Taller de Sistemas Ciberfísicos

Curso 2020

SDN para Enrutamiento en el Datacenter

Autor:
Agustina Parnizari

Supervisores:
Eduardo Grampín
Alberto Castro
Leonardo Alberro

9 de agosto de 2020

Índice

1. Introducción

El tráfico en los Datacenter ha cambiado en los últimos tiempos. Mientras que antes predominaba un tráfico Norte-Sur correspondiente con arquitecturas Cliente-Servidor donde la mayor carga de trabajo se presentaba en las redes WAN, hoy en día la mayor carga de trabajo se da en un sentido Este-Oeste dentro del Datacenter.

Esto se debe en gran parte al auge de servicios en la nube y de los proveedores de contenido. Los servidores interactúan entre si procesando cantidades de datos masivas, realizando respaldos y replicas o almacenando bases de datos distribuidas. Particularmente los sistemas ciberfísicos y la industria 4.0 utilizan servicios en la nube para el procesamiento de los datos que generan, por lo que se ven afectados ante los problemas que se presenten en esta infraestructura.

Por los motivos presentados, las topologías de Datacenter tradicionales y los protocolos de ruteo como OSPF, IS-IS y BGP ya no se ajustan a las necesidades actuales, como son la escala, resiliencia de la red ante fallas y distribución del tráfico.

Debido a ello han resurgido topologías de tipo Clos como Fat-Tree y Leaf-Spine. Este tipo de topologías tiene una forma de árbol en la cual cada rack del Datacenter contiene un ToR (Top of Rack) o Leaf, que se interconecta con los otros Leaf mediante switches de agregación (o Spines), estableciendo distintos caminos. Dichas topologías son robustas a fallas y permiten distribuir la carga mediante algoritmos equitativos como Equal Cost Multi Path (ECMP).

A su vez han surgido protocolos que explotan las cualidades de estas topologías. Se tienen soluciones distribuidas como BGP para el Datacenter, Open-Fabric o RIFT las cuales ya están siendo utilizadas o se encuentran en proceso de estandarización. Por otro lado se tienen soluciones centralizadas basadas en redes definidas por software (SDN) como Segment Routing, la cual se explorará en este trabajo.

El objetivo de este proyecto es entender el funcionamiento de una solución centralizada para el ruteo en el Datacenter, conocer herramientas utilizadas en este paradigma y crear un entorno de trabajo que permita probar su funcionamiento.

2. Marco Teórico

2.1. Redes Definidas por Software

Las Redes Definidas por Software o SDN (Software Defined Networks) plantean un paradigma en el cual se separa físicamente el plano de control del plano de forwarding o datos. El plano de control además tiene la capacidad de controlar varios dispositivos.

Su arquitectura se divide en tres capas, como se puede ver en la figura 1. En la capa de Aplicación corren las aplicaciones y servicios de la red, que van a utilizar los recursos disponibles. Las aplicaciones se comunicán con la capa de control mediante APIs REST, extrayendo información de los recursos del controlador SDN. La capa de Control, que contiene el controlador SDN, se comunica directamente con la infraestructura mediante interfaces como OpenFlow (una de las tecnologías abiertas más utilizadas) para controlar a los dispositivos de red (switches, routers, etc) que se encuentran en la capa inferior o de Infraestructura.

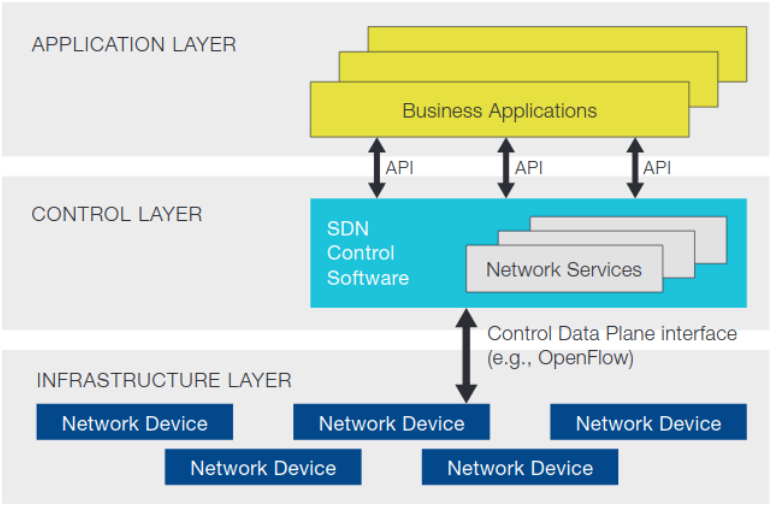


Figura 1: Arquitectura de SDN [1]

SDN presenta ciertos beneficios en comparación con el paradigma de red distribuido tradicional. Entre estos beneficios se pueden mencionar los siguientes:

- Directamente programable: al estar desacopladas las funciones de control de

las funciones de forwarding se puede configurar y programar el control de la red con herramientas de automatización.

- Ágil: Los administradores pueden realizar cambios sobre el tráfico de forma dinámica y aplicándolos a toda la red según las necesidades de ese momento.
- Gestión centralizada: La lógica de la red se centraliza en controladores que tienen una visión completa de la infraestructura. Esto facilita la gestión de los administradores que tienen acceso a una única herramienta para aplicar cambios en la red.
- Configuración programada: Los administradores puede gestionar, asegurar y optimizar los recursos de red mediante programas atomizados, que pueden desarrollar ellos mismos dado que no dependen de software propietario.
- Estándares abiertos: SDN simplifica el diseño y la operativa de red, dado que las instrucciones las ejecutan los controladores, generalmente con protocolos abiertos, en lugar de dispositivos y protocolos cerrados.

Por otro lado la adopción de SDN tiene algunos desafíos por enfrentar, dentro de los cuales se encuentran:

- Seguridad: La gestión centralizada presenta un único punto de falla. Si este es atacado toda la red se ve comprometida.
- Cuello de botella: Cuando se utiliza una única instancia de controlador SDN, puede convertirse en un cuello de botella ante una gran cantidad de tráfico.
- Universalización de APIs: No existe un estándar universalmente aceptado para la comunicación entre las aplicaciones y los controladores. Existen distintas APIs para distintos controladores SDN lo cual hace más difícil el desarrollo de nuevas aplicaciones que se comuniquen con distintos controladores.

Esta arquitectura presenta casos de uso interesantes, desde virtualización de redes donde se pueden encontrar herramientas comerciales como NSX de VMware o abiertas como Neutron de Openstack, su uso en Datacenters, redes WAN como la WAN privada de Google, hasta su uso en redes celulares 5G. Se puede consultar

en mas detalle sobre sus posibles casos de uso en [2].

Uno de los casos de uso predominantes es su aplicación en Datacenters donde se busca utilizar switches 'white-box' o abiertos. Estos switches al contrario de los switches 'black-box' tradicionalmente atados a las especificaciones de sus proveedores, permiten un control mediante estándares abiertos y programables como OpenFlow.

Los switches generalmente se encuentran en una topología leaf-spine como se puede ver en la figura 2. Mientras que entre los servidores de un mismo rack (Leaf) la comunicación es a nivel de capa de enlace, la interconexión entre los distintos racks es de capa de red. El ruteo no es realizado por los switches sino por un controlador centralizado el cual indica a los switches las reglas de forwarding que deben seguir.

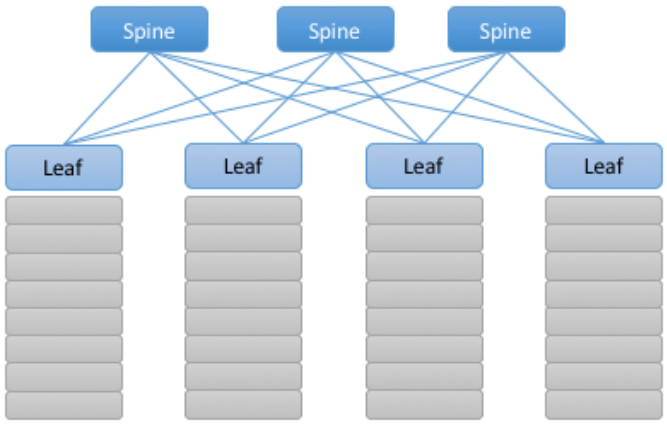


Figura 2: Ejemplo de leaf-spine Switching Fabric [2]

Se va a estudiar en detalle este caso de uso aplicando metodologías de ruteo basadas en origen para topologías leaf-spine.

2.2. OpenFlow

Openflow es un conjunto de protocolos y API utilizados en SDN para separar el plano de control del plano de forwarding. Los principales componentes de su arquitectura se muestran en la figura 3. Estos son el controlador Openflow, que genera las instrucciones del plano de forwarding, el protocolo de comunicación entre

el controlador y los dispositivos, y el switch como unidad cuya función es procesar paquetes.

El controlador OpenFlow genera las instrucciones para el plano de forwarding a partir de las decisiones de ruteo de las aplicaciones. Las instrucciones estan contenidas en estructuras de datos en forma de tabla. Luego el protocolo envía a los switches dichas tablas y cada switch procesa los paquetes entrantes según las instrucciones obtenidas.

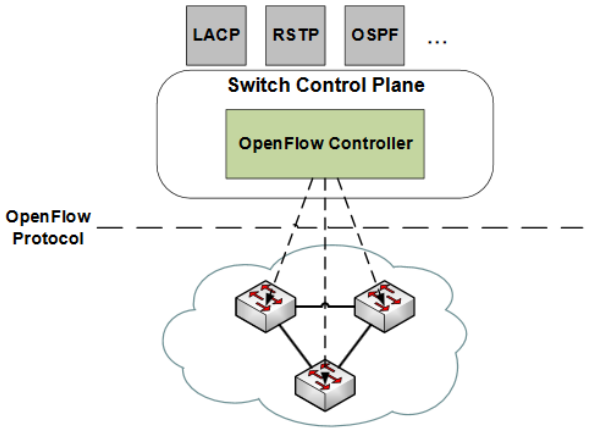


Figura 3: Arquitectura de OpenFlow [3]

Estructuras de datos

Las tablas de flujos y grupos son estructuras de datos que definen el comportamiento de los paquetes que ingresen y egresen de los switch en el plano de datos. Las mismas son generadas por el controlador SDN en el plano de control y se construyen mediante las API.

Las tablas de flujos analizan la información de los paquetes que ingresan al switch y ejecuta la acción que le corresponda. Se realiza un matcheo basado en el modelo TCP/IP, donde cada campo puede contener información de la interfaz física, dirección MAC, VLAN, dirección IP o puerto de transporte. Una vez se encuentra para el paquete coincidencia con un flujo se ejecuta una instrucción con un conjunto de acciones. La imagen 4 muestra la estructura de una tabla de flujos.

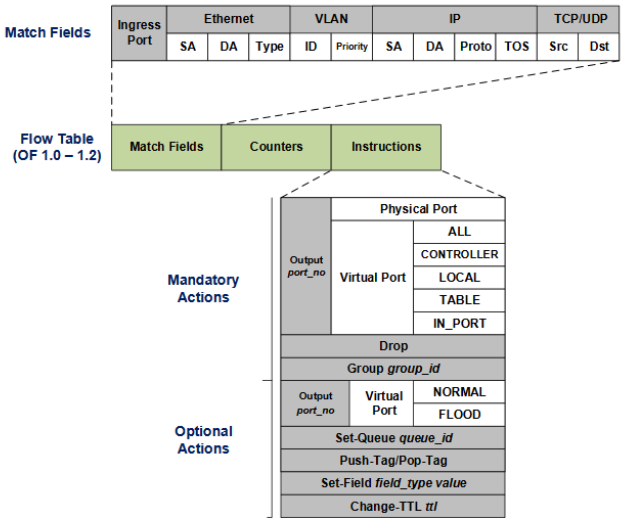


Figura 4: Tabla de Flujo OpenFlow [3]

El campo *Match* contiene la información de matcheo para comparar con el paquete y el campo *Counters* contiene contadores que permiten al sistema generar estadísticas (cantidad de paquetes, bytes, etc). En cuanto al capo *Instructions*, se puede observar que algunas acciones son obligatorias mientras que otras son opcionales. Se puede consultar en mayor detalle sobre la estructura en [3], pero resulta de interés comprender la acción *Output* que utiliza distintos tipos de puertos:

- El puerto físico es una interfaz de hardware del switch y a su vez una interfaz Ethernet.
- Un puerto lógico es una interfaz virtual de un Switch y una interfaz Ethernet.
- Un puerto reservado es una interfaz lógica que se asocia a una operación de forwarding específica y puede estar asociada a mas de una interfaz lógica o física. Los distintos puertos reservados son:
  - ALL: Envía paquetes a todas las interfaces.
  - CONTROLLER: Envía paquetes al controlador.
  - LOCAL: Envía paquetes a la interfaz de loopback.
  - TABLE: Envía paquetes a la proxima tabla de flujo.

- IN_PORT: Envía paquetes a la interfaz de entrada.
- NORMAL: Procesa los paquetes como un switch tradicional.
- FLOOD: Envía paquetes por todas interfaces salvo la de ingreso. Normalmente se utiliza para ARP y LLDP.

Por otro lado se tienen las tablas de grupos, que permiten enviar un paquete por diferentes puertos debido a diferentes propósitos. Si bien la acción FLOOD permite enviar un mensaje por varios puertos, esta limitada a protocolos como LLDP. Con las tablas de grupos sin embargo se pueden agrupar puertos para diferentes acciones, por ejemplo para utilizar ECMP.

En la imagen 5 se presenta la estructura de las tablas de grupo. Las entradas de las tablas contienen un identificador del grupo, un tipo de grupo, contadores para estadísticas al igual que en las tablas de flujo, y un conjunto de acciones o *buckets*. Los *buckets* contienen las instucciones a seguir, que pueden incluir enviar los paquetes por uno o multiples puertos, o enviar el paquete a otro grupo para que lo procese, permitiendo encadenar acciones lo cual veremos más adelante que es necesario en Segment Routing.

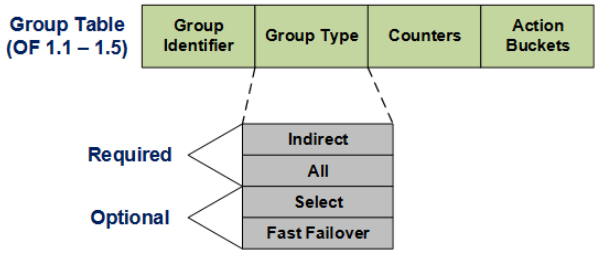


Figura 5: Tabla de Grupo OpenFlow [3]

Los grupos son de cuatro tipos distintos, donde dos de ellos son requeridos y siempre deben estar presentes mientras que el resto son opcionales. Los distintos tipos son:

- *Indirect*: Ejecuta un *bucket* del grupo asociado a un puerto de salida, permitiendo emular el comportamiento de capa de red, como apuntar al próximo destino IP.

- *All*: Se ejecutan todos los *buckets* del grupo y el paquete se envía por cada interfaz relacionada con un *bucket*, permitiendo emular el comportamiento de multicast o broadcast.
- *Select*: Ejecuta un *bucket* en el grupo. El mismo va cambiando y es seleccionado por un algoritmo de Round Robin o una función de Hash, permitiendo el balanceo de carga. Con este tipo de grupo se puede emular el comportamiento de ECMP por lo cual va a ser de especial interés.
- *Fast Failover*: Ejecuta el primer *bucket* cuya interfaz asociada está operativa. Si la interfaz falla se ejecuta el próximo *bucket* activo. Con este tipo de grupo se puede emular fast failover para tener un backup de links sin consultar al controlador SDN ante fallas.

Protocolo de red

El protocolo de red o canal Openflow representa una interconexión logica entre el controlador Openflow y los switches. El controlador configura y gestiona los switches y recibe eventos de los mismos para conocer el estado de la red.

Los mensajes pueden ser iniciados por el controlador, a lo cual se le llama *'controller-to-switch'*, con lo cual monitorean y configuran el comportamiento de los switch. También pueden ser inicializados por los switches para enviar información sobre el estado de la red al controlador, a lo cual se le llama mensajes asíncronos o pueden ser simétricos, es decir iniciados por ambas partes, lo cual generalmente se usa para establecer la conexión o enviar mensajes *keep alive*.

Es de interés para entender como funciona el protocolo dar una breve explicación de algunos mensajes relevantes. Por más detalle de los mismos se puede consultar [4].

- OFPT_HELLO: Este mensaje se utiliza por el controlador y los switches para establecer la conexión. Se identifica y establece que versión de OpenFlow se va a utilizar.
- OFPT_FEATURES_REQUEST / REPLY: El controlador envía la consulta a un switch para obtener sus *capabilities*. Este mensaje solo contiene el header

de OpenFlow. El switch responde indicando parámetros de interés para el controlador.

- OFPT_MULTIPART_REPLY: Utilizado para realizar consultas que puedan traer una cantidad de datos mayor a la soportada por el cuerpo de OpenFlow (limitada a 64KB). Generalmente se utiliza para obtener estadísticas y detalles de los switches. Dentro del mensaje se indica el tipo de datos que se quiere obtener. Por ejemplo OFPMP_PORT_DESC obtiene el detalle de cada puerto del switch.
- OFPT_GET_CONFIG_REQUEST / REPLY: El controlador puede consultar la configuración de los switches. La consulta es vacía mientras que la respuesta contiene distintos parámetros de configuración.
- OFPT_BARRIER_REQUEST / REPLY: El controlador puede consultar si el switch termino de procesar ciertas operaciones. Antes de contestar el switch debe terminar de procesar todos los mensajes pendientes.
- OFPT_FLOW_MOD: El controlador modifica las tablas de flujos del switch con este mensaje.
- OFPT_GROUP_MOD: El controlador modifica las tablas de grupos del switch con este mensaje.
- OFTP_PACKET_IN / OUT: Cuando se envía trafico del plano de datos al controlador lo recibe mediante un mensaje PACKET_IN. Cuando es el controlador el que desea enviar un mensaje envia un PACKET_OUT. Generalmente los mensajes ARP se envían al controlador que resuelve la consulta.

Switch OpenFlow

El switch OpenFlow es la unidad lógica que procesa paquetes a partir de las tablas de flujos y grupos vistos anteriormente. El switch se conecta al controlador mediante el canal Openflow, donde se manejan los mensajes del protocolo. En la figura 6 se puede ver su estructura.

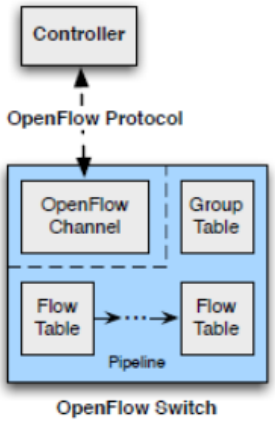


Figura 6: Switch OpenFlow [3]

Los paquetes son procesados a través de varias tablas de flujo. Estas tablas se procesan secuencialmente, a lo que se le llama *Pipeline* de OpenFlow. Los paquetes pueden ser enviados directo a una interfaz de salida o al controlador desde cualquiera de las tablas, o pueden ser descartados en este proceso.

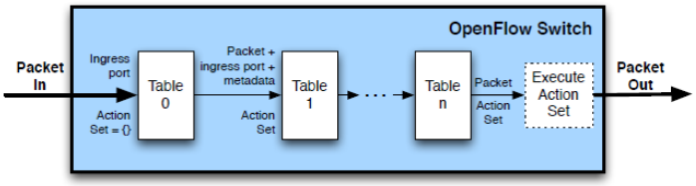


Figura 7: Pipleline de OpenFlow [3]

2.2.1. Open vSwitch

Open vSwitch es un switch de código abierto diseñado para utilizarse en producción como switch virtual. Entre sus funcionalidades tiene soporte de OpenFlow por lo que se puede utilizar en la arquitectura SDN.

La herramienta tiene soporte para sistemas operativos Linux a partir del kernel versión 3.10, por lo cual un sistema Linux con Open VSwitch instalado y las interfaces de red necesarias, conectado a un controlador SDN, actúa como switch

Openflow. Esto será de utilidad al momento de realizar pruebas en un entorno virtual. Se puede profundizar sobre Open vSwitch en [7].

Open vSwitch soporta las distintas versiones de OpenFlow y puede utilizarlas al mismo tiempo en distintos bridges lo cual permite una gran flexibilidad en su uso. Particularmente se utilizará OpenFlow version 1.3 dado que es compatible con los requerimientos de Segment Routing. Open vSwitch tiene soporte de esta versión de OpenFlow así como sus versiones anteriores.

2.3. Ruteo basado en origen

El ruteo basado en origen consiste en que el dispositivo de red de origen defina la ruta total o parcial de un paquete en la red, en lugar de que sea procesado por cada nodo intermedio en la red. El camino se agrega al cabezal del paquete y los nodos intermedios realizan forwarding a partir de esta información, en lugar de calcular el próximo salto en cada paso.

2.3.1. Segment Routing

Segment Routing es un método de ruteo basado en origen en el cual se tiene un conjunto de instrucciones o segmentos que guían a los paquetes a través de los dispositivos de red. Este concepto fue propuesto por Cisco y su estándar es mantenido por el grupo de la IETF llamado Source Packet Routing in Networking (SPRING).

Esta tecnología permite utilizar ingeniería de tráfico de una forma sencilla y escalable mediante un soporte de túneles y políticas, lo cual da una gran flexibilidad a los administradores de red.

Puede aplicarse a MPLS sobre IPv4 sin realizar cambios en su arquitectura, donde los segmentos son etiquetas MPLS y la lista de instrucciones es el Stack MPLS. Se puede aplicar también a IPv6 donde los segmentos son direcciones IPv6 y se debe agregar a los paquetes un cabezal de ruteo, conteniendo la lista de direcciones. En este trabajo se estudiara su aplicación a MPLS, razón por la cual se mencionarán algunos conceptos básicos de dicha metodología.

MPLS

MPLS (Multiprotocol Label Switching) es un mecanismo de conmutación de etiquetas que opera entre la capa de red y la capa de enlace. Se diseño para unificar los servicios de transporte de datos para las redes basadas en circuitos y paquetes, remplazando a tecnologías como ATM.

La conmutación de paquetes es el mecanismo de forwarding que utiliza, en el cual los paquetes contienen una etiqueta que indica a los nodos como procesar y reenviar los datos. La información de etiquetas se mantiene en una LIB (Label Infomation Base) en cada nodo y es distribuida mediante el protocolo LDP (Label-Distributed Protocol).

En la figura 8 se puede observar como opera MPLS. Todos los nodos realizan conmutación de etiquetas, lo cual se identifica como nodos LSR (Label Switch Router). Mientras que algunos nodos solo realizan forwarding de etiquetas MPLS, algunos nodos identificados como Edge Routers realizan la conversión entre IP y MPLS.

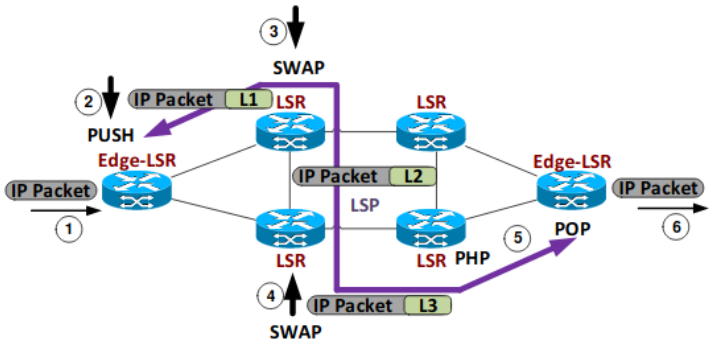


Figura 8: Modelo MPLS [3]

Las acciones que pueden tomar los nodos LSR en la conmutación de etiquetas son:

- PUSH: Se asigna una etiqueta a los paquetes IP que ingresan a la red MPLS.
- SWAP: Dirige el paquete al próximo nodo según la etiqueta MPLS.

- POP: Se quita la etiqueta MPLS del paquete IP. Generalmente esta acción la realiza el penúltimo nodo de la ruta y no el nodo de borde, lo que se conoce como PHP (Penultimate Hop Popping).

Modelo de Segment Routing

Se introducirá la arquitectura y funcionamiento de Segment Routing aplicado a MPLS. Por más detalles sobre dicha aplicación o su implementación en IPv6 se puede consultar en [8] y [3].

En la figura 9 se puede observar el comportamiento de Segment Routing utilizando MPLS. A partir del nodo de borde se genera una lista ordenada de segmentos identificados como SID (Segment ID), la cual atraviesa la red mientras se le realizan las siguientes operaciones:

- PUSH: Ejecutada por un nodo SR para introducir un segmento en la lista. Coincide con la acción PUSH MPLS.
- CONTINUE: Reenvía el paquete sin realizar cambios en la lista de segmentos, correspondiéndose con la acción SWAP MPLS.
- NEXT: Quita el segmento activo y pasa al próximo, correspondiendo con la accion POP MPLS.

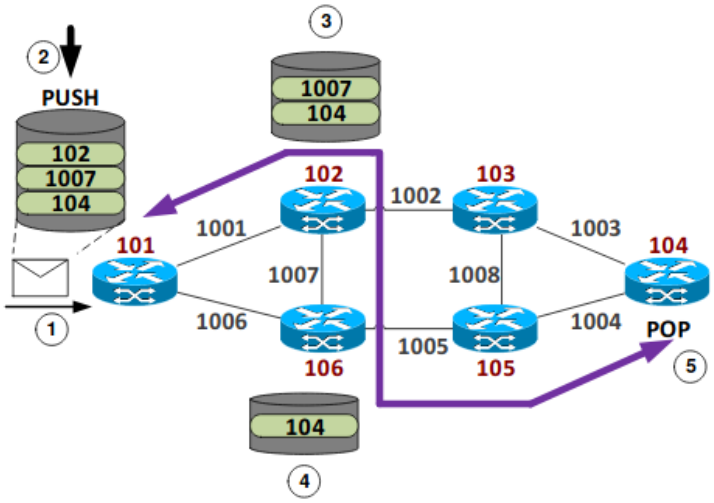


Figura 9: Segment Routing [3]

Como se menciono anteriormente, los segmentos se identifican por su SID, los cuales pueden ser de distintos tipos:

- Node-Sid: Identificador único de un nodo SR.
- Adjacency-Sid: Identifica un link de egreso de un nodo.
- Service-Sid: Identifica un servicio particular que provee un nodo. Puede ser por ejemplo un servicio de inspección de paquetes.
- Anycast-Sid: Identifica un grupo de nodos, lo cual permite funcionalidades como Multicast o ECMP.

Hasta ahora se vio el modelo de plano de datos de Segment Routing, es de interés conocer el funcionamiento del plano de control. Se tienen tres posibilidades para la implementación del mismo:

- Configuración estática: Uso de túneles configurados manualmente. Esto se utiliza sobre todo para encontrar problemas de forma temporal.
- Algoritmo distribuido: Los nodos calculan el camino más corto al destino utilizando otros algoritmos como OSPF e IS-IS y luego generan el stack de SID.

- SDN: Un controlador SDN realiza los cálculos teniendo una visión global de la red y envía el camino computado a los nodos.

Tanto en el caso distribuido como el caso centralizado, el algoritmo de calculo de rutas por defecto es SPF para ECMP, permitiendo en el caso centralizado agregar políticas que sobre escriben su comportamiento, permitiendo una mayor flexibilidad en la aplicación de técnicas de ingeniería de tráfico.

En el presente trabajo se estudiará una implementación particular de Segment Routing sobre SDN, utilizando el controlador ONOS.

2.4. ONOS

ONOS (Open Networking Operating System) es un controlador SDN que se define como un sistema operativo dado que provee APIs, asignacion de recursos, servicios y software orientado a usuarios, como un CLI y una GUI.

Se presentará el caso de uso de Ruteo en el Datacenter con sus aplicaciones y relación con el controlador ONOS. Se puede profundizar sobre la herramienta en [9].

Segment Routing en ONOS

Se presenta la arquitectura de Segment Routing implementada como aplicación de ONOS, la cual provee el plano de control para aplicar Segment Routing sobre SDN. La figura 10 presenta su estructura cuyos componentes son:

- Segment Routing Manager: Computa SPF de ECMP y genera las reglas de forwarding para los switches.
- ARP Handler: Se encarga de las consultas ARP. Contiene las tablas ARP de los routers que gestiona.
- ICMP Handler: Maneja las consultas ICMP dirigidas a los routers
- Generic IP Handler: Maneja los paquetes IP recibidos. Si el destino de los paquetes se encuentra dentro de las subredes de los routers, envia el paquete al host correspondiente. En caso contrario envia una consulta ARP al ARP Handler.

- Segment Routing Policy: Crea las políticas y asigna las reglas a las tablas de ACL de los routers. Si están asociadas a túneles, crea también los túneles.

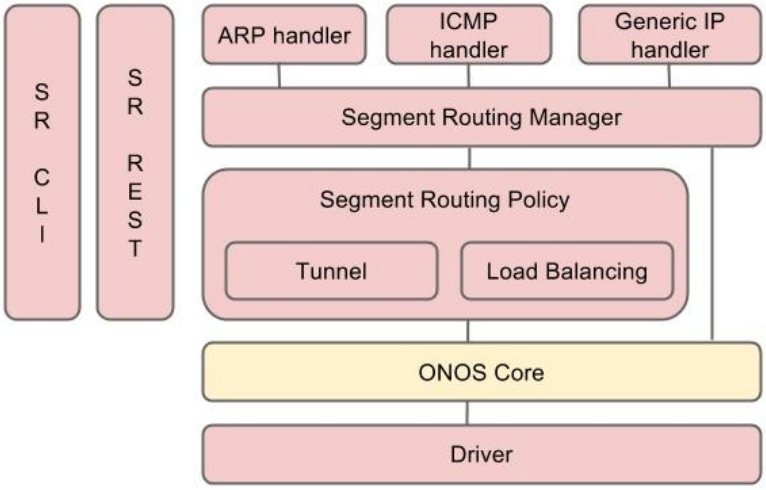


Figura 10: Arquitectura de Segment Routing en ONOS [10]

Un elemento importante de Segment Routing en Onos es el servicio de configuración, el cual se puede ver en la imagen 11. El mismo maneja la configuración necesaria para Segment Routing de todos los elementos de red. Se va a ver al momento de generar entornos de prueba que se necesita configurar los switches para que Onos aplique Segment Routing sobre ellos.

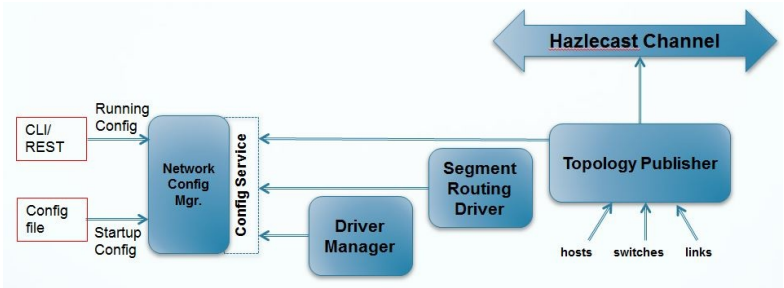


Figura 11: Servicio de configuración [10]

Trellis

Trellis es una suite de aplicaciónes SDN en el Datacenter que implementa una topología leaf-spine sobre *white switches*. Las aplicaciones corren sobre un controlador ONOS.

Trellis interconecta el Datacenter a otras redes, incluyendo Internet, mediante BGP. A su vez conecta el Datacenter a redes de acceso. Como se puede ver en la imagen 12, no se trata de una topología de Datacenter convencional, sino que Trellis se encuentra en el borde de la red interconectando los distintos componentes.

Este despliegue ya esta siendo utilizado por proveedores de servicio Tier 1 y es un caso de uso interesante, dado que utiliza puramente SDN tanto dentro del Datacenter como por fuera.

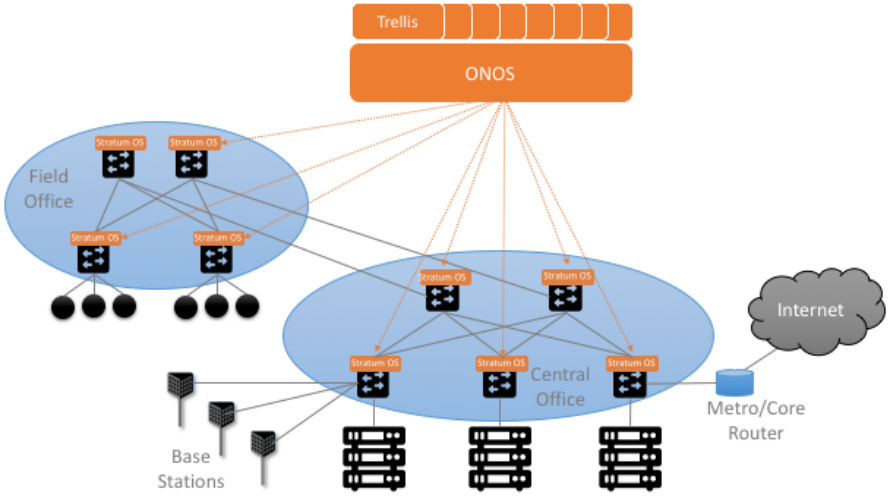


Figura 12: Despliegue de Trellis

3. Entorno de trabajo

Se utilizaron dos entornos de trabajo distintos para la implantación y prueba de funcionamiento de Segment Routing en Onos. En primer lugar se utilizo el entorno del proyecto SPRING-Open [11] propuesto por la Open Network Fundation (ONF [13]) el cual permitió comprender la aplicación y realizar pruebas que se detallarán en la sección 4. En segundo lugar se emuló un entorno de producción utilizando el framework Kathará [17] basado en contenedores.

3.1. SPRING-OPEN

El proyecto SPRING-OPEN presenta un ambiente que implementa Segment Routing utilizando una versión del controlador Onos interna de los desarrolladores, la cual no fue lanzada como una versión oficial. No es recomendable utilizar este entorno para un ambiente de producción pero es de gran utilidad para prototipos y pruebas funcionales. El entorno utiliza un cliente implementado en Python y está enfocado en el uso de la aplicación Segment Routing.

En cuanto a los dispositivos de red el proyecto permite tanto una implementación por Hardware utilizando switches Dell 4810 como switches virtuales utilizando Mininet [14] y OpenVSwitch CPqD con la versión de OpenFlow 1.3. Dado el alcance del proyecto y disponibilidad de recursos se utilizo el ambiente emulado.

El entorno se encuentra disponible incluido en una maquina virtual de Virtual Box con sistema operativo Ubuntu 14.04 incluyendo los elementos de software mencionados anteriormente instalados. Se le asignaron a la VM 2 CPU y 2GB de memoria RAM. Por simplicidad se utilizo dicha maquina virtual pero se pueden consultar las instrucciones de instalación de cada componente en [12].

3.1.1. Utilización del ambiente

El controlador debe se configurado antes de inicializar con un archivo de configuración de la topología que se desee utilizar. El proyecto incluye varias configuraciones de prueba, pero se puede crear una nueva siguiendo la sintaxis propuesta en [15]. La misma se debe ubicar en el directorio */spring-open/conf/*. Luego se debe editar el archivo *onos.properties* ubicado en dicho directorio cambiando el parámetro que se puede ver a continuación.

```
# Specify a network configuration file to be used
by the NetworkConfigManager
net.onrc.onos.core.configmanager.NetworkConfigManager.networkConfigFile
= conf/sr-3node.conf
```

Una vez configurado Onos se debe iniciar el controlador ejecutando el comando *./onos.sh start* dentro del directorio *spring-open*. Se puede verificar el estado del programa ejecutando */spring-open/onos.sh status*, una ejecución correcta debería mostrar una salida similar a la que se puede ver en la figura 13.

```
mininet@mininet-vm:~$ ~/spring-open/onos.sh status
[ZooKeeper]
JMX enabled by default
Using config: /home/mininet/spring-open/conf/zoo.cfg
Mode: standalone

[RAMCloud coordinator]
0 RAMCloud coordinator running

[RAMCloud server]
0 RAMCloud server running

[ONOS core]
1 instance of onos running
```

Figura 13: Estado de Controlador Onos correcto

Una vez iniciado el servicio se puede iniciar la topología en Mininet. El proyecto ofrece varios ejemplos, sin embargo en este caso se utilizo un script personalizado que genera topologías del tipo Spine-Leaf. Los scripts se ubican en el directorio */mininet/custom* y deben tener permiso de ejecución. En la figura 14 se puede ver un ejemplo de ejecución para una topología Spine-Leaf con dos Spine, dos Leaf y un servidor por Leaf. En la sección 4 se utilizaran otros ejemplos.

```
mininet@mininet-vm:~/mininet/custom$ sudo ./sr_2x2_spine_leaf-script.py
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(5ms delay) (5ms delay) (h1, s1) (5ms delay) (5ms delay) (h2, s2) (5ms delay) (5ms delay) (s1, s3) (5ms
elay) (5ms delay) (s1, s4) (5ms delay) (5ms delay) (s2, s3) (5ms delay) (5ms delay) (s2, s4)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 4 switches
s1 s2 s3 s4
*** Starting CLI:
mininet>
```

Figura 14: Ejecucion de script Mininet

Una vez iniciada la topología se puede acceder al cliente Python de la forma en que se muestra en la figura 15. Dicho cliente permite ver los dispositivos configurados y la información de ruteo, así como permite configurar túneles y políticas. La especificación de comandos disponibles se encuentra en [16].

```
mininet@mininet-vm:~$ cd spring-open-cli/
mininet@mininet-vm:~/spring-open-cli$ source ./workspace/ve/bin/activate
(ve)mininet@mininet-vm:~/spring-open-cli$ sudo make start-sdncon
if ! lsof -iTCP:8000 -sTCP:LISTEN >/dev/null; then \
        ( \
          cd /home/mininet/spring-open-cli/sdncon; \
          [ -d /home/mininet/spring-open-cli/workspace/ve/cassandra/data/sdncon ] || python manage.py syncdb --noinput; \
          /home/mininet/spring-open-cli/build/start-and-wait-for-port.sh -p 8000 -l /home/mininet/spring-open-cli/workspa
ce/ve/log/sdncon.log python manage.py runserver 0.0.0.0:8000; \
        ); \
        fi
+++ sdncon running
(ve)mininet@mininet-vm:~/spring-open-cli$ cd cli/
(ve)mininet@mininet-vm:~/spring-open-cli/cli$ ./cli.py
version200
default controller: 127.0.0.1:8000, SDN OS 1.0 - custom version
mininet-vm> sh router
# Router DPID             Router Name Router IP       Router Mac        Edge Router Node SId
-|----------------------|-----------|---------------|-----------------|-----------|--------
1 00:00:00:00:00:00:00:01 Leaf 1      172.16.101.1/32 00:00:00:00:00:01 true        101
2 00:00:00:00:00:00:00:02 Leaf 2      172.16.102.1/32 00:00:00:00:00:02 true        102
3 00:00:00:00:00:00:00:03 Spine 1     172.16.103.1/32 00:00:00:00:00:03 false       103
4 00:00:00:00:00:00:00:04 Spine 2     172.16.104.1/32 00:00:00:00:00:04 false       104
mininet-vm>
```

Figura 15: Conexión a cliente de Onos

Una vez realizados los pasos anteriores el ambiente esta pronto para ser utilizado. Se profundizara sobre dicho uso en la sección 4 con ejemplos concretos.

3.2. Kathará

Como se menciono anteriormente el proyecto Sping-Open no esta pensado para utilizarse en producción. Por dicho motivo se busco generar un escenario en el cual

se emulara un ambiente de producción para probar la aplicación Segment Routing de Onos y como se comporta en 'el mundo real'.

Se dispone además de un conjunto de pruebas para algoritmos distribuidos en topologías Fat Tree realizadas en Kathará (BGP, Open Fabric y RIFT), por lo cual implementando la solución en este entorno se podrían realizar comparaciones de performance, mensajería y otros parametros de interés. Por esta razón se decidió implantar Segment Routing sobre SDN en esta herramienta.

El framework Kathará permite emular escenarios de red utilizando Docker. El modelo se compone de tres conceptos base: El dispositivo, el dominio de colisión y el escenario de red.

- Un dispositivo es un componente virtual que actúa como un dispositivo de red (un router, servidor o switch por ejemplo). Tiene una o más interfaces de red, CPU, memoria, disco virtual y corre un sistema operativo. Se implementan con contenedores Docker.
- Un dominio de colisión es una red virtual de capa dos actuando como conexión física entre los dispositivos.
- Un escenario de red es un conjunto de dispositivos conectados por dominios de colisión. Se representa mediante un directorio que contiene la configuración de la topología y de los distintos dispositivos.

Este modelo y el hecho de utilizar contenedores Docker hace que el ambiente pueda pasar a un entorno de producción de una forma directa, salvando la configuración de red que se deba realizar. La arquitectura de Kathara así como su comparación con otros emuladores se puede consultar en [21].

El entorno se ejecutó sobre un host Ubuntu 20.04 64 bits con 8 GB de memoria RAM y CPU Intel Core i5. Por detalles sobre su instalación y dependencias se puede consultar [19]. El framework se encuentra disponible para sistemas operativos Linux basados en Debian, Windows y MacOSX por el momento.

3.2.1. Generación de Topologías Spine-Leaf

Para realizar pruebas y permitir mayor flexibilidad en el ambiente se desarrolló una herramienta que genera una topología de tipo Spine-Leaf y permite

ingresar la cantidad de Spines, Leafs y Servidores por Leaf mediante linea de comando o un archivo de configuración *config.json*. Al ejecutar la herramienta se generan los archivos de configuración necesarios para Kathará y ademas se genera un archivo *netconf.json* que contiene la configuración que luego utiliza Onos para configurar a los Switches con Segment Routing.

Se utilizó como base un generador de topologías Fat-Tree para los protocolos RIFT, BGP y OpenFabric. Dicho generador esta desarrollado en Python por el grupo de desarrolladores de Kathará.

Para el presente trabajo específicamente se desarrollaron las clases *Spine_Leaf* (definida en *Kathara/model/SR_Onos.py*) la cual genera y conecta los objetos de la topología, *Controller* (definida en */Kathara/model/node_types/Controller.py*) para definir el objeto Controlador y *SrConfigurator* (definida en */Kathara/protocol/sr/SrConfigurator.py*) que configura los archivos *lab.cong* y startup para el entorno.

Se modificaron luego aquellos parámetros necesarios para el ambiente y se creo la función *sr_generate_netconf_onos* dentro de *utils.py* para generar el archivo de configuración *netconf.json* de Onos.

3.2.2. Controlador Onos

El controlador Onos se define dentro del ambiente como un dispositivo de la red el cual utiliza la imagen *onos/kathara* y esta conectado a los Spine y Leaf mediante dominios de colisión en lo que denomino red de gestión, la cual no interfiere en el plano de datos, utilizando la subred *192.168.0.0/16* para este propósito.

Cada una de sus interfaces esta conectada a un dispositivo que ejecuta OpenvSwitch. Además de tener una interfaz a la cual se puede acceder desde el host que ejecuta Kathará.

La imagen de Docker utilizada se basa en la imagen modificada en [22]. Esta imagen se crea a partir de la versión de onos 1.15 y modifica las aplicaciones para iniciarlas por defecto, así como permite conexión entrante para los puertos necesarios.

En el archivo *Dockerfile* dentro del directorio *Onos* en [18] se encuentra la configuración de dicha imagen. Se habilitaron en este caso las siguientes aplicaciones de Onos:

- Openflow: Necesaria para control de los dispositivos.
- Net Config Host Provider: Permite configurar los Host a partir de un archivo de configuración.
- Route Service: Necesario para que Onos calcule las rutas.
- Segment Routing: Habilita el uso de la aplicación en los switches configurados para este propósito.

Se agrego además tcpdump para analizar los paquetes Openflow entre el controlador y los nodos.

Se encontró que la versión de Onos utilizada no permite Ingeniería de tráfico en Segment Routing al no contener los comandos en la CLI necesarios para crear túneles y políticas. Por lo cual no fue posible realizar pruebas de dichas funcionalidades.

3.2.3. Dispositivos OpenvSwitch

Para implementar los Spine y Leaf como switches OpenVSwitch se utilizo la imagen base de Kathará y se le agrego OpenVswitch instalando por *apt* los paquetes *openvswitch-common* y *openvswitch-switch*. La imagen docker se definió como *kathara/ovs* y se puede encontrar en *Openvswitch/Dockerfile* en el repositorio [18].

Es necesario iniciar el servicio al iniciar la topología, por lo cual en los archivos de configuración de los nodos OpenvSwitch se ejecuta el comando:

```
/usr/share/openvswitch/scripts/ovs-ctl start
```

Luego cada nodo ovs se debe configurar para comunicarse con el controlador y se deben configurar sus puertos. Para ello se configuró en cada archivo startup (el cual contiene la configuración de inicio de los dispositivos en Kathará) de los nodos Spine y Leaf con los siguientes comandos:

- Creación de bridge, se especifica dirección MAC de forma que coincida con la dirección del archivo *netconfig.json* de Onos. Por convención a su vez cada bridge lleva el nombre del nodo.

  ```
  ovs-vsctl add-br nombre_nodo -- set bridge nombre_nodo
  other-config:hwaddr=\"mac_addr\"
  ```

- Si bien no es una configuración de ovs, se configura una ip al bridge para que sea la IP de loopback en Segment Routing. Esta IP coincide con la del archivo *netconfig.json*, luego se levanta la interfaz.

```
ip addr ip_nodo dev nombre_nodo
ip link set nombre_nodo up
```

- Se configura id del datapath, con lo cual Onos va a identificar el Switch. Si no se especifica genera un ID aleatorio, pero es necesario en este caso generarlo para que coincida con el archivo *netconf.json*.

```
ovs-vsctl set bridge nombre_nodo other-config:datapath-id=id_nodo
```

- Indicar la direccion IP y puerto del controlador. En este caso se indica la IP del host que contiene Onos en la interfaz de control y el puerto 6653.

```
ovs-vsctl set-controller nombre_nodo tcp:ip_onos:6653
```

- Configuración de puertos:

```
ovs-vsctl add-port nombre_nodo interfaz
```

  En el caso de los Leaf se debe agregar una VLAN des-etiquetada en los puertos conectados a servidores:

```
ovs-vsctl add-port nombre_nodo interfaz tag=1
vlan_mode=native-untagged
```

Una vez ejecutadas las configuraciones al inicio, los switches pueden utilizar el protocolo Openflow para comunicarse con el controlador. También le enviaran información utilizando el protocolo LLDP (Link Layer Discovery Protocol).

3.2.4. Utilización del ambiente

Para generar la topología primero se debe editar el archivo *conf.json* que se encuentra en el directorio */Kathara* del proyecto [18]. En el ejemplo siguiente *k_leaf* representa la cantidad de Spines, *k_top* la cantidad de Leafs (dado que dicha configuracion parte del generador de Fat-Tree), *redundancy_factor* representa la cantidad de controladores Onos (por el momento solo se puede utilizar uno),

servers_for_rack la cantidad de servidores conectados a cada Leaf y *protocol* el protocolo utilizado. En este caso *sr* representa que se va a utilizar la aplicación Segment Routing de Onos.

```
{
  "k_leaf": 2,
  "k_top": 2,
  "redundancy_factor": 1,
  "servers_for_rack": 2,
  "protocol": "sr"
}
```

Luego se debe ejecutar el generador con el comando *./main.py* en dicho directorio. Esto generara un nuevo directorio cuyo nombre varia según la topología. En este ejemplo genera un directorio llamado *spine_leaf_2_2_1_sr*. Este directorio contiene dentro un archivo *topology_info.json* que contiene la configuración del laboratorio (igual al archivo *config.json*), el archivo de configuracion de Onos mencionado anteriormente *netconf.json* y un directorio *lab* donde se encuentra propiamente la configuración del laboratorio.

Dentro de *lab* se encuentra el archivo *lab.conf* que contiene la configuración del escenario de red. Se definen parámetros como que imagen de Docker van a utilizar los dispositivos, la conexión entre los mismos y si algún dispositivo es *bridged*, es decir que tiene una interfaz conectada al bridge de Docker, el cual se puede comunicar con el host.

Particularmente en este ambiente el controlador utiliza la imagen *onos/kathara*, los Leaf y los Spine utilizan la imagen *kathara/ovs*. Los servidores no especifican la imagen por lo cual por defecto usan la imagen *kathara/base*. Además el controlador tiene una interfaz bridge para poder acceder a la interfaz grafica y el cliente desde el host.

En el directorio *lab* también se encuentran los archivos *startup* de cada nodo del escenario, los cuales contienen comandos que los dispositivos ejecutan al iniciar el laboratorio. Entre ellos la configuración de interfaces de red, inicio de servicios y configuración de los mismos como en el caso de OpenVSwitch ya especificado.

Para iniciar el laboratorio se debe ejecutar el comando *sudo kathara lstart –privileged*. En ese momento se genera el directorio *shared* el cual esta presente en

todos los dispositivos y permite compartir archivos entre estos y el host.

Una vez iniciado el laboratorio se puede ingresar a la terminal de cada nodo con el comando *kathara connect nombre_nodo*. Por ejemplo para ver el log de Onos se puede acceder ejecutando *kathara connect controller* y luego dentro del controlador ejecutar *tail -f apache-karaf-3.0.8/data/log/karaf.log*.

Una vez iniciado Onos se puede acceder mediante su interfaz gráfica. Se debe identificar la ip por defecto para el bridge docker en el host, por mas información consultar [20]. La interfaz por defecto en linux tiene la ip *172.17.0.2* y la interfaz web de Onos en este caso es *http://172.17.0.2:8181/onos/ui*. Se puede acceder también al cli de Onos por SSH al puerto 8101. En ambos casos las credenciales de acceso son usuario *onos* y clave *rocks*.

Para configurar la topología en Onos se debe enviar el archivo *netconf.json* a la API REST de Onos mediante el metodo POST. En este caso se utilizó el comando *curl* para dicho propósito. Esto se debe realizar luego de iniciada la API, para comprobar si ya inicio se puede ver el log de Onos, pero alcanza con comprobar que se puede acceder a la interfaz web. Se ejecuta el comando:

```
curl --user --onos:rocks -X POST -H 'Content-Type: application/json'
http://172.17.0.2:8181/onos/v1/network/configuration/ -d @netconf.json
```

Una vez configurado Segment Routing se realizaron las pruebas detalladas en la sección 4. El ambiente dispone de herramientas como Ping, treceroute, ifconfig y tcpdump para analizar la red. Además los servidores tienen el servicio apache inicializado esperando conexiones en el puerto 80.

Para finalizar y liberar los recursos se debe ejecutar *kathara lclean*. En caso de que el entorno no finalice correctamente se debe ejecutar *kathara wipe* para destruir los objetos.

4. Pruebas Realizadas

Se presenta en esta sección el resultado de las pruebas realizadas en ambos ambientes.

Se utilizaron como ejemplo dos topologías del tipo Spine Leaf de distintos tamaños. La primera consiste en dos Spine, dos Leaf y dos servidores por Leaf, lo que llamaré como 2x2x2. La segunda topología permite simular un entorno más cercano al que se utilizaría en producción, dentro de los recursos que se pueden disponer en este caso. Comprende 4 Spines, 4 Leafs y 1 servidor por cada Leaf a lo que llamaré 4x4x1.

Se tienen diferencias en el diseño de las soluciones dado las posibilidades que cada ambiente permite, como configuración de direcciones IP y MAC, el control que se tiene sobre los dispositivos, los comandos disponibles y las herramientas a ejecutar.

4.1. Spring-Open Mininet

Se plantean para el entorno de Mininet dos topologías. En primer lugar la topología 2x2x2 que se puede ver en la figura 16 definida en los archivos *sr-spine_leaf_2x2x2-controller.conf* para configuración de Onos, *sr_spine_leaf_2x2x2_topo.py* para creación de la topología y *sr_2x2x2_spine_leaf-script.py* para ejecución del script.

En la imagen 16 se pueden ver los nombres de cada nodo junto con su identificador de segmento (Node SID) en el caso de los Leaf y Spine. A su vez se tiene una subred de prefijo 24 para cada servidor, dado que el ambiente no permite tener más de un puerto en el mismo dominio de broadcast (si bien el prefijo podría ser mas específico se decidió este valor para hacer mas sencilla la numeración).

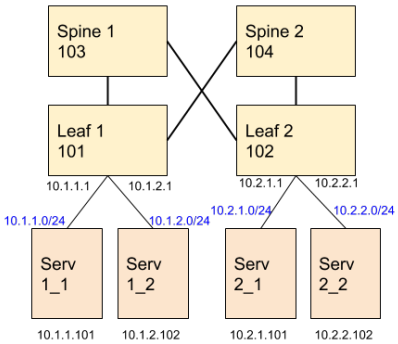


Figura 16: Spine-Leaf 2x2x2

En la imagen 17 por otro lado se tiene una topología 4x4x1. La misma se define en *sr-spine_leaf_4x4-controller.conf* para Onos, *sr_spine_leaf_4x4_topo.py* para la creación de la topología y *sr_4x4_spine_leaf-script.py* para el script de Mininet.

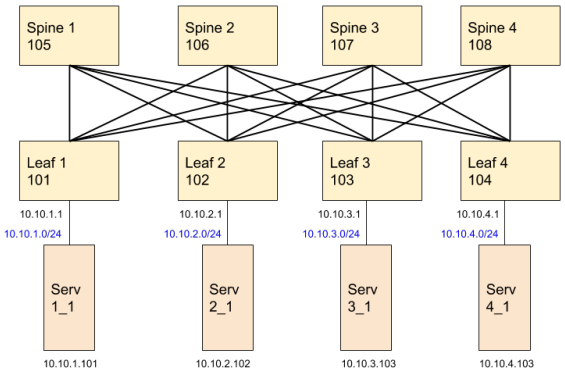


Figura 17: Spine-Leaf 4x4x1

4.1.1. Conectividad

Spine Leaf 2x2x2

En primer lugar se inicio la topología de la figura 16 y se probo la conectividad entre todos los host con el comando *pingall* de Mininet. En este ambiente los host se reconocen al intentar comunicarse con un router, por lo que esta acción ping es necesario para descubirlos. Luego se ejecuto un ping entre el servidor h_1_1 y el servidor h_2_2 como se puede ver en la figura 18.

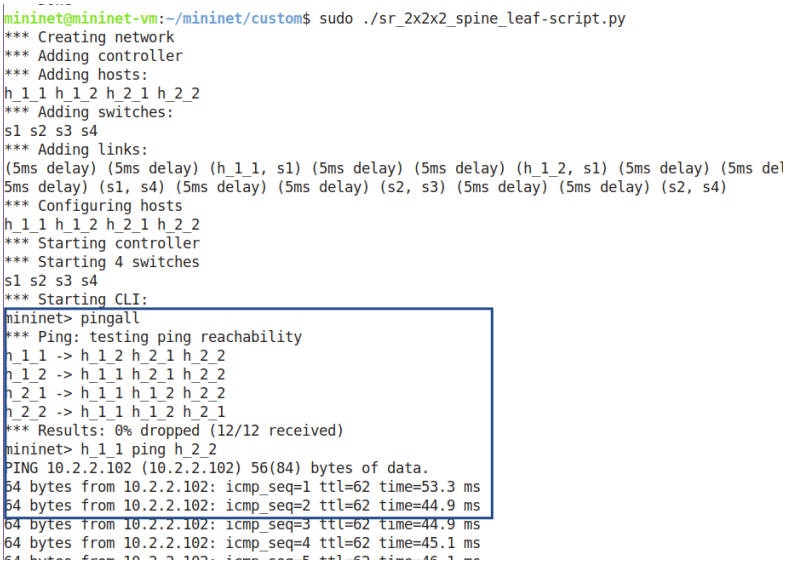


Figura 18: Salida de Mininet

En la figura 19 se muestra la información de los switch configurados como routers que ejecutan Segment Routing. Se puede ver el Datapath ID asignado en el archivo de configuración de Onos, el alias, las direcciones IP y MAC, la etiqueta MPLS asignada o NodeSid y el puerto con el cual reciben mensajes del controlador por la interfaz de gestión. En este ejemplo el Leaf 1 recibe los mensajes Openflow en el puerto 47151 de la interfaz de loopback de la maquina virtual.

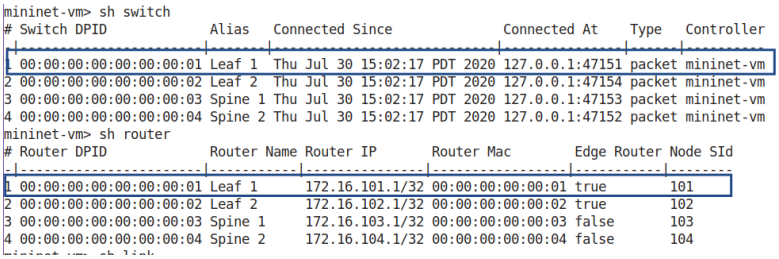


Figura 19: Informacion Switches Onos CLI

Para comprobar que el funcionamiento de Segment Routing es correcto se siguió el camino de los paquetes ICMP entre los dos servidores antes mencionados, así como las decisiones en cada router. En la figura 20 se marca la decisión que toma el Leaf 1 sobre los paquetes que tienen como ip de destino la subred *10.2.2.0/24*, a la cual pertenece el host h_2_2. Se indica que la instrucción a seguir se encuentra en

el grupo 2, el cual se puede ver en la imagen 21. Este grupo del tipo *Select* permite tomar dos acciones, agregando en ambos casos el SoD del Leaf 2 y enviando dicho paquete etiquetado a los dos Spine, lo cual se puede ver por la MAC indicada en la figura 19.

```
mininet-vm> sh sw 00:00:00:00:00:00:00:01 table ip
# Bytes Packets Dur(s) Cookie Priority Dst IP          Instructions
-|-----|-------|------|------|--------|---------------|-----------------------------------------------------------------------------------------------------------------
1 0     0       312    0      65535    172.16.102.1/32 {goto: {tableid: acl}, write: {group: 2}}
2 0     0       312    0      65535    172.16.103.1/32 {goto: {tableid: acl}, write: {dec_nw_ttl, group: 7}}
3 0     0       312    0      65535    172.16.104.1/32 {goto: {tableid: acl}, write: {dec_nw_ttl, group: 3}}
4 1862  19      308    0      65535    10.1.1.101/32   {goto: {tableid: acl}, write: {dst_mac: 00:00:00:00:01:02, src_mac: 00:00:00:00:00:01, output: 1}}
5 490   5       308    0      65535    10.1.2.102/32   {goto: {tableid: acl}, write: {dst_mac: 00:00:00:00:11:02, src_mac: 00:00:00:00:00:01, output: 2}}
6 392   4       312    0      49104    10.2.1.0/24     {goto: {tableid: acl}, write: {group: 2}}
7 1666  17      312    0      49104    10.2.2.0/24     {goto: {tableid: acl}, write: {group: 2}}
8 98    1       313    0      0        *               {goto: {tableid: acl}, write: {output: controller}}
```

Figura 20: Tabla IP de Leaf 1

```
mininet-vm> sh sw 00:00:00:00:00:00:00:01 group
#  Group Type Group Id Pkts Bytes Bucket Pkts Bucket Bytes Set Src Mac       Set Dst Mac       Push Mpls Set Bos COPY TTL Dec Mpls TTL Outport Group
--|----------|--------|----|-----|-----------|------------|-----------------|-----------------|---------|-------|--------|------------|-------|-----
1  SELECT     1        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03 102       true    True     True       3
2  SELECT     2        21   2058  11          1078         00:00:00:00:00:01 00:00:00:00:00:03 102       true    True     True       3
3  SELECT                21   2058  10          980          00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True       4
4  SELECT     3        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04                                      4
5  SELECT     4        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03 104       true    True     True       3
6  SELECT     5        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True       4
7  SELECT     6        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04 103       true    True     True       4
8  SELECT     7        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03                                      3
9  SELECT     8        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03                                      3
10 SELECT     _        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04                                      4
```

Figura 21: Grupos de Leaf 1

En las figuras 22 y 23 se pueden ver las tablas de flujo MPLS de los Spine 1 y 2 respectivamente. En ambos casos se toma la entrada en la cual la etiqueta 102 esta al final del Stack MPLS y la envían a sus respectivos grupos, realizan la operación POP y envían el paquete ya desetiquetado al Leaf 2.

```
mininet-vm> sh sw 00:00:00:00:00:00:00:03 table mpls
#  Bytes Packets Dur(s) Priority MPLS Label MPLS BOS MPLS TC Instructions
--|-----|-------|------|--------|----------|--------|-------|-------------------------------------------------------------------------------------------------------------------
-------------
1  1122  11      3859   65535    101        true     *       {goto: {tableid: acl}, write: {pop_mpls: 0x800, dec_nw_ttl, copy_ttl_in, group: 2}}
2  0     0       3859   65535    101        false    *       {goto: {tableid: acl}, write: {pop_mpls: 0x8847, copy_ttl_in, group: 2, dec_mpls_ttl}}
3  1122  11      3859   65535    102        true     *       {goto: {tableid: acl}, write: {pop_mpls: 0x800, dec_nw_ttl, copy_ttl_in, group: 1}}
4  0     0       3859   65535    102        false    *       {goto: {tableid: acl}, write: {pop_mpls: 0x8847, copy_ttl_in, group: 1, dec_mpls_ttl}}
5  0     0       3859   65535    104        true     *       {goto: {tableid: acl}, write: {group: 3, dec_mpls_ttl}}
6  0     0       3859   65535    104        false    *       {goto: {tableid: acl}, write: {group: 3, dec_mpls_ttl}}
7  0     0       3859   65535    103001     true     *       {goto: {tableid: acl}, write: {pop_mpls: 0x800, dst_mac: 00:00:00:00:00:01, dec_nw_ttl, src_mac: 00:00:00:00:00:03, output: 1, co
py_ttl_in}}
8  0     0       3859   65535    103001     false    *       {goto: {tableid: acl}, write: {dec_mpls_ttl, pop_mpls: 0x8847, dst_mac: 00:00:00:00:00:01, src_mac: 00:00:00:00:00:03, output: 1,
 copy_ttl_in}}
9  0     0       3859   65535    103002     true     *       {goto: {tableid: acl}, write: {pop_mpls: 0x800, dst_mac: 00:00:00:00:00:02, dec_nw_ttl, src_mac: 00:00:00:00:00:03, output: 2, co
py_ttl_in}}
10 0     0       3859   65535    103002     false    *       {goto: {tableid: acl}, write: {dec_mpls_ttl, pop_mpls: 0x8847, dst_mac: 00:00:00:00:00:02, src_mac: 00:00:00:00:00:03, output: 2,
 copy_ttl_in}}
11 0     0       3859   0        *          *        *       {goto: {tableid: acl}, write: {output: controller}}
mininet-vm> sh sw 00:00:00:00:00:00:00:03 group 1
# Group Type Group Id Pkts Bytes Bucket Pkts Bucket Bytes Set Src Mac       Set Dst Mac       Push Mpls Set Bos COPY TTL Dec Mpls TTL Outport Group
-|-------|--------|----|-----|-----------|------------|-----------------|-----------------|---------|-------|--------|------------|-------|-----
1 SELECT    1        11   1078  11          1078         00:00:00:00:00:03 00:00:00:00:00:02                                      2
mininet-vm>
```

Figura 22: Tabla MPLS de Spine 1

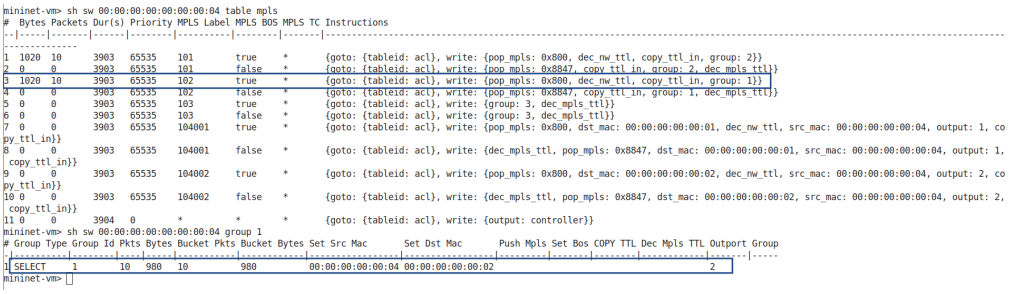


Figura 23: Tabla MPLS de Spine 2

Una vez el paquete ICMP llega al Leaf 2, el mismo decide segun su tabla IP. Dado que el servidor h_2_2 esta directamente conectado lo envía por el puerto correspondiente. Luego el Leaf 2 responde con un paquete ICMP Echo Reply cuyo flujo es análogo al visto. Se pueden ver por más información las tablas MPLS e IP de cada router en el directorio *Pruebas/Conectividad/Mininet/2x2x2* de [18].

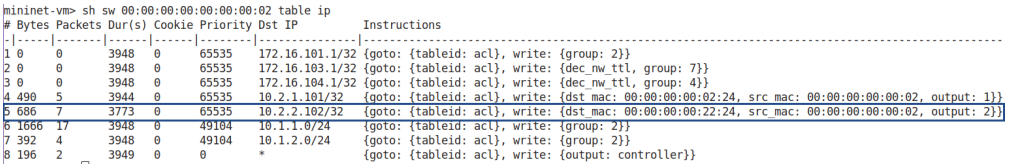


Figura 24: Tabla IP de Leaf 2

Se inspeccionaron los paquetes con *tcpdump* en las interfaces que conectan al Leaf 1 con los dos Spine para detectar si se utiliza ECMP y para verificar los paquetes MPLS. Se puede ver en las figuras 25 y 26 se transmiten paquetes y el tráfico desde el servidor h_1_1 al servidor h_2_2 contiene la etiqueta MPLS 102 del Leaf 2. Se comprobó por lo tanto que el funcionamiento en esta topología es el esperado para el algoritmo Segment Routing.

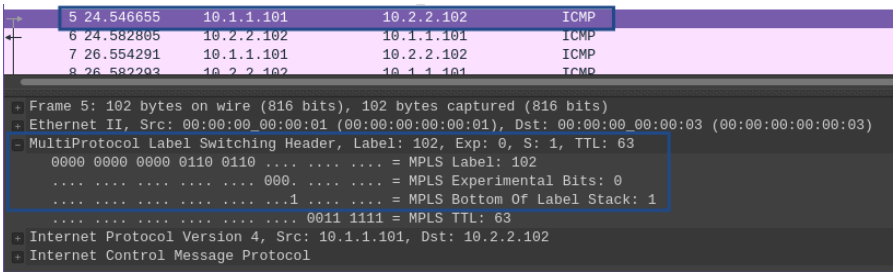


Figura 25: Paquete MPLS de Leaf 1 hacia Spine 1

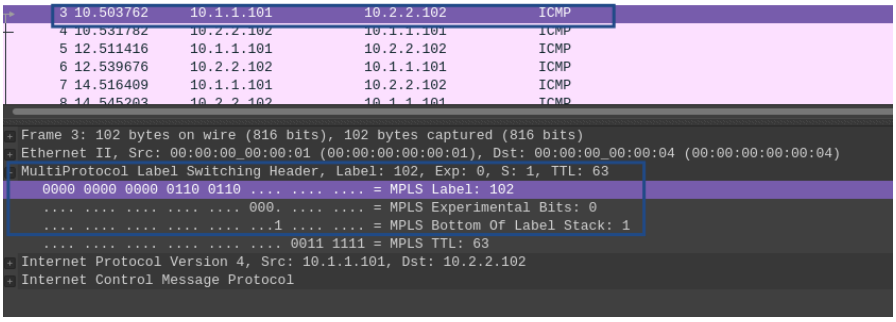


Figura 26: Paquete MPLS de Leaf 1 hacia Spine 2

Spine Leaf 4x4x1

Se probó por otro lado la topología de la figura 17. En este caso se ejecuto el comando *pingall* y se siguieron los flujos en las tablas de los respectivos nodos. En particular se inspeccionaron las interfaces del Leaf 1 hacia cada Spine y se comprobó que los paquetes estaban etiquetados correctamente. Las capturas y las tablas de cada nodo se pueden observar con más detalle en el directorio *Pruebas/-Conectividad/Mininet/4x4x1*. En la figura 27 se puede ver la tabla ip del Leaf 1 y sus respectivos grupos. En cada caso se envía a los cuatro Spine los paquetes con la etiqueta correspondiente utilizando ECMP.

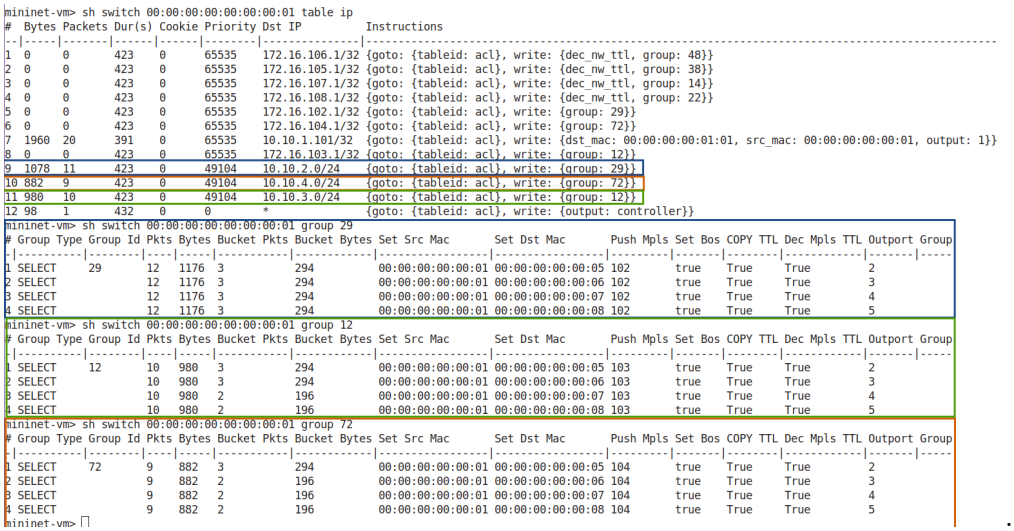


Figura 27: Tablas y Grupos de Leaf 1

Plano de Control: OpenFlow

Se inspeccionó para la topología de la figura 16 con *tcpdump* en la interfaz *localhost* la comunicación a nivel de plano de control de los nodos, siguiendo el flujo de los paquetes Openflow. Según se puede ver en la imagen 19 el Leaf 1 recibe los mensajes del controlador en el puerto 47151.

En las figuras 28 y 29 se puede ver el intercambio de mensajes *OFPT_HELLO* con los cuales se establece la conexión, especificando la versión de OpenFlow a utilizar. Como se puede observar por los puertos de destino y origen del paquete TCP la conexión es iniciada por el Switch.

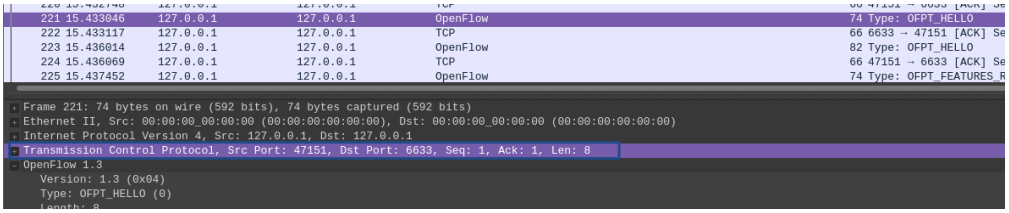


Figura 28: OpenFlow Hello de Leaf 1 hacia Onos

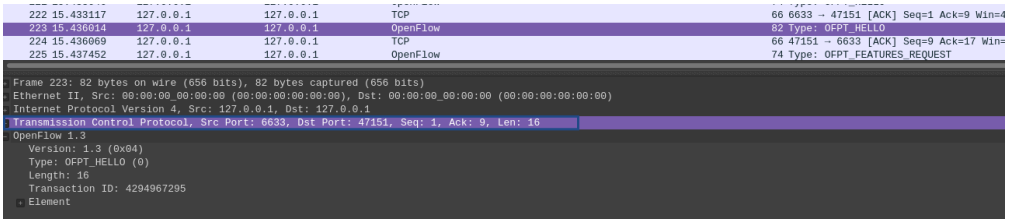


Figura 29: OpenFlow Hello de Onos hacia Leaf 1

Luego de establecerse la conexión, como se puede ver en la figura 30, el controlador envía al switch un mensaje *OFPT_FEATURES_REQUEST* el cual recibe como respuesta en la imagen 30 información del switch, como su Datapath, el número de tablas y las funcionalidades que permite el mismo.

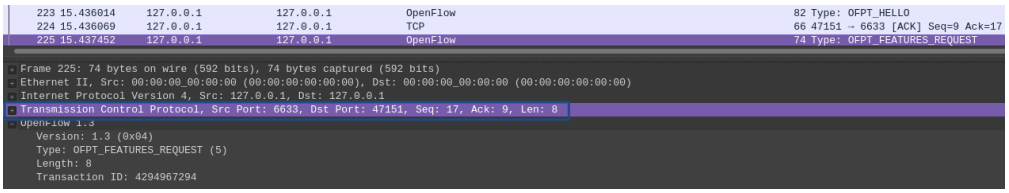


Figura 30: OpenFlow Features Request de Onos hacia Leaf 1

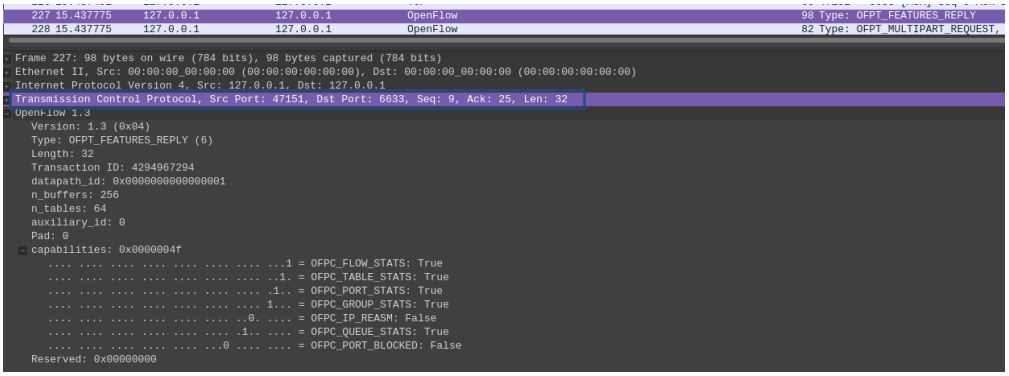


Figura 31: OpenFlow Features Reply de Leaf 1 a Onos

Después el controlador solicita al Leaf 1 que envíe la descripción de sus puertos, con un mensaje *OFPT_MULTIPART_REQUEST* del tipo *OFMP_PORT_DESC* (figura 32). El Leaf retorna la información de sus puertos como se puede ver en la imagen 33, en el cual se indican el número de cada puerto, su dirección MAC y configuración entre otros datos.

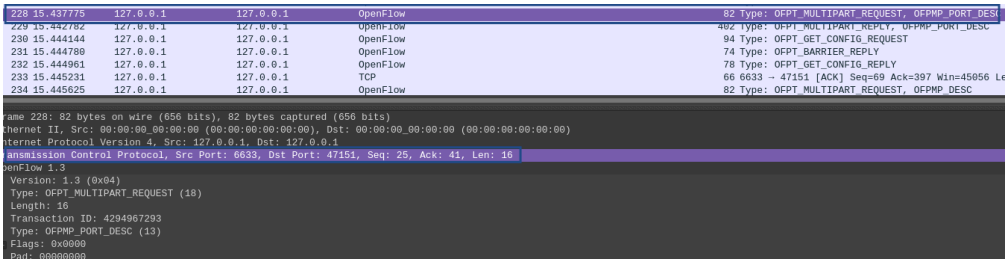


Figura 32: OpenFlow Multipart Request de Onos a Leaf 1

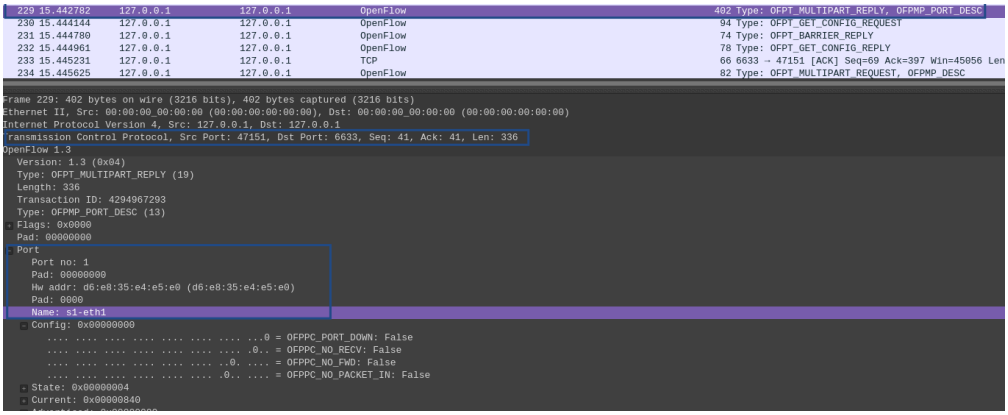


Figura 33: OpenFlow Multipart Reply Port Description de Leaf 1 a Onos

En la figura 34 se puede observar que el controlador envía un mensaje *OFPT_FLOW_MOD* con el cual modifica los flujos del Leaf, instalando los flujos generados por Segment Routing. Luego en la figura 35 se modifican los grupos del Leaf con el mensaje *OFPT_GROUP_MOD*, con las acciones a tomar para cada grupo.

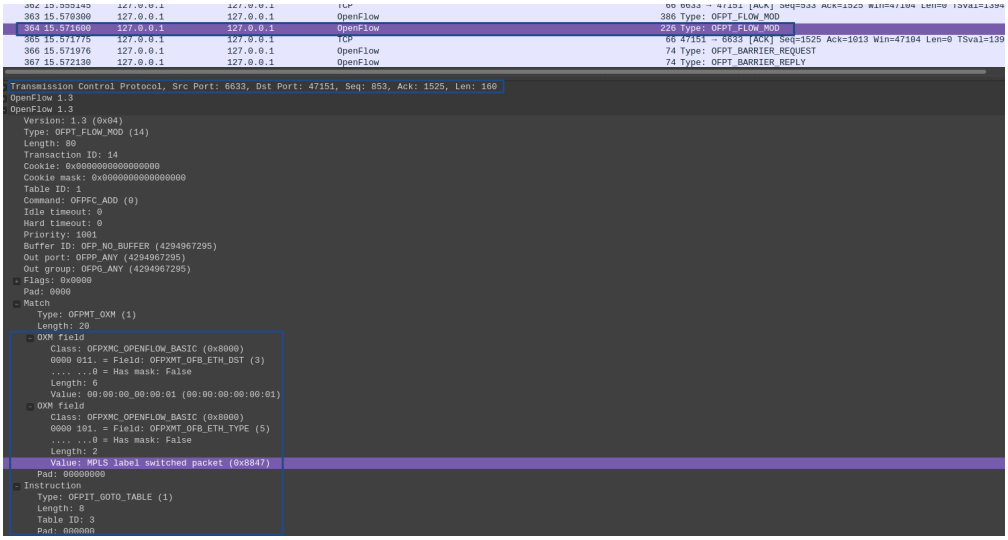


Figura 34: OpenFlow Flow Mod de Onos a Leaf 1

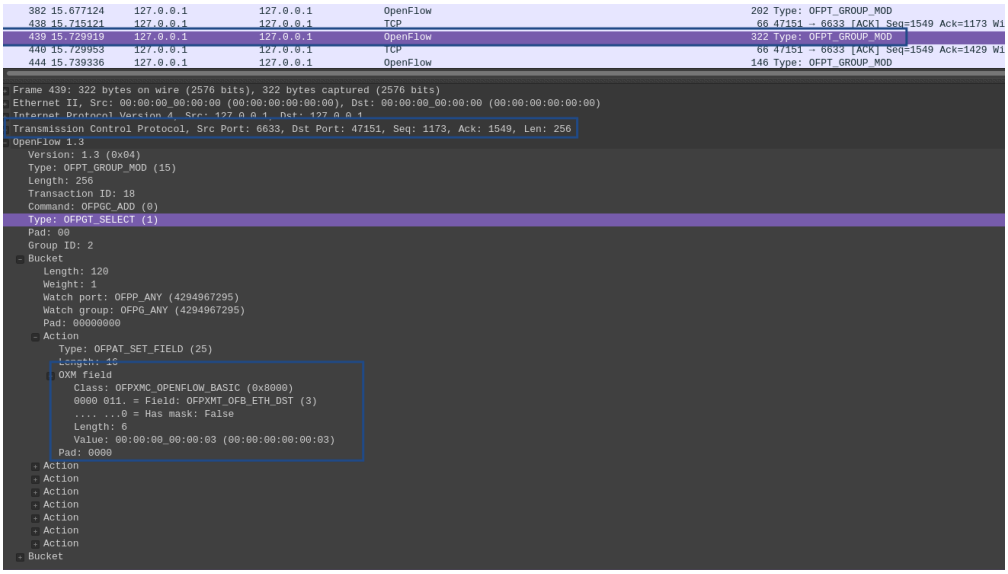


Figura 35: OpenFlow Group Mod de Onos a Leaf 1

Por último destacar los mensajes *OFPT_PACKET_IN* y *OFT_PACKET_OUT* de las figuras 36 y 37 al enviar mensajes ICMP y ARP. Inicialmente el controlador no conoce la ubicación de los host por lo cual el mensaje se dirige primero al controlador quien resuelve la consulta. Además, el controlador se encarga de responder los mensajes ARP dado que conoce la topología en su totalidad.

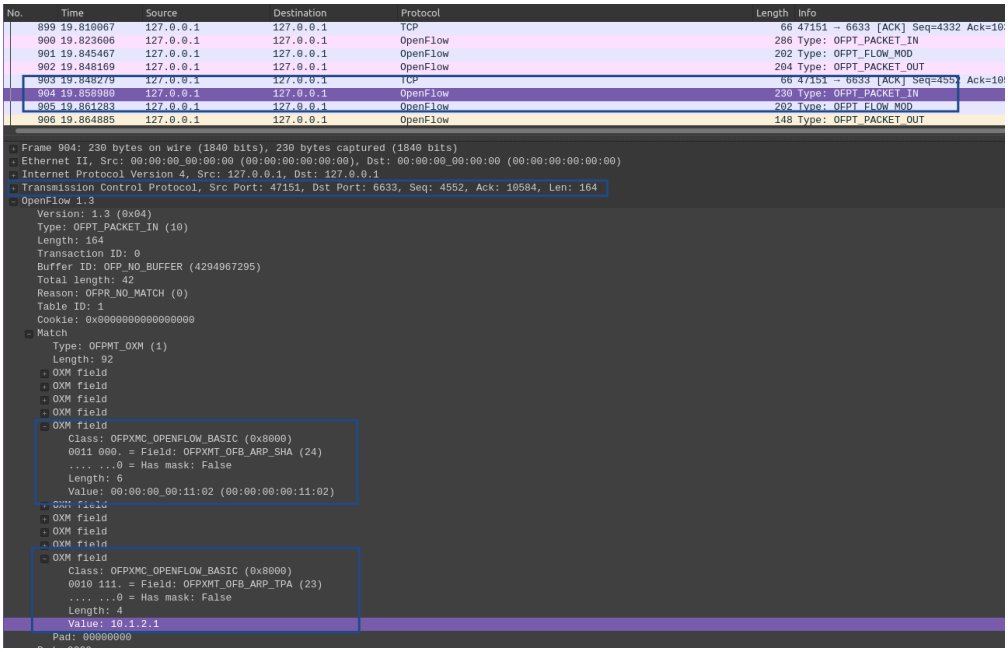


Figura 36: OpenFlow Packet IN ARP

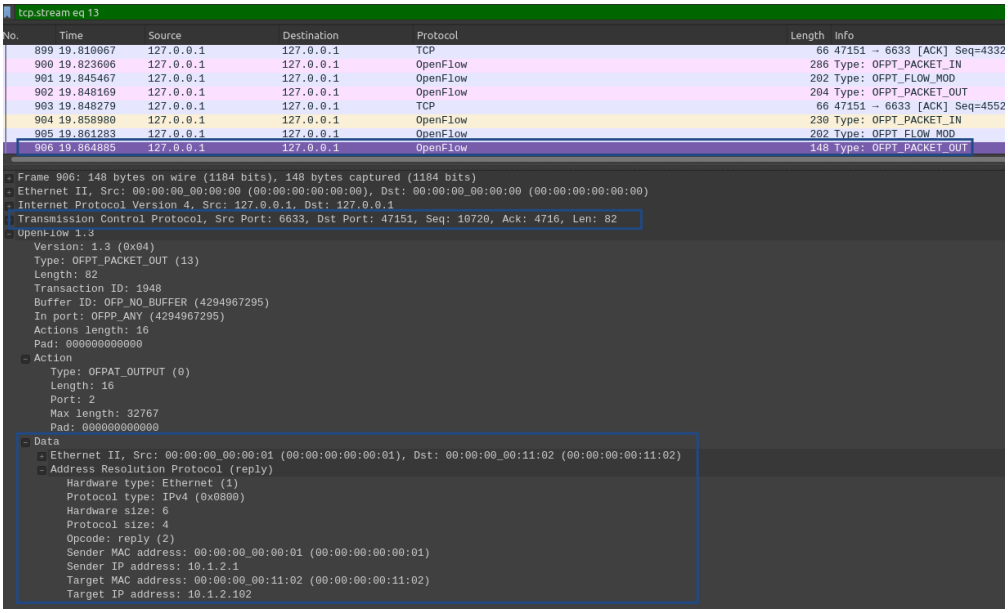


Figura 37: OpenFlow Packet Out ARP

4.1.2. Falla de Enlaces

Se probó que ocurre ante la caída de un link entre un Spine y un Leaf. La aplicación Segment Routing debería actualizar los flujos de los nodos para que no se pierda la conectividad siempre que se siga teniendo un camino posible entre dos servidores.

Spine Leaf 2x2x2

En el primer caso se bajó la interfaz que conecta al Spine 1 con el Leaf 2, como se puede ver en la figura 38, y se comprobó el cambio en el Leaf 1, el cual en teoría debería modificar su flujo para que se envíen los paquetes al Spine 2.

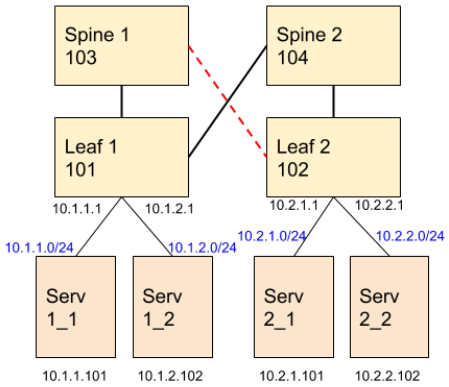


Figura 38: Falla de Link

En la imagen 39 se puede observar marcado en verde el estado de los flujos del Leaf 1 antes de bajar la interfaz y en azul luego de la falla. Se puede ver como en la tabla IP cambia el grupo 2, el cual envía el paquete MPLS con etiqueta 102 a ambos Spine, por el grupo 5 que solo envía el paquete MPLS al Spine 2. No se perdieron paquetes durante ni luego del cambio.

```
mininet-vm> sh sw 00:00:00:00:00:00:00:01 table ip
# Bytes Packets Dur(s) Cookie Priority Dst IP          Instructions
-|-----|-------|------|------|--------|---------------|-----------------------------------------------------------------------------------------------------------------
1 0     0       11     0      65535    172.16.102.1/32 {goto: {tableid: acl}, write: {group: 2}}
2 0     0       10     0      65535    172.16.103.1/32 {goto: {tableid: acl}, write: {dec_nw_ttl, group: 7}}
3 0     0       10     0      65535    172.16.104.1/32 {goto: {tableid: acl}, write: {dec_nw_ttl, group: 3}}
4 14994 153     159    0      65535    10.1.1.101/32   {goto: {tableid: acl}, write: {dst_mac: 00:00:00:00:01:02, src_mac: 00:00:00:00:00:01, output: 1}}
5 0     0       159    0      65535    10.1.2.102/32   {goto: {tableid: acl}, write: {dst_mac: 00:00:00:00:11:02, src_mac: 00:00:00:00:00:01, output: 2}}
6 0     0       11     0      49104    10.2.1.0/24     {goto: {tableid: acl}, write: {group: 2}}
7 1078  11      11     0      49104    10.2.2.0/24     {goto: {tableid: acl}, write: {group: 2}}
8 98    1       335    0      0        *               {goto: {tableid: acl}, write: {output: controller}}
mininet-vm> sh sw 00:00:00:00:00:00:00:01 group 2
# Group Type Group Id Pkts Bytes Bucket Pkts Bucket Bytes Set Src Mac       Set Dst Mac       Push Mpls Set Bos COPY TTL Dec Mpls TTL Outport Group
-|-----|--------|----|-----|-----|-----------|------------|-----------------|-----------------|---------|-------|--------|------------|-------|-----
1 SELECT     2        81   7938  41          4018         00:00:00:00:00:01 00:00:00:00:00:03 102       true    True     True         3
2 SELECT               81   7938  40          3920         00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True         4
mininet-vm> sh sw 00:00:00:00:00:00:00:01 table ip
# Bytes Packets Dur(s) Cookie Priority Dst IP          Instructions
-|-----|-------|------|------|--------|---------------|-----------------------------------------------------------------------------------------------------------------
1 0     0       3      0      65535    172.16.102.1/32 {goto: {tableid: acl}, write: {group: 5}}
2 0     0       3      0      65535    172.16.103.1/32 {goto: {tableid: acl}, write: {dec_nw_ttl, group: 7}}
3 0     0       3      0      65535    172.16.104.1/32 {goto: {tableid: acl}, write: {dec_nw_ttl, group: 3}}
4 17150 175     183    0      65535    10.1.1.101/32   {goto: {tableid: acl}, write: {dst_mac: 00:00:00:00:01:02, src_mac: 00:00:00:00:00:01, output: 1}}
5 0     0       183    0      65535    10.1.2.102/32   {goto: {tableid: acl}, write: {dst_mac: 00:00:00:00:11:02, src_mac: 00:00:00:00:00:01, output: 2}}
6 0     0       3      0      49104    10.2.1.0/24     {goto: {tableid: acl}, write: {group: 5}}
7 392   4       3      0      49104    10.2.2.0/24     {goto: {tableid: acl}, write: {group: 5}}
8 98    1       359    0      0        *               {goto: {tableid: acl}, write: {output: controller}}
mininet-vm> sh sw 00:00:00:00:00:00:00:01 group 5
# Group Type Group Id Pkts Bytes Bucket Pkts Bucket Bytes Set Src Mac       Set Dst Mac       Push Mpls Set Bos COPY TTL Dec Mpls TTL Outport Group
-|---------|--------|----|-----|-----------|------------|-----------------|-----------------|---------|-------|--------|------------|-------|-----
1 SELECT     5        101  9898  101         9898         00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True         4
```

Figura 39: Tabla de Leaf 1 antes y luego de la caida de Link

Spine Leaf 4x4x1

En este caso se desconectó el link entre el Spine 1 y el Leaf 3, como se puede observar en la imagen 40, y se ejecutó un ping desde el servidor 1 al servidor 3.

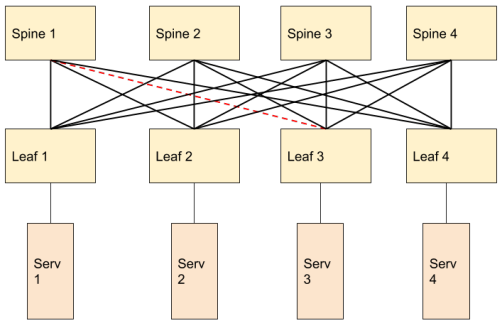


Figura 40: Falla de Link

En la tabla IP del Leaf 1 (figura 41) se puede observar que para la subred *10.10.3.0/24* cambió el grupo y los mensajes MPLS ya no se envían al Spine 1, comparando con la tabla IP de la figura 27 donde se enviaban los datos por todos

los Spine utilizando el grupo 12. No se perdieron paquetes durante ni luego del cambio.

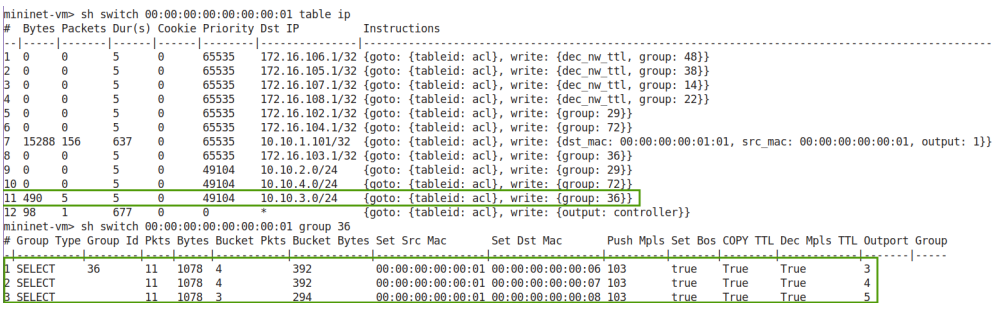


Figura 41: Tabla de Leaf 1 luego de la caída de Link

4.1.3. Prueba de Trafico TCP

La versión utilizada de Openvswitch en el caso de Mininet elije el next hop para el algoritmo ECMP paquete a paquete y no por flujo por lo cual los paquetes de un mismo flujo van a tomar distintos caminos en la red. Esto afecta a protocolos orientados a conexión como TCP dado que los paquetes pueden llegar desordenados y el protocolo puede interpretar perdida de paquetes por congestión en la red. Se puede profundizar sobre este problema en [23].

Por ese motivo se realizó una prueba levantando un servidor HTTP en un host y haciendo consultas desde otros, inspeccionando el comportamiento del protocolo de transporte.

Se utilizo en las pruebas un script de Mininet que levanta un servidor HTTP en el puerto 80 y se utilizo wget desde otro host.

```
mininet> h_2_2 python -m SimpleHTTPServer 80 &
mininet> h_1_1 wget -O - h_2_2
```

Spine Leaf 2x2x2

Las figuras 42 y 43 muestran la inspección de paquetes al momento de la consulta http entre el host h_{11} y el host h_{22} utilizando el camino por el Spine 1 y el Spine 2 respectivamente. Se pueden observar pedidos de retransmisión en ambos casos de los paquetes TCP lo cual afectó la comunicación entre ambos hosts.

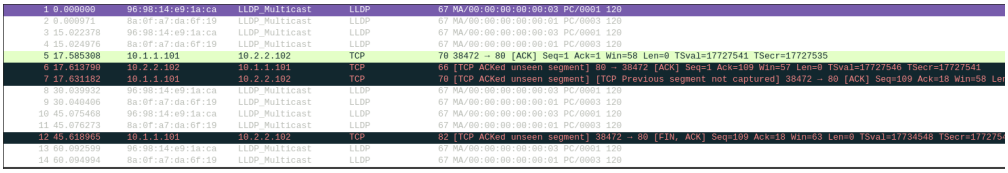


Figura 42: Trafico TCP entre Leaf 1 y Spine 1

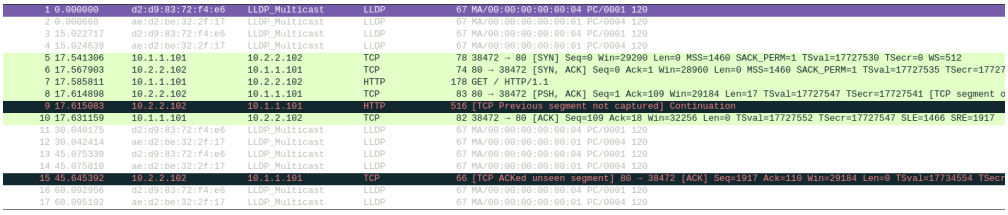


Figura 43: Trafico TCP entre Leaf 1 y Spine 2

Spine Leaf 4x4x1

En este caso se puede ver en las figuras 44 a 47 que el tráfico se distribuye entre los cuatro Spines y se observan también pedidos de retransmisión, además de tener paquetes TCP del mismo flujo yendo por distintos caminos.

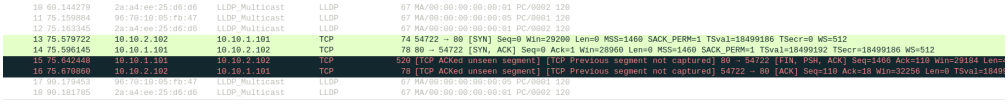


Figura 44: Trafico TCP entre Leaf 1 y Spine 1

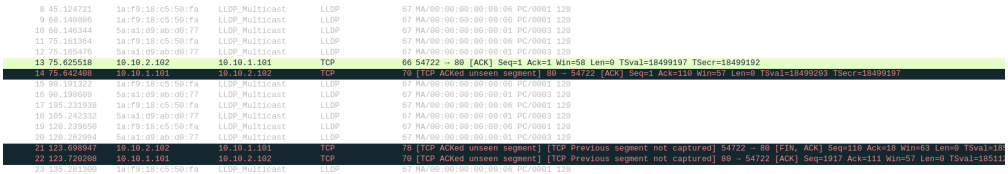


Figura 45: Trafico TCP entre Leaf 1 y Spine 2

Figura 46: Trafico TCP entre Leaf 1 y Spine 3

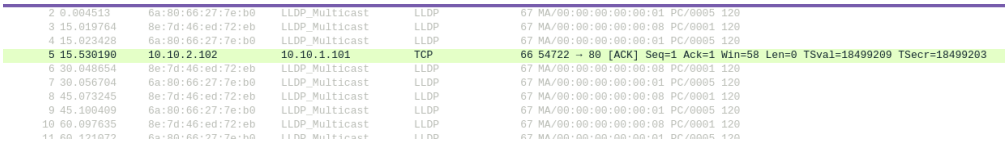


Figura 47: Trafico TCP entre Leaf 1 y Spine 4

Este problema es común en ECMP y en los algoritmos de distribución de carga en general. Es importante por lo tanto tener en cuenta este tipo de problemas cuando se utilizan protocolos sensibles a la conexión.

4.1.4. Ingeniería de Tráfico

Se realizaron pruebas introduciendo túneles y políticas aplicadas a los mismos para cambiar el comportamiento normal de ECMP con Shortest Path First por un comportamiento específico deseado.

Para ello se utilizo el cliente de Onos con la opción de configuración habilitada, se crearon los túneles necesarios y luego se le aplicaron políticas. Se comprobó el comportamiento del protocolo ejecutando un ping entre servidores a los cuales aplicaba alguna política.

La configuración de un túnel, por ejemplo el túnel *azul* de la figura 49 se realiza de la siguiente forma:

```
mininet-vm> enable
mininet-vm# configure
mininet-vm(config)# tunnel azul
mininet-vm(config-tunnel)# node 101
mininet-vm(config-tunnel)# node 103
mininet-vm(config-tunnel)# node 102
mininet-vm(config-tunnel)# exit
```

Para crear una política se configura una subred de origen, una subred de destino, una prioridad y el túnel al cual se asigna. Por ejemplo para la política *azul* de la figura 49 se ejecutaron los siguientes comandos.

```
mininet-vm(config)# policy azul policy-type tunnel-flow
mininet-vm(config-policy)# flow-entry ip 10.1.1.0/24 10.2.1.0/24
mininet-vm(config-policy)# tunnel azul
```

```
mininet-vm(config-policy)# priority 1000
mininet-vm(config-policy)# exit
```

Spine Leaf 2x2x2

Se crearon para esta topología cuatro túneles, para tener dos túneles en cada sentido y obligar a que todo el tráfico entre los servidores h_1_1 y h_2_1 vaya por el camino marcado en azul de la figura 48, mientras que el tráfico entre los servidores h_1_2 y h_2_2 vaya por el camino marcado en verde. Todo el tráfico no definido en las políticas debería utilizar el algoritmo SPF.

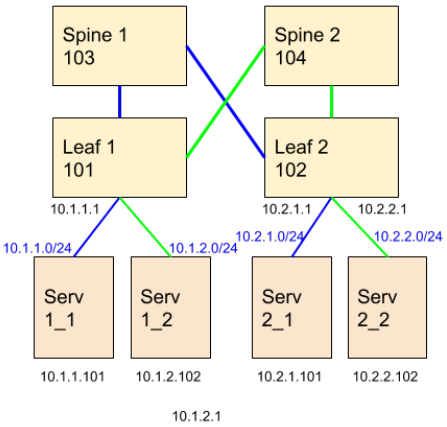


Figura 48: Ingeniería de trafico en Spine Leaf 2x2x2

En la figura 49 se pueden observar los túneles y políticas creados. Los túneles *azul* y *azul-reverso* marcan el camino azul de la figura 48 y se puede notar que el recorrido de etiquetas MPLS es el mismo pero en distintos sentidos. Lo mismo ocurre para los túneles *verde* y *verde-reverso* que marcan el camino en verde.

```
mininet-vm(config)# sh tunnel
# Id        Policies      Tunnel Path [Head-->Tail] Label Stack [Outer-->Inner]
-|--------|-------------|-------------------------|-----------------------
1 azul      azul          [101, 103, 102]          [[102]]
2 azul-rev  azul-reverso  [102, 103, 101]          [[101]]
3 verde     verde         [101, 104, 102]          [[102]]
4 verde-rev verde-reverso [102, 104, 101]          [[101]]
mininet-vm(config)# sh policy
# Policy Id     Policy Type Priority Dst Mac Src Mac Ether Type Dst IP      IP Protocol Src IP      Dst TcpPort Src TcpPort Tunnel Used
-|------------|-----------|--------|-------|-------|----------|-----------|-----------|----------|-----------|-----------|-----------
1 azul          TUNNEL_FLOW 1000     *       *       0x2048     10.2.1.0/24 *           10.1.1.0/24 *           *           azul
2 azul-reverso  TUNNEL_FLOW 1000     *       *       0x2048     10.1.1.0/24 *           10.2.1.0/24 *           *           azul-rev
3 verde         TUNNEL_FLOW 1000     *       *       0x2048     10.2.2.0/24 *           10.1.2.0/24 *           *           verde
4 verde-reverso TUNNEL_FLOW 1000     *       *       0x2048     10.1.2.0/24 *           10.2.2.0/24 *           *           verde-rev
```

Figura 49: Túneles y Políticas

Las figuras 50 y 51 muestran las tablas de flujo de ACL de los Leaf, junto con los grupos seleccionados en cada caso, marcando por su color el tunel correspondiente. Se puede ver como se aplica a cada Leaf la política correspondiente según el sentido de la conexión. Se ejecutaron ping entre los host h_1_1 y h_2_1 y entre los host h_1_2 y h_2_2 en los cuales se comprobó que el tráfico iba por el camino correcto y no se estaba utilizando SPF para estos casos.

```
mininet-vm(config)# sh sw 00:00:00:00:00:00:00:01 table acl
# Bytes Packets Dur(s) Cookie Priority In Port Src MAC Dst MAC EthType   Src IP     Dst IP      Protocol Src Port Dst Port Instructions
-|-----|-------|------|------|--------|-------|-------|-------|---------|----------|-----------|--------|--------|--------|------------
1 30968 316     670    0      1000     *       *       *       ip(0x800) 10.1.1.0/8 10.2.1.0/24 *        *        *        {clear_actions, write: {group: 11}}
2 5880  60      208    0      1000     *       *       *       ip(0x800) 10.1.2.0/8 10.2.2.0/24 *        *        *        {clear_actions, write: {group: 12}}
3 92318 935     1472   0      0        *       *       *       *         *          *           *        *        *
mininet-vm(config)# sh sw 00:00:00:00:00:00:00:01 group
#  Group Type Group Id Pkts Bytes Bucket Pkts Bucket Bytes Set Src Mac       Set Dst Mac       Push Mpls Set Bos COPY TTL Dec Mpls TTL Outport Group
--|----------|--------|----|-----|-----------|------------|-----------------|-----------------|---------|-------|--------|-----------|-------|-----
1  SELECT     1        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04 103       true    True     True        4
2  SELECT     2        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04                                     4
3  SELECT     3        172  16856 172         16856        00:00:00:00:00:01 00:00:00:00:00:03                                     3
4  SELECT     4        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True        4
5  SELECT     5        212  20776 106         10388        00:00:00:00:00:01 00:00:00:00:00:03 102       true    True     True        3
6  SELECT                212  20776 106         10388        00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True        4
7  SELECT     6        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03 104       true    True     True        3
8  SELECT     7        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03 102       true    True     True        3
9  SELECT     8        0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:03                                     3
10 SELECT                0    0     0           0            00:00:00:00:00:01 00:00:00:00:00:04                                     4
11 SELECT     11       316  30968 316         30968        00:00:00:00:00:01 00:00:00:00:00:03 102       true    True     True        3
12 SELECT     12       60   5880  60          5880         00:00:00:00:00:01 00:00:00:00:00:04 102       true    True     True        4
```

Figura 50: Tabla ACL de Leaf 1 y grupos correspondientes

```
mininet-vm(config)# sh sw 00:00:00:00:00:00:00:02 table acl
# Bytes  Packets Dur(s) Cookie Priority In Port Src MAC Dst MAC EthType   Src IP     Dst IP      Protocol Src Port Dst Port Instructions
-|------|-------|------|------|--------|-------|-------|-------|---------|----------|-----------|--------|--------|--------|------------
1 10388  106     475    0      1000     *       *       *       ip(0x800) 10.2.1.0/8 10.1.1.0/24 *        *        *        {clear_actions, write: {group: 10}}
2 5880   60      145    0      1000     *       *       *       ip(0x800) 10.2.2.0/8 10.1.2.0/24 *        *        *        {clear_actions, write: {group: 11}}
3 124950 1275    1488   0      0        *       *       *       *         *          *           *        *        *
mininet-vm(config)# sh sw 00:00:00:00:00:00:00:02 group
#  Group Type Group Id Pkts Bytes Bucket Pkts Bucket Bytes Set Src Mac       Set Dst Mac       Push Mpls Set Bos COPY TTL Dec Mpls TTL Outport Group
--|----------|--------|----|-----|-----------|------------|-----------------|-----------------|---------|-------|--------|-----------|-------|-----
1  SELECT     1        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:04 103       true    True     True        4
2  SELECT     2        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:04                                     4
3  SELECT     3        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:03                                     3
4  SELECT     4        381  37338 191         18718        00:00:00:00:00:02 00:00:00:00:00:03 101       true    True     True        3
5  SELECT                381  37338 190         18620        00:00:00:00:00:02 00:00:00:00:00:04 101       true    True     True        4
6  SELECT     5        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:03 104       true    True     True        3
7  SELECT     6        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:03 101       true    True     True        3
8  SELECT     7        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:04 101       true    True     True        4
9  SELECT     8        0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:03                                     3
10 SELECT                0    0     0           0            00:00:00:00:00:02 00:00:00:00:00:04                                     4
11 SELECT     10       106  10388 106         10388        00:00:00:00:00:02 00:00:00:00:00:03 101       true    True     True        3
12 SELECT     11       60   5880  60          5880         00:00:00:00:00:02 00:00:00:00:00:04 101       true    True     True        4
```

Figura 51: Tabla ACL de Leaf 2 y grupos correspondientes

Luego de ver el comportamiento de Segment Routing aplicando dichas políticas, se vio que podría ser una solución para el problema con el tráfico TCP y ECMP

visto anteriormente. Para probar su comportamiento se crearon dos políticas, *http* y *http-req* entre el servidor *h_1_1* y el servidor *h_2_2*. En las imágenes 52 y 53 se pueden ver la definición de las políticas y las tablas de flujo ACL con sus respectivos grupos para ambos Leaf. Por último en la imagen 53 se puede ver el trafico sobre el túnel. Si bien se tuvo una perdida de paquetes, la cual se puede deber al funcionamiento del script de mininet, todos los paquetes fueron por el camino correspondiente.

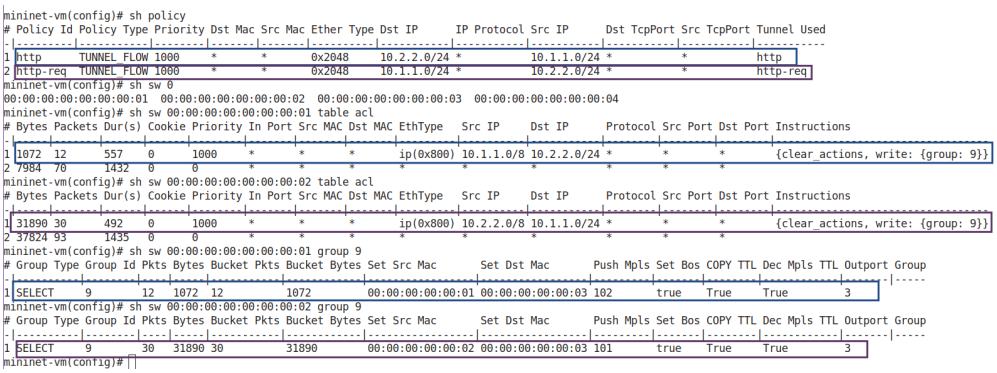


Figura 52: Politica para trafico HTTP entre Leaf 1 y Leaf 2

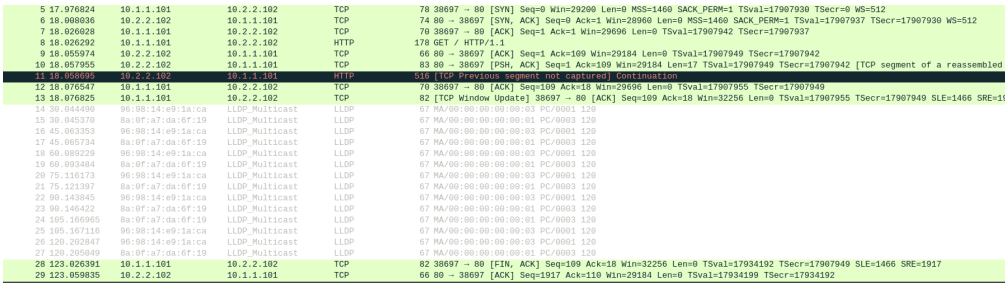


Figura 53: Política para trafico HTTP entre Leaf 1 y Leaf 2

Spine Leaf 4x4x1

Se aplicaron en esta topología túneles y políticas correspondientes a los caminos marcados en la figura 54. En la figura 55 se puede ver la configuración realizada en Onos, marcando según los colores correspondientes a los caminos.

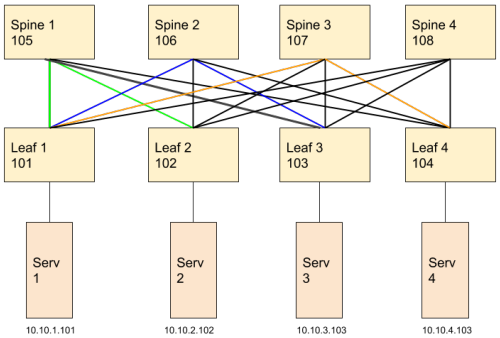


Figura 54: Ingeniería de trafico en Spine Leaf 4x4x1

```
mininet-vm(config)# sh tunnel
# Id          Policies     Tunnel Path [Head-->Tail] Label Stack [Outer-->Inner]
-|-----------|------------|-------------------------|-----------------------
1 azul        azul         [101, 106, 103]           [[103]]
2 azul-rev    azul-reverso [103, 106, 101]           [[101]]
3 naranja     naranja      [101, 107, 104]           [[104]]
4 naranja-rev naranja-rev  [104, 107, 101]           [[101]]
5 verde       verde        [101, 105, 102]           [[102]]
6 verde-rev   verde-rev    [102, 105, 101]           [[101]]
mininet-vm(config)# sh policy
# Policy Id    Policy Type Priority Dst Mac Src Mac Ether Type Dst IP       IP Protocol Src IP       Dst TcpPort Src TcpPort Tunnel Used
-|-----------|-----------|--------|-------|-------|----------|------------|-----------|------------|-----------|-----------|-----------
1 azul         TUNNEL_FLOW 1000     *       *       0x2048     10.10.3.0/24 *           10.10.1.0/24 *           *           azul
2 azul-reverso TUNNEL_FLOW 1000     *       *       0x2048     10.10.1.0/24 *           10.10.3.0/24 *           *           azul-rev
3 naranja      TUNNEL_FLOW 1000     *       *       0x2048     10.10.4.0/24 *           10.10.1.0/24 *           *           naranja
4 naranja-rev  TUNNEL_FLOW 1000     *       *       0x2048     10.10.1.0/24 *           10.10.4.0/24 *           *           naranja-rev
5 verde        TUNNEL_FLOW 1000     *       *       0x2048     10.10.2.0/24 *           10.10.1.0/24 *           *           verde
6 verde-rev    TUNNEL_FLOW 1000     *       *       0x2048     10.10.1.0/24 *           10.10.2.0/24 *           *           verde-rev
```

Figura 55: Túneles y Políticas

Se puede observar en la imagen 56 la tabla de flujos ACL del Leaf 1 junto con los grupos correspondientes a cada túnel. Se ejecuto ping desde el host h_1_1 al resto de los host y se puede identificar la cantidad de paquetes ICMP que se enviaron por cada grupo. Las tablas de las figuras 57, 58 y 59 muestran la política inversa y el grupo utilizado en cada caso.

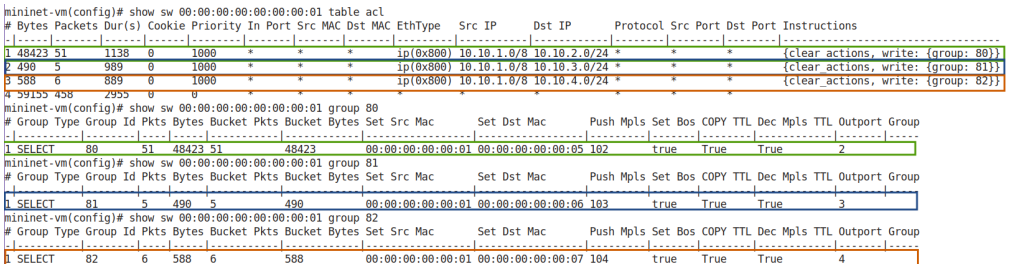


Figura 56: Tabla ACL de Leaf 1 y grupos correspondientes

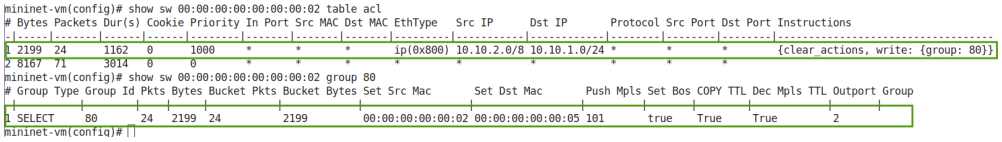


Figura 57: Tabla ACL de Leaf 2 y grupos correspondientes

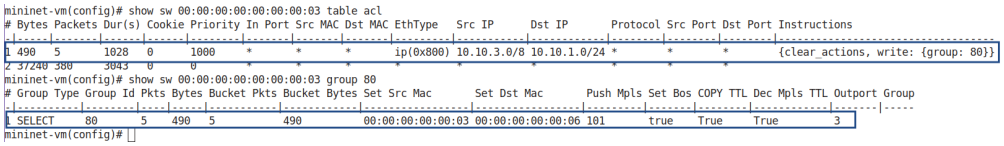


Figura 58: Tabla ACL de Leaf 3 y grupos correspondientes

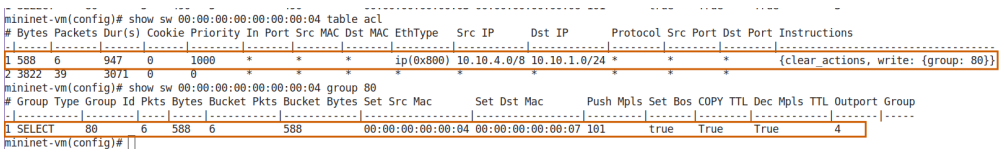


Figura 59: Tabla ACL de Leaf 4 y grupos correspondientes

Por último se probo el comportamiento de TCP en este caso realizando una consulta HTTP del servidor 2 al servidor 1. Se observo que si bien vuelve a perderse un paquete, todo el trafico TCP va por el camino marcado en verde.

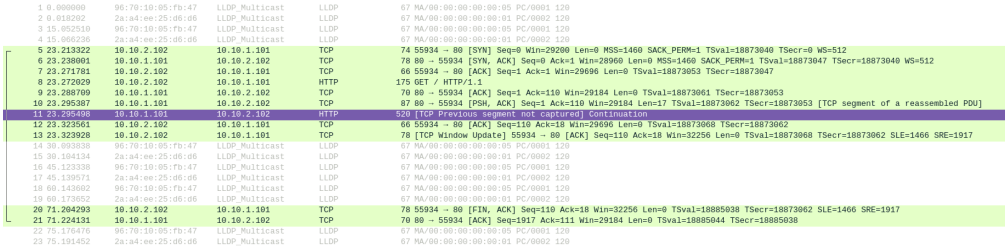


Figura 60: Política para trafico HTTP entre Leaf 1 y Leaf 2

Sobre la utilización de políticas y túneles un problema que se observo en general y se explica en [8] es que si un link que es parte de un túnel y tiene aplicada una política falla, el protocolo comienza a tomar caminos sub-óptimos o directamente se pierde la conectividad. Se probo generar políticas secundarias con menor prioridad que tomaran otro camino, pero el resultado fue el mismo.

4.2. Kathará

Se realizaron las mismas pruebas para el ambiente Kathará con algunos cambios. Las imágenes 61 y 62 muestran las topologías utilizadas, donde se puede observar que en primer lugar los servidores conectados a un Leaf pueden conectarse en un mismo dominio de broadcast, lo cual no era posible en el ambiente Mininet. El controlador se conecta a cada switch mediante un link formando la red de gestión y a su vez se conecta al host de Kathará para que sea posible acceder a la interfaz gráfica.

Para las pruebas asumo que el ambiente esta configurado para utilizar la aplicación Segment Routing.

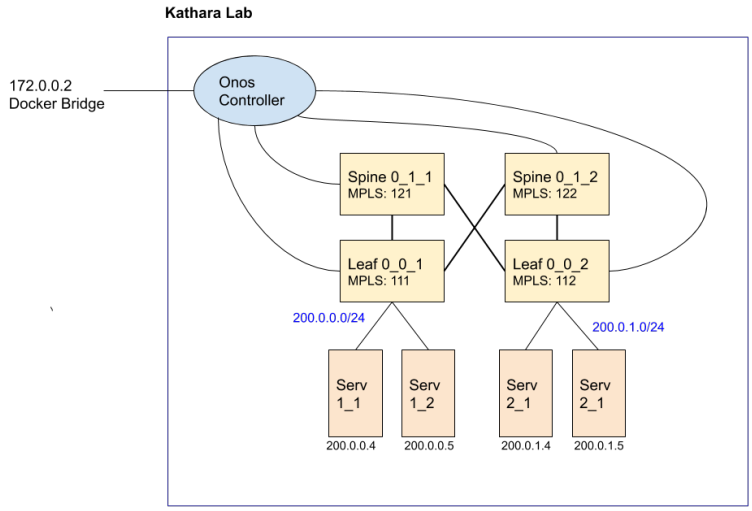


Figura 61: Spine Leaf 2x2x2 con Controlador

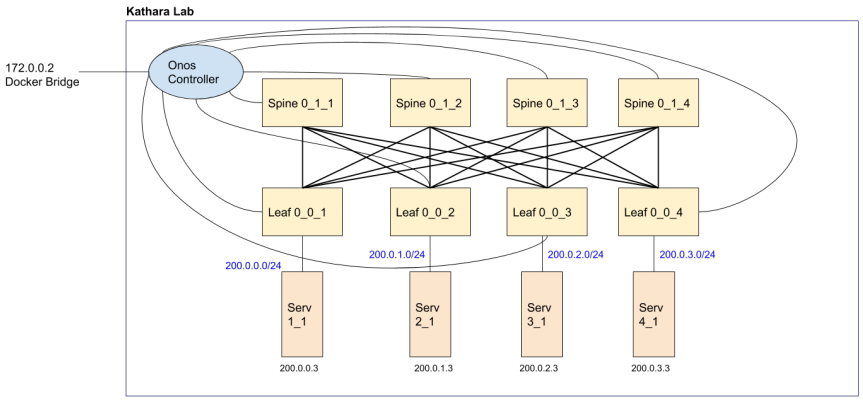


Figura 62: Spine Leaf 4x4x1 con Controlador

4.2.1. Conectividad

Spine Leaf 2x2x2

Se ejecutó un ping desde el host *serv*_1_1 al host *serv*_2_1. En la figura 63 se puede ver un resumen de los flujos y grupos seleccionados para el *leaf*_0_0_1.

Primero se muestra la tabla de flujos con la entrada correspondiente a la subred *200.0.1.0/24*, donde se encuentra el servidor *serv*_2_1. El flujo indica que se debe tomar la acción del grupo $0x70000007$. Dicho grupo es del tipo *Select* y presenta dos caminos para el paquete, uno tomando la acción especificada en el grupo $0x92000004$ y el otro por el grupo $0x92000006$. Se deberían tomar ambos caminos al tratarse de un *Select*, sin embargo se ve tráfico solo en uno de ellos, lo cual se discutirá más adelante. Siguiendo el grupo con trafijo ICMP, se puede ver que la acción que se toma para dicho grupo es la acción PUSH MPLS, agregando la etiqueta 112, NodeSid del *leaf*_0_0_2. Por último se muestra la MAC de destino que corresponde a la MAC del *spine*_0_1_1 y la interfaz de salida que es la interfaz que conecta al Leaf con dicho Spine.

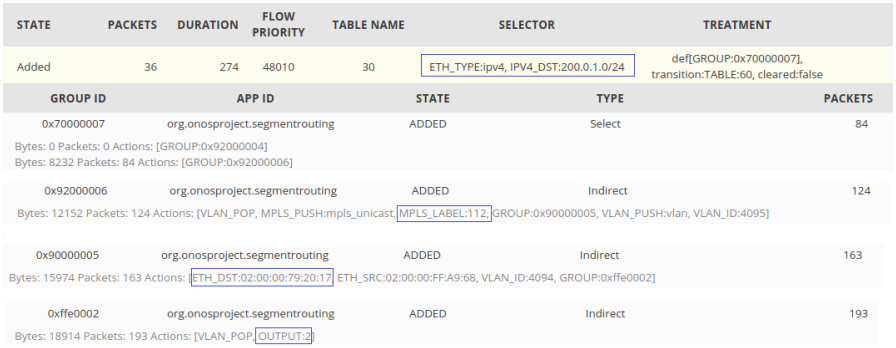


Figura 63: Definición de Flujos y Grupos desde Leaf 1 a subred 200.0.1.0/24

Una vez el trafico llega al *spine*_0_1_1, este realiza un POP de la etiqueta al ser el penúltimo salto y envía el paquete ICMP al *leaf*_0_0_2. Se pueden ver en detalle los flujos y grupos de los spine en el archivo *Pruebas/Conectividad/Kathara/2x2x2/2x2x2 Tablas.ods*.

Luego que el paquete llega ya desetiquetado al *leaf*_0_0_2, el flujo para la IP del servidor indica que se envíe a dicho servidor por la interfaz directamente conectada al mismo.

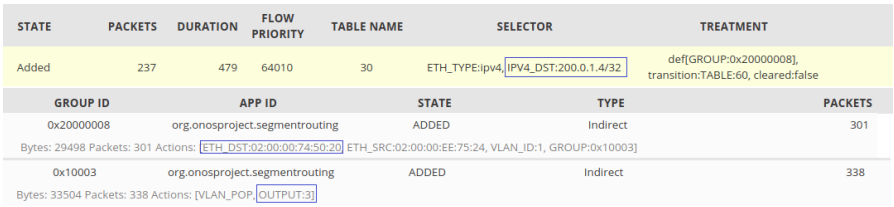
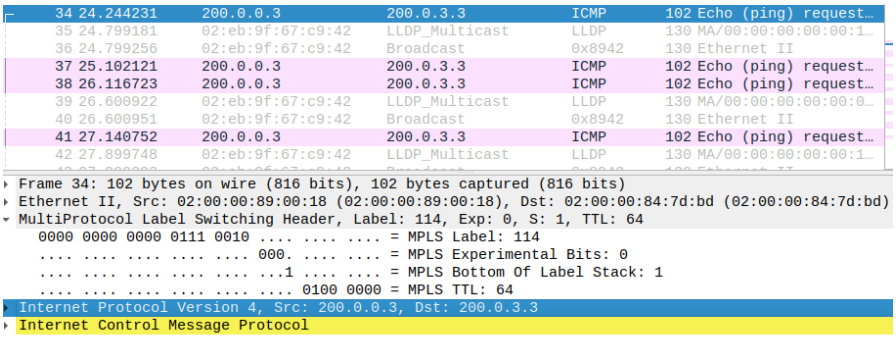
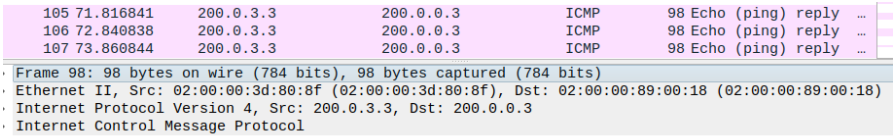


Figura 64: Definición de Flujos y Grupos desde Leaf 2 a servidor 2_1

En las figuras 65 y 66 se muestra la inspección de las interfaces que conectan al $leaf\_0\_0\_1$ con el $spine\_0\_1\_1$ y el $spine\_0\_1\_2$ respectivamente. Se puede observar que los paquetes ICMP echo request estan encapsulados por el paquete MPLS dirigiéndose al primer spine, mientras que la respuesta llega desetiquetada por el segundo (dado que el $spine\_0\_1\_2$ ya realizó la acción POP MPLS de la etiqueta 111 del $leaf\_0\_0\_1$).





En las capturas se detectó que el tráfico ICMP en cada interfaz es en un sentido y no se esta distribuyendo la carga paquete a paquete como ocurría con Mininet. Esto se debe a que en las nuevas versiones de OpenVSwitch el próximo destino para ECMP se elije por flujo, mitigando los problemas de TCP vistos anteriormente. Esto repercute en el balanceo de carga y se debe hacer un estudio más exhaustivo de dicho problema.

Spine Leaf 4x4x1

Se ejecutó un ping entre el host *serv*_1_1 y el host *serv*_4_1 de la topología presentada en la imagen 62. Se puede ver en la entrada de la tabla de flujos de la imagen 67 que para la subred *200.0.3.0/24* corresponde el grupo $0x70000038$, el cual es un *Select* con cuatro caminos definidos. Como se vio en la prueba anterior, solo uno de los grupos muestra tener tráfico, el grupo $0x9200002D$. Para dicho grupo se ejecuta la acción PUSH MPLS de la etiqueta 114, NodeSid del *leaf*_0_0_4. Luego se envía el paquete MPLS por el puerto 4 al *spine*_0_1_4.

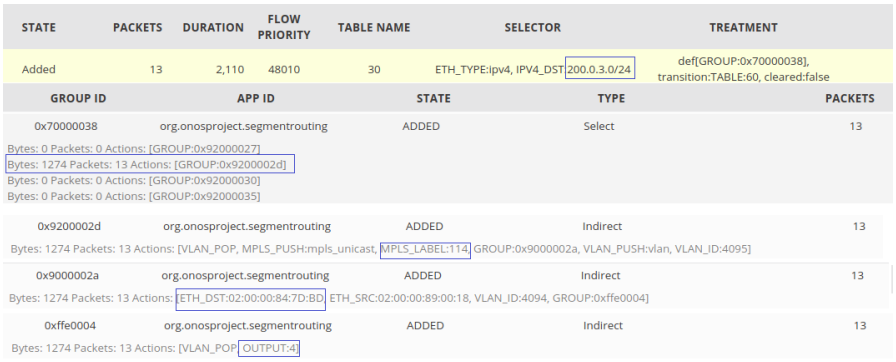


Figura 67: Definición de Flujos y Grupos desde Leaf 1 a subred 200.0.3.0/24

En la imagen 68 se pueden ver el flujo y grupos para la respuesta del servidor *serv*_1_4. Nuevamente se tienen cuatro caminos en el grupo $0x70000077$ del tipo *Select* y se observa tráfico en uno de ellos. La acción de dicho grupo es etiquetar los paquetes con el NodeSid del *leaf*_0_0_1 y enviarlo al *spine*_0_1_2 conectado en el puerto 2.

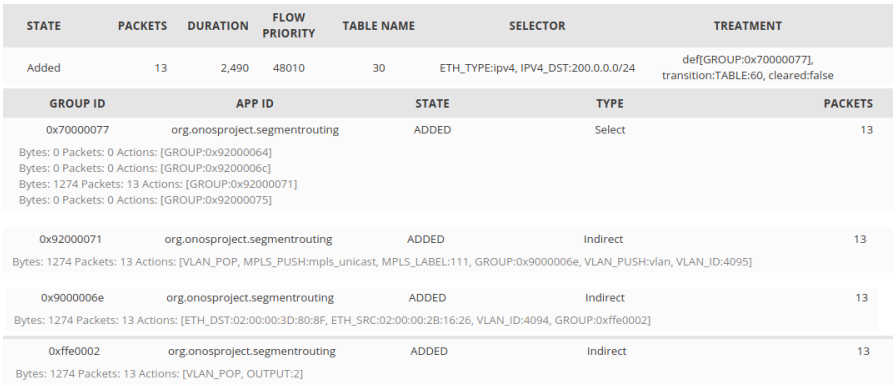


Figura 68: Definición de Flujos y Grupos desde Leaf 4 a subred 200.0.0.0/24

En el directorio *Pruebas/Conectividad/Kathara/4x4x1* se encuentra detallada la información de las tablas de flujos y grupos para cada Spine y Leaf. A su vez se encuentran las capturas de tráfico de las interfaces del $leaf\_0\_0\_1$ con cada Spine.

4.2.2. Falla de Enlaces

Para simular la desconexión de enlaces se utilizó el comando de linux *ip link set dev interface down* en las interfaces de los Spine, conectándose a la consola del nodo Kathará correspondiente.

Spine Leaf 2x2x2

Para validar el funcionamiento de Segment Routing ante fallas de link se ejecutó primero un ping entre el servidor $serv\_1\_1$ y el servidor $serv\_2\_1$. Durante este tiempo se bajó la interfaz que conecta al $spine\_0\_1\_1$ con el $leaf\_0\_0\_2$, como se muestra en la figura 69 , lo cual implicó ejecutar en el Spine el comando *ip link set dev eth2 down*.

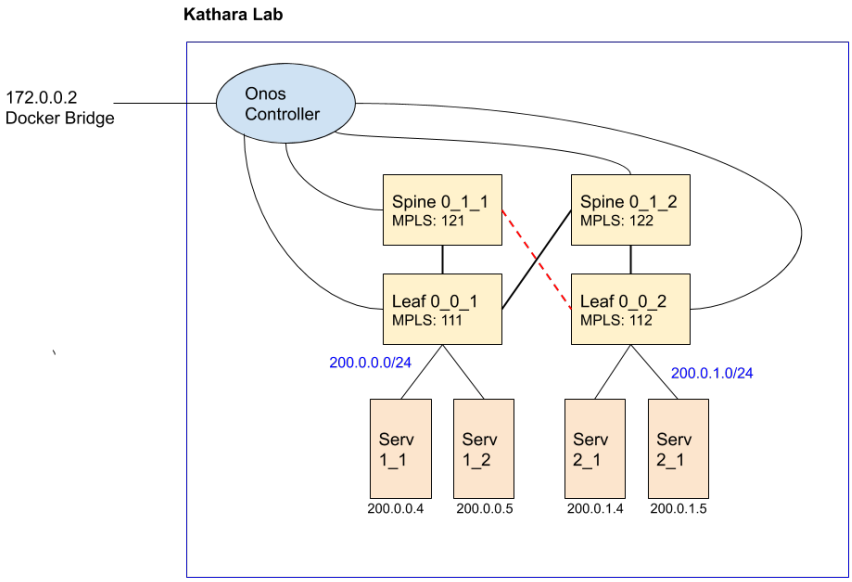


Figura 69: Falla de Link en Spine Leaf 2x2x2

Comparando con la figura 63 se puede ver que en la figura 70 se tiene solo un camino definido para el grupo $0x70000007$ el cual envía el NodeSid etiquetado al *spine*_0_1_2. Durante la caída del link no se perdieron paquetes ICMP entre los hosts.

GROUP ID	APP ID	STATE	TYPE	PACKETS
0x70000007	org.onosproject.segmentrouting	ADDED	Select	58
Bytes: 5684 Packets: 58 Actions: [GROUP:0x92000006]				
0x92000006	org.onosproject.segmentrouting	ADDED	Indirect	561
Bytes: 54814 Packets: 561 Actions: [VLAN_POP, MPLS_PUSH:mpls_unicast, MPLS_LABEL:112, GROUP:0x90000005, VLAN_PUSH:vlan, VLAN_ID:4095]				
0x90000005	org.onosproject.segmentrouting	ADDED	Indirect	590
Bytes: 57656 Packets: 590 Actions: [ETH_DST:02:00:00:79:20:17, ETH_SRC:02:00:00:FF:A9:68, VLAN_ID:4094, GROUP:0xffe0002]				

Figura 70: Flujos y Grupos de Leaf 1 Luego de caida de Link

Spine Leaf 4x4x1

Se bajó en este caso la interfaz que conecta el *spine*_0_1_1 con el *leaf*_0_0_3 como se puede ver en la figura 71. Mientras se ejecutó un ping entre los servidores *serv*_1_1 y *serv*_1_3, el cual no se vio afectado durante la caída del link. Se comprobó antes de bajar el link que el trafico ICMP estaba pasando por el mismo.

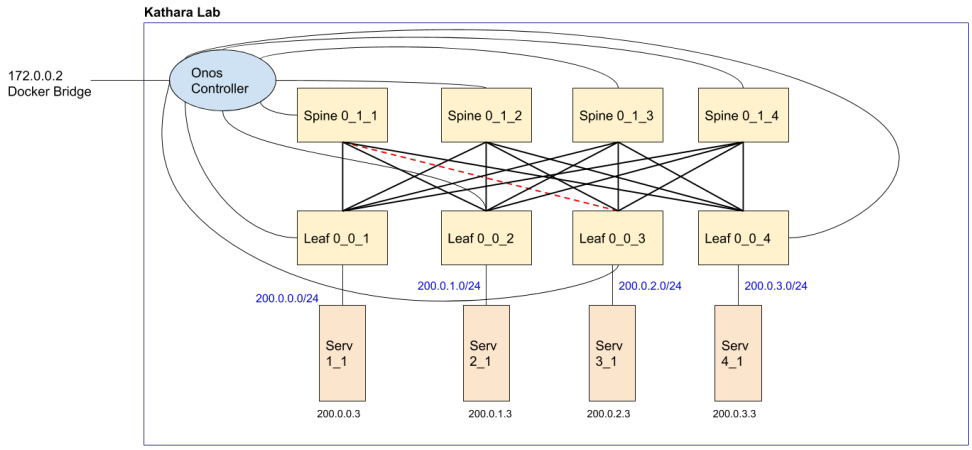


Figura 71: Falla de Link en Spine Leaf 4x4x1

Luego de bajar el link se puede observar en la imagen 72 que el grupo seleccionado tiene tres caminos en lugar de 4 por la cantidad de Spines. No se vio afectada la conectividad entre ambos host.

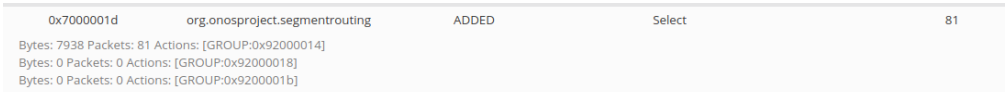


Figura 72: Flujos y Grupos de Leaf 1 Luego de caida de Link

4.2.3. Prueba de Trafico TCP

Dado que en el caso de Kathará se vio que el tráfico de OpenVswitch se distribuye por flujos, resulta de interés ejecutar pruebas con el protocolo TCP y comparar con el ambiente Mininet.

Los servidores Kathará inician con el servicio Apache levantado escuchando conexiones en el puerto 80. En este caso no se trata de un script de simulación como en Mininet, sino que se trata de servidores web sobre contenedores. Para probar se ejecutó *wget* desde los host y se capturo el tráfico en las interfaces de los Leaf.

Spine Leaf 2x2x2

En primer lugar se probó sobre la topología de la figura 61, realizando una consulta http desde el servidor *serv_1_2* al servidor *serv_2_1*, ejecutando el comando *wget 200.0.1.4*. En las imágenes 73 y 74 se puede observar en el sentido de los paquetes TCP que si bien la carga se distribuye no lo hace paquete a paquete, sino por sentido de la conexión. Además se resuelve correctamente la consulta http. Se puede ver en más detalle el tráfico en las capturas del directorio *Pruebas/Trafico TCP/Kathara/2x2x1*.

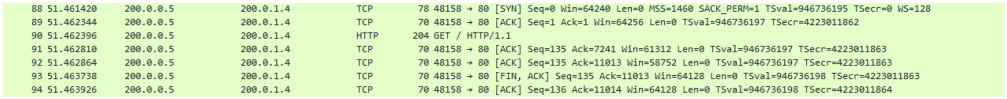


Figura 73: Trafico TCP entre Leaf 1 y Spine 1

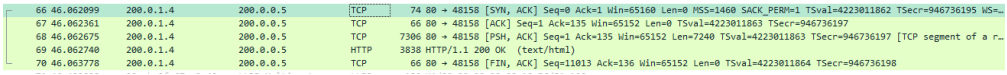


Figura 74: Trafico TCP entre Leaf 1 y Spine 2

Spine Leaf 4x4x1

Se probó luego para la topología de la figura 62 realizar consultas http del servidor *serv_1_1* a los servidores *serv_2_1*, *serv_3_1* y *serv_4_1*.

Se puede observar en la imagen 77 que las consultas a los servidores *serv_2_1* y *serv_3_1* se enviaron al *spine_0_1_2*. Las respuestas a ambas consultas llegaron por medio del *spine_0_1_4* como se registra en la figura 77. Por otro lado tanto la consulta al servidor *serv_1_4* como su respuesta enviaron el tráfico por el *spine_0_1_3* como se muestra en la imagen 76. En todos casos la conexión TCP se mostró estable sin pedidos de retransmisión de paquetes o pérdidas.

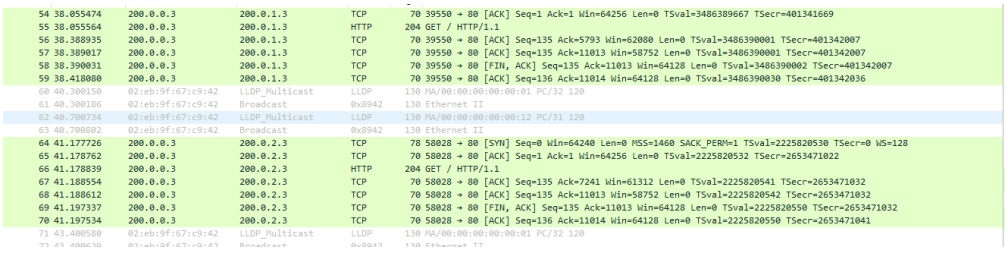


Figura 75: Trafico TCP entre Leaf 1 y Spine 2

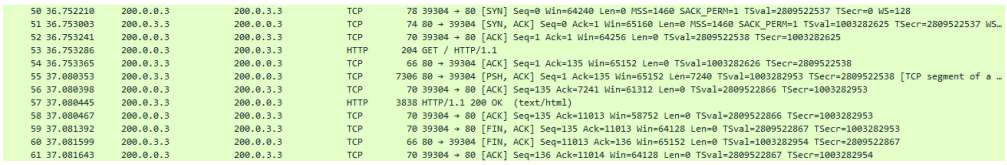


Figura 76: Trafico TCP entre Leaf 1 y Spine 3

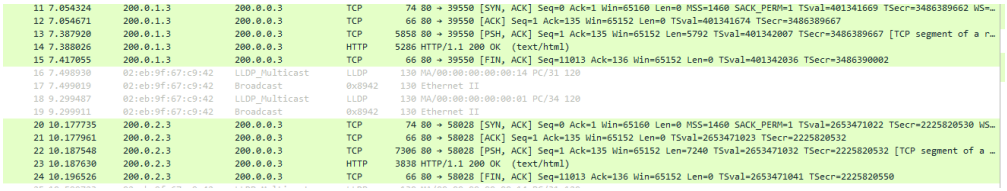


Figura 77: Trafico TCP entre Leaf 1 y Spine 4

Por lo visto en las pruebas del ambiente Kathará el trafico se distribuye por flujo tomando siempre un solo camino en cada sentido de la conexión, el cual puede ser el mismo entre dos nodos o distinto. El ambiente es robusto ante fallas de links y apto para utilizar el protocolo de transporte TCP. Sin embargo con las pruebas realizadas no se llega a apreciar un balanceo de carga. Una opción para comprobar el comportamiento de ECMP es realizar pruebas con grandes cargas de tráfico y analizar si se comienzan a tomar más caminos. Esta prueba no se realizó en el taller dado que requiere de mayor cantidad de recursos.

5. Conclusión

En el desarrollo de el presente trabajo se adquirieron conocimientos teóricos sobre SDN y su aplicación en Datacenters, además de técnicas como MPLS y Segment Routing donde el ruteo se basa en el origen. Por otro lado se profundizó en herramientas como OpenFlow, Onos y Open vSwitch, las cuales son ampliamente utilizadas hoy en día en ambientes productivos.

La solución Segment Routing en conjunto con SDN propone una solución centralizada para el ruteo en Datacenters donde se separa el plano de datos del plano de control. Esto permite una mayor flexibilidad al momento de aplicar ingeniería de trafico para propósitos específicos realizando cambios en toda la red. A su vez permite una escala menos costosa dado que alcanza con tener switches que implementen OpenFlow para aplicar la solución. Al ser una herramienta de código abierto no se esta sujeto a los proveedores de la industria.

Por otro lado se comprendió el funcionamiento de entornos de prueba como Mininet y Kathará, encontrando la forma de trasladar el funcionamiento de un entorno simulado y utilizado para prototipos, a un entorno que emula un ambiente de producción mediante tecnologías ampliamente utilizadas como los contenedores. Dicho entorno puede pasar a un entorno de prueba en Hardware real de una forma directa

En la sección de pruebas se encontró que el entorno Mininet presenta ciertas desventajas en el tratamiento del trafico TCP, dado que el algoritmo ECMP utiliza un algoritmo del tipo Round Robin. El entorno Kathará presento mejoras ante este aspecto ya que ECMP distribuye el trafico por flujos, pero no se detecta un balanceo de la carga.

Trabajo Futuro

Se pueden detectar varias lineas de trabajo a continuar y profundizar a partir del trabajo realizado:

- No se logró aplicar políticas para ingeniería de tráfico sobre el ambiente Kathará. Se debe encontrar y probar una versión de ONOS compatible con

el entorno que lo permita.

- Analizar el funcionamiento de ECMP en Kathará en cuanto a la distribución de carga. Las pruebas realizadas no contenían una carga de tráfico considerable para detectar si la carga se distribuye equitativamente.
- Comparar esta solución centralizada con algoritmos distribuidos ya implementados en el ambiente Kathará (BGP, OpenFabric y RIFT) aplicado a topologías de Datacenter. Interesaría comparar el comportamiento y medir cantidad de mensajería del plano de control, consumo de recursos (Memoria, CPU) y throughput de red entre otros datos.
- Extender el amiente Kathará para emular Trellis.

Referencias

[1] Software-Defined Networking: The New Norm for Networks, Withe Paper, April 13, 2012, https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] Software-Defined Networks: A Systems Approach, Acceso: 2020 https://sdn.systemsapproach.org/uses.html

[3] Research on path establishment methods performance in SDN-based networks, David José Quiroz Martiña, 2015.

[4] OpenFlow Switch Specification
ersion 1.3.5 (Protocol version 0x04), March 26 2015, ONF TS-023 https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf

[5] OpenFlow, Acceso: 2020
http://flowgrammable.org/sdn/openflow/#tab_switch

[6] OpenFlow – Basic Concepts and Theory, Acceso: 2020 https://overlaid.net/2017/02/15/openflow-basic-concepts-and-theory/

[7] Open vSwitch Documentation, Acceso 2020 https://docs.openvswitch.org/en/latest/

[8] Segment Routing, Iipo Litmanen, Heksinski Metropolia Univiersity of Applied Sciences, April 2017

[9] Open Network Operating System, Martin Pacheco, Sebastián Passaro, 2019.

[10] SPRING Open Project Software Architecture, Acceso: 2020 https://wiki.onosproject.org/display/ONOS/Software+Architecture

[11] ONF's SPRING-OPEN project, Acceso: 2020
https://wiki.onosproject.org/display/ONOS/ %5BArchived %5D+ONF %27s+SPRING-OPEN+project

[12] Installation Guide, Acceso: 2020
https://wiki.onosproject.org/display/ONOS/Installation+Guide

[13] Open Network Fundation, Acceso: 2020
https://www.opennetworking.org/

[14] Mininet, Acceso: 2020
http://mininet.org/

[15] Configuring Onos, Acceso: 2020
https://wiki.onosproject.org/pages/viewpage.action?pageId=2130918

[16] Using the CLI, Acceso: 2020
https://wiki.onosproject.org/display/ONOS/Using+the+CLI

[17] Kathara, Acceso: 2020
https://www.kathara.org/

[18] Repositorio Gitlab Taller. Acceso: 2020
https://gitlab.fing.edu.uy/agustina.parnizari/ciberfisicos

[19] Kathara Linux Install, Acceso: 2020
https://github.com/KatharaFramework/Kathara/wiki/Linux

[20] Use bridge networks, Acceso: 2020
https://docs.docker.com/network/bridge/

[21] Kathara: A Lightweight Network Emulation System, Mariano Scazzariello, Lorenzo Ariemma, Tommaso Caiazzi, NOMS 2020 - 2020 IEEE IFIP Network Operations and Management Symposium

[22] Taller de Onos en Docker, Sebastian Passaro, Modulo Taller, 2019

[23] Analysis of an Equal-Cost Multi-Path Algorithm, C. Hopps, https://tools.ietf.org/html/rfc2992