Taller de Sistemas Ciber-Físicos

2019

Routing in fat trees

Autor:
Leonardo Alberro

Supervisores:
Alberto Castro
Eduardo Grampín

13 de diciembre de 2019

Índice

1. Introducción

Data center networks (DCNs) han tomado gran relevancia debido, entre otros factores, al auge de los servicios en la nube. Estos servicios cubren una gran variedad de necesidades tanto para usuarios finales como para pequeñas y grandes organizaciones. Este tipo de redes tiene necesidades diferentes a las redes WAN clásicas, como hiper-confiabilidad, escala y automatización. En este sentido, la *Internet Engineering Task Force* (IETF) ha reconocido que las necesidades de enrutamiento de un datacenter son diferentes a las de la WAN y por esto ha puesto en marcha grupos de trabajo para la creación de nuevos protocolos de enrutamiento para este tipo de redes.

Una DCN que usa una topología de árbol, tiene en la parte inferior los racks de servidores a los que están conectados los hosts físicos. Cada host físico se puede configurar para proporcionar múltiples máquinas virtuales o servidores virtuales. Estos hosts están conectados a los conmutadores Top-of-Rack (ToR). Luego, los conmutadores ToR que están en una fila se conectan a un conmutador de *End of Row* (EoR) o de *borde*. Múltiples filas de servidores con un conmutador de borde en cada fila están conectadas a conmutador de *agregación*, para la agregación de tráfico. Finalmente, estos conmutadores de agregación están conectados a uno o más conmutadores *core*. Este *core* tiene un enlace saliente para conectarse a Internet. Puede haber más de un conmutador *core* y se pueden instalar múltiples enlaces paralelos para la conectividad a Internet.

Esto deja a la vista los tipos de tráfico que se pueden dar en este tipo de redes: tráfico este-oeste y tráfico norte-sur. El tráfico este-oeste refiere al tráfico entre racks de servidores, resultado de aplicaciones internas que requieren transferencias de datos. Por otro lado, el tráfico norte-sur refiere al tráfico como resultado de solicitudes externas de Internet que llegan al datacenter. Dependiendo del propósito comercial específico de un datacenter, puede variar ampliamente la cantidad de tráfico este-oeste en comparación con el norte-sur. [9]

Un ejemplo de esta topología de árbol son las topologías Fat-Tree. Una topología fat-tree para datacenter como se define en [1] es un caso especial de una Clos network [18]. Típicamente se refiere a esta topología en términos de la cantidad de pods [1]. De esta manera, una topología fat-tree consiste en k pods con tres niveles de nodos: *edge*, *aggregation* y *core*. Así, en una topología fat-tree de k pods hay k nodos (cada uno con k puertos) por pod divididos en dos niveles (edge y aggregation) de $k/2$ nodos. A su vez hay $(k/2)^2$ cores conectados a k pods. La figura 10 ilustra este concepto donde k toma el valor de 4.
Notar que en este tipo de topologías no existen enlaces entre nodos de un mismo nivel. Esto deja la oportunidad para hacer referencia al tráfico que fluye por ella como "hacia el sur", indicando el tráfico que fluye en dirección a los nodos *leaf*, o "hacia el norte" indicando el tráfico que fluye en dirección a los nodos *core*.

Este trabajo explora dos protocolos actualmente en desarrollo diseñados específicamente para

[1] Ver 3.1

contemplar este tipo de topologías: Routing in Fat Trees (RIFT) y Link State Vector Routing (LSVR). Debido al estado actual del draft de RIFT y la existencia de una implementación open source del protocolo, el desarrollo experimental pone foco en la prueba de esta implementación en un escenario de simulación desarrollado. Por otra parte, la exploración del protocolo LSVR queda acotada al estudio de sus requerimientos.

2. Trabajos anteriores

Este trabajo utiliza un entorno de simulación en Mininet basado en [10] y en talleres anteriores, que utilizan esta herramienta para la construcción de una topología Fat Tree.

En trabajos anteriores este entorno de simulación fue utilizado para realizar pruebas de protocolos incorporados en FRRouting [6] como BGP, ISIS y OpenFabric en una topología Fat Tree con $k = 4$.

3. RIFT

Según el Internet-Draft de la IETF [13] el protocolo RIFT:

1. Se ocupa de la construcción automatizada de topologías fat-tree basada en la detección de enlaces.
2. Minimiza la cantidad de información de estado de enrutamiento en cada nivel.
3. Implementa poda automática y balancea la carga del flooding de la topología en un subconjunto de enlaces.
4. Admite la desagregación automática de prefijos en caso de fallas en enlaces y nodos con el objetivo de evitar black-holing y enrutamiento subóptimo.
5. Permite direccionamiento del tráfico y políticas de re-routing.
6. Permite el reenvío non-ECMP sin loop.
7. Balancea automáticamente el tráfico hacia los *spines* en función del ancho de banda disponible.
8. Proporciona mecanismos para sincronizar un almacén de datos de clave-valor limitado, que se puede utilizar después de la convergencia del protocolo.

Como se muestra en la figura 1, la propiedad más singular de RIFT es que realiza flooding de la información de estado de enlace hacia el norte, para que cada nivel obtenga la topología completa de los niveles al sur. Esa información nunca se "inunda" de este a oeste (salvo excepciones) o

de vuelta al sur. En la dirección sur, el protocolo funciona como un protocolo de vector de distancias. La información se propaga un salto hacia el sur y es "re-anunciada"por nodos en el siguiente nivel inferior (normalmente solo la ruta predeterminada). Sin embargo, RIFT también utiliza flooding en la dirección sur para evitar la necesidad de construir una actualización por adyacencia.

Otra propiedad importante que presenta es la de "reflexión" la cual facilita implementar *flood reduction* y *automatic dis-aggregation*.

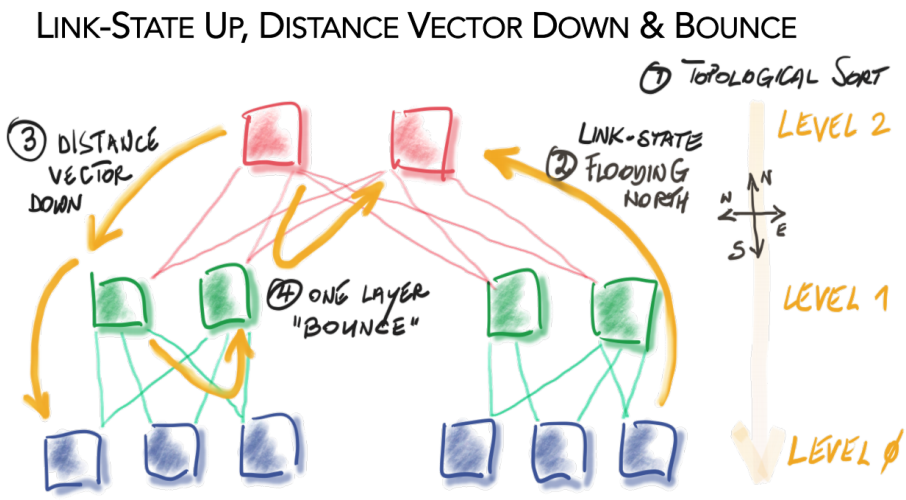

Figura 1: Link-state hacia el norte, Distance-Vector hacia el sur [17].

El draft de RIFT introduce una serie de terminología propia del protocolo, por ejemplo para sus mensajes. Además define algunos conceptos ya existentes en el contexto de fábricas en general, particularmente, renombra los nodos de cada nivel mencionados en 1 como *core*, *spine* y *leaf*.

3.1. Terminología

- **Point of Delivery (PoD)**: Un segmento o subconjunto vertical autónomo de una red Clos o Fat Tree que normalmente contiene solo nodos de nivel 0 y nivel 1. Un nodo en un PoD se comunica con nodos en otros PoD a través del Top-of-Fabric. Se numeran para distinguirlos y se usa PoD #0 para denotar PoD indefinido.
- **Top of PoD (ToP)**: El conjunto de nodos que proporcionan comunicación intra-PoD y tienen adyacencias hacia el norte fuera del PoD.
- **Top of Fabric (ToF)**: El conjunto de nodos que proporcionan comunicación entre PoD y no tienen adyacencias hacia el norte, es decir, están en la "parte superior"de la estructura. Los nodos ToF no pertenecen a ningún PoD y se les asigna un valor PoD "indefinido".
- **Leaf**: Un nodo sin adyacencias hacia el sur.

- **Spine**: Cualquier nodo al norte de los *leaf* y al sur de los ToF. Son posibles múltiples capas de spines en un PoD.
- **Top-of-fabric Plane or Partition**: En grandes fábricas, los ToF pueden no tener suficientes puertos para agregar todos los nodos al sur de ellos y con eso, el ToF se divide en múltiples planos independientes. Un plano es un subconjunto de nodos ToF que se ven entre sí a través de la reflexión sur o enlaces E-W.
- **Radix**: de un conmutador, es básicamente el número de puertos que proporciona. A veces también se llama *fanout*.
- **TIE**: "Elemento de información de topología". Los TIE se intercambian entre nodos RIFT para describir partes de una red, como enlaces y prefijos de dirección, de manera similar a los LSP de ISIS o los LSA de OSPF. Se habla de **N-TIE** cuando son TIEs en la representación hacia el norte y **S-TIE** para el equivalente hacia el sur.
- **South Reflection**: define un mecanismo en el que los S-TIE de nodos se "reflejan" hacia el norte para permitir que los nodos de un mismo nivel y sin enlaces E-W se "vean" entre sí.
- **LIE**: "Elemento de información de enlace", en gran parte es equivalente al *Hello* de IGP e intercambiado a través de todos los enlaces entre sistemas que ejecutan RIFT con el objetivo de formar adyacencias.
- **De-aggregation/Disaggregation**: Proceso en el que un nodo decide anunciar ciertos prefijos que recibió en un N-TIE para evitar el *black-holing* y el enrutamiento subóptimo ante fallas de enlaces.
- **Bandwidth Adjusted Distance (BAD)**: Cada nodo RIFT calcula la cantidad de ancho de banda hacia el norte disponible para un nodo en comparación con otros nodos en el mismo nivel y modifica la distancia de la ruta predeterminada en consecuencia con el objetivo de permitir que el nivel inferior ajuste su balanceo de carga hacia los spines.

3.2. Topología

Existen dos diseños para la topología fat-tree con la que trabaja RIFT: *single plane* y *multi plane*. En la topología *single plane* todos los *ToF* son iguales y están inicialmente conectados con todos los nodos del nivel inferior. Un ejemplo de esta topología se presenta en la figura 9. En contraste a esto, en los diseños *multi plane* no todos los *ToF* están conectados a todos los nodos del nivel inferior, dando lugar a múltiples planos. Estas topologías tienen modos de falla y comportamientos que no existen en las topologías *single plane*. Manteniendo la cantidad de nodos por nivel de la figura 9, en la figura 2 se puede apreciar la misma fábrica particionada en dos planos disjuntos.

El despliegue de topologías con un único plano introduce una limitante denominada "límite de plano único" dada por el *radix* de los nodos *ToF*. Sin embargo, una clara ventaja de un Top-of-Fabric no particionado es que todas las fallas pueden resolverse mediante una desagregación simple, no transitiva y positiva que se propaga solo dentro de un nivel de la fábrica. En otras palabras, los nodos ToF no particionados siempre pueden alcanzar los nodos que están debajo o quitar las rutas de los *PoD* que no pueden alcanzar de forma inequívoca.

Para escalar más allá del "límite de plano único", el nivel superior de la fábrica se puede dividir mediante un número N de planos, donde cada plano es "servido" por un subconjunto de nodos *ToF*, dando lugar al diseño *multi plane* mencionado. Este tipo de diseño es común en grandes fábricas o fábricas construidas con nodos de bajo *radix*.

3.3. Conceptos relevantes

En esta sección se presentan algunos de los conceptos más relevantes de RIFT. En particular se presentan los relevantes para este trabajo: **Fallen Leaf Problem**, **Positive Disaggregation**, **Negative Disaggregation** y **Flooding reduction**.

3.3.1. Fallen Leaf Problem

Se define como *Fallen Leaf* a un nodo *leaf* al que solo pueden llegar un subconjunto de nodos *ToF* pero no todos, debido a la falta de conectividad. Si R es el factor de redundancia, se necesitan al menos las R roturas para alcanzar una situación de *Fallen Leaf*.

Para ilustar este concepto en la Figura 2 se muestra una topología multi-plane con $k = 4$ donde debido a la caída del único link que conecta a los leafs del pod-1 con el super-2 resulta en dos fallen leafs: leaf-1-1 y leaf-1-2.

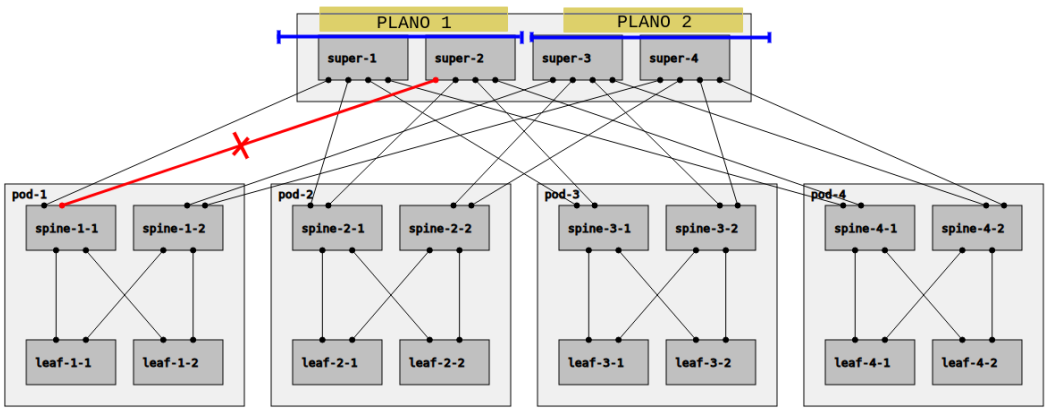

Figura 2: Fallen leaf

Este problema es más frecuente en topologías particionadas como la de la Figura 2. Notar que si se agrega la falla del link spine-1-1 <-> super-1, los fallen leafs mencionados quedan completamente inaccesibles desde el plano 1.

3.3.2. Positive, Non-transitive Disaggregation

En circunstancias normales, los S-TIE del nodo contienen solo las adyacencias y una ruta por defecto. Sin embargo, si un nodo detecta que su prefijo IP por defecto cubre uno o más prefijos a los que se puede acceder a través de él pero no a través de uno o más nodos del mismo nivel, **debe** anunciar explícitamente esos prefijos en un S-TIE. De lo contrario, un porcentaje del tráfico hacia el norte para esos prefijos se enviaría a nodos sin una accesibilidad adecuada, lo que provocaría el fenómeno de *black-holing*.

Se refiere al proceso de anunciar prefijos adicionales en dirección sur como *"positive de-aggregation"* o *"positive dis-aggregation"* . Tal desagregación no es transitiva, es decir, sus efectos siempre están contenidos en un solo nivel de la topología. Naturalmente, las fallas de múltiples nodos o enlaces pueden conducir a varias instancias independientes de desagregación positiva.

Un escenario donde debido a una falla en un link todo el tráfico destinado al nodo es perdido es el que se muestra en la siguiente figura:

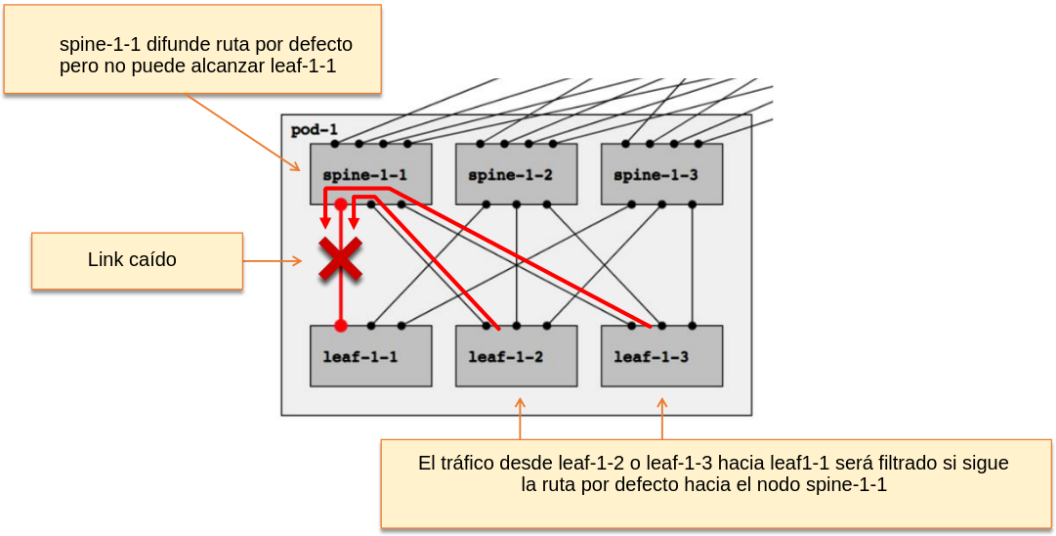

Figura 3: Ejemplo de pérdida de conectividad

Con este escenario el protocolo actuará ejecutando una desagregación positiva en los nodos spine-1-1 y spine-1-3. Estos nodos se enteran de la caída del link por *South Reflection* (y por tanto de la pérdida de conectividad entre el spine-1-1 y el leaf-1-1) y proceden a difundir rutas más específicas para el leaf-1-1 como se muestra en la siguiente figura:

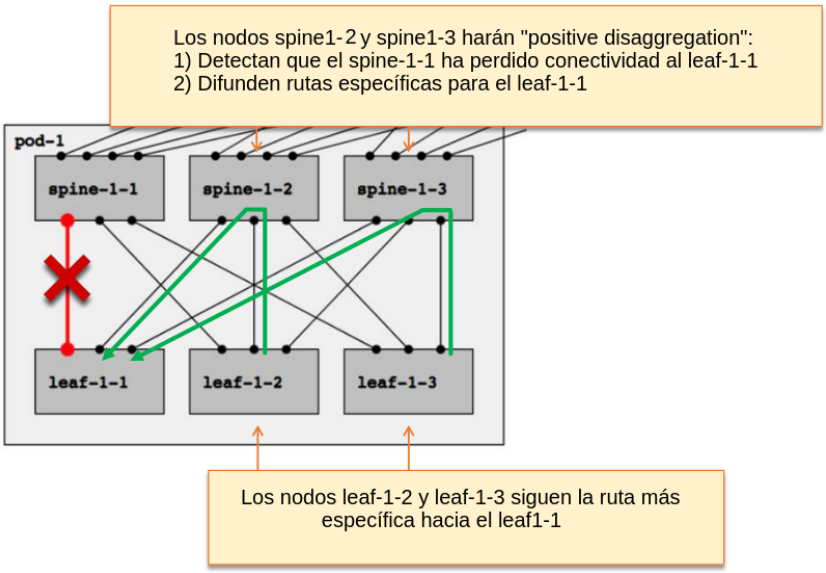

Figura 4: Positive disaggregation para recuperar conectividad luego de una falla

En definitiva el algoritmo sigue las siguientes etapas:

1. Detecta un blackhole.
2. Lanza la difusión de prefijos desagregados (más específicos).
3. Difunde los prefijos desagregados en un S-TIE.
4. Instala los prefijos desagregados en la tabla de rutas.

3.3.3. Negative, Transitive Disaggregation for Fallen Leafs

Las fallas en el nivel superior de una fábrica con topología *multi-plane*, o las fallas en más de cierto umbral de links en un diseño *single-plane* pueden generar *fallen leafs*. Estos escenarios no pueden abordarse solo por desagregación positiva y necesitan un mecanismo adicional.

Un nodo *ToF* que descubre que no puede alcanzar un nodo *fallen leaf* desagrega todos los prefijos de tales *leafs*.

Transitivamente, un nodo pierde la conectividad a un prefijo cuando ninguno de sus hijos lo anuncia y cuando el prefijo es desagregado "negativamente"por todos sus padres. Cuando eso sucede, el nodo anuncia el prefijo negativo hacia el sur. Dado que el mecanismo se aplica recursivamente al sur, este prefijo negativo puede propagarse transitivamente hasta la hoja. Esto es necesario debido a que las hojas conectadas a múltiples planos por medio de caminos disjuntos, pueden tener que elegir el plano correcto ya en la parte inferior de la fábrica, y deben asegurar que no se envíe tráfico hacia otra hoja usando un plano que está "caído".

En el ejemplo ilustrado en la figura 5 se puede ver como el mecanismo de desagregación negativa logra evitar que el nodo leaf-3-2 envíe tráfico al fallen leaf-1-2 a través del plano 1, por el cual este es inalcanzable. En este ejemplo la transitividad el mensaje es clave, debido a que evita que el leaf-3-2 tome la decisión inicial de enviar el tráfico al leaf-1-2 a través del spine-3-2 (que lo conecta con el plano 1).

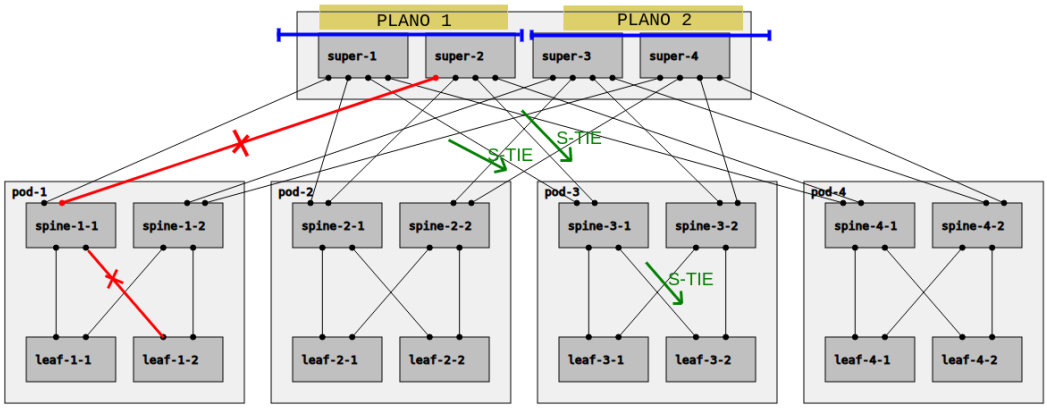

Figura 5: Negative disaggregation debido a fallen leaf

Cuando se restablece la conectividad, un nodo que desagregó un prefijo retira la desagregación negativa por el mecanismo habitual (TIE) omitiendo el prefijo negativo. Notar que en este ejemplo la reflexión de información entre nodos *core* ocurre para nodos del mismo plano, por esto es necesario contar con este mecanismo de desagregación negativa.

3.3.4. Flooding reduction

El mecanismo utilizado para distribuir los TIE es el mecanismo de flooding (modificado en varios aspectos para aprovechar la topología fat-tree) utilizado por los protocolos de estado de enlace actuales. Dado que el flooding tiende a presentar una carga no escalabe en topologías grandes y densamente malladas (e.g fat-tree como se muestra en la figura 6), RIFT proporciona como solución un mecanismo de flooding reduction y balanceo de carga global casi óptimo.

Una técnica similar al concepto de "multipoint relay" (MPR) introducido en la definición del *Optimized Link State Routing Protocol*[5] se aplica a RIFT para controlar el flooding hacia el norte.

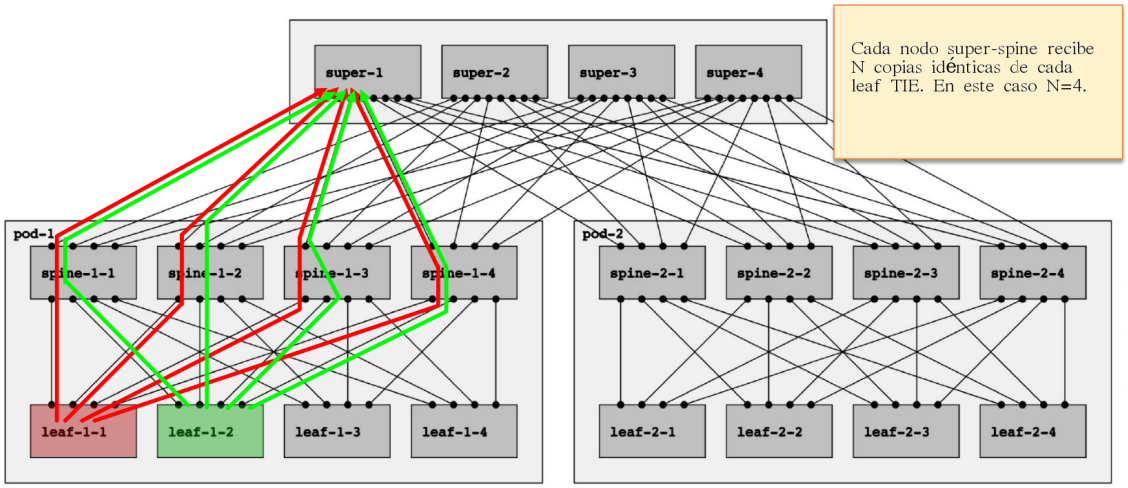

Figura 6: Recepción de N-TIEs repetidos debido al flooding en una topología fat-tree.

La especificación del algoritmo especificado en RIFT puede verse en detalle en [13]. Usar RIFT con esta funcionalidad puede mejorar la performance del protocolo. En la figura 7 se muestra un breve algoritmo acompañado de los resultados de aplicarlos al problema de la figura 6.

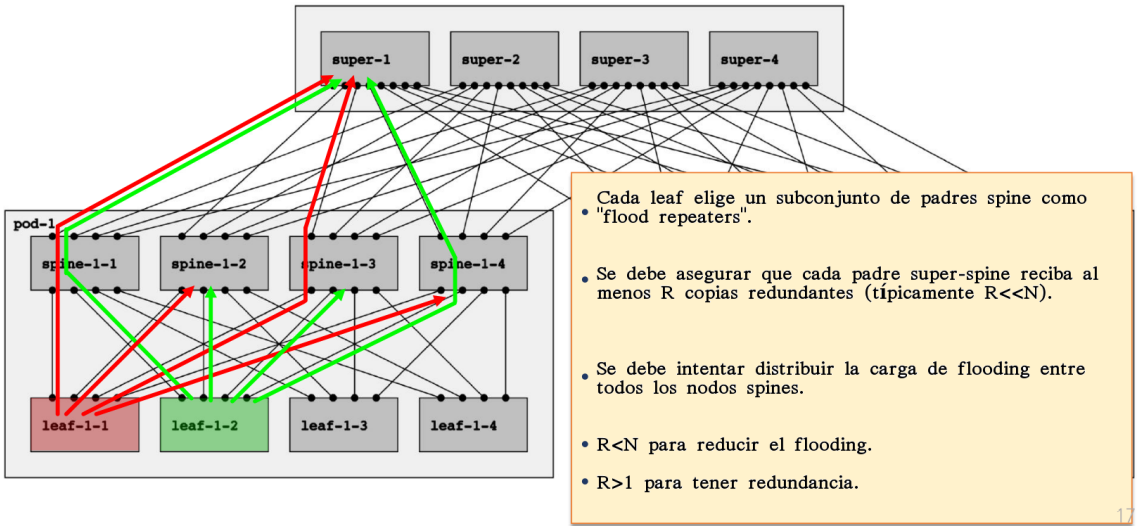

Figura 7: Intercambio de N-TIEs aplicando flooding reduction.

4. Desarrollo experimental

El desarrollo experimental de este trabajo consta de varias partes. En primer lugar se experimentó con la creación de topologías fat-tree en Mininet como se muestra en 4.1. Luego se investigó la implementación open source del protocolo y se experimentó con topologías fat-tree creadas desde una herramienta incluida en la implementación (4.2). Finalmente, con una visión

más profunda a cerca del funcionamiento del protocolo y su implementación, se trabajó en la incorporación del mismo para las topologías creadas en un ambiente Mininet: 4.3.

4.1. Topología en mininet

Trabajos anteriores desarrollaron, mediante el uso de la python API de mininet, una topología fat tree multi-plane, instanciando en ella las configuraciones necesarias para el funcionamiento de distintos protocolos como BGP, ISIS y Open Fabric.

En primera instancia, en este trabajo se experimentó con el funcionamiento de los distintos protocolos en base al trabajo mencionado, reproduciendo las pruebas realizadas y analizando los resultados documentados.

Como aporte se logró crear un nuevo programa en python que solo se encargue de crear la topología en base a la python API de mininet y establecer la configuración referente a la numeración IPv4. Abstrayendo así, la construcción del ambiente de las configuraciones particulares de cada protocolo.

El desarrollo solo contempla topologías multiplane y está limitado a 254 nodos por nivel.

Por ejemplo, para crear una topología fat-tree multiplane con $k = 4$ basta con la siguiente ejecución:

```
$ sudo python topo_y_num.py 4
```

La topología generada sigue las siguientes consideraciones:

- Por simplicidad los racks se modelan como si cada uno de ellos alojara un solo host.
- Los routers *core* se nombran r(1000+k). e.g r1001. Sus interfaces se nombran de derecha a izquierda como ethX, $X \in 0..(k-1)$.
- Los routers *spine* se nombran r(2000+($k \times k/2$)). e.g r2001. Sus interfaces se nombran de izquierda a derecha y de arriba a abajo como ethX, $X \in 0..(k-1)$.
- Los routers *leaf* se nombran como r(3000+($k \times k/2$)). e.g r3001. Sus interfaces se nombran de izquierda a derecha y de arriba a abajo como ethX, $X \in 0..(k-1)$.
- Los hosts se nombran como h(000 + ($k \times k$)).
- Las subredes entre los nodos *core* y *spine* tienen el rango 10.0.x.0/24.
- Las subredes entre los nodos *spine* y *leaf* tienen el rango 10.1.x.0/24.
- Las subredes entre los nodos *leaf* y los hosts tienen el rango 192.168.x.0/24.

De esta manera el ejemplo considerado queda representado por la siguiente figura:

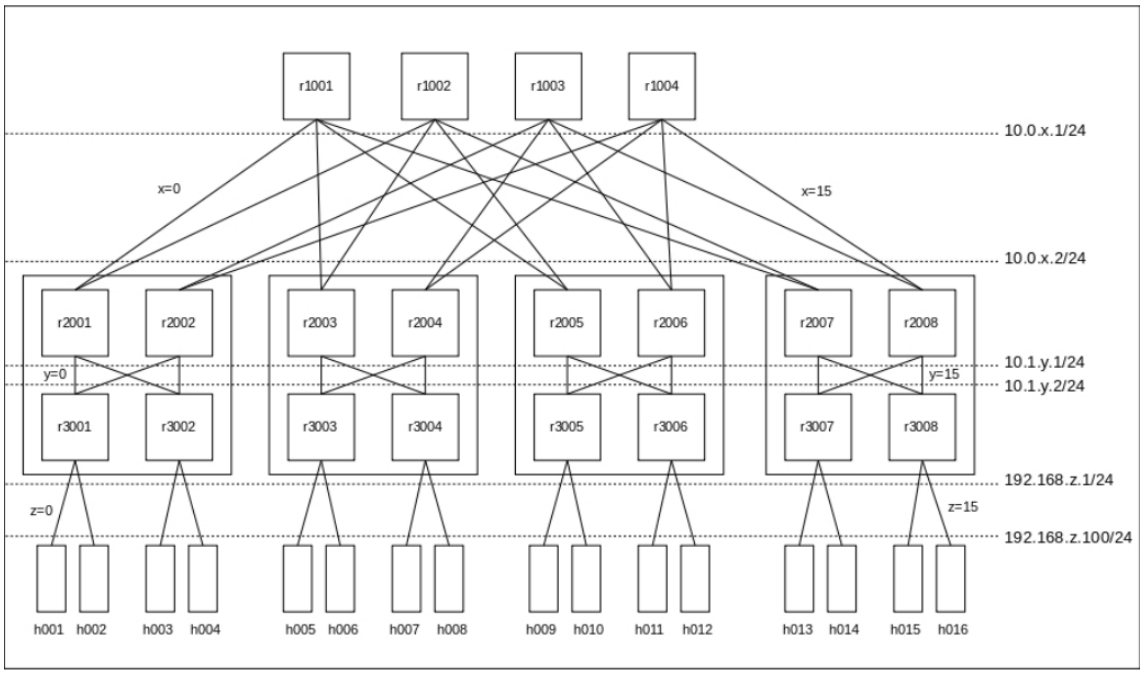

Figura 8: Topología fat-tree en Mininet con k=4.

4.2. Reproducción de escenario en rift-python

La implementación open source de RIFT es llevada a cabo por Bruno Rijsman en [15] y tiene como objetivo acompañar el desarrollo del draft. Esta implementación contiene un generador de topologías fat-tree single-plane y configuración que permite la generación de distintos escenarios sin la necesidad de estar compenetrado con el funcionamiento del protocolo ni de los archivos de configuración necesarios para su funcionamiento. Además se encuentra disponible la hackatón correspondiente a la IETF 104 meeting realizada a inicios de 2019 en Praga, la cuál contiene instructivos detallados para la puesta en funcionamiento del protocolo y la realización de tests aleatorios [16].

La reproducción del mismo escenario construido en mininet (fat-tree multi-plane con k=4) con la topología brindada por las herramientas del proyecto rift-python se realizó en dos etapas: primero mediante la generación de una topología single-plane base y luego la conversión de la topología single-plane a una multi-plane.

Respecto a los conceptos mencionados en 3.3, esta implementación actualmente no cuenta con el mecanismo de *Negative, Transitive Disaggregation for Fallen Leafs*. La lista completa de requerimientos implementados y pendientes está disponible en [14].

4.2.1. Generación de una topología single-plane

La generación de una topología single-plane y todos los archivos de configuración necesarios para la puesta en marcha de RIFT se encuentra automatizada en el proyecto rift-python mediante la herramienta *config generator*. Este generador recibe como parámetro un archivo de meta configuración al que se le indica la cantidad de nodos por nivel.

Para este caso, en el que se intenta modelar una topología con k=4 la meta configuración contiene la siguiente información:

```
nr-pods: 4
nr-leaf-nodes-per-pod: 2
nr-spine-nodes-per-pod: 2
nr-superspine-nodes: 4
```

Y mediante el comando:

```
$ tools/config_generator.py - -netns-per-node
meta_topology.yaml carpeta_destino
```

se obtienen todos los archivos necesarios para la construcción de los namespaces que simularán los nodos, scripts de inicialización y finalización, archivos de configuración necesarios para el funcionamiento del RIFT en cada nodo, etc. Esta topología con k=4 en un enfoque single-plane se puede observar en la figura 9 y es la que se tomará se base para la construcción de la topología k=4 multi-plane objetvo.

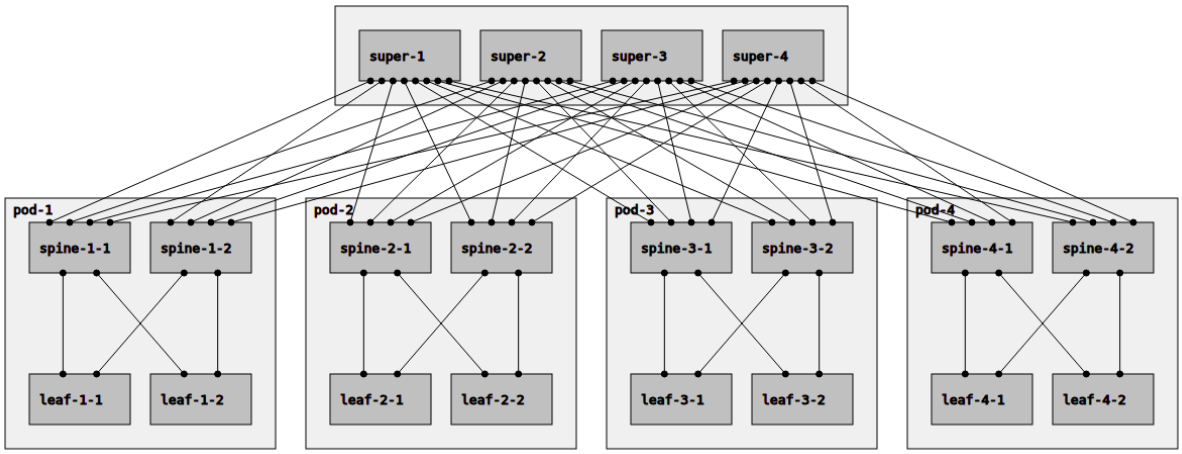

Figura 9: Topología single-plane k=4

4.2.2. Conversión de la topología single-plane a una multi-plane

Por definición, en una topología single-plane, cada nodo *core* está conectado a todo los nodos *spine*. Mientras que en una multi-plane esto no se cumple. Para lograr reproducir el escenario deseado tomando como base los archivos dados por el generador de configuración de una topología single-plane con k=4 basta con eliminar los links e interfaces adecuados.

Con ese objetivo se acomodaron los archivos de configuración de cada nodo *spine* y *core*. Se editó también la creación de los namespaces, eliminando la creación de las interfaces virtuales "redundantes" y los links que involucraban a las mismas. Se acondicionó, además, un `allocations.txt` que contiene la información relacionada a la topología creada: nombre de cada nodo, nivel, vecinos, direcciones IP, systemID, entre otros, con el objetivo de tener una visión general de la topología y numeración creada al momento de realizar las pruebas.

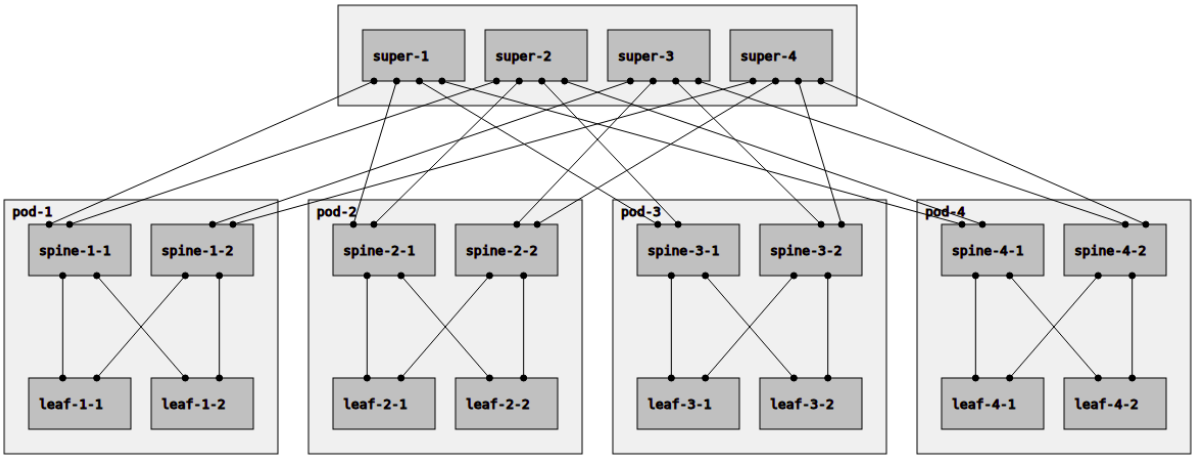

Figura 10: Topología multi-plane k=4

4.2.3. Ejecución y pruebas

La ejecución de rift junto con la topología brindada por las herramientas del proyecto queda simplificada en los scripts auto-generados. La interacción con los CLI de cada nodo también se encuentran en los scripts auto-generados.

El motor del protocolo escribe mensajes de log en el archivo `rift.log` alojado en el mismo directorio donde el motor del protocolo fue iniciado. Un comportamiento a notar es que cuando el motor inicia no borra el contenido anterior del archivo de log, sino que solo realiza operaciones de append sobre el mismo.

Algunos de los comandos disponibles con los que se verificó el funcionamiento del protocolo son:

- `show forwarding`: muestra todas las rutas en la Forwarding Information Base (FIB) del nodo actual. Muestra tanto la FIB de IPv4 como la FIB de IPv6. Estas rutas son las que

se deben instalar en el kernel.

- `show interfaces`: muestra un resumen de todas las interfaces RIFT (i.e. las interfaces en las que se ejecuta RIFT) del nodo actual.
- `show kernel addresses`: muestra un resumen de todas las direcciones en el kernel de Linux en el que se ejecuta el motor RIFT. En el despliegue actual (un motor por nodo) es equivalente a las direcciones del kernel para el nodo en el que se ejecuta el comando.
- `show node`: informa los detalles del nodo RIFT actual. Algunos de estos detalles utilizados en las pruebas realizadas son: *Name, System ID, Flooding Reduction Enabled*.
- `show routes`: muestra todas las rutas en la Routing Information Base (RIB) del nodo actual. Muestra tanto la RIB de IPv4 como la RIB de IPv6.
- `show routes family` *family*: muestra las rutas de una familia de direcciones dada en la RIB del nodo actual. El parámetro *family* puede ser IPv4 o IPv6.
- `show flooding-reduction`: muestra información sobre el mecanismo de *flooding reduction*. Específicamente, muestra qué routers han sido elegidos como *Flood Repeaters* (FR), e incluye información para comprender la elección según los pasos del algoritmo.
- `show spf`: muestra los resultados de la ejecución más reciente del algoritmo *Shortest Path First* (SPF) para el nodo actual.
- `show tie-db`: muestra el contenido de la *Topology Information Element Database* (TIE-DB) para el nodo actual.

Trabajando con la topología *multi-plane* mostrada en 4.2.2 se realiza la siguiente prueba:

```
leaf-1-1> show forwarding
IPv4 Routes:
+-----------+-----------+--------------------------+
| Prefix    | Owner     | Next-hops                |
+-----------+-----------+--------------------------+
| 0.0.0.0/0 | North SPF | veth-1001a-101a 99.1.2.2 |
|           |           | veth-1001b-102a 99.3.4.4 |
+-----------+-----------+--------------------------+
IPv6 Routes:
+--------+-----------+-----------------------------------------+
| Prefix | Owner     | Next-hops                               |
+--------+-----------+-----------------------------------------+
| ::/0   | North SPF | veth-1001a-101a fe80::5c81:64ff:fe76:8b39 |
|        |           | veth-1001b-102a fe80::d8e2:40ff:fe3f:652e |
+--------+-----------+-----------------------------------------+
```

En este ejemplo se muestra la ejecución del comando `show forwarding` dentro del CLI del nodo *leaf-1-1*, con el cual se puede comprobar el correcto funcionamiento de RIFT en el mismo. Dado que RIFT utiliza *distance vector* al sur, este nodo (que solo tiene adyacencias al norte) solo tiene las dos rutas por defecto anunciadas por los dos nodos *spine* que tiene directamente conectados.

En cuanto a las pruebas automatizadas realizadas en este ambiente, a parte de la verificación de conectividad total, se reprodujo una de las pruebas realizadas en el trabajo anterior antes mencionado para BGP, Open Fabric e ISIS en una topología $k = 4$. Esta prueba verifica que el balanceo de carga funcione correctamente. Si bien se desarrolló la prueba para este ambiente, no se puede observar el balanceo de carga debido a un bug en el código rift-python, el cual no permite agregar al kernel rutas redundantes.

El código creado para *python3* incluye pruebas con el comando `ping` y capturas de tráfico mediante `tcpdump` sobre los *network namespaces* con los que modela cada nodo el simulador RIFT.

Las verificaciones de funcionamiento entonces se realizaron mediante el uso de los comandos del CLI incorporado al protocolo presentados, donde se pudo chequear el correcto funcionamiento del poblado de las tablas de forwarding y del intercambio de mensajería RIFT.

4.3. Rift-python en mininet

4.3.1. Ambiente

Para la simulación se creó un ambiente con un sistema operativo Ubuntu Server 18.04 LTS. En este sistema se instaló, Python 3, pip 3 y el motor del protocolo python RIFT. Esto último se realizó utilizando el instalador brindado en el repositorio:

```
$ git clone https://github.com/brunorijsman/rift-python.git
$ cd rift-python
$ install/install.sh
```

El instalador actualiza los paquetes existentes, instala *python3-venv* para la creación de virtual enviroments, crea uno, lo activa e instala en él las dependencias necesarias para el funcionamiento de RIFT.

En cuanto a mininet, se instaló directamente desde la fuente, tomando como versión estable la branch `2.3.0d6` utilizada en los trabajos anteriores.

4.3.2. Configuración

RIFT-Python puede simular topologías de múltiples routers de dos maneras:

1. Ejecutando un solo proceso RIFT-Python que simule múltiples routers RIFT en un solo proceso. Todos los enlaces simulados entre ellos comparten un solo enlace físico y se utiliza números de puerto UDP para separar el tráfico entre enlaces simulados. Para este caso, hay un único archivo de configuración yaml que contiene la configuración para todos los routers y los enlaces entre ellos. Este es el enfoque que utilizan la mayoría de los scripts de prueba.
2. Ejecutando cada router RIFT en un proceso separado. Cada proceso se ejecuta en su propio namespace de red separado. Los enlaces simulados entre ellos se implementan utilizando interfaces virtuales de ethernet (veth). En este caso, hay un archivo de configuración yaml separado para cada router RIFT.

Para trabajar con RIFT en mininet el enfoque apropiado es el (2). En este enfoque lo más importante en el archivo de configuración son los nombres de las interfaces veth a las que está conectado el router RIFT. Con este objetivo, se desarrolló un programa `fattree_rift.py` que incluye en la creación de la topología con la API para python de mininet, la generación de los archivos de configuración yaml necesarios para cada nodo creado y la puesta en marcha del protocolo. El formato del archivo de configuración requerido por python RIFT aún no se encuentra documentado, por esto se generaron los archivos necesarios en base al schema disponible en el repositorio del proyecto y siguiendo los ejemplos brindados por la herramienta de autogeneración. Un ejemplo de archivo de configuración para un router *ToF* es:

```
shards:
  - id: 0
    nodes:
      - name: r1001
        level: 2
        systemid: 1
        interfaces:
          - name: r1001-eth0
            # r1001-eth0  -> r2001-eth0
          - name: r1001-eth1
            # r1001-eth1  -> r2003-eth0
          - name: r1001-eth2
            # r1001-eth2  -> r2005-eth0
          - name: r1001-eth3
            # r1001-eth3  -> r2007-eth0
```

Los comentarios (#) denotan la conexión de la interfaz en cuestión.

En este ejemplo se tiene *level: 2* debido a que es un router *ToF*. En caso de ser un router *leaf* tendríamos `level: 0` y en caso de un *spine*, `level: 1`. El campo `systemid` para los nodos *ToF*

contiene la posición contando de izquierda a derecha, para los *spines* contiene `10x` y en el caso de los *leaf* `100x`, con `x` el índice de la posición en el nivel.

Por otra parte dado que en RIFT no se espera que los nodos *leaf* puedan hacer ping a los nodos *spine* o *ToF*, el archivo de configuración dispone de la clave *v4prefixes* para indicar los prefijos que se deben distribuir. En este sentido se agregó a la configuración de los nodos *leaf* los prefijos hacia los hosts de la topología mostrada en la Figura 10. Así, el yaml para el *leaf-1-1* queda de la siguiente manera:

```
shards:
  - id: 0
    nodes:
      - name: r3001
        level: 0
        systemid: 1001
        interfaces:
          - name: r3001-eth0
            # r3001-eth0  -> r2001-eth2
          - name: r3001-eth1
            # r3001-eth1  -> r2002-eth2
        v4prefixes:
          - address: 192.168.0.0
            mask: 24
            metric: 1
          - address: 192.168.1.0
            mask: 24
            metric: 1
```

Notar que en los nodos *leaf* el archivo de configuración solo contiene las interfaces conectadas al nivel superior y no a las conectadas a los ToR (en este caso servidores simulados con mininet). Esto es debido a que solo deben estar presentes aquellas interfaces en las que debe hablar el protocolo. Incluirlas puede causar la caída del demonio que ejecuta RIFT en el dicho nodo. Otro cuidado que se debe tener es con el `systemid`, no respetar la unicidad puede causar inconsistencias en las estructuras del protocolo.

4.3.3. Ejecución

Para iniciar RIFT en cada nodo se debe utilizar el python enviroment creado durante la instalación 4.3.1 mediante el siguiente comando (a falta de acomodar las rutas relativas):

```
env/bin/python3 rift  --ipv6-multicast-loopback-disable --telnet-port-file
/tmp/rift-python-telnet-port-node_name node_name.yaml
```

Esto es posible debido a que mininet simula los nodos mediante la creación de *network namespaces*, lo que permite tener namespaces que comparten todo menos el stack TCP-IP.

En la ruta que se indique para el archivo `rift-python-telnet-port-node_name`, en este caso `/tmp/`, se deja, para cada nodo, el puerto telnet correspondiente para el acceso al CLI. Para obtener una terminal en uno nodo basta con utilizar en el CLI de Mininet el siguiente comando: `xterm node_name`. Otra manera útil de ganar acceso a una terminal de un nodo, es mediante la herramienta *m* de Mininet. Para esto simplemente se debe ejecutar la herramienta pasando como parámetros el nombre del nodo y el comando (en este caso bash). Por ejemplo, para abrir una terminal en el primer *core* se utiliza el comando: `mininet/util/m r1001 bash`.

Toda la ejecución del ambiente, incluida la creación de la topología, generación de archivos de configuración RIFT y ejecución del demonio RIFT en cada nodo, se incluyó en un programa en python disponible en [2].

4.3.4. Pruebas realizadas

Para la prueba de conectividad se desarrolló un script de "ping all" con el objetivo de comprobar la convergencia del protocolo.

Por otra parte, para las pruebas de funcionamiento, la creación de un entorno en Mininet que ejecute RIFT permite utilizar las mismas pruebas diseñadas en el proyecto anterior para los protocolos incorporados en FRRouting. Con el objetivo de separar la creación del ambiente de las pruebas, se rediseñaron las mismas haciendo uso de la herramienta *m* mencionada anteriormente.

Como se mencionó en las pruebas del escenario rift-python de 4.2.3, los resultados no son los esperados debido a un bug latente en la implementación actual del rift-python que no instala en el kernel del nodo más de una ruta por prefijo.

Una prueba para detectar el bug mencionado en este ambiente es la siguiente:

```
r3001> show forwarding
IPv4 Routes:
+-----------+-----------+---------------------+
| Prefix    | Owner     | Next-hops           |
+-----------+-----------+---------------------+
| 0.0.0.0/0 | North SPF | r3001-eth0 10.1.0.1 |
|           |           | r3001-eth1 10.1.2.1 |
+-----------+-----------+---------------------+
```

Luego, obteniendo una consola en el nodo *r3001* mediante la herramienta `m` y ejecutando en ella el comando `route`, se tiene:

```
Kernel IP routing table
Destination     Gateway         Genmask          Iface
default         10.1.0.1        0.0.0.0          r3001-eth0
10.1.0.0        0.0.0.0         255.255.255.0    r3001-eth0
10.1.2.0        0.0.0.0         255.255.255.0    r3001-eth1
192.168.0.0     0.0.0.0         255.255.255.0    r3001-eth2
192.168.1.0     0.0.0.0         255.255.255.0    r3001-eth3
```

Se puede ver entonces como la tabla de forwarding del kernel de este nodo hoja no refleja la tabla de forwarding que se muestra en el CLI del nodo.

Una de las tareas claves que se realizaron para localizar el bug en el código del protocolo fue la ejecución del motor RIFT en modo "debug" en un nodo Mininet. La implementación del protocolo cuenta con una funcionalidad que simplifica esta tarea. En la ejecución se debe agregar el argumento `-l` *log_level*, donde *log_level* puede tomar los valores de "debug", "info", "warning", "error" o "critical". Además de esto, se puede añadir fácilmente mensajes personalizados en las clases del código fuente mediante `self.debug("mensaje")`.

De todas maneras la verificación de convergencia mediante el script desarrollado fue exitosa y luego de esto se realizaron pruebas manuales mediante el uso de los comandos presentados para el CLI del protocolo.

4.3.4.1 Verificación de rutas mediante el CLI

Dadas las características de funcionamiento mencionadas para RIFT es esperable que los nodos *leaf* tengan solo las rutas por defecto como lo muestra la ejecución de `show forwarding` mostrada para el nodo r3001. Para completar la visión de cada nivel, se ejecutó este comando en un nodo de cada uno de los niveles faltantes (*spine* y *core*).

```
r2001> show forwarding
IPv4 Routes:
+---------------+-----------+--------------------+
| Prefix        | Owner     | Next-hops          |
+---------------+-----------+--------------------+
| 0.0.0.0/0     | North SPF | r2001-eth0 10.0.0.1 |
|               |           | r2001-eth1 10.0.1.1 |
+---------------+-----------+--------------------+
| 192.168.0.0/24 | South SPF | r2001-eth2 10.1.0.2 |
+---------------+-----------+--------------------+
```

```
| 192.168.1.0/24 | South SPF | r2001-eth2 10.1.0.2 |
+----------------+-----------+---------------------+
| 192.168.2.0/24 | South SPF | r2001-eth3 10.1.1.2 |
+----------------+-----------+---------------------+
| 192.168.3.0/24 | South SPF | r2001-eth3 10.1.1.2 |
+----------------+-----------+---------------------+
```

Las rutas de la tabla de este nodo *spine* son las correctas. Notar que las dos rutas por defecto son hacia los *core* y luego tiene los cuatro prefijos asociados a los cuatro nodos hosts conectados a los dos nodos *leaf* que conoce.

```
r1001> show forwarding
IPv4 Routes:
+-----------------+-----------+----------------------+
| Prefix          | Owner     | Next-hops            |
+-----------------+-----------+----------------------+
| 192.168.0.0/24  | South SPF | r1001-eth0 10.0.0.2  |
+-----------------+-----------+----------------------+
| 192.168.1.0/24  | South SPF | r1001-eth0 10.0.0.2  |
+-----------------+-----------+----------------------+
| 192.168.2.0/24  | South SPF | r1001-eth0 10.0.0.2  |
+-----------------+-----------+----------------------+
| 192.168.3.0/24  | South SPF | r1001-eth0 10.0.0.2  |
+-----------------+-----------+----------------------+
| 192.168.4.0/24  | South SPF | r1001-eth1 10.0.4.2  |
+-----------------+-----------+----------------------+
| 192.168.5.0/24  | South SPF | r1001-eth1 10.0.4.2  |
+-----------------+-----------+----------------------+
| 192.168.6.0/24  | South SPF | r1001-eth1 10.0.4.2  |
+-----------------+-----------+----------------------+
| 192.168.7.0/24  | South SPF | r1001-eth1 10.0.4.2  |
+-----------------+-----------+----------------------+
| 192.168.8.0/24  | South SPF | r1001-eth2 10.0.8.2  |
+-----------------+-----------+----------------------+
| 192.168.9.0/24  | South SPF | r1001-eth2 10.0.8.2  |
+-----------------+-----------+----------------------+
| 192.168.10.0/24 | South SPF | r1001-eth2 10.0.8.2  |
+-----------------+-----------+----------------------+
| 192.168.11.0/24 | South SPF | r1001-eth2 10.0.8.2  |
+-----------------+-----------+----------------------+
```

```
| 192.168.12.0/24 | South SPF | r1001-eth3 10.0.12.2 |
+-----------------+-----------+----------------------+
| 192.168.13.0/24 | South SPF | r1001-eth3 10.0.12.2 |
+-----------------+-----------+----------------------+
| 192.168.14.0/24 | South SPF | r1001-eth3 10.0.12.2 |
+-----------------+-----------+----------------------+
| 192.168.15.0/24 | South SPF | r1001-eth3 10.0.12.2 |
+-----------------+-----------+----------------------+
```

La tabla asociada al nodo *core* también es correcta. Estos nodos ToF tienen una visión completa de la fábrica.

5. Link State Vector Routing

LSVR es un protocolo emergente para datacenters "hiperescalables" desarrollado por la IETF.

Se está especificando como un protocolo de enrutamiento híbrido que utiliza una combinación de mecanismos de enrutamiento de estado de enlace y path-vector. Para esto se utilizará el transporte IPv4/IPv6 existente y los formatos de paquetes y el manejo de errores de BGP-4 para facilitar la distribución de información de enrutamiento del vector de estado de enlace (LSV). Se pretende que un LSV se especifique como una estructura de datos compuesta por atributos de enlaces, información de vecinos y otros atributos potenciales que se pueden utilizar para tomar decisiones de enrutamiento. [8]

El protocolo LSVR está pensado como un protocolo de enrutamiento autónomo, incluso si se utilizan mecanismos y codificaciones de transporte BGP existentes, o si se comparte la misma sesión de transporte con otras familias de direcciones BGP existentes. Similar a los protocolos de enrutamiento existentes, el protocolo LSVR no combinará internamente los mecanismos de selección de ruta ni compartirá información de enrutamiento, excepto a través de métodos comunes de interacción externa en el RIB.

La estandarización de la funcionalidad del protocolo define los vectores de estado de enlace (LSV) y la selección de ruta path-vector estándar utilizando el algoritmo basado en Dijkstra SPF, la mecánica del protocolo BGP-4 y la codificación BGP-LS NRLI.

La idea básica es aprovechar BGP-LS para el transporte de información y luego ejecutar cálculos SPF en los "LS-DBs" resultantes. Esto es logrado definiendo un *Network Layer Reachability Information* (NLRI) difundido con BGP-LS/BGP-LS-SPF AFI/SAFI. [11]

Si bien otros protocolos, como *Intermediate System to Intermediate System* (IS-IS) y *Open Shortest Path First* (OSPF), pueden realizar la misma tarea, LSVR promete ser más simple, con lo que genera un atractivo para datacenters muy grandes donde el enrutamiento OSPF e IS-IS no escala.

5.1. BGP-SPF

El draft *Shortest Path Routing Extensions for BGP Protocol*[12] describe una solución que aprovecha BGP Link-State y el algoritmo SPF, similar a los IGPs como OSPF. Para esto, BGP ejecuta Dikjstra en vez del proceso de *Best Path Decision*. [3]

Con este nuevo enfoque se tiene:

- Los atributos de *Next-Hop* y *Path Attributes* vienen dados con BGP Link-State Address Family.

- Las fases 1 y 2 del proceso de *Best Path Decision* son reemplazados por el algoritmo SPF.
- La fase 3 del proceso de *Best Path Decision*(tie break) se puede omitir ya que NLRI es único por BGP speaker.
- Es necesario asegurarse que siempre se use la versión más reciente de NLRI. Aumentado con números de secuencia.

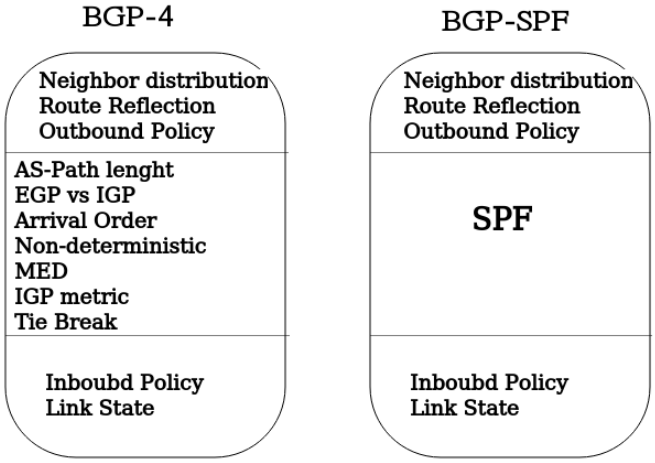

Figura 11: BGP4 vs. BGP-SPF

BGP-SPF necesita link neighbor discovery, liveness, y addressability. En este sentido el protocolo LSoE [4] tiene la intención de proporcionar liveness de capa 2, link neighbor discovery, un intercambio de encapsulaciones compatibles (IPv4, IPv6, ...), descubrimiento de direcciones, etc, a través de raw Ethernet. El protocolo LSoE en sí no mantiene el estado del enlace, sino que da soporte al protocolo LSVR con el descubrimiento, el intercambio de información y el liveness.

5.2. Resumen

- LSVR se basa en el conocido protocolo BGP.
- Existe un draft *Shortest Path Routing Extensions for BGP Protocol* [12] en el que se está trabajando de forma activa.
- Cuenta en particular con el apoyo de Arrcus inc. quien ya ofrece soluciones con el protocolo soportado, y le está dando un gran impulso y difusión. [7]
- Tiene como objetivo resolver el problema de las limitaciones de escala del flooding de IGP aprovechando BGP-LS como transporte para la información de LS y modificando el proceso de decisión de BGP para comprender este cambio y tomar ventaja.
- Permite una escala y convergencia más rápida en fábricas de tipo IP con topología Clos en crecimiento. [11]

- Permite arquitecturas de controlador de ruta centralizadas. Tiene en cuenta las SDN desde el diseño.
- Una desventaja es que se combina dos protocolos y se cambia sus respectivos comportamientos de manera que pueden conducir a consecuencias imprevistas.

6. Conclusiones

En este trabajo se estudiaron dos protocolos en vías de estandarización diseñados para el routing en un datacenter con topología Clos network o derivada.

En el caso de RIFT se logró entender sus principales funcionalidades mediante el estudio del draft disponible. Este protocolo al considerar las topologías fat-tree desde el diseño, logra cubrir una variedad de requerimientos específicos y disminuir considerablemente las principales desventajas presentes en los algoritmos de routing actuales como BGP, ISIS u Open Fabric al ser desplegados en este tipo de topologías.

RIFT, al ser un híbrido entre un algoritmo de estado de enlace y uno de vector de distancias, logra suprimir la mayoría de las desventajas que presentan cada uno de estos.

Las puesta en marcha del protocolo open source disponible tuvo sus desafíos, algunos propios de configuración de la tecnología, y otros por la naturaleza de la fuente, la cual está actualmente en desarrollo y cuenta con escasa documentación así como bugs latentes. Uno de los grandes aportes de este trabajo es la puesta en marcha del protocolo en un ambiente Mininet, donde la falta de documentación para la generación de los archivos de configuración del protocolo fue un problema crítico. Además, la experimentación con el ambiente proporcionado con el protocolo fue de gran aporte tanto para entender el protocolo, como para la simulación de infraestructura de red mediante la creación y configuración de network namespaces.

En cuanto a las pruebas, se logró implementar las existentes de un ambiente Mininet para el ambiente de network namespaces brindado por la herramienta incorporada al desarrollo del protocolo. Los resultados obtenidos comprueban el funcionamiento del protocolo en ambos ambientes.

Por otra parte, se creó una idea general de lo que se propone como protocolo LSVR. Aunque el avance en este protocolo no es tan notable como el de RIFT, se logró estudiar los principales pilares hasta ahora conocidos y realizar un análisis de las ventajas y desventajas que este presenta.

Horizontalmente al estudio y experimentación con estos protocolos, se ganó un fuerte conocimiento sobre la infraestructura y topología de los datacenters. Este dominio está en la mira debido a la proliferación de los servicios en la nube y el crecimiento que han tenido los datacen-

ters hoy en día. Este tipo de infraestructura en crecimiento requiere de protocolos de routing específicos, que contemplen las necesidades de un datacenter desde el diseño. Hoy en día se tiene en cuenta que no es lo mismo realizar este mecanismo en un datacenter, que en Internet, y esto lleva al nacimiento de protocolos como RIFT y LSVR.

Referencias

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[2] Leonardo Alberro. rift-mininet. `https://gitlab.fing.edu.uy/lalberro/rift-mininet`. Último acceso: 2019.

[3] Randy Bush. An Approach to Routing in a Clos. `https://2019.apricot.net/assets/files/APKS756/an-approach-to-routing-in-a-clos.pdf`. Último acceso: 2019.

[4] Randy Bush, Rob Austein, and Keyur Patel. Link state over ethernet. Internet-Draft draft-ietf-lsvr-lsoe-01, IETF Secretariat, February 2019. `http://www.ietf.org/internet-drafts/draft-ietf-lsvr-lsoe-01.txt`.

[5] Thomas H. Clausen and Philippe Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626, October 2003.

[6] Linux Foundation. FRRouting. `https://frrouting.org/`. Último acceso: 2019.

[7] Antone Gonsalves. Latest Arrcus protocols for webscale data centers. `https://searchnetworking.techtarget.com/news/252460295/Latest-Arrcus-protocols-for-webscale-data-centers`, 2019. Último acceso: 2019.

[8] IETF. Charter - Link State Vector Routing. `https://datatracker.ietf.org/doc/charter-ietf-lsvr/`, 2018. Último acceso: 2019.

[9] Deepankar Medhi and Karthik Ramasamy. *Network Routing : Algorithms, Protocols, and Architectures.*, volume Second edition of *The Morgan Kaufmann Series in Networking*. Morgan Kaufmann, 2018.

[10] Andreas Pantelopoulos. Github - mininet fattree. `https://github.com/panandr/mininet-fattree/blob/master/fattree.py`. Último acceso: 2019.

[11] Keyur Patel. Link State Vector Routing. `https://www.enog.org/wp-content/uploads/presentations/enog-16/4-ENOG-2019.pptx`, 2019. Último acceso: 2019.

[12] Keyur Patel, Acee Lindem, Shawn Zandi, and Wim Henderickx. Shortest path routing extensions for bgp protocol. Internet-Draft draft-ietf-lsvr-bgp-spf-06, IETF Secretariat, September 2019. `http://www.ietf.org/internet-drafts/draft-ietf-lsvr-bgp-spf-06.txt`.

[13] Tony Przygienda, Alankar Sharma, Pascal Thubert, and Dmitry Afanasiev. Rift: Routing in fat trees. Internet-Draft draft-ietf-rift-rift-08, IETF Secretariat, September 2019. `http://www.ietf.org/internet-drafts/draft-ietf-rift-rift-08.txt`.

[14] Bruno Rijsman. Feature List. https://github.com/brunorijsman/rift-python/blob/master/doc/features.md. Último acceso: 2019.

[15] Bruno Rijsman. Routing In Fat Trees (RIFT) implementation in Python. https://github.com/brunorijsman/rift-python. Último acceso: 2019.

[16] Bruno Rijsman. RIFT Hackathon: Chaos Monkey Testing, IETF 104. http://bit.ly/rift-hackathon-ietf-104, Febrero 2019.

[17] Jeff Tantsura. ENOG15 - Routing in FAT Trees . https://www.enog.org/wp-content/uploads/presentations/enog-15/32-RoutingDC-v1.4-ENOG.pdf, Abril 2018.

[18] X. Yuan. On nonblocking folded-clos networks in computer communication environments. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 188–196, May 2011.