eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Facultad de Informática

Grado de Ingeniería Informática

■ **T**rabajo **F**in de **G**rado ■

Ingeniería de Computadores

Redes definidas por software en el datacenter

Julen Pérez-Cortés Bakaikoa

Septiembre 2018

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Facultad de Informática

Grado de Ingeniería Informática

■ **T**rabajo **F**in de **G**rado ■

Ingeniería de Computadores

Redes definidas por software en el datacenter

Julen Pérez-Cortés Bakaikoa

Septiembre 2018

Director: José Miguel-Alonso

1. Agradecimientos

Me gustaría dedicar este trabajo de fin de grado sobre todo a mi familia: padre, madre y hermano. Han sido, indudablemente, las personas que más tiempo han estado a mi lado durante estos cuatro años, tanto en las buenas como en las malas. Son las personas que mejor conocen todas las etapas por las que he pasado. Siempre que lo he necesitado, me han ayudado en todo lo que han podido, y, si hay alguien que me ha enseñado a que en esta vida los objetivos se consiguen gracias a la constancia y el trabajo diario, esas personas son ellos y estoy realmente agradecido. Gracias familia.

Tampoco quiero dejar de lado a aquellos compañeros que he conocido durante este tiempo. Compañeros con los cuales he compartido mucho tiempo, he aprendido mucho y que también espero haberles enseñado algo. Este trabajo también va por vosotros. Gracias amigos.

Para finalizar, querría dar las gracias al tutor José Miguel-Alonso por brindarme la oportunidad de realizar este TFG y por haberme enseñado tanto en tan poco tiempo. Gracias a su predisposición, implicación y toda la confianza depositada en mí ha hecho que este trabajo sea realmente llevadero. Una pena no haber tenido la oportunidad de haberlo conocido antes durante estos cuatro años de carrera. Gracias José.

2. Introducción / Sarrera

Castellano [ES]

Con este TFG se pretende poner a prueba una tecnología que a día de hoy está cogiendo cada vez más fuerza, la tecnología SDN ("*Software Defined Networking*"), en español, Redes Definidas por Software. Se tratará de demostrar que con esta tecnología es posible conseguir la comunicación entre nodos que se encuentran en un mismo datacenter. La gran diferencia respecto a las redes convencionales es que, en este caso, todos y cada uno de los switches carecerán de inteligencia. Dicha inteligencia se implantará mediante un controlador, cuyo cometido es instruir a los switches para que se comporten como el programador lo desee.

Concretamente, las estructuras de red que se estudiarán son aquellas que se utilizan en los centros de datos. Al no trabajar con hardware real, se utilizará el entorno de emulación *Mininet* para generar las topologías. El controlador por el que se ha optado ha sido el controlador *POX*, debido a que en Internet hay mucha documentación sobre él y está totalmente desarrollado en Python.

Para llegar a cumplir todos los objetivos establecidos se requiere una fase previa de aprendizaje. Para ello, se estudiarán algunos conceptos imprescindibles como "*la tecnología SDN", "el protocolo OpenFlow", "el controlador POX", "el entorno de emulación de Mininet"* o "la instalación y configuración de las herramientas necesarias *para llevar a cabo este proyecto"*.

Resumiendo, con este TFG se intenta plantear una nueva alternativa para el futuro para la comunicación entre hosts, utilizando la tecnología SDN, dejando de lado los switches convencionales y apostando por los switches OpenFlow.

Euskara [EUS]

Proiektu honekin, gaur egun indar oso handia hartzen ari den teknologia bat erakutsi nahi da, SDN (“Software Defined Networking”) teknologia. Datacenter berean aurkitzen diren nodoen arteko komunikazioa posible dela erakutsi nahi da SDN teknologiaren bitartez. Sare konbentzialekin duen alde nagusia honakoa da, sarea osatzen duten switch bakoitzari adimena kenduko zaio. Adimen hori, kontroladore batek eskainiko die, programatzaileak nahi duen bezala lan egin dezaten.

Zehazki, landuko diren sare-egiturak *datacenterretan* erabiltzen direnak izango dira. Benetako hardwarearekin ez denez lanik egingo, *Mininet* emulazio-ingurunea erabiliko da topologiak sortzeko. Aldiz, erabiliko den kontroladorea *POX* izango da. Izan ere, Interneten dokumentazio eta euskarri pila dago eta Pythonen dago garatua guztiz.

Ezarritako helburu guztiak betetzeko ikaskuntza-fase batetik igarotzea ezinbestekoa da. Horretarako, zenbait kontzeptu ikasiko dira, horien artean “*SDN teknologia”, “OpenFlow protokoloa”, “POX kontroladorea”, “Mininet emulazio-ingurunea” eta “proiektu hau aurrera eramateko beharrezkoak diren erreminten instalazio eta konfigurazioa”*.

Laburbilduz, proiektu honekin, sare berean kokatzen diren host desberdinak komunikatzeko etorkizunerako aukera berri bat proposatu nahi da, SDN teknologia erabiliz, switch konbentzionalak alde batera utziz, OpenFlow switchen alde apustu eginez.

Índice de contenido

3. Objetivo

Primeramente, en este TFG, se pretende demostrar que mediante la tecnología “redes definidas por software” podemos conseguir replicar redes tradicionales que se utilizan en los datacenter. Al no trabajar con hardware real, las topologías se emularán en el entorno de trabajo de Mininet y se demostrará la conectividad entre los distintos host, intercambiando paquetes entre los mismos. Para que sea posible el intercambio de datos, será imprescindible programar un controlador para que éste instruya a los switches, utilizando el protocolo OpenFlow.

Asimismo, se pretende evaluar la idoneidad de usar SDNs en un datacenter, ofreciendo un punto centralizado de gestión. Para ello, el primer paso consistiría en definir las necesidades de comunicación en el datacenter, proponer o elegir una arquitectura de red (topología, protocolos), un sistema de gestión de direcciones, implementar un prototipo que permita evaluar diferentes opciones de redes, utilizando una plataforma como Mininet para el despliegue y un controlador como POX para la gestión de la red. Finalmente, realizar una evaluación de la solución en cuanto a funcionalidad, que puede ser limitada en tamaño, teniendo en cuenta que no se tratará con equipamiento real.

Los requisitos mínimos para poder llevar a cabo este TFG son los siguientes: un equipo capaz de ejecutar Mininet y un controlador OpenFlow, en este caso, POX.

4. La tecnología SDN

En este apartado se explicará de qué trata la tecnología que se empleará para desarrollar este proyecto, es decir, la tecnología "Software-Defined Networking" o *SDN*, en español conocida como "Redes Definidas por Software". Además, para tener una idea general de cuáles son sus principios y cuál ha sido su evolución en el tiempo, se hará una breve historia entorno a las SDN. Después, se analizarán cuáles son sus fundamentos y características.

4.1 Historia

La historia de la tecnología SDN se remonta 20 años atrás [1], más o menos, en la creación de lo que hoy conocemos como Internet. Con la creación de Internet se puso cierto interés en gestionar y evolucionar las infraestructuras de redes, en este caso, hacerlas programables. Desde ese momento, muchas han sido las ideas innovadoras orientadas a hacer más programables las redes. Por lo tanto, se puede decir que es una tecnología que en estos últimos años está cogiendo una fuerza inmensa, pero que la idea fundamental viene desde hace varios años atrás.

La breve pero intensa historia de las SDN se divide en tres etapas bastante marcadas. La primera, desde el año 1995 hasta principios del siglo XXI, la segunda, desde el año 2001 hasta el año 2007, aproximadamente y, la tercera y última, desde el año 2007 hasta el año 2010, más o menos.

4.1.1 Primera etapa: Redes activas

En esta etapa, el uso de las aplicaciones aumentó de forma considerable, por lo que los investigadores tuvieron que empezar a diseñar y probar nuevos protocolos de red. Después de probarlos, los protocolos tenían que ser estandarizados por el IETF, pero era un proceso muy lento y frustrante. Por lo tanto, como las redes convencionales son programables, surgió la idea de las redes activas orientadas al control de red, redes que mediante una API proporcionaban distintos recursos como el procesamiento, almacenamiento y colas de paquetes para luego aplicar funciones personalizadas a un subconjunto de paquetes que pasan a través del nodo.

Las redes activas [2] proporcionan a un nodo un procesamiento "a medida" sobre los paquetes que atraviesan dicho nodo. Además, aparece un nuevo concepto, que es el hecho de poder programar el comportamiento de los nodos "en línea", ya sea mediante un administrador de red o mediante el propio usuario. En otras palabras, se consigue poder cambiar dinámicamente el comportamiento de la red.

Mediante la programación de la red se consiguió, entre otras cosas, introducir nuevos protocolos y adaptar nuevos servicios o crear servicios ya existentes en las computadoras de los usuarios, según sus necesidades.

Por lo tanto, en esta etapa se investigaron nuevas funcionalidades de los servicios disponibles mediante Internet, en esta caso, vía IP. Además, se dieron las primeras contribuciones a las SDN, por ejemplo, la separación lógica de distintos recursos.

4.1.2 Segunda etapa: Separación plano control/datos

Al principio de esta etapa, ya que la tecnología del momento lo permitía, empresas importantes de equipos de hardware empezaron a implementar dispositivos de red cuyo plano de datos y control estaban separados, con el fín de aportar servicios más fiables y seguros a sus clientes.

Se crearon arquitecturas como *SoftRouter* y el protocolo *PCE* (Path Computation Element), dos conceptos claves en algunas tecnologías utilizadas en SDN a día de hoy.

4.1.3 Tercera etapa: Aparición de la API de OpenFlow

En la última etapa, se empezó a pensar en nuevas arquitecturas para la separación de los planos y el control lógico centralizado. Uno de los resultados más importantes de un grupo de investigación de la universidad de Stanford fue el proyecto *4D*. En él, se definieron cuatro capas principales:

- Plano de datos: Procesar paquetes basándose en reglas configurables
- Plano de descubrimiento: Coleccionar medidas topológicas
- Plano de diseminación: Instalar reglas de procesado de paquete
- Plano de decisión: Controladores lógicos centralizados encargados de convertir objetivos a nivel de red en estado de paquetes

Lo más destacable en esta etapa, sin lugar a duda, fueron dos trabajos que se realizaron en la universidad de Stanford. Para ser exactos, los proyectos *SANE* y *Ethane* [1], ambos basados en el proyecto *4D* y en ellos se pueden observar las bases de la *API* de lo que posteriormente sería *OpenFlow*.

Años más tarde, unos investigadores de la universidad de Stanford crearon el *Clean Slate Program*, un programa de investigación cuyo objetivo era experimentar con redes universitarias. El resultado de este proyecto fué ni más ni menos lo que hoy en día conocemos como el protocolo *OpenFlow*, aunque cabe decir que la creación de este protocolo sería imposible si no fuera por todas las ideas innovadoras aportadas durante los 20 años anteriores.

Finalmente, muchas empresas importantes dieron su visto bueno a *OpenFlow* y abrieron sus *API* para permitir a los programadores controlar ciertos comportamientos de reenvío de

paquetes. La versión inicial del protocolo OpenFlow se implementó mediante una actualización del firmware de los dispositivos de red, sin necesidad de tener que cambiar ningún tipo de hardware.

4.2 Definición y características

Las SDNs son un conjunto de técnicas que permiten controlar, monitorizar y gestionar una red, desde un nodo centralizado, lo cual promete simplificar la gestión de red e incluir innovación a través de su programación. Además, facilitan la implementación de servicios de red sin tener que hacer uso de hardware real de propósito específico, siendo su principal objetivo implantar dichos servicios de manera dinámica y escalable, evitando en todo momento tener que gestionar cada uno de los dispositivos que conforman la red, de manera independiente.

En una red definida por software, un administrador de red puede darle forma al tráfico, sin tener que configurar conmutadores individuales uno por uno. El administrador es libre de cambiar las reglas de reenvío según lo vea conveniente, puede dar o quitar prioridad a según qué tipo de flujos e incluso bloquear aquellos paquetes con características específicas muy detalladas. Por tanto, permite al administrador manejar grandes cargas de tráfico de manera muy eficiente y flexible.

4.3 Necesidad de las SDNs

La necesidad de las SDNs surge principalmente por dos factores. Por una parte, por las limitaciones propias que presentan las redes tradicionales, y por otra parte, por la necesidad de una nueva arquitectura de red que requieren algunos servicios de red para satisfacer todas los necesidades de muchas empresas actuales y usuarios finales. Ante la rápida evolución de las diferentes tecnologías, las arquitecturas de red estáticas tradicionales no son capaces de soportar la computación dinámica que a día de hoy requieren los *datacenter* más modernos. Para hacer frente a ese problema, la tecnología SDN se nos presenta como una interesante solución.

Algunas de las limitaciones que se presentan con el uso de redes tradicionales son las siguientes [3]:

- Complejidad: Uso de configuraciones de bajo nivel

- Problemas de escalabiilidad: Los centros de datos de hoy en día, tienden a extenderse de manera más rápida. En consecuencia, la red se convierte mucho más compleja y hay infinidad de aparatos que deben ser configurados

- Políticas incoherentes: Para implementar una política que abarque a la red completamente, los administradores de red, deben configurar multitud de mecanismos y aparatos

- Dependencia del vendedor: Las necesidades que podamos tener a la hora de diseñar y construir nuestra red dependen directamente de la capacidad de los dispositivos que haya en el mercado. Puede ocurrir que los vendedores tarden en comercializar dichos dispositivos y, por lo tanto, no cubrir nuestras necesidades y retrasar la implementación de la red

- Incertidumbre: Mediante el sistema de red convencional, no es fácil predecir el comportamiento de la red

- Lenguaje de bajo nivel: Empleo de lenguajes especificados por el fabricante para configurar los dispositivos

En la siguiente lista, se detallan algunos de los servicios de red que requieren una nueva arquitectura de red, servicios a los cuales la arquitectura de red convencional les queda pobre para cubrir las necesidades de hoy en día:

- Nuevos patrones de tráfico debido a aplicaciones que acceden a bases de datos

- Aumento de uso de servicios en la nube que requieren el acceso a aplicaciones, infraestructuras y otros recursos

- Crecimiento de nuevos tipos de usuarios, que acceden a recursos usando nuevos tipos de dispositivos para acceder a la red corporativa, conectándose a todas horas y desde cualquier lugar

- Aplicaciones "*Big Data*" que demandan gran ancho de banda y requieren nuevas formas más flexibles de gestionar la red

Es tal la fuerza que están cogiendo las SDN a día de hoy, que para que tengamos una idea, en el año 2013, hubo un total de ventas de 360 millones de dólares con productos integrados con SDN (infraestructura de red, soluciones de plano de control y aplicaciones) y servicios profesionales relacionados con empresas. Tres años después, en el año 2016, el ingreso total de los mencionados productos pasó de 360 millones a 3.700 millones.

En una encuesta [4] realizada a operadores de "*Infonetic Research*", en 2013, un 29% de los operadores estaban ya trabajando en la implementación de SDN, y para, 2014, se esperaba que esa cantidad aumentase al 52%. Además, remarcar que la gran mayoría de los

operadores, por no decir todos, tenían la intención de implementar la tecnología, si no en toda su red, al menos en algún segmento de la misma.

Estos datos, indudablemente, son una razón que justifica que muchas empresas implementarán la tecnología SDN, o, al menos, contarán con la opción de que sea implementada. Y, en un futuro no muy lejano, no será nada extraño que encontremos las SDN implementadas en gran cantidad de empresas, ya sea ofreciendo sus servicios o ya sea en su infraestructura de red.

4.4 Diferencias entre redes tradicionales y SDNs

La diferencia principal que tienen las SDN respecto a las redes tradicionales es que el plano de control (software) y el de datos (hardware) están separados. En las redes tradicionales, el plano de control y el de datos están implementados en el firmware de los nodos, ya sea en los routers o en los switches. En cambio, en las SDN, se separan los dos planos, y, por lo tanto, la gestión en los diferentes nodos se simplifica y es menos propensa a errores.

Para que haya comunicación entre los citados planos es indispensable tener en marcha un controlador. Cómo funciona un controlador y cuál es el utilizado para este TFG se detalla en su correspondiente apartado. Desde el controlador externo, programando la funcionalidad de toda una serie de elementos de reenvío como pueden ser los switches o routers, conseguiremos administrar toda la red desde un único punto, de manera completamente centralizada. De esta manera, los switches se convierten en simples transmisores de tráfico y el comportamiento de los mismos es completamente programable.

4.5 Arquitectura de red

La arquitectura de red [5] diseñada para las SDN se puede definir de la siguiente manera. La lógica de control se elimina de todos los nodos, el routing no se hace únicamente en base a las direcciones MAC (switching clásico) e IP (routing clásico), sino también en base a otras características como puertos TCP/IP, etiquetas VLAN o puertos de entrada del switch. La lógica empleada en cada uno de los elementos de reenvío en las redes estáticas tradicionales, se traslada a un controlador externo, que ejecuta un *NOS* (*Network Operating System*), en español conocido como Sistema Operativo de Red. El NOS es un sistema operativo que incluye programas para comunicarse con otras computadoras a través de una red y compartir recursos. Finalmente, la red es programable a través de aplicaciones que funcionan sobre el NOS y se comunican con los dispositivos de red (plano de datos). En la figura 4.1 podemos observar la diferencia fundamental entre las redes tradicionales y las redes que emplean la tecnología SDN.

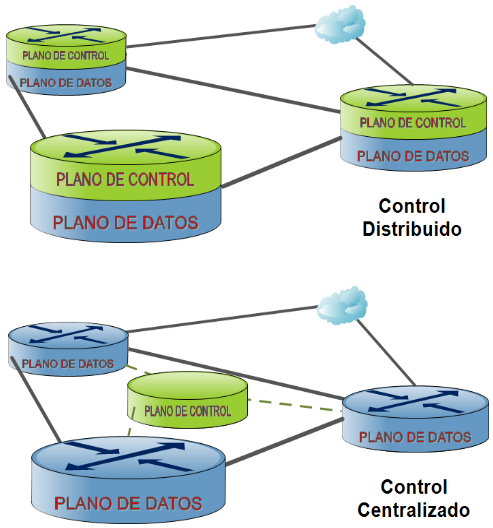

Figura 4.1: diferencia entre redes tradicionales y SDNs [6]

4.5.1 Capas

En este apartado, se explicarán las tres capas que conforman el modelo de la arquitectura SDN. Por una parte, la capa de infraestructura, perteneciente al plano de datos y, por otra parte, la capa de control y la capa de aplicación. Esas dos últimas pertenecientes al plano de control. En la figura 4.2 se muestra cual es el modelo de arquitectura que utiliza la arquitectura SDN.

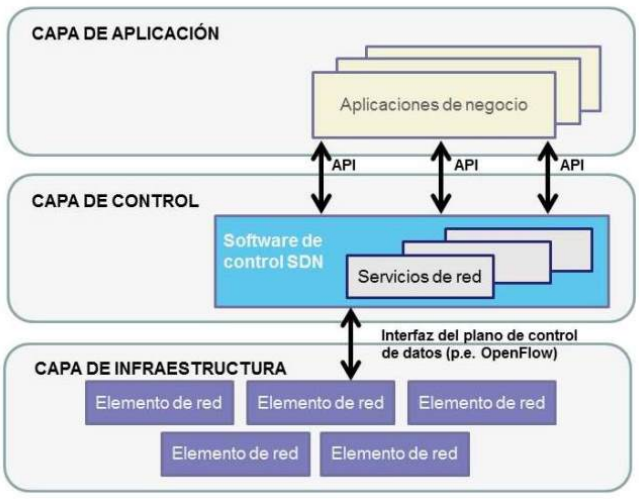

Figura 4.2: modelo de arquitectura [7]

Capa de infraestructura

También llamada capa de conmutación. En esta capa residen los dispositivos de red, ya sean hosts, switches o routers. Como se ha explicado antes, solamente se tiene en cuenta el hardware que componen los dispositivos, no los protocolos que contengan ni ningún tipo de inteligencia. Estos dispositivos contienen *ASICs* (*Application Specific Integrated Circuits*), que son circuitos que símplemente conmutan los paquetes que entran por un puerto para que salgan por otro, según lo que les instruya el controlador. Carecen de inteligencia, pero eso es justamente lo que le aporta el plano de control, el o los controladores para ser exactos.

Por lo tanto, los dispositivos de red utilizados pueden ser equipos de muy bajo coste económico y no tienen que ser de un único fabricante. En cambio, los switches utilizados en redes tradicionales son bastante más caros ya que contienen el plano de datos (conmutación) y el de control (inteligencia) integrados en el mismo. Eso sí, los switches utilizados en las SDN deben proporcionar una manera para interactuar con el plano de control, para dialogar con el controlador y así puedan ser instruidos, aportándoles inteligencia. En este proyecto, a dicha manera la llamaremos "*OpenFlow*", puesto que es el protocolo más utilizado hoy en día que permite interactuar con el plano de control y será el utilizado a la hora de desarrollar este proyecto.

En este punto, los dispositivos seguirían brindando el mecanismo de conmutación, pero toda la inteligencia estaría centralizada. Por lo tanto, el dispositivo, que al principio únicamente estaba compuesto por simples *ASICs*, sería capaz de, una vez tenga las políticas de reenvío definidas por el panel de control, llevar a cabo el proceso de conmutación.

En la figura 4.3 podemos observar la infraestructura interna de cada uno de los dispositivos ubicados en la capa de infraestructura.

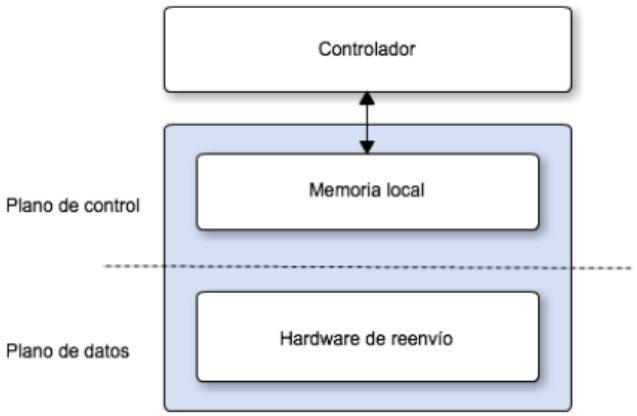

Figura 4.3: estructura interna de los dispositivos que se encuentran en la capa de infraestructura

Cabe destacar que cada uno de los dispositivos de red cuenta con una memoria local. El tema de la memoria local se estudia en el apartado 9.6, aunque conviene explicarlo brevemente. La memoria local será el lugar donde los switches almacenarán de manera temporal las reglas de reenvío de paquetes que les comunica el controlador. Esta cantidad de memoria es vital ya que en caso de no ser suficiente, el dispositivo de red puede llegar a descartar paquetes, enviando al controlador aquellos con los cuales no sabe qué hacer. Para solucionar este problema existen varios métodos:

- Algoritmos de optimización de memoria local

- Prefijo común. No crear una regla distinta para cada pareja posible origen/destino, sino hacer agrupaciones
    - Ejemplo: Los paquetes que tengan como destino alguna dirección perteneciente a la subred 192.168.0.0/24 y 192.168.1.0/24 mandar por puerto 3. Dos opciones:
        - Crear una regla distinta por cada pareja posible origen/destino y reenviarla por el puerto 3 (muchísimas reglas)
        - Crear una sola regla buscando un prefijo común, es decir, agrupar las direcciones haciendo *supernetting*, jugando con la máscara y las direcciones IP y reenviarla por el puerto correspondiente. En este caso, poniendo como destino 192.168.0.0/23, disminuyendo la máscara, y reenviarlo por el puerto 3 sería suficiente (una única regla).

Capa de control

En esta capa se encuentra el componente más innovador de la arquitectura SDN, es decir, el controlador, encargado de gestionar la capa de aplicación y la capa de infraestructura. Además, tiene una visión general de toda la red y proporciona un sistema operativo de red (NOS). Lógicamente centralizado, su principal cometido es, mediante la programación, proporcionar inteligencia a los dispositivos que se encuentran en la capa de infraestructura, para que puedan realizar el reenvío de paquetes. El controlador se comunicará con los dispositivos de red (hardware puro) situados en la capa de infraestructura mediante la API hacia abajo, también conocida como *southbound* o *interfaz sur*, y que en este caso será el protocolo OpenFlow. Además, nos proporciona una API hacia arriba, también conocida como *northbound* o *interfaz norte*, para comunicarse con otras aplicaciones SDN, utilizando lenguajes de alto nivel como *Java*, *C++* o *Python*.

En esta capa es muy importante tener un controlador como '*backup'*, ya que en topologías de datacenter puede haber distintos tipos de fallos y, por lo tanto, causar la caída del controlador. Además, al estar el plano de control (comportamiento de cada uno de los routers) únicamente centralizado en un dispositivo, un mínimo fallo puede hacer que caiga la red por completo. Cabe destacar que esta opción no ha sido implementada en este TFG ya que es algo que ha quedado fuera del alcance del mismo.

Para implementar un plano de control no centralizado sino distribuido para OpenFlow, existe *HyperFlow* [8], que consiste en centralizar el plano de control lógicamente pero no físicamente. Es decir, todos los controladores tienen la misma información sobre la red pero esta información se replica en varios equipos físicos que actúan como controladores. Los dispositivos que se encuentran en red se comunican con el controlador más cercano, y si alguno fallase, no habría ningún problema, puesto que esa misma información estaría replicada en otro controlador. Lo único que cambia es el controlador de destino con el que se conectan algunos dispositivos para empezar a realizar las nuevas comunicaciones.

Con *HyperFlow*, además de conseguir hacer copias de los controladores para reaccionar ante fallos, implícitamente, la carga de mensajes switch-controlador controlador-switch se reduce drásticamente, ya que de esta manera cada switch se comunica con su correspondiente controlador. De la otra manera, sin utilizar HyperFlow, toda la comunicación entre switch-controlador controlador-switch se hace con un único controlador y, por lo tanto, la carga es mayor.

En la figura 4.4 podemos ver, por una parte, una SDN con un único controlador, que es el que se comunica con todos los nodos de la red, y, por otra parte, una SDN que utiliza HyperFlow compuesta por 5 controladores, donde cada dispositivo de red se comunica con su correspondiente controlador, según la distancia a la que se encuentra.

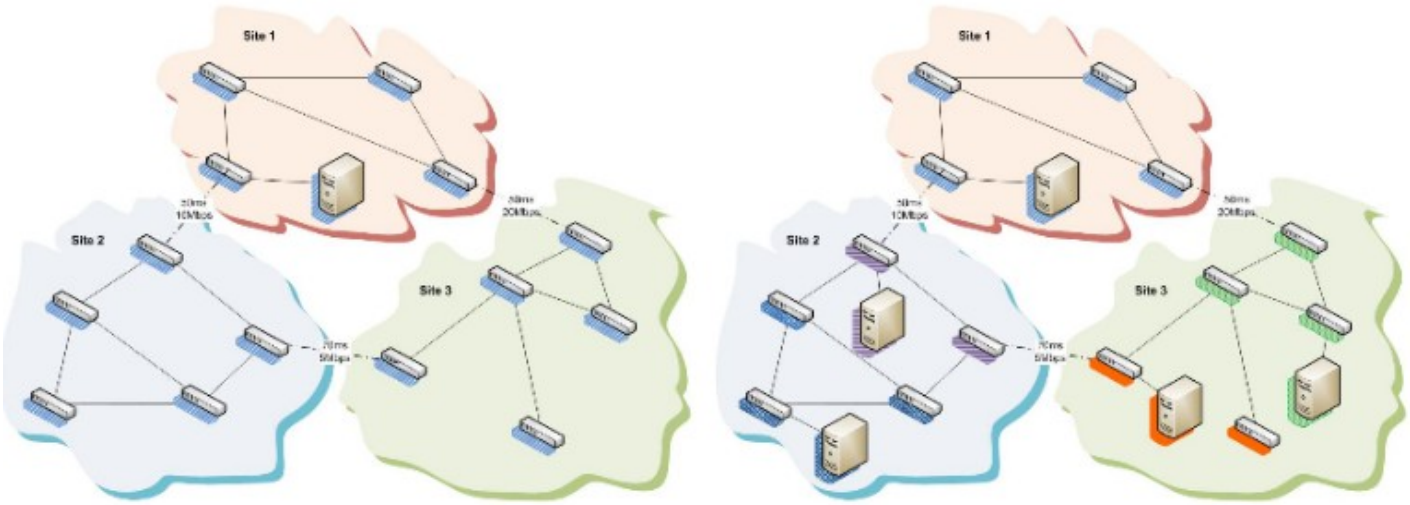

Figura 4.4: comparación de una red que no utiliza Hyperflow con una red que utiliza Hyperflow [8]

Capa de aplicación

Considerada también como la capa de nivel más alto o superior. Mediante esta capa se personaliza y se controla el comportamiento de la red. Para ello, se utiliza la API que proporciona el controlador y puede ser utilizada por los usuarios para satisfacer sus necesidades. Consiste en las aplicaciones de negocio para usuarios finales, que utilizan servicios de comunicación de SDN a través de la API hacia arriba. La API hacia arriba permite a los servicios y aplicaciones simplificar y automatizar tareas de configuración y gestionar nuevos servicios en la red. Mediante la API se pueden hacer aplicaciones interesantes para que los dispositivos realicen tareas de conmutación, encaminamiento, cortafuegos, conformado de tráfico o equilibrado de carga.

Asimismo, aplicaciones para la monitorización y la representación gráfica de cada una de las topologías creadas pueden ser interesantes. Mediante la API *northbound*, podemos obtener cual es el estado de la red, es decir, los nodos implicados en la red, ya sean hosts o switches y qué enlaces existen entre dichos elementos. Además, podemos obtener, por una parte, diferentes estadísticas que recogen los dispositivos que hay en red y, por otra parte, hacer configuraciones como por ejemplo, añadir/modificar/eliminar flujos posibles entre los posibles pares orígenes y destinos de la red, medir el ancho de banda entre nodos o hacer la configuración de los mismos.

Con esa información, haciendo uso del lenguaje de nivel alto que mejor nos manejemos, en mi caso, Python, y utilizando los correspondientes paquetes para la representación gráfica, podemos dibujar las topologías, por ejemplo. De esta manera, tendremos una mejor visión de la red y podremos detectar más fácilmente los fallos que puedan ocurrir.

4.5.2 Funcionamiento general de las tres capas

Las aplicaciones definidas en la capa de aplicación definen el uso que se le va a dar a la red y se lo comunican al controlador SDN. El controlador, según la información que recibe, tomará una decisión u otra y comunicará dicha decisión a los elementos de red que se encuentran en la capa de infraestructura, mediante la API southbound, en este caso, OpenFlow. Dado el rol fundamental que tiene el protocolo OpenFlow en las SDN, se describirá en detalle en la sección 5.

4.6 Beneficios de las SDNs

Un factor muy importante que hasta el momento no ha sido mencionado es el coste. En este TFG, al trabajar con material emulado y no con hardware real, el dinero que hubiera que invertir para implementar esta tecnología no se tiene en cuenta en ningún momento. Por ejemplo, no nos importa que un switch OpenFlow sea de 4 puertos que de 200. Pero, si hubiera que implementarlo en un caso real, la preocupación, casi seguro principal, que tendría una empresa sería el coste para implementar dicho proyecto. Para ello, aquí se indican algunos beneficios que nos ofrecen las SDNs ante las redes tradicionales [1]:

- Capacidad de reacción: Se puede reaccionar rápidamente ante cambios en la red
- Reducción de gastos económicos:
    - Posibilidad de reutilizar hardware disponible, si se permite la actualización de firmware
    - Dispositivos más baratos al no tener inteligencia integrada
- Simplificación en los dispositivos de red: Los elementos de red no tienen que procesar distintos protocolos estándares, simplemente deben aceptar las instrucciones que les envía el controlador
- Innovación: Permite crear prototipos personalizados e implementar ideas innovadoras dirigidas a empresas, según sus necesidades, beneficiándolas y aumentando el valor de sus redes
- Flexibilidad: La flexibilidad del software permite desarrollar funciones nuevas y servicios. Evitamos, por una parte, tener que añadir aparatos para añadir nuevas funcionalidades (cortafuegos, NAT…) a los switches OpenFlow, ya que estas funcionalidades se implementan por software, y, por otra parte, configurar equipamiento específico de la red
- Confiabilidad: Permite la monitorización de la red, además de predecir el comportamiento de la misma
- Uso de lenguaje de alto nivel: Permite el uso de lenguajes como Python, Java o C++

5. El protocolo OpenFlow

5.1 Definición

OpenFlow [9] es el estándar de código abierto más utilizado en la API *southbound* que fué creado a través del programa de investigación “Clean Slate Program” en la universidad de Stanford, situada en Palo Alto, California. Es el protocolo que permite la comunicación entre el plano de control y el de datos, para ser más exactos, la capa de control y la capa de infraestructura.

Este protocolo consiste en definir diferentes entradas en las tablas de flujo de los switches OpenFlow, asociándoles acciones distintas a cada una de las entradas, indicándole cómo debe procesar el flujo. Cada entrada, coincide con un subconjunto del tráfico y realiza ciertas acciones como descarte, reenvío, modificación o no hacer nada. Entonces, mediante las reglas o flujos definidos, un controlador puede instruir a un dispositivo de red para que se comporte como router, switch, NAT o cortafuegos, por ejemplo. Cabe destacar que, cuantas menos entradas tenga la tabla de flujo, el procesamiento de paquetes será más simple y, a la vez, más eficiente.

Podemos considerarlo como un elemento principal de las SDN. Además de OpenFlow, existen más protocolos como por ejemplo ForCES, OpFlex, NETCONF, XMPP o OVSDB [10]. Sin embargo, OpenFlow es el que se está convirtiendo en el módelo estándar en el área de las SDN. En la figura 5.1 podemos apreciar donde se encontraría el protocolo OpenFlow en la arquitectura de las SDN.

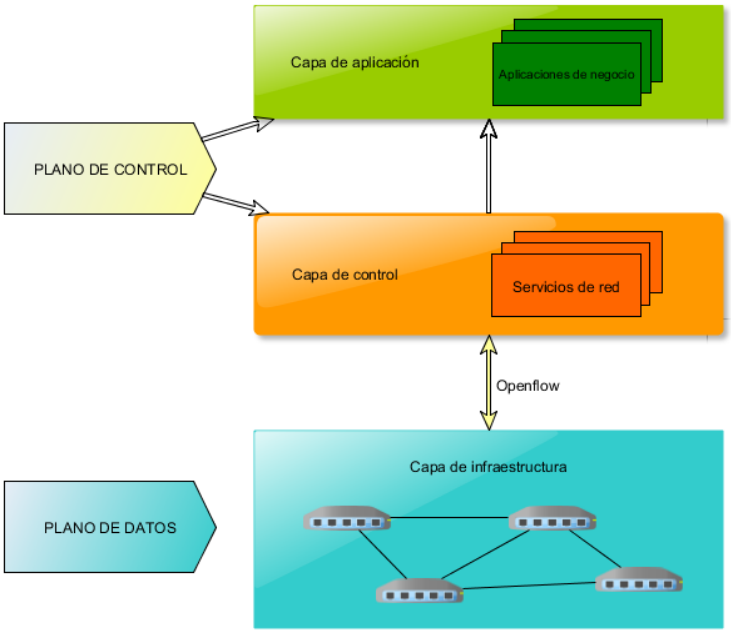

Figura 5.1: lugar donde se encuentra el protocolo OpenFlow en la arquitectura SDN [1]

5.2 Funcionamiento

El funcionamiento de este protocolo es bastante sencillo una vez que entendamos cuál es la función del controlador, que es el elemento que está conectado a cada uno de los switches OpenFlow y es el encargado de instruir a cada uno de ellos cómo se debe comportar. Además de la red física que conecta los elementos, debe existir una red de control que una a los switches OpenFlow con el controlador o controladores para que éste les indique a los switches cómo deben actuar.

En el momento que un paquete recibido por alguno de los switches no coincida (*matching*) con ninguna de las entradas de la tabla de flujo, el switch encapsula y reenvía ese mismo paquete al controlador y éste decide qué decisión debe tomar (esta acción es habitual cuando se trata de un primer paquete, hasta que se crea el flujo correspondiente que empareje ese paquete). En cambio, si el paquete recibido coincide con alguna de las entradas de la tabla de flujo, se procede a realizar la acción definida, ya sea reenviando el paquete por un puerto específico, ya sea reenviando el paquete al controlador porque así se ha decidido o simplemente descartando el paquete.

Para entender cuál es la composición de un flujo, consultar la figura 5.2. Un flujo está básicamente compuesto por tres campos elementales. La regla es el criterio utilizado para filtrar o identificar los paquetes que llegan al switch, la acción es el campo que determina la acción a realizar (cómo deben ser procesados los paquetes) con el grupo de paquetes identificados por la regla, y finalmente, las estadísticas, que es el campo que recoge diferentes estadísticas, para luego poder monitorizar información de los flujos. Entre las estadísticas, se recogen los bytes de cada flujo o el tiempo que lleva una entrada sin coincidir con ningún paquete, permitiendo borrar temporalmente flujos inactivos, para así aprovechar mejor la memoria local utilizada en los switches.

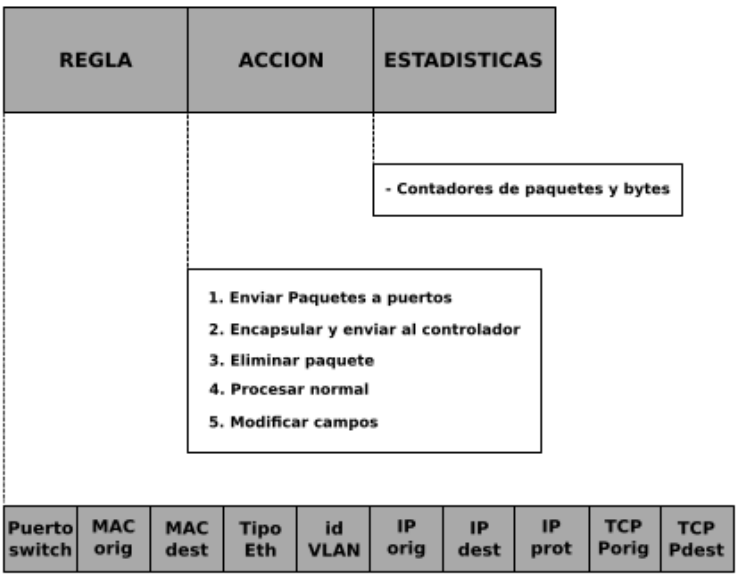

Figura 5.2: estructura de un flujo OpenFlow en diferentes campos [11]

En el ANEXO C se pueden consultar todas las reglas y acciones que se pueden llevar a cabo.

Para finalizar con el funcionamiento, se detallan cuáles son los mensajes que se intercambian entre los switches y el controlador. Primeramente, se explican los mensajes que manda el switch al controlador, en el momento que el switch recibe un paquete:

- Packet-in: Mensaje que se envía al controlador en el momento que a un paquete no se le encuentra coincidencia en alguna entrada de la tabla de flujo o la acción a llevar a cabo sea reenviar el paquete al controlador

- Flow-modify: Mensaje asíncrono que se envía al controlador para hacer saber a éste que se ha agregado una nueva entrada en la tabla de flujos

- Flow-timeout: Mensaje que envía cuando elimina un flujo por inactividad

- Port-status: Mensaje que se envía cuando el estado de un puerto ha cambiado

- Barrier reply: Manda información al controlador sobre operaciones completadas (respuesta a Barrier request)

- Error: Mensaje que se envía cuando ocurre un error

Los mensajes que el controlador manda al switch para conocerlo y actuar sobre el mismo son los siguientes:

- Modify-State: Mensaje enviado al switch para modificar el comportamiento de una entrada en la tabla de flujos del switch, añadir una nueva entrada, o eliminar una entrada existente

- Packet-out: Mensaje que manda al switch para decirle qué hacer con el paquete que no se le ha encontrado coincidencia en ninguna entrada de la tabla de flujo. Instruye al switch para enviar paquetes por el puerto indicado. Es la respuesta al mensaje *packet-in*.

- Barrier request: Pide información al switch sobre flujos activos u operaciones completadas

- Read-State: Pide información al switch sobre estadísticas

5.3 Ventajas

Mediante el protocolo OpenFlow, teniendo en cuenta que el plano de control está totalmente centralizado, es más sencillo configurar los distintos dispositivos que se encuentran en la red, pudiendo definir diferentes flujos para que los paquetes vayan de un host a otro, tanto estáticamente, definiéndolos de manera proactiva, como dinámicamente, definiéndolos de manera reactiva (estas dos configuraciones se estudiarán a fondo las secciones 6.1 y 6.2).

Como todos sabemos, en las redes tradicionales, en todo momento, se intentan evitar bucles y redundancia, por ejemplo, utilizando el protocolo Spanning Tree (STP). En este caso, la presencia de caminos redundantes, beneficia el tráfico, ya que nos permitirá distribuir distintos tipos de flujos de manera simultánea. Además, en caso de que alguno de los enlaces falle, es fácil redistribuir el flujo, mediante alguna aplicación, haciendo que el controlador instruya a los switches de forma inteligente para que busquen caminos alternativos y así poder hacer frente a problemas que puedan ocurrir.

5.4 Mercado

Poco a poco, algunas empresas están implementando la tecnología OpenFlow en sus propias redes de producción, empresas con tanta fuerza como Google o Yahoo. Pero, para que este protocolo sea implementado, es necesario que los dispositivos de red (hardware) sean capaces de simular el protocolo OpenFlow. Afortunadamente, son muchas las empresas que están ofreciendo soluciones para el soporte de esta nueva tecnología, como por ejemplo, HP, IBM, Juniper, NEC o Cisco [7].

A día de hoy, son dos los tipos de switches compatibles con OpenFlow. Por una parte, los switches OpenFlow-híbridos, que son aquellos switches que soportan el protocolo OpenFlow, pero además, se comportan como un switch Ethernet convencional. Por otra parte, encontramos los switches OpenFlow-only, que son aquellos switches que únicamente soportan el protocolo OpenFlow y solamente se utilizan en redes que emplean la tecnología SDN.

6. El controlador POX

El controlador empleado para desarrollar este TFG es POX [12]. Se encargará de instruir a los switches OpenFlow emulados en las topologías que se utilizan en datacenter, proporcionando inteligencia a cada uno de los switches mediante la API *southbound* OpenFlow y aceptando la ejecución de aplicaciones mediante la interfaz norte, para finalmente, realizar el enrutamiento de paquetes.

Además del controlador POX, hay otras opciones como NOX, OpenDayLight o RYU. Pero, en este TFG, se ha decidido trabajar con POX por ser uno de los más sencillos de entender y manejar usando Python. El controlador POX cuenta con las siguientes características:

- Funciona sobre cualquier sistema operativo
- Una vez instalado, contiene código y componentes reutilizables como:
    - Algoritmos que buscan el camino mínimo entre nodos [13]:
        - Bellmanford
        - Dijkstra
    - Diferentes comportamientos del switch:
        - Hub (fichero *hub.py*)
        - L2 learning switch (comportamiento switch, fichero *l2_switch.py*)
        - L3 learning switch (comportamiento router, fichero *l2_switch.py*)
    - Protocolos aplicables a los elementos de red:
        - Spanning Tree Protocol (STP)
- Ofrece una interfaz *northbound* en Python, que permite la programación de switches OpenFlow mediante este lenguaje de alto nivel

El controlador es el encargado de gestionar toda la red de control, red que une todos los switches y el controlador. Existen tres maneras principales de programar el controlador, se puede configurar de manera reactiva, proactiva y mixta [14] (mezcla entre la configuración reactiva y proactiva).

6.1 Configuración reactiva

Si queremos programar la red de control de manera reactiva, la política es la siguiente:

Los switches, inicialmente son tontos, diremos que la tabla de flujos de cada uno de los switches se encuentra vacía, y por lo tanto, un switch no sabe qué hacer con los paquetes entrantes. Cuando no sabe qué hacer con un paquete, el switch manda al controlador un mensaje *Packet-in*. En el momento que el controlador reciba ese mensaje, el controlador instruirá al switch con un mensaje *Packet-out*, diciéndole qué debe hacer con ese paquete. Además, instruirá al switch sobre qué hacer cuando en el futuro reciba paquetes similares, mediante la instalación de una entrada en su trabla de flujos (mensaje M*odify-State*). De esta manera, en un futuro, si el switch recibe un paquete perteneciente a una entrada de la tabla de flujos, podrá realizar la acción indicada sin tener que volver a preguntar al controlador. Si más adelante encuentra algún paquete que no coincida con ninguna entrada, se repite el proceso de mandar un *Packet-in* al controlador. De esta manera, según se van viendo nuevos flujos, el controlador va poblando las tablas de los switches. Tras un tiempo, la mayoría de los paquetes son reenviados sin la intervención del controlador. Al principio, esta manera es un tanto ineficiente, pero pronto se sabe la suficiente información para realizar un reenvío eficiente.

En la figura 6.1 podemos ver que los switches son instruidos dinámicamente (sobre la marcha) una vez éstos no hayan encontrado una coincidencia en alguna de las entradas de la tabla de flujo y hayan mandado un mensaje *Packet-in* al controlador. A partir de recibir el *Packet-out,* PC1 y PC2 podrán intercambiar paquetes.

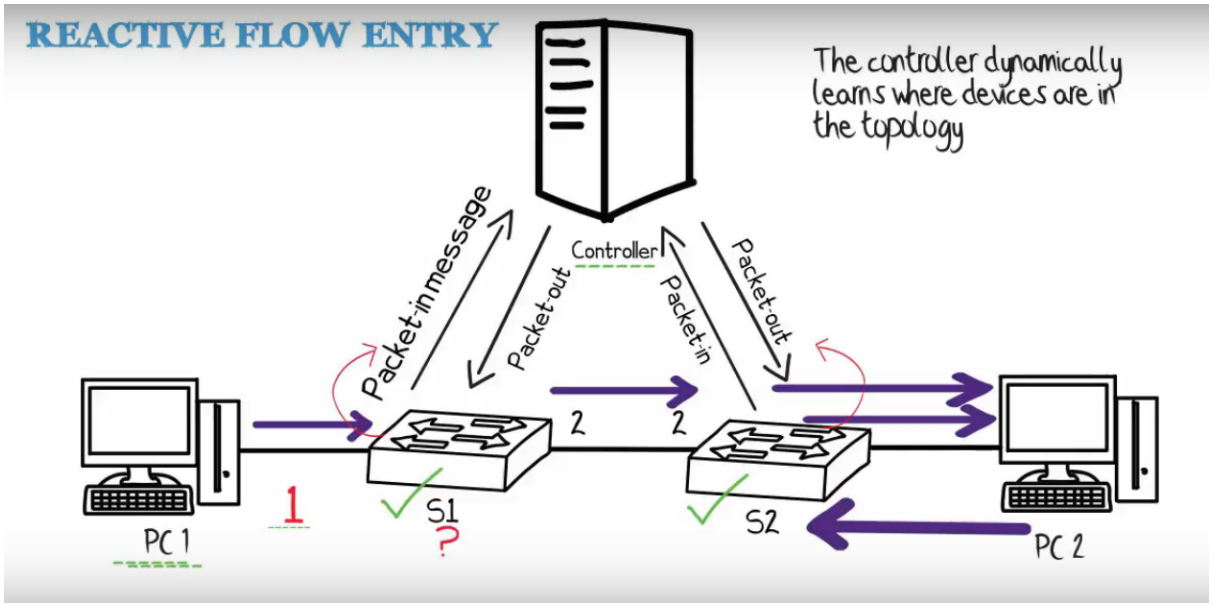

Figura 6.1: instalación reactiva de los flujos [14]

6.2 Configuración proactiva

Si queremos programar la red de control de manera proactiva, la política es la siguiente:

El controlador, de antemano, sabe todas las parejas posibles origen-destino, y por lo tanto, puede calcular todos los flujos necesarios. Entonces, antes de que se reciba ningún paquete, el controlador instruye a todos los switches, cargando todos los flujos que a priori va a haber. Cuando empieza el intercambio de paquetes entre los diferentes host, los switches ya saben qué es lo que tienen que hacer con los paquetes que reciben, encuentran a qué entrada de la tabla corresponde el paquete y reenvían el paquete, según las acciones indicadas. De esta forma, evitaremos reducir considerablemente el número de paquetes que van y vienen por la red de control, ya que apenas habrá paquetes que no coincidan con ninguna entrada de la tabla de flujos, de forma que no se procede a preguntar al controlador qué hacer con los paquetes desconocidos.

En la figura 6.2 podemos ver que los switches son instruidos de antemano, sin que haya un mensaje *Packet-in*. A partir de ahí, PC1 y PC2 podrán intercambiar paquetes.

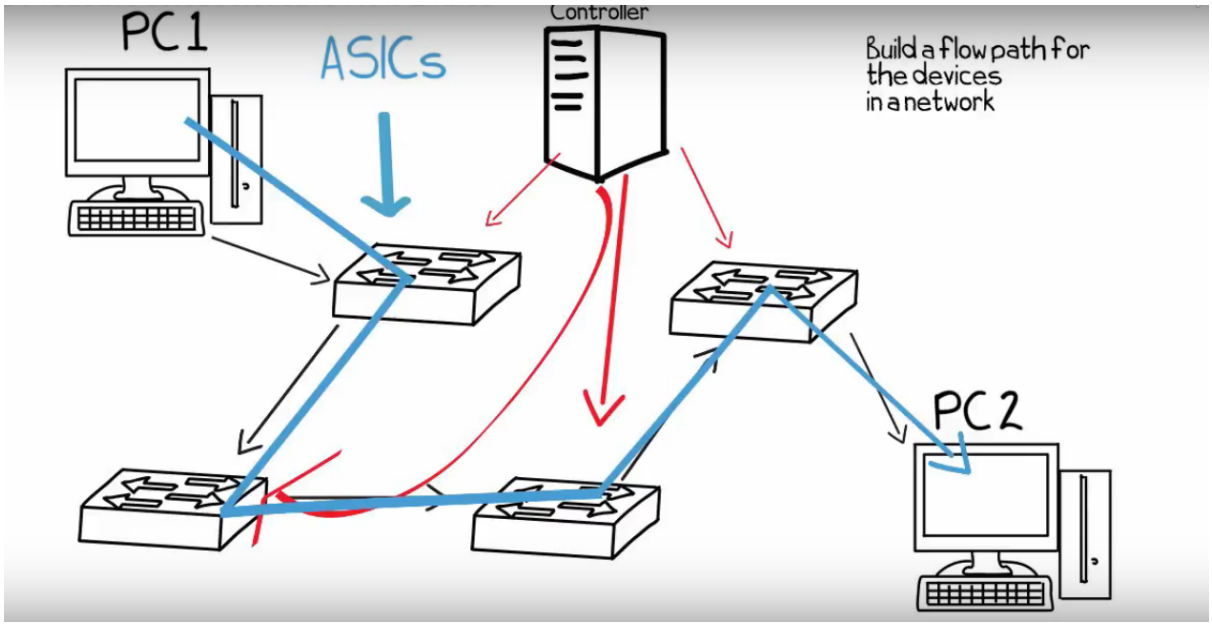

Figura 6.2: instalación proactiva de los flujos [14]

En este TFG, el controlador se implementará de manera proactiva. De hecho, la estructura de las topologías orientadas a datacenter que se construirán se conocen de antemano. Por lo tanto, sin que haya comunicación previa entre hosts, se conocen todos los caminos posibles y podremos cargar, a priori, todos los flujos que los switches se van a encontrar y qué acciones se aplicarán a cada uno de los flujos. Además, en algunos casos, será posible optimizar las tablas de flujo, reduciendo el número de entradas existes de las mismas. Para ello, será vital hacer una gestión de direcciones IP adecuada para después poder agrupar de manera sencilla

diferentes subconjuntos de direcciones. Trabajar de forma reactiva supondría un número enorme de flujos y unos retardos muy elevados para los primeros paquetes de cada flujo.

6.3 Instalación proactiva de los flujos

El controlador POX trabaja de manera concurrente, es decir, para cada comunicación que se establece entre switch-controlador, POX crea un hilo distinto, ejecutando un proceso distinto en paralelo. De esta manera, se consigue que las tablas de flujo de los switches OpenFlow se completen rápidamente y simplifica la vida al programador.

En el controlador se ha programado una función llamada *install*, cuyo cometido es, según los parámetros que le pasemos, instalar una nueva entrada en la tabla de flujo con el switch que se está comunicando. Por cada entrada nueva que se quiera instalar en un switch tendremos que hacer lo siguiente:

- Preparar el paquete para que se instale como una nueva entrada (paquete tipo *ofp_flow_mod)*
- Añadir la coincidencia para el paquete que hemos preparado (campo *match*)
- Añadir la prioridad para el paquete que hemos preparado (campo *priority)*
- Añadir la acción a realizar con los paquetes a los cuales se les encuentre coincidencia (campo *actions*)
- Una vez preparado el paquete tipo *ofp_flow_mod*, enviar dicho paquete al switch para que quede instalado el flujo (función *send*)

Para finalizar con esta sección, se muestra la función *install* implementada en cada uno de los controladores utilizados:

```
 def install(self, port, match, priority):


    msg = of.ofp_flow_mod() # preparar el paquete msg para que posteriormente
se instale como una nueva entrada en la tabla de flujos del switch OpenFlow
correspondiente

    msg.match = match # anade el matching que pasamos como parametro a ese
paquete msg. Por ejemplo, direccion IP destino (nw_dst) o puerto de entrada
(in_port)

    msg.priority = priority # anade la prioridad que pasamos como parametro a
esa paquete msg

    msg.actions.append(of.ofp_action_output(port = port)) # anade la accion a
realizar con las direcciones IP o puertos entrantes que coincidan con esta
entrada, para reenviarla por el puerto port que pasamos como parametro

     self.connection.send(msg) # el controlador envia el mensaje msg al switch
para instalar el flujo en el switch OpenFlow
```

7. Instalación y configuración de las herramientas necesarias

Antes de empezar a trabajar con las diferentes topologías de datacenter, es necesario instalar y configurar las herramientas necesarias. En todo momento, se trabajará dentro de una máquina virtual, para así evitar desperfectos posibles en mi ordenador personal, así como problemas con servicios existentes que se están ejecutando. En este apartado, se explicará cómo se ha instalado y configurado la máquina virtual donde se trabajará. Asimismo, se explicará, por una parte, cómo se ha instalado *Mininet*, que será el entorno de emulación donde trabajaremos, ya que no trabajaremos con hardware real, y por otra parte, la instalación del controlador *POX*, el controlador escogido para realizar este trabajo. Para finalizar, se muestra la instalación del complemento *RipL*, complemento que será muy útil para generar las topologías.

Mi computadora personal tiene instalado un SO Ubuntu-16.04 de 64 bits, mientras que la máquina virtual se instalará con un SO Ubuntu-14.04-server-amd64.

7.1 Instalación y configuración máquina virtual

Para crear la máquina virtual he utilizado el software de virtualización "*VMware Workstation Player*". Para realizar la correspondiente instalación he seguido los siguiente pasos:

- Descargar el *.zip* que nos proporciona *Mininet,* que contiene la imagen VM (formato *.ovf*), un X server y el programa SSH, para que nos permita manejar por completo el servidor mediante un intérprete de comandos desde mi ordenador.

  https://github.com/mininet/openflow-tutorial/wiki/Installing-Required-Software

- Escoger la opción de 32 o 64 bits, según nuestro sistema operativo.

- Abrir el programa VMware Player y "*create a new virtual machine*".

- Importar el archivo *.ovf*

- Escoger el SO que se instalará: Linux versión ubuntu-64 bits.

- Escoger el tamaño máximo que ocupará la máquina virtual en memoria, en GB: 20 GB (recomendado).

En este punto, tendríamos la máquina virtual instalada y el usuario y contraseña para acceder a ella será:

usuario: mininet
contraseña: mininet

Pero ahora, necesitamos configurar los adaptadores de red virtuales para que, por una parte, la máquina virtual pueda conectarse internamente con el ordenador físico, y, por otra parte, para que pueda conectarse al exterior vía NAT. Por lo tanto, necesitaremos configurar dos adaptadores de red, uno para cada comunicación citada.

En mi caso, los dos adaptadores de red virtuales se configurarán de la siguiente forma:

- eth0 (172.16.221.128): La interfaz eth0 se configurará como "*host-only*" para la comunicación interna entre host y máquina virtual

- eth1 (192.168.212.135): La interfaz eth1 se configurará como "*NAT*", para acceder desde la máquina virtual a Internet

Una vez iniciemos la sesión en la máquina virtual tendremos que escribir en la terminal los dos siguientes comandos para que los adaptadores se configuren como tal:

sudo dhclient eth0 (host only, 172.16.221.128)
sudo dhclient eth1 (NAT, 192.168.212.135)

Sin embargo, hay una manera muy sencilla de automatizar este proceso mediante *DHCP*, para evitar tener que configurar los adaptadores cada vez que iniciemos sesión. Tendremos que editar y añadir las siguientes líneas en el archivo de configuración "*/etc/network/interfaces"*:

```
#Red interna, host-only
auto eth0
iface eth0 inet dhcp


#Para conectarnos a internet vía NAT
auto eth1
iface eth1 inet dhcp
```

La configuración de los adaptadores se hace mediante la pestaña "*Virtual Machine* / *Virtual Machine Settings"* de *VMWare Player*. En la figura 7.1 podemos ver la correspondiente configuración de cada adaptador.

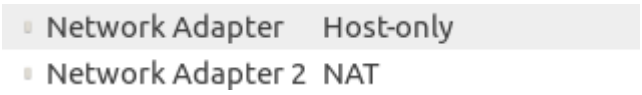

Figura 7.1: configuración de los adaptadores de red

7.2 Instalación de Mininet y POX

La máquina virtual que nos proporciona Mininet mediante *github* incluye todos los componentes necesarios. De hecho, trae preinstalada la versión 2.2.0 de *Mininet* [15] y el controlador *POX* [12], con sus necesarios complementos, lista para trabajar con la creación de topologías mediante Mininet y poner en marcha algunas aplicaciones de control que nos ofrecen como ejemplo. Cabe destacar que, en este TFG, aunque fundamentalmente se vaya a utilizar el controlador *POX*, también se dedicará un apartado específico para mostrar el potencial del controlador *OpenDayLight*, ya que lo considero interesante.

No obstante, las versiones de *Mininet* y *POX* resultarán un problema más adelante, ya que la herramienta *RipL* está desarrollada en versiones anteriores de *Mininet* y *POX*, distintas a las que vienen preinstaladas. Para afrontar ese problema, habrá que modificar partes del RipL, para adaptar este complemento al API de Mininet 2.2.0, y así conseguir que los complementos que necesitemos funcionen de manera correcta y eficiente.

7.3 Instalación del escritorio gráfico

La máquina virtual que nos proporciona Mininet, no proporciona ningún tipo de escritorio gráfico, y, por lo tanto, para conectarnos a ella tendríamos que hacerlo en todo momento a través de *ssh*, desde una terminal del host, concretamente, de la siguiente manera:

```
ssh usuario@IP_máquina_virtual_interna
```

Esta opción puede resultar con el tiempo un tanto agotadora, por ello, se ha decidido instalar como complemento una interfaz gráfica en la máquina virtual.

Primeramente, nos conectamos a la máquina virtual vía *ssh*, así:

```
ssh mininet@172.16.221.128
```

Una vez tengamos el control de la máquina virtual remotamente, instalaremos los siguientes paquetes necesarios:

```
sudo apt-get install xinit

sudo apt-get install lxde

sudo apt-get install virtualbox-guest-dkms
```

Tras la instalación de estos paquetes, iremos a la ventana donde se está ejecutando la máquina virtual (pestaña *VMware player*), y escribiremos los dos siguientes comandos:

```
sudo su # pasamos a modo superusuario

startx # lanzamos el gestor de ventanas lxde
```

A continuación, podremos manejarnos en la máquina virtual a través del escritorio gráfico. Es bastante simple, pero suficiente para los trabajos que tenemos que llevar a cabo. En la figura 7.2 podemos ver cuál es el resultado una vez lanzado el comando *startx:*

Figura 7.2: Escritorio gráfico donde trabajaremos

De este punto en adelante, por lo tanto, nos ahorraremos tener que conectarnos a la máquina virtual vía *ssh* para realizar nuestras tareas cada vez que iniciemos sesión. Por comodidad, todo se hará mediante el escritorio gráfico que se ha instalado.

7.4 Instalación del controlador ODL

La instalación del controlador *OpenDayLight* [16] no es necesaria, pero, al proporcionar la visualización gráfica de las topologías lanzadas con Mininet, ya sea por medio de línea de comando o ya sea mediante scripts en Python, considero que puede resultar un complemento bastante útil e interesante. En líneas generales, este complemento nos ayudará a entender mejor la estructura de la red y los cambios que pueda suponer la misma.

La versión del controlador *OpenDayLight* que se instalará será la versión "*Berryllium karaf 0.4.0"*. En nuestra máquina virtual, al tener los adaptadores de red configurados para conectarnos a Internet vía NAT, no tendremos ningún problema para descargar los paquetes necesarios.

- Descargar el siguiente paquete comprimido mediante una terminal con el comando *wget* en la ruta que queramos*:*

```
wget
https://nexus.opendaylight.org/content/groups/public/org/opendaylight/integration/distribution-karaf/0.4.0−Beryllium/distribution-karaf−0.4.0−Beryllium.tar.gz
```

- Descomprimir el paquete recién descargado:

```
tar -xvf distribution-karaf-0.4.0-Beryllium.tar.gz
```

- Colocarnos donde se encuentra el ejecutable que arranca el controlador y ejecutarlo:

```
cd /distribution-karaf-0.4.0-Beryllium/bin
./karaf
```

Esperar a que cambie el entorno de trabajo. Una vez haya cambiado el entorno de trabajo significa que se está ejecutando el controlador *OpenDayLight* y está esperando a que se lanze una topología para reconocerla y visualizarla. Antes de lanzar una topología, algo que debemos hacer cada vez que arranquemos el controlador es instalar unos mínimos paquetes para que *OpenDayLight* pueda trabajar (por ejemplo: Para que los switches detectados funcionen como switches de nivel 2 o para tener acceso a la *API Yang* mediante el navegador, que será la encargada de representar la estructura de la red de manera gráfica).

En la figura 7.3 podemos observar cómo cambia el entorno de la terminal al ejecutar *karaf*, así como cuáles son los paquetes mínimos que tenemos que instalar cada vez que ejecutemos el controlador *OpenDayLight* y cómo se procede a instalarlos.

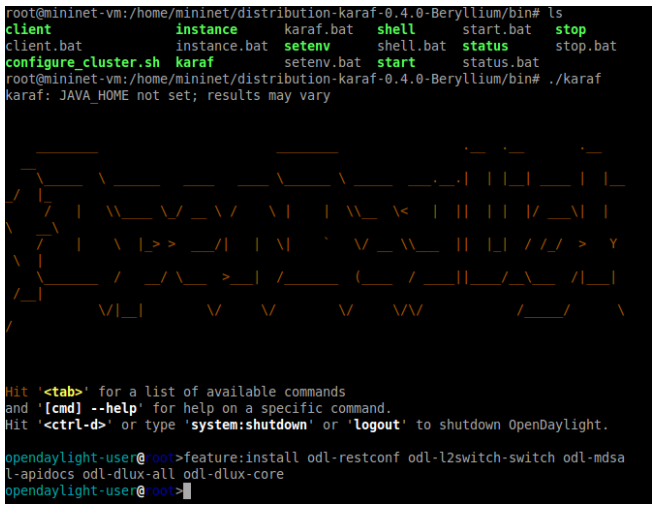

Figura 7.3: entorno al iniciar el controlador OpenDayLight e instalación de paquetes mínimos

En este caso, todos los paquetes se han instalado mediante un único comando, por comodidad. No obstante, si se quieren hacer de manera independiente, la sintaxis es la siguiente:

Para instalar: *feature:install complemento*
Ejemplo: *feature:install odl-restconf*

Por último, solamente nos queda explicar cómo conectarnos al controlador mediante el navegador para poder visualizar las topologías lanzadas mediante Mininet. La aplicación se llama *OpenDaylight Dlux* (instalada mediante el paquete *odl-dlux-all)* y para conectarnos a ella, deberemos abrir nuestro navegador preferido y en la barra de direcciones escribir lo siguiente:

http://IP_eth0:puerto/index.html

- IP_eth0 = dirección IP de eth0, habilitada para la comunicación con el host. En mi caso, la dirección correspondiente es la 172.16.221.128

- puerto = puerto definido por la aplicación. En este caso, por defecto, el 8181

En la figura 7.4 se muestra la interfaz que nos ofrece la aplicación una vez especifiquemos la dirección mencionada en la barra de direcciones. Cabe destacar que para iniciar sesión en la misma deberemos ingresar un usuario y una contraseña. Tanto el nombre de usuario como la contraseña son "*admin".*

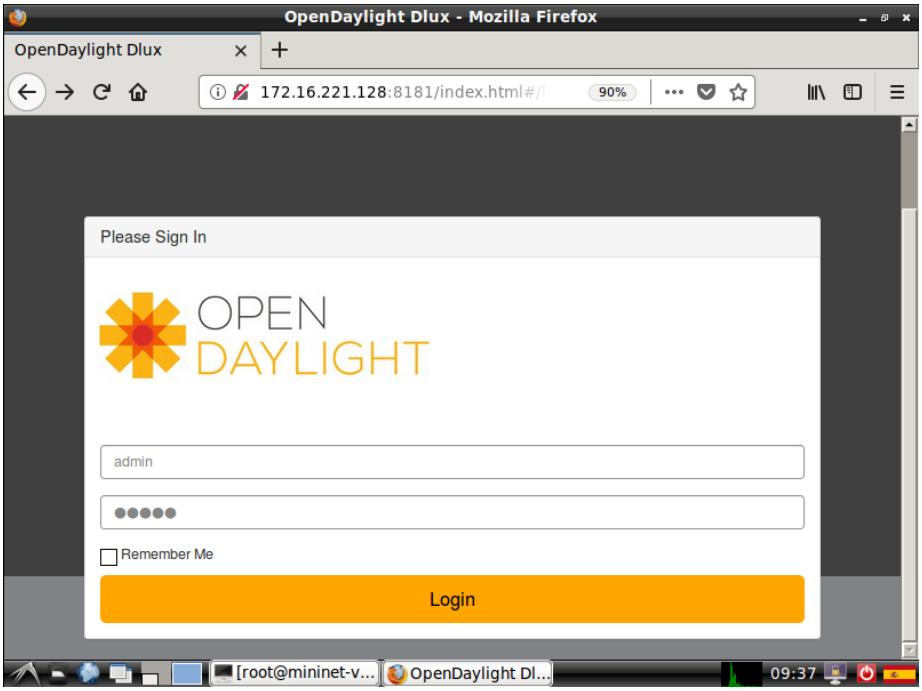

Figura 7.4: interfaz que nos ofrece la aplicación

No conviene mucho profundizar en este tema pero si dos características destacables. Por una parte, una vez iniciemos sesión, en el caso de que haya una topología lanzada, dicha estructura de red podremos verla a través de la pestaña "*topology*". De esta manera, podremos analizar fácilmente qué cambios puede haber en la red en caso de que se creen, modifiquen o se eliminen enlaces y dispositivos de red. Por otra parte, mediante la pestaña "*nodes*" podemos ver la información de cada uno de los nodos, switches para ser exactos, que se encuentran en la red: El identificador del nodo (*Node ID*), el nombre de nodo (*Node Name*), las conexiones con otros nodos (*Node Connectors*) y las estadísticas sobre tablas de flujo y paquetes transmitidos (*Statistics*).

Finalmente, existen varios modos para terminar con la ejecución del controlador *OpenDayLight*. Escribiendo cualquiera de los siguientes comandos en la terminal donde se está ejecutando el controlador será suficiente:

- *halt*
- *logout*
- *system:shutdown*

Con lo instalado hasta este punto se han conseguido construir y trabajar con topologías de complejidad no muy alta y no justamente orientadas a topologías utilizadas en centros de datos, que es el principal objetivo de este TFG. Sin embargo, la experiencia adquirida con estas herramientas ha sido vital para después poder desarrollar el trabajo posterior a la fase aprendizaje.

Finalmente, se hará una breve descripción de una herramienta interesante que he utilizado para trabajar con las redes utilizadas en los centros de datos, además de su correspondiente instalación.

7.5 Instalación de la herramienta RipL

RipL (*Ripcord-Lite*, también conocido como "*ripple*") [17] es una librería de Python para simplificar la creación de topologías utilizadas en los centros de datos, sin hacer grandes cambios en el código. El código de esta librería proviene de un proyecto anterior llamado *Ripcord*, un proyecto que se creó con el fin de crear y controlar topologías de datacenter estructuradas de manera sencilla.

La instalación para poner en marcha esta herramienta es bastante sencilla, simplemente hay que seguir los siguientes pasos:

1. Instalar el paquete *python-setuptools*. Es una librería de python utilizada para la instalación de software. En nuestro caso, nos permitirá instalar la herramienta mediante un fichero llamado *setup.py*

   ```
   sudo apt-get install -y python-setuptools
   ```

2. Descargar en la ruta correspondiente la herramienta RipL

   ```
   git clone git://github.com/brandonheller/ripl.git
   ```

3. Colocarnos en el directorio *ripl* que se acaba de crear

   ```
   cd ripl
   ```

4. Proceder a instalar la herramienta *RipL* a través del fichero *setup.py* mediante *python-setuptools*

   ```
   sudo python setup.py develop
   ```

8. Mininet

Mininet [18] es la herramienta escogida para llevar a cabo todos los experimentos en este TFG. Es una herramienta de código abierto para la emulación y prototipado de Redes Definidas por Software disponible para los sistemas operativos Linux, Os X y Windows. Mininet crea, en cuestión de segundos, una red virtual "realista" y proporciona un camino fácilmente adaptable para la migración de hardware.

Al ser una herramienta de código abierto y por lo tanto, al alcance de muchos usuarios, gracias a la colaboración de una gran cantidad de investigadores, cuenta con gran soporte de la comunidad investigadora, documentación oficial, desarrollo constante y mejora continua. De esta manera, se ha convertido en la herramienta más popular para la creación y prototipado de las topologías basadas en la tecnología SDN.

La herramienta incluye switches convencionales y switches *OpenFlow*, routers convencionales y routers *OpenFlow*, enlaces virtuales y una colección de hosts para poder crear una estructura de red completa. Al estar hablando de un emulador, permite diseñar, construir y hacer pruebas sin generar gasto económico alguno. Además, los proyectos creados son completamente escalables y repetibles.

Algo que deberemos tener muy en cuenta es que, al no trabajar ni con hardware real ni con comportamientos reales de la red no se podrá medir el rendimiento de forma fiable (la distribución de la carga no dependerá de las aplicaciones que se utilicen sobre la red emulada, se tomará en cuenta que la carga creada por cada host que conforma la red es el mismo, el tráfico será puramente artificial). Además, la fluidez y la eficiencia con la que trabajará este emulador dependerá de manera directa de los recursos que ofrece el ordenador y de la configuración y características de la máquina virtual donde se ha instalado.

Para terminar con esta breve introducción a la herramienta Mininet, en la figura 8.1 se muestran tres softwares libremente disponibles para la experimentación con SDNs que existen en el mercado y qué diferencias tienen entre sí:

	ESTINET	NS-3	MININET
Modo de simulación	Sí	Sí	No
Modo de emulación	Sí	No	Sí
Compatible con controladores reales	Sí	No	Sí
Resultado repetible	Sí	No	Sí
Escalabilidad	Alta (para un solo proceso)	Alta (para un solo proceso)	Media (para múltiples procesos)
Exactitud en los resultados de rendimiento	Sí	No admite protocolo de árbol y ningún controlador real	Sin fidelidad de rendimiento
Soporte de GUI	Para configuración y observación	Sólo para observación	Sólo para observación

Figura 8.1: características de algunos entornos de experimentación con SDNs [19]

En este apartado, sin profundizar en todos los comandos constructores que ofrece Mininet (para ello se hará uso del apéndice ANEXO A) para distintos fines, se mostrarán cuáles son los tres métodos fundamentales que nos ofrece esta herramienta para la creación de diferentes topologías, que es lo que realmente nos interesa. En algunos casos, se resaltarán aquellas características o comandos que se consideren remarcables. Además de explicar cómo se crean las topologías con *Mininet*, se harán unas simples demostraciones con los controladores *POX* y *OpenDaylight* para entender el funcionamiento de los mismos y demostrar la conectividad entre los diferentes nodos de la estructura de red creada.

Tenemos que tener claro que son tres las maneras que nos ofrece *Mininet* para la creación de topologías:

- Línea de comandos (topologías integradas en *Mininet*)
- Miniedit
- Scripts en lenguaje de nivel alto (topologías personalizadas)

8.1 Creación de las topologías

Para crear topologías mediante línea de comandos, debemos abrir una terminal. El comando para lanzar la topología es el comando "*mn"*. La estructura del comando "*mn"* es la siguiente, sin los corchetes:

sudo mn --[OPCIÓN] = [PARAMETRO],[ARGUMENTOS] --[OPCIÓN_n] = [PARAMETRO_n], [ARGUMENTOS] ...

Todas las opciones, parámetros y argumentos existentes para el comando "*mn"* se muestran en el ANEXO A. Aquí, solamente se mostrarán las necesarias.

OPCIÓN TOPO

Esta opción nos permite crear las topologías que Mininet ya tiene integradas. Las topologías (*PARAMETRO*) que nos ofrece son las siguientes, y vienen acompañadas por sus correspondientes argumentos (*ARGUMENTOS*).

- *sudo mn --topo=minimal* o, simplemente, *sudo mn*

  Genera una topología simple de 1 switch al que se le conectan 2 hosts.

- *sudo mn --topo=single,k*

  Genera una topología de 1 switch al que se le conectan *k* hosts. Asigna los puertos del switch en orden ascendente.

- *sudo mn –topo=linear,k,n*

  Genera una topología de *k* switches en serie, donde *n* es el número de hosts conectados a cada uno de los switches.

- *sudo mn --topo=reversed,k*

  Genera una topología similar a la de single. La diferencia es que asigna los puertos del switch en orden descendente.

- *sudo mn --topo=tree,depth,fanout*

  Genera una estructura en forma de árbol simple compuesta por '*fanout'* ramas y de profundidad '*depth'*. El número de hosts conectados a cada uno de los switches depende del '*fanout'* que definamos.

- *sudo mn --topo=torus,x,y*

  Genera una topología toro 2D, donde *x* son los switches en la dimensión *x*, e *y*, los switches en la dimensión *y*. A cada uno de los switches se le añade un único host. Existe la posibilidad de añadir un tercer parámetro *n*, que define el número de hosts conectados a cada switch.

OPCIÓN CONTROLLER

- *sudo mn –topo=XX --controller=remote,ip,port*

  Permite el uso de un controlador externo a Mininet compatible con *OpenFlow,* ya sea *POX* o *ODL,* donde '*ip*' es la dirección donde se está ejecutando el controlador y '*port*' el puerto. Esta es la opción que utilizaremos. En mi caso, no se tienen que indicar esos dos parámetros puesto que tengo automatizado el proceso de reconocer el controlador que se está ejecutando, en cuanto se lanza una topología

8.1.1 Topologías que nos ofrece Mininet

En este pequeño experimento, se muestra cómo poner en marcha una topología en árbol de dos niveles (*depth*=2) y cuatro ramas (*fanout*=4). Además, se pone en funcionamiento el módulo *hub.py* que nos ofrece el controlador *POX*, para demostrar la conectividad entre los diferentes host. En este caso, los switches *OpenFlow* se comportarán como hub, es decir, el paquete que reciben por un puerto, lo reenvían por todos los puertos excepto por el puerto que han recibido el paquete. Finalmente, se hará uso del controlador *ODL* (*OpenDayLight*) para tener una representación gráfica de la topología creada.

Los pasos a seguir para completar este experimento son los siguientes:

1. Abrir dos terminales

2. Desde una terminal, ejecutar el controlador *POX* con el módulo *hub*

```
root@mininet-vm:/home/mininet# ./pox/pox.py forwarding.hub
POX 0.0.0 / Copyright 2011 James McCauley
INFO:forwarding.hub:Hub running.
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.6/Nov 23 2017 15:49:48)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY.  This program is free software,
and you are welcome to redistribute it under certain conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
Ready.
```

Figura 8.2: poner el controlador en marcha cargando el módulo *hub.py*

Lanzamos el controlador *POX* (*pox.py*) situado en el directorio *POX*, y le pasamos como parámetro el módulo *hub* ubicado en la carpeta *forwarding*. En este momento, el controlador está a la espera de que se lance una topología, y una vez se lance, ir instruyendo a cada uno de los switches OpenFlow para que se comporten como un concentrador Ethernet.

3. Desde otra terminal, lanzamos la topología mencionada con las opciones y parámetros adecuados

```
root@mininet-vm:/home/mininet# sudo mn --topo=tree,2,4 --switch ovsk --controller remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s1, s5) (s2, h1) (s2, h2) (s2, h3) (s2, h4) (s3, h5) (s3, h6) (s
3, h7) (s3, h8) (s4, h9) (s4, h10) (s4, h11) (s4, h12) (s5, h13) (s5, h14) (s5, h15) (s5, h1
6)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
mininet>
```

Figura 8.3: lanzar la topología *tree,2,4*

4. Nos aseguramos de que el controlador ha reconocido la topología

```
root@mininet-vm:/home/mininet# ./pox/pox.py forwarding.hub
POX 0.0.0 / Copyright 2011 James McCauley
INFO:forwarding.hub:Hub running.
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.6/Nov 23 2017 15:49:48)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY.  This program is free software,
and you are welcome to redistribute it under certain conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
Ready.
POX> INFO:openflow.of_01:[Con 1/None] closing connection
INFO:openflow.of_01:[Con 4/2] Connected to 00-00-00-00-00-02
INFO:forwarding.hub:Hubifying 00-00-00-00-00-02
INFO:openflow.of_01:[Con 3/5] Connected to 00-00-00-00-00-05
INFO:forwarding.hub:Hubifying 00-00-00-00-00-05
INFO:openflow.of_01:[Con 2/1] Connected to 00-00-00-00-00-01
INFO:forwarding.hub:Hubifying 00-00-00-00-00-01
INFO:openflow.of_01:[Con 5/3] Connected to 00-00-00-00-00-03
INFO:forwarding.hub:Hubifying 00-00-00-00-00-03
INFO:openflow.of_01:[Con 6/4] Connected to 00-00-00-00-00-04
INFO:forwarding.hub:Hubifying 00-00-00-00-00-04
```

Figura 8.4: comprobar que se ha establecido la conexión entre el controlador y los switches

5. En la ventana que hemos lanzado la topología, mediante el comando *'pingall'* de Mininet, demostrar la conectividad entre los nodos que conforman la red.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12 h13 h14 h15 h16
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12 h13 h14 h15 h16
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12 h13 h14 h15 h16
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 h16
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 h16
h14 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 h16
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h16
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15
*** Results: 0% dropped (240/240 received)
```

Figura 8.5: comprobar la conectividad entre todos los host con todos mediante el comando pingall

Una vez llegados a este punto, la herramienta de administración de switches OpenFlow "*ovs-ofctl o dpctl*" puede resultarnos muy útil, ya que, entre otras cosas, nos muestra la tabla de flujos actual del switch que indiquemos. Todo el potencial que nos ofrece esta herramienta se recoge en el ANEXO C, en caso de que se quiera profundizar en su uso. En la imagen 8.6 se muestra la tabla de flujos del switch s2.

```
root@mininet-vm:/home/mininet# sudo ovs-ofctl dump-flows s2
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1046.874s, table=0, n_packets=1059, n_bytes=75042, idle_ag
e=751, actions=FLOOD
```

Figura 8.6: entradas que hay en la tabla de flujo del switch s2

Para terminar con este experimento se muestra cómo visualizar la topología lanzada mediante el controlador *ODL*. Para ello, por una parte, tendremos que terminar el proceso que lanza el controlador mediante el comando *exit()* o con *ctrl-c* y, por otra parte, terminar el proceso que lanza la topología mediante el comando *exit* (en el prompt de Mininet).

Para poner el controlador *ODL* en marcha, seguiremos los pasos definidos en la sección "INSTALAR EL CONTROLADOR OpenDayLight EN LA MÁQUINA VIRTUAL" y, una vez tengamos el controlador en marcha, se instala un módulo que hace que los switches OpenFlow se comporten igual que los switches Ethernet convencionales. Después, lanzaremos la topología con Mininet y la visualizaremos, en mi caso, mediante el navegador Mozilla Firefox. En la imagen 8.7 se muestra la topología *tree,2,4* lanzada mediante Mininet, constituída por dos niveles, cuatro ramas y cuatro host por cada swtich.

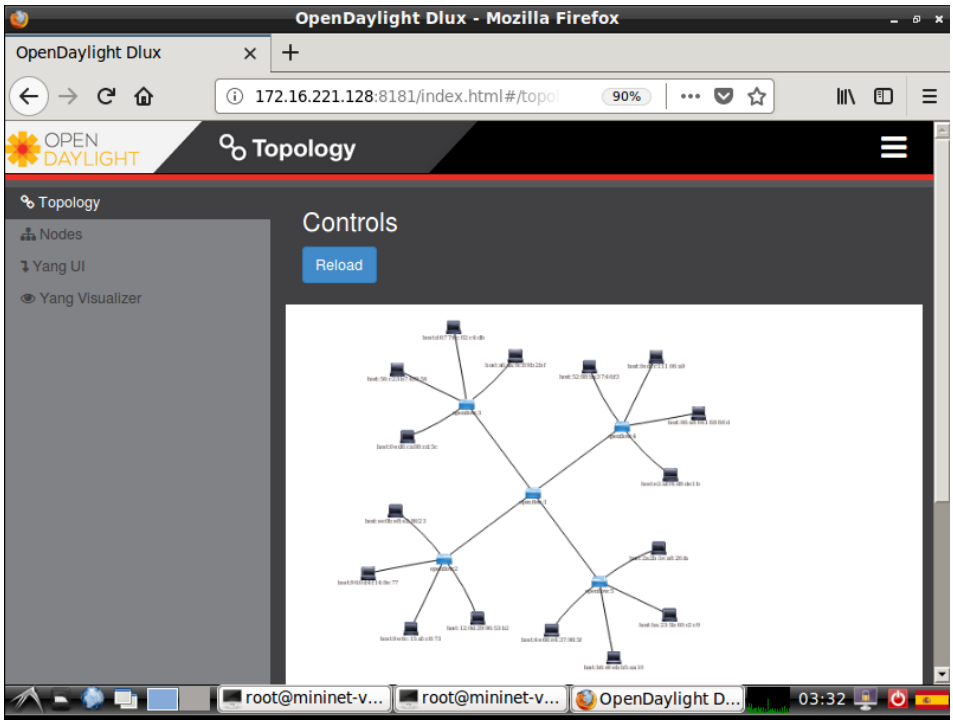

Figura 8.7: topología *tree,2,4* visualizada a través del controlador ODL

En esta sección, no solo hemos aprendido cómo lanzar topologías mediante línea de comandos, sino también varias herramientas y conceptos que nos serán realmente útiles cuando nos adentremos en las topologías orientadas a los centros de datos. Este primer método ha sido explicado con mayor profundidad puesto que había conceptos que anteriormente no habían sido explicados y para darnos cuenta del potencial que nos ofrecen tanto Mininet como los controladores. Los dos siguientes métodos se explican de manera más general, sin tanto detalle.

8.1.2 Topología creada con Miniedit

Mininet nos proporciona una herramienta que nos permite crear topologías mediante una interfaz gráfica. El programa para diseñar dichas topologías se llama *miniedit.py*, situado en la carpeta */mininet/examples*.

Para ejecutar el programa *miniedit*, situado en la ruta */home/mininet/mininet/examples* tendremos que escribir el siguiente comando:

```
./home/mininet/mininet/examples/miniedit.py
```

En este punto, se nos abrirá una ventana donde se nos permite dibujar la topología que queramos y hacer la configuración necesaria de los elementos que conformarán la red. Cuando demos por finalizada creación, tendremos que exportar la topología con la opción "*File/Export Level 2 Script",* darle un nombre y guardarla con la extensión *.py,* para que posteriormente pueda ser lanzada. En la imagen 8.8 se muestra la ventana que nos permite crear una topología con *Miniedit.*

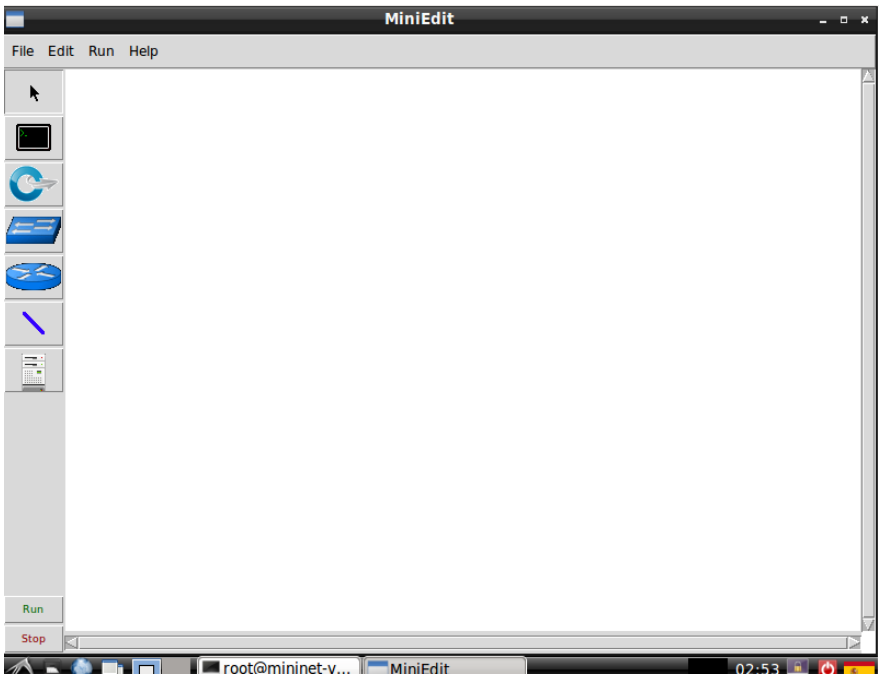

Figura 8.8: Miniedit. Entorno para crear topologías de manera gráfica

En la imagen 8.9 se muestra como ejemplo una topología tipo *hipercubo* creada mediante el programa *Miniedit*. Simplemente, consiste en dibujar la topología creando los host, switches, en este caso de tipo *OpenFlow,* y controladores y realizar las conexiones virtuales correspondientes entre los elementos. El programa mismo se encarga de generar el script que hemos dibujado para que posteriormente la topología pueda ser lanzada con *Mininet*. Además, podemos observar que, por una parte, las líneas azules representan las conexiones físicas entre los distintos dispositivos (switch-to-switch y switch-to-host) y por otra parte, las líneas rojas representan la red de control (conexiones controlador-to-switch).

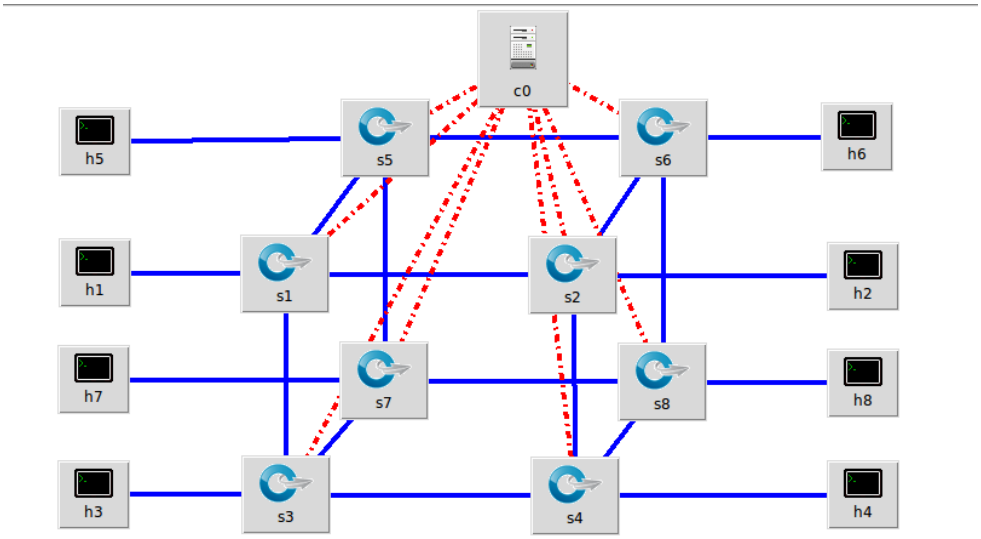

Figura 8.9: topología tipo hipercubo dibujada con Miniedit

Para ver el script que *Miniedit* ha generado de manera automática, consultar el ANEXO D.

Una vez se haya generado el script automáticamente, la estructura del comando para lanzar la topología creada mediante *Miniedit* es la siguiente:

sudo python topologia_creada.py
En este caso concreto: *sudo python hipercubo.py*

8.1.3 Topología creada mediante un script en Python

Finalmente, se muestra cómo lanzar una topología implementada mediante un script escrito en lenguaje de alto nivel, en este caso, Python. Para esta sección, como ejemplo, se utiliza el fichero *FatTree_IP.py* (en el ANEXO D, se muestra el código correspondiente para generar la topología, así como la gestión de direcciones IP). En qué consiste la topología *Fat-Tree* se explicará más adelante, en su correspondiente sección, aquí solamente se pretende enseñar cómo lanzar las topologías implementadas mediante scripts en Python.

Una vez tengamos el script Python, lanzar dicho script es muy fácil. Al ya conocido comando "*sudo mn"* le tendremos que añadir la siguiente opción:

- --custom=[PARAMETRO]

  Lee archivos de configuración escritos en Python. Dicho archivo debe tener la extensión *.py*. Según en qué ruta nos encontremos, tendremos que escribir la ruta relativa o absoluta del fichero.

- --topo=[Variable]

  Nombre que recibe la topología para referirse a sí misma, en este caso "*fattree"*. Esta variable se define en el fichero donde está descrita la topología, en este caso, en el fichero *FatTree_IP.py,* concretamente, en la variable *topos*.

En la imagen 8.10 se muestra cual es la estructura del comando para lanzar el script y su ejecución:

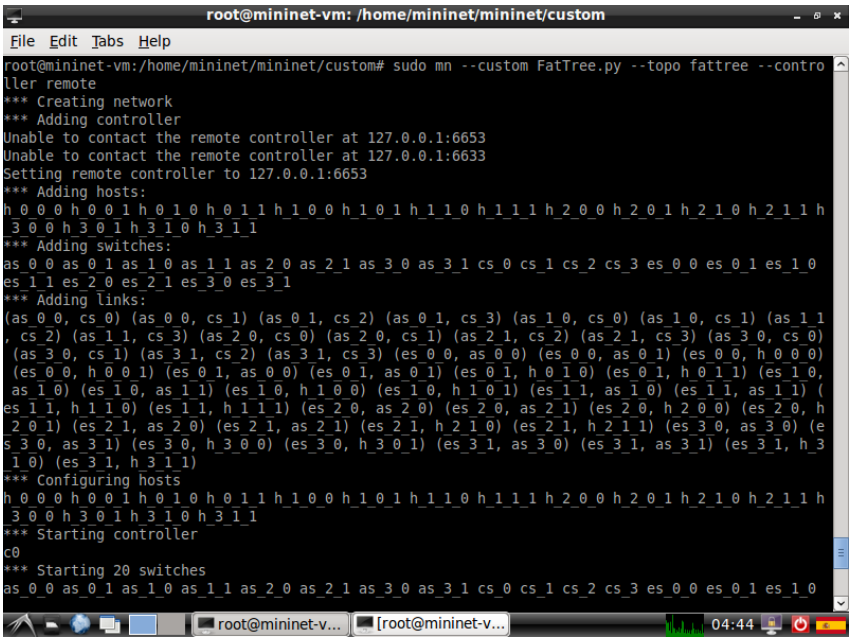

Figura 8.10: estructura del comando para lanzar una topología definida en un script

Cabe destacar que una vez se inicia una emulación mediante el comando *sudo mn*, Mininet dispone de una lista de comandos *shell* para ejecutarlos tanto sobre la topología lanzada como en los dispositivos emulados. Para consultar todos los comandos disponibles consultar el ANEXO B.

Una vez llegado a este punto, ya conocemos todo el entorno de trabajo y podemos empezar a analizar las distintas topologías orientadas a datacenter que se han decidido implementar en este TFG. Entre otras cosas, se analizarán los siguientes puntos: Características de cada una de las topologías, eficiencia y coste de cada una de las topologías implementadas, criterio de gestión de direcciones de cara al routing que se vaya a utilizar, routing para las topologías, comparación entre las topologías y problemas que puedan presentar cada una de ellas.

9. Topologías para datacenter

En este apartado se explican cuáles han sido las topologías de datacenter que se han generado en el entorno de emulación Mininet. Para cada caso, se concreta de qué trata cada una de los topologías, explicando los puntos débiles y fuertes de las mismas. Las topologías que se han estudiado han sido, por una parte, las topologías estructuradas de familia *tree* y, por otra parte, la topología aleatoria *Jellyfish.* Para llevar a cabo esta fase del proyecto, *RipL* ha sido una herramienta útil para generar las topologías. Por otra parte, el routing proactivo se ha implementado a través de unos módulos para *POX* realizados totalmente como parte de este TFG.

Como se ha indicado anteriormente, la gestión de direcciones de los host es un tema imprescindible de cara a que después el routing se haga de manera más eficiente posible. Para cada caso, se analizará cuál ha sido el criterio tomado para asignar las direcciones IP.

9.1 Topologías familia árbol

En este TFG, se han implementado tres casos específicos que corresponden a una estructura de red regular como es la familia árbol (*tree*). Para ser exactos, los tres casos que se analizarán son los *simple-tree, thin-tree y fat-tree*. Antes de entrar en detalle con cada uno de los casos, conviene explicar cuál ha sido la política establecida y qué parámetros se han tenido que definir para generar los mencionados árboles.

En este caso, todos los árboles generados constan de tres capas compuestas por diferentes tipos de switches y otra capa adicional compuesta por los hosts a los cuales proporcionaremos servicio y demostraremos la conectividad entre los mismos. Para ello, será muy importante gestionar las direcciones de los hosts de manera adecuada. Además, una característica muy a tener en cuenta es que la estructura de red de los árboles es totalmente regular, por lo que nos facilitará el trabajo a la hora de hacer el routing.

Las diferentes capas compuestas por los switches son las siguientes: Capa superior o capa *core*, capa intermedia o capa de *agregación* y capa inferior o capa *borde.* El objetivo de la capa borde será dar acceso a los hosts y, por lo tanto, será la capa que se encargue de unir los switches con los hosts. La capa borde tiene dos funciones principales, por una parte, intercambiar paquetes si el host de destino está conectado al mismo switch (comunicación local), es decir, unir nodos conectados al mismo switch borde, y, por otra parte, encargarse de mandar paquetes a la capa intermedia cuando el paquete recibido tenga como destino un host conectado a otro switch. Las otras dos capas, la capa core y la de agregación, serán las encargadas de proporcionar distintos caminos para permitir la comunicación entre los host que se sitúan en distintas subredes. Naturalmente, cada caso específico de las topologías de familia árbol proporcionará distintos caminos para conectar los hosts entre sí y tendrán sus ventajas y desventajas en cuanto a coste y rendimiento.

Entre las distintas propiedades que puede tener un switch, nos encontramos lo que se llama el *grado* del switch, es decir, el número de puertos que tiene. Las topologías tipo árbol, al considerarse una estructura de red regular jerárquica, permite que el grado de los switches se pueda definir de la siguiente manera:

$$grado\_switch = L + K$$

donde:

- L = up-ports (puertos hacia arriba)

- K = down-ports (puertos hacia abajo)

Por lo tanto, definiendo, por una parte, el parámetro 'K' como los puertos hacia abajo de un switch y, por otra parte, el parámetro 'L' como los puertos hacia arriba de un switch, podremos generar las distintas topologías, dando los valores adecuados a los parámetros 'K' y 'L'. Cabe destacar que el parámetro 'K', además de representar los *down-ports*, también representa el número de ramas (*pods*) que tiene el árbol. Definir el número de ramas del árbol ha sido necesario para poder acotar la anchura del árbol, sea del tipo que sea y poder identificar cada nodo a la rama correspondiente.

En la imagen 9.1 se detalla cuántos puertos hacia arriba y cuántos hacia abajo tienen cada uno de los switches OpenFlow, según la capa a la que corresponden los switches, así como la estructura de red genérica de las topologías de la familia árbol.

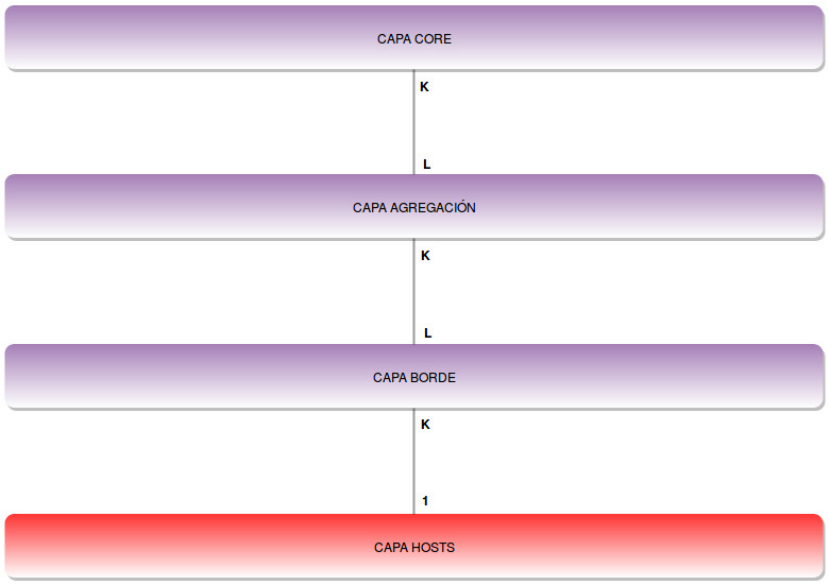

Figura 9.1: estructura de red genérica de las topologías familia *tree*

Capa core: Capa únicamente compuesta por switches OpenFlow. Cada switch de esta capa tendrá 'K' puertos hacia abajo hacia diferentes switches de la capa agregación. Al no haber ninguna capa por encima de ella, no tendrá ningún puerto hacia arriba y por lo tanto, el grado de los switches que conforman esta capa es el siguiente:

grado_switch: 0 + K = K (down-ports)

Cabe destacar que en este TFG la comunicación entre los hosts es totalmente interna, es decir, el intercambio de paquetes se hace entre nodos que se encuentran dentro del mismo datacenter. No se hace ninguna comunicación al exterior. En el caso de que se quisieran hacer comunicaciones al exterior, habría que conectar los switches de la capa *core* a routers externos para poder comunicarnos a Internet, por ejemplo.

Capa agregación: Capa únicamente compuesta por switches OpenFlow. Situada entre la capa *core* y *borde*. Cada switch de esta capa tendrá 'L' puertos hacia arriba hacia diferentes switches de la capa *core* y 'K' puertos hacia abajo hacia diferentes switches de la capa *borde*. Por lo tanto, el grado de los switches de esta capa es el siguiente:

grado_switch: L + K (up-ports + down-ports)

Capa borde: Capa únicamente compuesta por switches OpenFlow. Situada entre la capa de agregación y la capa hosts. Cada switch tendrá 'L' puertos hacia arriba hacia diferentes switches de la capa agregación y 'K' puertos hacia abajo hacia diferentes hosts. El grado de los switches será el mismo que el de los switches que conforman la capa de agregación:

grado_switch: L + K (up-ports + down-ports)

Capa hosts: Capa únicamente compuesta por hosts emulados en Mininet. Cada host estará conectado a su correspondiente switch OpenFlow de la capa *borde*.

Una vez se han analizado las características de las topologías de familia tree, el número de switches OpenFlow que componen cada capa, dependen de manera directa de los parámetros 'K' y 'L'. En la tabla 9.1 se indican, definiendo los parámetros 'K' y 'L', por una parte, el número de switches que componen cada capa, y por otra parte, el número de hosts que se les proporciona servicio.

Tabla 9.1: ecuaciones y ejemplos para calcular el número de switches y hosts

	capa *core*	capa *agregación*	capa *borde*	capa *hosts*
tree,K,L	L^2	K * L	K^2	K^3
tree,8,3	$3^2 = 9$	8 * 3 = 24	$8^2 = 64$	$8^3= 512$

Ahora, una vez definidos cuántos switches y host se crean para cada caso específico, es necesario determinar cuántos enlaces virtuales se necesitan para generar la topología. Los enlaces son conexiones que se hacen, por una parte, entre las interfaces de los switches que se encuentran en las diferentes capas, y, por otra parte, entre los switches de la capa *borde* y los hosts. La fórmula para obtener el número de enlaces para una topología tipo árbol es la siguiente:

$$num\_enlaces = capa\_edge * K + capa\_agregacion * K + capa\_borde * K$$

donde, *capa_edge, capa_agregacion* y *capa_borde* son el número de switches que necesitamos para cada capa, calculados en la tabla 9.1. Siguiendo el ejemplo anterior quedaría así:

$$num\_enlaces = 9 * 8 + 24 * 8 + 64 * 8 = 776$$

Por lo tanto, para generar un *tree,8,3* necesitaríamos los siguientes componentes: 9 switches para la capa *edge*, 24 switches para la capa *agregación*, 64 switches para la capa *borde* y 776 enlaces, por una parte, para conectar los switches entre sí, y, por otra parte, para conectar los 512 host a los switches de la capa *borde*.

Una vez llegados a este punto, nos ha quedado claro que modificando los valores de 'K' y 'L' obtendremos los distintos casos para las topologías de la familia *tree*, así como su respectivo número de switches y enlaces para generar la topología. Para generar las topologías partiremos de un fichero que contiene la herramienta *RipL (mn.py)* y el comando típico para lanzar topologías en Mininet, "*sudo mn"*.

Primeramente, nos colocaremos en el directorio ~/ripl/ripl , ya que los script para lanzar las diferentes topologías se encuentran en ese directorio:

```
cd ~/ripl/ripl
```

Para lanzar las topologías tendremos que utilizar el fichero *mn.py*. En este fichero, se importan todas las clases que se utilizan para generar topologías. Dichas clases se encuentran en el fichero *dctopo.py*, el cual se analizará posteriormente. En la imagen 9.2 se muestra el contenido del fichero *mn.py*:

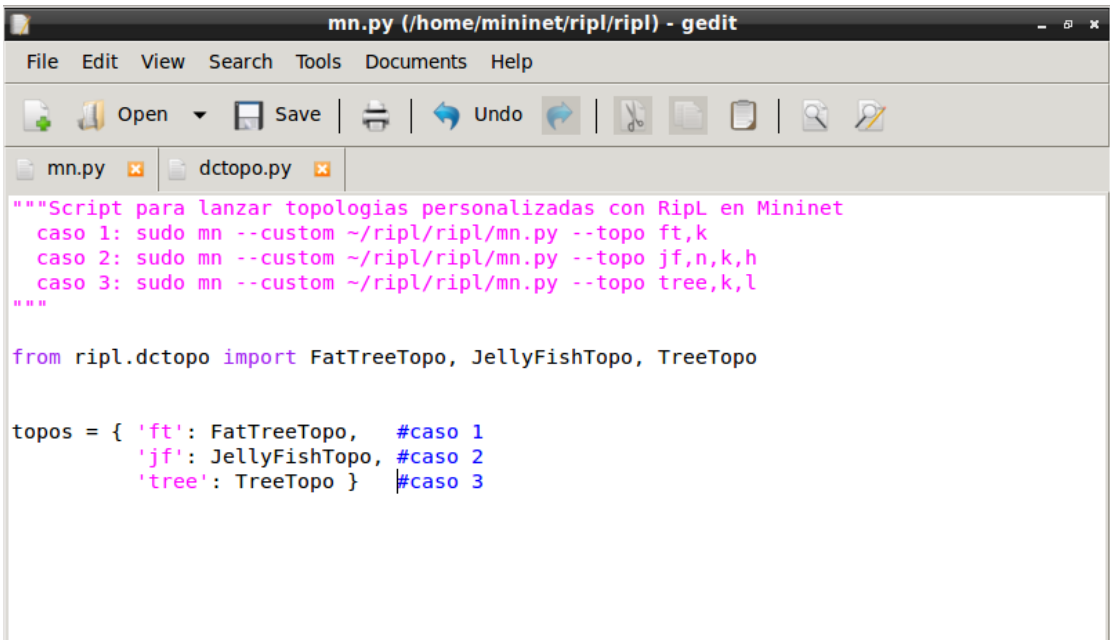

Figura 9.2: fichero *mn.py*

Como podemos ver en los comentarios de la anterior imagen, la estructura del comando para lanzar las topologías es la siguiente:

sudo mn --custom mn.py --topo topos,parametros

Recordemos que la opción *topo* recibe como parámetro un nombre que le hemos dado a la variable *topos* en el script para referirse a la topología misma. En el script vienen comentados qué parámetros reciben cada una de las topologías. En nuestro caso, como nos interesa lanzar una topología de la familia *tree* tendremos que escribir lo siguiente:

sudo mn --custom mn.py --topo tree,K,L

donde:

- K = número de puertos hacia abajo de los switches

- L = número de puertos hacia arriba de los switches

Para entender mejor el funcionamiento de la herramienta RipL, en la tabla 9.2 se muestran varios ejemplos de qué es lo que realmente se ejecutaría al escribir los distintos comandos:

Tabla 9.2: funcionamiento comandos

Comando	Acción
sudo mn --custom mn.py --topo jf,10,6,10	Ejecuta la clase *JellyFishTopo* que se encuentra en el fichero *dctopo.py* del directorio *ripl* pasando como parámetro '10,6,10'
sudo mn --custom mn.py --topo tree,8,2	Ejecuta la clase *TreeTopo* que se encuentra en el fichero *dctopo.py* del directorio *ripl* pasando como parámetro '8,2'

- *JellyFishTopo* es la clase que genera una topología *JellyFish* según los parámetros que le pasemos. La topología *Jellyfish* se analizará posteriormente.

- *TreeTopo* es la clase que genera una topología específica de la familia *tree,* dependiendo de los parámetros que le pasemos.

De aquí en adelante, se analizan las características de los diferentes *tree* que podemos generar dando los valores correspondientes a los parámetros K y L.

9.1.1 Árbol simple

La estructura de red del árbol *simple*, como el nombre mismo lo dice, es la topología más simple de toda la familia de árboles. De hecho, es la topología que menos switches y enlaces requiere para generarla, y, por lo tanto, podemos pensar que el coste de la misma sea muy bajo. No obstante, el gran problema de esta topología es la alta utilización de los enlaces, que pueden llegar a saturarse. Asimismo, si en alguno de los nodos hubiere algún problema, ese problema podría hacer que la red falle por completo, ya que no hay caminos alternativos y el tráfico de datos está muy centralizado. Resumiendo, podemos decir que este caso específico es de coste muy bajo pero a la vez muy ineficiente.

Esta topología está pensada para explotar totalmente la localidad, es decir, está pensado para que se comuniquen nodos que se sitúan en la misma subred. Por tanto, si se hacen conexiones de manera simultánea hacia una subred distinta, probablemente tengamos problemas de latencia, puesto que los enlaces que nos llevan hasta la otra subred deberán ser compartidos.

CARACTERÍSTICAS DE LOS SWITCHES

Todos los switches OpenFlow que conforman esta red contarán con el mismo número de puertos hacia abajo (parámetro K), pero el número de puertos hacia arriba de cada uno de los switches, es decir, el valor que recibirá el parámetro 'L', será de 1. Por lo tanto, sustituyendo la 'L' por su correspondiente valor, conseguiríamos la siguiente estructura de red:

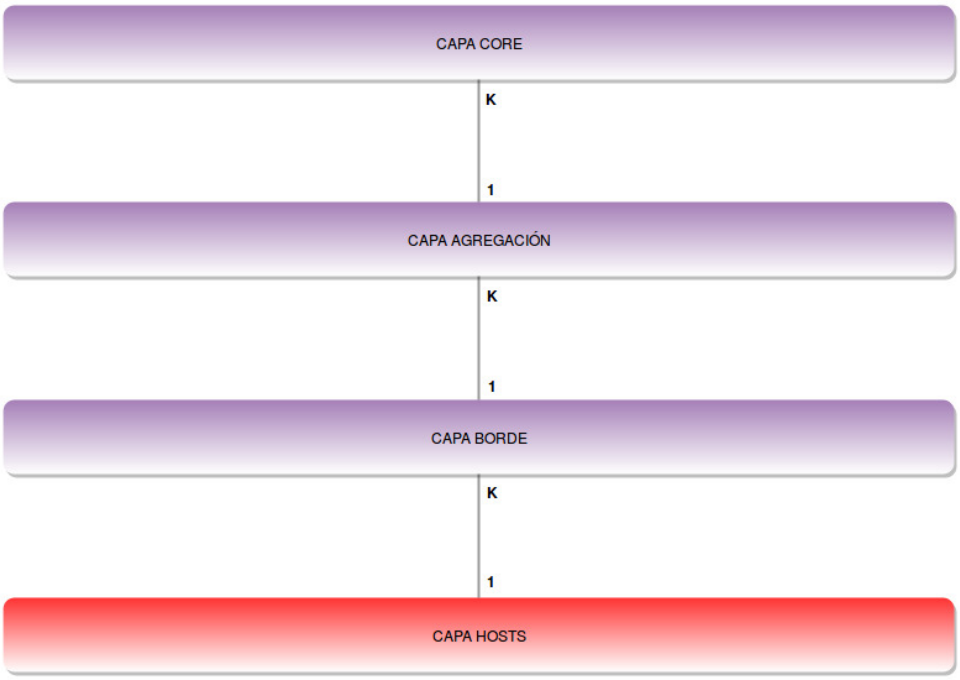

Figura 9.3: estructura de red genérica de la topología árbol simple

En la imagen 9.4 se recoge un ejemplo de un *tree,4,1*. Además, se calculan los switches que necesitaremos por capa y el número de enlaces para generar dicha topología.

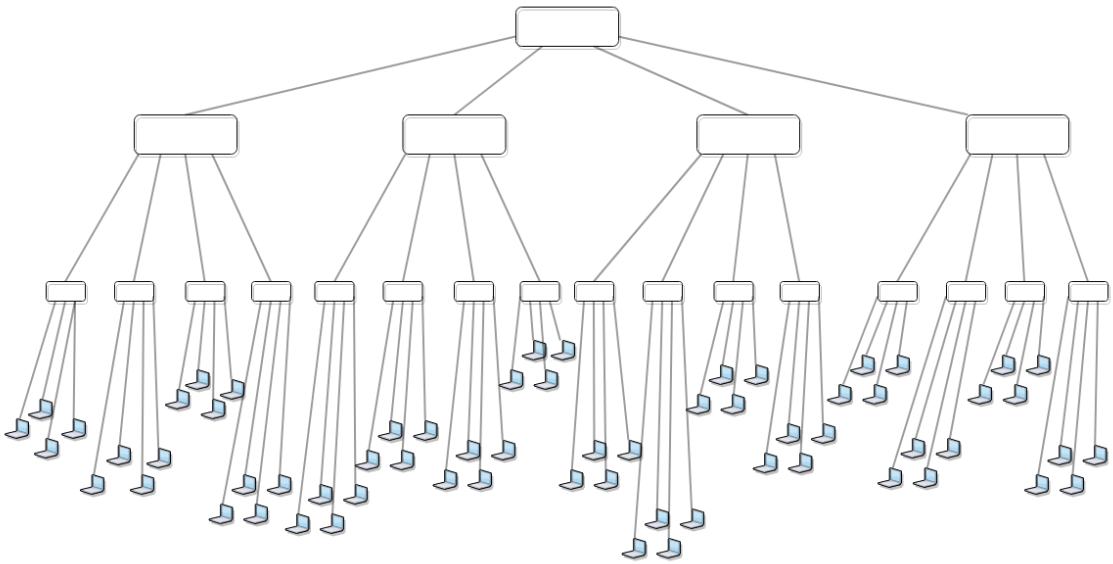

Figura 9.4: ejemplo topología *tree,4,1*

Cálculo de los switches que necesitamos (tabla 9.3):

Tabla 9.3: Cálculo de número de switches y hosts para el caso *tree,4,1*

	capa core	capa agregación	capa borde	capa hosts
tree,K,L	L^2	K * L	K^2	K^3
tree,4,1	$1^2 = 1$	1 * 4 = 4	$4^2 = 16$	$4^3 = 64$

Cálculo del número de enlaces:

*num_enlaces = capa_edge * K + capa_agregacion * K + capa_borde * K*
*num_enlaces = 1 * 4 + 4 * 4 + 16 * 4 = 84*

9.1.2 Árbol completo o fat-tree

Esta topología es la más costosa y más eficiente de la familia *tree*. De hecho, para cada distinto host, existe un camino diferente para llegar al nodo de destino. Además, si en algún nodo hubiera algún problema, permite redistribuir fácilmente el tráfico, puesto que hay una gran cantidad de caminos alternativos para redirigir el tráfico. Es una red que, al haber tanto enlaces y tantos switches, si alguna aplicación genera mucho tráfico, no supone ningún problema, en caso de que el tráfico generado por las aplicaciones esté muy distribuido, ya que si todo el tráfico va al mismo host, no se soluciona nada.

Cabe destacar que la topología *simple* y la *fat-tree* son los casos extremos de las topologías de la familia *tree*, y, por lo tanto, puede que no sean la mejor solución, ya que por una parte, la primera es muy barata pero a la vez muy ineficiente y, por otra parte, la segunda es muy eficiente pero a la vez muy costosa. De esta forma, sería muy interesante encontrar una solución que apueste por el equilibrio entre el rendimiento y coste. Para ello, se analizarán los *thin-tree* posteriormente.

CARACTERÍSTICAS DE LOS SWITCHES

Todos los switches OpenFlow que conforman esta red, como siempre, contarán con el mismo número de puertos hacia abajo (parámetro K). Como acabamos de comentar que para cada puerto hacia abajo tenemos un camino distinto hacia arriba, por definición, el número de puertos hacia arriba (parámetro L) debe ser el mismo que el parámetro 'K'. Al cumplirse la condición *down_ports(K) = up_ports(L)*, aunque subamos o bajemos de capa, nos acabaremos encontrando con el mismo número de enlaces y switches, por lo que en cada capa necesitaremos el mismo número de switches para componer la estructura de red. Esta topología es apropiada para cualquier aplicación, pero en muchos casos, no se aprovecha, ni mucho menos, todo el potencial de la red.

Por tanto, sustituyendo el parámetro 'L' por el 'K', conseguiremos la siguiente estructura genérica de los *fat-tree*:

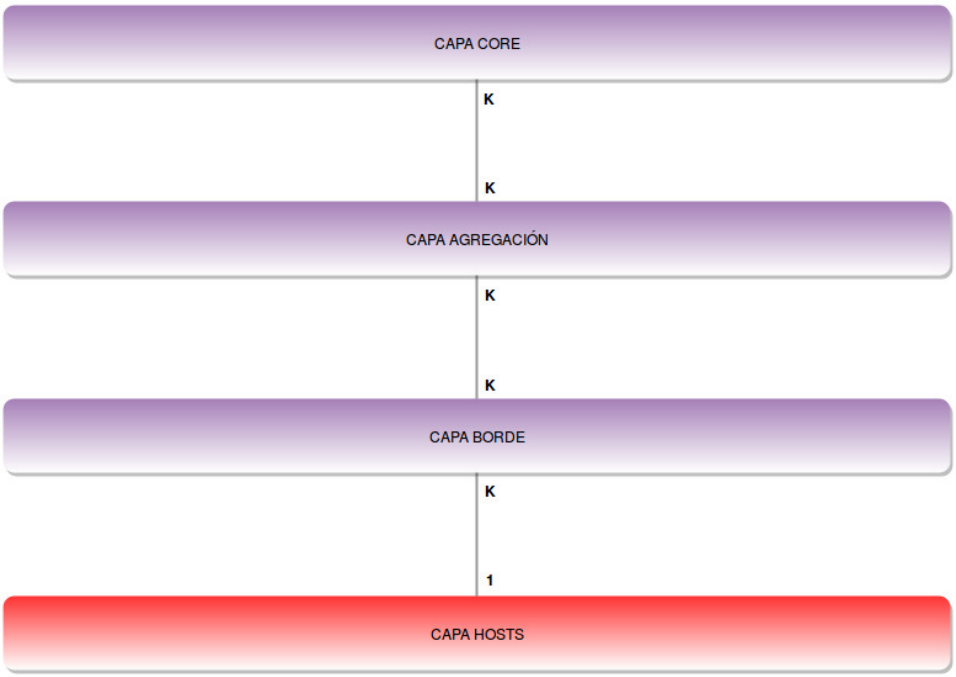

Figura 9.5: estructura de red genérica de la topología árbol *fat-tree*

En la imagen 9.6 se recoge un ejemplo de un *tree,2,2*. Pequeño, pero que sirve para entender cómo se generan. Además, se calculan los switches que necesitaremos por capa y el número de enlaces que se necesitan para generar dicha topología.

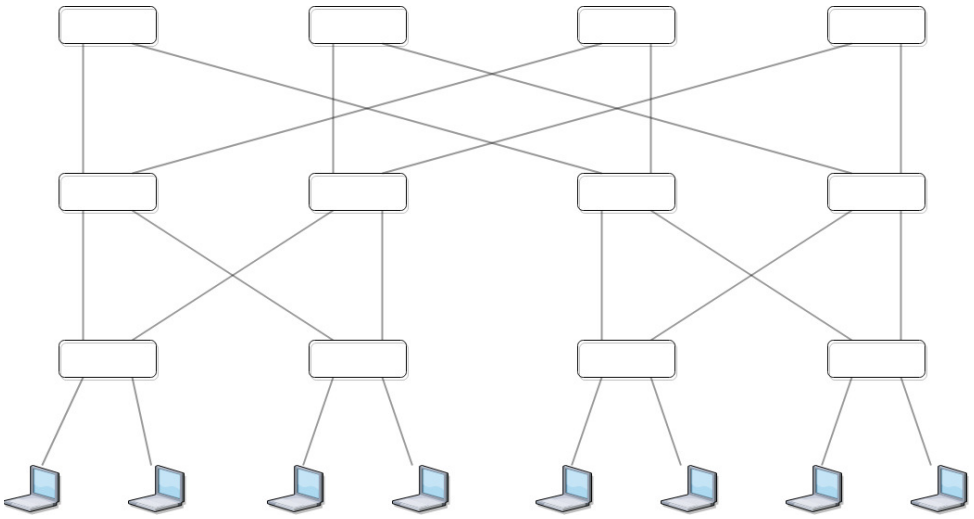

Figura 9.6: Ejemplo topología *tree,2,2*

Cálculo de los switches que necesitamos (tabla 9.4):

Tabla 9.4: Cálculo de número de switches y hosts para el caso *tree,2,2*

	capa core	capa agregación	capa borde	capa hosts
tree,K,L	L^2	K * L	K^2	K^3
tree,2,2	$2^2 = 4$	2 * 2 = 4	$2^2 = 4$	$2^3 = 8$

Cálculo del número de enlaces:

$$num\_enlaces = capa\_edge * K + capa\_agregacion * K + capa\_borde * K$$

$$num\_enlaces = 4 * 2 + 4 * 2 + 4 * 2 = 24$$

9.1.3 Thin-tree

Este tipo de árbol es el que busca una solución intermedia entre el árbol *simple* y al árbol *fat-tree*, intentando equilibrar el coste y rendimiento. Es una topología más cara y más eficiente que el *simple* y más barata y menos eficiente que el *fat-tree*. Para entender mejor en qué consiste esta topología, conviene analizar cuáles son las características de los switches y qué parámetro es el que regula el equilibrio entre el coste/rendimiento.

CARACTERÍSTICAS DE LOS SWITCHES

Todos los switches OpenFlow que conforman esta red, como siempre, contarán con el mismo número de puertos hacia abajo (parámetro K). En este caso, se define un parámetro para buscar distintos grados de coste/rendimiento, llamado factor de adelgazamiento, en inglés, *slimming factor*, también conocido como factor de sobresuscripción. Este valor representa la relación entre los parámetros 'K' y 'L' (K / L).

Mediante este valor lo que se consigue es reducir el número de puertos hacia arriba respecto al número de puertos hacia abajo. Por lo tanto, el número de puertos hacia arriba será siempre menor al número de puertos hacia abajo. Ahora bien, para buscar un equilibrio entre coste/rendimiento, tendremos que estudiar el uso que se les da a las aplicaciones para determinar los parámetros 'K' y 'L'. Eso sí, tendremos que tener en cuenta que al cumplirse la condición *down_ports (K) > up_ports (L)*, según vayamos bajando de capa, nos iremos encontrando con más enlaces y más switches, y, por lo tanto, puede resultar una topología interesante para aplicaciones que explotan localidad. Para determinar el valor más adecuado del *slimming factor*, tendremos que analizar el uso de cada una de las aplicaciones que se ejecutan, cuánto % de las comunicaciones serán locales y cuánto % de las comunicaciones serán externas.

REGULACIÓN DEL *SLIMMING FACTOR*

En este apartado se explica el efecto que podemos conseguir modificando el valor del factor de adelgazamiento, según nuestras necesidades:

- Si el valor de la L se va acercando a la K: Nos acercamos a un *fat-tree*
    - Más switches y más enlaces
        - Más rendimiento pero más coste

- Si el valor de la L se va acercando a 1: Nos acercamos un *simple tree*
    - Menos switches y menos enlaces
        - Menos rendimiento pero menos coste

Diferentes factores de adelgazamiento resultan en diferentes valores para L (entre 2 y K-1), lo que tiene un impacto en coste y rendmiento.

En la imagen 9.7 podemos observar cuál es la estructura genérica para los *thin-tree.*

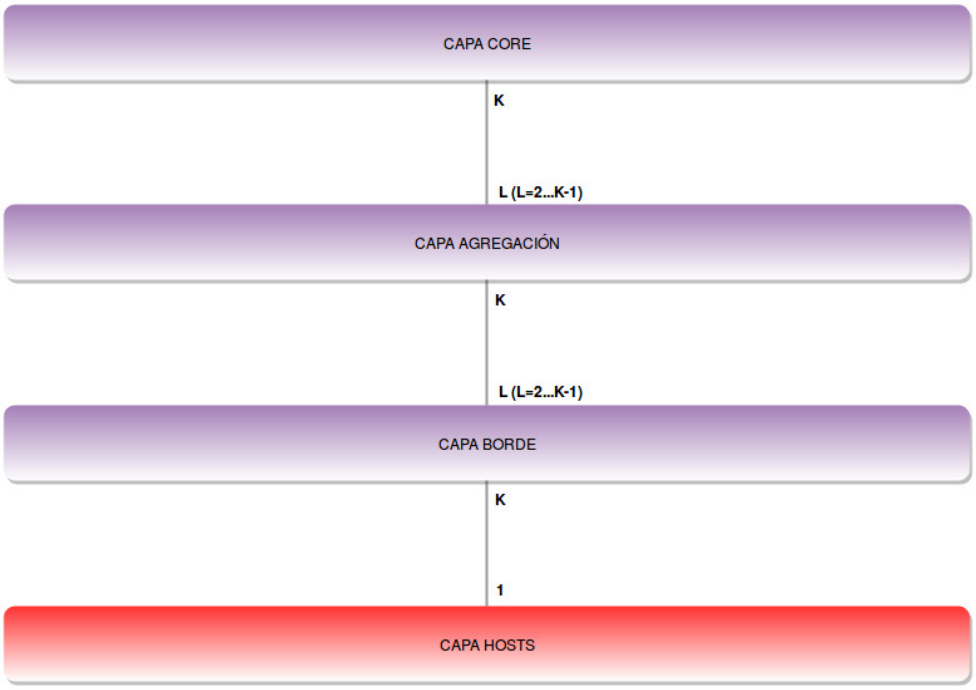

Figura 9.7: estructura de red genérica de la topología árbol *thin-tree*

En la imagen 9.8 se recoge un ejemplo de un *tree,3,2* para entender cómo se generan este tipo de árboles. Además, se calculan los switches que necesitaremos por capa y el número de enlaces que se necesitan para generar dicha topología. En este caso, el factor de sobresuscripción es 3/2 = 1,5

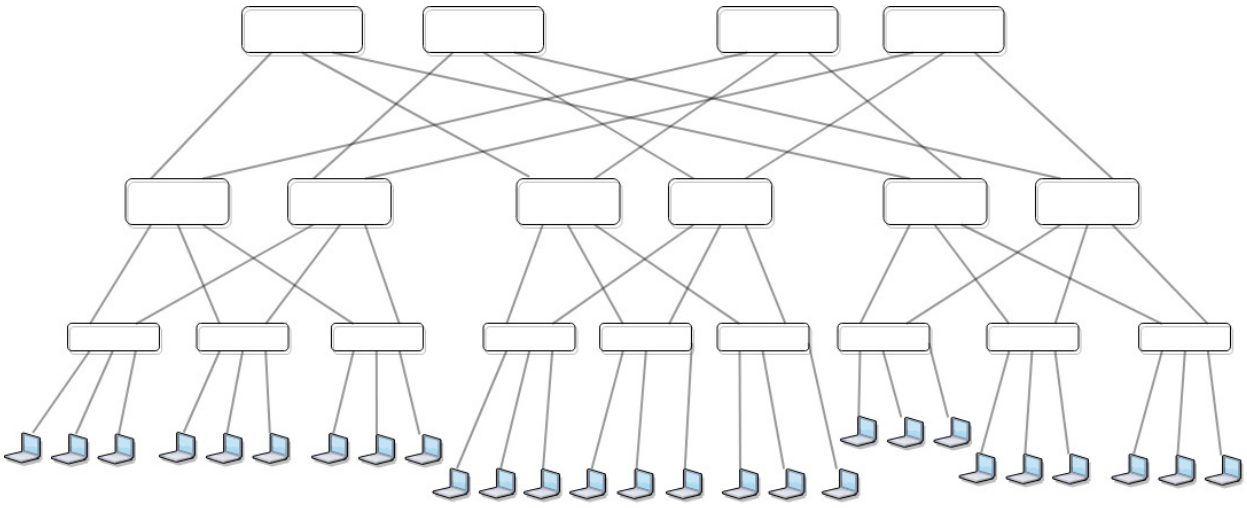

Figura 9.8: Ejemplo topología *tree,3,2*

Cálculo de los switches que necesitamos (tabla 9.5):

Tabla 9.5: Cálculo de número de switches y hosts para el caso *tree,3,2*

	capa core	capa agregación	capa borde	capa hosts
tree,K,L	L^2	K * L	K^2	K^3
tree,3,2	$2^2 = 4$	3 * 2 = 6	$3^2 = 9$	$3^3 = 27$

Cálculo del número de enlaces:

$$num\_enlaces = capa\_edge * K + capa\_agregacion * K + capa\_borde * K$$
$$num\_enlaces = 4 * 3 + 6 * 3 + 9 * 3 = 57$$

9.1.4 Generación de topologías en árbol con Mininet

Mediante la clase *TreeTopo* del fichero *dctopo.py* se generan topologías de la familia tree según los parámetros *K* y *L* citados anteriormente. Este script se encarga de crear los dispositivos de red, ya sean los switches OpenFlow necesarios para cada capa o el número de hosts que se conectan a la capa borde. Además, siguiendo un algoritmo concreto, se hacen los enlaces correspondientes para conectar los elementos de red entre sí. En este script, además de generar las topologías, se han tenido en cuenta tres características para después poder hacer el routing proactivo eficiente para cada uno de los casos.

- Asignación de puertos: Para posteriormente poder hacer un routing efectivo, se ha seguido un cierto orden a la hora de asignar los puertos a los switches. El orden de los puertos se puede ver en la imagen 9.9.

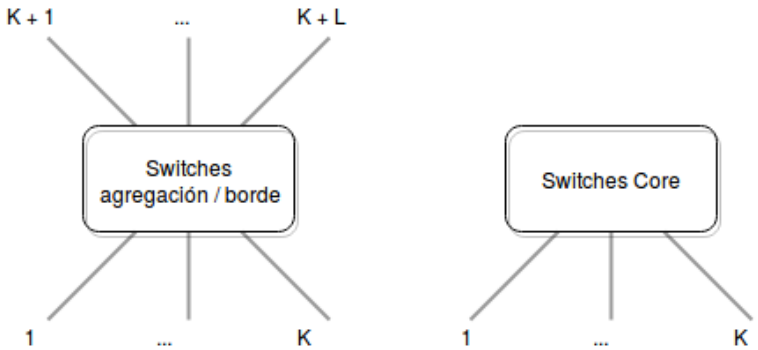

Figura 9.9: orden de asignación de puertos en los *tree*

- Gestión de direcciones: La asignación de direcciones es algo imprescindible si tenemos pensado hacer un routing proactivo basado en la agrupación de direcciones. Nos hemos basado en las direcciones IPv4 para hacer el routing proactivo, concretamente en el conjunto de direcciones IP privadas "10.0.0.0/8". A la hora de definir los flujos, el valor que se le da a la máscara será algo muy a tener en cuenta, ya que se pretende optimizar el número de entradas en la tabla de flujos, con la técnica de agrupación de direcciones. El criterio que he seguido para asignar las direcciones IP a los host ha sido la siguiente:

  ip = '10.{}.{}.{}/8'.format(p,e,h)

  donde:

  - ***p*** es la rama que corresponde el switch *borde*
  - ***e*** es el número del switch *borde* que corresponde a la rama *p*
  - ***h*** es el número de host que corresponde al switch borde *e* de la rama *p*

  La razón de que la máscara sea 8 es la siguiente: Al estar trabajando en topologías de datacenter, todos los host deben pertenecer a una misma subred para poder comunicarse. Para ello, una de las soluciones es fijar la máscara en 8.

- Gestión de *dpids*: Mininet parte de un identificador llamado *dpid* para establecer la comunicación entre el controlador y los switches OpenFlow. Exactamente, el *dpid* es un identificador de 16 dígitos que se les asigna a los switches, de los cuales Mininet extrae sus últimos 12 dígitos para generar una dirección MAC y esa dirección MAC es la que identifica a cada uno de los switches OpenFlow emulados. Por lo tanto, se ha asignado el identificador *dpid* de manera que a partir de la MAC generada por Mininet, el controlador sea capaz de identificar con qué switch se comunica, y así instalar los flujos correspondientes para ese switch. El criterio para generar los *dpid* ha sido el siguiente:

  dpid = '0000_capa_0000_core_agregacion_borde'
  direccion_MAC = [capa-00-00-core-agregacion-borde]

  donde:

  - ***capa*** es un número de dos dígitos que nos indica la capa que corresponde el switch OpenFlow (01: capa core, 02: capa agregación, 03: capa borde).
  - ***core*** es un número de dos dígitos que nos indica el número de switch de la capa core
  - ***agregacion*** es un número de dos dígitos que nos indica el número de switch de la capa agregación.
  - ***borde*** es un número compuesto por dos dígitos, donde el primer dígito nos indica la rama que corresponde el switch *borde*, y el segundo dígito nos indica el número de switch *borde* correspondiente a la rama indicada con el primer dígito

Para que tengamos más claro qué es lo que se genera mediante este script, se recoge, primeramente, en la imagen 9.10, la descripción de la topología de manera visual, después, algunos comandos interesantes de Mininet para la topología creada y una tabla con diferentes características de los nodos generados. Este ejemplo ha sido creado para la topología *tree,2,2.*

Topología generada:

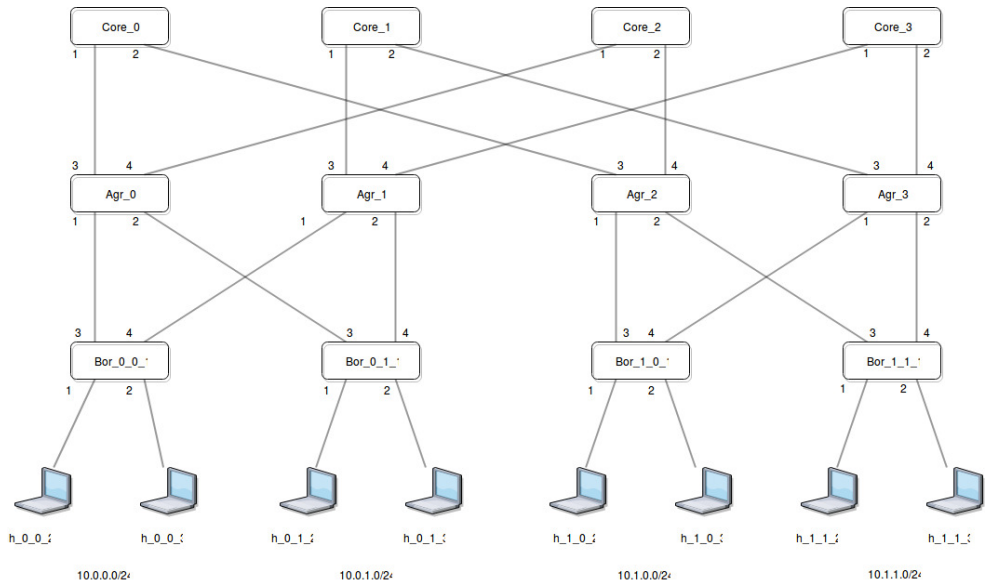

Figura 9.10: *tree,2,2* completo

Comandos Mininet:

```
root@mininet-vm:/home/mininet/ripl/ripl# sudo mn --custom mn.py --topo tree,2,2
*** Creating network
*** Adding controller
*** Adding hosts:
h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
*** Adding switches:
Agr_0 Agr_1 Agr_2 Agr_3 Bor_0_0_1 Bor_0_1_1 Bor_1_0_1 Bor_1_1_1 Core_0 Core_1 Core_2 Core_3
*** Adding links:
(Bor_0_0_1, Agr_0) (Bor_0_0_1, Agr_1) (Bor_0_1_1, Agr_0) (Bor_0_1_1, Agr_1) (Bor_1_0_1, Agr_2) (B
or_1_0_1, Agr_3) (Bor_1_1_1, Agr_2) (Bor_1_1_1, Agr_3) (Core_0, Agr_0) (Core_0, Agr_2) (Core_1, A
gr_1) (Core_1, Agr_3) (Core_2, Agr_0) (Core_2, Agr_2) (Core_3, Agr_1) (Core_3, Agr_3) (h_0_0_2, B
or_0_0_1) (h_0_0_3, Bor_0_0_1) (h_0_1_2, Bor_0_1_1) (h_0_1_3, Bor_0_1_1) (h_1_0_2, Bor_1_0_1) (h_
1_0_3, Bor_1_0_1) (h_1_1_2, Bor_1_1_1) (h_1_1_3, Bor_1_1_1)
*** Configuring hosts
h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
*** Starting controller
c0
*** Starting 12 switches
Agr_0 Agr_1 Agr_2 Agr_3 Bor_0_0_1 Bor_0_1_1 Bor_1_0_1 Bor_1_1_1 Core_0 Core_1 Core_2 Core_3 ...
*** Starting CLI:
mininet>
```

Figura 9.11: lanzamiento y creación de la topología

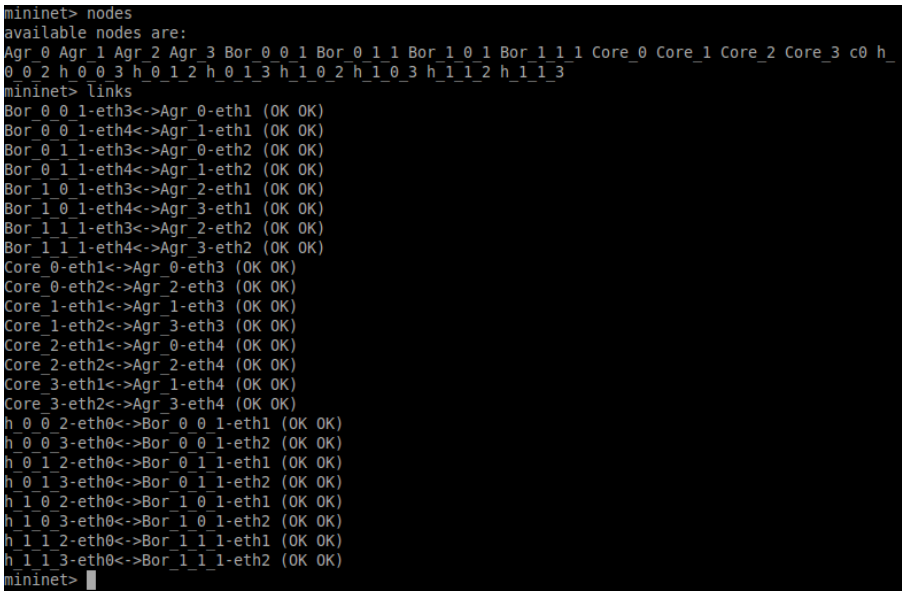

Figura 9.12: lista de nodos que hay en la topología y lista de enlaces entre los puertos de los distintos nodos

Tabla de características de los nodos (Tabla 9.6):

Tabla 9.6: Características de los nodos que hay en la topología *tree,2,2*

Identificador Nodo	identificador *dpid*	dirección MAC	dirección IP
Core_0	0000**01**0000**00**0000	[**01**-00-00-**00**-00-00]	-
Core_1	0000**01**0000**01**0000	[**01**-00-00-**01**-00-00]	-
Core_2	0000**01**0000**02**0000	[**01**-00-00-**02**-00-00]	-
Core_3	0000**01**0000**03**0000	[**01**-00-00-**03**-00-00]	-
Agr_0	0000**02**000000**00**00	[**02**-00-00-00-**00**-00]	-
Agr_1	0000**02**000000**01**00	[**02**-00-00-00-**01**-00]	-
Agr_2	0000**02**000000**02**00	[**02**-00-00-00-**02**-00]	-
Agr_3	0000**02**000000**03**00	[**02**-00-00-00-**03**-00]	-
Bor_0_0_1	0000**03**0000000**00**	[**03**-00-00-00-00-**00**]	-
Bor_0_1_1	0000**03**0000000**01**	[**03**-00-00-00-00-**01**]	-
Bor_1_0_1	0000**03**0000000**10**	[**03**-00-00-00-00-**10**]	-
Bor_1_1_1	0000**03**0000000**11**	[**03**-00-00-00-00-**11**]	-
h_0_0_2	-	-	10.0.0.2
h_0_0_3	-	-	10.0.0.3

h_0_1_2	-	-	10.0.1.2
h_0_1_3	-	-	10.0.1.3
h_1_0_2	-	-	10.1.0.2
h_1_0_3	-	-	10.1.0.3
h_1_1_2	-	-	10.1.1.2
h_1_1_3	-	-	10.1.1.3

Como se ha explicado antes, la clase *TreeTopo* del fichero dctopo.py se encarga de generar las topologías de la familia *tree.* Para terminar con esta sección, se muestra, en pseudocódigo, el algoritmo empleado para implementar la clase *TreeTopo* (consultar ANEXO E para código completo):

```
        caracteristicas de la topologia
        switches core = L * L
        switches agregacion = K * L
        switches borde = K * K
        hosts = K * K * K

      1. definir un array para cada capa de switches. En los arrays, se irán
guardando los switches para posteriormente poder hacer los correspondientes
enlaces


      2. crear switches borde y guardarlos en su correspondiente array y
asignar los correspondientes dpid
            2.1 para cada switch borde, crear los correspondientes host
asignando la correspondiente dirección IP según el criterio tomado y crear
enlace entre switch borde-host

      3. crear switches agregacion y guardarlos en su correspondiente array y
asignar los correspondientes dpid

      4. crear enlaces entre los correspondientes switches borde-agregacion,
accediendo a las correspondientes posiciones de los arrays donde se han
guardado los switches borde y agregacion


      5. crear switches core y guardarlos en su correspondiente array y
asignar los correspondientes dpid


      6. crear enlaces entre los correspondientes switches core-agregacion,
accediendo a las correspondientes posiciones de los arrays donde se han
guardado los switches core y agregacion
```

9.1.5 Encaminamiento para las topologías de la familia *tree*

En este apartado se explicará cómo se ha programado el controlador para que éste instruya de manera proactiva a los switches OpenFlow, permitiendo la comunicación entre los hosts que están en la red, sin que haya comunicación previa entre los mismos. En todo momento, tendremos que tener alguna forma de saber con qué switch estamos comunicándonos para instalar las entradas correspondientes en su tabla de flujos.

Al controlador, al igual que al script que genera las topologías, tendremos que pasarle los parámetros K y L, para que haya coordinación total entre la topología generada y el routing a realizar. En el momento que el controlador se haya conectado con los switches OpenFlow a través de la red de control, sin que haya ninguna comunicación previa, el controlador rellenará las tablas de flujo de los switches para hacer posible la comunicación entre los host que conforman la red.

El routing que se ha implementado para las topologías de la familia tree se divide en dos partes principales. Por una parte, el tráfico hacia abajo (entradas de la tabla de flujo cuya acción sea redistribuir la información por alguno de los puertos hacia abajo del switch OpenFlow) y, por otra parte, el tráfico hacia arriba (entradas de la tabla de flujo cuya acción sea redistribuir la información por alguno de los puertos hacia arriba del switch OpenFlow). El primer caso es muy fácil de implementar, ya que el criterio de escoger por qué puerto de abajo redirigir el tráfico se basa simplemente en analizar cuál es la dirección IP destino. Para ello, a veces habrá que agrupar distintas subredes, ajustando adecuadamente la máscara, pero el camino para llegar a la siguiente subred (switch) o host será único y no habrá alternativas.

En cambio, la distribución del tráfico hacia arriba para llegar a otras subredes no es única, ya que existen diferentes caminos. Entonces, para decidir por qué puerto de arriba redirigir el tráfico he tenido que escoger un criterio. En mi caso, el criterio escogido ha sido agrupar los puertos hacia abajo de los switches en ‘L’ grupos a través del operador módulo (% L), y en base a qué grupo de L corresponde cada puerto hacia abajo, reenviar la información por el camino correspondiente por alguno de los puertos hacia arriba. De esta manera, se consigue que todo el tráfico entrante por los puertos 1...K de los switches se distribuya por los puertos K+1...K+L de manera equilibrada. Cabe destacar que la distribución del tráfico hacia arriba será totalmente equilibrada únicamente en el caso que el parámetro ‘L’ sea múltiplo del parámetro ‘K’.

El orden de distribuir el tráfico será el siguiente: El número del puerto hacia abajo cuyo módulo L sea 0 se distribuirá por el primer puerto hacia arriba (K+1). El número de puerto hacia abajo cuyo módulo sea 1 se distribuirá por el segundo puerto hacia arriba (K+1+1), y así sucesivamente. El número del puerto cuyo resultado del módulo sea L-1, se distribuirá por el último de los puertos hacia arriba (K+L).

Considerando que el tráfico generado por cada uno de los hosts fuese homogéneo, es decir, cada uno de los host generan tráfico similiar, la política tomada para la distribución del tráfico hacia arriba es adecuada. En otros casos, puede no ser óptima.

Según la capa a la que corresponde un switch, los flujos orientados para redirigir el tráfico por los puertos hacia abajo se dividen en tres partes:

- Swiches capa core: Se instalan de manera idéntica en cada uno de los switches de la capa core. Dependiendo del parámetro 'K', habrá tantas subredes como 'K' y tendremos que hacer uso de *supernetting* y ajustar el correspondiente valor de la máscara, para poder agrupar las distintas subredes:
    - Para cada puerto hacia abajo:
        - 10.K.0.0/mascara

- Switches capa agregación: Hay que identificar a qué grupo "K" de tamaño "L" corresponde el switch.
    - Para cada puerto hacia abajo:
        - 10.grupo.K.0/24

- Switches capa borde: Identificar a qué rama y borde corresponde cada switch de acceso y crear la entrada para llegar a cada uno de los hosts
    - Para cada puerto hacia abajo del switch:
        - 10.rama.borde.host

Los flujos orientados a redirigir tráfico por los puertos hacia arriba de los switches OpenFlow se dividen en dos partes, según la capa a la que corresponde el switch.

- Switches capa agregación: Se instalan de manera idéntica en cada uno de los switches de la capa agregación
    - Para cada puerto hacia arriba del switch:
        - Calcular el grupo de L que corresponde el puerto hacia abajo y reenviar el tráfico por el puerto hacia arriba que corresponde ese grupo L

- Switches capa borde: Se instalan de manera idéntica en cada uno de los switches de la capa borde
    - Para cada puerto hacia arriba del switch:
        - Calcular el grupo de L que corresponde el puerto hacia abajo y reenviar el tráfico por el puerto hacia arriba que corresponde ese grupo L

PRIORIDADES

En mi caso, las entradas que hacen posible el routing las divido en dos partes. Por una parte, aquellas entradas que se encargan de redirigir el tráfico hacia abajo, basado en direcciones IPv4 destino (nw_dst), y por otra parte, aquellas entradas que se encargan de redirigir el tráfico hacia arriba, basado en puertos entrantes (in_port). Puede que se dé el caso que un paquete coincida con las dos entradas, pero nosotros únicamente queremos hacer que corresponda a una. Para ello, se define la prioridad del flujo (campo priority). A mayor número que asignemos en ese campo, mayor prioridad le daremos a una entrada. Por tanto, a las entradas relacionadas para redirigir el tráfico hacia abajo les daremos mayor prioridad que a las entradas relacionadas para redirigir el tráfico hacia arriba. De este modo, conseguiremos darles mayor prioridad a los destinos que están en el mismo switch y evitaremos que en esos casos los paquetes se envíen hacia arriba, porque realmente lo que queremos es enviarlos hacia abajo.

Ejemplo: Imaginemos que tenemos dos host: h1 (10.0.0.2) y h2 (10.0.0.3) y un switch S1 al que se le conectan los dos host en los puertos 1 y 2 respectivamente y tiene un camino hacia arriba por el puerto 3. Suponemos que hago un ping de h1 a h2. Cuando el switch S1 reciba el paquete, se dará cuenta de que ese paquete corresponde a dos entradas distintas de la tabla de flujos. Por una parte, paquete cuya IP_destino sea 10.0.0.3 reenviarla por el puerto 2 (tráfico hacia abajo) y por otra parte, el tráfico entrante por el puerto 1 o 2, redirigirlo por el puerto 3 (tráfico hacia arriba). La acción que nos interesa realizar es la primera, ya que queremos que el paquete se distribuya de manera local sin tener que enviarlo hacia arriba. Para solucionar este problema, daremos mayor prioridad a las entradas que se encarguen de distribuir el tráfico hacia abajo.

Para entender mejor cómo se comunica el controlador con los diferentes switches y cómo instala las entradas correspondientes para cada switch específico, se muestra el módulo que carga el controlador POX, desarrollado en Python3, llamado *tree_controller_julen.py* (Consultar anexo E para el código).

9.1.6 Ejemplo completo tree

Para finalizar con las topologías en árbol que se han analizado, se demostrará la conectividad entre los nodos en a través del comando *pingall* de Mininet, comunicación todos con todos. El ejemplo con el que se trabaja es el *tree,2,2* previamente analizado.

Se indicarán qué parámetros debemos pasarle tanto al script que se encarga de generar la topología como al controlador. Como bien sabemos, el controlador manda mensajes para rellenar las tablas de flujos de los switches sin que haya comunicación previa. Por ello, también se mostrarán algunas tablas de flujo para entender cómo se distribuye el tráfico y permitir la comunicación entre los host.

Controlador:

Lanzar controlador

```
root@mininet-vm:/home/mininet/pox# ./pox.py forwarding.tree_controller_julen
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.5.0 (eel) is up.
```

Figura 9.13: lanzar el controlador con el módulo *tree_controller_julen*

Una vez se genera la topología, comprobar que el controlador se ha conectado con los switches OpenFlow creados.

```
root@mininet-vm:/home/mininet/pox# ./pox.py forwarding.tree_controller_julen
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[01-00-00-00-00-00 7] connected
INFO:openflow.of_01:[03-00-00-00-00-11 8] connected
INFO:openflow.of_01:[03-00-00-00-00-00 5] connected
INFO:openflow.of_01:[03-00-00-00-00-01 10] connected
INFO:openflow.of_01:[01-00-00-02-00-00 9] connected
INFO:openflow.of_01:[03-00-00-00-00-10 2] connected
INFO:openflow.of_01:[02-00-00-00-02-00 4] connected
INFO:openflow.of_01:[02-00-00-00-01-00 6] connected
INFO:openflow.of_01:[02-00-00-00-03-00 11] connected
INFO:openflow.of_01:[01-00-00-01-00-00 13] connected
INFO:openflow.of_01:[02-00-00-00-00-00 12] connected
INFO:openflow.of_01:[01-00-00-03-00-00 3] connected
```

Figura 9.14: Comprobar que los switches OpenFlow se han conectado al controlador

Parámetros necesarios para lanzar el controlador:

- K: Para crear flujos que distribuyen el tráfico hacia abajo
- L: Para crear flujos que distribuyen el tráfico hacia arriba

Topología:

Generar topología

```
root@mininet-vm:/home/mininet/ripl/ripl# sudo mn --custom mn.py --topo tree,2,2 --controller remo
te --arp
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
*** Adding switches:
Agr_0 Agr_1 Agr_2 Agr_3 Bor_0_0_1 Bor_0_1_1 Bor_1_0_1 Bor_1_1_1 Core_0 Core_1 Core_2 Core_3
*** Adding links:
(Bor_0_0_1, Agr_0) (Bor_0_0_1, Agr_1) (Bor_0_1_1, Agr_0) (Bor_0_1_1, Agr_1) (Bor_1_0_1, Agr_2) (B
or_1_0_1, Agr_3) (Bor_1_1_1, Agr_2) (Bor_1_1_1, Agr_3) (Core_0, Agr_0) (Core_0, Agr_2) (Core_1, A
gr_1) (Core_1, Agr_3) (Core_2, Agr_0) (Core_2, Agr_2) (Core_3, Agr_1) (Core_3, Agr_3) (h_0_0_2, B
or_0_0_1) (h_0_0_3, Bor_0_0_1) (h_0_1_2, Bor_0_1_1) (h_0_1_3, Bor_0_1_1) (h_1_0_2, Bor_1_0_1) (h_
1_0_3, Bor_1_0_1) (h_1_1_2, Bor_1_1_1) (h_1_1_3, Bor_1_1_1)
*** Configuring hosts
h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
*** Starting controller
c0
*** Starting 12 switches
Agr_0 Agr_1 Agr_2 Agr_3 Bor_0_0_1 Bor_0_1_1 Bor_1_0_1 Bor_1_1_1 Core_0 Core_1 Core_2 Core_3 ...
*** Starting CLI:
mininet>
```

Figura 9.15: lanzar la topología *tree,2,2* con el comando *sudo mn*

Parámetros necesarios para generar la topología:

- K: Número de *down-ports* de los switches
- L: Número de *up-ports de los switches*

Tablas de flujo:

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows Agr_1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=938.282s, table=0, n_packets=12, n_bytes=1176, idle_age=72
0, priority=2,ip,nw_dst=10.0.0.0/24 actions=output:1
 cookie=0x0, duration=938.261s, table=0, n_packets=12, n_bytes=1176, idle_age=72
0, priority=2,ip,nw_dst=10.0.1.0/24 actions=output:2
 cookie=0x0, duration=938.261s, table=0, n_packets=13, n_bytes=1174, idle_age=72
0, priority=1,in_port=1 actions=output:4
 cookie=0x0, duration=938.261s, table=0, n_packets=14, n_bytes=1252, idle_age=72
0, priority=1,in_port=2 actions=output:3
```

Figura 9.16: tabla de flujo del switch Agr1

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows Core_1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=908.259s, table=0, n_packets=8, n_bytes=784, idle_age=690,
 priority=2,ip,nw_dst=10.0.0.0/23 actions=output:1
 cookie=0x0, duration=908.206s, table=0, n_packets=8, n_bytes=784, idle_age=690,
 priority=2,ip,nw_dst=10.1.0.0/23 actions=output:2
```

Figura 9.17: tabla de flujo del switch Core_1

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows Bor_0_1_1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=961.329s, table=0, n_packets=14, n_bytes=1372, idle_age=74
3, priority=2,ip,nw_dst=10.0.1.2 actions=output:1
 cookie=0x0, duration=961.329s, table=0, n_packets=14, n_bytes=1372, idle_age=74
3, priority=2,ip,nw_dst=10.0.1.3 actions=output:2
 cookie=0x0, duration=961.35s, table=0, n_packets=18, n_bytes=1644, idle_age=743
, priority=1,in_port=1 actions=output:4
 cookie=0x0, duration=961.329s, table=0, n_packets=17, n_bytes=1566, idle_age=74
3, priority=1,in_port=2 actions=output:3
```

Figura 9.18: tabla de flujo del switch Bor_0_1_1

En la tabla 9.7 se muestra la información de cada flujo. La columna matching es la coincidencia para los paquetes, la columna action es el puerto de salida por el cual se reenvía el paquete al que se la ha encontrado coincidencia y la columna priority es la prioridad que se le ha dado a un flujo.

Tabla 9.7: información de las tablas de flujo que se han mostrado

Switch	Flujo	Matching	Action	Priority	Nota
Core_1	1	10.0.0.0/23	Puerto salida 1	2	Tráfico hacia abajo
	2	10.1.0.0/23	Puerto salida 2	2	Tráfico hacia abajo
Agr_1	1	10.0.0.0/24	Puerto salida 1	2	Tráfico hacia abajo
	2	10.0.1.0/24	Puerto salida 2	2	Tráfico hacia abajo
	3	Puerto entrada 1	Puerto salida 4	1	Tráfico hacia arriba
	4	Puerto entrada 2	Puerto salida 3	1	Tráfico hacia arriba
Bor_0_1_1	1	10.0.1.2	Puerto salida 1	2	Tráfico local
	2	10.0.1.3	Puerto salida 2	2	Tráfico local
	3	Puerto entrada 1	Puerto salida 4	1	Tráfico hacia arriba
	4	Puerto entrada 2	Puerto salida 3	1	Tráfico hacia arriba

Conectividad entre hosts:

```
mininet> pingall
*** Ping: testing ping reachability
h_0_0_2 -> h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
h_0_0_3 -> h_0_0_2 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
h_0_1_2 -> h_0_0_2 h_0_0_3 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
h_0_1_3 -> h_0_0_2 h_0_0_3 h_0_1_2 h_1_0_2 h_1_0_3 h_1_1_2 h_1_1_3
h_1_0_2 -> h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_3 h_1_1_2 h_1_1_3
h_1_0_3 -> h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_1_2 h_1_1_3
h_1_1_2 -> h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_3
h_1_1_3 -> h_0_0_2 h_0_0_3 h_0_1_2 h_0_1_3 h_1_0_2 h_1_0_3 h_1_1_2
*** Results: 0% dropped (56/56 received)
```

Figura 9.19: demostrar la conectividad entre todos los host que hay en la topología *tree,2,2*

9.2 Topologías *Jellyfish*

La topología *Jellyfish* [20] es una estructura de red basada en la aleatoriedad, totalmente diferente a las estructuras regulares como el *hipercubo 2D/3D* (utilizados tanto en supercomputadores como en datacenters), *toros 2D/3D* (principalmente utilizados en supercomputadores) o las topologías de la familia *tree* que se acaban de analizar. Es una topología muy flexible, expandible y muy tolerante a fallos, ya que permite reaccionar inmediatamente a posibles fallos. No obstante, esta topología se puede entender como un grafo aleatorio, y por lo tanto, supone un nuevo reto a la hora de plantear un routing eficaz, ya que no hay manera de agrupar las direcciones como en los *tree* u otras topologías regulares.

Para generar la topología *Jellyfish* se han definido tres parámetros: Número de switches (*numSwitches*), puertos hacia abajo (*K*) y puertos switch-to-switch aleatorios (*L*). A diferencia de las topologías *tree*, al no ser una topología regular sino aleatoria, los switches no se organizan en capas. Si queremos hacer analogía a las topologías *tree,* podemos entender que todos los switches de la *Jellyfish* se sitúan en una misma capa ficticia.

CARACTERÍSTICAS DE LOS SWITCHES

Los puertos de los switches se han dividido en dos partes, según con el tipo de dispositivo que se conectan. Por una parte, todos y cada uno de los switches OpenFlow que conforman la red, contarán con el mismo número de puertos hacia abajo (parámetro *K*). A estos puertos se les conectarán únicamente los hosts a los cuales se les proporcionará servicio. Por otra parte, el parámetro "*L"*, determina el número de puertos switch-to-switch aleatorios, o en otra palabras, el número de enlaces máximos para comunicarse únicamente con otros switches, de forma aleatoria. Siguiendo un algoritmo concreto, conseguiremos generar un único grafo donde se conectan todos los switches entre sí y se conectan los correspondientes host a cada uno de los switches, siempre cumpliéndose la condición "*down_ports (K) >= switch-to-switch random ports (L)"* y "*grado_switch <= K+L".*

El grado de un switch se define de la misma manera que en los *tree:*

grado_switch: K + L

donde:

- K = Puertos hacia abajo del switch (switch-to-host)

- L = Puertos switch-to-switch aleatorios

En la tabla 9.8, se muestra el cálculo de los enlaces que necesitamos para generar una topología *Jellyfish* y el número de hosts a los que podemos dar servicio, dependiendo de los tres parámetros mencionados:

Tabla 9.8: Cálculo de número de enlaces y hosts para la topología *Jellyfish*

	enlaces	hosts
jf,numSwitches,K,L	(numSwitches * K) + ((numSwitches * L) / 2)	numSwitches * K
jf,20,5,2	(20 * 5) + ((20 * 2) / 2) = 120	20 * 5 = 100

9.2.1 Generación de topologías *Jellyfish* con Mininet

Mediante la clase *JellyFishTopo* del fichero *dctopo.py* se genera una topología *Jellyfish*, según los parámetros *numSwitches, K* y *L* citados anteriormente. Este script se encarga de crear los dispositivos de red, ya sean los switches OpenFlow necesarios o los hosts que se conectan a los mismos. Además, utiliza un algoritmo específico para hacer las conexiones switch-to-switch de manera aleatoria dependiendo de los parámetros *L* y *numswitches*. En este script, además de generar las topologías, se han tenido en cuenta tres características principales para después poder hacer el routing proactivo.

- Asignación de puertos: El orden de los puertos hacia abajo, puertos a los cuales únicamente se conectan los host, siguen un orden concreto (*primer host=puerto 1, último host=puerto K*). No obstante, los puertos que conectan switches entre sí, obviamente, no se han podido asignar de manera lógica, ya que como bien se ha dicho antes, un algoritmo se encarga de hacer las conexiones switch-to-switch de manera aleatoria y, por lo tanto, la asignación de esos puertos también lo es. Podemos entender que el número de esos puertos está comprendido entre los valores *K+1* y *K+L*. El orden de asignación de los puertos se puede ver en la imagen 9.13.

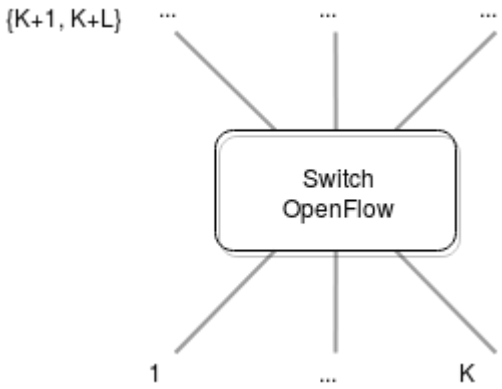

Figura 9.20: orden de asignación de puertos en la topología *Jellyfish*

- Gestión de direcciones: A diferencia de las topologías *tree*, en este caso, no podemos pensar en hacer un direccionamiento basado en la agrupación de direcciones, ya que cada vez que generamos una nueva *Jellyfish*, se genera un nuevo grafo aleatorio que conecta los switches entre sí. En este caso, partiendo del subconjunto de direcciones IPv4 privadas "10.0.0.0/16", el criterio que he seguido para asignar las direcciones IP a los host ha sido la siguiente:

  ip = '10.0.{}.{}/16'.format(x,h)

  donde:

  - ***x*** es el número que identifica a un switch
  - ***h*** es el número de host que corresponde al switch ***x***

La razón de que la máscara sea 16 es la siguiente: Al estar trabajando en topologías de datacenter, todos los host deben pertenecer a una misma subred para poder comunicarse. Para ello, una de las soluciones es fijar la máscara en 16.

De esta manera, analizando el tercer byte de la dirección IP, sabremos si el tráfico tenemos que distribuirlo en la misma subred o hay que reenviar el paquete a otro switch. En caso de que haya que redirigir el tráfico de manera local, analizando el cuarto byte, sabremos por qué puerto específico reenviar el paquete. Por otra parte, si la comunicación no es local, podremos saber a qué switch debe ser destinado el paquete. Para hacer llegar ese paquete al switch correspondiente, optamos por un encaminamiento por caminos mínimos. Esto se analizará en profundidad en el apartado del routing para la topología *Jellyfish*.

- Gestión de *dpids*: Al igual que en los tree, la gestión de dpids es imprescindible para que en todo momento el controlador sepa con cuál de los switches OpenFlow se está comunicando. Exactamente, el *dpid* es un identificador de 16 dígitos que se les asigna a los switches, de los cuales Mininet extrae sus últimos 12 dígitos para generar una dirección MAC y esa dirección MAC es la que identifica a cada uno de los switches OpenFlow emulados. Por lo tanto, se ha asignado el identificador *dpid* de manera que a

partir de la MAC generada por Mininet, el controlador sea capaz de identificar con qué switch se comunica, y así instalar los flujos correspondientes para ese switch. El criterio para generar los *dpid* ha sido el siguiente:

dpid = '0000comp00000000switch'
direccion_MAC = [comp-00-00-00-00-switch]

donde:

- ***comp*** es un número que evita que los 16 dígitos sean 0. Mininet no permite asignar el dpid = '0000000000000000' en ningún caso. Por lo tanto, evitamos que salte un error al asignar el dpid al switch cuyo identificador está totalmente compuesto por 0 (siempre comp=01).
- ***switch*** es un número de dos dígitos que identifica el switch (número entre 00 y 99)

Para que tengamos más claro qué es lo que se genera mediante el script, se recoge, primeramente, una imagen que describe la topología de manera visual. Después, algunos comandos interesantes de Mininet para la topología creada y una tabla con diferentes características de los nodos generados. Este ejemplo ha sido creado para la topología *jf,6,2,2.*

TOPOLOGÍA GENERADA:

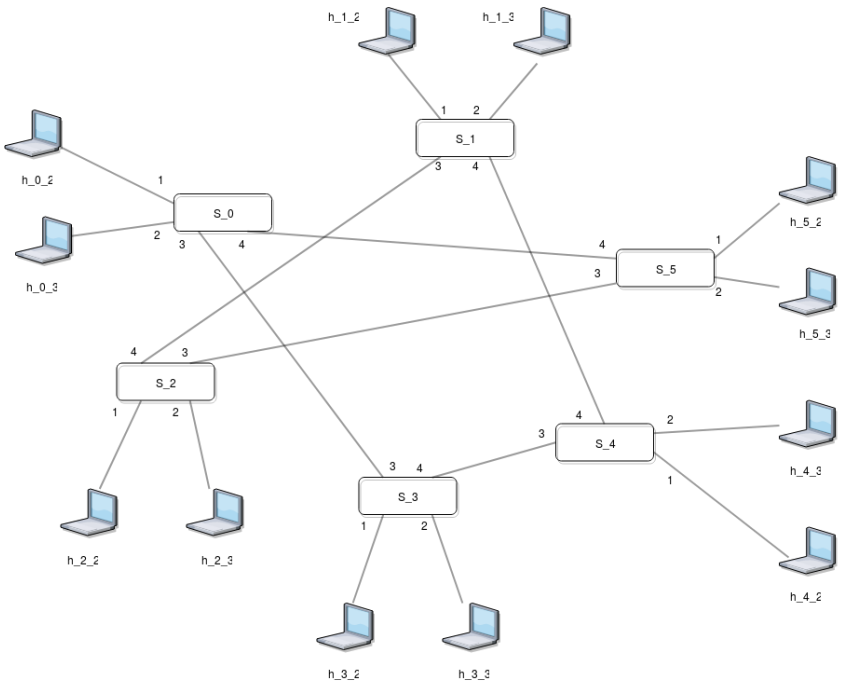

Figura 9.21: ejemplo jellyfish,6,2,2

Comandos Mininet:

```
root@mininet-vm:/home/mininet/ripl/ripl# sudo mn --custom mn.py --topo jf,6,2,2 --controller remo
te --arp
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h_0_2 h_0_3 h_1_2 h_1_3 h_2_2 h_2_3 h_3_2 h_3_3 h_4_2 h_4_3 h_5_2 h_5_3
*** Adding switches:
S_0 S_1 S_2 S_3 S_4 S_5
*** Adding links:
(S_0, S_3) (S_0, S_5) (S_2, S_1) (S_3, S_4) (S_4, S_1) (S_5, S_2) (h_0_2, S_0) (h_0_3, S_0) (h_1_
2, S_1) (h_1_3, S_1) (h_2_2, S_2) (h_2_3, S_2) (h_3_2, S_3) (h_3_3, S_3) (h_4_2, S_4) (h_4_3, S_4
) (h_5_2, S_5) (h_5_3, S_5)
*** Configuring hosts
h_0_2 h_0_3 h_1_2 h_1_3 h_2_2 h_2_3 h_3_2 h_3_3 h_4_2 h_4_3 h_5_2 h_5_3
*** Starting controller
c0
*** Starting 6 switches
S_0 S_1 S_2 S_3 S_4 S_5 ...
*** Starting CLI:
mininet>
```

Figura 9.21: lanzamiento y creación de la topología

```
mininet> nodes
available nodes are:
S_0 S_1 S_2 S_3 S_4 S_5 c0 h_0_2 h_0_3 h_1_2 h_1_3 h_2_2 h_2_3 h_3_2 h_3_3 h_4_2 h_4_3 h_5_2 h_5_
3
mininet> links
S_0-eth3<->S_3-eth3 (OK OK)
S_0-eth4<->S_5-eth4 (OK OK)
S_2-eth4<->S_1-eth3 (OK OK)
S_3-eth4<->S_4-eth3 (OK OK)
S_4-eth4<->S_1-eth4 (OK OK)
S_5-eth3<->S_2-eth3 (OK OK)
h_0_2-eth0<->S_0-eth1 (OK OK)
h_0_3-eth0<->S_0-eth2 (OK OK)
h_1_2-eth0<->S_1-eth1 (OK OK)
h_1_3-eth0<->S_1-eth2 (OK OK)
h_2_2-eth0<->S_2-eth1 (OK OK)
h_2_3-eth0<->S_2-eth2 (OK OK)
h_3_2-eth0<->S_3-eth1 (OK OK)
h_3_3-eth0<->S_3-eth2 (OK OK)
h_4_2-eth0<->S_4-eth1 (OK OK)
h_4_3-eth0<->S_4-eth2 (OK OK)
h_5_2-eth0<->S_5-eth1 (OK OK)
h_5_3-eth0<->S_5-eth2 (OK OK)
mininet>
```

Figura 9.22: lista de nodos que hay en la topología y lista de enlaces entre los puertos de los distintos nodos

Tabla de características de los nodos (tabla 9.9):

Tabla 9.9: Características de los nodos que hay en la topología *jf,6,2,2*

Identificador Nodo	identificador *dpid*	dirección MAC	dirección IP
S_0	0000**01**0000000**00**	[**01**-00-00-00-00-**00**]	-
S_1	0000**01**0000000**01**	[**01**-00-00-00-00-**01**]	-
S_2	0000**01**0000000**02**	[**01**-00-00-00-00-**02**]	-
S_3	0000**01**0000000**03**	[**01**-00-00-00-00-**03**]	-
S_4	0000**01**0000000**04**	[**01**-00-00-00-00-**04**]	-
S_5	0000**01**0000000**05**	[**01**-00-00-00-00-**05**]	-
h_0_2	-	-	10.0.0.2
h_0_3	-	-	10.0.0.3
h_1_2	-	-	10.0.1.2
h_1_3	-	-	10.0.1.3
h_2_2	-	-	10.0.2.2
h_2_3	-	-	10.0.2.3
h_3_2	-	-	10.0.3.2
h_3_3	-	-	10.0.3.3
h_4_2	-	-	10.0.4.2
h_4_3	-	-	10.0.4.3
h_5_2	-	-	10.0.5.2
h_5_3	-	-	10.0.5.3

Como se ha explicado antes, la clase *JellyFishTopo* del fichero *dctopo*.py se encarga de generar las topologías de la familia *Jellyfish.* Para terminar con esta sección, se muestra, en pseudocódigo, el algoritmo empleado para implementar la clase *JellyFishTopo* (consultar ANEXO E para código completo).

Política de construcción de la topología: Ir uniendo distintos switches al azar teniendo en cuenta que el número máximo de veces que puede estar implicado un switch en los enlaces es L. Puede que queden nodos sueltos, para conectarlos, se eliminan unos enlaces al azar, y desde cada nodo donde se ha eliminado el enlace escogido al azar, creamos dos enlaces al nodo suelto.

```
caracteristicas de la topologia
numSwitches = numSwitches # numero de switches
K = K # numero de puertos hacia abajo (switch-to-host)
L = L # numero de puertos aleatorios (switch-to-switch)


1. definir la semilla. Esto nos permite poder replicar experimentos y así
poder implementar el routing para casos aleatorios controlados.


2. definir el array switches[] donde se van a ir guardando los switches. Segun
se vaya completando el grado de un switch, ese switch se saca de la lista,
para que el mismo no pueda ser involucrado en un nuevo posible enlace

3. crear switches y guardarlos en el array switches[] y asignar los
correspondientes dpid
      3.1 para cada switch creado, crear los correspondientes host asignando
la correspondiente dirección IP según el criterio tomado y crear enlace entre
switch-host

4. definir el array enlaces_guardados[]. En cada posicion del array se guarda
la pareja de switches switch-switch2, representando un enlace bidireccional
entre los mismos


5. mientras que en el array switches[] quede mas de un elemento y los switches
no esten totalmente conectados, loop:


      5.1 escoger al azar un switch switch de la lista de switches[]


      5.2 escoger al azar un switch switch2 de la lista de switches[]


      5.3 comprobar si los dos switches escogidos al azar son los mismos. Si
son los mismos, no hacer nada


      5.4 si no ha salido el mismo switch, guardar el nuevo posible enlace en
la lista enlaces_guardados[]


      5.5 comprobar si la pareja bidireccional switch-switch2 ya existe en la
lista enlaces_guardados[]. Si ya existe, no hacer nada
```

```
      5.6 si la pareja bidireccional no existe, guardar ese enlace en la lista
enlaces_guardados[]

      5.7 si el switch switch o switch2 aparece L (max. puertos switch-to-
switch) veces en la lista enlaces_guardados[] o esta totalmente conectado
          5.7.1 eliminar switch switch o switch2 de la lista switches[], ya
que el switch switch o switch2 ha quedado completo y no queremos que se vea
implicado en un nuevo posible enlace

      5.8 proceso para unir algun nodo que haya podido quedar suelto. La idea
es, encontrar un switch cuyo grado no esté completo. Una vez encontrado ese
switch, quitando un enlace random donde no este implicado ese switch, enlazar
el nodo encontrado, a los dos nodos donde se ha eliminado ese enlace random.

6. una vez tengamos la lista de enlaces guardados switch-switch2, sin que haya
quedado ningun nodo suelto, crear los enlaces bidireccionales. Cada posicion
de la lista representa un enlace bidireccional distinto (swtich-switch2)
```

Nota: La lista de enlaces_guardados[] se la pasaremos al controlador. Esa lista representa las conexiones aleatorias generadas switch-to-switch y asi el controlador podra hacer un routing para ese caso concreto

9.2.2 Encaminamiento para las topologías *Jellyfish*

En este apartado se explicará cómo se ha programado el controlador para que éste instruya de manera proactiva a los switches OpenFlow, permitiendo la comunicación entre los hosts que están en la red, sin que haya comunicación previa entre los mismos. En todo momento, tendremos que tener alguna forma de saber con qué switch estamos comunicándonos para instalar las entradas correspondientes en su tabla de flujos.

Al controlador, al igual que al script que genera las topologías, le tendremos que pasar cierta información. Tendremos que pasarle el parámetro '*K'* para que el controlador pueda instruir a los switches para que redirijan el tráfico local, pero además, en este caso, tendremos que pasarle un mapa que representa el grafo aleatorio generado que indica las conexiones entre los distintos puertos de los switches OpenFlow (enlaces_guardados). De esta manera, conseguiremos que el controlador conozca la red aleatoria generada para así hacer el routing correspondiente para esa estructura de red. En el momento que el controlador se haya conectado a los switches OpenFlow, sin que haya ninguna comunicación previa, el controlador rellenará las tablas de flujo de los switches para hacer posible la comunicación entre los host que conforman la red.

El routing implementado para las topología *Jellyfish* se divide en dos partes principales. Por una parte, el tráfico local (entradas de la tabla de flujo cuya acción sea redistribuir la información por alguno de los puertos hacia abajo del switch OpenFlow) y, por otra parte, el tráfico switch-to-switch aleatorio (entradas de la tabla de flujo cuya acción sea redistribuir la información por alguno de los puertos que se conectan con otros switches OpenFlow). El primer caso se implementa analizando la dirección IP destino. Según la IP de destino, si el paquete corresponde a ese mismo switch, ese switch redirigirá el paquete por el puerto correspondiente al host de destino.

En cambio, la distribución del tráfico entre switches es bastante compleja, ya que el camino para llegar de un nodo origen a un nodo destino no es único. Para ello, el controlador utiliza un algoritmo que recorre el grafo aleatorio en anchura [21] [22] para buscar todos los caminos entre los nodos de origen y destino. El algoritmo devuelve, dado el grafo que representa las conexiones entre los switches, un nodo origen y un nodo destino, una lista de todos los caminos posibles para llegar desde ese nodo origen hasta ese nodo destino, ordenados desde el camino más corto hasta el más largo (la longitud del camino se mide en el número de saltos para llegar del nodo origen al nodo destino).

Según el criterio tomado, en caso de que encuentre varios caminos de la misma longitud, se cogerá el primero identificado por el algoritmo. Por lo tanto, para cada nodo origen, se irá guardando el primer camino mínimo respecto a los demás destinos, en otras palabras, el primer elemento (camino) de la lista que representa todos los caminos posibles entre dos nodos. Dicho esto, se deduce que cada switch OpenFlow contará con las siguientes entradas destinadas para reenviar el tráfico switch-to-switch:

- *(NumSwitches - 1)* entradas
    - *L* entradas destinadas para distribuir tráfico a switches directamente conectados
    - *(NumSwitches - 1) - L* entradas para distribuir tráfico a switches que no están directamente conectados

Los flujos orientados al tráfico local se redirigen a través de los puertos hacia abajo de los switches (puertos 1...K). En cada switch, se instalan de manera idéntica las reglas orientadas a redirigir el tráfico local.

- Identificar cuál es el switch con el que nos estamos comunicando y crear la entrada correspondiente para llegar a cada uno de los hosts:
    - Para cada puerto hacia abajo del switch:
        - *10.0.switch.host*

Los flujos orientados para distribuir tráfico switch-to-switch se basan principalmente en el array donde se han guardado cada uno de los caminos mínimos para cada switch de origen, respecto a los demás destinos. Debemos tener en cuenta que un camino mínimo, por muchos saltos que tenga desde el origen hasta el destino, siempre es la composición de caminos mínimos más cortos.

Ejemplo: Si sabemos que el camino mínimo entre los switches 'S_0' y ' S_4' es 'S_0 - S_2 – S3 - S_4', se deduce que todos los caminos que componen ese camino mínimo, también son caminos mínimos. Eso significa que no habrá que instalar flujos en los nodos intermedios de los caminos, ya que se repetirían los flujos, con instalarlo en el primer switch de cada camino es suficiente. Por tanto, analizando cada posición del array (cada camino más corto), los flujos se instalan de la siguiente manera en cada uno de los switches OpenFlow:

- Comunicarnos con cada uno de los switches OpenFlow a través del identificador correspondiente (*idint).*

- Calcular qué caminos mínimos debe procesar cada switch, para ser exactos, *(numSwitches-1)* caminos. Para cada camino a procesar, generar un flujo. La lista de caminos mínimos es *all_shortest_paths.*

    - El *primer identificador* de cada camino mínimo identifica cuál es el switch donde debe instalarse el flujo, con las siguientes características:

        - *Matching:* El *último identificador* de cada camino mínimo determina cuál será la subred de destino (switch de destino) para llegar al fín de ese camino. Por lo tanto, ese identificador representa cuál será la dirección IP destino para completar el *matching* del flujo.
            - 10.0.*switch*.0/*24*

- *Output action:* Para poder completar el flujo, únicamente nos falta detectar por qué puerto de salida del switch de origen (primer identificador) debemos distribuir la información. Para ello, el *segundo identificador* de cada camino nos indica por qué puerto del primer switch debemos distribuir el tráfico switch-to-switch. Una vez detectemos cuál es la posición del primer switch en la lista de enlaces que representa el grafo, calcularemos el orden de aparición de ese primer switch para saber por qué puerto reenviar el flujo. Por ejemplo, la primera aparición de un switch en esa lista hace referencia al puerto "*K+1",* la segunda aparición, al puerto "*K+2"* y, la última aparición, al puerto "*K+L".*

Cabe destacar que cada vez que generamos una topología *Jellyfish*, se genera un grafo aleatorio distinto. Esto significa que cada vez que se genera un grafo se necesita una configuración distinta de los flujos, ya que el conjunto de caminos mínimos es distinto. En nuestro caso, primero generamos el grafo con Mininet y luego se lo pasamos a una aplicación para que el controlador POX gestione la carga proactiva de flujos. Para poder implementar el routing para casos aleatorios controlados, se ha utilizado la semilla aleatoria (seed). El controlar la semilla aleatoria nos permite replicar experimentos para así poder trabajar con esos casos aleatorios controlados.

PRIORIDADES

Las entradas que hacen posible el routing completo se dividen en dos partes. Por una parte, aquellas entradas que se encargan de redirigir el tráfico local, basado en direcciones IPv4 destino (*nw_dst*), y por otra parte, aquellas entradas que se encargan de redirigir el tráfico entre distintos switches, también basado en direcciones IPv4 destino *(nw_dst)*. A diferencia de las topologías *tree*, en la topología *Jellyfish* no se dará el caso que un paquete coincida con más de una entrada en una tabla de flujos. Esto significa que no habrá que recurrir a la prioridad del flujo para romper los posibles empates, ya que no los hay. Aun así, se ha decidido que a las entradas relacionadas para redirigir el tráfico local se les de mayor prioridad que a las entradas relacionadas para redirigir el tráfico switch-to-switch.

Para entender mejor cómo se comunica el controlador con los diferentes switches y cómo instala las entradas correspondientes para cada switch específico de manera proactiva, en el ANEXO E se muestra el módulo que carga el controlador POX, desarrollado en Python3, llamado jelly*_controller_julen.py*.

9.2.3 Ejemplo Jellyfish completo

Para finalizar con las topologías Jellyfish, se demostrará la conectividad entre los nodos a través del comando *pingall* de Mininet, comunicación todos con todos. El ejemplo con el que se trabaja es el *jf,7,3,2* con semilla 27.

Se indicarán qué parámetros debemos pasarle tanto al script que se encarga de generar la topología como al controlador. Como bien sabemos, el controlador manda mensajes para rellenar las tablas de flujos de los switches sin que haya comunicación previa. Por ello, también se mostrarán algunas tablas de flujo para entender cómo se distribuye el tráfico y permitir la comunicación entre los hosts.

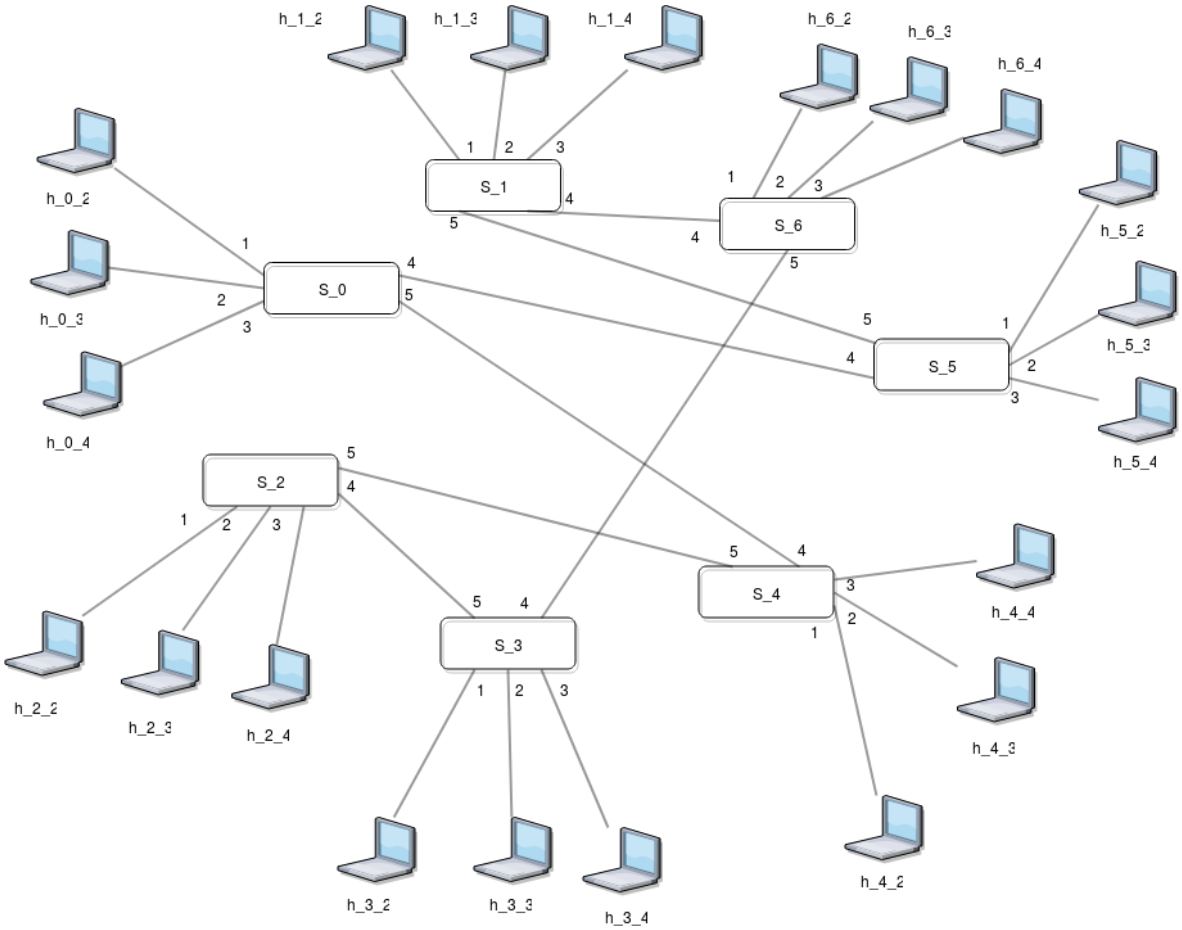

Figura 9.23: topología jf,7,3,2 generada con semilla 27

Controlador:

Lanzar controlador

```
root@mininet-vm:/home/mininet/pox# ./pox.py forwarding.jelly_controller_julen
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
Todos los caminos cortos para cada nodo de origen son:
[['S_0', 'S_5', 'S_1'], ['S_0', 'S_4', 'S_2'], ['S_0', 'S_4', 'S_2', 'S_3'], ['S_0', 'S_4'], ['S_
0', 'S_5'], ['S_0', 'S_5', 'S_1', 'S_6'], ['S_1', 'S_5', 'S_0'], ['S_1', 'S_6', 'S_3', 'S_2'], ['
S_1', 'S_6', 'S_3'], ['S_1', 'S_5', 'S_0', 'S_4'], ['S_1', 'S_5'], ['S_1', 'S_6'], ['S_2', 'S_4',
 'S_0'], ['S_2', 'S_3', 'S_6', 'S_1'], ['S_2', 'S_3'], ['S_2', 'S_4'], ['S_2', 'S_4', 'S_0', 'S_5
'], ['S_2', 'S_3', 'S_6'], ['S_3', 'S_2', 'S_4', 'S_0'], ['S_3', 'S_6', 'S_1'], ['S_3', 'S_2'], [
'S_3', 'S_2', 'S_4'], ['S_3', 'S_6', 'S_1', 'S_5'], ['S_3', 'S_6'], ['S_4', 'S_0'], ['S_4', 'S_0'
, 'S_5', 'S_1'], ['S_4', 'S_2'], ['S_4', 'S_2', 'S_3'], ['S_4', 'S_0', 'S_5'], ['S_4', 'S_2', 'S_
3', 'S_6'], ['S_5', 'S_0'], ['S_5', 'S_1'], ['S_5', 'S_0', 'S_4', 'S_2'], ['S_5', 'S_1', 'S_6', '
S_3'], ['S_5', 'S_0', 'S_4'], ['S_5', 'S_1', 'S_6'], ['S_6', 'S_1', 'S_5', 'S_0'], ['S_6', 'S_1']
, ['S_6', 'S_3', 'S_2'], ['S_6', 'S_3'], ['S_6', 'S_3', 'S_2', 'S_4'], ['S_6', 'S_1', 'S_5']]
INFO:core:POX 0.5.0 (eel) is up.
```

Figura 9.24: lanzar el controlador con el módulo jelly*_controller_julen*

Una vez se genera la topología, comprobar que el controlador se ha conectado con los switches OpenFlow creados.

```
root@mininet-vm:/home/mininet/pox# ./pox.py forwarding.jelly_controller_julen
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
Todos los caminos cortos para cada nodo de origen son:
[['S_0', 'S_5', 'S_1'], ['S_0', 'S_4', 'S_2'], ['S_0', 'S_4', 'S_2', 'S_3'], ['S_0', 'S_4'], ['S_
0', 'S_5'], ['S_0', 'S_5', 'S_1', 'S_6'], ['S_1', 'S_5', 'S_0'], ['S_1', 'S_6', 'S_3', 'S_2'], ['
S_1', 'S_6', 'S_3'], ['S_1', 'S_5', 'S_0', 'S_4'], ['S_1', 'S_5'], ['S_1', 'S_6'], ['S_2', 'S_4',
 'S_0'], ['S_2', 'S_3', 'S_6', 'S_1'], ['S_2', 'S_3'], ['S_2', 'S_4'], ['S_2', 'S_4', 'S_0', 'S_5
'], ['S_2', 'S_3', 'S_6'], ['S_3', 'S_2', 'S_4', 'S_0'], ['S_3', 'S_6', 'S_1'], ['S_3', 'S_2'], [
'S_3', 'S_2', 'S_4'], ['S_3', 'S_6', 'S_1', 'S_5'], ['S_3', 'S_6'], ['S_4', 'S_0'], ['S_4', 'S_0'
, 'S_5', 'S_1'], ['S_4', 'S_2'], ['S_4', 'S_2', 'S_3'], ['S_4', 'S_0', 'S_5'], ['S_4', 'S_2', 'S_
3', 'S_6'], ['S_5', 'S_0'], ['S_5', 'S_1'], ['S_5', 'S_0', 'S_4', 'S_2'], ['S_5', 'S_1', 'S_6', '
S_3'], ['S_5', 'S_0', 'S_4'], ['S_5', 'S_1', 'S_6'], ['S_6', 'S_1', 'S_5', 'S_0'], ['S_6', 'S_1']
, ['S_6', 'S_3', 'S_2'], ['S_6', 'S_3'], ['S_6', 'S_3', 'S_2', 'S_4'], ['S_6', 'S_1', 'S_5']]
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[01-00-00-00-00-05 6] connected
INFO:openflow.of_01:[01-00-00-00-00-00 5] connected
INFO:openflow.of_01:[01-00-00-00-00-04 2] connected
INFO:openflow.of_01:[01-00-00-00-00-03 4] connected
INFO:openflow.of_01:[01-00-00-00-00-02 3] connected
INFO:openflow.of_01:[01-00-00-00-00-06 7] connected
INFO:openflow.of_01:[01-00-00-00-00-01 8] connected
```

Figura 9.25: Comprobar que los switches OpenFlow se han conectado al controlador

Parámetros necesarios para lanzar el controlador:

- numSwitches: Para implementar tráfico switch-to-switch
- K: Para implementar tráfico local
- mapa: Grafo que representa las conexiones entre los switches y así el controlador pueda calcular los caminos mínimos para cada switch de origen
- enlaces_guardados: Lista que representa las conexiones entre los distintos puertos de los switches. Necesario para saber por qué puerto de salida distribuir el tráfico

Topología:

Generar topología

```
root@mininet-vm:/home/mininet/ripl/ripl# sudo mn --custom mn.py --topo jf,7,3,2 --controller remo
te --arp
Lista de enlaces guardados generada que pasaremos al controlador:
[('S_6', 'S_1'), ('S_0', 'S_5'), ('S_5', 'S_1'), ('S_6', 'S_3'), ('S_0', 'S_4'), ('S_3', 'S_2'),
('S_4', 'S_2')]
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h_0_2 h_0_3 h_0_4 h_1_2 h_1_3 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_2 h
_5_3 h_5_4 h_6_2 h_6_3 h_6_4
*** Adding switches:
S_0 S_1 S_2 S_3 S_4 S_5 S_6
*** Adding links:
(S_0, S_4) (S_0, S_5) (S_3, S_2) (S_4, S_2) (S_5, S_1) (S_6, S_1) (S_6, S_3) (h_0_2, S_0) (h_0_3,
 S_0) (h_0_4, S_0) (h_1_2, S_1) (h_1_3, S_1) (h_1_4, S_1) (h_2_2, S_2) (h_2_3, S_2) (h_2_4, S_2)
(h_3_2, S_3) (h_3_3, S_3) (h_3_4, S_3) (h_4_2, S_4) (h_4_3, S_4) (h_4_4, S_4) (h_5_2, S_5) (h_5_3
, S_5) (h_5_4, S_5) (h_6_2, S_6) (h_6_3, S_6) (h_6_4, S_6)
*** Configuring hosts
h_0_2 h_0_3 h_0_4 h_1_2 h_1_3 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_2 h
_5_3 h_5_4 h_6_2 h_6_3 h_6_4
*** Starting controller
c0
*** Starting 7 switches
S_0 S_1 S_2 S_3 S_4 S_5 S_6 ...
*** Starting CLI:
mininet>
```

Figura 9.26: lanzar la topología jf,7,3,2 con el comando *sudo mn*

Parámetros necesarios para lanzar la topología:

- semilla: Para implementar el routing para casos aleatorios controlados. De esta manera, antes de lanzar la topología, sabremos cuál será el grafo que representa las conexiones
- numSwitches: Número de switches que habrá en la *Jellyfish*
- K: Número de *down-ports* de cada switch (switch-to-host)
- L: Número de puertos aleatorios de cada switch (switch-to-switch)

Tablas de flujo:

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows S_0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=337.742s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=2,ip,nw_dst=10.0.0.2 actions=output:1
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=2,ip,nw_dst=10.0.0.4 actions=output:3
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=2,ip,nw_dst=10.0.0.3 actions=output:2
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=1,ip,nw_dst=10.0.4.0/24 actions=output:5
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=1,ip,nw_dst=10.0.2.0/24 actions=output:5
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=1,ip,nw_dst=10.0.6.0/24 actions=output:4
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=1,ip,nw_dst=10.0.3.0/24 actions=output:5
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=1,ip,nw_dst=10.0.5.0/24 actions=output:4
 cookie=0x0, duration=337.719s, table=0, n_packets=0, n_bytes=0, idle_age=337, p
riority=1,ip,nw_dst=10.0.1.0/24 actions=output:4
```

Figura 9.27: tabla de flujo del switch S_0

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows S_5
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=2,ip,nw_dst=10.0.5.3 actions=output:2
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=2,ip,nw_dst=10.0.5.4 actions=output:3
 cookie=0x0, duration=362.465s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=2,ip,nw_dst=10.0.5.2 actions=output:1
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=1,ip,nw_dst=10.0.4.0/24 actions=output:4
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=1,ip,nw_dst=10.0.2.0/24 actions=output:4
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=1,ip,nw_dst=10.0.6.0/24 actions=output:5
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=1,ip,nw_dst=10.0.3.0/24 actions=output:5
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=1,ip,nw_dst=10.0.0.0/24 actions=output:4
 cookie=0x0, duration=362.442s, table=0, n_packets=0, n_bytes=0, idle_age=362, p
riority=1,ip,nw_dst=10.0.1.0/24 actions=output:5
```

Figura 9.28: tabla de flujo del switch S_5

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows S_1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=2,ip,nw_dst=10.0.1.4 actions=output:3
 cookie=0x0, duration=381.151s, table=0, n_packets=0, n_bytes=0, idle_age=381, p
riority=2,ip,nw_dst=10.0.1.2 actions=output:1
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=2,ip,nw_dst=10.0.1.3 actions=output:2
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=1,ip,nw_dst=10.0.4.0/24 actions=output:5
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=1,ip,nw_dst=10.0.2.0/24 actions=output:4
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=1,ip,nw_dst=10.0.6.0/24 actions=output:4
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=1,ip,nw_dst=10.0.3.0/24 actions=output:4
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=1,ip,nw_dst=10.0.0.0/24 actions=output:5
 cookie=0x0, duration=381.14s, table=0, n_packets=0, n_bytes=0, idle_age=381, pr
iority=1,ip,nw_dst=10.0.5.0/24 actions=output:5
```

Figura 9.29: tabla de flujo del switch S_1

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows S_6
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=416.206s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=2,ip,nw_dst=10.0.6.2 actions=output:1
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=2,ip,nw_dst=10.0.6.3 actions=output:2
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=2,ip,nw_dst=10.0.6.4 actions=output:3
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=1,ip,nw_dst=10.0.4.0/24 actions=output:5
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=1,ip,nw_dst=10.0.2.0/24 actions=output:5
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=1,ip,nw_dst=10.0.3.0/24 actions=output:5
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=1,ip,nw_dst=10.0.0.0/24 actions=output:4
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=1,ip,nw_dst=10.0.5.0/24 actions=output:4
 cookie=0x0, duration=416.167s, table=0, n_packets=0, n_bytes=0, idle_age=416, p
riority=1,ip,nw_dst=10.0.1.0/24 actions=output:4
```

Figura 9.30: tabla de flujo del switch S_6

En la tabla 9.10, se muestra la información de la tabla de flujos del Switch S_0. La columna matching es la coincidencia para los paquetes, la columna action es el puerto de salida por el cual se reenvía el paquete al que se la ha encontrado coincidencia y la columna priority es la prioridad que se le ha dado a un flujo, aunque en este caso no tiene ninguna relevancia.

Tabla 9.10: Tabla de flujo del switch S_0

Switch	Flujo	Matching	Action	Priority	Nota
S_0	1	10.0.0.2	Puerto salida 1	2	Tráfico local
	2	10.0.0.4	Puerto salida 3	2	Tráfico local
	3	10.0.0.3	Puerto salida 2	2	Tráfico local
	4	10.0.4.0/24	Puerto salida 5	1	Tráfico sw-to-sw
	5	10.0.2.0/24	Puerto salida 5	1	Tráfico sw-to-sw
	6	10.0.6.0/24	Puerto salida 4	1	Tráfico sw-to-sw
	7	10.0.3.0/24	Puerto salida 5	1	Tráfico sw-to-sw
	8	10.0.5.0/24	Puerto salida 4	1	Tráfico sw-to-sw
	9	10.0.1.0/24	Puerto salida 4	1	Tráfico sw-to-sw

Conectividad entre hosts:

```
mininet> pingall
*** Ping: testing ping reachability
h_0_2 -> h_0_3 h_0_4 h_1_2 h_1_3 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_0_3 -> h_0_2 h_0_4 h_1_2 h_1_3 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_0_4 -> h_0_2 h_0_3 h_1_2 h_1_3 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_1_2 -> h_0_2 h_0_3 h_0_4 h_1_3 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_1_3 -> h_0_2 h_0_3 h_0_4 h_1_2 h_1_4 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_1_4 -> h_0_2 h_0_3 h_0_4 h_1_2 h_1_3 h_2_2 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_2_2 -> h_0_2 h_0_3 h_0_4 h_1_2 h_1_3 h_1_4 h_2_3 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_2_3 -> h_0_2 h_0_3 h_0_4 h_1_2 h_1_3 h_1_4 h_2_2 h_2_4 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
h_2_4 -> h_0_2 h_0_3 h_0_4 h_1_2 h_1_3 h_1_4 h_2_2 h_2_3 h_3_2 h_3_3 h_3_4 h_4_2 h_4_3 h_4_4 h_5_
2 h_5_3 h_5_4 h_6_2 h_6_3 h_6_4
```

Figura 9.31: demostrar la conectividad entre los host que hay en la topología jf,7,3,2 (1)

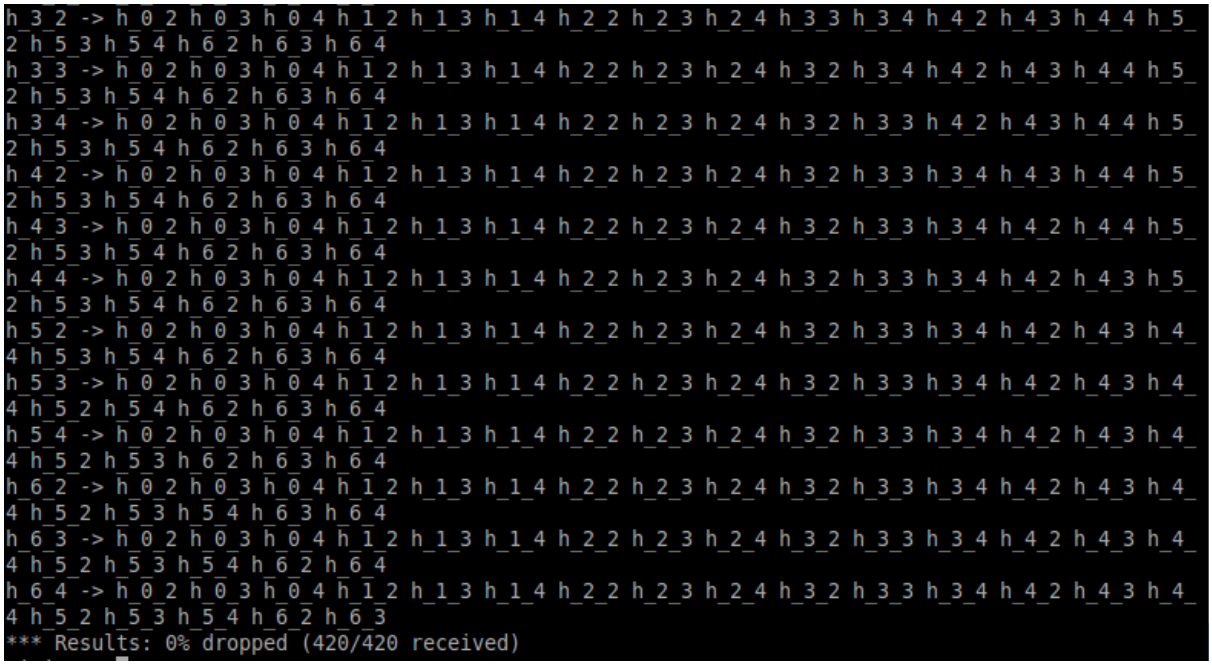

Figura 9.32: demostrar la conectividad entre los host que hay en la topología jf,7,3,2 (2)

9.3 El protocolo ARP

Además de implementar el routing, al estar trabajando con direcciones IP, es necesario resolver el protocolo ARP [23]. ARP es el protocolo de comunicaciones de la capa enlace, responsable de encontrar la dirección MAC correspondiente a una dirección IP para permitir a los hosts poder realizar conexiones a través de direcciones IP.

Para resolver el protocolo ARP, existen tres alternativas:

- Broadcast (entradas dinámicas): Cada host hace un broadcast masivo por toda la red para dar a conocer la dirección física correspondiente de cada uno de ellos con sus respectivas direcciones IP. Esta opción genera tráfico ARP de tipo inundación (flooding) en la red y no es aplicable en topologías donde existen bucles.

- Evitar ARP (entradas estáticas): Evitar la difusión de los mensajes ARP, poblando de manera estática las tablas de los host. Existen dos maneras:

    - opción *--arp*: Alternativa que proporciona Mininet para rellenar las tablas de manera automática, añadiendo la opción "*--arp"* al comando "*sudo mn".* Esta opción es válida solo en el entorno de emulación Mininet.

- El programador de la red conoce de antemano qué dirección física le corresponde a cada host. Por lo tanto, el programador configura un fichero indicando la dirección física correspondiente a la dirección IP de cada uno de los host, de tal manera que cuando el host arranque, las tablas de los host se completarán automáticamente, evitando cualquier tipo de tráfico.

- El controlador se encarga de responder los mensajes ARP (entradas dinámicas): Cada vez que algún switch reciba un mensaje ARP y no sepa qué hacer con él, se lo manda al controlador. El controlador conoce todas las parejas MAC-IP de los host. Éste, le dirá qué hacer con el paquete. Con el tiempo, se irán añadiendo aquellas entradas que ha ido aprendiendo de manera dinámica.

En este TFG, hemos optado por usar la opción –arp de Mininet al lanzar la topología, principalmente porque es la alternativa más sencilla y la más eficaz de todas, ya que no genera ningún tipo de tráfico en la red. Como bien se ha indicado, Mininet se encarga de rellenar las tablas de reenvío de los hosts, asociando a cada dirección IP, su dirección física correspondiente y así resolver el protocolo ARP, permitiendo la comunicación basada en IPv4.

En un caso real, la opción por la que optaría sería la de configurar un fichero con todas las correspondencias MAC-IP de cada host e indicar a los host que carguen ese fichero cada vez que arranquen, para que sus correspondientes tablas sean completadas de manera automática.

9.4 Evaluación de las topologías *tree*

En este apartado, por una parte, se analizarán las topologías de la familia *tree* generadas (*simple tree, thin-tree* y *fat-tree)* en cuanto a coste y rendimiento. El coste dependerá del grado de los switches (número de puertos, *"K+L"* en el caso de los switches de la capa agregación y borde, y *"K",* en el caso de los switches de la capa core), cantidad de switches y número de enlaces. Por otra parte, para justificar el rendimiento de cada una de las topologías me apoyaré en la evaluación que se ha hecho en el artículo [24] sobre las topologías de la familia *tree.*

La nomenclatura que utiliza el artículo mencionado para estudiar los distintos árboles es idéntica a la que se utiliza en este TFG. En este TFG, se demuestra únicamente la conectividad entre nodos, suponiendo que el tráfico que generan los mismos es homogéneo. En el artículo, se hacen distintos experimentos sobre cada una de las topologías *tree*, ejecutando diferentes aplicaciones de diferentes características, para medir el rendimiento de las mismas.

Para hacer una comparación lógica, se ajustan dos parámetros. Por una parte, las topologías estarán compuestas por tres capas (core, agregación y borde), al igual que en el TFG, y por otra parte, se limita el tamaño de la red, es decir, el número de hosts a los que podremos dar servicio, 512 para ser exactos. En consecuencia, el parámetro '*K*' deberá ser 8. Habiendo definido estos parámetros, en la tabla 9.11 se muestraa cuántos switches y enlaces necesitaríamos para generar cada topología, así como cuál ha sido el rendimiento obtenido mediante los experimentos y simulaciones que se describen en el artículo. El valor del rendimiento de cada una de las topologías, medido en *"paquetes/segundo",* se ha estudiado ejecutando distintos tipos de aplicaciones sobre las mismas [24]. En la tabla 9.11, el valor del rendimiento está normalizado al del *fat-tree*, que es el máximo.

Tabla 9.11: características de las topologías *tree* de tres niveles y K=8. Asimismo, resultados obtenidos en cuanto a rendimiento a través de los experimentos que se hacen en el artículo

topología (tree,K,L)	Switches	Enlaces	Hosts	Rendimiento
tree,8,1 (simple)	73	584	512	0.1410
tree,8,2 (thin)	84	672	512	0.4272
tree,8,3 (thin)	97	776	512	0.6746
tree,8,4 (thin)	112	896	512	0.8273
tree,8,5 (thin)	129	1032	512	0.9088
tree,8,6 (thin)	148	1184	512	0.9523
tree,8,7 (thin)	169	1352	512	0.9695
tree,8,8 (fat)	192	1536	512	1.0000

Si nos damos cuenta, como es normal, el rendimiento aumenta en función del aumento del número de switches (coste) y número de enlaces (coste). Al contar con más switches y más enlaces, la topología es más flexible a la hora redistribuir el tráfico por camino alternativos, evitando congestiones y cuellos de botella en diferentes switches.

Antes de analizar los distintos casos, en la tabla 9.12 se recoge la reducción del coste del número de switches y enlaces respecto al *fat-tree*.

El porcentaje de la reducción del coste en número de switches o enlaces para una topología *tree* respecto a la topología *fat-tree* se computa de la siguiente manera:

$$A = (1 - (B / C)) * 100$$

donde:

- A = reducción del coste de una topología de familia *tree* en enlaces o número de switches respecto al *fat-tree*

- B = número de switches o enlaces de la topología que queremos calcular la reducción

- C = número de switches o enlaces de la topología *fat-tree*

Tabla 9.12: Reducción del coste de cada topología respecto al *fat-tree*

topología (tree,K,L)	Switches	Enlaces	Reducción de coste
tree,8,1 (simple)	73	584	61.97%
tree,8,2 (thin)	84	672	56.25%
tree,8,3 (thin)	97	776	49.47%
tree,8,4 (thin)	112	896	41.67%
tree,8,5 (thin)	129	1032	32.81%
tree,8,6 (thin)	148	1184	22.91%
tree,8,7 (thin)	169	1352	11.97%
tree,8,8 (fat)	192	1536	00.00%

En primer lugar, el caso del *simple-tree* lo descartaría, ya que, a pesar de que el coste sea muy bajo, el rendimiento que ofrece también es muy bajo (14.10% respecto al *fat-tree*). En segundo lugar, descartaría el caso del *fat-tree*, ya que éste, ofrece un rendimiento óptimo a cambio de un coste muy elevado. Por lo tanto, en mi opinión, lo ideal sería buscar alguna alternativa entre los *thin-tree*, teniendo en cuenta qué tipo de aplicaciones se ejecutarían sobre la red. En el caso del *thin-tree,8,5* podemos observar que se obtienen unos resultados excelentes, ya que consiguiríamos más o menos un rendimiento 10% menor que el que nos ofrece el *fat-tree,* pero, el coste en número de switches y enlaces se reduciría en un 32.81%. Y si a esto le sumamos que el grado de cada uno de los switches que componen la capa de agregación y borde también se reduce de 16 *(fat-tree)* a 13 (*thin-tree,8,5)*, el coste será aún menor. Por lo tanto, conseguiremos reducir un coste total de la topología en un poco más de un 32.81%, a cambio de reducir únicamente el rendimiento de la misma de 100% a 90.88%

Aunque en este apartado se proponga una solución, lo ideal sería analizar las aplicaciones que se ejecutarán realmente y qué características tienen las mismas. Por ejemplo, qué porcentaje de la aplicación explota la comunicación local y qué porcentaje de la aplicación explota comunicación externa. En base a ese análisis, nos orientaremos más a topologías que explotan localidad o no. Pero esta evaluación es algo que ha quedado fuera del alcance de este TFG.

9.5 Comparativa entre las topologías *tree* y *Jellyfish*

En cuanto a rendimiento, según unos estudios, la topología *fat-tree* es la que más se asemeja a la topología *Jellyfish.* Por lo tanto, la comparativa se hará principalmente entre la *Jellyfish* y los *fat-tree*.

Para empezar, podemos decir que en cuanto a coste, la topología *Jellyfish* es menos costosa en comparación con los *fat-tree*, ya que necesitamos menos switches para generarla. Por ejemplo, si tuviéramos que dar servicio a 512 host en un datacenter, en la topología *fat-tree* necesitaríamos 192 switches. En cambio, si quisiéramos utilizar una topología *Jellyfish*, con 64 switches nos bastaría. La principal razón de la reducción del número de switches es porque en la topología *Jellyfish*, a todos y cada uno de los switches se le conectan hosts. Sin embargo, en las topologías de familia *tree*, solamente se conectan hosts a los switches que componen la capa borde.

Cabe destacar que el hecho de que en los *tree* se definan distintas capas para distribuir el tráfico, posibilita optimizar el routing agrupando las direcciones IP. Esta optimización no es bajo ningún concepto aplicable para la topología *Jellyfish,* ya que al ser una topología totalmente aleatoria, los switches se conectan de cualquier manera según el parámetro *L,* y por lo tanto, para cada subred de destino, habrá una entrada distinta en cada una de las tablas de flujo de cada uno de los switches OpenFlow.

Resumiendo, podemos decir que el alto coste de la topología de los *fat-tree* se compensa después con la posibilidad de optimizar el routing, reduciendo considerablemente el número de entradas en las tablas de flujo de los switches, con la técnica de agrupar las direcciones IP. En cambio, aunque la topología *Jellyfish* sea bastante menos costosa que los *fat-tree*, implementar un routing óptimo supone un verdadero reto.

Para terminar, en el artículo [20] se analiza el rendimiento de la topología *Jellyfish* y se hace una comparativa con los *fat-tree*. Para que nos hagamos una idea, estos son algunos resultados remarcables que se han obtenido:

- Con el mismo equipamiento (mismo número de switches de mismo grado), la topología *Jellyfish* puede dar servicio a un 25% más de hosts proporcionando un mismo rendimiento (throughput) que la topología *fat-tree.*

- La topología *fat-tree* está compuesta por caminos más largos que la topología *Jellyfish.* En la figura 9.33 podemos observar el número de nodos accesibles en según el número de saltos.

    - *Fat-tree (16 nodos, 20 switches, grado 4):* Desde un nodo aleatorio, en menos de 6 hops (saltos), únicamente alcanzaremos 3 de los 15 nodos restantes.

- *Jellyfish (16 nodos, 20 switches, grado 4):* Desde un nodo aleatorio, en menos de 6 hops (saltos), podremos alcanzar 11 de los 15 nodos restantes.

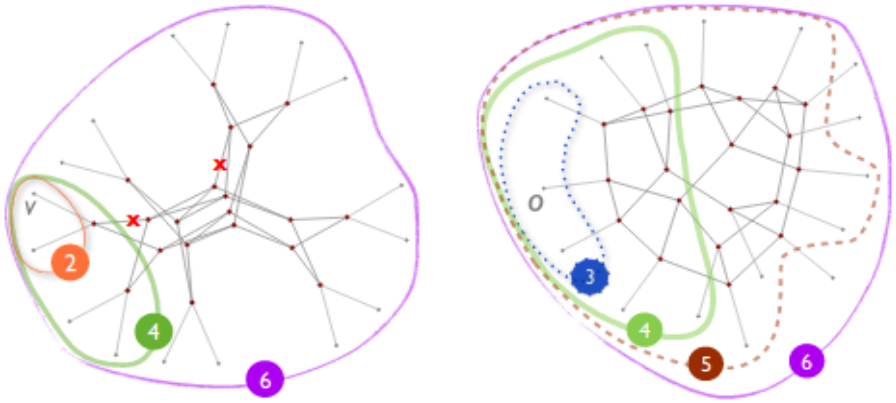

Figura 9.33: nodos accesibles según el número de saltos: (a) Fat-tree 20 switches (grado 4), 16 host (b) Jellyifish mismo equipamiento [20]

- Para dar servicio a un mismo número de nodos, se necesita un 20% menos de cable de switch a switch, respecto a los *fat-tree*. Esto supone que podremos dar servicio a un mismo número de nodos, reduciendo el coste del cableado en un 20%.

9.6 Memoria local de los switches OpenFlow

Los switches convencionales utilizan la tabla *forwarding information base (FIB),* mientras que los switches OpenFlow utilizan la tabla de flujos para determinar por qué interfaz direccionar los paquetes entrantes. Hasta ahora un parámetro que no hemos tenido en cuenta en los switches openFlow ha sido la cantidad de memoria que tienen para guardar cada una de las entradas en su tabla de flujo, también conocida como la memoria local de los switches.

Para que nos hagamos una idea, un switch normal tiene la capacidad de guardar entre 100K y 1000K de entradas en su *FIB* y un switch OpenFlow tiene la capacidad de guardar entre 1K y 10K de entradas en su tabla de flujo [25]. El número de entradas es algo que depende del modelo del switch.

En el routing clásico que se hace en Internet, muchos son los protocolos que ayudan a que el routing sea posible y escalable. Sin embargo, los switches utilizados en las SDN, concretamente en aquellas estructuras de red que se utilizan en datacenter, cuentan con un número bastante limitado de entradas en la tabla de flujos y por lo tanto, hay que pensar en algunas estrategias para hacer que el routing sea escalable y eficaz. Cabe destacar que los

switches que más entradas tendrán a priori son aquellos switches que están conectados a los host, es decir, los switches que dan acceso a los host, ya que para cada host conectado tendrá que tener asociada una entrada distinta y en ese caso no hay manera de reducir el tamaño de la tabla de flujos.

Una técnica muy utilizada para hacerle frente al problema de tener que definir una regla (entrada) distinta para cada destino diferente es agrupar los diferentes destinos en bloques más grandes, en la medida de los posible. De esta manera, conseguiremos no tener una entrada para cada destino, sino que tendremos los destinos de manera agrupada. Una vez llegue la información al switch de acceso, éste último será el que finalmente se encargue de que la información llegue al host, reenviando la información por el puerto correspondiente.

Para que la agrupación en bloques más grandes se realice eficientemente depende de manera directa del criterio que tomamos para asignar las direcciones a los host, ya sean direcciones IP o MAC. Si hacemos una correcta asignación, disminuyendo la máscara conseguiremos agrupar diferentes destinos de manera muy sencilla y conseguiremos reducir considerablemente el número de entradas de la tabla de flujo.

Por ejemplo, en un *tree*, si se trabaja con redes de 512 hosts, se necesitarían 64 switches en la capa borde. Cada switch de la capa borde tendrá, en el peor de los casos, tantas entradas como número de hosts que hay en la red y una regla distinta para llegar a los distintos nodos de la red. Esta opción no es escalable bajo ningún concepto. De hecho, las tablas de flujos de los switches se saturarían y éstos se verían obligados a tener que eliminar entradas para añadir nuevas, algo nada viable. Para afrontar este problema, en la praćtica, se utiliza una técnica de optimización del tamaño de las tablas de flujo, basada en la agrupación de direcciones IPv4, para aprovechar mejor la memoria local de los switches OpenFlow y hacer que el routing sea mucho más escalable.

Para finalizar con esta sección, se incluyen el número de flujos que requieren los switches para las distintas topologías, en función de los parámetros ‘K’, ‘L’ y ‘numSwitches’.

TOPOLOGÍAS FAMILIA *TREE*

- Switches capa edge: K (local) + L (sw-to-sw hacia arriba)
- Switches capa agregación: K (sw-to-sw hacia abajo) + L (sw-to-sw hacia arriba)
- Switches capa core: K (sw-to-sw hacia abajo)

TOPOLOGÍAS *JELLYFISH*

- K (local) + (numSwitches-1) (sw-to-sw)

Nota: Si en las topologías *Jellyfish,* se quisiera hacer un routing *multipath*, es decir, distribuir la carga por distintos caminos mínimos, el número de entradas en las tablas de flujo sería aún mayor

Cada una de las entradas en las tablas de flujo tienen un coste en la memoria, y esa memoria depende directamente del número de flujos a instalar. En caso de los *tree*, ya que permite la agrupación de direcciones, es una ventaja. Sin embargo, en las topologías *Jellyfish*, no hay manera de optimizar el tamaño de las tablas de flujo y puede resultar un problema si la red es muy grande.

9.7 Coste de una topología y sobresusripción

1. COSTE PARA GENERAR UNA TOPOLOGÍA

En este apartado, se analizan qué factores son los que influyen de manera directa en el coste para generar cualquiera de estas topologías en un caso real. El coste es el factor principal que tienen en cuenta la mayoría de las empresas a la hora de implantar una nueva estructura de red. En la siguiente lista se detallan cuáles son los factores:

- Número de switches OpenFlow
    - Grado del switch (número de puertos)
        - Cuantos más puertos tenga el switch, más caro
- Número de enlaces

Debemos tener en cuenta que el precio de los switches OpenFlow disminuye considerablemente respecto a los convencionales, ya que no cuentan con ningún tipo de inteligencia. Esa inteligencia se les aporta a través del controlador y éste debe estar programado de manera adecuada para instruir a los switches para que se comporten de manera apropiada.

2. SOBRESUSCRIPCIÓN

El concepto de sobresuscripción podemos entenderlo como lo que pasaría si todos los host de una misma subred quisieran intercambiar paquetes con otros hosts de otra subred al mismo tiempo. La sobresuscripción en los switches de la capa borde de las topologías *tree* la podemos definir como la división entre los puertos hacia abajo de un switch y los puertos hacia arriba de un switch:

$$S = (\text{down-ports } (K)) \,/\, (\text{up-ports o puertos switch-to-switch } (L))$$

En la tabla 9.13 se recogen los valores de sobresuscripción de los switches que se encuentran en la capa *borde*, según las distintas topologías de la familia *tree*. Debemos tener en cuenta que en los *tree,* excepto en el caso del *fat-tree*, al subir en los niveles, la sobresuscripción iría en aumento en la proporción que disminuye el número de switches por capa.

Tabla 9.13: Valores de sobresuscripción de las topologías en árbol en la capa borde

tree,K,L	Sobresuscripción
tree,8,1 (simple tree)	*S=8/1=8*
tree,8,2 (thin-tree)	*S=8/2=* 4
tree,8,3 (thin-tree)	*S=8/3= 2,6 = 3*
tree,8,4 (thin-tree)	*S=8/4= 2*
tree,8,5 (thin-tree)	*S=8/5= 1,6 = 2*
tree,8,6 (thin-tree)	*S=8/6= 1,33 = 2*
tree,8,7 (thin-tree)	*S=8/7= 1,14 = 2*
tree,8,8 (fat-tree)	*S=8/8= 1*

Como podemos ver, en el caso del *simple tree*, si 8 hosts de una misma subred quisieran intercambiar datos al mismo tiempo con algún nodo de otra red, tendrían un único enlace hacia arriba, y por lo tanto, tendrían que compartir ese único enlace entre los 8 hosts. Sin embargo, en el caso del *fat-tree*, si 8 host de una misma subred quisieran comunicarse al mismo tiempo con algún nodo de otra subred, cada host tendría un enlace reservado y no habría sobresuscripción. En el caso del *tree,8,5* por ejemplo, significa que tenemos 5 caminos(enlaces) hacia arriba para 8 hosts, por lo tanto, en el peor de los casos, cada enlace hacia arriba habrá que compartirlo entre 2 hosts. Para ser exactos, 3 enlaces distintos se compartirían entre 6 hosts y los 2 enlaces restantes, 1 para cada host restante.

Por lo tanto, el nivel de sobresuscripción significa, en el peor de los casos, si todos los nodos de una subred quieren comunicarse con nodos de otra subred al mismo tiempo, entre cuántos host hay que compartir los enlaces hacia arriba. Cuanto mayor sea el nivel de sobresuscripción, peor, ya que habrá que compartir el enlace hacia arriba entre más hosts. En cambio, cuanto menor sea el nivel de sobresuscripción, mejor, ya que habrá más alternativas para no tener que compartir el mismo enlace. En la práctica, para aprovecharla bien, es preciso un buen algoritmo de routing que reparta el tráfico.

El cálculo de la sobresuscripción en las topologías *Jellyfish*: Para calcular la sobresuscripción en las topologías Jellyfish necesitamos definir cuatro parámetros:

- N = número de switches
- K = número de enlaces hacia abajo (switch-to-host)
- L = número de enlaces switch-to-switch
- P = longitud media de los caminos (distancia media), teniendo en cuenta únicamente enlaces switch-to-switch
  - P disminuye si aumenta L (hay más alternativas para ir de un switch a otro)
  - P aumenta si aumenta N (hay más caminos entre los switches)

Para calcular la sobresuscripción tenemos en cuenta el “peor caso”:

- NK hosts (todos los hosts) intentan enviar tráfico a nodos no locales.
- Como cada camino tiene longitud media P, se necesita el uso de NKP enlaces.
- Únicamente contamos con NL enlaces.

Por lo tanto, el valor de la sobresuscripción es:

$$NKP/NL = KP/L$$

En comparación con los *trees*, en los que la sobresuscripción es K/L en la capa borde, las redes *Jellyfish* son algo peores. Pero no olvidemos que, salvo en los *fat-tree*, al ir subiendo de capa, la sobresuscripción también aumenta, puesto que tenemos menos switches (recurusos) para compartir el mismo número de enlaces.

Podemos decir que la sobresuscripción de las topologías *Jellyfish* es mayor que en los *thin-tree* equivalentes, pero esta comparación solo es válida en la capa borde, ya que en la capa superior o *core*, la sobresuscripción en el *thin-tree* no sería K/L, sino K^3/KL^2 (número de hosts / número de enlaces bajo la capa core) = K^2 / L^2.

Resumiendo, se puede decir que la sobresuscripción de las topologías *Jellyfish* es mayor que la sobresuscripción en la capa borde de los *thin-tree*.

$$KP/L \text{ (Jellyfish)} > K/L \text{ (Tree capa borde)}$$

Sin embargo, la sobresuscripción de las topologías *Jellyfish* es menor que la sobresuscripción en la capa core de los *thin-tree.*

$$KP/L \text{ (Jellyfish)} < K^2/L^2 \text{ (Tree capa core)}$$

10. Gestión y planificación del TFG

La gestión y planificación es algo indispensable a la hora de realizar un proyecto, en mi caso, el Trabajo de Fin de Grado. Para ello, conviene dividir todo el trabajo en distintas fases. En mi caso, las fases determinadas han sido las siguientes: Fase de gestión, fase de desarrollo, fase de documentación y fase de comunicación. En cada una de las fases se han definido los distintos paquetes de trabajo considerados. Los paquetes de trabajo son aquellos elementos finales que determinan el alcance del proyecto y son la base para la planificación del mismo. Cada paquete de trabajo estará compuesto por un conjunto de tareas a realizar que se describirán posteriormente.

Para mostrar los paquetes de trabajo en un diagrama según la fase a la que le corresponde, he construido un EDT (Estructura de Descomposición del Trabajo). Lo que se consigue mediante el EDT es descomponer de manera jerárquica todo el trabajo que se debe realizar para cumplir las expectativas y objetivos del TFG. En la imagen 10.1 se puede observar el EDT que determina los paquetes de trabajo correspondientes a cada fase propia del TFG.

10.1 Diagrama EDT

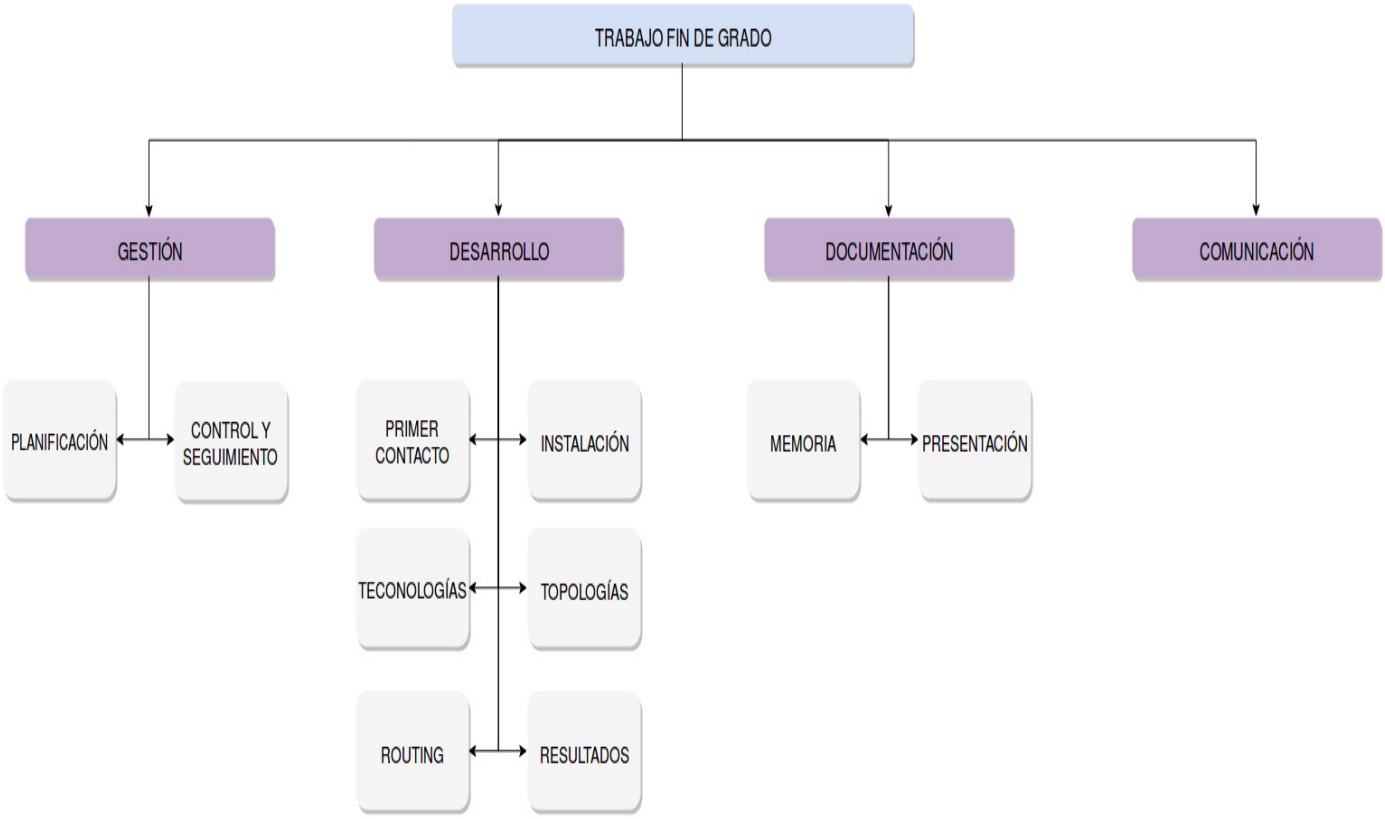

Figura 10.1: Estructura de Descomposición del Trabajo (Diagrama EDT)

CAMBIOS EN EL EDT 1

El proyecto ha sufrido algunos cambios en dos fases distintas. Estos cambios los considero muy normales en este tipo de trabajos debido a que en todo el periodo de realización del proyecto puede haber muchos desajustes y variaciones.

En la primera fase, para ser exactos, la tarea “Implementar routing proactivo con POX” no ha sido realizada, puesto que durante la realización del proyecto, se ha descubierto una manera distinta de llevar a cabo esa tarea, exactamente, mediante la herramienta RipLPox. Además, al no tener constancia de la herramienta RipL desde un principio, en la primera versión del EDT, las tareas “implementar topologías de datacenter con la herramienta RipL” e “Implementar routing proactivo con RipLPox” no habían sido creadas, puesto que desde un principio, estas dos tareas pretendían hacerse, por una parte, el controlador, únicamente mediante POX, y, por otra, la creación de las topologías, mediante scripts en Python, sin la ayuda de la herramienta RipL.

Resumiendo, en la segunda versión del EDT se han añadido dos tareas nuevas, para ser concretos, en los paquetes de trabajo “topología” y “routing”. Por otra parte, se ha eliminado una tarea correspondiente al paquete de trabajo “routing” respecto a la primera versión del EDT, por razones citadas previamente.

Estos cambios sufridos en los diagramas son totalmente naturales en los proyectos y no debe ser nada alarmante, todo lo contrario, el hecho de que un proyecto sufra cambios durante su realización es algo positivo. Además, tener que cambiar algo respecto a la primera planificación, significa que nos hemos dado cuenta de que existen nuevos caminos para cumplir los objetivos previamente establecidos. La estructura del diagrama EDT se ha mantenido igual, puesto que no se han creado paquetes de trabajo nuevos ni se han eliminado paquetes de trabajo ya existentes. Lo único que ha cambiado son algunas tareas que conforman cada uno de los paquetes, en este caso, los paquetes “topología” y “routing”.

CAMBIOS EN EL EDT 2

En la segunda fase, más o menos a falta de poco más de un mes de dar el visto bueno al TFG, después de haber invertido unas cuantas horas intentando implementar el routing con la herramienta RiplPox, me he dado cuenta de que debido a ciertos problemas, era muy difícil adaptar el routing que proponía la herramienta a mis topologías. Para afrontar ese problema, he decidido realizar el routing mediante la tarea que había planificado desde un principio, “Implementar routing proactivo con POX”, sin la ayuda de ningún complemento.

10.2 Descripcion de las tareas que componen los paquetes de trabajo

En este apartado se explicará en qué consiste cada tarea correspondiente a cada uno de los paquetes de trabajo definidos para este proyecto. Además, mediante unas tablas, se indicará cuál ha sido el tiempo estimado para cada una de las tareas antes de haber sido realizadas, así como el tiempo real que se ha empleado en ellas, es decir, el tiempo real invertido en cada una de las tareas. Con el tiempo estimado y real de cada tarea, se calculará la desviación correspondiente en el tiempo, y en algunos casos, se analizarán cuáles han sido los factores que han influido en tal desviación.

Las tareas han sido extraídas a partir de los paquetes de trabajo definidos en el EDT diseñado la primera semana de realización del proyecto, concretamente, en la fase de planificación. Cabe destacar que algunas de las tareas han ido apareciendo durante el desarrollo del proyecto, ya que desde un principio no se habían previsto.

10.2.1 Gestión

Esta fase se llevará a cabo durante toda la realización del proyecto, pero, sobre todo, en las primeras dos semanas, ya que un proyecto de estas dimensiones, antes de empezar a desarrollarlo, requiere hacer una planificación exhaustiva del mismo, y dividir la carga de trabajo de manera más equitativa posible. Las dos primeras tareas de este apartado corresponden al paquete de trabajo “planificación” y las dos últimas al paquete de trabajo “control y seguimiento”. Las tareas definidas en esta fase son las siguientes:

1. Establecer objetivos: En este apartado, se concretan de manera más específica posible los objetivos a cumplir en este proyecto. Definir bien estos objetivos me ayudará a tener lo más claro posible el trabajo a realizar en cada momento.

2. Planificar paquetes de trabajo: Identificar cada paquete de trabajo a realizar, y teniendo en cuenta el trabajo que nos pueda suponer cada uno de ellos, hacer una estimación del mismo. Asimismo, dividir cada uno de los paquetes de trabajo en unidades más pequeñas, es decir, en tareas.

3. Calcular y analizar desviaciones: El objetivo de este paquete será calcular las desviaciones entre el tiempo real y estimado de cada paquete. Se hará un análisis de los resultados más significativos.

4. Control del proyecto: Mediante este paquete, se tratará de llevar un control y seguimiento preciso del proyecto. Cada día que se dedica para desarrollar el proyecto, se documentará brevemente el trabajo realizado y las horas invertidas ese mismo día. De esta manera, sabremos el tiempo real dedicado cada día, semana y mes, por ejemplo. Por lo tanto, este paquete nos facilitará el trabajo a la hora de calcular y analizar las desviaciones.

10.2.2 Fase de desarrollo

Esta fase será la que más tiempo nos llevará en cuanto a horas invertidas, puesto que, en mi caso, requiere el aprendizaje de nuevas herramientas y tecnologías aplicadas en el área de *datacenter* que nunca antes había visto. La tarea número 1, corresponde al paquete de trabajo “primer contacto”, la tarea número 2, al paquete de trabajo “instalación”, las tareas 3,4,5 y 6 al paquete de trabajo “tecnologías”. Las tareas 7,8,9 y 10 corresponden al paquete de trabajo “topologías” y las tareas 11,12 y 13 corresponden al paquete de trabajo “routing”. Finalmente, la última tarea corresponde al paquete de trabajo “resultados. A continuación, se hace una descripción de cada una de las tareas a realizar en la fase de desarrollo.

1. Primer contacto con las herramientas, entorno de trabajo y tecnologías: A esta tarea también se le puede denominar como la introducción a la fase de desarrollo. Consiste en aprender cada una de las tecnologías que se utilizarán y el entorno de trabajo donde se trabajará. Una vez obtenida una visión general y haberme familiarizado con cada una de las tecnologías que se emplearán y el entorno donde se trabajará, intentar identificar qué relación existe entre los citados elementos.

2. Instalación de las herramientas a utilizar: Las instalaciones se harán en una máquina virtual que nos proporciona Mininet. Dicha máquina viene instalada con Mininet y POX. Además, habrá que instalar las herramientas RipL y RipLPox.

3. Estudiar las SDN: Estudiar de qué trata la tecnología SDN y cuál es su situación hoy en día. Además, analizar cuál ha sido el camino para llegar hasta las redes SDN.

4. Aprender Mininet: Aprender a manejarme en el entorno de simulación Mininet.

5. Estudiar controladores: Analizar qué controladores hay en el mercado, y aprender a utilizar en concreto el controlador POX, que será el que se utilizará en este proyecto.

6. Estudiar el protocolo OpenFlow: Estudiar el protocolo utilizado en la capa de control.

7. Estudiar topologías: Estudiar cómo se generan las topologías mediante el emulador Mininet.

8. Analizar topologías de datacenter: Una vez aprenda cómo se crean las topologías básicas en Mininet, investigar que topologías se utilizan en los datacenter.

9. Implementar con Mininet topologías utilizadas en datacenter: Una vez aprenda cómo se implementan las topologías y cuáles son las utilizadas en los datacenter, implementar dichas topologías.

10. Implementar topologías de datacenter con la herramienta RipL: Esta tarea ha sido creada durante la fase de desarrollo del proyecto, ya que, desde un principio, no tenía constancia de este complemento. En este apartado, tendré que aprender cómo se crean y se representan de manera gráfica las topologías mediante la herramienta RipL e implementar algunas.

11. Realizar la gestión de direcciones de los host: Decidir qué criterio se utilizará para la asignación de direcciones (MAC, IP, VLAN…). Esta tarea estará muy relacionada con el routing proactivo que se hará posteriormente.

12. Implementar routing proactivo con POX: Implementar el routing proactivo en topologías de datacenter creadas mediante el controlador POX.

13. Implementar routing proactivo con RipLPox: Esta tarea ha sido creado durante la fase de desarrollo del proyecto, ya que, desde un principio, no tenía constancia de esta herramienta. Con este complemento del controlador POX, tendré que aprender cómo se implementa el routing proactivo en topologías de datacenter e implementar un módulo para realizar el routing de las topologías creadas mediante RipL.

14. Analizar resultados: Evaluar las topologías en cuanto a funcionalidad, no en cuanto a rendimiento, al no trabajar con hardware real. Asimismo, hacer una comparativa entre las distintas topologías de datacenter que se han conseguido implementar.

10.2.3 Fase de documentación

Esta fase consiste en ir documentando el trabajo realizado durante todo el periodo de realización del proyecto. En mi caso, la información correspondiente a cada tema, la he ido documentando en diferentes ficheros, y, una vez el tutor haya dado su visto bueno, he redactado el documento final, dándole un estilo formal y profesional. La primera tarea corresponde al paquete de trabajo “memoria” mientras que las tareas 2 y 3 corresponden al paquete de trabajo “presentación”. A continuación, se describe cada una de las tareas a realizar en esta fase.

1. Documentar trabajo: Documentación de los ficheros correspondientes de cada uno de los temas estudiados y redacción formal del documento final.

2. Hacer presentación: Crear la presentación del trabajo realizado. Tener en cuenta que la duración de la misma deberá ser de unos 20-25 minutos.

3. Preparar presentación: Preparación de la presentación de cara al día de la exposición del trabajo.

10.2.4 Comunicación

El objetivo de esta fase será hacer un seguimiento del proyecto, desde el primer día que se inicia el trabajo hasta el último. La manera acordada para llevar a cabo dicho seguimiento será mediante reuniones periódicas y, en cada reunión, se redactará un acta. De esta manera, las dos únicas tareas pertenecientes al paquete de trabajo “comunicación” son las siguientes:

1. Reuniones periódicas: Con el fin de establecer la comunicación entre alumno y tutor, se realizarán reuniones, normalmente, semanales, de entre 30 y 45 minutos, básicamente para analizar en qué punto del proyecto nos encontramos y fijar nuevos objetivos a corto plazo. De esta manera, lo que se tratará de conseguir será equilibrar la carga de trabajo y hacer un seguimiento completo durante todo el periodo de realización del proyecto.

2. Actas: Para cada reunión establecida, de antemano, se recogerán los puntos que se quieran trabajar (cosas que se quieran comentar o problemas que hayan ido surgiendo) en cada una de las actas. Además, una vez realizada la reunión, se hará una síntesis de lo hablado. De esta manera, se conseguirán aclarar las dudas que vayan surgiendo, fijar nuevos objetivos a corto plazo y actualizar el punto en el que me encuentro en el proyecto. Resumiendo, es un manera de hacer una documentación y seguimiento de cada una de las reuniones.

10.3 Estimación y desviación de las tareas

Una vez explicadas las fases por las que transcurre el proyecto y descritas las tareas a realizar, en la tabla 10.1 se recoge información de cada una de las tareas (estimación de tiempo en horas y tiempo invertido en cada tarea, en horas), identificando a qué paquete de trabajo le corresponde.

Tabla 10.1: Estimación y tiempo real invertido de cada una de las tareas que componen los paquetes de trabajo

	Estimación (h)	Real (h)
TFG	312	349
Gestión	45	38
Planificación	23	18
Establecer objetivos	3	3
Planificar paquetes de trabajo	20	15
Control y seguimiento	22	20
Calcular y analizar desviaciones	10	10
Control del proyecto	12	10
Desarrollo	175	208
Primer contacto	10	8
Primer contacto con las herramientas, entorno de trabajo y tecnologías	10	8
Instalación	5	12
Instalación de las herramientas a utilizar	5	12
Tecnologías	65	58
Estudiar las SDN	10	8
Aprender Mininet	20	20
Estudiar controladores	15	15
Estudiar el protocolo OpenFlow	20	15
Topologías	55	70
Estudiar topologías	15	15
Analizar topologías de datacenter	10	15
Implementar con Mininet topologías utilizadas en datacenter	30	30
Implementar topologías de datacenter con la herramienta RipL	No se había previsto	10

Routing	25	50
Realizar la gestión de direcciones de los host	10	10
Implementar routing proactivo con POX	15	30
Implementar routing proactivo con RipLPox	No se había previsto	10
Resultados	15	10
Analizar resultados	15	10
Documentación	80	91
Memoria	60	75
Documentar trabajo	60	75
Presentación	20	16
Hacer presentación	10	6
Preparar presentación	10	10
Comunicación	12	12
Reuniones periódicas	9	9
Actas	3	3

En la tabla 10.2 se muestran las desviaciones en el tiempo de cada una de las tareas definidas, es decir, la diferencia en el tiempo entre el tiempo real y el tiempo estimado. Si la desviación es un número negativo significa que el tiempo real ha sido menor al estimado. En cambio, si la desviación se presenta con un número positivo, significa que el tiempo real ha sido mayor al estimado.

Tabla 10.2: desviación en horas de cada una de las tareas

	Desviación (Real - Estimación)
TFG	349 - 312 = 37
Gestión	38 - 45 = - 7
Planificación	18 - 23 = - 5
Establecer objetivos	3 - 3 = 0
Planificar paquetes de trabajo	15 - 20 = - 5
Control y seguimiento	20 - 22 = - 2
Calcular y analizar desviaciones	10 - 10 = 0
Control del proyecto	10 - 12 = - 2
Desarrollo	208 - 175 = 33
Primer contacto	8 - 10 = -2
Primer contacto con las herramientas, entorno de trabajo y tecnologías	8 - 10 = - 2
Instalación	12 - 5 = 7
Instalación de las herramientas a utilizar	12 - 5 = 7
Tecnologías	58 - 65 = - 7
Estudiar las SDN	8 - 10 = - 2
Aprender Mininet	20 - 20 = 0
Estudiar controladores	15 - 15 = 0
Estudiar el protocolo OpenFlow	15 - 20 = - 5
Topologías	70 - 55 = 15
Estudiar topologías	15 - 15 = 0
Analizar topologías de datacenter	15 - 10 = 5
Implementar con Mininet topologías utilizadas en datacenter	30 - 30 = 0
Implementar topologías de datacenter con la herramienta RipL	10 - 0 = 10
Routing	50 - 25 = 25
Realizar la gestión de direcciones de los host	10 - 10 = 0
Implementar routing proactivo con POX	30 - 15 = 15
Implementar routing proactivo con RipLPox	10 - 0 = 10
Resultados	10 - 15 = - 5
Analizar resultados	10 - 15 = - 5
Documentación	91 - 80 = 11
Memoria	75 - 60 = 15
Documentar trabajo	75 - 60 = 15
Presentación	16 - 20 = - 4
Hacer presentación	6 - 10 = - 4
Preparar presentación	10 - 10 = 0
Comunicación	12 - 12 = 0
Reuniones periódicas	9 – 9 = 0
Actas	3 - 3 = 0

10.3.1 Razones más significativas de las desviaciones

En este apartado se analizarán las desviaciones más remarcables, indicando cuál ha sido el principal factor que ha afectado en tal desviación.

- Instalación de las herramientas a utilizar: El hecho de haber conocido las herramientas RipL y RipLPox, han provocado una desviación considerable en la tarea “instalación de las herramientas a utilizar”. La instalación de las dos herramientas ha sido muy problemática, ya que he tenido que adaptar muchas partes del código para evitar incoherencias entre distintas versiones y así conseguir que funcionen correctamente el emulador Mininet y el controlador POX con RipL.

- Implementación del routing: Primeramente, la idea de implementar el routing estaba prevista hacerla simplemente mediante un módulo que se cargaría con el controlador POX. Durante la realización del proyecto, conocí dos herramientas orientadas a generar toplogías e implementar routing para topologías orientadas a datacenter, RipL y RipLPox. Decidí que la creación de las topologías y el routing lo haría mediante las dos herramientas. La primera de ellas, RipL, considero que ha sido útil para ayudarme a generar topologías. No obstante, he intentado adaptar el código de RipLPox para realizar mi routing propio, pero no lo conseguía, ya que había muchos problemas de incompatibilidad de versiones. Finalmente, el routing se ha realizado mediante un módulo que carga el controlador POX, como se había planificado desde un principio. Pero, el tiempo invertido en entender e intentar implementar el routing con RipLPox considero que ha sido bastante elevado. Creo que debería haberme dado cuenta antes de que ese camino no era el correcto. Pero sirvió para aprender cómo hacer el routing, al margen de RipL.

- Documentación final del trabajo: Durante la realización del proyecto, la documentación correspondiente a cada tema se ha ido guardando en la plataforma *drive*. Una vez se han corregido los diferentes documentos de *drive*, el tiempo invertido en generar el *odt* final ha sido bastante mayor al estimado. por una parte, por el hecho de tener que utilizar una plantilla para dar un toque profesional al trabajo, y por otra parte, por tener que indicar las todas las referencias tomadas de Internet, a la hora de insertar alguna imagen o contenido específico.

- Desviación general en el proyecto: Principalmente, este proyecto está orientado al área de investigación, y, por lo tanto, presenta un nivel de incertidumbre adicional que no lo había tenido muy en cuenta en la primera planificación. Esto ha supuesto cambios imprevisibles en el proyecto y han ralentizado el desarrollo del mismo. No obstante, considero que me he adaptado bien a los cambios que este TFG ha ido sugiriendo y creo que las desviaciones son las esperables en un TFG de esta naturaleza.

10.4 Plan de riesgo

En proyectos de estas dimensiones, incluso en proyectos de bastante menos duración, es imprescindible contar con un plan de riesgo, ya que, por una cosa u otra, el proyecto podría sufrir retrasos en varios aspectos, por factores, muchas veces, imprevisibles. Por lo tanto, para poder reaccionar ante retrasos y problemas que seguramente ocurrirán, propongo el siguiente plan de riesgo, que estará presente en todo momento en este proyecto.

Con el fín de evitar problemas a la hora de entregar este proyecto, se ha puesto un *colchón* a las fechas límite. En mi caso, la fecha límite para entregar el proyecto es el día 6 de septiembre. Teniendo en cuenta esa fecha, toda la planificación de este proyecto se ha hecho para que la memoria esté lista y supervisada para una fecha acordada con el director, día 20 de julio para ser exactos (4 días antes de la fecha en la que el estudiante solicita la defensa y el director emite el informe favorable a la defensa).

En algunos de los paquetes, al ser paquetes que requieren mucho tiempo invertido (>25 horas…) y hacer una estimación aproximada es bastante complicado, comparado con los paquetes que se realizan en tiempo más reducido, la estimación que se ha hecho ha sido un poquitín más alta de lo realmente estimado (por ejemplo, si se estiman 25-35 horas para realizar el paquete de trabajo, el número fijado han sido 35 horas, para así tener mayor margen de reacción). En consecuencia, la estimación que se ha hecho para las tareas correspondientes a dichos paquetes, también ha sido alta.

10.5 Metodología de trabajo

Para llevar a cabo proyectos de estas dimensiones, requiere definir una metodología de trabajo que se intentará cumplir desde un principio y que conviene ir acostumbrándose a ella para tomarse este trabajo como una tarea diaria y algo serio. En mi caso, he definido dos metodologías distintas a seguir, que se aplicarán según en qué momento del segundo cuatrimestre me encuentro.

La primera metodología corresponde al periodo del día de inicio del proyecto (19/02/2018) hasta el día que finalizan las asignaturas optativas de cuarto curso (en mi caso, 27/04/2018). Durante ese periodo, al tener clases por la tarde, y, por lo tanto, tener las tardes ocupadas, he decidido dedicarle al proyecto una media de 2 horas y media diarias por la mañana, de Lunes a Viernes.

Una vez transcurrido ese periodo, desde que terminan las clases de cuarto hasta que finaliza el proyecto (24/07/2018, he considerado el último día la fecha en la que el estudiante solicita la defensa y el director emite el informe favorable a la defensa), he trabajado con la segunda metodología. En este periodo, al tener tiempo todo el día única y exclusivamente para dedicarle al proyecto, he decidido dedicarle una media de 4 horas diarias entre la mañana y la tarde, de Lunes a Viernes.

Cabe destacar que, a veces, a causa de diferentes problemas, he llegado a dedicar más de 6 horas algunos días. Otros días en cambio, no he podido dedicarle ningún minuto. Pero, como he indicado en los dos anteriores párrafos, el tiempo indicado representa una media diaria. En general, creo que me he adaptado adecuadamente a las dos metodologías definidas para realizar el trabajo y me han ayudado mucho.

Con este plan de estudio he conseguido tomarme el Trabajo de Fin de Grado con seriedad. Al principio, me costó adaptarme porque muchas veces no sabía por dónde empezar a trabajar y tenía dudas de si realmente el tiempo invertido resultaría útil. Pero una vez comprendido todo lo que había que hacer en cada momento y seguir lo planificado, con trabajo diario y compromiso, considero que he completado un buen trabajo.

En conclusión, este plan de estudios me ha servido para tomarme este trabajo como un deber diario. Además, he conseguido repartir la carga de trabajo en muchos días de manera equitativa, evitando que las últimas semanas no se amontone demasiado el trabajo y no haya agobios.

11. Conclusiones y posibles mejoras

CONCLUSIONES

A lo largo de este proyecto, se ha intentado transmitir de qué tratan las Redes Definidas por Software, un concepto innovador en las redes actuales y que procede de años de pruebas e investigación. Es una tecnología que a día de hoy se encuentra en estado de desarrollo, pero que un futuro muy próximo, pretenden ser la solución a los problemas de las actuales estructuras de red, que van apareciendo debido al incremento en la demanda de recursos por la aparición de nuevas tecnologías.

Respecto a las conclusiones técnicas, se puede decir que se han cumplido de manera satisfactoria los objetivos establecidos de antemano, a pesar de tener que hacer hacer frente a algunos problemas, algo muy habitual en este tipo de trabajos. En la siguiente lista, se muestran algunos de los conceptos más importantes que se han aprendido en este TFG.

- Conceptos teóricos: De qué trata la tecnología **SDN**, el protocolo **OpenFlow**, el controlador **POX**, investigación de distintas **topologías** que se utilizan en los *datacenter*, tipos de **routing** para topologías de datacenter...

- Conceptos prácticos: Entorno de trabajo **Mininet**, profundizar en el lenguaje **Python,** implementación de diversas **estructuras de red** utilizadas en datacenter, implementación del **routing** proactivo con el controlador **POX** para las topologías generadas con **Mininet**, sistema de gestión de direcciones **IP**, análisis y comparación de los resultados obtenidos...

POSIBLES MEJORAS

En todo momento se ha trabajado en un entorno de trabajo emulado en una máquina virtual, y considero que es un buen punto de partida para todo aquel que tenga intención de comenzar a practicar con esta tecnología. Sin embargo, la capacidad de mi ordenador personal y el entorno de emulación Mininet cuenta con algunas limitaciones, como por ejemplo no poder generar topologías con un gran número de switches, enlaces y hosts, impidiendo poder replicar casos de datacenter reales. Para un futuro, sería interesante contar con un ordenador y un software de emulación más potente para hacer frente a estas limitaciones. Pero lo dicho, considero que Mininet es la herramienta ideal para aquellos que quieran dar los primeros pasos.

Asimismo, otras de las posibles mejoras interesantes que se podrían implementar en este TFG son las siguientes:

- Optimizar la distribución de la carga de las topologías estudiadas:

    - Topologías familia *tree*: Pensar en diferentes estrategias para distribuir la carga entre diferentes caminos. Estrategias más sofisticadas que el operador % utilizado en este TFG.

    - Topologías *Jellyfish:* En el caso del routing para la topología *Jellyfish,* basada en los caminos mínimos entre los distintos nodos, implementar la posibilidad de distribuir la carga del tráfico entre distintos caminos mínimos (*multipath routing)*. Por ejemplo, si para llegar de un nodo origen a un nodo destino existen tres caminos mínimos diferentes, distribuir el tráfico por esos tres caminos. En este proyecto, el tráfico se distribuye por el primer camino mínimo que se encuentra.

- Gráficos de las topologías: Utilizar alguna librería que permita dibujar gráficos de las topologías generadas a través de Mininet. De esta manera, podremos comprobar más fácilmente si la topología generada es la correcta o no, así como los cambios que pueda suponer la misma.

- Tolerancia a fallos: Implementar la tecnología *Hyperflow*. Replicar más controladores físicos en la red de control. Con esto conseguiremos que toda la inteligencia de la red no esté centralizada en un único punto, sino en varios. De este modo, si por cualquier razón, algún controlador fallase, la red no notaría ningún cambio y seguiría funcionando sin ningún problema.

- Descubrimiento automático de las topologías *Jellyfish:* De este modo, no será necesario pasarle un mapa que representa las conexiones switch-to-switch al controlador. El controlador mismo se encargaría de descubrir la topología aleatoria generada y haría el routing específico para cada caso.

12. Actas

Acta Reunión de Control y Seguimiento

Número de acta: 1 **Duración:** 1 hora

Fecha: 15/02/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Introducción al TFG.
- Acordar cómo se llevará a cabo el TFG.
- Establecer comunicación tutor-alumno.

Compromisos adquiridos:

- Investigar de qué trata la tecnología SDN, sin profundizar.
- Búsqueda de información sobre elementos que aparecerán en el proyecto (entorno de trabajo, controlador, protocolos…).

Acta Reunión de Control y Seguimiento

Número de acta: 2 **Duración:** 45 min

Fecha: 02/03/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Asegurar que se han identificado y entendido los elementos necesarios para llevar a cabo este proyecto.

- Analizar trabajos relacionados con topologías utilizadas en datacenter (researchgate.wordpress) y redes definidas por software.

Compromisos adquiridos:

- Practicar con el entorno de trabajo Mininet y el controlador POX.

Acta Reunión de Control y Seguimiento

Número de acta: 3 **Duración:** 90 min

Fecha: 14/03/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Arreglar problemas que han surgido con las topologías creadas.
- Analizar qué tipo de routing se emplea en topologías utilizadas en datacenter.
- Decidir con qué topologías se va a trabajar.
- Revisión de lo documentado hasta el momento.

Compromisos adquiridos:

- Comenzar con la implementación de las topologías acordadas.
- Adecuar la documentación según las notas tomadas.

Acta Reunión de Control y Seguimiento

Número de acta: 4 **Duración:** 60 min

Fecha: 26/03/2017 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Decidir en qué parámetros apoyarnos para analizar, comparar y estudiar las diferentes topologías.

- Configuración de la máquina virtual para llevar a cabo este proyecto.

Compromisos adquiridos:

- Para después de semana santa: Dominar la construcción de topologías con Mininet y plantear un sistema gestión de identificadores o direcciones, muy relacionado con el routing proactivo que se hará posteriormente.

Acta Reunión de Control y Seguimiento

Número de acta: 5 **Duración:** 45 min

Fecha: 11/04/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Enseñar las topologías que he conseguido construir y cómo las he conseguido construir.
- Propuesta de añadir el análisis de parámetros topológicos para las distintas estructuras de red.
- Arreglar errores generales que nos impiden seguir con la implementación y desarrollo.
- Revisión de lo documentado hasta el momento

Compromisos adquiridos:

- Decidir en base a qué criterio se hará la gestión de direcciones de cara al routing.
- Implementación de las distintas topologías mediante el lenguaje Python.
- Adecuar la documentación según las notas tomadas.

Acta Reunión de Control y Seguimiento

Número de acta: 6 **Duración:** 30 min

Fecha: 18/04/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Analizar características de las topologías tree, fat-tree y thin-tree.
- Analizar las distintas maneras que hay para hacer la asignación de direcciones.

Compromisos adquiridos:

- Estudiar las diferencias entre las topologías propuestas.
- Estudiar qué routing se hace en las topologías tree.

Acta Reunión de Control y Seguimiento

Número de acta: 7 **Duración:** 1 hora

Fecha: 24/04/2017 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Uso de las herramientas RipL y RipLPox para la creación simplificada de las topologías utilizadas en datacenter.
- Arreglar el problema del módulo openvswitch.
- Revisión de lo documentado hasta el momento.

Compromisos adquiridos:

- Instalar las nuevas herramientas acordadas.
- Implementar las topologías acordadas mediante las herramientas citadas y hacer el routing correspondiente.
- Adecuar la documentación según las notas tomadas.

Acta Reunión de Control y Seguimiento

Número de acta: 8 **Duración:** 150 min

Fecha: 26/04/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Estudiar el concepto de sobresuscripción.
- Mostrar las topologías conseguidas hasta el momento.
- Corrección del error de importación y actualizar cada elemento a su versión adecuada versión (POX, RipL, funciones utilizadas en Mininet…).

Compromisos adquiridos:

- Analizar características de la topología Jellyifish (creación topología, sistema de gestión de direcciones, routing…).

Acta Reunión de Control y Seguimiento

Número de acta: 9 **Duración:** 1 hora

Fecha: 06/05/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Revisión de lo documentado hasta el momento.
- Representación gráfica de las topologías creadas con la herramienta RipL.

Compromisos adquiridos:

- Implementar la opción para representación de manera gráfica las topologías creadas con la herramienta RipL.
- Adecuar la documentación según las notas tomadas.

Acta Reunión de Control y Seguimiento

Número de acta: 10 **Duración:** 1 hora

Fecha: 16/05/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Criterio a utilizar para las topologías de familia tree.
- Debatir sobre el uso real de la tecnología SDN hoy en día.
- Hablar sobre la capacidad de los switches convencionales y openflow.
- Problema y solución al protocolo ARP (inicializar las tablas ARP de los hosts emulados).

Compromisos adquiridos:

- Implementar las topologías de familia tree en una única clase, sin hacer especial uso de la herramienta RipL.
- Analizar la memoria local de los switches, convencionales FIB, openFlow tabla de flujos, para después estudiar el routing de las topologías.

Acta Reunión de Control y Seguimiento

Número de acta: 11 **Duración:** 1 hora

Fecha: 25/05/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Proponer controlador hecho a mano con la herramienta ovs-ofctl, configurando las tablas de flujo de los switches para hacer un routing eficiente, reduciendo las tablas de flujo.
- Comentar limitaciones de mininet.
- Comentar cómo hacer comunicaciones exteriores al centro de datos.
- Comentar la gestión de direcciones para las topologías generadas con RipL.
- Corregir el algoritmo diseñado para implementar todos los casos de la familia tree en una única clase.

Compromisos adquiridos:

- Implementar algoritmo familia *tree*.
- Implementar aplicación de control proactiva para los *tree*.

Acta Reunión de Control y Seguimiento

Número de acta: 12 **Duración:** 45 min

Fecha: 30/05/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Enseñar las topologías que lanzo mediante el programa RipL.
- Sintaxis de openflow para hacer routing proactivo.
- Entender las tablas de forwarding de los hosts y switches y el protocolo ARP.
- Prioridades de las entradas en la tabla de flujos.

Compromisos adquiridos:

- Implementar routing proactivo para la familia tree

Acta Reunión de Control y Seguimiento

Número de acta: 13 **Duración:** 1 hora

Fecha: 06/06/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Comunicación entre el controlador y los objetos que crea Mininet, basado en ‘dpids’.
- Routing proactivo realizado para las topologías de la familia tree.
- Limitaciones del routing realizado.

Compromisos adquiridos:

- Documentar criterios tomados para crear las entradas en las tablas de flujos de los switches.

Acta Reunión de Control y Seguimiento

Número de acta: 14 **Duración:** 30 min

Fecha: 11/06/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Dar por terminadas las topologías tree.
- Solución implementada para resolver el protocolo ARP.

Compromisos adquiridos:

- Implementación de la topología y routing de *Jellyfish.*

Acta Reunión de Control y Seguimiento

Número de acta: 15 **Duración:** 30 min

Fecha: 20/06/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Routing para la topología *Jellyfish* (asignación de puertos).
- Características y diferencias del *Jellyfish* respecto a los tree.
- Problema de la asignación del *dpid* a los switches.
- Hablar sobre el routing para casos aleatorios controlados por semillas.

Compromisos adquiridos:

- Routing para la topología *Jellyfish* (encontrar un algoritmo que encuentre caminos mínimos entre dos nodos).
- Dar por terminadas las topologías datacenter y documentarlas de manera adecuada.

Acta Reunión de Control y Seguimiento

Número de acta: 16 **Duración:** 30 min

Fecha: 27/06/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Routing *Jellyfish.*
- Cálculo de número de enlaces para topologías *Jellyfish.*

Compromisos adquiridos:

- Terminar routing switch-to-switch para la topología aleatoria *Jellyfish.*
- Redactar documento final de todo el proyecto.

Acta Reunión de Control y Seguimiento

Número de acta: 17 **Duración:** 30 min

Fecha: 04/07/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Mostrar los resultados.
- Hacer una valoración global del proyecto.
- Estilo documentar (índices, pie de imagenes/tablas, márgenes…).
- Cosas a tener en cuenta para hacer la presentación.

Compromisos adquiridos:

- Dar por terminado el documento final.
- Corregir correcciones de cara a la última versión.
- Hacer y preparar la presentación para la fecha 17-20 de septiembre.

Acta Reunión de Control y Seguimiento

Número de acta: 18 **Duración:** 30 min

Fecha: 13/07/2018 **Lugar:** Despacho tutor

Asistentes: Julen Pérez-Cortés (alumno) y José Miguel-Alonso (tutor)

Temas tratados:

- Comentar correcciones del documento final versión 1 y 2.

Compromisos adquiridos:

- Trámite final del TFG.

13. Anexos

13.1 Anexo A

El comando *mn* se utiliza para iniciar el emulador Mininet, y mediante las opciones y parámetros que nos ofrece, nos permite personalizar la estructura de red y el funcionamiento de la misma. En este apartado se recogen todas las opciones que puede tener el comando "*mn",* excepto aquellas que se han citado anteriormente. Como recordatorio, la estructura del comando es la siguiente:

sudo mn --[OPCIÓN] = [PARÁMETRO],[ARGUMENTOS] --[OPCIÓN_n] = [PARÁMETRO_n], [ARGUMENTOS] …

--help : Muestra en la terminal la lista de opciones que se ofrecen para lanzar el comando *mn*, con sus respectivos parámetros y argumentos.

--switch = [PARÁMETRO] : Permite invocar el tipo de switch, según el PARÁMETRO

- default : Switch *Open vSwitch* por defecto
- ivs : Switch Openflow que utiliza la tecnología *Indigo Virtual Switch*, requiere previa instalación
- lxbr : *Linux Bridge*, implementado en código abierto que soporta el protocolo *STP*. Para activar *STP*, debemos pasar como ARGUMENTO *stp=1*
- ovs : Switch *Open vSwitch* compatible con *OpenFlow*, es el mismo que se crea cuando escogemos el PARÁMETRO default
- ovsbr : Switch Ethernet que soporta *STP*. Para activar el protocolo *STP*, debemos pasar como ARGUMENTO *stp=1*
- ovsk : Switch *Open vSwitch* en modo kernel para cada switch
- user : Switch implementado con el protocolo *OpenFlow* que se ejecuta en un espacio externo al emulador *Mininet*

--host = [PARÁMETRO] : Limita el ancho de banda del procesador de un host virtual

- cfs : Planifica el uso de recursos según la política Fair Queueing
- rt : Planifica el uso de recursos en tiempo real (*real time)*

--link = [PARÁMETRO] : Configura la latencia, ancho de banda o porcentaje de pérdida de paquetes de los enlaces virtuales

- default : Configura los enlaces con ancho de banda, latencia y porcentaje de pérdida predeterminados por defecto
- tc : Permite parametrizar los tres valores mediante el ARGUMENTO que pasemos.
  - bw = [ancho_de_banda en Mb/s]
  - delay = [latencia en ms]
  - loss = [porcentaje_pérdida en porcentaje]
  - Ejemplo de uso: sudo mn --link=tc, bw=10, delay=10, loss=5

--clean : Limpia los registros de emulación y cierra el emulador. Tecleando "*-c*" conseguiremos el mismo efecto. Es recomendable ejecutar "*sudo mn -c*" cada vez que finalizamos con una emulación

--test = [PARÁMETRO] : Todos los parámetros posibles de esta opción tienen tres características en común, es decir, todas inician y finalizan la emulación, y además. imprimen el tiempo de ejecución transcurrido entre el inicio y el fín. La diferencia es que permiten testear la red emulada en base a los siguientes PARAMETROS:

- none : Comprueba si la topología se ha creado correctamente e Imprime el tiempo de ejecución empleado para la creación
- build : Inicia la emulación hasta que la virtualización esté operativa. Después, imprime el tiempo de ejecución empleado para la creación
- all "(pingall + iperf)" : Inicia la emulación y comprueba la conectividad entre los host, utilizando internamente el parámetro *pingall*. Además, mide el ancho de banda entre el primer host y el último, utilizando internamente el parámetro *iperf*
- iperf : Mide la velocidad de una única conexión TCP, concretamente, entre el primer host y el último. Internamente, se crean dos servidores TCP, uno en

cada host virtual, y, una vez que se hayan conectado, se intercambian paquetes entre ambos y muestra los resultados.

- pingpair : Comprueba la conectividad entre los dos primeros hosts

- iperfudp : Mide el máximo ancho de banda entre dos hosts basado en el protocolo UDP

- pingall : Comprueba la conectividad de todos los host con todos, *all-to-all*

--xterms : Abre una terminal para cada uno de los elementos de red emulados, ya sean hosts, switches o el controlador

--ipbase = [PARÁMETRO] : Define el conjunto de direcciones que se utilizará en la red emulada. Por defecto, las direcciones asignadas parten del conjunto 10.0.0.0/8. Para asignar la que mejor nos convenga podemos hacerlo de la siguiente manera:

- --ipbase = red/máscara
- ejemplo: sudo mn --ipbase = 192.168.212.0/24

--mac : Permite asignar las direcciones MAC de manera ordenada. Sino, Mininet hace la asignación de direcciones de manera aleatoria

- ejemplo: host1 = 00:00:00:00:00:01
  host2 = 00:00:00:00:00:02

--arp : Inicializa las tablas ARP de los hosts emulados y para permitir a los mismos realizar conexiones a través de direcciones IP

--verbosity = [PARÁMETRO] : Permite imprimir información interna de la emulación en distintos niveles según los distintos PARÁMETROS:

- info
- warning
- critical
- error
- debug
- output

--innamespace : Por defecto, los switches y el controlador se encuentran en el mismo espacio de nombre, en el espacio *root*. Esta opción permite separar los switches del espacio de nombre *root*

--listenport = [PARÁMETRO] : Permite modificar los puertos lógicos asignados a los switches. Por defecto, al switch *s1* se le asigna el puerto 6634 e incrementa una unidad por cada switch (s2 = 6635, s3 = 6636…)

--pre = [PARÁMETRO] : Ejecuta el script que se pasa como PARÁMETRO. Dicho script se carga antes de la comprobación de la red

--post = [PARÁMETRO] : Ejecuta el script que se pasa como PARÁMETRO. Dicho script se carga después de la comprobación de la red

--nat : Agrega el servicio NAT, permitiendo que se conecten elementos emulados en Mininet y el host anfitrión

--version : Imprime la versión actual de Mininet que se está utilizando

13.2 Anexo B

En esta sección se muestra la guía de comandos que ofrece Mininet durante la emulación, también llamados comando CLI. Una vez iniciada una emulación, la *prompt* que nos aparecerá será ***mininet>*** y a partir de ahí, podremos ingresar los comandos para obtener diferente información sobre por ejemplo los elementos de red emulados, ya sean switches, hosts o el controlador. Los comandos disponibles, ordenados alfabéticamente son los siguientes:

- EOF: Finaliza la emulación y cierra el programa

- dpctl [comando][argumentos]: Permite la monitorización y administración de cada uno de los switches OpenFlow emulados. Al tener muchas opciones, dichas opciones se recogen en el ANEXO C

- dump: Muestra información detallada de cada uno de los dispositivos que conforman la red

- exit: Finaliza la emulación y cierra el programa

- gterm[nodo1, …, nodo_n]: Abre una terminal *gnome* para cada *nodo* especificado. Alternativa del comando *xterm*

- help: Muestra los comandos disponibles y el uso de algunos
    - variación: help [comando] : Informa sobre la función del *comando* específico

- intfs: Muestra las interfaces que utiliza cada uno de los dispositivos

- iperf [host1] [host2] : Mide la velocidad de una única conexión TCP, concretamente, entre el *host1* y el *host2*
    - Si no especificamos los parámetros *host*, mide la velocidad de una única conexión TCP, concretamente, entre el primer host y el último

- iperfudp [bw] [host1] [host2]: Mide la velocidad entre el *host1* y el *host2*, indicando el ancho de banda, en *Mb/s*, basado en el protocolo UDP
    - Si no especificamos los *host*, mide la velocidad entre el primer y el último host, basado en el protocolo UDP

- link [nodo1] [nodo2] [up/down] : Habilita (*up*) o deshabilita (*down*) el enlace existente entre el *nodo1* y el *nodo2*

- links: Muestra los enlaces que se encuentran en estado operativo

- net: Muestra los enlaces virtuales de la red. Se indican qué puertos de qué dispositivos son usados para formar tales enlaces

- nodes: Muestra una lista de los nodos que conforman la red

- noecho [host] [cmd args]: Permite ejecutar comandos (*cmd args*) en el *host* como si estuviéramos en él, evitando el uso de xterm

- pingall: Comprueba la conectividad de todos los host con todos, *all-to-all*

- pingallfull: Comprueba la conectividad de todos los host con todos, *all-to-all*, mostrando el RTT mínimo, medio y máximo

- pingpair: Comprueba la conectividad entre los dos primeros hosts

- pingpairfull: Comprueba la conectividad entre los dos primeros hosts, mostrando el RTT mínimo, medio y máximo
- ports: muestra las interfaces y los puertos utilizados por los switches

- px: Ejecuta declaraciones de variables en el lenguaje Python, puediéndolas combinar con funciones de Mininet

- py [objeto,funcion()]: Evalúa expresiones en Python, especificando *objeto* y *funcion()*
    - Ejemplo*: mininet> py net.addHost("h4") #crea un nuevo host con el identificador "h4"*

- quit: Finaliza la emulación actual

- sh [cmd args]: Permite ejecutar comandos (*cmd args*) del sistema operativo Linux

- source [file]: Lee comandos Mininet desde el archivo *file*

- switch [switch] [start/stop] : Inicia (*start*) o detiene (*stop*) el funcionamiento del *switch* indicado

- time [comando] : Muestra en pantalla cuál ha sido el tiempo que ha necesitado el *comando* indicado para ejecutarse

- x [host] [cmd args]: Crea un túnel X11 al *host* específico

- xterm [nodo1, …, nodo_n]: Abre una terminal para cada *nodo* especificado. Es útil para ejecutar comandos como si estuviéramos en el mismo dispositivo. Obtenemos lo mismo mediante el comando *noecho*

13.3 Anexo C

La utilidad *dpctl* o *ovs-ofctl* es una herramienta para administrar (crear, modificar, eliminar...) las tablas de flujo de los switches OpenFlow emulados. Para administrar las reglas no hace falta un controlador, ya que está pensado para establecer entradas de flujo puntuales manualmente. En este apartado se muestran los comandos de administración aplicables a cada uno de los switches emulados en nuestra red. Esta herramienta se puede utilizar tanto mediante la prompt de Mininet "*comandos CLI"*, como mediante la terminal de Linux. La primera opción es más limitada, por lo tanto, la guía que se presenta está hecha para la segunda opción, es decir, para la línea de comandos Linux.

La estructura de los comandos de la herramienta *ovs-ofctl* es la siguiente:

ovs-ofctl [OPCIÓN] COMANDO [argumentos]

OPCIÓN: Este campo no es obligatorio y su fin es dar información y cambiar el comportamiento de los comandos por defecto. Este campo puede recibir los siguientes argumentos:

- --timeout=[SEGUNDOS], --verbose, --log-file, --help, --version y --strict

COMANDO: Cabe destacar que los comandos *("COMANDO")* que ofrece la herramienta *dpctl* están orientados a tres fines distintos. En primer lugar, los comandos orientados para modificar las tablas de flujos de los switches OpenFlow, en segundo lugar, los comandos orientados para mostrar información sobre los switches, y por último lugar, comandos orientados a la comprobación de conectividad entre el controlador y los switches OpenFlow.

argumentos: valor que reciben algunos comandos. En esta guía, solo se trabajará con el *argumento* "*flujo"*, que vendrá explicado posteriormente cuál es su sintaxis y su cometido.

COMANDOS ORIENTADOS PARA MODIFICAR TABLAS DE FLUJO

En este apartado se muestran los comandos que ofrece la herramienta *ovs-ofctl* para administrar tablas de flujo de los switches:

- add-flow [SWITCH] [flujo]: Añade nuevos flujos en la tabla de flujo del switch indicado. La sintaxis y las opciones del argumento *flujo* se detallan al final de este anexo.

- add-flows [SWITCH] [FICHERO] : Añade nuevos flujos en la tabla de flujo del switch indicado. Dichos flujos están definidos en el fichero *FICHERO* y cada línea del fichero corresponde a una entrada de flujo distinta. Tener en cuenta la ruta relativa o absoluta donde se encuentra el fichero.

- mod-flows [SWITCH] [flujo] : Modifica las acciones de los distintos flujos ya definidos en la tabla de flujo del switch indicado.

- del-flows [SWITCH] [flujo] : Elimina la entrada de flujo correspondiente al switch indicado. Si no se define explícitamente cuál es el flujo, se elimina completamente la tabla de flujo del switch indicado.

COMANDOS ORIENTADOS PARA MOSTRAR INFORMACIÓN DE LOS SWITCHES

- show [SWITCH] : Muestra en pantalla información OpenFlow del switch indicado

- dump-desc [SWITCH] : Muestra en pantalla la descripción del switch indicado

- dump-tables [SWITCH] : Muestra en pantalla estadísticas de las tablas de flujo del switch indicado

- dump-ports [SWITCH][PORT] : Muestra en pantalla estadísticas del puerto del switch indicado. El parámetro es opcional, si no se indica, muestra las estadísticas de todos los puertos del switch indicado

- dump-ports-desc [SWITCH] : Muestra en pantalla la descripción de los puertos del switch indicado

- dump-flows [SWITCH] : Muestra en pantalla todas las entradas de la tabla de flujo del switch indicado

- monitor [SWITCH] : Muestra en pantalla todos los mensajes OpenFlow recibidos en el switch indicado

COMANDOS ORIENTADOS A LA COMPROBACIÓN DE CONECTIVIDAD

- probe [TARGET] : Intercambio de paquetes entre el controlador y el objetivo (switch) especificado, con el fin de comprobar la comunicación OpenFlow entre los dos dispositivos

- ping [TARGET](n) : El controlador manda 10 paquetes de *n* bytes al objetivo indicado y mide el tiempo de respuesta

- benchmark [TARGET](n) [COUNT] : Envía *COUNT* paquetes de *n* bytes al objetivo especificado con el propósito de medir el ancho de banda (paquetes/s o B/s) del enlace existente entre el controlador y el objetivo

Para finalizar con esta guía, se detalla cual es la estructura del argumento *flujo,* indicado en la subsección "COMANDOS ORIENTADOS PARA MODIFICAR TABLAS DE FLUJO". La estructura del argumento *flujo* se diferencia en dos campos:

[REGLA], actions=[ACCIONES]

REGLA: Criterio para identificar el paquete o grupo de paquetes a los que se le aplicarán las acciones. Este campo se compone por las siguientes opciones:

- in_port=[PUERTO] : Procesa paquetes de un puerto físico, dicho puerto se identifica por un número entero

- dl_src=[MAC] : Procesa paquetes Ethernet cuyo origen es la dirección MAC indicada. El formato de la dirección MAC es la siguiente: XX:XX:XX:XX:XX:XX

- dl_dst=[MAC] : Procesa paquetes Ethernet cuyo destino es la dirección MAC indicada

- dl_type=[ETHERTYPE] : Procesa paquetes del protocolo Ethernet indicado. El protocolo se indica con un número entero entre 0 y 65535, inclusive. En el caso de direcciones IPv4, dl_type=0x0800

- nw_src=[IP/MÁSCARA] : Procesa paquetes cuyo origen es la dirección IPv4 especificada. Indicar la máscara es opcional

- nw_dst=[IP/MÁSCARA] : Procesa paquetes cuyo destino es la dirección IPv4 especificada. Indicar la máscara es opcional

- nw_proto=[PROTOCOLO] : Procesa paquetes correspondientes al protocolo indicado. El protocolo se indica con un número entero entre 0 y 255, inclusive. En el caso del protocolo TCP, nw_proto=6

- tp_src=[PUERTO] : Procesa paquetes según el puerto UDP o TCP origen indicado. El puerto se indica con un número entre 0 y 65535, inclusive

- tp_dst=[PUERTO] : Procesa paquetes según el puerto UDP o TCP destino indicado. El puerto se indica con un número entre 0 y 65535, inclusive

ACCIONES: Indica las operaciones que se le aplicarán a cada entrada de la tabla de flujos. Este segundo campo se compone por las siguiente opciones:

- output:[PUERTO] : Reenvía los paquetes por el PUERTO físico indicado

- normal : Procesa el paquete como si estuviera en un dispositivo de capa 1 y 2. Para ello, el switch debe ser compatible con funcionalidades de capa 1 y 2

- flood: Reenvía el paquete por todos los puertos excepto por el puerto que se haya recibido el paquete y los puertos con *flooding* deshabilitado. Comportamiento *hub*

- all: Reenvía el paquete por todos los puertos excepto por el puerto que se haya recibido el paquete. Comportamiento *hub*

- controller:[MAX_LENGTH]: El paquete es enviado al controlador como un mensaje openflow *paquet_in*. Es opcional añadir el tamaño del paquete. Indicaremos "*ALL"* en caso de que se quiera enviar el paquete completo o "*número_de_Bytes",* representado por un número de tipo entero, si se desea enviar un tamaño específico del paquete

- local: Reenvía el paquete por el puerto establecido como *local*

- mod_dl_src:[MAC]: Modifica la dirección física origen del paquete. El formato de la dirección MAC es la siguiente: XX:XX:XX:XX:XX:XX

- mod_dl_dst:[MAC]: Modifica la dirección física destino del paquete. El formato de la dirección MAC es la siguiente: XX:XX:XX:XX:XX:XX

Para entender el funcionamiento de esta herramienta, dos ejemplos:

Ejemplo 1: Tenemos un switch OpenFlow de 3 puertos, llamado *s1 (switch 1)*. Queremos que toda aquella información entrante del puerto físico número 1 se redirija por el puerto físico número 3. El comando correspondiente para que el switch se comporte como tal sería el siguiente:

ovs-ofctl add-flow s1 in_port=1,actions=outport:3

```
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
root@mininet-vm:/home/mininet# ovs-ofctl add-flow s1 in_port=1,actions=output:3
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=79.096s, table=0, n_packets=0, n_bytes=0, idle_age=79, in_
port=1 actions=output:3
```

Ejemplo 2: Eliminar todas las entradas correspondientes a paquetes entrantes por el puerto físico número 1 del switch OpenFlow *s1:*

ovs-ofctl del-flows s1 in_port=1

```
root@mininet-vm:/home/mininet# ovs-ofctl del-flows s1 in_port=1
root@mininet-vm:/home/mininet# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
```

13.4 Anexo D

En esta sección se adjuntan los dos ficheros a los cuales se les hace referencia en el apartado “Mininet”. Por una parte, el código generado automáticamente por Miniedit (*hipercubo.py*), en este caso, un hipercubo. Por otra parte, el código implementado en Python para la creación de una topología *fat-tree* y su correspondiente gestión de direcciones (*FatTree_IP.py*).

hipercubo.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():


    # personaliza las caracteristicas de la topologia con las que se le
llamara a Mininet
    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')


    # anadir el controlador, al igual que se hubiera hecho con la opcion
--controller=remote, pero en codigo Python
    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',
                         controller=RemoteController,
                         protocol='tcp',
                         port=6633)


    # anadir los switches de tipo ovsk (Open vSwitch), al igual que se hubiera
hecho con la opcion --switch=ovsk, pero en codigo Python
    # dar un identificador a cada uno de los switches generados
    info( '*** Add switches\n')
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
    s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
```

```
    s8 = net.addSwitch('s8', cls=OVSKernelSwitch)


    # anadir los host, dandoles un identificador y su correspondiente
direccion IP segun la direccion IP base establecida anteriormente. En este
caso, no se ha indicado cual es su gateway
    info( '*** Add hosts\n')
    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
    h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
    h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
    h7 = net.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
    h8 = net.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)


    # anadir los enlaces virtuales entre los switches y los host
    info( '*** Add links\n')
    net.addLink(s1, s2)
    net.addLink(s1, s3)
    net.addLink(s3, s4)
    net.addLink(s4, s2)
    net.addLink(s7, s5)
    net.addLink(s5, s6)
    net.addLink(s6, s8)
    net.addLink(s8, s7)
    net.addLink(s1, s5)
    net.addLink(s2, s6)
    net.addLink(s3, s7)
    net.addLink(s4, s8)
    net.addLink(h1, s1)
    net.addLink(s2, h2)
    net.addLink(s5, h5)
    net.addLink(s6, h6)
    net.addLink(h7, s7)
    net.addLink(s8, h8)
    net.addLink(s3, h3)
    net.addLink(s4, h4)


    # iniciar la emulacion y el controlador
    info( '*** Starting network\n')
    net.build()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
       controller.start()


    # inicializar los switches y conectarlos con el controlador para formar la
red de control
    info( '*** Starting switches\n')
    net.get('s1').start([c0])
    net.get('s2').start([c0])
    net.get('s3').start([c0])
```

```
    net.get('s4').start([c0])
    net.get('s5').start([c0])
    net.get('s6').start([c0])
    net.get('s7').start([c0])
    net.get('s8').start([c0])


    info( '*** Post configure switches and hosts\n')


    CLI(net)
    net.stop()


if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

FatTree_IP.py

```
# mediante este script, cambiando el valor del parametro k, genera un fat-tree
de 3 capas (core, agregacion, borde) y k ramas, compuesto por switches de
grado k.


# nomenclatura de los elementos de red creados una vez se genera la topologia:


# switches core: cs_X
# switches agregacion: as_X_Y
# switches borde: es_X_Y
# hosts: h_X_Y_Z


# donde: X es la rama a la que le corresponde
#     Y es el numero de switch correspondiente a la rama X
#     Z es el numero de host, correspondiente al switch Y de la rama X


from mininet.node import OVSSwitch
from mininet.topo import Topo
class FatTree( Topo ): # definir la clase


    def __init__( self ):


        # caracteristicas de la topologia
        K = 4                           # grado del switch (num_puertos hacia
abajo + num_puertos hacia arriba)
        podNum = K                      # numero de ramas del fat-tree
        coreSwitchNum = pow((K/2),2)    # numero switches core (cs)
        aggrSwitchNum = ((K/2)*K)       # numero switches agregacion (as)
        edgeSwitchNum = ((K/2)*K)       # numero switches borde (es)
        hostNum = (K*pow((K/2),2))      # numero de hosts (h)
```

```
        # inicializar topologia
        Topo.__init__( self )


    # definir arrays para guardar los objetos (switches) que se van creando,
para luego poder hacer los enlaces virtuales correspondientes
        coreSwitches = []
        aggrSwitches = []
        edgeSwitches = []


    # definir array para ir guardando los host creados para posteriormente
anadirles la mascara e IP y hacer los enlaces virtuales correspondientes
    hosts         = []


    z=2 # variable que llevara el control de la direccion IP que se le asigna a
cada host de la subred, 192.168.1.%d/24 %z en este caso. Esta variable se
inicializa con 2 porque el 1 siempre se reserva para el gateway

    # creacion de los switches core e insertarlos en el array correspondiente
        for core in range(0, coreSwitchNum):
            coreSwitches.append(self.addSwitch("cs_"+str(core)))


        # por cada rama definida, loop (depende del valor del parametro K)
        for pod in range(0, podNum):


            # crear switch agregacion e insertarlos en el array correspondiente
            for aggr in range(0, aggrSwitchNum/podNum):
                aggrThis = self.addSwitch("as_"+str(pod)+"_"+str(aggr))
                aggrSwitches.append(aggrThis)


       # crear enlaces virtuales entre los switches agregacion y core
                for x in range((K/2)*aggr, (K/2)*(aggr+1)):
                    self.addLink(aggrThis, coreSwitches[x])


            # creacion switches borde e insertarlos en el array correspondiente
            for edge in range(0, edgeSwitchNum/podNum):
                edgeThis = self.addSwitch("es_"+str(pod)+"_"+str(edge))
                edgeSwitches.append(edgeThis)


       # crear enlaces virtuales entre switches borde y agregacion
                for x in range((edgeSwitchNum/podNum)*pod,
((edgeSwitchNum/podNum)*(pod+1))):
                    self.addLink(edgeThis, aggrSwitches[x])


           # creacion de hosts y enlaces virtuales entre los hosts y switches
borde
                for x in range(0, (hostNum/podNum/(edgeSwitchNum/podNum))):
                  self.addLink(edgeThis, self.addHost("h_"+str(pod)
+"_"+str(edge)+"_"+str(x)))
```

```
        # guardar los hosts en su correspondiente array para poder anadirles
una direccion IP y mascara
        hosts.append(self.addHost("h_"+str(pod)+"_"+str(edge)+"_"+str(x),
ip='192.168.1.%d/24' % z))
        z=z+1 # empezando por 192.168.1.2/24 para el primer host creado, por
cada host creado, se asignara la siguiente IP (+1) a la anterior, porque asi
se ha decidido hacer la gestion de direcciones para este ejemplo


        # ejemplo
        # host h_0_0_0 = 192.168.1.2/24
        # host h_0_0_1 = 192.168.1.3/24
        # host h_0_1_0 = 192.168.1.4/24
        # host h_0_1_1 = 192.168.1.5/24


#darle un nombre a la topología para que luego pueda ser llamada desde fuera
con la opcion --topo
topos = { 'fattree': ( lambda: FatTree() ) }
```

13.5 Anexo E

En este anexo, se indica el código correspondiente para generar las topologías e implementar el routing correspondiente de las topologias de datacenter que se han estudiado en este TFG. Por una parte, se indica el código correspondiente a las clases que generan topologías de familia *tree* (*TreeTopo) y Jellyfish* (*JellyFishTopo*) que se encuentran en el fichero *dctopo.py*.

Por otra parte, se indican los dos modulos que carga el controlador POX para implementar el routing tanto para las topologías *tree (tree_controller_julen.py)* como para la topología *Jellyfish (jelly_controller_julen.py).*

Class TreeTopo

```
class TreeTopo( Topo ): # clase que se le llama desde el fichero mn.py con
parametros tree,K,L


       def __init__(self, K, L):


          Topo.__init__(self, K, L)

          self.K = K # puertos hacia abajo switch
          self.L = L # puertos hacia arriba switch

          # caracteristicas de la topologia
          # switches core = L * L
          # switches agregacion = K * L
          # switches borde = K * K
          # hosts = K * K * K


          # definir los arrays donde se van a ir guardando los objetos para
despues hacer los enlaces correspondientes
          hosts = []
          switchesBorde = []
          switchesAgregacion =[]
          switchesCore = []


          # crear switches borde y hosts y sus correspondientes enlaces
          for p in range (0, K):
             for e in range (0, K):
                 dpid = '00000300000000' + str(p) + str(e) # asignacion dpid
                 borde = self.addSwitch("Bor_"+str(p)+"_"+str(e)+"_"+str(1),
dpid=dpid)
```

```
                switchesBorde.append(borde)
                for h in range (2, K + 2):
                    host = self.addHost("h_"+str(p)+"_"+str(e)
+"_"+str(h),ip='10.{}.{}.{}/8'.format(p,e,h), defaultRoute=None) # al primer
host se le da la direccion .2
                    hosts.append(host)
                    self.addLink(host, borde)


        # crear switches agregacion
        for x in range (0, K * L):
           dpid = '000002000000' + str(x).zfill(2) + '00' # asignacion dpid
           switchesAgregacion.append(self.addSwitch("Agr_"+str(x),
dpid=dpid))


        # enlaces switches borde-agregacion
        for x in range (0, K * K):
           for y in range (0, K * L):
               if math.floor(x / K) == math.floor(y / L):
                   self.addLink(switchesBorde[x], switchesAgregacion[y])


        # crear switches core
        for x in range (0, L * L):
           dpid = '0000010000' + str(x).zfill(2) + '0000' # asignacion dpid
           switchesCore.append(self.addSwitch("Core_"+str(x), dpid=dpid))


        # enlaces switches core-agregacion
        for x in range (0, L * L):
           for y in range (0, K * L):
               if x % L == y % L:
                   self.addLink(switchesCore[x], switchesAgregacion[y])
```

Class JellyFishTopo

```
class JellyFishTopo( Topo ):  # clase que se le llama desde el fichero mn.py
con parametros jf,numSwitches,K,L para generar una topologia Jellyfish


    def __init__( self, numSwitches, K, L ):

   semilla = 27
```

```
   random.seed(semilla) # ajustar la semilla para poder replicar experimentos
y poder hacer el routing para casos aleatorios controlados

   Topo.__init__( self, numSwitches, K, L )

   # caracteristicas de la topologia
   self.numSwitches = numSwitches # numero de switches
   self.K = K # numero de puertos hacia abajo (switch-to-host)
   self.L = L # numero de puertos aleatorios (switch-to-switch)

   # definir el array donde se van a ir guardando los switches. Segun se vaya
completando el grado de un switch, ese switch se saca de la lista, para que el
mismo no pueda ser involucrado en un nuevo enlace
   switches = []

   # crear switches y hosts y sus correspondientes enlaces
   for x in range (0, numSwitches):
      dpid = '00000100000000'+ str(x).zfill(2)
      switch = self.addSwitch("S_" + str(x), dpid=dpid)
      switches.append(switch)
      for h in range (2, K + 2):
          host = self.addHost("h_" + str(x) + "_" + str(h),ip='10.0.{}.
{}/16'.format(x,h), defaultRoute=None) # al primer host se ha decidido darle
la direccion .2
          self.addLink(host, switch)

   enlaces_guardados = [] # lista donde se guarda la pareja de switches
switch-switch2, representando un enlace entre los mismos

   while len(switches) > 1 and no_conectado(self, switches,
enlaces_guardados): # Mientras quede mas de un elemento en el array de
switches y los switches no esten conectados, loop
      switch = random.choice(switches) # coge un objeto switch al azar del
array switches[]
      switch2 = random.choice(switches) # coge un objeto switch al azar del
array switches[]

      if switch == switch2: # comprobar si ha salido el mismo objeto
          continue
      enlace = (switch, switch2) # si no ha salido, guardar el enlace
```

```
      if enlace in enlaces_guardados or (switch2, switch) in
enlaces_guardados: # comprobar si la pareja bidireccional switch-switch2
existe en el array de enlaces guardados
          continue
      enlaces_guardados.append(enlace) # si esa pareja bidireccional no
existe, guardar el enlace que representa pareja switch-switch2 en el array de
enlaces guardados


      if (apariciones_switch(self, switch, enlaces_guardados) == L or
switch_conectado(self, switch, enlaces_guardados, switches)): # mirar si el
switch switch aparece L (max. puerto aleatorios) veces en la lista de enlaces
guardados o esta totalmente conectado
          switches.remove(switch) # eliminar switch switch de la lista
switches[], ya que el switch ha quedado completo y no queremos que se vea
implicado en un nuevo posible enlace


      if (apariciones_switch(self, switch2, enlaces_guardados) == L or
switch_conectado(self, switch2, enlaces_guardados, switches)): # mirar si el
switch switch2 aparece L (max. puerto aleatorios) veces en la lista de enlaces
guardados o esta totalmente conectado
          switches.remove(switch2) # eliminar switch switch2 de la lista
switches[], ya que el switch ha quedado completo y no queremos que se vea
implicado en un nuevo posible enlace


   while True: # proceso para unir algun nodo que haya podido quedar suelto.
La idea es, quitando un enlace random, enlazar el nodo que ha quedado fuera, a
los dos nodos donde se ha eliminado el enlace random.
      completado = True
      for switch in switches:
          if switch_conectado(self, switch, enlaces_guardados, switches): # si
esta totalmente conectado, no hacer nada
              continue
          if self.apariciones_switch(self, enlaces_guardados) < (L-1): # si la
cantidad de puertos switch-to-switch del switch no ha alcanzado el parametro
L, hemos encontrado el switch
              completado = False
              enlace = random.choice(enlaces_guardados) # escoger un enlace
aleatorio
              if switch == enlace[0] or switch == enlace[1]: # si algun nodo
implicado en ese enlace coincide con el nodo cuyo grado no esta completado
(nodo que ha quedado suelto), no hacer nada
                  continue
              nuevo_enlace = (switch, enlace[0]) # si no ha coincidido,
guardar nuevo enlace
```

```
                nuevo_enlace2 = (switch, enlace[1]) # si no ha coincidido,
guardar nuevo enlace
                enlaces_guardados.remove(enlace) # eliminar enlace que ha tocado
al azar
                enlaces_guardados.append(nuevo_enlace) # anadir nuevo enlace al
nodo cuyo grado no esta completado
                enlaces_guardados.append(nuevo_enlace2) # anadir nuevo enlace al
nodo cuyo grado no esta completado
        if completado:
            break


    for enlace in enlaces_guardados: # una vez tengamos la lista de enlaces
guardados switch-switch2, crear los enlaces bidireccionales. enlace[0] es
switch y enlace[1] switch2
        self.addLink(enlace[0], enlace[1]) # construir enlaces entre parejas
switch-switch2


    mapa_topologia = enlaces_guardados # mapa que le pasaremos al controlador
que representa las conexiones aleatorias generadas entre los switches y asi
poder hacer un routing para ese caso concreto
    print "Lista de enlaces guardados generada que pasaremos al controlador:"
    print mapa_topologia # necesario para despues que despues el controlador
conozca la red aleatoria generada


# funciones utilizadas
def no_conectado(self, lista_switches, lista_enlaces): # comprueba si algun
enlace bidireccional entre switches no esta en la lista de enlaces guardados
    for i in range(len(lista_switches)):
        switch = lista_switches[i]
        for j in range(i + 1, len(lista_switches)):
            switch2 = lista_switches[j]
            enlace = (switch, switch2)
            if enlace not in lista_enlaces and (switch2, switch) not in
lista_enlaces:
                return True
    return False



def apariciones_switch(self, switch, lista_enlaces): # cuenta las apariciones
de un switch en la lista donde se guardan las parejas de enlaces. Necesario
para saber si un switch aparece L veces y por lo tanto, controlar si esta
completo o no
    count = 0
```

```
   for enlace in lista_enlaces:
      if switch in enlace:
          count += 1
   return count



def switch_conectado(self, switch, lista_enlaces, lista_switches): # dado un
switch, una lista de enlaces y una lista de switches, comprueba si el switch
esta totalmente conectado
   for s in lista_switches:
      if s == switch: # si es el mismo, no hacer nada
          continue
      if (switch, s) not in lista_enlaces or (s, switch) not in lista_enlaces:
          return False
   return True
```

tree_controller_julen.py

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.util import dpidToStr
import time
import math


log = core.getLogger()


# ajustar los parametros K y L segun la topologia tree que se lanza
K =
L =


class ProactiveTree (EventMixin):


 def __init__ (self,connection):
    self.connection = connection # devuelve como string la MAC del switch
OpenFlow en el siguiente formato: [XX-XX-XX-XX-XX-XX_numb], los corchetes y el
numb es algo que ignoraremos. Numb simplemente representa el orden de que se
han conectado los switches con el controlador. Partiremos desde aqui para
saber con quien nos comunicamos.
    self.listenTo(connection)


    self.ids1 = str(connection).split("-")[0]
    self.ids1 = self.ids1.split("[")[1]
    self.idint1 = int(self.ids1) # nos dice la capa que corresponde un switch.
Digitos 1 y 2 de la MAC
```

```
    self.ids2 = str(connection).split("-")[5]
    self.ids2 = self.ids2.split(" ")[0] # Digitos 11 y 12 de la MAC


    # identificador en dos partes del switch borde
    self.ids3 = self.ids2[0] # Digito 11 de la MAC
    self.idint3 = int(self.ids3) # rama que corresponde un switch
    self.ids4 = self.ids2[1] # Digito 12 de la MAC
    self.idint4 = int(self.ids4) # numero de switch de acceso correspondiente
a la rama idint3


    # identificador del switch de agregacion
    self.ids5 = str(connection).split("-")[4] # Digitos 9 y 10 de la MAC
    self.idint5 = int(self.ids5) # numero de switch de agregacion


    match1 = of.ofp_match() # emparejamiento basado en direcciones IP destino,
para distribuir trafico hacia abajo
    match1.dl_type = 0x0800 # es necesario indicar con que tipo de direcciones
se hace hacer el emparejamiento. En este caso, direcciones IPv4 (0x0800)


    match2 = of.ofp_match() # emparejamiento basado en puertos de entrada para
distribuir trafico hacia arriba


       if(self.idint1 == 03): # reglas para los switches de capa borde

             # trafico hacia arriba (prioridad 1)
           for x in range (1, K + 1): # para cada puerto hacia abajo
              puerto_hacia_abajo = x % L # hacer grupos de L con los puertos
hacia abajo
              match2.in_port = x # emparejar con el puerto hacia abajo
              puerto_saliente = puerto_hacia_abajo + K + 1 # accion reenvio
puerto hacia arriba
             self.install(puerto_saliente, match2, 1) # instalar regla trafico
hacia arriba


          # trafico hacia abajo (prioridad 2)
        for h in range (2, K + 2): # entrada para llegar a cada host conectado
a cada switch de acceso, para cada puerto hacia abajo
              rama = self.idint3 # nos indica la rama que corresponde
              borde = self.idint4 # nos indica el numero de switch que
corresponde a la rama
              host = h # indica el numero de host que corresponde, empiezando
desde .2
              match1.nw_dst = "10." + str(rama) + "." + str(borde) + "." +
str(host) # emparejar con la direccion IP destino
              self.install(h - 1, match1, 2) # instalar regla trafico hacia
abajo


    if(self.idint1 == 01): # reglas para los switches de capa core
```

```
       # trafico hacia abajo (prioridad 2)
         for k in range (0, K): # para cada puerto hacia abajo
               mascara1 = 24 - math.ceil(math.log(K,2)) # ajustar el valor de
la mascara en funcion del parametro 'K', que determina cuantas subredes hay
               mascara2 = str(mascara1) # math.ceil devuelve float
(mascara.0), para coger la parte 'mascara' pasar a string y obtener la primera
parte
              mascara = mascara2.split(".")[0]
              match1.nw_dst = "10." + str(k) + ".0.0/" + str(mascara) #
emparejar con la direccion IP destino
              self.install(k + 1, match1, 2) # instalar regla trafico hacia
abajo, prioridad 2



    if(self.idint1 == 02): # reglas para los switches de capa agregacion

           grup = self.idint5
          grupo =  math.floor(grup / L) # calcular el grupo al que corresponde
el switch
          grupo = str(grupo) # math.floor devuelve float (grupo.0), para coger
la parte 'grupo' pasar a string y obtener la primera parte
          grupo = grupo.split(".")[0]
          # trafico hacia abajo (prioridad 2)
          for k in range (0, K): # para cada puerto hacia abajo
              match1.nw_dst = "10." + str(grupo) + "." + str(k) + ".0/24" #
emparejar con la direccion IP destino
              self.install(k + 1, match1, 2) # instalar regla trafico hacia
abajo


          # trafico hacia arriba, (prioridad 1)
          for x in range (1, K + 1):
              puerto_hacia_abajo = x % L # hacer grupos de L con los puertos
hacia abajo
              match2.in_port = x # emparejar con el puerto hacia abajo
              puerto_saliente = puerto_hacia_abajo + K + 1 # accion reenvio
puerto hacia arriba
             self.install(puerto_saliente, match2, 1) # instalar regla trafico
hacia arriba


 def install(self, port, match, priority): # funcion que se ejecuta cada vez
que se instala una nueva entrada en algun switch. Como parametros recibe el
puerto port, puerto por el que se reenvian los paquetes que coincidan, el
emparejamiento match, la direccion IP o conjunto de direcciones IP (trafico
hacia abajo) o puertos entrantes (trafico hacia arriba) para emparejar y la
prioridad de los flujos priority
    msg = of.ofp_flow_mod() # prepara el paquete msg para que posteriormente
se instale como una nueva entrada en la tabla de flujos del switch OpenFlow
correspondiente
    msg.match = match # anade el matching que pasamos como parametro a ese
mensaje msg. En este caso, el matching se basa en direcciones IP destino o
```

```
puertos de entrada, por lo tanto, los posibles campos seran nw_dst o in_port,
respectivamente
    msg.priority = priority # anade la prioridad que pasamos como parametro a
esa nueva entrada
    msg.actions.append(of.ofp_action_output(port = port)) # anade la accion a
realizar con las direcciones IP o puertos entrantes que coincidan con la
entrada para reenviarla por el puerto que pasamos como parametro
    self.connection.send(msg) # el controlador envia el mensaje msg al switch
para instalar el flujo en el switch OpenFlow


 def _handle_PacketIn (self, event): # funcion que se encarga de procesar los
paquetes a los cuales un switch no les ha encontrado coincidencia. Posibilita
que el controlador reciba esos paquetes de tipo packet_in, aunque no se vaya a
hacer nada con ellos, ya que todos los switches se han instruido de manera
correcta de antemano. De lo contrario, saltarían errores, puesto que no se les
daria ningun tratamiento a los paquetes que no se les ha encontrado
coincidencia.
    packet = event.parse()
    msg = of.ofp_packet_out()
    msg.buffer_id = event.ofp.buffer_id
    msg.in_port = event.port



class proactive_tree (EventMixin):


 def __init__(self):
    self.listenTo(core.openflow)


 def _handle_ConnectionUp (self, event):
    log.debug("Connection %s" % (event.connection,))
    ProactiveTree(event.connection)


def launch ():
 core.registerNew(proactive_tree)
```

jelly_controller_julen.py

```
# Este controlador esta implementado para posibilitar el routing para el caso
aleatorio controlado Jellyfish 7,3,2 con semilla 27


from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *
from pox.lib.util import dpidToStr
import time
import math


log = core.getLogger()
```

```
# ajustar los parametros K (routing trafico local) y numSwitches (routing
switch-to-switch), segun la topologia que se lanza
numSwitches = 7
K = 3


switches =[] # lista donde se guardan los identificadores de los switches para
calcular los correspondientes shortest path


for x in range (0, numSwitches): # crear una lista de identificadores
representada por los mismos identificadores que se les ha dado a los switches
OpenFlow, para despues poder calcular los caminos minimos entre los mismos
   switch = "S_" + str(x)
   switches.append(switch)


def bfs_paths(mapa, origen, destino): # Recorre el grafo en anchura para
calcular los caminos existentes entre dos nodos. Dado un mapa que representa
las conexiones entre los switches, un nodo de origen y un nodo de destino,
devuelve todos los caminos entre el nodo origen y nodo destino, ordenados
desde el mas corto hasta el mas largo. En caso de que haya mas de un camino
con misma longitud, coloca el primero el que antes haya encontrado el
algoritmo. A nosotros siempre nos interesara el primer camino minimo que
encuentre.
   cola = [(origen, [origen])]
   while cola:
      (vertice, ruta) = cola.pop(0)
      for siguiente in mapa[vertice] - set(ruta):
          if siguiente == destino:
              yield ruta + [siguiente]
          else:
              cola.append((siguiente, ruta + [siguiente]))


# Caso 1: caso aleatorio controlado Jellyfish 7,3,2 seed 27
# grafo jf,7,3,2 seed(27), debe generarse a partir de enlaces_guardados1. El
grafo debe estar represetando como mapa1 para que la funcion bfs_paths pueda
calcular los caminos minimos
mapa1 = {'S_0': set(['S_5','S_4']),
    'S_1': set(['S_6','S_5']),
    'S_2': set(['S_3','S_4']),
    'S_3': set(['S_6','S_2']),
    'S_4': set(['S_0','S_2']),
    'S_5': set(['S_0','S_1']),
    'S_6': set(['S_1','S_3'])}
enlaces_guardados1 = [('S_6','S_1'),('S_0','S_5'),('S_5','S_1'),('S_6','S_3'),
('S_0','S_4'),('S_3','S_2'),('S_4','S_2')] # A partir de enlaces_guardados1 se
genera mapa1. enlaces_guardados1 se genera al lanzarse la topologia con los
parametros jf,7,3,2 seed 27


# Caso 2: caso aleatorio controlado Jellyfish 3,3,2 seed 27
# grafo jf,3,2,2 seed(27), debe generarse a partir de enlaces_guardados2
```

```
mapa2 = {'S_0': set(['S_1','S_2']),
    'S_1': set(['S_0','S_2']),
    'S_2': set(['S_0','S_1'])}
enlaces_guardados2 = [('S_0','S_2'),('S_1','S_2'),('S_1','S_0')] # A partir de
enlaces_guardados2 se genera mapa2. enlaces_guardados2 se genera al lanzarse
la topologia con los parametros jf,3,2,2 seed 27
all_shortest_paths = [] # array para guardar todos los caminos mas cortos
entre todos los nodos


# para cada switch origen, calcular todos los caminos respecto a los demas
destinos y para cada destino distinto, guardar el minimo (el primero) en el
array all_shortest_paths
for origen in range (0, numSwitches):
   for destino in range (0, numSwitches):
      if origen != destino: # evita calcular el camino entre el nodo origen
consigo mismo
          paths = list(bfs_paths(mapa1, switches[origen], switches[destino]))
# calcular caminos entre nodo origen y nodo destino
          shortest_path = paths[0] # obtener el primero de todos los caminos
posibles, el mas corto, o, en caso de que haya mas de un camino minimo, el
primero que el algoritmo encuentra
          all_shortest_paths.append(shortest_path)


print "Todos los caminos minimos para cada nodo de origen son:"
print all_shortest_paths


# funciones para saber por que puerto de salida del switch de origen debemos
distribuir la informacion
def posicion_enlace(enlace1, enlace2, lista_enlaces): # devuelve la posicion
que tiene un enlace bidireccional en la lista de enlaces guardados
   if (enlace1 in lista_enlaces):
      posicion = lista_enlaces.index(enlace1)
   if (enlace2 in lista_enlaces):
      posicion = lista_enlaces.index(enlace2)
   return posicion


def num_aparicion_switch(switch, lista_enlaces, posicion): # devuelve el
numero de apariciones del primer switch que aparece en el enlace, en la lista
de enlaces guardados, hasta la posicion que se ha calculado
   aparicion = 0
   for enlace in range (0, posicion):
      if switch in lista_enlaces[enlace]:
          aparicion += 1
   return aparicion


class ProactiveJelly (EventMixin):


 def __init__ (self,connection):
    self.connection = connection # devuelve como string la MAC del switch
OpenFlow en el siguiente formato: [XX-XX-XX-XX-XX-XX_numb], los corchetes y el
```

```
numb es algo que ignoraremos. Numb simplemente representa el orden de conexion
de los switches con el controlador. Partiremos desde aqui para saber con quien
nos comunicamos.
    self.listenTo(connection)


    self.ids = str(connection).split("-")[5]
    self.ids = self.ids.split(" ")[0] # Digitos 11 y 12 de la MAC
    self.idint = int(self.ids) # identificador del switch OpenFlow


    self.ids1 = str(connection).split("-")[0]
    self.ids1 = self.ids1.split("[")[1] # Digitos 1 y 2 de la MAC
    self.idint1 = int(self.ids1) # identifica todos los switches OpenFlow


    match1 = of.ofp_match() # emparejamiento basado en direcciones IP destino
para distribuir trafico local, prioridad 2
    match1.dl_type = 0x0800 # es necesario indicar con que tipo de direcciones
se quiere hacer el emparejamiento. En este caso, IPv4 (0x0800)


    match2 = of.ofp_match() # emparejamiento basado en direcciones IP destino
para distribuir trafico switch-to-switch, prioridad 1
    match2.dl_type = 0x0800 # es necesario indicar con que tipo de direcciones
se quiere hacer el emparejamiento. En este caso, IPv4 (0x0800)


if(self.idint1 == 01): # reglas para todos los switches OpenFlow


   # trafico hacia abajo (local, prioridad 2)
   for h in range (2, K + 2): # entrada para llegar a cada host conectado a
cada switch OpenFlow, para cada puerto hacia abajo
      switch = self.idint # switch correspondiente
      host = h # las direcciones de los host empiezan desde .2
      match1.nw_dst = "10.0." + str(switch) + "." + str(host) # emparejar con
la direccion IP destino
      self.install(h - 1, match1, 2) # instalar regla trafico hacia abajo


   # trafico switch-to-switch (prioridad 1)
   if(self.idint == 00): # Flujos para S_0


      # matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch procesa (numSwitches-1)
caminos, por lo tanto, (numSwitches-1) flujos
          #print camino
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto. Indica la subred de destino para cada
flujo (matching)
          match2.nw_dst = "10.0." + str(destino) + ".0/24"

          # accion output_port del flujo
```

```
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
          posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1) #
guardar la posicion del enlace bidireccional entre los dos primeros elementos
de la lista, para luego contar el numero de apariciones del primer elemento
hasta esa posicion y saber por que puerto redirigir la informacion
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion) # calcula el numero de apariciones del primer elemento de una lista
hasta el indice 'posicion' calculado mediante la funcion posicion_enlace
          puerto = K + aparicion + 1 # calcular puerto de salida
          self.install(puerto, match2, 1) # instalar regla trafico switch-to-
switch


   if(self.idint == 01): # Flujos para S_1


      # matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch, (numSwitches-1) caminos, por
lo tanto, (numSwitches-1) flujos
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto
          match2.nw_dst = "10.0." + str(destino) + ".0/24"

          # accion output_port del flujo
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
             posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1)
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion)
          puerto = K + aparicion + 1
             self.install(puerto, match2, 1)


   if(self.idint == 02): # Flujos para S_2


      # matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch, (numSwitches-1) caminos, por
lo tanto, (numSwitches-1) flujos
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto
          match2.nw_dst = "10.0." + str(destino) + ".0/24"
```

```
          # accion output_port del flujo
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
          posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1)
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion)
          puerto = K + aparicion + 1
          self.install(puerto, match2, 1)


   if(self.idint == 03): # Flujos para S_3


      # matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch, (numSwitches-1) caminos, por
lo tanto, (numSwitches-1) flujos
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto
          match2.nw_dst = "10.0." + str(destino) + ".0/24"

          # accion output_port del flujo
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
          posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1)
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion)
          puerto = K + aparicion + 1
          self.install(puerto, match2, 1)


   if(self.idint == 04): # Flujos para S_4


      # matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch, (numSwitches-1) caminos, por
lo tanto, (numSwitches-1) flujos
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto
          match2.nw_dst = "10.0." + str(destino) + ".0/24"

          # accion output_port del flujo
```

```
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
          posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1)
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion)
          puerto = K + aparicion + 1
          self.install(puerto, match2, 1)



   if(self.idint == 05): # Flujos para S_5


      # matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch, (numSwitches-1) caminos, por
lo tanto, (numSwitches-1) flujos
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto
          match2.nw_dst = "10.0." + str(destino) + ".0/24"

          # accion output_port del flujo
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
          posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1)
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion)
          puerto = K + aparicion + 1
          self.install(puerto, match2, 1)


   if(self.idint == 06): # Flujos para S_6


# matching del flujo
      for camino in range (self.idint * (numSwitches - 1),(numSwitches - 2) +
((numSwitches - 1) * self.idint) + 1) : # hacer que cada switch procese los
caminos cortos que le correspondan. Cada switch, (numSwitches-1) caminos, por
lo tanto, (numSwitches-1) flujos
          destino = all_shortest_paths[camino][-1].split("_")[1] # ultimo
identificador de cada camino mas corto
          match2.nw_dst = "10.0." + str(destino) + ".0/24"

          # accion output_port del flujo
          primer_elem = all_shortest_paths[camino][0] # primer identificador
de cada camino mas corto
```

```
          segundo_elem = all_shortest_paths[camino][1] # segundo identificador
de cada camino mas corto
          enlace1 = (primer_elem, segundo_elem)
          enlace2 = (segundo_elem, primer_elem)
          posicion = posicion_enlace(enlace1, enlace2, enlaces_guardados1)
          aparicion = num_aparicion_switch(primer_elem, enlaces_guardados1,
posicion)
          puerto = K + aparicion + 1
          self.install(puerto, match2, 1)
 def install(self, port, match, priority): # funcion que se ejecuta cada vez
que se instala una nueva entrada en algun switch. Como parametros recibe el
puerto port, puerto por el que se reenvian los paquetes que coincidan (output
action), la direccion IP o conjunto de direcciones IP (matching) y la
prioridad (priority)
    msg = of.ofp_flow_mod() # prepara el paquete msg para que posteriormente
se instale como una entrada nueva en la tabla de flujos correspondiente
    msg.match = match # anade el matching que pasamos como parametro a esa
nueva entrada, en este caso el matching se basa en direcciones IP destino, por
lo tanto, el campo nw_dst es el que utilizaremos
    msg.priority = priority # anade la prioridad que pasamos como parametro a
esa nueva entrada
    msg.actions.append(of.ofp_action_output(port = port)) # anade la accion a
realizar con las direcciones que coincidan con la entrada para reenviarla por
el puerto que pasamos por el parametro
    self.connection.send(msg) # el controlador envia el mensaje msg al switch
para instalar el flujo correspondiente en el switch OpenFlow


 def _handle_PacketIn (self, event): # funcion que se encarga de procesar
paquetes a los cuales un switch no les ha encontrado coincidencia. Posibilita
que el controlador reciba esos paquetes de tipo packet_in, aunque no se vaya a
hacer nada con ellos, ya que todos los switches se han instruido de manera
correcta de antemano. De lo contrario, saltarian errores, puesto que no se les
daria ningun tratamiento a los paquetes que no se les ha encontrado
coincidencia.
    packet = event.parse()
    msg = of.ofp_packet_out()
    msg.buffer_id = event.ofp.buffer_id
    msg.in_port = event.port

class proactive_jelly (EventMixin):


 def __init__(self):
    self.listenTo(core.openflow)


 def _handle_ConnectionUp (self, event):
    log.debug("Connection %s" % (event.connection,))
    ProactiveJelly(event.connection)


def launch ():
 core.registerNew(proactive_jelly)
```

14. Referencias

[1] Wikipedia - Redes definidas por software – 2018
URL: https://es.wikipedia.org/wiki/Redes_definidas_por_software

[2] María Calderón Pastor, Marifeli Sedano Ruiz, Santiago Eibe Garcia. “Principios de las Redes Activas”. - 2018
URL: https://e-archivo.uc3m.es/bitstream/handle/10016/2343/apli_RA_jitel99.pdf

[3] Carlos Spera (Logicalis) - “Software Defined Network: el futuro de las arquitecturas de red”. - 2018
URL: https://www.la.logicalis.com/globalassets/latin-america/logicalisnow/revista-20/lnow20-nota-42-45.pdf

[4] Infonetics - SDN and NFV strategy survey - 2018
URL: https://technology.ihs.com/aboutus/pressreleases

[5] Open Networking Foundation - “SDN Architecture”. - 2018
URL: https://www.ramonmillan.com/documentos/bibliografia/SDNArchitecture_ONF.pdf

[6] theMETISfiles – The future of the network is software defined -2018
URL: http://www.themetisfiles.com/2012/10/the-future-of-the-network-is-software-defined/

[7] Ramón Jesús Millán Tejedor - “El futuro de las redes inteligentes”. - 2018
URL: https://www.ramonmillan.com/tutoriales/sdnredesinteligentes.php

[8] Amin Tootoonchian, Yashar Ganjali. “Hyperflow: A Distributed Control Plane for OpenFlow”. - 2018
URL: https://www.usenix.org/legacy/event/inm/tech/full_papers/Tootoonchian.pdf

[9] Wikipedia – OpenFlow – 2018
URL: https://es.wikipedia.org/wiki/OpenFlow

[10] SearchDataCenter – Cinco protocolos SDN que no son OpenFlow - 2018
URL: https://searchdatacenter.techtarget.com/es/cronica/Cinco-protocolos-SDN-que-no-son-OpenFlow

[11] Srini Seetharaman, Stanford Clean Slate Lab, Deutsche Telekom Innovation Center. “OpenFlow-Software-defined Networking”. - 2018
URL: https://www.slideshare.net/openflow/openflow-tutorial

[12] Github – Noxrepo/POX – 2018
URL: https://github.com/noxrepo/pox

[13] Arnav Shivendu, Depandra Dhakal, Diwas Sharma. “Emulation of Shortest Path Algorithm in Software Defined Networking Environment”. - 2018
URL: https://research.ijcaonline.org/volume116/number1/pxc3902344.pdf

[14] NetworkStatic – OpenFlow: Proactive vs Reactive Flows – 2018
URL: http://networkstatic.net/openflow-proactive-vs-reactive-flows/

[15] Github – Mininet - 2018
URL: https://github.com/mininet/mininet

[16] OpenDayLight – OpenDaylight Controller:Installation – 2018
URL: https://wiki.opendaylight.org/view/OpenDaylight_Controller:Installation

[17] Brandon Heller – RipL - 2018
URL: https://github.com/brandonheller/ripl

[18] Mininet – An instant Virtual Network on your Laptop – 2018
URL: http://mininet.org/

[19] Scielo – Mininet: una herramienta versátil para emulación y prototipado de Redes Definidas por Software – 2018
URL: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S1909-83672015000100009

[20] Ankit Singla, Chi-Yao Hong, Lucian Popa, P. Brighten Godfrey. “Jellyfish: Networking Data Centers Randomly”. - 2018
URL: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final82.pdf

[21] Python.org – Pyton Patterns Implementing Graphs - 2018
URL: https://www.python.org/doc/essays/graphs/

[22] Edd Mann. “Depth-First Search and Breadth-First Search in Python”. - 2018
URL: https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/

[23] Wikipedia – Protocolo de resolución de direcciones ARP – 2018
URL: https://es.wikipedia.org/wiki/Protocolo_de_resoluci%C3%B3n_de_direcciones

[24] Javier Navaridas, Jose Miguel-Alonso, Javier Ridruejo, Wolfgang Denzel. “Reducing Complexity in Tree-like Computer Interconnection Networks”. - 2018
URL: https://www.sciencedirect.com/science/article/pii/S0167819109001264?via%3Dihub

[25] Open Networking Foundation – OpenFlow Switch Specification – 2018
URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf