera y probablemente la más conocida pero no la única, existen varias NetConf[1], Lisp[2] y OpFlex[3] entre otras.

Por otro lado, las *Northbound APIs* son generalmente Rest APIs utilizadas para la comunicación entre el controlador SDN y los servicios y aplicaciones que corren por encima de la red, en la capa de aplicación, como muestra la figura 2.1. También se utilizan para integrar el controlador SDN con pilas de automatización como OpenStack[4].

Normalmente, las *Northbound APIs* están integradas dentro del controlador SDN, un claro ejemplo es el controlador Ryu, que permite esa comunicación entre el controlador y las aplicaciones directamente desde el script que ejecuta la aplicación. Ocurre lo mismo con los controladores POX[5] y OpenDayLight[5].

Por todo ello, las *Northbound* y *Southbound* APIs son clave en SDN. Sin ellas no sería posible esa separación de los planos de control y de datos ni la creación de aplicaciones.

2.1.2. Configuración física

Antes de explicar cada uno de los elementos implicados en la realización de este proyecto, se pretende dar una visión general de cómo encajan cada uno de ellos en el entorno de trabajo.

En la figura 2.2 se muestra el esquema físico del proyecto, en el que figuran los elementos principales. Comenzando desde fuera, se encuentra el PC, desde donde se realiza el trabajo.

[1] **NetConf –** https://trac.ietf.org/trac/netconf
[2] **Lisp –** https://wiki.opendaylight.org/view/OpenDaylight_Lisp_Flow_Mapping:Integration_Tests
[3] **Cisco OpFlex –** https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html
[4] **OpenStack –** https://www.openstack.org/software/
[5] **POX –** https://noxrepo.github.io/pox-doc/html/
[5] **OpenDayLight –** https://wiki.opendaylight.org/view/Main_Page

En el PC, se tiene instalado el programa *VirtualBox*, que permite correr sistemas operativos como máquinas virtuales. En este caso, se correrá el sistema operativo de *Linux*, *Ubuntu* en su versión *16.04*.

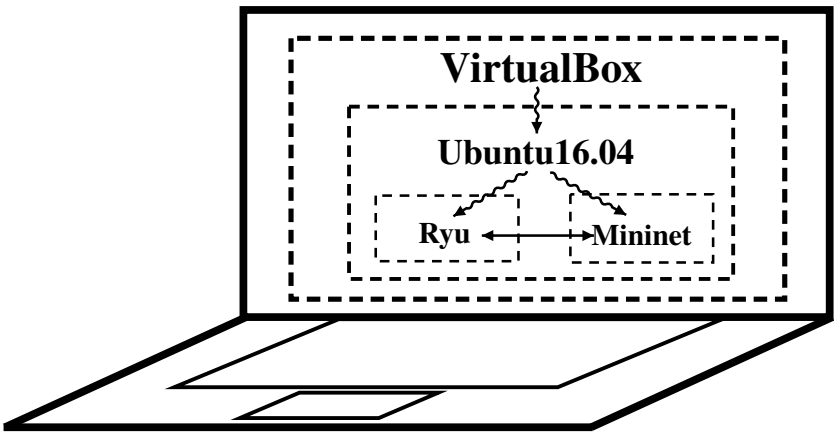

Figura 2.2: Esquema físico del proyecto.

Dentro de la máquina virtual de *Ubuntu* se instalará y ejecutará *Mininet*, que permitirá la creación de las redes SDN de forma virtual. También se instalará *Ryu*, el controlador SDN elegido, en *Ubuntu*, consiguiendo así que *Mininet* y *Ryu* se puedan comunicar.

2.1.3. Protocolo OpenFlow

OpenFlow es un protocolo estandarizado por *Open Networking Foundation* en 2013 que define la comunicación hacia el sur (Southbound) entre un controlador y un switch SDN, como se detalla en las especificaciones del protocolo, en [3]. Los mensajes de comunicación entre los dos se transmiten a través de un canal seguro que se implementa haciendo uso de una conexión TLS (*Transport Layer Security*) a través de TCP. Mediante el intercambio de comandos y paquetes, el controlador define y programa el comportamiento de forwarding del switch y este realiza el reenvío de acuerdo a ello, informando de su estado de configuración y de las condiciones del tráfico al controlador.

El tráfico de usuario se clasifica en *flows* (flujos) en función de sus características más significativas. Un switch OpenFlow realiza búsquedas y reenvío de paquetes de acuerdo con el flow al que pertenecen. Un flow es un conjunto de paquetes transferidos desde un punto final de red (o conjunto de puntos finales) a otro punto extremo de la red (o conjunto de puntos finales) y a su vez, los puntos

finales (*endpoints*) se pueden definir como pares de direcciones IP-TCP/UDP, puntos terminales VLAN o puertos de entrada-salida de un switch OpenFlow, etc.

Para el controlador, un Switch OpenFlow no es más que un conjunto de tablas de flows o *flow tables*. una flow table consta de entradas de flows. Una entrada de flow comprende campos de cabeceras, de contador y de acción:

- Los campos de cabeceras se utilizan para identificar el flow de un paquete.
- Los campos de contador para recopilar estadísticas del flujo identificado.
- Los campos de acción proporcionan las instrucciones y acciones que el switch debe realizar en los paquetes de ese flow.

Como resultado del intercambio de los anteriores mensajes, el controlador, en función de su conocimiento de la topología de la red subyacente, la capacidad del dispositivo y de los requisitos de la capa de aplicación, define un flujo con los campos apropiados y utiliza el protocolo OpenFlow para programar el Switch con las tablas y entradas de flows necesarias. La figura 2.3 resume las funciones principales de un switch OpenFlow, todas implementadas alrededor de las tablas de flows.

Dichas funciones están asociadas al intercambio de tres tipos de mensajes:

Controller-to-Switch messages – Los mensajes de controlador a Switch se utilizan para gestionar y programar el Switch (e.g. configurar el Switch, enviar tablas de flows, pedir estadísticas de tráfico, etc).

Asynchronous messages – Los mensajes asíncronos son enviados desde el Switch al controlador sin haber sido solicitador por el controlador. Se utilizan para notificar al controlador sobre cambios en el estado del Switch y para informar sobre eventos en la red, incluyendo errores.

Symmetric messages – Los mensajes simétricos son utilizados tanto por el Switch como por el controlador para determinar la vida de la conexión.

Las funciones principales de un Switch OpenFlow incluyen la interacción con el controlador a través del protocolo OpenFlow, la identificación de flows de tráfico a través de la correspondencia (*Matching*) de paquetes, el forwarding de paquetes y las estadísticas de informes y el cambio de estado al controlador, como se indica en la figura 2.3 diseñada con la información obtenida en [3].

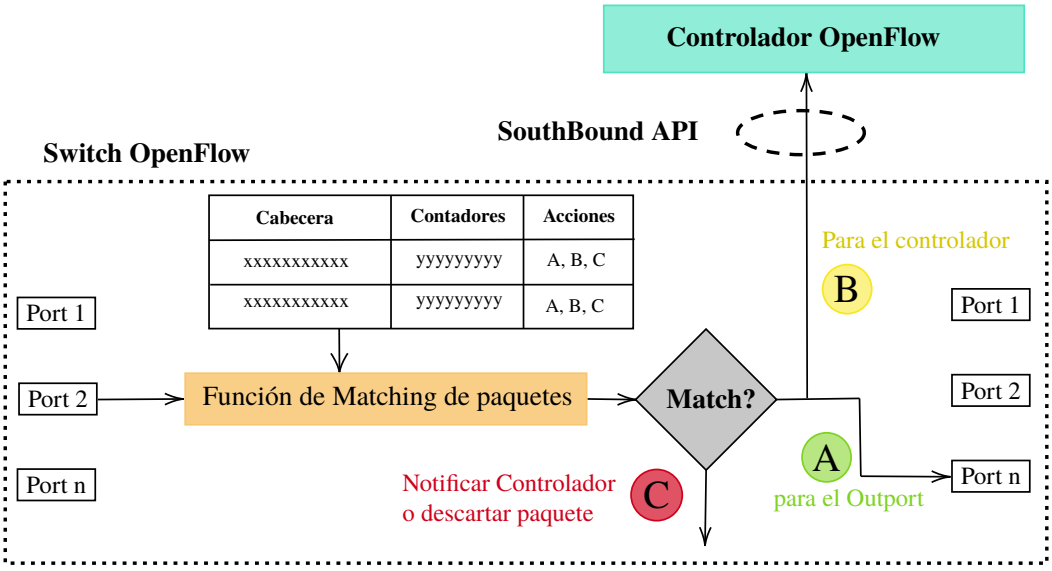

Figura 2.3: Switch OpenFlow. Operación básica.

2.1.4. Controlador OpenFlow

El controlador es el software que gestiona los recursos compartidos de la infraestructura de red subyacente entre las aplicaciones. Se comunica con los dispositivos de red para obtener información sobre ellos con el fin de construir el estado de información global de la infraestructura de red subyacente, y programa las configuraciones y políticas específicas de la aplicación en estos dispositivos para controlar su comportamiento de forwarding. Además, el controlador también ofrece un entorno de ejecución para la programación de la red.

En la actualidad, existen bastantes controladores, muchos de ellos son propietarios de empresas aunque los más populares son opensource. Algunos de los controladores opensource más conocidos son *OpenDayLight*, *Ryu*[1], *Floodlight*[2] y *POX* , entre otros. En [2] se hace uso del controlador *OpenDayLight*, sin embargo en este caso el controlador elegido es Ryu, debido a su facilidad de uso, su popularidad y todas las características y funcionalidades extra que incluye en comparación con el resto.

Es importante cuando se trabaja con proyectos opensource, que estos tengan una buena comunidad que los respalde, pues sin ella se pueden quedar abandonados. Es por eso que todos los controladores antes nombrados gozan de buenas comunidades y desarrolladores entre los que se encuentran grandes empresas del sector.

[1]**Ryu –** https://osrg.github.io/ryu/
[2]**Floodlight –** http://www.projectfloodlight.org/floodlight/

Sobre los cuatro controladores comentados, se encuentra una comparación en la tabla 2.1, en la que se muestran las principales características de los mismos.

Características	POX	Ryu	FloodLight	OpenDayLight
Rest API	No	Sí	Sí	Sí
OpenSource	Sí	Sí	Sí	Sí
Edad	5 años	5 años	6 años	4 años
Lenguajes	Python	Python	Java	Java
OpenStack Networking	No	Fuertemente	Medio	Medio
Virtualización	Mininet, Open vSwitch	Mininet, Open vSwitch	Mininet, Open vSwitch	Mininet, Open vSwitch

Tabla 2.1: Comparación entre los principales controladores SDN.

Cabe destacar que todos son opensource y que admiten la virtualización por medio de Mininet y Open vSwitch. Además, Ryu es el único que no tiene problema en integrarse con *OpenStack*, la plataforma de computación en la nube opensource más importante y que más ha crecido en los últimos años.

2.1.4.1. Ryu

Ryu es un *Component-based SDN framework*, [4], es decir, un entorno de trabajo que proporciona componentes software que se utilizan en SDN, entre ellos un controlador, con una API bien definida que facilita a los desarrolladores la creación de nuevas aplicaciones de administración y control de red. Ryu es compatible con varios protocolos para la gestión de dispositivos de red, como *OpenFlow, Netconf, OF-config,* etc. Acerca de OpenFlow, Ryu admite totalmente las versiones 1.0, 1.2, 1.3, 1.4, 1.5 y Nicira. Ryu ha sido escrito completamente en el lenguaje de programación *Python* y al ser un proyecto opensource, todo su código está disponible gratuitamente bajo la licencia Apache 2.0.

El controlador Ryu es un controlador SDN opensource diseñado para aumentar

la agilidad de la red al facilitar la administración y la adaptación al tráfico. Es el elegido para este trabajo y, mediante su Northbound API, se creará una aplicación en forma de script en Python para balancear la carga de una SDN.

2.1.5. Mininet

Desde el sitio web de Mininet,[5], definen a la misma como una red virtual instantánea en su portátil o PC. Mininet crea una red virtual realista ejecutando kernel real, switch y código de aplicación, en una sola máquina (Máquina Virtual, Nube o nativa) en cuestión de segundos y con un solo comando: `$ sudo mn`.

En este proyecto, se implementan las redes SDN necesarias en Mininet, que corren dentro de la máquina virtual creada en VirtualBox. Una de las ventajas de emplear Mininet es que incluye *MiniEdit*, una simple interfaz gráfica. MiniEdit permite tener una visión general de la red, puesto que muestra los elementos que la forman.

Mininet incluye un controlador SDN pero permite hacer uso de un controlador externo, que en este caso será Ryu. Entonces, habrá que configurar los switches para que conozcan la dirección IP del controlador Ryu.

2.1.6. VirtualBox

VirtualBox es una aplicación de virtualización multiplataforma, como se aclara en [6]. Esto significa que se instala en computadores existentes basados en Intel o AMD, y está disponible para los sistemas Windows, Mac, Linux y Solaris. Amplía las capacidades del computador existente para que puedan ejecutar múltiples sistemas operativos (dentro de múltiples máquinas virtuales) al mismo tiempo.

Este software es engañosamente simple pero también muy poderoso. Puede ejecutarse desde cualquier lugar, de pequeños sistemas integrados o máquinas de escritorio a implementaciones de centros de datos e incluso entornos en la nube.

En este proyecto, se emplea VirtualBox para crear una máquina virtual con el sistema operativo *Linux* en una distribución de *Debian*, concretamente *Ubuntu* en su versión 16.04.3. De esta forma, una vez se haya completado el trabajo, la máquina virtual podrá ser clonada y empleada en diferentes entornos de prácticas con todo el software necesario ya instalado y listo para el aprendizaje.

2.1.7. iPerf

Aunque no está relacionada directamente con las redes SDN, iPerf es una herramienta para medir activamente el máximo ancho de banda alcanzable en redes IP. Es software opensource y multiplataforma. Fue desarrollado por el *Distributed Applications Support Team (DAST)* en el *National Laboratory for Applied Network Research (NLANR)* y está escrito en C++. Es compatible con varios parámetros relacionados con el timing, buffers y protocolos (TCP, UDP, SCTP con IPv4 e IPv6), como se determina en [7]

El funcionamiento habitual es crear flujos de datos TCP y UDP y medir el rendimiento de la red. iPerf puede funcionar como cliente o servidor y puede medir el rendimiento entre los dos extremos de la comunicación, unidireccional o bidireccionalmente.

En este trabajo, se aprovecha iPerf para comprobar por un lado si el balanceo de carga se está realizando correctamente y por otro el ancho de banda de la conexión, creando en un extremo un cliente en el otro un servidor. De esta forma, si existen varios caminos entre ellos, se puede comprobar si se ha balanceado la carga entre dichos caminos mediante el número de paquetes enviados por cada uno de ellos.

2.2. Ingeniería de tráfico

A continuación se definen los escenarios de aplicación que existen y cómo se pueden mejorar utilizando SDN. Se desarrollan en este trabajo dos casos de uso: Un balanceador de carga multicamino y una monitorización del tráfico con una cierta QoS. También, se explica que es la ingeniería de tráfico ya que para poder realizar estos casos de uso, son necesarias ciertas técnicas especializadas.

2.2.1. Definición de ingeniería de tráfico

la Ingeniería de tráfico, (*TE*), es un mecanismo importante para optimizar el rendimiento de la red de datos, analizando dinámicamente, prediciendo y regulando el comportamiento de los datos transmitidos por la red, como se manifiesta en [8].

La misma se puede considerar una aplicación de red, en referencia a la figura 2.1, puesto se encontraría en el plano de aplicación. La ingeniería de tráfico estudia la medición y gestión del tráfico de red y diseña mecanismos de enrutamiento

razonables para guiar el tráfico de red a fin de mejorar la utilización de los recursos de red y cumplir mejor los requisitos de la calidad de servicio (*QoS*) de la misma.

En comparación con las redes tradicionales, SDN tiene muchas ventajas para ser compatible con TE debido a sus características distintivas, como el aislamiento de los planos de control y datos, el control centralizado global y la programabilidad del comportamiento de la red.

Si bien es cierto que SDN presenta ventajas, no significa que hasta ahora las técnicas de TE no fueran eficientes. Existen gran variedad de técnicas TE, la mayoría de ellas implementadas sobre redes MPLS (Multiprotocol Label Switching). Dos de las arquitecturas que se emplean son los servicios integrados (Integrated Services, IntServ) y los servicios diferenciados (Differentiated Services, DiffServ), como señala [9].

Por un lado, los servicios integrados constituyen una arquitectura cuyo cometido es gestionar los recursos necesarios para garantizar calidad de servicio en una red. Puede ser utilizado por ejemplo para enviar vídeo y sonido al receptor sin interrupción.

Por otro lado, los servicios diferenciados constituyen una arquitectura que especifica un mecanismo simple y escalable para clasificar y gestionar el tráfico de red y proporcionar calidad de servicio en redes IP modernas. Diffserv puede, por ejemplo, utilizarse para proporcionar baja latencia al tráfico de red crítico como voz o streaming, al tiempo que proporciona un servicio simple best-effort a servicios que no críticos, como el tráfico web o las transferencias de archivos.

Estos servicios están relacionados directamente con la siguiente subsección, la calidad de servicio, ya que se pretende con dichas arquitecturas garantizar una calidad determinada para los diferentes tipos de usuarios.

2.2.2. Quality of service (*QoS*)

La calidad de servicio en redes o QoS, es un conjunto de estándares y mecanismos de toda la industria para garantizar un rendimiento de alta calidad para aplicaciones críticas. Mediante el uso de mecanismos QoS, los administradores de red pueden usar los recursos existentes de manera eficiente y garantizar el nivel de servicio requerido sin expandir de forma reactiva ni aprovisionar en exceso o sobredimensionar sus redes.

El concepto de calidad de servicio es aquel en el que los requisitos de algunas aplicaciones y usuarios son más críticos que otros, lo que significa que parte del tráfico necesita un tratamiento preferencial. Ese es el objetivo de QoS, proporcionar un servicio de entrega preferencial para las aplicaciones que lo necesitan asegurando un ancho de banda suficiente, controlando la latencia y reduciendo la pérdida de datos, como se explica en [10].

En las redes tradicionales, es necesario configurar cada router para que utilice estos mecanismos de QoS. Es algo aparentemente sencillo si se trata de una red de área local como podría ser una vivienda: con un sólo router con unos pocos puertos RJ-45. Sin embargo, si se trata de una red de mayor se presentan problemas de implementación de QoS.

Un ejemplo clásico de calidad de servicio en redes es la solución al problema del pez, mostrado en la figura 2.4, donde los routers r6 y r7 formarían la cola y el router r8 la cabeza del pez.

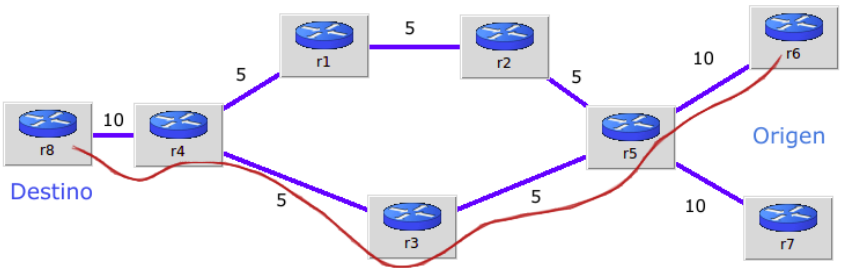

Figura 2.4: Problema del pez.

El problema que surgía en las redes tradicionales en esta topología, es que los protocolos de routing establecen la mejor ruta basándose en parámetros como el coste (indicado con números en la figura 2.4) y envían todo el tráfico por esa ruta. En este caso, todo el tráfico sería enviado por la ruta de menor coste, r6–r5–r3–r4–r8, marcada con la línea roja de la figura.

A la vista salta el pésimo aprovechamiento de la red, que es solucionado en las redes tradicionales de diferentes formas. Una de ellas es la utilización de MPLS-TE, que establecería dos túneles, uno por cada ruta y conseguiría aprovechar de manera óptima la red.

2.2.3. Balanceo de carga

Hoy en día, nuestras redes tienen que manejar una gran cantidad de tráfico, atender a miles de clientes y cumplir los requisitos impuestos. Es muy difícil para un

solo servidor soportar una carga tan grande. La solución es usar múltiples servidores con un balanceador de carga que actúa como interfaz.

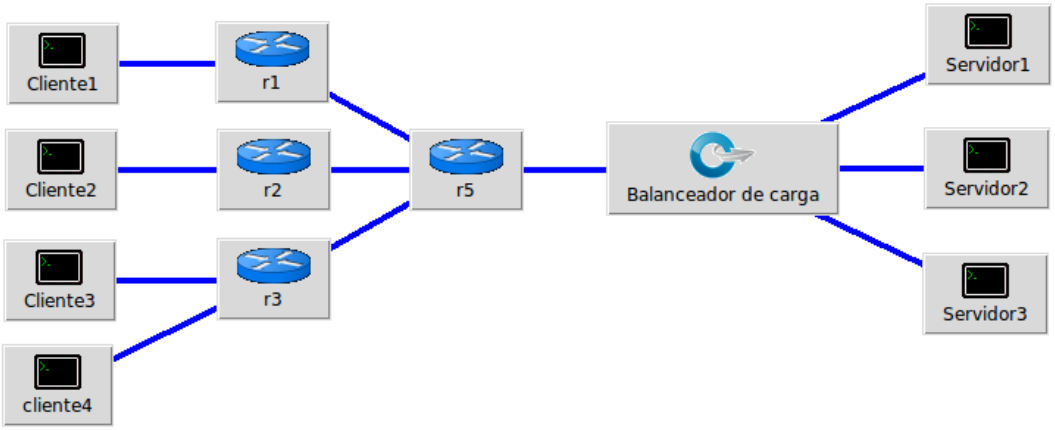

Figura 2.5: Ejemplo de un balanceador de carga.

De esta forma, y como se muestra en la figura 2.5, cuando los clientes envíen sus peticiones el balanceador de carga las recibirá y reenviará a los diferentes servidores según la estrategia de balanceo de carga.

De las estrategias más comunes es la llamada *Round Robin*, que consiste en distribuir las peticiones secuencialmente entre el grupo de servidores. Otra de ellas es conocida como *Least Connections*, donde se envía la nueva petición al servidor con la menor cantidad de conexiones con clientes en ese momento. También se usa la estrategia de *IP hash* en la que la dirección IP del cliente es utilizada para determinar que servidor le atenderá.

El balanceador usa un hardware dedicado que es costoso e inflexible. Los balanceadores de carga actualmente contienen pocos algoritmos que se pueden utilizar. Los administradores de red no pueden usar sus propios algoritmos de balanceo puesto que los balanceadores de carga no son programables, están bloqueados por el vendedor (*Vendor locked*).

2.2.4. Mejora de los escenarios Existentes con SDN

Mediante SDN es posible solucionar algunos de los problemas a los que se enfrentan las redes tradicionales.

El balanceador de carga basado en SDN presenta ciertas ventajas en comparación con el método tradicional, puede mejorar efectivamente el rendimiento del balanceador y reducir la complejidad de la implementación. Los balanceadores

de carga SDN son programables y permiten diseñar e implementar una estrategia de equilibrio de carga propia. Otras virtudes del balanceador de carga SDN es que no se necesita hardware dedicado, ahorrando así coste en la red. Un mismo switch se puede convertir en un potente balanceador mediante el uso de controladores SDN.

En referencia a QoS, con SDN se puede tener el control de QoS centralizado, dejando a los routers que se encarguen de sus cometidos y mediante el controlador SDN implementar QoS directamente en los switches. Esto soluciona los problemas ocasionados en una red de mayor tamaño y facilitan la gestión de una forma eficaz de la QoS de una red al completo. Con esta mejora se puede en cualquier momento cambiar las normas de QoS dinámicamente, haciendo así que usar SDN sea realmente útil.

2.3. Routing multicamino con balanceador de carga

Una de las primeras aplicaciones expuestas en el capítulo anterior es el balanceador de carga con routing multicamino, habiéndose creado para su desarrollo un script en *Python*, `multicamino.py` con ese objetivo. Más adelante, se describe como ejecutarlo y se comprueba su funcionamiento en la red.

Este caso de uso es uno de los más comunes e implementados en SDN por lo que no se profundiza en todos los aspectos, sino que se da una visión general de su implementación y funcionamiento. Si se desea más información dirigirse a la bibliografía.

2.3.1. Routing Multicamino

El Routing multicamino es una técnica que explota los recursos de la red mediante la propagación del tráfico desde un nodo de origen a un nodo de destino por medio de múltiples rutas a lo largo de la red.

Esta técnica es empleada en un gran número de propósitos, incluyendo la agregación de ancho de banda, la minimización del retardo de extremo a extremo, el aumento de la tolerancia a fallos, la mejora de la fiabilidad y el balanceo de carga, entre otras.

Existen tres elementos fundamentales en el routing multicamino: El descubrimiento de rutas (*Path discovery*), la distribución del tráfico y el

mantenimiento de rutas, como se explica en [11]. Fundamentalmente este trabajo se centra en el descubrimiento de rutas. La distribución del tráfico se hace de acuerdo a la aplicación implementada, un balanceador de carga, con lo que el tráfico se va a repartir por igual a través de las rutas encontradas y considerando que los caminos existentes se mantienen fijos durante todo el experimento.

2.3.2. PathFinding Algorithms

Los algoritmos *PathFinding* o de *Pathing* son los encargados de obtener la ruta más corta entre dos puntos. Este campo de investigación se basa en gran medida en el algoritmo de *Dijkstra*, capaz de encontrar el camino más corto en un grafo con pesos.

Este es el ejemplo típico de problema de la ruta más corta, en la teoría de grafos, que examina como identificar el camino que mejor se ajusta a ciertos criterios (el más corto, el más barato, el más rápido, etc.)

Dos de los algoritmos más comunes son los conocidos como BFS (Breadth-first Search Algorithm) y DFS (Depth-first Search Algorithm) que, a diferencia del algoritmo de *Dijkstra*, en su búsqueda agotan todas las posibilidades. Para ello, comenzando en el nodo de origen, iteran sobre todos los caminos posibles hasta que alcanzan el nodo de destino. Estos algoritmos se ejecutan en tiempo lineal o según la notación Big-O:

$$O(N + E) \tag{2.1}$$

donde:

O = Notación Big-O.
N = Número de nodos en la red.
E = Número de enlaces entre los nodos de la red.

Ambos algoritmos, recorren el grafo en su totalidad con el fin de conseguir encontrar los caminos existentes. Hay que tener en cuenta que recorrer un grafo consiste en "visitar" cada uno de los nodos a través de los enlaces de los mismos. Se trata de realizar recorridos de grafos de manera eficiente. Con este objetivo, se pondrá una marca en cada nodo en el momento en que es visitado, de tal manera que, inicialmente, no estará marcado ningún nodo del grafo.

A continuación se detalla el proceso de los dos algoritmos (BFS y DFS) en grafos no dirigidos, observando que en el caso de ser dirigido el proceso es análogo, sólo cambia el significado del concepto de *adyacencia*.

2.3.2.1. Breadth-first Search Algorithm (*BFS*)

Se trata de un algoritmo basado en la búsqueda en anchura de un grafo, en este caso, no dirigido, *G*. Por ello, se comienza en el nodo de origen y se visitan todos sus vecinos. A continuación para cada uno de los vecinos se explora sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo, como explica [12].

El pseudocódigo de este algoritmo explica perfectamente como realiza el recorrido en amplitud del grafo utilizando una cola en la que se van añadiendo los nodos a visitar:

```
BFS (G, s)                  // Donde G es el grafo y s el nodo origen
  let Q be queue
  Q.enqueue( s )            // Insertamos s en la cola
  mark s as visited.
  while ( Q is not empty )
                            // Eliminamos el nodo v de la cola
    v  =  Q.dequeue( )
                            // Visitamos todos los vecinos de v
    for all neighbours w of v in G
      if w is not visited
        Q.enqueue( w )      // Guardamos w en Q para visitar luego sus vecinos
        mark w as visited.
```

De forma gráfica, En la figura 2.6 se observa como itera el algoritmo (de izquierda a derecha y de arriba abajo) para un determinado grafo bidireccional siendo el nodo de origen *A* y el nodo de destino *G*. Marcándose en azul oscuro los nodos visitados y en color azul claro los nodos adyacentes a los nodos ya visitados.

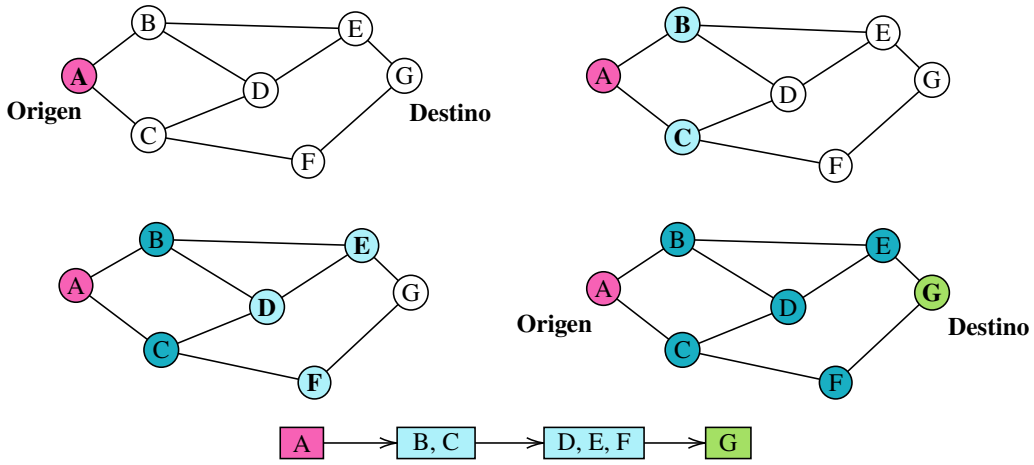

Figura 2.6: Iteraciones del algoritmo BFS.

2.3.2.2. Depth-first Search Algorithm (*DFS*)

A diferencia del BFS, este algoritmo se centra en la búsqueda en profundidad del grafo. Es el elegido en este trabajo para cumplir la tarea de encontrar todos los caminos existentes entre dos nodos.

DFS explora el primer vecino del nodo origen, después el primer vecino del vecino del origen, así hasta encontrar el más profundo, antes de retroceder con el propósito de encontrar otros nodos, haciendo uso de una pila (*stack*), como se detalla en [13].

```
DFS (G, s)              // Donde G es el grafo y s el nodo origen
  visited = set( )      // Serie no repetida de elementos
  stack  =  [ s ]       // inicializamos la pila con s
  while ( stack is not empty )
                        // Guardamos y eliminamos el ltimo elemento de la pila
    node  =  stack.pop( )
    if node not in visited:
                        // Ponemos como visitado el nodo
      visited.add( node )
                        // incorporamos los nodos adyacentes no visitados
      stack.extend( G[node] - visited )
  return visited
```

El pseudocódigo del algoritmo usa una estructura de datos de tipo pila. En la siguiente figura se muestra como opera en profundidad el algoritmo. teniendo que repetir este proceso hasta que la pila se haya vaciado (*i.e. todos los nodos hayan sido visitados*).

En la figura 2.7 se muestra las iteraciones del algoritmo DFS para un determinado grafo bidireccional, siguiendo el mismo esquema que en la figura 2.6. No están plasmadas todas las iteraciones que realizaría el algoritmo, sino que solo aparecen las iteraciones hasta conseguir descubrir dos rutas al destino (A–C-F-G y A–B–D–C–F). El resto de iteraciones se realizan de forma análoga, continuando con el nodo E, que sería el único restante en la pila. Es por el uso de la pila, que este algoritmo es muy útil, especialmente en el routing multicamino, porque se puede modificar el algoritmo para encontrar todas las rutas posibles entre dos nodos. Por eso mismo, es el que se emplea dentro del script `multicamino.py` que permite el balanceo de la carga en la red encontrando diferentes rutas al mismo destino.

Poniéndolo en contexto, con esas modificaciones del algoritmo, se es capaz de encontrar todos las posibles rutas entre dos nodos (switches) en una red. El siguiente código es parte de `multicamino.py` y contiene el algoritmo DFS.

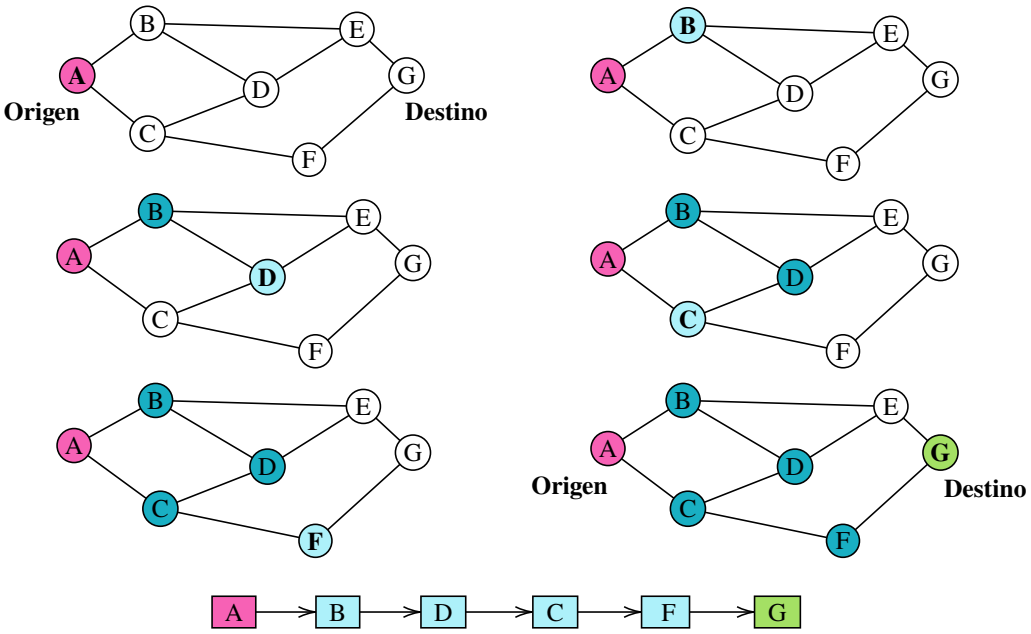

Figura 2.7: Iteraciones del algoritmo DFS.

```
49  def get_paths(self, src, dst):
50    '''
51    Conseguir todas las rutas de src a dst usando el algoritmo DFS
52    '''
53    if src == dst:
54      '''El host destino se encuentra en el mismo switch'''
55      return [[src]]
56    paths = []
57    stack = [(src, [src])]
58    while stack:
59      (node, path) = stack.pop()
60      for next in set(self.adjacency[node].keys()) - set(path):
61        if next is dst:
62          paths.append(path + [next])
63        else:
64          stack.append((next, path + [next]))
65    print "Caminos disponibles de ", src, " a ", dst, " : ", paths
66    return paths
```

El algoritmo se ha implementado en una función, `get_paths`. Recibe como argumentos de entrada el nodo de origen (*`src`*) y el nodo de destino (*`dst`*), *`self.adjacency`* contiene la matriz de adyacencia del grafo (nuestra red). La función devuelve una lista con las diferentes rutas encontradas entre los dos nodos.

2.3.3. Cálculo del coste por camino

El algoritmo DFS devuelve una lista con las diferentes rutas, pero sin peso. Es por ello que es necesario medir el coste de los caminos o rutas. Existe una gran

diversidad de métodos que permiten su medición, dependiendo de los requisitos de la red, como por ejemplo, poner un coste más alto si una ruta tiene menor ancho de banda o pasa por más elementos de red (más saltos), pero en esta ocasión, se calcula el coste de las rutas siguiendo estos pasos:

1. Calcular todos los costes de enlaces[1] que haya en la ruta.
2. Calcular el coste total de la ruta.

Para el primer paso, se calcula el coste de la misma forma que en el protocolo *OSPF, (Open Shortest Path First)*, protocolo de red muy conocido y usado en el encaminamiento jerárquico, como se explica en [14]. *OSPF* se basa en la ecuación de la izquierda, y, con el objetivo de conocer el ancho de banda del enlace, se ha decidido que sea el mínimo ancho de banda entre los switches adyacentes (ancho de banda de las interfaces que los comunican):

$$Cost(l) = \frac{BW_{Reference}}{BW(l)} \quad ; \quad BW(l) = \text{mín } (BW_{Switch1}, BW_{Switch2}) \tag{2.2}$$

donde para un enlace,*link*, l:

$Cost(l)$ = Coste de enlace.
$BW_{Reference}$ = Ancho de banda de referencia, por defecto 1 Gbps.
$BW(l)$ = Ancho de banda del enlace.
BW_{Switch} = Ancho de banda de la interfaz del switch.

Por último, se calcula el coste total de la ruta, sumando los costes de los enlaces que conforman la ruta, calculados en el primer paso. Por ejemplo en una ruta con 5 enlaces,se deben sumar esos 5 costes de los enlaces para conseguir el coste final de la ruta.

Se llevan a cabo cada uno de estos pasos dentro de una función del script `multicamino.py`. El primer paso en la función `get_link_cost` que recibe como argumentos los dos switches vecinos y devuelve el coste de ese enlace. El segundo paso, se encuentra desarrollado en la función `get_path_cost` que recibe como argumento de entrada la ruta de la que se quiere calcular el coste. En la siguiente página se muestra el código de las dos funciones.

[1] **Coste de enlace –** Coste entre dos switches adyacentes/vecinos.

```
68  def get_link_cost(self, s1, s2):
69    '''
70    Calcular el coste de enlace entre dos switches
71    '''
72    e1 = self.adjacency[s1][s2]
73    e2 = self.adjacency[s2][s1]
74    bl = min(self.bandwidths[s1][e1], self.bandwidths[s2][e2])
75    ew = REFERENCE_BW/bl
76    return ew
77
78  def get_path_cost(self, path):
79    '''
80    Calcular el coste de la ruta/camino
81    '''
82    cost = 0
83    for i in range(len(path) - 1):
84      cost += self.get_link_cost(path[i], path[i+1])
85    return cost
```

En la primera función, *self.bandwidth* contiene el ancho de banda de la interfaz del switch. A parte, *REFERENCE_BW* es el ancho de banda de referencia (constante) usado en el coste de OSPF. Al igual que en el código del algoritmo DFS, *self.adjacency* es la matriz de adyacencia de la red.

2.3.3.1. Bucket weight en OpenFlow

Se ha conseguido hasta el momento desarrollar el algoritmo DFS, que encuentra las diferentes rutas entre dos nodos y también cómo calcular el coste de esas rutas, sin embargo en lo que respecta al protocolo OpenFlow, este se basa en el uso de un controlador SDN (Ryu en este caso) configurado para controlar la red de switches OpenFlow.

Existen dos formas de controlar la red: hacer que el controlador tome todas las decisiones de routing, o enseñar a los switches a tomar esas decisiones. Nos vamos a centrar en la segunda ya que es la mejor opción si se habla de una red permanente, evitando que los switches pregunten al controlador cada vez que les llega un paquete. De esta forma, se enseña a los switches instalando reglas OpenFlow (*OpenFlow rules*) en ellos, reglas sobre qué hacer cuando lleguen determinados paquetes. La intención es instalar los caminos encontrados con el algoritmo DFS y explorar como convertir esos costes de los caminos previamente calculados en reglas OpenFlow con motivo de determinar qué porcentaje del tráfico hay que enviar por cada ruta (i.e. mandar más paquetes por las rutas de menor coste y menos por las de un coste mayor). En `multicamino.py` se encuentra la función `install_path` que se encarga de eso mismo, instalar los caminos, como se explica a continuación.

1. `Lista las rutas disponibles de origen a destino.`

2. `Itera mediante un bucle a través de todos los switches que contengan un enlace en alguna de las rutas.`

   (a) `Lista todos los puertos del switch que contengan un enlace.`

   (b) `Si múltiples puertos del switch contienen un enlace, crea un` *`Group table flow`* `con el tipo` *`Select`* `y sino instala un flow normal.`

El último paso es el más interesante, pero es preciso explicar antes el concepto de grupos (*Groups*). Incorporados por el protocolo OpenFlow en sus últimas versiones, representan una serie de puertos como una única entidad para el envío de paquetes.

Con el fin de entender el concepto de grupo y *group table*, en la figura 2.8 se muestra la estructura de un grupo OpenFlow. Como se puede observar, un grupo está formado por varios campos: un identificador de grupo (`ID`), un tipo (`Type`) y contadores (`Counters`).

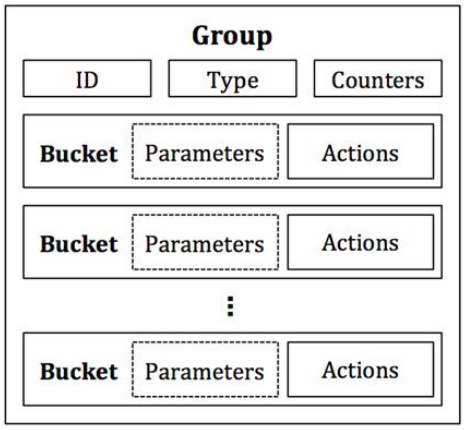

Figura 2.8: Grupo OpenFlow.

Se distinguen varios tipos de grupos, a fin de representar diferentes conceptos como el multicasting (tipo *All*) o el multicamino (tipo *Select*). Cada grupo consta de una *Group table*, formada de una serie de entre 1 y 32 entradas llamadas *buckets*, mostrados en la figura 2.8, y cada uno de esos *buckets* contiene una serie de acciones (`Actions` en la figura 2.8) que se aplican antes de enviar los paquetes por el puerto de salida de ese *bucket*.

Así, teniendo agrupados los puertos, una entrada (un flow) puede apuntar a todo un grupo, siendo esta habilidad la que permite que OpenFlow represente diferentes métodos de envío.

En concreto, el más interesante es el grupo de tipo *Select*, que es el que permite el multicamino debido a que ejecuta solo un *bucket* en el grupo. Los *buckets* tienen varios campos que se deben especificar: *bucket weight, watch port, watch group* y

actions. Así, *watch port* contiene el puerto de salida por el que se enviará el paquete si se ejecuta ese *bucket*, siempre y cuando ese puerto esté abierto. Por su parte, el campo *bucket weight* contiene un número entero que nos indicará que *bucket* elegir.

Cuando nos referimos al coste o peso de un enlace, donde se quiere usar primero el camino mas corto, cuanto menor sea el coste, mejor es el camino. En el contexto de *buckets* en OpenFlow ocurre lo contrario, se ejecutará el *bucket* con mayor *bucket weight*. Por esto mismo, cuanto más grande sea el número en ese campo, mejor.

Es necesario solucionar el problema que surge, ajustar el criterio de los *bucket weights* con los costes de las rutas. De forma muy sencilla, se muestra en la siguiente ecuación como resolverlo. Esencialmente en la ecuación, se busca el ratio del peso del camino con respecto al peso total de todos los caminos disponibles.

$$bw(p) = \left(1 - \frac{Cost(p)}{\sum_{i=0}^{i<n} Cost(i)}\right) \times 10 \tag{2.3}$$

donde para una ruta, *path*, p:

$bw(p)$ = *Bucket weight* de la ruta, $0 \leq bw(p) < 10$.
$Cost(p)$ = Coste de la ruta, calculado previamente (*OSPF*).
n = Número total de rutas encontradas.

Expresando de esta forma el *bucket weight*, siempre tendrá un valor entero entre 0 y 9. Se verá en el siguiente capítulo su implementación y funcionamiento. Debajo se muestra parte del código de la función `install_path`.

```
177 buckets = []
178
179 for port, weight in out_ports:
180   bucket_weight = int(round((1 - weight/sum_of_pw) * 10))
181   bucket_action = [ofp_parser.OFPActionOutput(port)]
182   buckets.append(
183     ofp_parser.OFPBucket(
184       weight = bucket_weight,
185       watch_port = port,
186       watch_group = ofp.OFPG_ANY,
187       actions = bucket_action
188     )
189   )
```

En el código se especifican los *buckets*, se ha desarrollado la ecuación anterior y es aplicada en cada puerto de salida que contenga un enlace. Por tanto, si un switch tuviera dos puertos con enlace, se pondrá cada uno de esos puertos como un *bucket* en la *group table*.

Con todo esto, ya es posible implementar el routing multicamino al haber conexionado por un lado la parte de OpenFlow y por otra el routing.

Capítulo 3

Implementación

Teniendo en consideración todos los teóricos explicados anteriormente, se procede a la implementación de los dos casos de uso. Se dedica una sección a cada uno, conteniendo ambos un desarrollo similar. Debido a su similitud se profundizará más en el primer caso de uso, mientras que en el segundo se darán por conocidos aquellos aspectos analizados en el anterior.

3.1. Entorno de desarrollo

Es necesario recordar que se desarrolla la implementación en una máquina virtual ejecutada desde VirtualBox. Esta máquina corre un sistema operativo *Linux* en una de sus distribuciones *Debian*, *Ubuntu 16.04.3*.

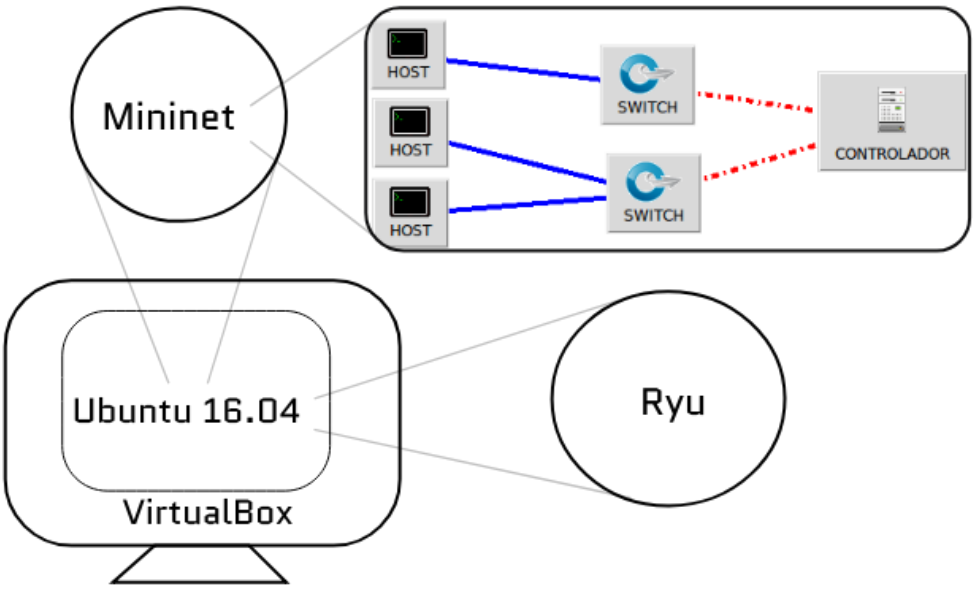

Figura 3.1: Entorno de desarrollo.

En la figura 3.1 se muestra un esquema del entorno de desarrollo del proyecto. Desde el punto de vista más amplio se encuentra el PC o máquina desde donde se realiza el experimento. Dentro de este se ejecuta una Máquina virtual (*Ubuntu 16.04*) de VirtualBox, como se ha comentado anteriormente. Y dentro de está máquina virtual se encuentra el controlador Ryu y Mininet. Por último, en Mininet se implementa la topología, siendo la de la figura 3.1 un mero ejemplo.

Una vez dentro de la máquina virtual, se debe instalar *Ryu*. Para lo cual se necesita tener instalado *Python*, que viene por defecto con el *S.O*. Para instalar Ryu se abre una terminal y existen dos opciones para instalarlo: hacerlo desde el administrador de paquetes de Python o desde el código fuente. De la primera forma sería:

```
tfg@tfgVM:~$ pip install ryu
```

y de la segunda, desde el código fuente:

```
tfg@tfgVM:~$ git clone git://github.com/osrg/ryu.git
tfg@tfgVM:~$ cd ryu; pip install
```

Se comprueba que la instalación se ha realizado correctamente ejecutando Ryu. Deberá devolver por pantalla lo siguiente.

```
tfg@tfgVM:~$ ryu-manager
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of
OFPHandler
```

Lo último antes de comenzar con cada caso de uso es instalar *Mininet*, imprescindible para la implementación.

```
tfg@tfgVM:~$git clone git://github.com/mininet/mininet
```

Este comando clonará el repositorio en la máquina virtual. En él constará la última versión de Mininet. En este trabajo se ha utilizado la versión `2.2.1`, es recomendable usar la misma para reproducir este trabajo, pero si esta versión no fuera la última en un momento dado, es posible elegir la que deseemos: se listan las versiones disponibles (`git tag`) y se elige la deseada(`git checkout`).

```
tfg@tfgVM:~$ cd mininet && git tag
tfg@tfgVM:~$ git checkout -b 2.2.1 2.2.1
tfg@tfgVM:~$ cd ..
```

Mininet se instala usando la opción `-a` que considera todas las opciones de Mininet, incluido dependencias y extras como *Wireshark* para *OpenFlow*:

```
tfg@tfgVM:~$ mininet/util/install.sh -a
```

Después de la instalación, se puede comprobar la funcionalidad de Mininet:

```
tfg@tfgVM:~$ sudo mn --test pingall
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
.
.
.
*** Stopping 2 hosts
h1 h2
*** Done
completed in 0.678 seconds
```

Aparecen los logs de Mininet que, por simplicidad, están mostrados solo los iniciales y finales.

Llegados a este punto, la máquina virtual está preparada para la implementación de cada ejemplo. Se recomienda clonar la máquina virtual para que, en caso de que surja algún fallo no localizable en la implementación, poder volver al inicio con esa copia de la máquina virtual.

3.2. Balanceador de carga multicamino

Aunque se ha comentario con anterioridad, se pretende en este caso de uso implementar un balanceador de carga multicamino por medio del script `multicamino.py`. Este script es realmente una aplicación por encima de OpenFlow, que utilizando SDN consigue balancear la carga de una red entre dos puntos en función del coste de sus rutas.

3.2.1. Definición del escenario de red

Se propone la red de la figura 3.2 como ejemplo para implementar el routing multicamino con balanceo de carga. La figura es una captura de *Miniedit*, que a continuación se enseña como manejar. Para el ejemplo, se va a suponer que el host h1 es el servidor y se hará que actué como tal mientras que el host h2 será el cliente. Una vez esté implementado el sistema, mediante *iPerf* se enviarán datos desde el cliente hacia el servidor para comprobar el balanceo de carga. Por tanto, el host h2 enviará todo el tráfico al switch s5 y este dirigirá un porcentaje de los paquetes hacia

el switch s2 y otro hacia el s4. A su vez, c0 es el controlador, conectado con los 5 switches de la topología.

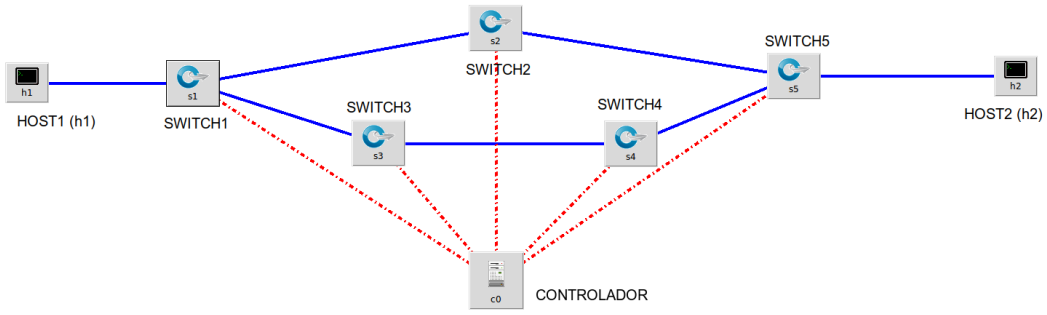

Figura 3.2: Topología elegida para el routing multicamino con balanceo de carga.

Se puede observar claramente que el camino más corto de h2 a h1 es el que pasa por los switches s5−s2−s1 mientras que el camino que sigue s5−s4−s3−s1 incorpora un salto más y será más costoso, por lo que se debería enviar menos tráfico por ese camino.

Primero hay que crear la topología y para lo cual existen dos opciones, crear un script en python haciendo uso de los módulos de Mininet o emplear el entorno gráfico que nos proporciona Mininet, Miniedit. A continuación se va a explicar como crear el script y como ejecutarlo y así como crear la topología en Miniedit.

La ventaja del script radica en que no se necesita entorno gráfico para ejecutarlo, siendo válido para diferentes versiones de Mininet. Por su parte, la ventaja de Miniedit es que proporciona una visión de la red y hace que el aprendizaje sea más fácil sobre todo para los menos familiarizados con Mininet.

El script de la topología que se va a diseñar, se encuentra en el Anexo B bajo el nombre `topologia_multicamino.py`. No se profundiza en su contenido ya que al final se diseña la topología desde Miniedit. Aún así, la forma de ejecutarlo sería:

```
tfg@tfgVM:~$ sudo -E python topologia_multicamino.py
```

La opción `-E` permite preservar las variables de entorno. Una vez ejecutado la red estará en funcionamiento, pero no habrá conexión entre los hosts (los *pings* fallarán) porque no está listo el controlador Ryu todavía.

Dejando a un lado el script, es interesante ver que pasos seguir para crear la topología en Miniedit. se abre una terminal y se ejecuta Miniedit. Para ello es necesario situarse en el directorio dónde se encuentre el fichero `miniedit.py`:

```
tfg@tfgVM:~$ sudo python mininet/examples/miniedit.py
```

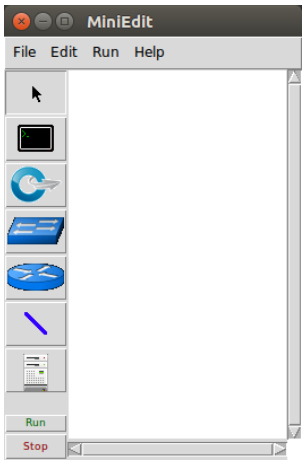

Figura 3.3: Miniedit.

Se iniciará Miniedit en una nueva ventana tal y como se muestra en la figura 3.3. Se distinguen tres zonas principales dentro de Miniedit: el menú de opciones de arriba a la izquierda, la barra lateral con los elementos de red y el workspace (la zona blanca de la ventana, donde crearemos la topología). Además, en la esquina inferior izquierda se divisan dos botones (Run y Stop). La barra lateral permite seleccionar elementos de red y arrastrarlos al workspace; se pueden diferenciar, en orden descendente: host, switch (OpenFlow), legacy switch, legacy router, enlace de red y controlador.

En el menú de opciones es posible configurar Miniedit, guardar la topología con extensión (`.mn`) o como script, cargar topologías anteriormente guardadas y poner en funcionamiento la red, entre otros. Antes de nada, hay que ir a `Edit` → `Preferences` y configurar como muestra la figura 3.4, especificando la versión 1.3 de OpenFlow y que se inicie una terminal de Mininet al iniciar la red.

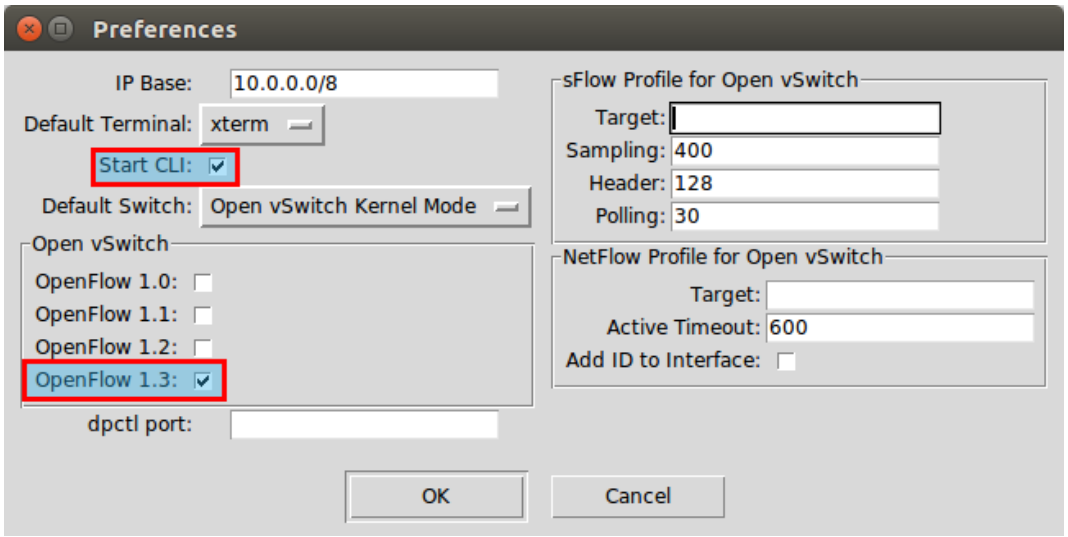

Figura 3.4: Preferencias de Miniedit.

Lo siguiente es arrastrar los host y switches al workspace y unirlos mediante el enlace de red. También es necesario arrastrar un controlador y conectarlo con los switches. Se observa que los enlaces entre controlador y switches son de otro color, puesto que no están físicamente unidos. Se recuerda que la topología que se quiere crear es la que enseña la figura 3.2.

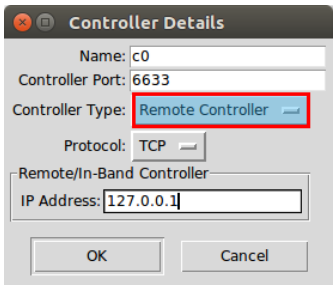

Figura 3.5: Detalles del controlador.

Una vez con la topología diseñada como en la figura 3.2 y antes pulsar el botón Run, hay que configurar el controlador, por lo que se hace click derecho encima del controlador de la red, seleccionamos en `Properties` y aparecerán los detalles del controlador como se muestran en la figura 3.5 de la izquierda. En esta ventana se pueden configuran los aspectos generales del controlador: cambiar el nombre del controlador, especificar su puerto, el tipo de controlador, el protocolo que usara para comunicarse con los switches OpenFlow (TCP/SSL) y la dirección IP en la que se encuentra.

En nuestro caso el controlador Ryu que correrá en la misma máquina virtual lo que significa que la dirección IP del controlador será 127.0.0.1 y escuchará en el puerto 6633. Estos valores vienen por defecto en los detalles del controlador así que no hay necesidad de cambiarlos. Solo es necesario modificar el `Controller type` de *OpenFlow Reference* a *Remote Controller*. Esto se debe a que Mininet incorpora un controlador OpenFlow y por eso la opción *OpenFlow Reference* está pre-configurada.

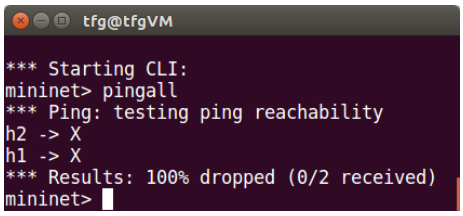

Figura 3.6: Terminal de Mininet y ping fallido.

Ya esta todo preparado para iniciar la red, pulsando en el botón `Run`. Entonces en la terminal desde la que se corrío Miniedit, se iniciará una terminal de Mininet que nos permitirá ejecutar comando de Mininet más adelante. Es interesante probar a hacer un ping en la red, que fallará porque no tenemos el controlador preparado. Este proceso queda ilustrado en la figura 3.6, en la que se ve que el 100 % no han sido recibidos y por ello descartados.

El siguiente paso es ejecutar el controlador con la aplicación, `multicamino.py`.

3.2.2. Inicio del controlador con el script multicamino

con la red en funcionamiento solo queda iniciar el controlador y añadir la aplicación de multicamino. Para ello, en una nueva terminal, en el directorio donde se encuentre `multicamino.py` se ejecuta:

```
tfg@tfgVM:~$ryu-manager --observe-links multicamino.py
```

La opción `-observe-links` consigue que se muestren por pantalla los logs relacionados con el descubrimiento de enlaces. Ahora ya están las dos partes en funcionamiento: la red y el controlador.

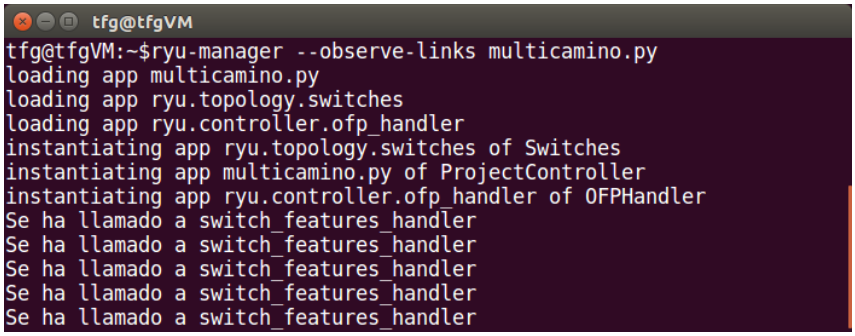

Figura 3.7: Inicio del controlador.

En la figura 3.7 se muestra la ejecución del comando y los logs que se muestran por pantalla, llamando 5 veces *switch_features_handler*, una por cada uno de los 5 switches de la topología.

En este punto, la aplicación no ha descubierto las rutas entre los dos host, pero basta con realizar un ping de un host al otro para que se encuentren las rutas disponibles entre ambos.

Mininet asigna automáticamente una dirección IP a cada host, concediendo al host1 (h1) la dirección 10.0.0.1 y al host2 (h2) la 10.0.0.2 y así sucesivamente en el caso de que la red estuviera formada con más hosts. En esta ocasión h1 es el servidor y h2 el cliente.

Con una terminal *Xterm* en cada host abierta desde Miniedit: click derecho en cada host, opción *Terminal*. Desde la terminal de h2 al hacer un ping a h1 se obtiene:

```
root@tfgVM:~$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=58 time=24.8 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=58 time=24.5 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=58 time=32.3 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=58 time=28.3 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=58 time=31.1 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=58 time=24.7 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=58 time=26.9 ms
^C
--- 10.0.0.1 ping statistics ---
7 packets transmitted, 7 received,
```

```
0% packet loss, time 8004ms
rtt min/avg/max/mdev = 0.094/27.335/81.818/38.525 ms
```

Esto confirma que no han surgido problemas; se han recibido todos los paquetes transmitidos, y el controlador está conectado correctamente con los switches.

Otra vez en la terminal donde habíamos iniciado el controlador, se comprueba que efectivamente se han descubierto las rutas entre el cliente (h2) y servidor (h1), en ambos sentidos, puesto que se suponía que eran enlaces bidireccionales. Además, se ha calculado el coste de cada ruta encontrada y el tiempo en instalar los caminos.

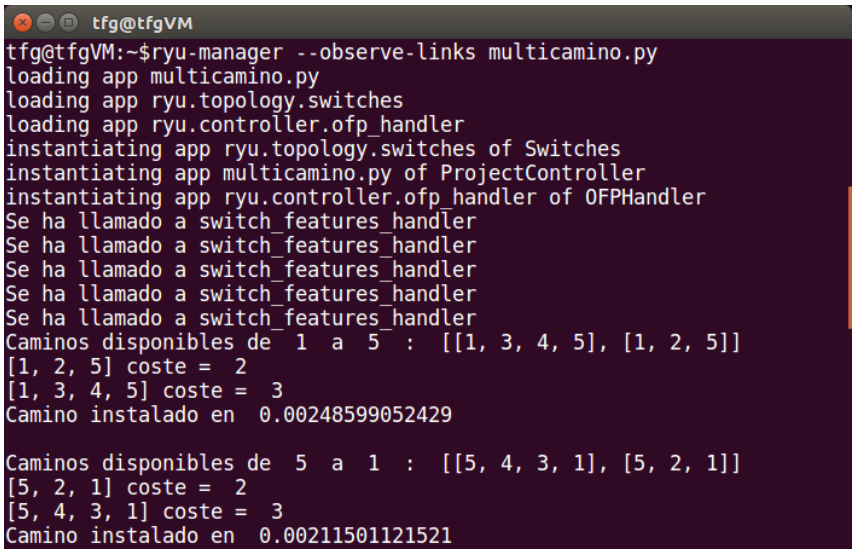

Figura 3.8: Inicio del controlador y descubrimiento de rutas.

Se comprueba en la figura 3.8 que la aplicación ha encontrado las rutas disponibles entre los hosts, de acuerdo con la topología vista en la figura 3.2. Para ir de h2 a h1 existe una ruta que pasa por s5-s2-s1, mientras que otra es s5-s4-s3-s1 como se refleja en la captura en la parte que dice: `Caminos disponibles de 5 a 1`, en donde 5 es el switch s5 y 1 es el switch s1 (igual para el resto de los switches).

Respecto al coste de los enlaces, por temas de simplicidad, en el script `multicamino.py` se ha asignado a todos los enlaces de la topología el mismo ancho de banda `DEFAULT_BW`:

```
    self.bandwidths = defaultdict(
                      lambda: defaultdict(lambda: DEFAULT_BW))
```

Asimismo, se ha otorgado a este ancho de banda de los enlaces el mismo valor que el ancho de banda de referencia `REFRERENCE_BW`. Con esto se logra que el coste del enlace, de acuerdo con la ecuación 2.2, sea de 1 unidad. Por lo tanto el coste total

de la ruta, al ser el sumatorio de los costes de cada enlace, tiene el valor del número de switches que atraviesan los paquetes sin contar el switch s5, porque es donde se ejecuta el multicamino. Es decir, que los costes totales de las rutas (como se enseña en la captura 3.8) son:

Ruta 1 h2−s5−**s2**−**s1**−h1 $\rightarrow Coste_{total} = 2$

Ruta 2 h2−s5−**s4**−**s3**−**s1**−h1 $\rightarrow Coste_{total} = 3$

Desde h1 se enviarán todos los paquetes a s5 ya que es el único switch directamente conectado a h1. En s5 se ejecutará el multicamino, puesto que hay dos puertos con enlaces de diferentes rutas y se enviarán los paquetes en función del coste de cada ruta .

En una nueva terminal en Ubuntu se puede comprobar que las dos rutas encontradas han sido instaladas correctamente en el switch s5 ejecutando el siguiente comando :

```
tfg@tfgVM:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s5

OFPST_FLOW reply (OF1.3) (xid=0x2):

cookie=0x0, duration=4345.812s, table=0, n_packets=4,
n_bytes=392, ip, nw_src=10.0.0.2,
nw_dst=10.0.0.1 actions=group:2742512190

cookie=0x0, duration=4345.812s, table=0, n_packets=2,
n_bytes=84, priority=1, arp, arp_spa=10.0.0.2,
arp_tpa=10.0.0.1 actions=group:2742512190
```

El comando anterior tiene como salida los flows instalados en el switch s5, que está directamente conectado a h2 (10.0.0.2). El objetivo es que h2 consiga llegar hasta h1 (10.0.0.1). Hay que tener en cuenta que se ha modificado el output del comando para que sea más fácil de interpretar. Aunque hay más flows instalados en s5, mostramos solo dos.

Existen varios puertos de salida en s5 para llegar hasta h1 por lo que se ha instalado una acción de grupo, esto es, dirigir los paquetes a un grupo, donde estará la tabla de grupo y demás. Los dos flows anteriores se encargan del matching de paquetes IP y ARP (uno de cada uno), teniendo en el campo de `actions` la redirección al grupo `group:2742512190`. Esto significa que una que las acciones aplicadas a esos flows es una acción de grupo con el id `2742512190`.

Llegados a este punto, es posible observar los grupos existentes en s5:

```
tfg@tfgVM:~$ sudo ovs-ofctl -O OpenFlow13  dump-groups s5

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):

group_id=2742512190,type=select,
bucket=weight:6,watch_port:2, actions=output:2,
bucket=weight:4,watch_port:1,actions=output:1
```

Esta vez no se ha eliminado contenido del output. Esto significa que como cabía esperar tenemos un grupo de tipo *select* con id *2742512190* formado por 2 *buckets* con diferentes *bucket weights*.

El campo *actions* de cada *bucket* indica el puerto de salida (*output:2* y *output:1*) de los paquetes hacia el servidor (h1) Para saber a qué switch está conectado cada puerto y por tanto conocer la ruta completa, es necesario ejecutar el siguiente comando desde la terminal de Mininet:

```
mininet> net ports
h1 h1-eth0:s1-eth3
h2 h2-eth0:s5-eth3
s1 lo: s1-eth1:s2-eth1 s1-eth2:s3-eth1 s1-eth3:h1-eth0
s5 lo: s5-eth1:s4-eth2 s5-eth2:s2-eth2 s5-eth3:h2-eth0
s2 lo: s2-eth1:s1-eth1 s2-eth2:s5-eth2
s3 lo: s3-eth1:s1-eth2 s3-eth2:s4-eth1
s4 lo: s4-eth1:s3-eth2 s4-eth2:s5-eth1
c0
```

El switch s5 está conectado mediante el puerto 1 (*s5-eth1*) al switch s4, por lo que el *bucket* con *actions=output:1* y *weight:4* corresponde a la ***Ruta 2*** que está formada por h2−s5−s4−s3−s1−h1. De igual modo con el puerto 2 (*s5-eth2*), s5 se conecta directamente al switch s2 como parte de la ***Ruta 1*** formada por h2−s5−s2−s1−h1, correspondiéndose con el *bucket* restante del grupo.

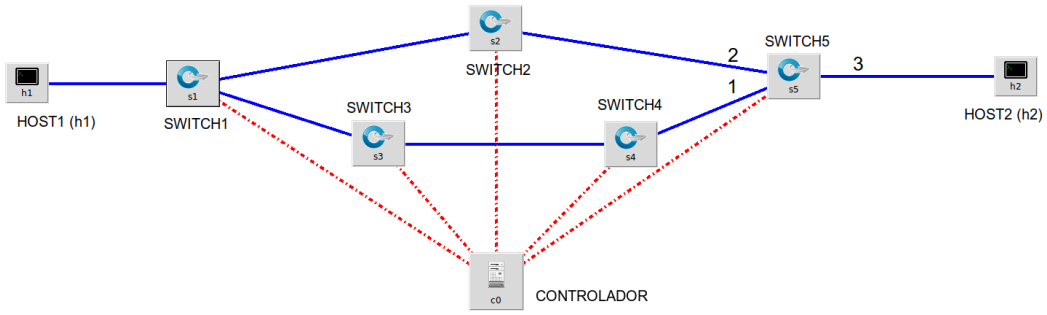

Figura 3.9: Topología Multicamino con los puertos indicados en s5.

En la figura 3.9 se muestra la topología de red con los puertos indicados en s5 para mayor claridad.

Con todo lo anterior, es el momento de revisar si se ha realizado correctamente la conversión del coste total a *bucket weight*, como se expresó en la ecuación 2.3. Se ha comprobado con el penúltimo comando (*dump-groups*) que los *bucket weights* son 4 y 6 para la ruta más y menos costosas respectivamente. Repasando los cálculos:

Ruta 1 h2–s5–**s2**–**s1**–h1 $\rightarrow Coste_{total} = 2 \rightarrow bw(1) = \left(1 - \frac{2}{2+3}\right) \times 10 = 6$

Ruta 2 h2–s5–**s4**–**s3**–**s1**–h1 $\rightarrow Coste_{total} = 3 \rightarrow bw(2) = \left(1 - \frac{3}{2+3}\right) \times 10 = 4$

Tal y como se había planificado, la ruta con menor $Coste_{total}$ es la ruta con mayor *bucket weight*, y en consecuencia, en la que más tráfico se transportará. Por otro lado, la de mayor $Coste_{total}$ es la ruta con menor *bucket weight*.

Con estas operaciones se confirma que se han descubierto las rutas, se han instalado los caminos y que la conversión de $Coste_{total}$ a bw ha sido implementada correctamente. Una vez terminada la parte del multicamino, queda por realizar el balanceo de la carga.

3.2.3. Comprobación del balanceo de carga

en este apartado se va a examinar si se realiza el balanceo de la carga correctamente en la topología elegida. De acuerdo con la figura 3.9, el tráfico que entra por el puerto 3 del switch s5 saldrá en función del *bucket weight* por los puertos 1 y 2.

Mediante el uso de la herramienta iPerf se va a crear tráfico entre el cliente (h2) y el servidor (h1). A continuación, mediante comandos `ovs-ofctl` se monitoriza el envío de ese tráfico en función de los puertos.

Para ello es necesario acceder a las terminales de los hosts, que ya habíamos iniciado en el apartado anterior. En caso de haberlas cerrado, es posible a iniciarlas ejecutando el comando `xterm h1 h2` desde la consola de Mininet.

En las terminales de los hosts no es posible aumentar el tamaño de la fuente por lo que si es necesario es posible abrir una nueva terminal para cada host con el fin de que sea más cómodo a la hora de ejecutar comandos y realizar capturas. En ambos hosts, con el siguiente comando, se abrirá una nueva terminal con la fuente `Monospace` y una altura de 13 puntos.

```
root@tfgVM:~# xterm -fa 'Monospace' -fs 13
```

Con las dos nuevas terminales abiertas se procede a configurar a h1 como servidor y h2 como cliente:

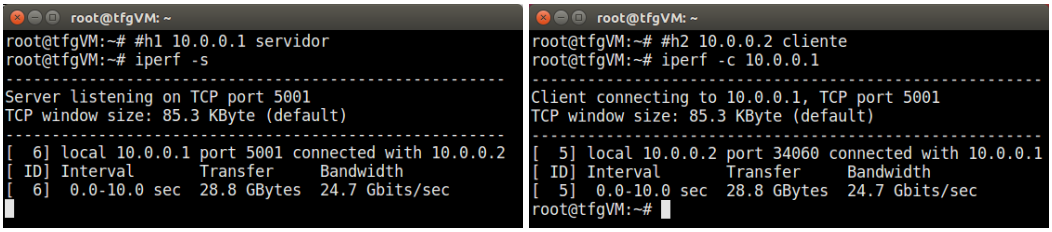

Figura 3.10: Captura del envío de tráfico entre cliente y servidor iPerf.

En la figura 3.10 se muestran sendas capturas de las terminales del servidor (izquierda) y del cliente (derecha). En ellas se observa que se han transferido 28.8 GB de tráfico TCP.

Confirmado que iPerf ha cumplido su cometido, es hora de buscar la información sobre los puertos de s5 por medio del comando *dump-ports* de `ovs-ofctl` en cuyo output figurará la cantidad de paquetes enviados y recibidos por todos los puertos, entre otra información.

```
tfg@tfgVM:~$sudo ovs-ofctl -O OpenFlow13 dump-ports s5

OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports

port  1: rx pkts=8703, bytes=525378, drop=0,
         tx pkts=8702, bytes=525338, drop=0
port  2: rx pkts=131932, bytes=8658484, drop=0,
         tx pkts=549906, bytes=30949444970, drop=0,
port  3: rx pkts=541232, bytes=30948921652, drop=0,
         tx pkts=131934, bytes=8658634, drop=0,
```

Aunque en la primera línea del output se indica que existen 4 puertos y solo se enseñan 3, es debido a que se ha suprimido parte de la salida del comando,el puerto local, para mostrar solamente el contenido realmente interesante.

De acuerdo con la figura 3.9, los paquetes entrantes (*rx pkts*) del puerto 3 se transmiten (*tx pkts*) por los puertos 1 y 2 de tal modo que los recibidos en el puerto 3 deberán estar en un rango similar a la suma de los transmitidos por los otros puertos : $Rx3 \approx Tx2 + Tx1 \rightarrow 541232 \approx 549906 + 8702$. Exactamente como era de esperar, es un resultado similar.

Sin embargo, según esas cifras y teniendo en cuenta que los switches se envían paquetes entre ellos, todo el tráfico ha seguido la misma ruta. ¿Se ha producido

entonces un balanceo de carga? La respuesta es sí, y es que hay un factor que no se ha tenido en cuenta a la hora de realizar este test: en *Open vSwitch 2.4* y en la siguientes versiones para los paquetes del protocolo TCP se realiza un hash con los puertos de origen y destino y se elige el *bucket* teniendo en cuenta el hash. Este hash es simétrico y por tanto no hay ningún cambio en la selección del *bucket* al intercambiar el puerto de origen por el de destino y viceversa.

Resumiendo, en el test que se ha realizado previamente se han enviado los paquetes entre h1 y h2 usando los puertos 5001[h1] y 34060[h2] (figura 3.10). Se ha hecho el hash con esos dos números y se ha transmitido el primer paquete eligiendo el *bucket* con mayor *bucket weight* y se ha asociado el hash a ese bucket. Por eso todos los paquetes con el mismo hash han seguido la misma ruta.

De lo sucedido se obtiene una primera conclusión: no se trata de balancear el número de paquetes, sino el número de clientes. Se ha creado un cliente y se ha realizado el balanceo por la ruta de menor coste. Por lo tanto, es necesario crear más clientes en paralelo con iPerf y ver lo que ocurre. Se reinicia todo el sistema para comenzar de nuevo, aunque no hay problema en seguir realizando el test y después restar los paquetes que se hayan obtenido en la prueba anterior.

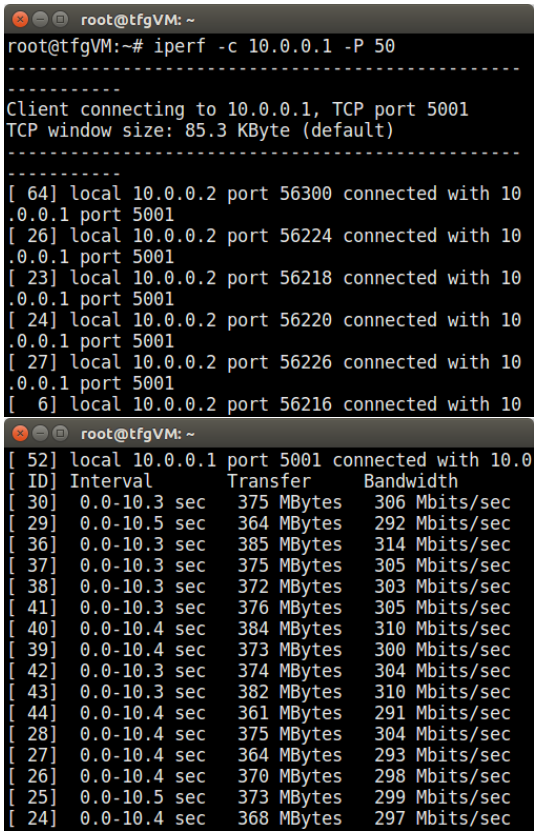

Figura 3.11: Clientes iPerf en paralelo.

En iPerf, existe la opción `-P` o `-parallel`, específica para los clientes, que permite crear varios clientes en paralelo. Sabiendo esto, es posible ejecutar varias veces en el cliente (h2) el comando `iperf -c 10.0.0.1 -P 50` y así conseguir un resultado más preciso.

Se procede a crear 50 clientes, todos conectados, como es lógico, con diferentes puertos y con anchos de banda similares. La figura 3.11 muestra, por un lado, al cliente ejecutando el comando anterior y, por otro lado, al servidor recibiendo el tráfico de los clientes, todos con un ancho de banda en torno a 300 Mbps.

Antes de analizar el resultado de este test, es necesario recordar que las rutas deberían portar teóricamente el siguiente porcentaje del tráfico total:

Ruta 1 h2−s5−**s2**−**s1**−h1 $\longrightarrow bw(1) = 6 \longrightarrow Traffic(1) = \frac{6}{6+4} \times 100 = 60\,\%$

Ruta 2 h2−s5−**s4**−**s3**−**s1**−h1 $\longrightarrow bw(2) = 4 \longrightarrow Traffic(2) = \frac{4}{6+4} \times 100 = 40\,\%$

Después de haber ejecutado los clientes iPerf, con el mismo comando de antes, *dump-ports*, se obtiene la información sobre los puertos de s5:

```
tfg@tfgVM:~$sudo ovs-ofctl -O OpenFlow13 dump-ports s5
OFPST_PORT reply (OF1.3) (xid=0x2): 4 ports
port  1: rx pkts=5376164, bytes=355469919, drop=0,
         errs=0, frame=0, over=0, crc=0
         tx pkts=8304407, bytes=280097226367, drop=0,
         errs=0, coll=0 duration=12317.532s
port  2: rx pkts=8157000, bytes=540397987, drop=0,
         errs=0, frame=0, over=0, crc=0
         tx pkts=13448402, bytes=465062254229, drop=0,
         errs=0, coll=0 duration=12317.533s
port  3: rx pkts=21746866, bytes=745159117812, drop=0,
         errs=0, frame=0, over=0, crc=0
         tx pkts=13530184, bytes=895685867, drop=0,
         errs=0, coll=0 duration=12317.532s
```

Se verifica rápidamente que los paquetes que se reciben en el puerto 3 de s5 son aproximadamente la suma de los paquetes salientes en los puertos 1 y 2 del mismo switch: $Rx3 \approx Tx2 + Tx1 \longrightarrow 21746866 \approx 13448402 + 8304407$.

Volviendo al balanceo, es momento de analizar el porcentaje de tráfico enviado por cada ruta. Para ello:

Ruta 1 h2−s5−**s2**−**s1**−h1 $\longrightarrow Traffic(1) = \frac{Tx2}{Rx3} \times 100 = \frac{13448402}{21746866} \times 100 = 61,83\,\%$

Ruta 2 h2−s5−**s4**−**s3**−**s1**−h1 $\longrightarrow Traffic(2) = \frac{Tx1}{Rx3} \times 100 = \frac{8304407}{21746866} \times 100 = 38,18\,\%$

Este resultado se asemeja considerablemente al porcentaje teórico, y más sabiendo que una pequeña fracción del tráfico no forma parte del tráfico creado con iPerf, sino de la comunicación entre switches.

Como conclusión, el balanceo de carga se ha implementado correctamente y ha permitido adquirir nociones prácticas sobre todos los comandos y aplicaciones empleadas, que son de utilidad en el siguiente caso de uso.

3.3. Monitorización del tráfico con cierta QoS

Hoy en día la mayoría de las redes hacen uso de la calidad de servicio o QoS, por lo tanto, si las redes SDN querían triunfar en el mercado debían ofrecer al menos los mismos servicios y aplicaciones que las redes tradicionales.

Este caso de uso incorpora QoS en una red SDN con la finalidad de mostrar la facilidad de su implementación en las redes definidas por software. Se pretende enseñar cómo, mediante QoS, se pueden transferir los datos de una red en función de la prioridad basada en los tipos de datos, y reservar ancho de banda para una comunicación con el objetivo de comunicar dos puntos con un constante ancho de banda en la red.

3.3.1. Definición del escenario de red

Con el fin de darle realismo, se va a suponer que existen varios tipos de tarifas móviles en una compañía. Una de ellas será la tarifa *estándar* y otra la tarifa *premium*. Las características principales de cada tarifa serán:

Estándar - Si el cliente contrata la tarifa *estándar*, dispondrá de 2 MB de datos a una velocidad de 1 Mbps. Si agota los 2 MB se le reducirá la velocidad el 80 % (i.e. 200 Kbps) para el siguiente 1 MB. Si el cliente consigue gastar los 3 MB, se quedará sin poder transmitir ni recibir datos (i.e. la velocidad será 0 Kbps).

Premium - Si por el contrario, el cliente contrata la tarifa *premium*, gozará de datos ilimitados durante todo el mes, con una velocidad de transmisión mínima garantizada de 2 Mbps.

El caso implementado se centra en la tarifa estándar ya que es la más compleja en cuanto a QoS. La tarifa *premium* se implementaría de forma análoga, por lo que se explicarán algunos conceptos de la misma.

La tabla 3.1 muestra las fases que se atraviesan en las tarifas *estándar* y *premium* en función de los datos consumidos. En la tarifa *premium* se observa que no existen tres fases como tal, porque siempre se mantienen las mismas condiciones. Sin embargo, en la tarifa *estándar* sí que hay tres fases o etapas según se van consumiendo datos.

Tabla 3.1: Diferentes tarifas móviles y sus fases en función del consumo de datos.

	Fase 1		Fase 2		Fase 3	
Tarifa	Datos	Tasa	Datos	Tasa	Datos	Tasa
Estándar	2 MB	1 Mbps	1 MB	0.2 Mbps	–	–
Premium	∞	2 Mbps	∞	2 Mbps	∞	2 Mbps

Nota: Las tasas y datos han sido inventados para ejemplificar el caso de uso. Los datos son del orden de los MB para que sea rápido verificar el funcionamiento.

Para llevarlo a la práctica, se hace uso de las colas o *queues* en OpenFlow. Cada cola se corresponde con una fase de la tarifa, por lo que habrá que crear tres colas cada una con un *id* diferente. Mediante el script `tarifas.py` que se encuentra en el Anexo C se establece el cambio entre las fases (i.e. colas) haciendo uso de dos `if` (líneas 62 y 74): el primero para el cambio de la fase 1 a la fase 2 cuando se han transmitido 2 MB de datos y el segundo para el cambio de la fase 2 a la fase 3 cuando se han transmitido 3 MB de datos (los 2 MB anteriores y 1 MB de la fase 2). Por defecto se empieza en la cola 0 y se pasa mediante el primer `if` a la cola 1 y después con el segundo `if` a la cola 2, con prioridades de 0,1 y 2 respectivamente.

Este script supone que ya existen y están configuradas dichas colas, que por tanto se crearán antes de ejecutar el script.

Como primer paso, hay que confirmar que no existe ninguna tabla QoS o colas ya creadas o residuales de otro proyecto (buena práctica que señalan los creadores de Ryu). En una terminal se sigue la secuencia de comandos:

```
tfg@tfgVM:~$ sudo ovs-vsctl --all destroy Qos
tfg@tfgVM:~$ sudo ovs-vsctl --all destroy Queue
tfg@tfgVM:~$ sudo ovs-vsctl list Qos
tfg@tfgVM:~$ sudo ovs-vsctl list Queue
```

Con los dos primeros comandos se limpian las tablas QoS y colas. Con los dos segundos comandos se listan las tablas QoS y colas que no deberían devolver ningún resultado puesto que han sido limpiadas.

Teniéndolo todo preparado para comenzar, se diseña una red en Mininet formada por dos hosts conectados al mismo switch: h1−s1−h2. Es posible crear la topología directamente desde Mininet sin Miniedit. Para ello en una nueva terminal y se ejecuta el comando:

```
tfg@tfgVM:~$ sudo mn --mac \
             --switch ovsk,protocols=OpenFlow13 \
```

```
--controller remote,ip=127.0.0.1,port=6633
```

El parámetro *mac* configura automáticamente las direcciones MAC de los hosts. Por su parte, el parámetro *switch* permite usar un switch OpenFlow (*ovsk*) y especificar la versión de OpenFlow (1.3). Por último el parámetro *controller* sirve para señalar que se usa un controlador remoto, con cierta IP y en determinado puerto.

Lo siguiente es especificar de nuevo la versión de OpenFlow usada en el Switch y que escuche en el puerto 6632 para acceder a OVSDB (Open vSwitch Database Management Protocol) en la terminal que ya está abierta para los comandos `ovs`:

```
tfg@tfgVM:~$ sudo ovs-vsctl set Bridge s1 protocols=OpenFlow13
tfg@tfgVM:~$ sudo ovs-vsctl set-manager ptcp:6632
```

Los comandos anteriores no devuelven ningún output. Es posible saber que han sido ejecutados correctamente si en el futuro, al crear las colas, no existe un problema de conexión entre el switch y el controlador.

Antes de lanzar el controlador con nuestro script, `tarifas.py`, hay que tener localizado el script en `/usr/local/lib/python2.7/dist-packages/ryu/app` para poder ejecutarlo con tan solo poner `ryu.app.tarifas`, en vez de tener que especificar una ruta del script. En la figura 3.12 se lanza el controlador y el script. El mismo output que se muestra, en la línea `[QoS][INFO] dpid=0000000000000001: Join qos switch.` indica que se ha conectado correctamente el controlador con el switch. Además cada 10 segundos se mostrarán las estadísticas de la parte inferior de la captura, con datos actualizados de cada puerto.

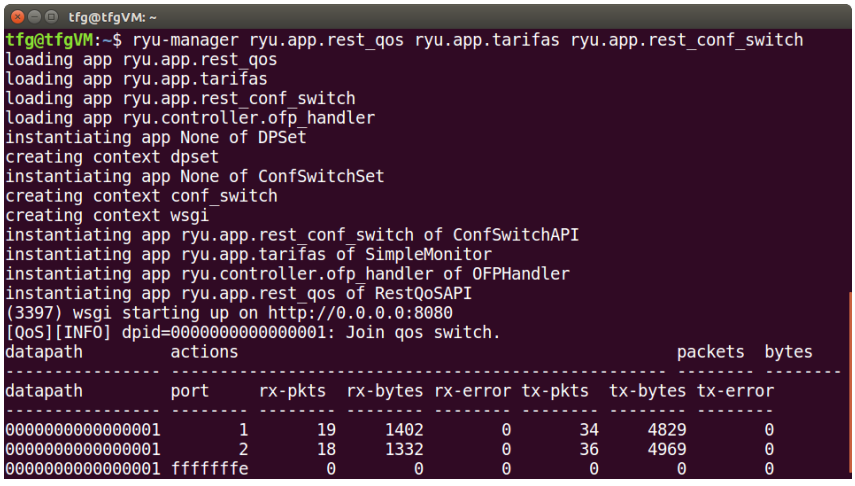

Figura 3.12: Inicio del controlador y script `tarifas.py`.

Tras especificar el primer comando para la localización de OVSDB, permitirá después crear las colas y que no devuelva ningún resultado. Con el siguiente comando se define el ancho de banda de las diferentes colas con los campos *min_rate* y *max_rate* que permiten asegurar un mínimo ancho de banda y un máximo, respectivamente.

```
tfg@tfgVM:~$ curl -X PUT -d '"tcp:127.0.0.1:6632"' \
http://localhost:8080/v1.0/conf/switches/ \
0000000000000001/ovsdb_addr
tfg@tfgVM:~$ curl -X POST -d '{"port_name": "s1-eth1",\
"type": "linux-htb", "max_rate":"1000000", queues":[ \
{"max_rate": "1000000"}, \
{"min_rate": "200000","max_rate": "200000"}, \
{"min_rate":"0","max_rate": "0"}]}' \
http://localhost:8080/qos/queue/0000000000000001

#OUTPUT
[{"switch_id": "0000000000000001",
"command_result": {"result": "success", "details":{
"0": {"config": {"max-rate": "1000000"}},
"1": {"config": {"max-rate": "200000",
                 "min-rate": "200000"}},
"2": {"config": {"max-rate": "0", "min-rate": "0"}}}}}]
```

Con el comando POST anterior se definen las diferentes colas y sus anchos de banda. En la figura 3.12, concretamente en la línea que se ha descrito previamente, hace referencia a un switch con un *dpid* de valor 0000000000000001, siendo este su identidad o *id*. En el comando POST y su output se hace referencia a ese mismo *id*. Esto significa que las colas se han creado en el switch con ese *id*. En este caso no hay problema porque solo existe un switch pero en redes más grandes se debe tener cuidado con las *ids* de cada switch para evitar confusiones.

El output del comando POST indica que el resultado ha sido exitoso (*"result":"success"*) y muestra cada cola añadida y sus anchos de banda.

Para comprobar que la tabla QoS y las colas están bien definidas y han sido instaladas en el switch:

```
tfg@tfgVM:~$ sudo ovs-vsctl list Qos
_uuid        : ac634664-2d74-4aa2-83bf-59ab559639d2
external_ids : {}
other_config : {max-rate="1000000"}
queues       : {0=fd0c16e7-e0c9-4093-9427-84391dabd804,
               1=61219016-7e1b-4dce-a09f-6148b092124e,
               2=0070f3dd-773c-4c1a-ae92-71ee33ad66dd}
type         : linux-htb
tfg@tfgVM:~$ sudo ovs-vsctl list Queue
```

```
_uuid        : 0070f3dd-773c-4c1a-ae92-71ee33ad66dd
dscp         : []
external_ids : {}
other_config : {max-rate="0", min-rate="0"}

_uuid        : 61219016-7e1b-4dce-a09f-6148b092124e
dscp         : []
external_ids : {}
other_config : {max-rate="200000", min-rate="200000"}

_uuid        : fd0c16e7-e0c9-4093-9427-84391dabd804
dscp         : []
external_ids : {}
other_config : {max-rate="1000000"}
```

Mediante los comandos utilizados anteriormente, se obtienen la QoS y las colas. Si la lista QoS devuelve *other_config : max-rate="1000000"* significa que actualmente ese es el máximo ancho de banda, tal y como se esperaba. Al no haber creado tráfico estamos en la fase 1 (1 Mbps). También es posible cotejar los *ids* de las colas con los que figuran en QoS y ver que todo concuerda.

Una vez todo listo para comenzar con el test, Mininet y controlador iniciados con el script y creada la tabla QoS y las colas, es necesario examinar el funcionamiento de la red.

3.3.2. Test

Como en el otro caso de uso, se utiliza iPerf para crear tráfico y revisar si se cumple cada fase de la tarifa *estándar*. Para ello, se abre una terminal en cada host desde la línea de comandos de Mininet:

```
mininet> xterm h1 h2
```

Se abrirán dos terminales, una por cada host, y en ambas se introduce el siguiente comando:

```
root@tfgVM:~# xterm -fa 'Monospace' -fs 8
```

Con este comando se abren de nuevo dos terminales con una fuente de tamaño 8 pts y de tipo `Monospace`. Este paso es opcional, permite tener unas terminales con una fuente mayor y un tipo de letra mejor para realizar las capturas de las fases de la tarifa.

Al igual que en el primer caso de uso, se hace que el host h1 sea el servidor y el host h2 el cliente, aunque es indiferente qué host desarrolle el rol de cliente o

de servidor. Se crea tráfico *UDP* (User Datagram Protocol) utilizando la opción `-u` en vez de tráfico TCP porque iPerf admite la opción `-b` solo en UDP. Esta opción, `-b`, permite establecer el ancho de banda de la conexión, que en nuestro caso será 1 Mbps por parte del cliente y será en el switch donde se limitará esta tasa conforme a la fase en la que estemos, por lo que será el servidor el que percibirá la tasa ya limitada. Como opción extra se utiliza `-i`, que permite elegir el periodo en segundos entre las estadísticas del ancho de banda.

Se da comienzo al test creando tráfico por primera vez como se muestra en la figura 3.13, en la que la terminal de la izquierda es la correspondiente al cliente y la de la derecha al servidor. En esta primera fase, como el ancho de banda impuesto en la cola 0 es el mismo que el que se ha especificado en el comando iPerf del cliente, los dos perciben el mismo ancho de banda (1 Mbps).

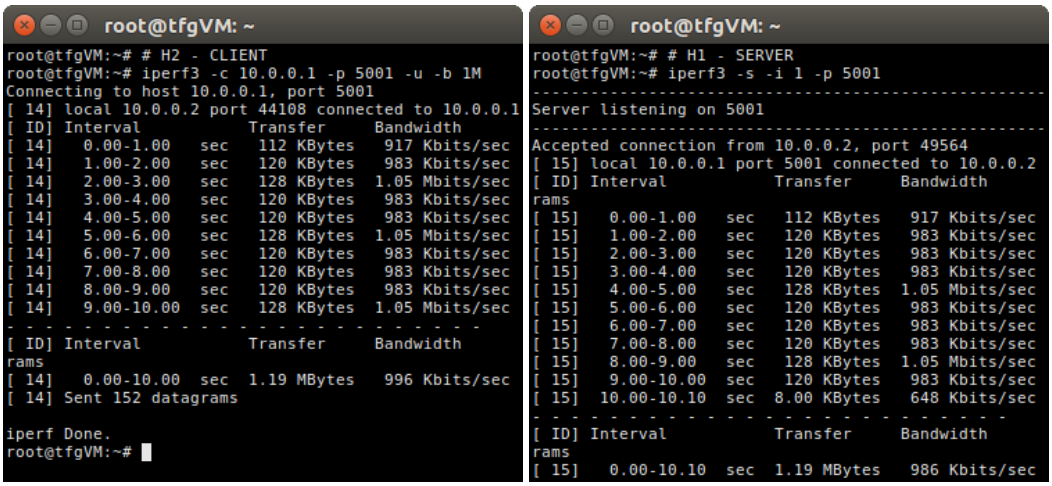

Figura 3.13: Fase 1 de la tarifa.

Se han transferido en total en esta transmisión 1.19 MB, como se observa en la figura 3.13, con un ancho de banda de aproximadamente 0.996 Mbps. Un resultado bastante preciso, teniendo en cuenta que al crear las colas se impuso en esta primera fase que la tasa máxima fuera 1 Mbps (`"max_rate": "1000000"`) pero no se especificó la mínima.

Generalmente, cada vez que se ejecute ese mismo comando, en el cliente se creará un tráfico aproximado a 1 MB, como se ha observado, habiendo creado un tráfico total de 1.19 MB. La siguiente vez que se ejecuta se consiguen unas capturas similares a las de la figura 3.13 ya que se había configurado 2 MB de datos para esta primera fase tarifa. Por lo tanto, tras ejecutar 2 veces el comando iPerf del cliente ya ha usado 2 MB de datos.

La figura 3.14 se corresponde con la tercera ejecución del comando:

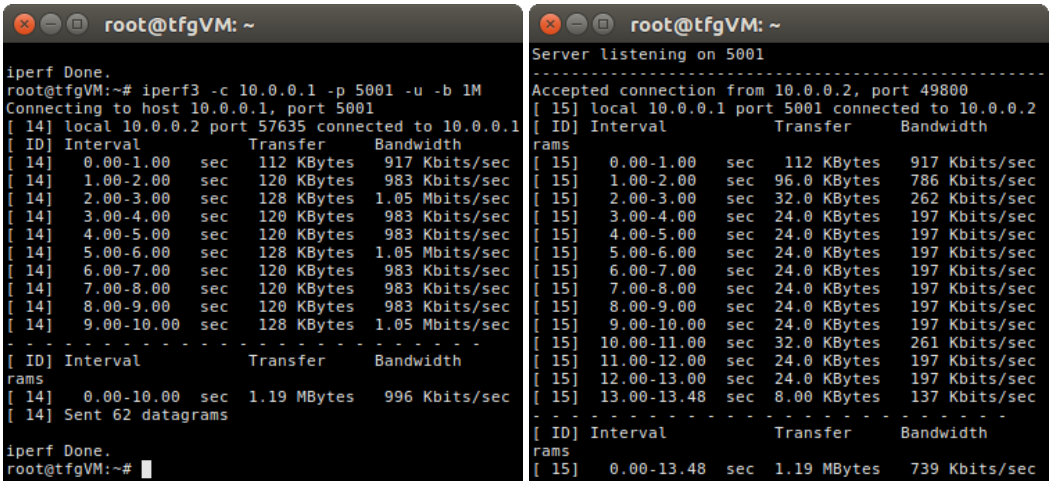

Figura 3.14: Fase 2 de la tarifa.

Igual que en la figura anterior, en las capturas de la figura 3.14 la izquierda corresponde al cliente y la derecha al servidor. El cliente sigue transmitiendo a 1 Mbps mientras que en el servidor se baja el ancho de banda hasta aproximadamente 200 Kbps, entrando en la segunda fase de la tarifa.

Como esta vez también se han transmitido 1.19 MB y para esta segunda fase se había reservado 1 MB cabe esperar que ya se haya agotado y entre la próxima vez que se ejecute el comando, en la tercera y última fase.

Al ejecutar de nuevo el comando, en la figura 3.15 se comprueba que efectivamente ha entrado en la tercera fase: el cliente envía el tráfico con un ancho de banda de 1 Mbps pero el servidor no recibe ni transmite nada. En total se han transmitido 0 Bytes con un ancho de banda de 0 bps.

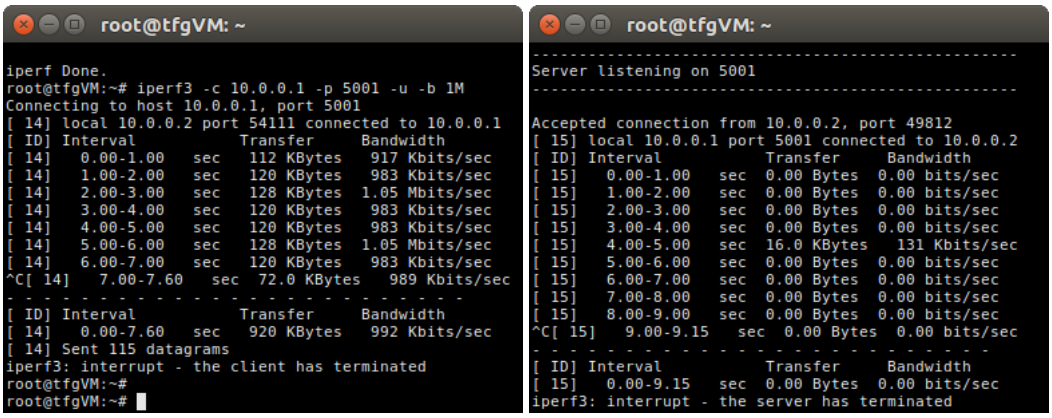

Figura 3.15: Fase 3 de la tarifa.

Se confirma que se ha cortado la conexión entre los extremos desde la consola de Mininet, realizando un ping:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X ; h2 -> X
*** Results: 100% dropped (0/2 received)
```

Como era de esperar, ha sido fallido y se han descartado el 100 % de los paquetes.

Con este resultado, se dá por válido este caso de uso. Se ha implementado la red y aprendido a incluir una calidad de servicio en ella consiguiendo pasar por varias fases con diferentes anchos de banda.

Para implementar la tarifa *Premium*, se debería haber creado solo una cola con un `"min_rate": "2000000"` y poner en el script un límite real de datos (dependiendo del contrato).

Capítulo 4

Conclusiones y líneas futuras

La ingeniería de tráfico es un mecanismo importante y ha sido ampliamente explotado en las redes de datos pasadas y actuales, como ATM e IP/MPLS. Sin embargo, estos paradigmas y sus correspondientes soluciones TE (traffic engineering) se quedan obsoletos en un futuro próximo debido a las soluciones SDN.

4.1. Conclusiones

Las aplicaciones de Internet actuales requieren de la arquitectura de red subyacente para reaccionar en tiempo real y generan gran cantidad de tráfico La arquitectura debe poder clasificar una variedad de tipos de tráfico de diferentes aplicaciones y proporcionar un servicio específico y adecuado para cada tipo de tráfico en un período de tiempo muy corto (ms).

Por si no fuera poco, hay una marcada tendencia hacia el rápido crecimiento de la computación en la nube y, por lo tanto, la demanda de centros de datos de escala masiva, lo que obliga a que una administración de red adecuada debiera de ser capaz de mejorar la utilización de los recursos para un mejor rendimiento del sistema.

Por lo tanto, se necesitan nuevas arquitecturas de red y herramientas TE más inteligentes y eficientes.

En este trabajo se han abordado dos aplicaciones que hacen uso de las técnicas TE de las redes SDN y se ha visto reflejada su rápida implementación y fácil aprendizaje. Además, ha quedado claro que existe un elemento unificador que ha hecho posible el rápido avance de las redes SDN, el software.

A través de los lenguajes de programación más conocidos, como ha sido python en este caso, se pueden diseñar aplicaciones que no dependan de los vendedores y que den soluciones a problemas muy específicos.

En conjunto, SDN no resuelve necesariamente problemas que no hayan sido solventados antes, sino que logra resolver esos problemas de una forma mucho más limpia, fácil y eficaz. SDN proporciona un nivel de automatización que hace posible guardar *templates* (plantillas) de las redes e implementarlas cuando sea necesario, con el ahorro de tiempo que ello supone.

4.2. Líneas futuras

Una de las grandes ventajas de las redes definidas por software es esa centralización del control de la red. Poder monitorizar y gestionar la red desde una sola consola es todo un avance aunque también tiene su parte negativa. Esta centralización supone también un problema como es el de la seguridad de la red, teniendo un único punto de fallo.

En este trabajo se han llevado a cabo dos casos de uso de las redes SDN pero no se ha versado sobre la seguridad de esas aplicaciones. Como líneas futuras a este trabajo, se plantea crear una aplicación que añada seguridad a estos casos de uso o a una aplicación de las redes SDN en general.

Los principales ataques al controlador en redes SDN se dan en forma de:

Spoofing: Una persona o aplicación desautorizada se comporta como un usuario autorizado copiando y falsificando sus datos y por tanto ganándole ventaja.

Elevation of privilege: Un usuario sin privilegios consigue acceso privilegiado y por tanto puede comprometer o destruir el sistema completo.

Denial of service (DoS): Un usuario autorizado es denegado por el servidor porque los recursos del servidor están siendo utilizados en exceso por el atacante.

En el caso del ataque DoS, las capacidades de SDN hacen que sea fácil detectar y reaccionar a un ataque de este tipo frente a las redes tradicionales. Es por ello que se propone crear una topología en Mininet y simular un ataque DoS con el fin de aprender a detectarlo y a reaccionar.

Bibliografía

[1] B. Casemore, P. Jirovsky, and R. Mehra. Datacenter SDN Moves Into the Early Mainstream, Growth Moderates Through 2021. *IDC study*, 2016.

[2] Rubén Isa Hidalgo. Virtual lab implementation for SDN learning. Trabajo fin de grado, Universidad de Cantabria, 2016.

[3] Open Networking Foundation. *OpenFlow Switch Specification*. Version 1.3.0 (Wire Protocol 0x04), 2012.

[4] Ryu Documentation, https://ryu.readthedocs.io/en/latest/. [Última consulta el 05/05/2018].

[5] Mininet Walkthrough, http://mininet.org/walkthrough/. [Última consulta el 18/03/2018].

[6] VirtualBox Documentation, https://www.virtualbox.org/manual/UserManual.html. [Última consulta el 12/03/2018].

[7] iPerf User Documentation, https://iperf.fr/iperf-doc.php. [Última consulta el 20/06/2018].

[8] L. Pu W. Wu C. Akyildiz, f. Ahyoung. A roadmap for traffic engineering in SDN-OpenFlow networks. 2014.

[9] Eric Obsorne, Simha Ajay. Traffic Engineering with MPLS. 2002.

[10] Cisco. Quality of Service Networking, docwiki.cisco.com/wiki/Quality_of_Service_Networking. [Última consulta el 10/06/2018].

[11] jack Tsai, Tim Moors. A Review of Multipath Routing Protocols: From Wireless Ad Hoc to Mesh Networks. *National ICT Austalia / University of New South Wales, Australia*, 2006.

[12] Dpto. de Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla. Recorrido en Anchura de un Grafo (Breadth-First Search). *Computabilidad y complejidad*, 2009/2010.

[13] Dpto. de Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla. Recorrido en Profundidad de un Grafo (Depth-First Search). *Computabilidad y complejidad*, 2009/2010.

[14] S. Agarwal, T.V. Lakshman, and M. Kodialam. Traffic Engineering in Software Defined Networks. *Holmdel, NJ, USA*, 2013.

[15] WeiKang Mao, Qilin Shen. A Load Balancing Method Based on SDN. *Seventh International Conference on Measuring Technology and Mechatronics Automation*, 2015.

Lista de Acrónimos

SDN Software Defined Network (Red definida por software).

IoT Internet of Things (Internet de las Cosas).

API Application Programming Interface (Interfaz de programación de aplicaciones).

TLS Transport Layer Security (Seguridad en la Capa de Transporte).

TCP Transmission Control Protocol (Protocolo de Control de Transmisión).

IP Internet Protocol (Protocolo de Internet).

UDP User Datagram Protocol (Protocolo de Datagramas de Usuario).

VLAN Virtual Local Area Network (Red de Área Local Virtual).

PC Personal Computer (Ordenador Personal).

SCTP Stream Control Transmission Protocol (Protocolo de Control de Transmisión Streaming).

TE Traffic Engineering (Ingeniería de tráfico).

QoS Quality of Service (Calidad de Servicio).

BFS Breadth-first Search Algorithm (Algoritmo de búsqueda en amplitud).

DFS Depth-first Search Algorithm (Algoritmo de búsqueda en profundidad).

OSPF Open Shortest Path First (Primer Camino más Corto).

BW BandWidth (Ancho de Banda).

SSL Secure Sockets Layer (Capa de Sockets Seguros).

OvS Open vSwitch.

Kbps Kilobits per second (Kilobits por segundo).

Mbps Megabits per second (Megabits por segundo).

MB MegaByte.

MAC Media Access Control address (Dirección física).

OvSDB Open vSwitch DataBase (Base de datos de Open vSwitch).

ATM Asynchronous Transfer Mode (Modo de Transferencia Asíncrona).

MPLS MultiProtocol Label Switching (Conmutación de Etiquetas MultiProtocolo).

DoS Denial of Service (Denegación de Servicio).

Anexos A

multicamino.py

```
from ryu.base import app_manager
from ryu.controller import mac_to_port
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import arp
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import ipv6
from ryu.lib.packet import ether_types
from ryu.lib import mac, ip
from ryu.topology.api import get_switch, get_link
from ryu.app.wsgi import ControllerBase
from ryu.topology import event

from collections import defaultdict
from operator import itemgetter

import os
import random
import time

''' Ancho de banda de referencia'''
REFERENCE_BW = 10000000

DEFAULT_BW = 10000000

MAX_PATHS = 2

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

  def __init__(self, *args, **kwargs):
    super(ProjectController, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self.topology_api_app = self
    self.datapath_list = {}
    self.arp_table = {}
    self.switches = []
    self.hosts = {}
    self.multipath_group_ids = {}
    self.group_ids = []
    self.adjacency = defaultdict(dict)
```

```
47      self.bandwidths = defaultdict(lambda: defaultdict(lambda: DEFAULT_BW))
48
49    def get_paths(self, src, dst):
50      '''
51      Conseguir todas las rutas de src a dst usando el algoritmo DFS
52      '''
53      if src == dst:
54        '''El host destino se encuentra en el mismo switch'''
55        return [[src]]
56      paths = []
57      stack = [(src, [src])]
58      while stack:
59        (node, path) = stack.pop()
60        for next in set(self.adjacency[node].keys()) - set(path):
61          if next is dst:
62            paths.append(path + [next])
63          else:
64            stack.append((next, path + [next]))
65      print "Caminos disponibles de ", src, " a ", dst, " : ", paths
66      return paths
67
68    def get_link_cost(self, s1, s2):
69      '''
70      Calcular el coste de enlace entre dos switches
71      '''
72      e1 = self.adjacency[s1][s2]
73      e2 = self.adjacency[s2][s1]
74      bl = min(self.bandwidths[s1][e1], self.bandwidths[s2][e2])
75      ew = REFERENCE_BW/bl
76      return ew
77
78    def get_path_cost(self, path):
79      '''
80      Calcular el coste de la ruta/camino
81      '''
82      cost = 0
83      for i in range(len(path) - 1):
84        cost += self.get_link_cost(path[i], path[i+1])
85      return cost
86
87    def get_optimal_paths(self, src, dst):
88      '''
89      Get the n-most optimal paths according to MAX_PATHS
90      '''
91      paths = self.get_paths(src, dst)
92      paths_count = len(paths) if len(paths) < MAX_PATHS else MAX_PATHS
93      return sorted(paths,
94                    key=lambda x: self.get_path_cost(x))[0:(paths_count)]
95
96    def add_ports_to_paths(self, paths, first_port, last_port):
97      '''
98      Add the ports that connects the switches for all paths
99      '''
100     paths_p = []
101     for path in paths:
102       p = {}
103       in_port = first_port
104       for s1, s2 in zip(path[:-1], path[1:]):
105         out_port = self.adjacency[s1][s2]
106         p[s1] = (in_port, out_port)
107         in_port = self.adjacency[s2][s1]
108       p[path[-1]] = (in_port, last_port)
109       paths_p.append(p)
110     return paths_p
111
```

```
112   def generate_openflow_gid(self):
113     '''
114     Returns a random OpenFlow group id
115     '''
116     n = random.randint(0, 2**32)
117     while n in self.group_ids:
118       n = random.randint(0, 2**32)
119     return n
120
121
122   def install_paths(self, src, first_port, dst, last_port, ip_src, ip_dst):
123     computation_start = time.time()
124     paths = self.get_optimal_paths(src, dst)
125     pw = []
126     for path in paths:
127       pw.append(self.get_path_cost(path))
128       print path, "coste = ", pw[len(pw) - 1]
129     sum_of_pw = sum(pw) * 1.0
130     paths_with_ports = self.add_ports_to_paths(paths, first_port, last_port)
131     switches_in_paths = set().union(*paths)
132
133     for node in switches_in_paths:
134
135       dp = self.datapath_list[node]
136       ofp = dp.ofproto
137       ofp_parser = dp.ofproto_parser
138
139       ports = defaultdict(list)
140       actions = []
141       i = 0
142
143       for path in paths_with_ports:
144         if node in path:
145           in_port = path[node][0]
146           out_port = path[node][1]
147           if (out_port, pw[i]) not in ports[in_port]:
148             ports[in_port].append((out_port, pw[i]))
149         i += 1
150
151       for in_port in ports:
152
153         match_ip = ofp_parser.OFPMatch(
154           eth_type=0x0800,
155           ipv4_src=ip_src,
156           ipv4_dst=ip_dst
157         )
158         match_arp = ofp_parser.OFPMatch(
159           eth_type=0x0806,
160           arp_spa=ip_src,
161           arp_tpa=ip_dst
162         )
163
164         out_ports = ports[in_port]
165
166
167         if len(out_ports) > 1:
168           group_id = None
169           group_new = False
170
171           if (node, src, dst) not in self.multipath_group_ids:
172             group_new = True
173             self.multipath_group_ids[
174                 node, src, dst] = self.generate_openflow_gid()
175           group_id = self.multipath_group_ids[node, src, dst]
176
```

```
177           buckets = []
178
179           for port, weight in out_ports:
180             bucket_weight = int(round((1 - weight/sum_of_pw) * 10))
181             bucket_action = [ofp_parser.OFPActionOutput(port)]
182             buckets.append(
183                 ofp_parser.OFPBucket(
184                     weight=bucket_weight,
185                     watch_port=port,
186                     watch_group=ofp.OFPG_ANY,
187                     actions=bucket_action
188                 )
189             )
190
191           if group_new:
192             req = ofp_parser.OFPGroupMod(
193                 dp, ofp.OFPGC_ADD, ofp.OFPGT_SELECT, group_id,
194                 buckets
195             )
196             dp.send_msg(req)
197           else:
198             req = ofp_parser.OFPGroupMod(
199                 dp, ofp.OFPGC_MODIFY, ofp.OFPGT_SELECT,
200                 group_id, buckets)
201             dp.send_msg(req)
202
203           actions = [ofp_parser.OFPActionGroup(group_id)]
204
205           self.add_flow(dp, 32768, match_ip, actions)
206           self.add_flow(dp, 1, match_arp, actions)
207
208         elif len(out_ports) == 1:
209           actions = [ofp_parser.OFPActionOutput(out_ports[0][0])]
210
211           self.add_flow(dp, 32768, match_ip, actions)
212           self.add_flow(dp, 1, match_arp, actions)
213     print "Camino instalado en ", time.time() - computation_start, "\n"
214     return paths_with_ports[0][src][1]
215
216   def add_flow(self, datapath, priority, match, actions, buffer_id=None):
217
218     ofproto = datapath.ofproto
219     parser = datapath.ofproto_parser
220
221     inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
222                                          actions)]
223     if buffer_id:
224       mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
225                               priority=priority, match=match,
226                               instructions=inst)
227     else:
228       mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
229                               match=match, instructions=inst)
230     datapath.send_msg(mod)
231
232   @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
233   def _switch_features_handler(self, ev):
234     print "Se ha llamado a switch_features_handler"
235     datapath = ev.msg.datapath
236     ofproto = datapath.ofproto
237     parser = datapath.ofproto_parser
238
239     match = parser.OFPMatch()
240     actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
241                                       ofproto.OFPCML_NO_BUFFER)]
```

```
    self.add_flow(datapath, 0, match, actions)

  @set_ev_cls(ofp_event.EventOFPPortDescStatsReply, MAIN_DISPATCHER)
  def port_desc_stats_reply_handler(self, ev):
    switch = ev.msg.datapath
    for p in ev.msg.body:
      self.bandwidths[switch.id][p.port_no] = p.curr_speed

  @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
  def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    arp_pkt = pkt.get_protocol(arp.arp)

    ''' Evitar broadcast de LLDP (Link Layer Discover Protocol)'''
    if eth.ethertype == 35020:
      return

    ''' Drop paquetes IPV6.'''
    if pkt.get_protocol(ipv6.ipv6):
      match = parser.OFPMatch(eth_type=eth.ethertype)
      actions = []
      self.add_flow(datapath, 1, match, actions)
      return None

    dst = eth.dst
    src = eth.src
    dpid = datapath.id

    if src not in self.hosts:
      self.hosts[src] = (dpid, in_port)

    out_port = ofproto.OFPP_FLOOD

    if arp_pkt:

      src_ip = arp_pkt.src_ip
      dst_ip = arp_pkt.dst_ip
      if arp_pkt.opcode == arp.ARP_REPLY:
        self.arp_table[src_ip] = src
        h1 = self.hosts[src]
        h2 = self.hosts[dst]
        out_port = self.install_paths(h1[0], h1[1], h2[0], h2[1],
                                      src_ip, dst_ip)
        self.install_paths(h2[0], h2[1], h1[0], h1[1], dst_ip, src_ip)
        '''al contrario, son enlaces bidireccionales'''
      elif arp_pkt.opcode == arp.ARP_REQUEST:
        if dst_ip in self.arp_table:
          self.arp_table[src_ip] = src
          dst_mac = self.arp_table[dst_ip]
          h1 = self.hosts[src]
          h2 = self.hosts[dst_mac]
          out_port = self.install_paths(h1[0], h1[1], h2[0], h2[1],
                                        src_ip, dst_ip)
          self.install_paths(h2[0], h2[1], h1[0], h1[1], dst_ip, src_ip)
          '''al contrario, son enlaces bidireccionales'''
```

```
307     actions = [parser.OFPActionOutput(out_port)]
308
309     data = None
310     if msg.buffer_id == ofproto.OFP_NO_BUFFER:
311       data = msg.data
312
313     out = parser.OFPPacketOut(
314         datapath=datapath, buffer_id=msg.buffer_id, in_port=in_port,
315         actions=actions, data=data)
316     datapath.send_msg(out)
317
318   @set_ev_cls(event.EventSwitchEnter)
319   def switch_enter_handler(self, ev):
320     switch = ev.switch.dp
321     ofp_parser = switch.ofproto_parser
322
323     if switch.id not in self.switches:
324       self.switches.append(switch.id)
325       self.datapath_list[switch.id] = switch
326
327       ''' Peticion de la descripcion de puertos/enlaces. util para obetner BW'''
328       req = ofp_parser.OFPPortDescStatsRequest(switch)
329       switch.send_msg(req)
330
331   @set_ev_cls(event.EventSwitchLeave, MAIN_DISPATCHER)
332   def switch_leave_handler(self, ev):
333     print ev
334     switch = ev.switch.dp.id
335     if switch in self.switches:
336       self.switches.remove(switch)
337       del self.datapath_list[switch]
338       del self.adjacency[switch]
339
340   @set_ev_cls(event.EventLinkAdd, MAIN_DISPATCHER)
341   def link_add_handler(self, ev):
342     s1 = ev.link.src
343     s2 = ev.link.dst
344     self.adjacency[s1.dpid][s2.dpid] = s1.port_no
345     self.adjacency[s2.dpid][s1.dpid] = s2.port_no
346
347   @set_ev_cls(event.EventLinkDelete, MAIN_DISPATCHER)
348   def link_delete_handler(self, ev):
349     s1 = ev.link.src
350     s2 = ev.link.dst
351     ''' control de la excepcion si el switch ya ha sido eliminado'''
352     try:
353       del self.adjacency[s1.dpid][s2.dpid]
354       del self.adjacency[s2.dpid][s1.dpid]
355     except KeyError:
356       pass
```

Anexos B

topologia_multicamino.py

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',
                         controller=Controller,
                         protocol='tcp',
                         port=6633)

    info( '*** Add switches\n')
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)

    info( '*** Add links\n')
    net.addLink(s1, s2)
    net.addLink(s1, s3)
    net.addLink(s3, s4)
    net.addLink(s4, s5)
    net.addLink(s2, s5)
    net.addLink(s5, h2)
    net.addLink(h1, s1)

    info( '*** Starting network\n')
    net.build()
```

```
47      info( '*** Starting controllers\n')
48      for controller in net.controllers:
49          controller.start()
50
51      info( '*** Starting switches\n')
52      net.get('s1').start([c0])
53      net.get('s5').start([c0])
54      net.get('s2').start([c0])
55      net.get('s3').start([c0])
56      net.get('s4').start([c0])
57
58      info( '*** Post configure switches and hosts\n')
59
60      CLI(net)
61      net.stop()
62
63  if __name__ == '__main__':
64      setLogLevel( 'info' )
65      myNetwork()
```

Anexos C

tarifas.py

```python
from operator import attrgetter

from ryu.app import qos_simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub


class SimpleMonitor(qos_simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)

    @set_ev_cls(ofp_event.EventOFPStateChange,
                [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x', datapath.id)
                del self.datapaths[datapath.id]

    def _monitor(self):
        while True:
            for dp in self.datapaths.values():
                self._request_stats(dp)
            hub.sleep(10)

    def _request_stats(self, datapath):
        self.logger.debug('send stats request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        req = parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
        datapath.send_msg(req)

```

```
47      @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
48      def _flow_stats_reply_handler(self, ev):
49          body = ev.msg.body
50
51          self.logger.info('datapath         '
52         'actions                                                 '
53                           'packets  bytes')
54          self.logger.info('---------------- '
55                           '-------------------------------------------------- '
56                           '-------- --------')
57          for stat in sorted([flow for flow in body if flow.priority >= 1], \
58          key=lambda flow: (flow.instructions[0].actions[0])):
59              self.logger.info('%016x %51s %8d %8d',
60                               ev.msg.datapath.id,stat.instructions[0].actions[0],
61                               stat.packet_count, stat.byte_count)
62        if(int(stat.byte_count) > 2000000):
63      ''' 2a fase: mediante un flow nuevo ralentizamos el ancho de banda '''
64      dp = ev.msg.datapath
65      ofp = dp.ofproto
66      parser = dp.ofproto_parser
67
68      match = parser.OFPMatch(in_port=stat.match['in_port'])
69      actions = [parser.OFPActionSetQueue(1)]
70      inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,actions),
71              parser.OFPInstructionGotoTable(1)]3
72      mod = parser.OFPFlowMod(datapath=dp, priority=1,match=match, instructions=inst)
73      dp.send_msg(mod)
74        if(int(stat.byte_count) > 3000000):
75      ''' 3a fase: creamos un flow para descartar (drop) todos los paquetes'''
76      dp = ev.msg.datapath
77      ofp = dp.ofproto
78      parser = dp.ofproto_parser
79
80      match = parser.OFPMatch(in_port=stat.match['in_port'])
81      actions = [parser.OFPActionSetQueue(2)]
82      inst = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,actions),
83              parser.OFPInstructionGotoTable(1)]
84      mod = parser.OFPFlowMod(datapath=dp, priority=2,match=match, instructions=inst)
85      dp.send_msg(mod)
86
87
88      @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
89      def _port_stats_reply_handler(self, ev):
90          body = ev.msg.body
91
92          self.logger.info('datapath         port     '
93                           'rx-pkts  rx-bytes rx-error '
94                           'tx-pkts  tx-bytes tx-error')
95          self.logger.info('---------------- -------- '
96                           '-------- -------- -------- '
97                           '-------- -------- --------')
98          for stat in sorted(body, key=attrgetter('port_no')):
99              self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',
100                              ev.msg.datapath.id, stat.port_no,
101                              stat.rx_packets, stat.rx_bytes, stat.rx_errors,
102                              stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```