UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Escalabilidad del Enrutamiento en Data Centers Masivos

Proyecto de Grado

Autores:

Maximiliano Lucero
Agustina Parnizari

Supervisores:

Eduardo Grampín
Leonardo Alberro
Alberto Castro

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Instituto de Computación, Facultad de Ingeniería
Universidad de la República

19 de marzo de 2021

Agradecimientos

Quisiéramos agradecer en primer lugar a nuestros supervisores por la guía y asistencia en en el transcurso de este proyecto. Queremos agradecer también a nuestros colegas de la Universidad Roma Tre, Italia, por sus importantes aportes para la creación del disector de RIFT y asistencia en la configuración del entorno de pruebas. Dedicamos un agradecimiento a Bruno Rijsman, Tony Przygienda y todo el grupo de investigación de RIFT por sus valiosos comentarios que nos guiaron a la solución realizada. Por último queremos agradecer a nuestras respectivas familias y amigos por el apoyo invaluable durante el transcurso de la carrera. Este trabajo no habría sido posible sin ustedes.

Resumen

Los Data Centers masivos han tenido un papel fundamental en la transformación gradual que ha sufrido la arquitectura de Internet, desde una red de redes hacia una red dirigida por contenidos. Este tipo de Data Centers brinda diferentes servicios esenciales, como son el poder de cómputo, el almacenamiento masivo y la replicación. A lo largo de los últimos años, la gran cantidad de datos manejada por los Data Centers, ha obligado a estos últimos a escalar su infraestructura (servidores, switches, routers), creando la necesidad de escalar en la comunicación entre dispositivos y del ancho de banda en varios órdenes de magnitud superior al del tráfico WAN clásico de Internet.

Se han propuesto diferentes soluciones al problema del enrutamiento en los Data Centers. Algunas están basadas en protocolos de enrutamiento distribuido ya existentes (BGP, IS-IS), mientras que otras soluciones se han basado en protocolos nuevos (RIFT) adaptados a los problemas que presentan las topologías más comunes de los Data Centers (Fat-Tree). También, se han presentado soluciones basadas en paradigmas de enrutamiento centralizado como las redes definidas por software (SDN, por su sigla en inglés).

Este trabajo se centró en estudiar, analizar y comparar el diseño y rendimiento de estas distintas soluciones. Se realizó en primer lugar un análisis teórico de los protocolos BGP y RIFT, estudiando la complejidad de mensajería en el *bootstrap*[1]. La elección de dichos protocolos se basó, en el caso de BGP, en el hecho de que es el estándar de facto de enrutamiento en el Data Centers, mientras que RIFT fue elegido porque presenta una idea alternativa la cual combina características entre un protocolo de estado de enlace y un algoritmo de vector de distancia. A su vez, es importante destacar que RIFT

[1]Término que proviene del inglés y se utiliza a lo largo de este trabajo: define la etapa de arranque de una topología.

fue específicamente desarrollada para topologías Fat-Tree, muy comunes en el diseño de los Data Centers hoy en día.

Por otro lado, con un enfoque experimental, se diseñó una metodología de experimentación utilizando el framework de Kathará, que permitió analizar y comparar cuantitativamente todos los protocolos que fueron de interés en el presente estudio. En particular, se desarrollaron pruebas para comparar la cantidad de mensajes de RIFT y BGP al realizar el *bootstrap* de la topología. Los resultados experimentales se compararon con los resultados teóricos, con el objetivo de validar el análisis realizado. A su vez, se observó que si bien RIFT envía una cantidad menor de mensajes relevantes a la convergencia del enrutamiento, BGP en sí genera en total una cantidad menor de mensajes.

La metodología de experimentación diseñada se basa en la disección de mensajes, lo cual no era posible en el caso de RIFT al no existir una implementación de inspección para el protocolo. Por dicho motivo, se desarrolló un disector de Wireshark que permitiera la inspección de paquetes RIFT. A su vez, este disector, también fue de utilidad para la depuración del protocolo por parte de los desarrolladores del mismo.

Se incursionó también en soluciones de tipo SDN, con el objetivo de compararlas con los protocolos distribuidos analizados. A pesar de que se encontraron fuertes limitaciones en el tipo de topologías que la herramienta para enrutamiento SDN soportaba, las pruebas permitieron comprender mejor dicho paradigma.

Tabla de contenidos

Capítulo 1

Introducción

Si se quisiera introducir de manera simple el concepto de Data Center, se podría decir que es una localización centralizada donde equipamiento de red y cómputo conviven con el propósito de recolectar, almacenar, procesar, distribuir y/o permitir acceso a grandes cantidades de datos. Han existido de alguna manera u otra desde la llegada de las computadoras ["¿Qué es un Data Center y cuál es su importancia?", 2020].

En la época de las primeras computadoras, que ocupaban habitaciones enteras, un Data Center consistía comúnmente de una sola computadora. A medida que el equipamiento se empezó a hacer más barato y pequeño, y las necesidades de procesamiento de datos comenzaron a aumentar, se pasó a tener redes de múltiples servidores juntos de manera de incrementar el poder de procesamiento. Un Data Center hoy puede fácilmente tener miles de servidores activos sin parar en ningún momento del día ["What is a Data Center?", 2021]. Los Data Centers masivos particularmente comprenden cientos de miles de componentes de red, teniendo un espacio aproximado para entre 3.001 y 9.000 racks ["Landscape of the Data Center Industry", 2017], por lo cual se puede ver que las dimensiones han variado drásticamente, existiendo inclusive Data Centers de mayor tamaño.

En la sociedad y economía digital de hoy en día, cumplen un rol fundamental, debido a que albergan una parte muy importante del contenido que existe en la red. Es por esto que para las organizaciones es de vital importancia trabajar en su seguridad y confiabilidad. Debido a que las operaciones

de TI[1] son fundamentales para la continuidad del negocio, comúnmente incluyen componentes de respaldo y reservas de energía eléctrica, conexiones para comunicación de datos, control ambiental (e.g. aires acondicionados, sistemas anti-incendios) y varios dispositivos de seguridad.

Las necesidades de los nuevos Data Center presentan un desafío para las comunicaciones. Por un lado se necesitan topologías que permitan escalar de forma flexible tanto a nivel de equipamiento como conectividad, así como robusta a fallas, dando por hecho que las mismas ocurrirán. Por otro lado el enrutamiento debe ser eficiente en el uso de los recursos y la convergencia, así como proveer mecanismos robustos ante fallas. Por último, el forwarding de mensajes debe ser eficiente, permitiendo utilizar todos los recursos disponibles para el envío de datos.

En la última década se han presentado soluciones adaptadas los requerimientos de un Data Center. Las topologías de tipo Clos han cobrado interés y se han adaptado protocolos como BGP, o desarrollado nuevos como OpenFabric o RIFT[1]. También han surgido las SDN[2] como alternativa al enrutamiento distribuido, permitiendo un mayor control de los operadores. En cuanto al forwarding de paquetes se han implementado distintos métodos que permiten enviar los mensajes por distintos caminos, beneficiándose de la redundancia de las topologías utilizadas.

El siguiente trabajo se centra en el análisis teórico, la experimentación y comparación de distintas soluciones de enrutamiento en Data Centers masivos. Particularmente el análisis teórico, así como el desarrollo de una metodología de experimentación eficiente presentan los mayores desafíos del mismo.

Este documento se estructura de la siguiente manera. En el capítulo 2 se presenta el relevamiento de antecedentes en el cual se estudian los requerimientos de un Data Center masivo, las topologías más utilizadas, distintos algoritmos de enrutamiento, se presentan los problemas de forwarding y gestión Cloud. En el capítulo 3 se presenta un análisis teórico de dos algoritmos de

[1] Tecnología de la Información.
[1] Routing in Fat Trees.
[2] Software Defined Networks.

enrutamiento distribuido, RIFT y BGP. En el capítulo 4 se presenta el desarrollo de una herramienta: un disector de mensajes para el protocolo RIFT. El capítulo 5 se pone foco en la experimentación sobre un entorno virtual tanto de una solución centralizada como de los protocolos distribuidos BGP y RIFT. Por último, el capítulo 6 desarrolla las conclusiones que se obtuvieron del presente trabajo y propone líneas de investigación para futuros proyectos.

Capítulo 2

Revisión de antecedentes

Se presenta un estudio del estado del arte del enrutamiento en Data Centers de gran escala. Se abordarán los requerimientos actuales, las topologías más utilizadas y protocolos tanto ampliamente utilizados como en desarrollo.

2.1. Requerimientos del enrutamiento en el Data Center

Las aplicaciones que son utilizadas hoy en día han cambiado su arquitectura con respecto a los programas de procesamiento en lotes o las aplicaciones Cliente-Servidor que solían contener los centros de datos. Tecnologías como Big Data, Cloud Computing o Microservicios son ejemplos claros de este cambio y presentan requerimientos distintos a nivel de conectividad, fiabilidad y escalabilidad [Caiazzi, 2019; Dutt, 2017].

- **Interconectividad entre servidores:** Las arquitecturas cliente-servidor consistían en la conexión de distintos clientes comunicándose con servidores que manejaban las consultas completamente o se comunicaban con servidores de bases de datos, pero con una limitada interconexión en la red interna del Data Center.
  En contraste, una aplicación moderna de búsqueda puede intercomunicarse con cientos de servidores antes de retornar con una respuesta al cliente. En la nube, una máquina virtual de un cliente puede residir dentro de múltiples nodos interconectados. En arquitecturas de Microservicios, una función se descompone en funciones más simples y se procesan en distintos nodos, interconectándose para obtener el resultado final.

Esta comunicación dentro del Data Center se suele llamar tráfico Este-Oeste, mientras que la comunicación hacia los clientes se llama tráfico Norte-Sur. Hoy en día el tráfico Este-Oeste supera en varios órdenes de magnitud al tráfico Norte-Sur, por lo cual se necesitan topologías y protocolos que optimicen los recursos de red en este sentido.

- **Escalabilidad:** Los Data Center modernos están compuestos de cientos de miles de servidores en una única locación física. Esto en conjunto con el gran caudal de tráfico Este-Oeste conlleva a requerimientos de conectividad estrictos que permitan escalar y soporten el movimiento de servicios y máquinas virtuales dentro del centro de datos.
- **Resiliencia:** Al contrario de anteriores arquitecturas, donde se asumía que la red era confiable, las aplicaciones modernas asumen que van a ocurrir fallas. Esto permite diseñar aplicaciones robustas que buscan limitar el efecto de esas fallas y que el usuario final no se vea afectado por problemas a nivel de red o servidores. Las topologías de Data Centers deben proveer redundancia en la red y los protocolos de enrutamiento deben tener mecanismos para detectar y actuar ante dichas fallas.

Cualquier Data Center moderno debe cumplir con estos requisitos. Por otro lado algunos proveedores de servicio de nube pública o privada, deben además proveer un rápido despliegue de redes virtuales.

Los modelos tradicionales estaban diseñados para escalar desplegando switches y routers más grandes, lo que se conoce como un modelo *scale-in*. Estos switches grandes son costosos y complejos, además de que pueden escalar hasta cierto punto. Cuanto más grandes y complejos son mayor es el daño que produce una falla.

Las topologías de Data Center modernas escalan agregando dispositivos a la red, a lo cual se le llama *scale-out* y buscan que la interconexión con estos nuevos nodos sea lo menos costosa posible. Esto permite tener dispositivos menos complejos y no verse atados a ciertos proveedores, invirtiendo en equipamiento a medida que es necesario.

2.2. Topologías utilizadas

Al momento de diseñar y construir un Data Center, existen dos opciones a nivel de infraestructura. Una de ellas es usar hardware propietario especializado, el cual es capaz de escalar a clústeres de miles de nodos con una alta tasa de ancho de banda (e.g. Google Jupiter [Singh y col., 2016] o switches InfiniBand ["InfiniBand in the Enterprise Data Center", 2006]), opción que tiene un enfoque más expansivo. La otra opción consiste en utilizar switches *commodity*[1] Ethernet y routers para conectar nodos del Data Center. Este último enfoque desemboca en una infraestructura clásica de gestión que necesita atender algunos desafíos en términos de:

- **Escalabilidad**: debería ser posible escalar horizontalmente para poder expandir el clúster. Esto es, para expandir la fábrica debe ser posible comprar hardware similar al que ya está siendo utilizado.
- **Ancho de banda**: los hosts en la fábrica deben comunicarse entre sí usando la totalidad del ancho de banda de sus interfaces de red locales.
- **Economía de la escalabilidad**: el uso de commodity hardware tiene que llevar a un balance económico en la escalabilidad para fijar la base de los Data Centers de gran escala en los switches *off-the-shelf*[2] Ethernet.

A continuación nos vamos a centrar en la segunda opción e introducir topologías típicas de Data Center, describiendo en detalle la topología Fat-Tree y revisando otras topologías utilizadas.

2.2.1. Fat-Tree

Una topología Fat-Tree es un caso especial de una red de Clos [Clos, 1953], como se puede observar en la en la Figura 2.1. La idea de usar este tipo de topología fue propuesta originalmente en [Leiserson, 1985] para supercomputación y fue adaptada por los Data Centers.

Las arquitecturas Fat-Tree habituales hoy en día consisten en árboles de dos o tres niveles de switches o routers. Un diseño típico de tres niveles, está

[1] En el contexto de TI, refiere a dispositivos o componentes de bajo costo, de amplia disponibilidad y que por lo general son intercambiables con otros dispositivos de su mismo tipo.

[2] Productos comerciales que están listos para ser usados sin modificaciones.

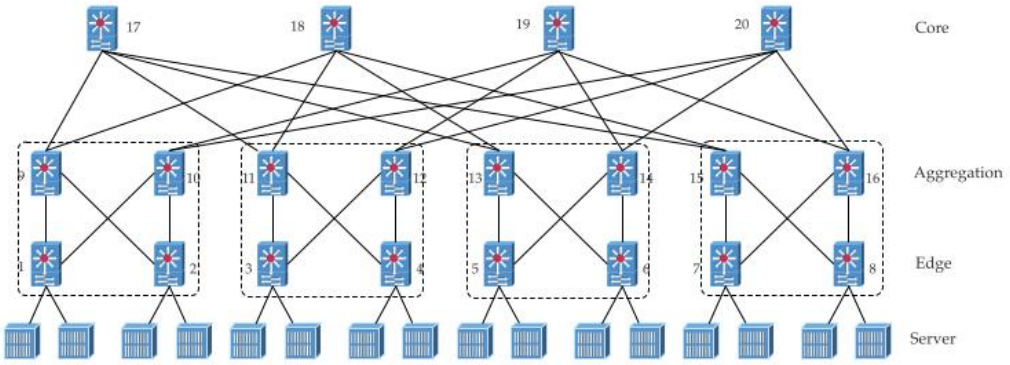


Figura 2.1: Topología Fat-Tree con 4 PoDs [Medhi y Ramasamy, 2018]

compuesto por una capa de agregación que enlaza los diferentes *PoDs* (Point-of-Delivery) de dos niveles de switches. Un diseño de dos niveles podría ser considerado como un único PoD donde los *spines* actúan como *ToF* (Top-of-Fabric), y también se le denomina topología *spine-leaf*.

A continuación, definimos la terminología utilizada habitualmente en este tipo de escenarios:

- **Point-of-Delivery**: 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.
- **Radix**: de un switch, es básicamente el número de puertos que proporciona. También se le llama *fanout*.
- **Top-of-Fabric Plane o Partition**: En las 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.

- P: es usado para denotar el número de PoDs en una topología.
- S: es usado para denotar el número de nodos ToF en una topología.
- K: es usado para denotar el número de puertos en el *Radix* de un switch que apuntan hacia el norte o sur. Se utiliza más específicamente K_LEAF para la cantidad de puertos que apuntan al sur y K_TOP para la cantidad de puertos que apuntan hacia el norte.
- N: es usado para denotar la cantidad de planos independientes en una topología.
- R: es usado para denotar un factor de redundancia. En las topologías *single plane*[1] K_TOP equivale a R.

2.2.1.1. Single Plane

Se tiene un número P de PoDs conectados con un número S de ToFs. Un nodo dentro de un PoD tiene una cierta cantidad de puertos o *Radix*, de los cuales $K\_LEAF = Radix/2$ son utilizados para conectarse con hosts hacia el sur, mientras que los otros $K\_TOP = Radix/2$ son utilizados para conectar con los switches ToP. El Radix de un nodo ToP puede diferir al de un Leaf, aunque por lo general se utiliza el mismo tipo de switch para ambos casos, formando un cuadrado $(K * K)$. En el caso general, es posible tener switches con K_TOP puertos hacia el sur en los nodos del nivel superior del PoD, con un valor no necesariamente igual a K_LEAF. Los K_TOP Leafs están interconectados con los K_LEAF ToPs, en una conexión que se puede visualizar como un *crossbar*. Lo que se obtiene de esto es que, de no existir alguna caída de enlace o nodo, un paquete que ingrese a un PoD por cualquier puerto desde el norte puede ser enrutado a cualquier puerto hacia el sur del PoD y viceversa. Los PoDs se conectan entre sí a través de los ToF, que se ubican en el límite norte de la fábrica. Se dice que un ToF no está *particionado* si y sólo si cada ToP está conectado con cada ToF, y esto es lo que se conoce como una configuración *single plane*. Para poder obtener una conectividad $1:1$ entre los ToF y los Leaf, existen K_TOP ToFs porque cada puerto de un ToP se conecta con

[1] Un único plano de ToFs.

un ToF diferente, y por el mismo motivo existen K_LEAF ToPs. Entonces, cada ToF requiere de $P * K\_LEAF$ puertos para conectar con cada uno de los K_LEAF ToPs de los P PoDs. Esto introduce una cota inferior (denominada "*single plane limit*") para esta configuración en cuanto a la cantidad de puertos disponibles de un ToF, que debe ser al menos $P * K\_LEAF$.

2.2.1.2. Multi-Plane

Los ToF se pueden particionar en un número N de planos idénticamente cableados, donde N debe ser un divisor entero de K_LEAF. Esto sirve para poder escalar más allá del límite que impone la configuración single plane. Se sigue cumpliendo la conectividad $1:1$ y la simetría del single plane, pero esta vez con $K\_TOP * N$ ToFs, cada uno de ellos con $P * K\_LEAF/N$ puertos. Si se establece $N = 1$ se obtiene un Spine no particionado, mientras que $N = K\_LEAF$ implica un Spine particionado maximalmente. Luego, si R es un divisor cualquiera de K_LEAF, entonces $N = K\_LEAF/R$ es una cantidad factible de planos y R un factor de redundancia. En la Figura 2.2 se puede observar un ejemplo de configuración Multi-Plane.

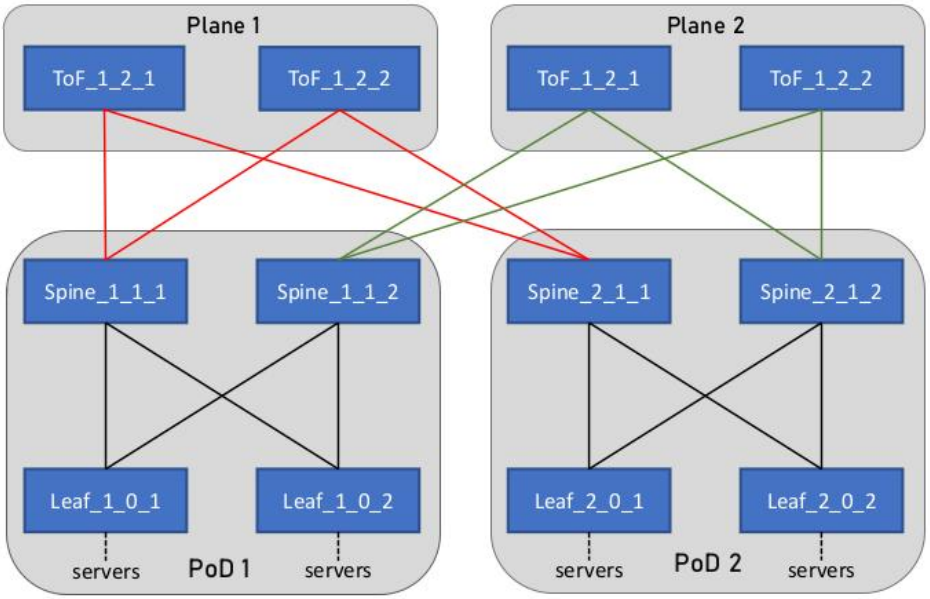


Figura 2.2: Topología Multi-Plane con 2 PoDs

2.2.2. Otras topologías

A continuación, se describen otras topologías factibles de encontrar en arquitecturas de Data Centers.

2.2.2.1. VL2

VL2 [Greenberg y col., 2009] es una topología basada en una red de Clos con enrutamiento IP y commodity switches al igual que Fat-Tree, pero con modificaciones. Está basada en la construcción de una red *overlay* donde todo el tráfico se encapsula en el origen y se desencapsula en el destino. En VL2 se separa la dirección local (*Location Address*) de la dirección de aplicación (*Application Address*) para hacer todo más ágil y permitir migrar fácilmente las VMs. Para esto se tiene un *Central Directory* que realiza las correspondencias y búsquedas Location-Application, mejorando la escalabilidad, confiabilidad y performance de búsquedas. También utiliza técnicas de *Valiant Load-Balancing* [Zhang-Shen y McKeown, 2008] para distribuir flujos de tráfico aleatoriamente entre los distintos caminos y nodos.

2.2.2.2. BCube

BCube [Guo y col., 2009] es una topología que se define de forma recursiva y se organiza modularmente. O sea, una topología $BCube_k$ es construida de manera recursiva a partir de una $BCube_{k-1}$. Como paso base con $k = 0$, un $BCube_0$ tiene n hosts conectados a un switch con n puertos. Luego, un $BCube_1$ se construye con n $BCube_0$s usando n switches con n puertos cada uno, como se observa en la Figura 2.3.

De manera más general, un $BCube_{k+1}$, con $k \geq 0$ se construye a partir de n $BCube_k$s con n^{k+1} switches de n puertos. Un $BCube_k$ tiene $N = n^{k+1}$ hosts, cada host con $k+1$ puertos. Cada host se conecta con cada uno de los niveles desde el nivel 0 al nivel k. Sean los n switches en el nivel k numerados como $< k, 0 >, < k, 1 >, ..., < k, k >$ y sus puertos numerados desde 0 a $n-1$, entonces se conecta el puerto del nivel k del i-ésimo host, donde $i = 0, 1, ..., n^k - 1$, al j-ésimo puerto del i-ésimo switch en el nivel k. De esta manera, en total hay $n * n^k = n^{k+1}$ hosts. Cada host se conecta con un switch de cada nivel (en total se conecta con $k+1$ switches). En conclusión, la cantidad de enlaces en

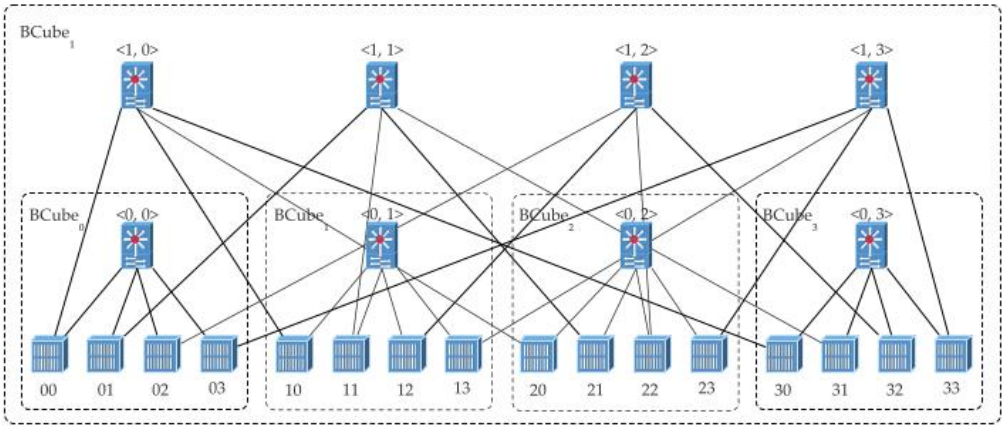


Figura 2.3: $BCube_1$ con $n = 4$ [Medhi y Ramasamy, 2018]

un $BCube_k$ es de $(k + 1) * n^{k+1}$.

Un punto relevante a destacar en este tipo de topologías es que un switch se conecta con hosts, pero jamás con otro switch. Además, un camino entre dos hosts cualquiera puede contener otro host como nodo intermedio. Esto genera una gran desventaja en una arquitectura BCube ya que un host requiere realizar acciones de reenvío de paquetes además de su función principal de ser un host, lo que no es algo deseable que ocurra dentro de un Data Center.

2.2.2.3. DCell

DCell [Guo y col., 2008] es otra arquitectura recursiva centrada en los hosts, que comparte similitudes con BCube, donde los servidores puede conectarse entre sí utilizando múltiples NICs[1]. Los nodos de la red (DCells de niveles inferiores) están compuestos por una cantidad n de hosts conectados entre sí y a un commodity switch que es usado para conectar con otros cells. Entonces, un DCell de nivel superior se forma conectando entre sí distintos cells de niveles inferiores. Esto convierte a la arquitectura DCell en una altamente escalable que puede ser expandida fácilmente y no tiene puntos únicos de falla. Se utiliza *DCell Fault-Tolerant Routing* (DFR) como algoritmo de enrutamiento, el cual puede gestionar el enrutamiento en los hosts y que puede prevenir la mayoría de problemas comunes del Data Center. En la Figura 2.4 se puede observar un ejemplo de arquitectura DCell.

[1]Network Interface Cards.

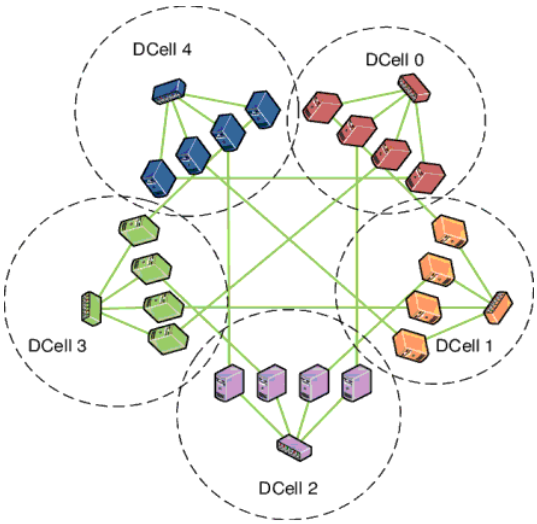


Figura 2.4: Ejemplo de arquitectura DCell [Pries y col., 2012]

2.2.2.4. Jellyfish

Jellyfish [Singla y col., 2011] es definido como un diseño de Data Center expansible incrementalmente, basado en un grafo aleatorio, y que provee un alto ancho de banda. Este tipo de topología soluciona algunas limitaciones de topologías Fat-Tree, proveyendo división completa del ancho de banda, utilización de más hosts al mismo costo, y además ofrece una rendimiento promedio más alto con la misma cantidad de falla de enlaces. El problema más grande de este diseño es que no puede utilizar de manera efectiva todos los caminos generados por los protocolos de enrutamiento actuales, debido a que la cantidad de saltos en los distintos caminos paralelos puede diferir en un número muy grande.

2.3. Enrutamiento en el Data Center

Tradicionalmente los Data Centers han utilizado protocolos distribuidos como OSPF e IS-IS los cuales hasta el día de hoy son ampliamente aceptados como IGP dentro de organizaciones, sin embargo estos protocolos no se ajustan a los requerimientos de los Data Centers modernos. Como protocolos de estado de enlace no son adecuados para topologías con una interconectividad densa, como es el ejemplo de Fat-Tree, ya que se genera una gran cantidad de flooding y los cambios atraviesan toda la red, sobrecargando la misma de mensajes del plano de control generalmente innecesarios.

Se vió por lo tanto la necesidad de encontrar protocolos que solucionen el problema de enrutamiento eficiente y robusto, explotando los beneficios de topologías utilizadas en el Data Center. Para ello se modificaron protocolos existentes como BGP y se desarrollaron protocolos nuevos como OpenFabric o RIFT.

Por otro lado, en conjunto con el crecimiento de los Data Center definidos por software, surgieron las redes definidas por software, que separan el plano de control del plano de forwarding, permitiendo un control centralizado de la red que resuelve el enrutamiento con un conocimiento completo de la topología. En la figura 2.5 se pueden ver ambos modelos para los cuales presentaremos distintas soluciones, específicamente de enrutamiento en topologías de tipo Clos.

Si bien todas las soluciones a presentar tienen distintas características, comparten algunas de ellas en común, las cuales buscan cumplir con los requerimientos planteados:

- Deben ser escalables a las proporciones de un Data Center.
- Deben ser robustos a fallas, teniendo mecanismos para detectarlas y reducir su impacto.
- Deben permitir múltiples caminos entre cualquier par de servidores, balanceando la carga entre ellos.
- El tráfico del plano de control no debe afectar al tráfico de datos. En cuanto a los protocolos distribuidos se podría decir que se deben evitar mensajes innecesarios por flooding.
- La configuración tiene que ser mínima y sencilla para evitar errores en la operación.

2.3.1. Enrutamiento distribuido

Introduciremos en primer lugar los algoritmos de enrutamiento distribuido. Todos ellos tienen la particularidad de compartir la red de datos con el plano de control de los protocolos. Frente a protocolos ruidosos esto influye directamente en la carga que sufre cada equipo así como los enlaces, sobre todo al

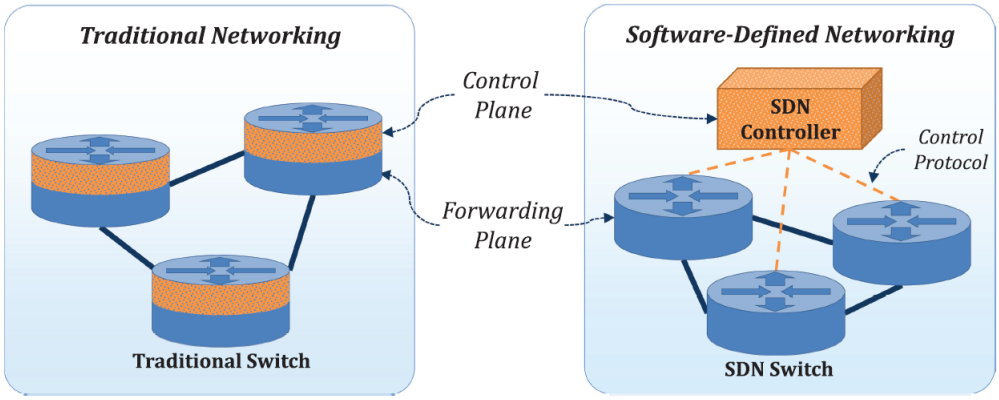


Figura 2.5: Enrutamiento distribuido y centralizado [Son y Buyya, 2018a]

momento de escalar.

En este tipo de protocolos nos encontramos con el problema de descubrimiento de la topología para tener conectividad. Los protocolos que vamos a presentar se basan en algoritmos de estado de enlace, vector de distancia o una combinación de los mismos.

2.3.1.1. BGP

BGP es el protocolo de facto de enrutamiento en el *backbone* de Internet [Rekhter y col., 2006], pero también se ha convertido en el protocolo de facto de enrutamiento en el Data Center [Lapukhov y col., 2016]. Por un lado la presencia de implementaciones estables y robustas inclusive de código abierto, ha propiciado su uso. Por otro lado genera menos flooding que los protocolos de estado de enlace tradicionales (OSPF, IS IS por ejemplo) lo cual permite que la topología escale sin aumentar considerablemente la mensajería. A su vez soporta una gran cantidad de protocolos de forma nativa como IPv4, IPv6, MPLS o VPN.

BGP fue diseñado para enrutamiento entre sistemas autónomos y su uso principal está orientado a estos, por lo que para utilizarse en Data Centers necesita algunas modificaciones. En [Dutt, 2017] se detallan las mismas, las cuales introduciremos a continuación, mencionando la necesidad de estos cambios.

En primer lugar entre dominios se prefiere estabilidad en la red a cambios rápidos, por ese motivo generalmente se retrasan las notificaciones de cambios. Sin embargo en Data Centers se necesita que las actualizaciones se hagan lo más rápido posible. En segundo lugar, las fallas de cualquier enlace en BGP se transmiten por todos los nodos, esto debería evitarse en los Data Centers. Por último, mientras que en BGP se construye un único mejor camino cuando los prefijos se conocen desde distintos AS, en Data Centers se requieren múltiples caminos.
Estas modificaciones se logran a partir de los siguientes cambios en el uso y la configuración del protocolo:

- **eBGP en lugar de iBGP:** si bien un Data Center puede considerarse un único sistema autónomo y ello llevaría a creer que iBGP es la solución adecuada, es más complejo que eBGP en cuanto a implementación y configuración. Ademas presenta limitaciones al momento de elegir múltiples caminos. Por estas razones se implementa eBGP dentro del Data Center, donde cada router BGP es considerado un sistema autónomo. Se debe tener en cuenta que no se interactúa con un IGP, sino que eBGP cumple con la función de protocolo de enrutamiento interno.
- **Numeración AS:** Debido a que cada router es un sistema autónomo, se deben identificar los routers por un número de AS. Para ello se tienen que utilizar números de AS privados, dado que los ASN públicos están asignados y son bien conocidos, pudiendo ocasionar filtraciones de rutas internas que provoquen un secuestro de rutas a nivel global. Para ello se utilizan los números de AS de 4 bytes, que permiten millones de números ASN, al contrario de los ASN de 2 bytes que permiten solo 1023 números, insuficientes para la cantidad de routers en Data Centers modernos.
  En topologías de tipo Clos, como lo es Fat-Tree, se debe tener especial consideración en la numeración AS. Si un nodo, por ejemplo un Leaf, tiene alcance a un ToF por dos posibles caminos, que son Spines del PoD, y ambos tienen distinto número AS, se puede dar una situación de conteo infinito si se pierde la conectividad con el ToF. Para evitar este problema que también se puede dar a nivel de Spines con ToF, se numeran los nodos de la siguiente forma:
  - Cada Leaf tiene un AS único.
  - Los Spines de un mismo PoD comparten el mismo número de AS,

que es distinto entre PoDs.
- Todos los ToF comparten un mismo ASN.

- **Algoritmo de decisión:** BGP elige el mejor camino a partir de un proceso de decisión ordenado que tiene en cuenta un conjunto de métricas. En el caso de BGP en el Data Center se tiene en cuenta como única métrica el AS-PATH.
- **Selección de múltiples caminos:** El algoritmo de decisión mencionado anteriormente toma dos AS-PATH como iguales solo si todos los ASN que contiene son iguales. Para lograr tener un algoritmo Multipath, se cambia el algoritmo para que tome el largo del AS-PATH en lugar del camino exacto.
- **Modificación en los tiempos de respuesta:** La comunicación entre pares BGP está controlada por temporizadores que por defecto están configurados para priorizar la estabilidad en la red sobre la rápida convergencia. En el caso de BGP en el Data Center, se prioriza la rápida convergencia por lo cual se modifican los temporizadores de publicación o *Advertisment*, conexión o *Connect*, *Keep Alive* y *Hold timers*.

2.3.1.2. OpenFabric

OpenFabric es un protocolo desarrollado por LinkedIn y descripto en el draft de la IETF [White y Zandi, 2018], el cual es apropiado para enrutamiento en topologías densas, como es el caso de las redes Fat Tree.

Es una adaptación del protocolo de estado de enlace IS-IS[1] [Ginsberg y col., 2016] el cual está diseñado para proveer una vista completa de la topología desde un punto de la red para simplificar las operaciones, minimizar la configuración de los dispositivos y optimizar las operaciones de IS-IS en una topología Spine-Leaf para tener un protocolo escalable. Se puede consultar sobre el funcionamiento de IS-IS en [Caiazzi, 2019], se introducen aquí las características de OpenFabric para enrutamiento en el Data Center.

OpenFabric fue diseñado para enrutamiento en topologías que generalmente son predecibles, lo cual permite simplificar algunos aspectos que IS-IS debe generalizar, quitando algunos elementos.

[1] Intermediate System to Intermediate System.

- **Múltiples dominios de flooding:** OpenFabric no funciona correctamente en topologías que tienen múltiples dominios de Flooding.
- **Enlaces de acceso múltiple:** Las topologías de tipo Clos generalmente soportan solo conexiones punto a punto, por lo cual solo este tipo de conexión es requerido en OpenFabric.
- **Métricas externas:** No hay necesidad de métricas externas al contrario de IS-IS, se asume que todas las métricas van a ser correctamente configuradas en cualquier punto de redistribución de rutas.
- **Ingeniería de tráfico:** Está diseñado para proveer información de la topología y alcanzabilidad dentro de la misma. OpenFabric no está diseñado para aplicar ingeniería de trafico, preferencias de rutas por etiquetas u otros mecanismos que utilicen políticas.

Por otro lado OpenFabric agrega algunas modificaciones para construir un protocolo de estado de enlace escalable.

- Solo soporta adyacencias punto a punto de nivel dos, esto evita que se creen distintos dominios de flooding, dado que la optimización de flooding de OpenFabric necesita una vista completa de la topología. Además se modifica el proceso de formación de adyacencias, generándolas en un orden en lugar de a medida que se descubren los vecinos y al mismo tiempo.
- Agrega un mecanismo para determinar el nivel de un nodo en una topología Fat-Tree.
- Agrega un mecanismo que reduce el flooding mientras asegura que los sistemas tengan sus bases de datos sincronizadas en la topología.

El protocolo requiere a su vez soporte para identificar a los routers con un nombre de host [Shen y McPherson, 2008] y soporte de Purge Originator Identification [Qin y col., 2011], para identificar el nodo causante de una purga en la topología. OpenFabric prohíbe que se utilice en conjunto con implementaciones estándar de IS-IS en ambientes productivos, se deben tratar como dos protocolos separados.

2.3.1.3. RIFT

Routing in Fat Trees es un protocolo de enrutamiento para topologías Fat-Tree actualmente en desarrollo por el grupo de IETF RIFT Working Group

y descripto en el draft [Przygienda y col., 2020], el cual está en proceso de estandarización. Busca optimizar la complejidad y minimizar la configuración en topologías del tipo Clos o Fat-Tree.

- Automatiza la configuración, construcción de la topología y detección de enlaces mediante el mecanismo Zero Touch Provisioning (ZTP).
- Minimiza la información de enrutamiento enviada en cada nivel.
- Balancea las cargas entre un conjunto de caminos, teniendo en cuenta el ancho de banda disponible.
- Previene falla de los enlaces o nodos y automatiza su detección, así como los cambios necesarios en el enrutamiento para evitar *blackholing* o enrutamiento subóptimo.

En particular funciona como un protocolo híbrido según el sentido de la mensajería. Por un lado, envía la información en sentido Sur-Norte como un protocolo de estado de enlace para que los nodos superiores tengan una visión completa de la topología. Mientras que actúa como un protocolo de vector de distancias en la dirección Norte-Sur enviando solo la información necesaria y procesada a los nodos de niveles inferiores. Los routers intercambian principalmente dos tipos de mensajes:

- **LIE (Link Information Elements):** son equivalentes a los mensajes *hello* de protocolos como OSPF e IS-IS y son utilizados para descubrir a los vecinos y formar adyacencias. A su vez se incluyen dentro del mecanismo ZTP para aprovisionar y detectar enlaces.
- **TIE (Topology Information Element):** transmiten la información de la topología y son equivalentes a los LSA en OSPF o LSP en IS-IS. Los mensajes TIE siempre tienen una dirección. Los N-TIE (north) contienen una descripción de la topología por estado de enlace de los niveles más bajos, mientras que los S-TIE (south) contienen solo las rutas por defecto que envía el nivel anterior.

A su vez se cuenta con los mensajes TIDE[1] y TIRE[2] que son equivalentes a los mensajes CSNP y PSNP de el protocolo IS-IS respectivamente.

[1] Topology Information Description Element.
[2] Topology Information Request Element.

2.3.1.3.1. Problema del Fallen Leaf El comportamiento dual que presenta RIFT según la orientación presenta algunos inconvenientes. Se puede tener una pérdida de conectividad total en una fábrica sin particionar entre un ToF y un Leaf, el cual se puede solucionar utilizando lo que se conoce como *desagregación positiva*. En fábricas grandes o con switches de bajo radix, un ToF puede terminar siendo particionado en planos, lo que ocasiona que un Leaf sea alcanzable solo desde un subconjunto de los ToF.

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

Este problema se puede observar en la Figura 2.6, donde se exhibe una topología Multi-Plane, con $k = 4$. Debido a la consecuente caída del único enlace que conecta a los Leaf del primer PoD con el segundo ToF, se terminan obteniendo como resultado dos Fallen Leaf.

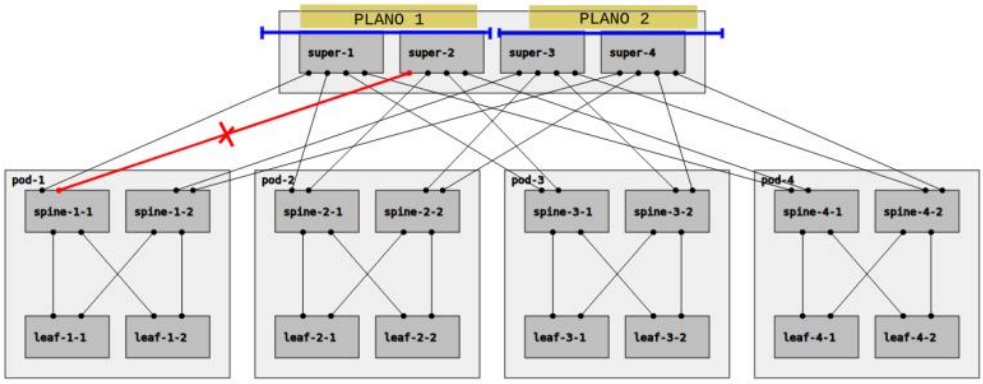


Figura 2.6: Topología Multi-Plane con caída de un enlace entre spine-1-1 y super-2. Se generan los Fallen Leaf leaf-1-1 y leaf-1-2 [“Routing in fat trees”, 2019]

Por otro lado, para poder lidiar con el problema de los Fallen Leaf en topologías Multi-Plane, es que RIFT necesita que todos los ToF compartan la misma base de datos de la topología norte. Esto ocurre de manera natural cuando hay un único plano gracias al flooding hacia el norte y reflexión hacia el sur, pero necesita consideraciones adicionales en las fábricas Multi-Plane. Para esto RIFT utiliza en estos casos la interconexión en forma de anillo entre switches en distintos planos. Otro tipo de soluciones terminan necesitando más

cableado, o resultan en caminos de flooding mucho más largos y/o puntos de fallo únicos.

Reservándose dos puertos en cada ToF, es posible conectarlos para poder formar anillos bidireccionales entre planos. Estos anillos se usan para intercambiar información completa de la topología norte entre planos. Que todos los ToF compartan la topología norte permite que mediante la *desagregación negativa*, también denominada *transitiva*, solucionar cualquier caso posible de Fallen Leaf, además de cumplir el requisito de tener visión total de la fábrica en el nivel de los ToF.

2.3.1.3.2. Desagregación positiva, no transitiva En condiciones normales, los S-TIE contienen únicamente las adyacencias y una ruta por defecto. Sin embargo, si un nodo detecta que su prefijo IP por defecto cubre uno o más prefijos que son alcanzables a través de sí mismo pero no por uno o más nodos en su mismo nivel, entonces debe explícitamente anunciar esos prefijos en un S-TIE. De lo contrario, una fracción del tráfico hacia el norte para esos prefijos sería enviado a nodos sin la alcanzabilidad correspondiente, causando que exista *black-holing*[1]. Incluso cuando no ocurre esto, el reenvío podría causar que los paquetes vayan siendo enviados entre los Spine de niveles superiores, lo que genera una condición no deseable para la fábrica.

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.

2.3.1.3.3. Desagregación negativa, transitiva Este mecanismo se plantea como solución a escenarios que no se pueden resolver únicamente mediante desagregación positiva, como fallas en el nivel superior de la fábrica, o múltiples fallas de enlaces en un diseño single plane, donde se genera el problema Fallen Leaf.

[1]Paquetes que terminan siendo siendo descartados por un nodo que no puede alcanzar el destino que tienen.

La desagregación negativa sí es transitiva, y además se propaga hacia el sur cuando todas las rutas posibles han sido anunciadas como excepciones negativas. Un anuncio negativo de una ruta solo puede llevarse a cabo cuando el prefijo negativo es agregado por un anuncio positivo de una ruta para un prefijo más corto. En dicho caso, el anuncio negativo genera que el prefijo negativo sea alcanzable pero considerando excluir algunos de los vecinos como next-hop anunciados por el mismo. Cuando el ToF no está particionado, el flooding colectivo hacia el sur de la desagregación positiva por los ToF que aún pueden alcanzar el prefijo afectado es suficiente para cubrir todos los switches del próximo nivel al sur, típicamente los nodos ToP. Si todos esos switches son conscientes de la desagregación, colectivamente crean una "barrera" que intercepta todo el tráfico hacia el norte y lo reenvían a los ToF que anuncian la ruta más específica. En ese caso, la desagregación positiva es suficiente para resolver el problema Fallen Leaf.

Sin embargo, cuando la fábrica está particionada en planos, la desagregación positiva de los ToF en planos distintos no alcanza los switches ToP del plano afectado y no se puede resolver el problema. Esto es, una falla en un plano solo puede resolverse dentro del mismo plano. Además, la selección del plano para un paquete se realiza a nivel del Leaf, y la desagregación debe entonces ser transitiva y alcanzar todos los Leaf. En ese caso, se necesita la desagregación negativa.

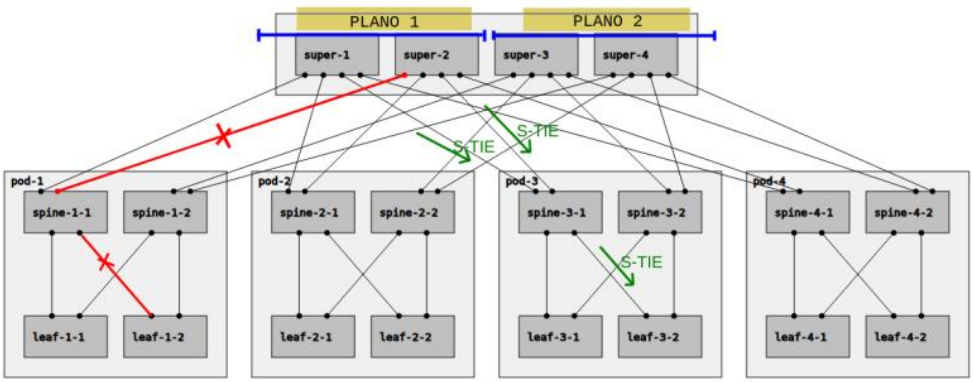


Figura 2.7: Mecanismo de desagregación negativa. Se logra evitar que el leaf-3-2 envíe tráfico al Fallen Leaf leaf-1-2 a través del plano 1, por el cual este es inalcanzable ["Routing in fat trees", 2019].

2.3.1.3.4. Flooding reduction La distribución de TIEs hacia el norte se realiza mediante flooding, aunque con variaciones que buscan aprovechar la topología Fat-Tree, pero basados principalmente en los protocolos de estado de enlace más actuales. RIFT proporciona una solución de "flooding reduction" para paliar la falta de escalabilidad del flooding clásico en topologías con un alto grado de conectividad, además de un balanceo de carga global que se acerca al óptimo. Utiliza una técnica de control de flooding hacia el norte, similar al concepto de *Multipoint Relay* (MPR) que se introduce en la definición del siguiente protocolo [Clausen y col., 2003]. Se puede consultar [Przygienda y col., 2020], donde se presenta la especificación del algoritmo utilizado.

2.3.1.4. LSVR

Link State Vector Routing es un protocolo actualmente en desarrollo por la IETF [Patel y col., 2020] que combina mecanismos de enrutamiento de estado de enlace y vector de distancias. Para ello utiliza el formato de los paquetes BGP-4 y el manejo de errores, distribuyendo la información de enrutamiento en un vector de estado de enlace (LSV). El mismo se especifica como una estructura de datos compuesta por atributos de enlaces, información de vecinos y otros atributos.

Si bien se utilizan mecanismos y mensajes de BGP, el protocolo está pensado como una solución independiente, que define los vectores LSV y la selección de ruta utilizando el algoritmo basado en Dijkstra SPF (Shortest Path First). El protocolo permite ECMP y pretende ser simple en su operación además de escalable.

Si bien no se conoce una implementación pública del protocolo, el mismo tiene apoyo de Arrcus inc. y existe un draft en el cual se está trabajando.

2.3.2. Enrutamiento centralizado

Las redes definidas por software o SDN plantean una separación del plano de control o enrutamiento con el plano de datos o forwarding, dando a su vez un control centralizado de los dispositivos de red. El algoritmo de enrutamiento se ejecuta mediante aplicaciones en un controlador SDN que instala luego las reglas de forwarding en los dispositivos.

En la figura 2.8 se presenta su arquitectura, la cual se compone de tres capas que se comunican entre ellas mediante APIs. La capa superior contiene las aplicaciones que van a comunicarse con la capa de control para obtener información y ejecutar acciones sobre los servicios de red disponibles. En la capa de control se encuentra el software controlador propiamente dicho, el cual va a proveer servicios a las aplicaciones y se va a comunicar con los dispositivos de red para aplicar las reglas de forwarding y obtener información de la red. Por último la capa de infraestructura contiene los dispositivos de red, los cuales pueden ser tanto virtuales como físicos.

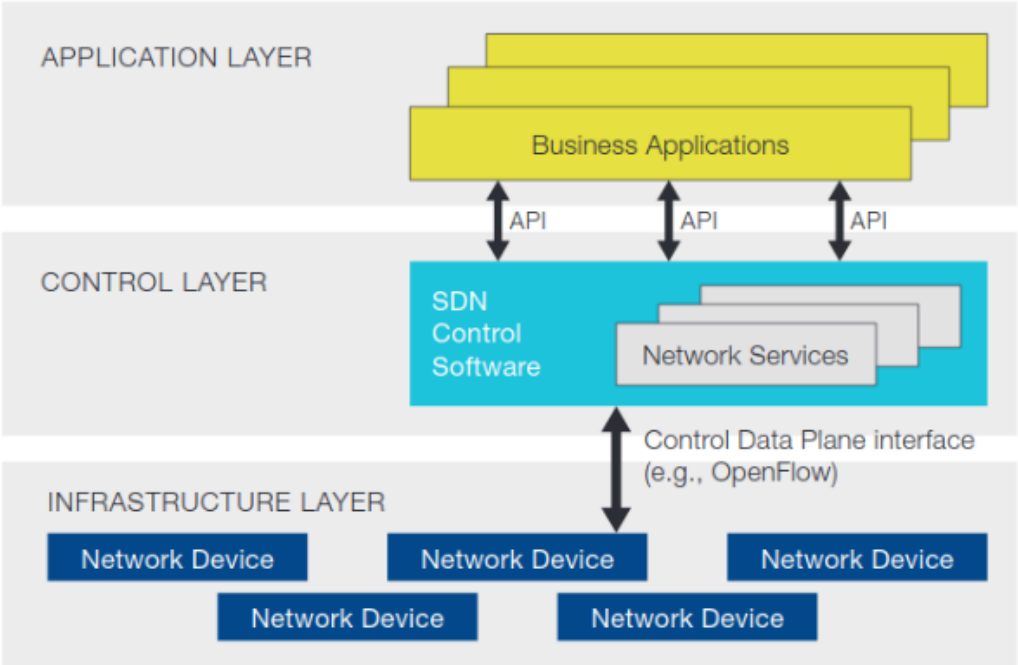


Figura 2.8: Arquitectura SDN [Truong y col., 2015]

Se introducirá el concepto de enrutamiento basado en el origen así como la solución Segment Routing, los cuales no están directamente conectados al enrutamiento centralizado pero son la base de soluciones como Trellis. Introduciremos más adelante esta solución, en conjunto con otras, que se basa en el controlador ONOS[1], utilizando OpenFlow para comunicación con la capa de infraestructura.

[1]Open Network Operating System.

OpenFlow es un conjunto de protocolos y API utilizados en SDN para separar el plano de control del plano de forwarding, se puede consultar sobre su funcionamiento en [Quiroz Martiña, 2015] y ["SDN para enrutamiento en el Datacenter", 2020].

2.3.2.1. Segment Routing

Segment Routing es un método de enrutamiento 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) [Filsfils y col., 2018].

El enrutamiento 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. 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.

La arquitectura de Segment Routing tiene dos componentes principales: el plano de control y el plano de datos. El plano de datos define cómo se codifican los segmentos en un paquete definiendo un cabezal Segment Routing Header (SRH). El plano de control define cómo se identifican los nodos dentro de un mismo dominio y cómo procesan las listas de segmentos [Litmanen, 2017].

- **Plano de forwarding**
  Un SRH contiene una lista ordenada de segmentos, los cuales pueden ser de distintos tipos. Los cuatro tipos principales son:

  - *Node-SID:* Identificador único de un nodo en el dominio de Segment Routing.
  - *Adjacency-SID:* Identifica un puerto de egreso con un nodo adyacente.
  - *Service-SID:* Identifica un servicio de un nodo, por ejemplo inspeccion de seguridad de paquetes.
  - *Anycast-SID:* Identifica un conjunto de nodos bajo un SID común.

Para que un nodo soporte Segment Routing, debe ser capaz de realizar las siguientes acciones en el pano de datos:

- *NEXT:* Mover el puntero SRH al próximo SID activo en la lista de segmentos. El puntero apunta al segmento activo de la lista.
- *PUSH:* Agregar segmentos a la lista. El segmento se agrega al inicio de la lista y este se convierte en el segmento activo.
- *CONTINUE:* Se reenvía al paquete al próximo destino sin realizar cambios en la lista o el puntero SRH.

Segment Routing tiene soporte para IPv4 utilizando MPLS sin realizar cambios en su arquitectura. En la figura 2.9 se puede ver una asociación entre los conceptos de MPLS y Segment Routing.

Se puede aplicar también a IPv6 donde los segmentos son direcciones IPv6 y se debe agregar a los paquetes un cabezal de enrutamiento, conteniendo la lista de direcciones. En la figura 2.10 se muestra cómo se encapsula el paquete IPv6 original a ser enviado dentro de otro paquete IPv6 que contiene la extensión SRH. Por otro lado en la figura 2.11 se puede ver la estructura del paquete IPv6 con la extensión SRH.

SR	**MPLS**
SR Header	Label Stack
Active Segment	Topmost Label
PUSH Operation	Label Push
NEXT Operation	Label POP
CONTINUE Operation	Label Swap

Figura 2.9: Mapeo de componentes de MPLS en Segment Routing [Litmanen, 2017]

- **Plano de control**
  Se define cómo se publican los SID dentro de la red y a su vez cómo un nodo de borde en el dominio de Segment Routing debe elegir un camino para un paquete, para lo cual existen tres posibilidades.

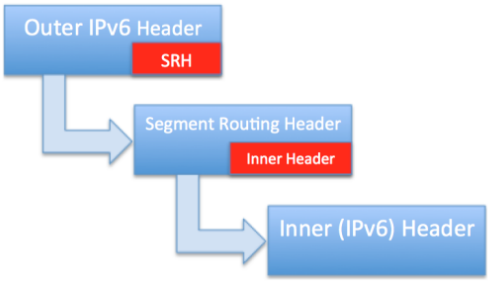


Figura 2.10: Encapsulamiento de IPv6 para Segment Routing [Litmanen, 2017]

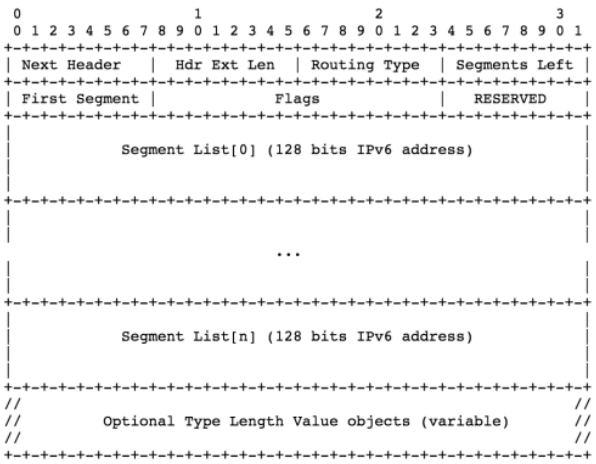


Figura 2.11: Formato del paquete IPv6 para Segment Routing [Litmanen, 2017]

- *Configuración estática*: se configuran túneles de forma manual en los nodos. Este método se utiliza para solucionar problemas.
- *Cálculo de ruta distribuido*: el nodo utiliza un protocolo distribuido para calcular el camino más corto hacia un destino y genera el SRH para dicho paquete.
- *Cálculo de ruta centralizado*: los nodos se comunican con un controlador SDN para obtener el camino hacia un destino desconocido. El controlador que tiene una visión global de la topología calcula el camino y lo envía al nodo, que genera el SRH para el paquete.

Tanto en el caso distribuido como el caso centralizado, el algoritmo de cálculo de rutas por defecto es SPF para ECMP, lo cual es interesante en este contexto dado que habilita la posibilidad de balanceo de carga y redundancia. Además en el caso centralizado se pueden agregar políticas que sobrescriben su comportamiento, permitiendo una mayor flexibilidad en la aplicación de técnicas de ingeniería de tráfico.

2.3.2.2. Implementaciones de Segment Routing en SDN

Existe una variedad de controladores SDN, desde ONOS ["ONOS Open Network Operating System", 2021], OpenDaylight ["OpenDayLight", 2018], Ryu ["Build SDN Agilely", 2017] a controladores más simples como POX de Mininet ["Using the POX SDN controller", 2017]. Cada controlador utiliza para comunicarse con los switches protocolos abiertos como OpenFlow ["OpenFlow - Open Networking Foundation", 2017], siendo el protocolo de facto, aunque pueden llegar a utilizar protocolos propietarios. Sin embargo, las aplicaciones que los desarrollan suelen ser propietarias y se pueden encontrar implementaciones distintas para resolver el mismo problema. Se describen a continuación algunas de las soluciones de Segment Routing implementadas en distintos controladores.

- **ONOS:** La implementación de SR en ONOS parte del proyecto CORD ["CORD: Leaf-Spine Fabric with Segment Routing", 2016], presentado por la ONF[1] y es una solución puramente SDN, que tiene soporte tanto para SR sobre MPLS como sobre IPv6. El proyecto evolucionó a la solución Trellis, una suite de aplicaciones SDN en el Data Center que implementa una topología Clos de dos niveles o *spine-leaf* sobre *white switches* ["Trellis", 2021].
  Trellis interconecta el Data Center a otras redes, como redes WAN o 5G. A su vez conecta el Data Center a redes de acceso y dentro del Data Center utiliza Segment Routing. Como se puede ver en la figura 2.12, no se trata de una topología de Data Center convencional, sino que Trellis se encuentra en el borde de la red interconectando los distintos componentes. Particularmente esta solución utiliza OpenFlow para comunicarse con la capa de infraestructura.

- **OpenDayLight:** Existe una solución para OpenDaylight que utiliza mensajes BGP Link-State para enviar información de estado de enlace entre los routers y el controlador, siendo una solución híbrida entre centralizada y distribuida ["OpenDaylight Pathman SR App", 2018].

- **Ryu:** Existe una versión en desarrollo de Segment Routing sobre IPv6 ["srv6-sdn: An SDN ecosystem for SRv6 on Linux", 2020].

[1] Open Network Foundation.

Protocolo	Tipo	Descubrimiento de topología
RIFT	Distribuido	Híbrido (Link-State al Norte, Distance-Vector al Sur)
BGP	Distribuido	Distance-Vector o caminos
OpenFabric	Distribuido	Link-State
LSVR	Distribuido	Híbrido
SDN Segment Routing	Centralizado	OpenFlow entre cada nodo y el controlador

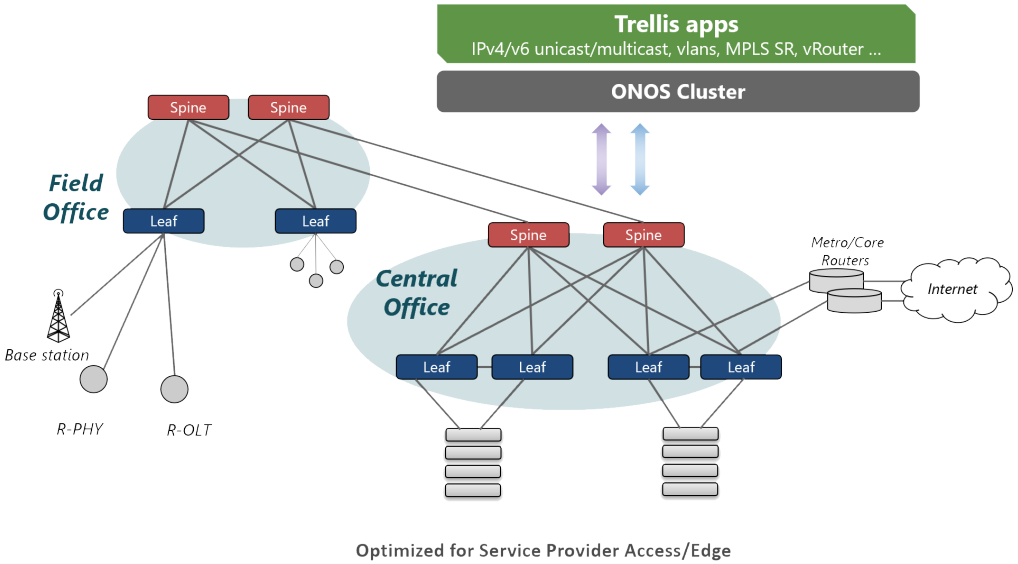


Figura 2.12: Despliegue de Trellis Fabric [“Trellis - Production-ready Multi-purpose leaf-spine fabric”, 2021]

2.3.3. Comparación de soluciones

En la tabla 2.1 se puede ver una comparación de las soluciones estudiadas con mayor detalle, una diferenciación según el tipo de enrutamiento (centralizado o distribuido), las herramientas o algoritmos que utilizan para el descubrimiento de la topología, implementaciones existentes y tipo de configuración.

Se presentaron en esta sección algunos de los protocolos estándar o que se encuentran en proceso de estandarización, sin embargo algunas de las grandes empresas Cloud utilizan sus propios protocolos. Se presentan brevemente algunos de ellos a continuación.

- **Open/R**: Desarrollado y utilizado por Facebook, es un protocolo distribuido de estado de enlace que se basa en la utilización de hardware moderno para superar los límites de escalabilidad en grandes topologías.

Implementaciones	Configuración
RIFT-Python JunOS RIFT	Automática: ZTP (Zero Touch Provisioning). Configuración minima ToF.
FRR bgpd Implementaciones comerciales de BGP	Manual o por automatizaciones externas al protocolo.
FRR Fabricd	Automáticamente se determina nivel de un nodo. Mecanismo de formación de adyacencias.
En proceso por Arrcus Inc	Manual o por automatizaciones externas al protocolo.
ONOS Segment Routing Opendaylight Pathman SR App Ryu srv6-sdn	Configuración centralizada desde el controlador y por API REST.

Tabla 2.1: Comparación de Soluciones de Enrutamiento

El protocolo soporta distintas topologías (como redes WAN, Data Center, y redes Wireless) y distintos sistemas (Arista, Juniper JunOS, Linux Routing, entre otros), además puede integrarse con controladores SDN. Por más información consultar en ["Open/R: Open routing for modern networks", 2017].

- **Firepath**: Google utiliza un protocolo centralizado el cual tiene un conjunto de controladores centrales o Firepath masters, y un conjunto de switches que ejecutan un agente o Firepath agent. Cada switch conoce a sus vecinos mediante sus enlaces e intercambia la información con el Firepath master, que computa la Link State Database y envía dicha información a los mismos. Además, los routers en el borde implementan BGP y redistribuyen las rutas aprendidas por Firepath, permitiendo utilizar servicios externos o conectarse con otros Data Center. En [Singh y col., 2016] se tiene información sobre su evolución, así como su funcionamiento.
- **Brainslug**: Microsoft utiliza una solución híbrida entre BGP para el Data Center y un controlador SDN. La idea es utilizar las ventajas de SDN para modificar el forwarding y aplicar ingeniería de trafico cuando sea necesario, pero con una preferencia por las rutas generadas por BGP. Se presenta su funcionamiento en ["Brain-Slug: a BGP-Only SDN for Large-Scale Data-Centers", 2013].

2.4. Forwarding en el Data Center

Una vez los algoritmos de enrutamiento calculan los distintos caminos entre los servidores, es fundamental que se utilicen plenamente los recursos de comunicación. De esto se encarga el plano de forwarding o datos, que realiza

el reenvío de mensajes según la tabla de enrutamiento.

Introduciremos algunos conceptos importantes para entender cómo funciona el forwarding en el Data Center y cómo se relaciona con los algoritmos de enrutamiento. Si bien no desarrollaremos estos conceptos, los mismos son una parte fundamental para cumplir los requerimientos de enrutamiento en el Data Center descriptos en la sección 2.1.

2.4.1. Equal-Cost Multipath

Distintos protocolos de enrutamiento, como pueden ser OSPF o ISIS, permiten explícitamente el enrutamiento *Equal-Cost Multipath* (ECMP)[Hopps, 2000; Thaler y Hopps, 2000]. Algunas implementaciones incluso permiten el uso de ECMP en conjunto con RIP y otros protocolos de enrutamiento. Al usar ECMP, lo que se hace en particular es balancear la carga de tráfico (paquetes) entre múltiples posibles rutas a un destino que tengan el mismo costo. Esto quiere decir que dichas rutas pueden ser descubiertas y utilizadas con algún criterio para repartir el tráfico entre las mismas.

El enrutamiento entre múltiples rutas en un dispositivo lo que genera es que el mismo disponga de varios next-hops para un destino dado, y deba utilizar algún método para seleccionar cuál de ellos utilizará para enviar el próximo paquete. En cuanto a estos mecanismos de selección, se pueden diferenciar principalmente en dos tipos: balanceo **per-packet** y **per-flow**.

2.4.1.1. Balanceo de carga per-packet

El balanceo de carga per-packet significa que el router envía un paquete para cierto destino sobre un camino, luego otro paquete para el mismo destino sobre el segundo camino, y así sucesivamente. Lo que garantiza este método es una distribución completamente equitativa del tráfico a través de los enlaces. Podemos observar un ejemplo gráfico de balanceo per-packet en la Figura 2.13.

Algunos de los inconvenientes que posee este mecanismo son:

- **MTU**: Debido a que muchos de los caminos pueden tener un MTU[1] diferente, esto significa que el MTU total puede cambiar entre paquete

[1]Maximum Tranmission Unit.

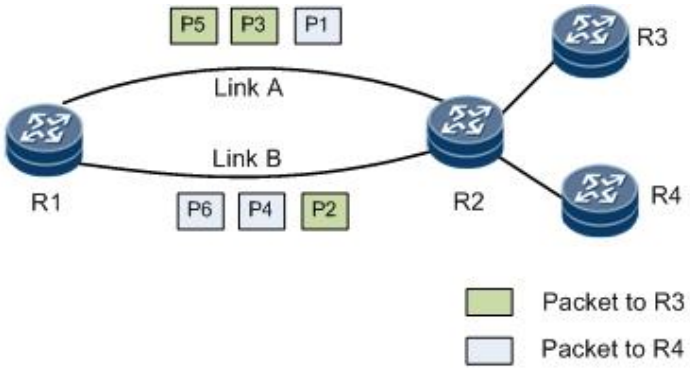


Figura 2.13: El router R1 distribuye sus paquetes hacia dos destinos diferentes con el mismo next-hop R2 entre los dos enlaces disponibles que tiene ["Per-Flow and Per-Packet Load Balancing", 2021]

y paquete, desaprovechando la función del *path MTU discovery*.

- **Latencia**: Debido a que los distintos caminos pueden tener una latencia diferente, hacer que los paquetes tomen caminos distintos puede causar que los paquetes lleguen en distinto orden (al de partida), aumentando la latencia de entrega y los requerimientos de *buffering*. Por ejemplo, el reordenamiento de paquetes genera que en TCP se crea que se hayan perdido paquetes al arribar primero paquetes con un número de secuencia mayor que los posteriores. Cuando tres o más paquetes se reciben antes que un paquete "tardío", TCP entra en el modo "fast-retransmit" [Blanton y col., 2009] que consume un ancho de banda extra, al intentar que se retransmitan los paquetes retrasados innecesariamente. Esto además puede causar potencialmente más pérdida de paquetes, afectando el rendimiento, y siendo en conclusión perjudicial para la performance de la red.
- **Debugging**: Muchas utilidades comunes como *ping* y *traceroute* son mucho menos confiables al existir distintos caminos e incluso pueden presentar resultados incorrectos.

2.4.1.2. Balanceo de carga per-flow

En general, se define un *flow* (flujo) para representar la granularidad que utiliza un router para mantener el estado de las distintas clases de tráfico. La definición exacta de un flujo puede depender de la implementación a la que se esté referenciando. Por ejemplo, un flujo puede identificarse únicamente por la dirección de destino, o puede definirse por la tupla

$< origen,\ destino,\ ID\ de\ protocolo >$.

El método per-flow introduce una variación con respecto al método per-packet, donde el balanceo de carga clasifica paquetes dentro de estos distintos flujos, basado en alguna regla configurada previamente. Los paquetes que pertenecen al mismo flujo se envían sobre el mismo enlace al next-hop. En la Figura 2.14 se puede observar un ejemplo de balanceo per-flow.

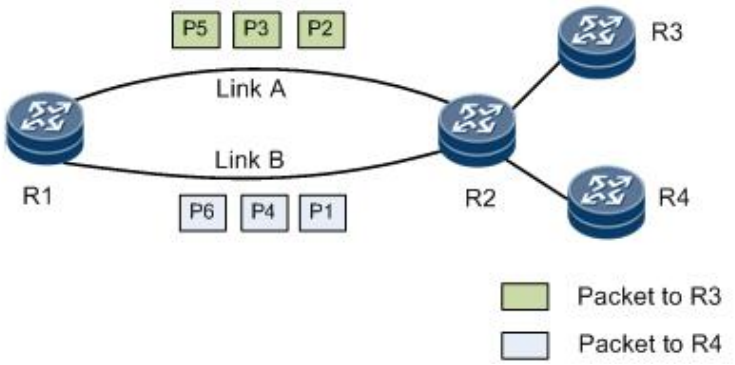


Figura 2.14: Los paquetes que van desde R1 a R3 y R4 pertenecen a dos flujos distintos. R1 envía los paquetes a R2 por distintos enlaces, según el flujo al que pertenezcan [“Per-Flow and Per-Packet Load Balancing”, 2021].

Para definir los distintos enlaces que se van a utilizar para cada flujo de tráfico, el balanceo per-flow utiliza una estrategia un poco más eficiente: mediante un algoritmo de hash que puede combinar distintas variables (direcciones origen y destino, puertos, etc) determina las distintas interfaces de salida. El resultado de este tipo de combinaciones debe reflejar en la varianza probabilísitca el balanceo de carga. O sea, mientras más cambien los valores relevantes para el algoritmo de cada paquete dentro de un mismo flujo de tráfico, más se aproxima el resultado final a un balanceo más equitativo. A pesar de que la distribución no es tan equitativa como el balanceo per-packet, soluciona muchos de los problemas que el último conlleva.

2.5. Gestión del Data Center

Las soluciones vistas hasta el momento buscan resolver la conectividad dentro del Data Center, particularmente las decisiones de routing y forwarding. Pero en un Data Center que tiene miles de routers o switches y escala a cientos

de miles de servidores, la comunicación es solo una de sus componentes.

Los Data Centers necesitan ser gestionados de forma eficiente y automatizada. Particularmente los proveedores de servicios de nube o CSP[1] necesitan la flexibilidad y el control suficientes para que un cliente pueda desplegar máquinas virtuales a demanda en cuestión de minutos. Por dichos motivos se gestionan de forma centralizada, agilizando y haciendo más sencillo el trabajo de los operadores.

Es importante entender cómo se involucra la gestión de la infraestructura del Data Center con las distintas soluciones de enrutamiento tanto distribuido como centralizado. Para ello se introducirá la arquitectura y conceptos de infraestructuras Cloud, además de mencionarse algunas soluciones existentes.

2.5.0.1. Infraestructura Cloud

En la figura 2.15 se pueden observar los componentes, los cuales se detallan a continuación, de una infraestructura de Data Center. Los mismos se comunican entre ellos mediante APIs. Se puede consultar en [Son y Buyya, 2018b] por más información.

- **Cloud Manager:** gestiona los recursos, recibe solicitudes para crear máquinas virtuales, controla que el uso de energía de los recursos sea eficiente y monitorea a los mismos. Es el punto de comunicación entre la infraestructura y los operadores por lo cual provee también una interfaz de usuario.
- **SDN Controller:** las funciones relacionadas con la red son manejadas por la controladora SDN, la cual se conecta al Cloud Manager por intermedio de APIs en sentido norte (es decir que provee servicios al Cloud Manager). Se encarga de funcionalidades como descubrimiento de topología de red, monitoreo, configuraciones dinámicas y gestión de redes virtuales. Algunos ejemplos de controladora son OpenDaylight y ONOS. Cabe destacar que el componente de enrutamiento podría ser una solución distribuida y procesada por los routers en lugar de una solución centralizada. En el segundo caso sería necesaria una comunicación entre

[1] Cloud Service Providers.

los mismos y el Cloud Manager para el monitoreo y aprovisionamiento de red a los servidores.

- **Cloud Resources:** los recursos de cómputo que son aprovisionados por el Cloud Manager para correr máquinas virtuales sobre un hipervisor y los recursos de red manejados por el controlador SDN que se comunica con los switches mediante APIs como por ejemplo OpenFlow.

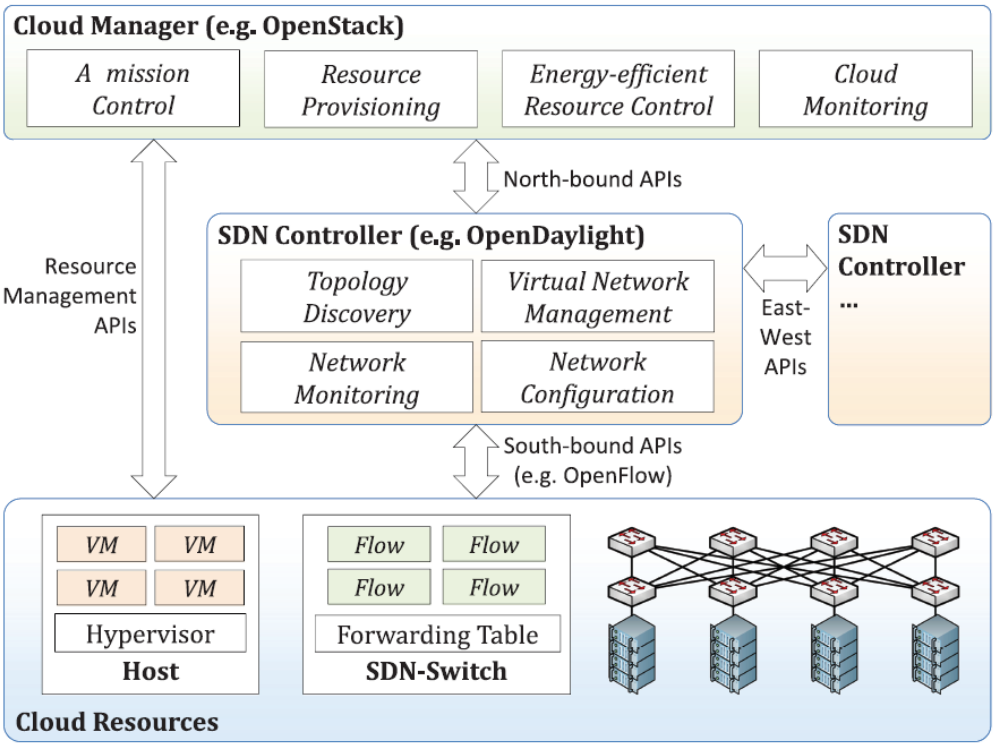


Figura 2.15: Arquitectura de Cloud Computing con SDN [Son y Buyya, 2018b]

2.5.0.2. Soluciones existentes

En el transcurso del proyecto, por decisiones de definición del alcance, no se hicieron pruebas sobre plataformas de gestión, aunque se describe a continuación algunas de las soluciones actualmente existentes para gestión en el Cloud que se pudieron encontrar:

- **Openstack**: Openstack es un Cloud Manager de código abierto que permite gestionar nubes privadas y públicas. Su diseño es modular y se basa

en un conjunto de herramientas que llama proyectos, los cuales se encargan de las distintas funcionalidades necesarias para la gestión de cómputo, almacenamiento y red. Por más información, consultar ["Openstack", 2021].

- **VMWare vCloud**: VMware vCloud Director es la plataforma estrella de gestión de la cloud de VMware para los proveedores. Las características principales que presenta la plataforma son creación de depósitos de recursos compartidos, gestión multisitio, contenedores como servicio, y flujo de trabajos automatizado, entre otras. Se puede consultar ["VMWare Cloud Director", 2021] para más información.

Capítulo 3

Modelado teórico y análisis

Se realizó un análisis teórico de los protocolos RIFT y BGP de forma de evaluar y comparar su complejidad. Para ello, se toma como base la descripción del protocolo RIFT propuesta en [Przygienda y col., 2020] así como en la documentación de BGP-4 [Rekhter y col., 2006] y BGP para Data Centers [Lapukhov y col., 2016]. A su vez se utilizaron los principios teóricos del análisis de algoritmos distribuidos introducidos en [Santoro, 2006].

Se decidió analizar el protocolo RIFT debido a que presenta la particularidad de comportarse como un algoritmo de flooding en una dirección y como un vector de distancias en el sentido opuesto. Este comportamiento resultó de gran interés al no tener un conocimiento previo de la complejidad. A su vez el protocolo está en proceso de estandarización, por lo cual un estudio de su convergencia aporta información sobre el funcionamiento del mismo. Por otro lado, se analizó BGP, con las consideraciones de su aplicación en Data Centers, de forma de poder comparar RIFT con un protocolo ampliamente utilizado, siendo el último un algoritmo del tipo *path vector* o vector de caminos.

Al momento de analizar teóricamente los algoritmos es de interés responder a ciertas incógnitas que se presentan: ¿Qué atributos de complejidad se desean evaluar? ¿Qué casos se desean estudiar? ¿Cuándo comienza el algoritmo? ¿El algoritmo finaliza en algún momento? ¿Cómo podemos determinar si convergió?

Es de interés evaluar el inicio o *bootstrap* de ambos protocolos, desde que

los nodos inician y comienzan a conocer la topología hasta que el algoritmo converge. Cuando se analiza un algoritmo interesa evaluar tanto la complejidad de mensajería como algorítmica (o de tiempo). Sin embargo un punto particular de los algoritmos de enrutamiento es que no finalizan, si bien convergen en cierto momento. Además, se quieren realizar experimentos emulados para comparar con los resultados obtenidos, pero las medidas de tiempo no son precisas en estos casos, por lo que este análisis se centrará en la complejidad de mensajería.

Para simplificar el análisis se considera que los enlaces que comunican a los nodos son fiables y no hay pérdidas de información, buscando tener un resultado del mejor caso posible como cota inferior.
Debido a la complejidad tanto de los algoritmos como de la topología, es necesario establecer el análisis sobre una topología que se considere básica, no teniendo en cuenta casos de borde. Se deben por lo tanto establecer cuáles son estas restricciones. La topología básica a presentar es un Fat-Tree Multiplane de tres niveles (Leaf, Spine y ToF), que no presenta enlaces entre nodos de un mismo nivel (Este-Oeste) y en el caso de RIFT, todos son Flood Repeaters, en la figura 3.1 se puede ver un ejemplo. Además se asumirá que la cantidad de Leafs es igual a la cantidad de Spines y por lo tanto se tienen $\frac{k}{2}$ Leafs y $\frac{k}{2}$ Spines por Pod.

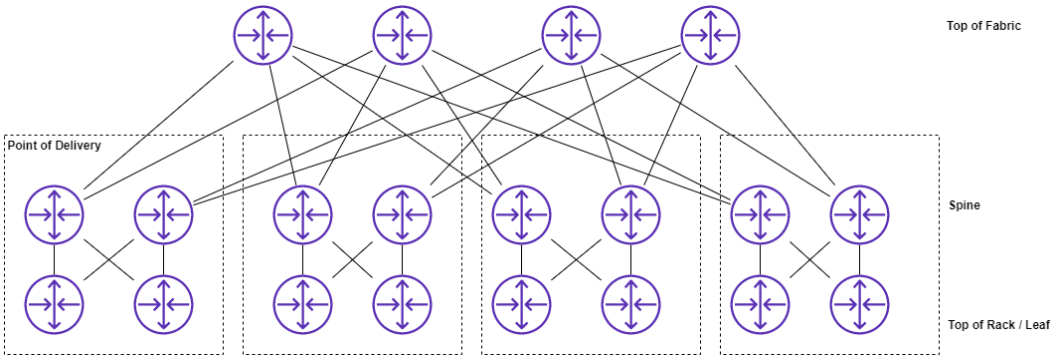


Figura 3.1: Fat Tree Multiplane $k = 4$ y $R = 1$

Por otro lado, se debe especificar una nomenclatura para el análisis y relación entre los distintos valores, de forma de realizar el análisis en función de parámetros de entrada conocidos.

- **k**: Cantidad de puertos de un switch, va a ser un parámetro de entrada. En este caso $k_{top} = k_{leaf} = \frac{k}{2}$

- **R**: Redundancia, va a ser un parámetro de entrada. En el caso single plane es igual a k_{top}.
- **P**: Cantidad de Pods. $P = \frac{k}{R}$
- **N**: Cantidad de planos. $N = \frac{k}{2R}$
- **S**: Cantidad de ToF. $S = \frac{k^2}{4R}$
- **T**: Cantidad de ToF por plano. $T = \frac{S}{N}$
- **Tr**: Cantidad de puertos hacia el sur en ToF. $Tr = P * \frac{k}{2N}$
- **p**: Cantidad de prefijos a anunciar por Leaf. En un principio vamos a asumir $p = 1$.

3.1. Análisis de RIFT

El protocolo presenta dos fases que se analizarán por separado. En la primera fase se envían mensajes LIE para establecimiento de conexiones entre los nodos directamente conectados. Una vez finalizada dicha etapa, se comienzan a enviar mensajes TIE para descubrimiento de la topología. Los mensajes TIDE y TIRE no se tienen en cuenta para el análisis debido a que se envían periódicamente incluso después de la convergencia.

3.1.1. Análisis de mensajes LIE

En la figura 3.2 se puede observar la máquina de estados de RIFT para el descubrimiento de adyacencias. Dicha máquina se ejecuta por cada adyacencia que tengan los nodos.
El protocolo RIFT utiliza un mecanismo de *handshake* en tres fases para la formación de adyacencias que consta de los estados $OneWay$, $TwoWay$, donde las adyacencias están en proceso de formarse, y $ThreeWay$, donde la adyacencia ya se formó completamente, siendo el estado donde comienzan a enviarse mensajes TIE, TIDE y TIRE. Durante el proceso de formación de adyacencias, auto-descubrimiento de vecinos y configuración, el protocolo intercambia mensajes LIE.

Se tiene también el estado $MultipleNeighborsWait$ al cual se llega cuando se encuentran dos vecinos en un mismo enlace o si algún vecino se reconfigura o reinicia sin pasar antes por el estado $OneWay$. Dicho estado no es parte de la formación de adyacencias sino un estado transitorio, luego del cual se debe

volver a iniciar el *handshake*.

En la figura 3.3 se puede ver una versión simplificada de la máquina de estados de los mensajes LIE, en la cual se quitaron todos aquellos estados y eventos que no se van a considerar en el análisis causados por fallas o múltiples vecinos en una misma adyacencia. Debido a que se asumió que la topología comienza configurada en este análisis, no se profundiza en el análisis del Zero Touch Provisioning, se puede consultar sobre su máquina de estados en [Przygienda y col., 2020].

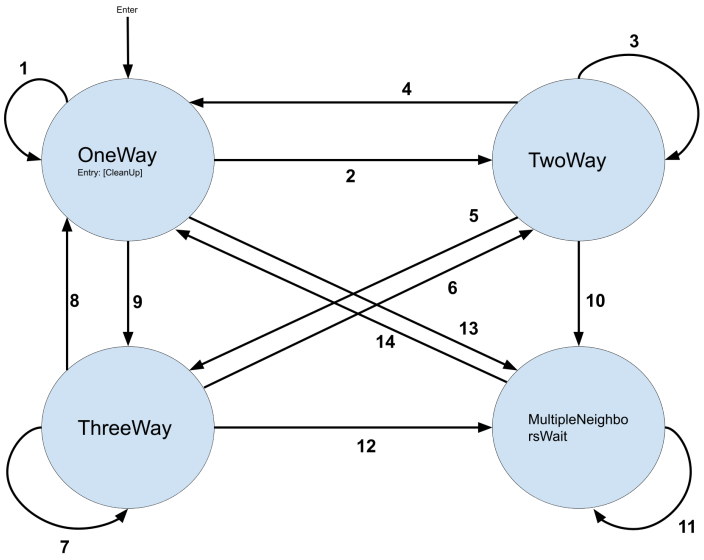


Figura 3.2: Máquina de estados de mensajes LIE

Se especifican a continuación las acciones de cada transición de la máquina de estados de RIFT en la figura 3.2. Los parámetros y funciones se especifican con mayor detalle en [Przygienda y col., 2020].

1. $OneWay \rightarrow OneWay$

   HALChanged [StoreHAL]
   HALSChanged [StoreHALS]
   HATChanged [StoreHAT]
   HoldTimerExpired [-]}
   InstanceNameMismatch [-]
   LevelChanged [UpdateLevel, PUSH SendLIE]
   LIEReceived [PROCESS_LIE]
   MTUMismatch [-]
   NeighborAddressAdded [-]
   NeighborChangedAddress [-]
   NeighborChangedLevel [-]
   NeighborChangedMinorFields [-]
   NeighborDroppedReflection [-]
   PODMismatch [-]
   SendLIE [SEND_LIE]
   TimerTick [PUSH SendLIE]
   UnacceptableHeader
   UpdateZTPOffer [SendOfferToZTPFSM]

2. $OneWay \rightarrow TwoWay$

   NewNeighbor [PUSH SendLIE]

3. $TwoWay \rightarrow TwoWay$

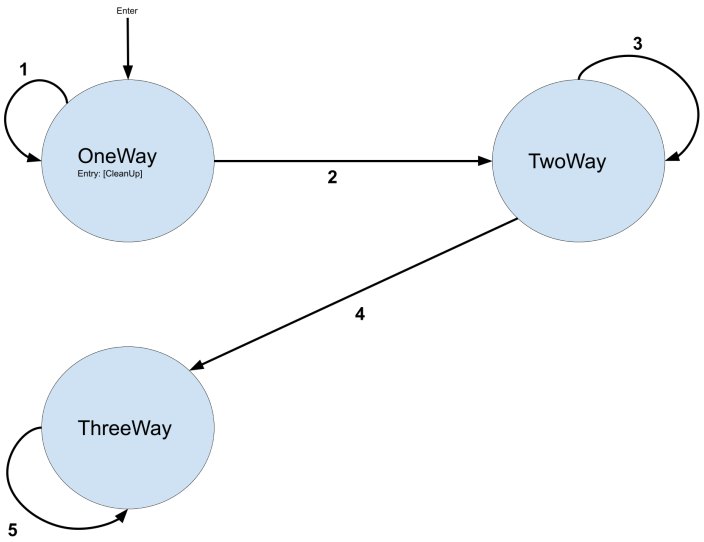


Figura 3.3: Máquina de estados de mensajes LIE simplificada

HALChanged [StoreHAL]
HALSChanged [StoreHALS]
HATChanged [StoreHAT]
LevelChanged [StoreLevel]
LIEReceived [PROCESS_LIE]
SendLIE [SEND_LIE]
TimerTick [PUSH SendLIE,
IF HoldTtimer expired
PUSH HoldTimerExpired]
UpdateZTPOffer [SendOfferToZTPFSM]

4. *TwoWay* → *OneWay*

HoldTimeExpired [−]
InstanceNameMismatch [−]
LevelChanged [StoreLevel]
MTUMismatch [−]
NeighborChangedAddress [−]
NeighborChangedLevel [−]
PODMismatch [−]
UnacceptableHeader [−]

5. *TwoWay* → *ThreeWay*

ValidReflection [−]

6. *ThreeWay* → *TwoWay*

NeighborDroppedReflection [−]

7. *ThreeWay* → *ThreeWay*

HALChanged [StoreHAL]
HALSChanged [StoreHALS]
HATChanged [StoreHAT]
LIEReceived [PROCESS_LIE]
SendLIE [SEND_LIE]
TimerTick [PUSH SendLIE,
IF HoldTimer expired
PUSH HoldTimerExpired]
UpdateZTPOffer [SendOfferToZTPFSM]
ValidReflection [−]

8. *ThreeWay* → *OneWay*

HoldTimerExpired [−]
InstanceNameMismatch [−]
LevelChanged [UpdateLevel]
MTUMismatch [−]
NeighborChangedAddress [−]
NeighborChangedLevel [−]
PODMismatch [−]
UnacceptableHeader [−]

9. *TwoWay* → *ThreeWay*

ValidReflection [−]

10. *TwoWay* → *MultipleNeighborsWait*

NewNeighbor [StartMulNeighTimer]
MultipleNeighbors [StartMulNeighTimer]

11. *MultipleNeighborsWait* → *MultipleNeighborsWait*

HALChanged [StoreHAL]
HALSChanged [StoreHALS]
HATChanged [StoreHAT]
MultipleNeighbors [StartMulNeighTimer]
TimerTick [IF MulNeighTimer expired
PUSH MultipleNeighborsDone]
UpdateZTPOffer [SendOfferToZTPFSM]

12. *ThreeWay* → *MultipleNeighborsWait*

MultipleNeighbors [StartMulNeighTimer]

13. *OneWay* → *MultipleNeighborsWait*

MultipleNeighbors [StartMulNeighTimer]

14. *MultipleNeighborsWait* → *OneWay*

LevelChanged [StoreLevel]
MultipleNeighborsDone [−]

En cuanto a la máquina de estados de mensajes simplificada, la misma contiene las transiciones que se pueden observar en la figura 3.3 y las acciones que contiene se describen a continuación.

1. $OneWay \rightarrow OneWay$

```
HoldTimerExpired [-]}
LIEReceived [PROCESS_LIE]
NeighborAddressAdded [-]
SendLIE [SEND_LIE]
TimerTick [PUSH SendLIE]
```

2. $OneWay \rightarrow TwoWay$

```
NewNeighbor [PUSH SendLIE]
```

3. $TwoWay \rightarrow TwoWay$

```
LIEReceived [PROCESS_LIE]
SendLIE [SEND_LIE]
TimerTick [PUSH SendLIE,
IF HoldTtimer expired
    PUSH HoldTimerExpired]
```

4. $TwoWay \rightarrow ThreeWay$

```
ValidReflection [-]
```

5. $ThreeWay \rightarrow ThreeWay$

```
HALChanged [StoreHAL]
HALSChanged [StoreHALS]
HATChanged [StoreHAT]
LIEReceived [PROCESS_LIE]
SendLIE [SEND_LIE]
TimerTick [PUSH SendLIE,
    IF HoldTimer expired
        PUSH HoldTimerExpired]
UpdateZTPOffer [SendOfferToZTPFSM]
ValidReflection [-]
```

Como se puede observar no se incluye el estado $MultipleNeighborsWait$ ya que no se tienen múltiples vecinos en una misma adyacencia, asumiendo conexiones punto a punto entre cada nodo.

En la máquina de estados de la figura 3.3, cada nodo envía a sus adyacencias mensajes LIE hasta recibir su respuesta, momento en el cual se establece la conexión. Como se mencionó anteriormente, los mensajes TIE, TIRE o TIDE solo se pueden enviar una vez se está en el estado ThreeWay. Como mínimo se necesitan por lo tanto dos mensajes por cada puerto de cada nodo, uno para anunciarse y uno de respuesta. Luego el protocolo sigue mandando mensajes LIE para comprobar la conexión. La cota inferior de mensajes necesaria para que comience la fase de enrutamiento es la siguiente:

$$
\begin{aligned}
M[LIE] &= 2 * (k * \#Spines + \frac{k}{2} * \#Leafs + Tr * \#ToFs) \\
&= 2 * (k * P * \frac{k}{2} + \frac{k}{2} * P * \frac{k}{2} + S * P * \frac{k}{2N}) \\
&= 2 * P * (\frac{k^2}{2} + \frac{k^2}{4} + S * \frac{k}{2N}) \\
&= P * k(\frac{3k}{2} + S/N)
\end{aligned}
$$

Tipo/Direccion	Sur	Norte
Node/Prefix North	Nunca se envía	Siempre se envía
Node South	Envía los mensajes que genera el mismo nodo y los que generan los nodos del mismo nivel	Se envían los mensajes que llegan de niveles mayores al nodo
Prefix South	Solo envía los TIE originados por el mismo nodo	Solo le envía al nodo adyacente que originó el South TIE en primer instancia

Tabla 3.1: Flooding de mensajes TIE.

Calculando S, P y N en función de k se tiene:

$$M[LIE] = \frac{k^2}{R} * (\frac{3k}{2} + \frac{\frac{k^2}{4R}}{\frac{k}{2R}})$$

$$= \frac{k^2}{R} * (\frac{3k}{2} + \frac{k}{2}) = \frac{2k^3}{R}$$

Por lo tanto, una cota inferior de mensajes LIE en el mejor caso es de orden $\Theta(k^3)$, previo a la fase de descubrimiento de la topología. Si bien el protocolo sigue enviando LIEs como forma de confirmar que las conexiones siguen activas luego del primer mensaje TIE.

3.1.2. Análisis de mensajes TIE

Los mensajes TIE tienen un comportamiento distinto según su dirección y según el tipo de mensaje. Para la convergencia en el inicio de la topología se envían solo mensajes de los tipos TIE Node y TIE Prefix, por lo que el análisis se centrará en los mismos y su comportamiento según la dirección en que se envíen, que debido a que se asume la ausencia de enlaces Este-Oeste, van a ser solo en dirección Norte y Sur. A su vez, se asumió que los nodos son todos Flood Repeaters, lo cual es un peor caso de la topología, debido a que no instrumenta un flooding óptimo desde los Leaf hacia los ToF, pero permite un estudio más claro del comportamiento de cada nodo en función de k.

En la tabla 3.1 se puede observar el comportamiento de los mensajes TIE según su tipo y dirección. Este comportamiento se extrajo de la tabla 3 (Normative Flooding Scopes) de [Przygienda y col., 2020]. La cantidad de mensajes que el protocolo debe enviar para converger va a tener la cota mínima siguiente:

$$M[TIE] = M[Node\ N-TIE] + M[Node\ S-TIE] + M[Prefix\ N-TIE]$$
$$+M[Prefix\ S-TIE]$$

Donde N-TIE y S-TIE se refieren a las direcciones Norte y Sur respectivamente. De la tabla 3.1 se describe el comportamiento de los mensajes en cada caso.

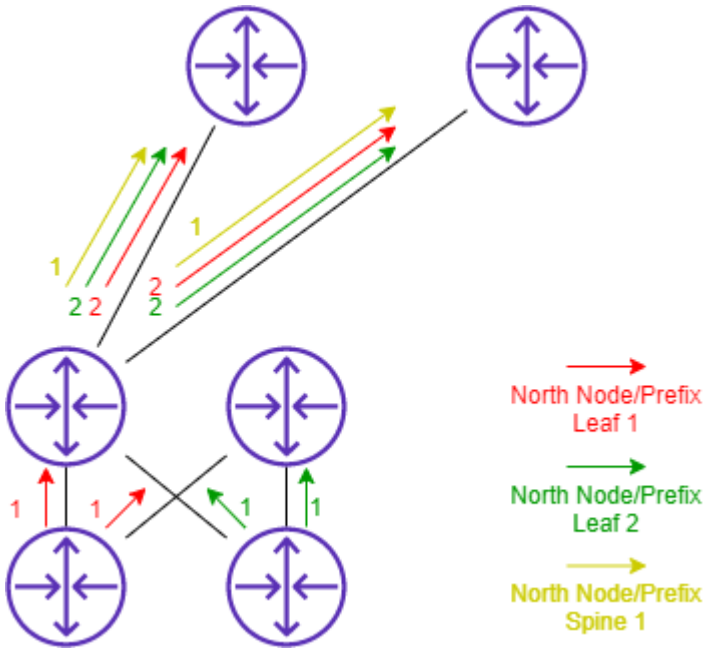


Figura 3.4: Propagación de North Node/Prefix TIEs

Los Node y Prefix North-TIE van a ser enviados hacia el norte por los Leaf y Spine. Observando la figura 3.4 se pueden identificar los siguientes flujos de mensajes.

- Cada Leaf va a enviar TIEs a todos los Spines de su mismo Pod generando $\frac{k}{2}$ mensajes.
- Los TIE originados por los Leaf van a ser enviados por cada Spine a sus ToF (debido a que todos son Flood Repeaters) enviando $(\frac{k}{2})^2$.
- Los Spine van a enviar los TIE generados por ellos mismos a los ToF siendo $\frac{k}{2}$ TIE por Spine.

Dicho flujo de mensajes genera la siguiente expresión:

$$M[Node\ N{-}TIE] = M[Prefix\ N{-}TIE] = \#Leafs{*}(\frac{k}{2}{+}(\frac{k}{2})^2){+}\#Spines{*}\frac{k}{2}$$

$$= P * \frac{k}{2} * (\frac{k}{2} + \frac{k^2}{4}) + P * \frac{k^2}{4}$$

$$= P * \frac{k^2}{4} * (2 + \frac{k}{2})$$

Calculando P en función de K se tiene:

$$M[Node\ N-TIE] = M[Prefix\ N-TIE] = \frac{k}{R} * \frac{k^2}{4} * (2 + \frac{k}{2}) = \frac{k^4}{4R} + \frac{k^3}{R}$$

Luego, los mensajes Node South-TIE van a ser enviados al sur por los ToF y Spine y hacia el norte como reflexión por los Leaf y Spine. La reflexión se debe a que al no tener enlaces Este-Oeste, los nodos de un mismo nivel se conocen entre sí por medio de los nodos de nivel inmediato. Observando la figura 3.5 el flujo va a ser el siguiente.

- En dirección hacia el sur cada ToF va a enviar un nodo S-TIE hacia los Spines.
- Los Spines luego van a calcular la mejor ruta y generar un mensaje hacia sus Leaf.
- En dirección hacia el norte los Leafs van a reenviar a todos sus Spines menos al que originó cada S-TIE que reciben.
- Los Spines van a reenviar a cada ToF menos el que envió cada S-TIE que reciben.

La expresión generada por los flujos mencionados es la siguiente:

$$M[Node\ S-TIE] = M[Node\ S-TIE \text{ hacia el Sur}]$$

$$+M[Node\ S-TIE \text{ hacia el Norte}]$$

$$= P * \frac{k}{2} * \frac{k}{2} * \frac{k}{2} + P * \frac{k}{2} * \frac{S}{N^2} +$$

$$P * k * (\frac{k}{2} - 1) * \frac{k}{2}$$

$$= P * (\frac{k}{2})^3 + P * (\frac{k}{2}) * \frac{S}{N^2} + P * k * (\frac{k}{2} - 1) * \frac{k}{2}$$

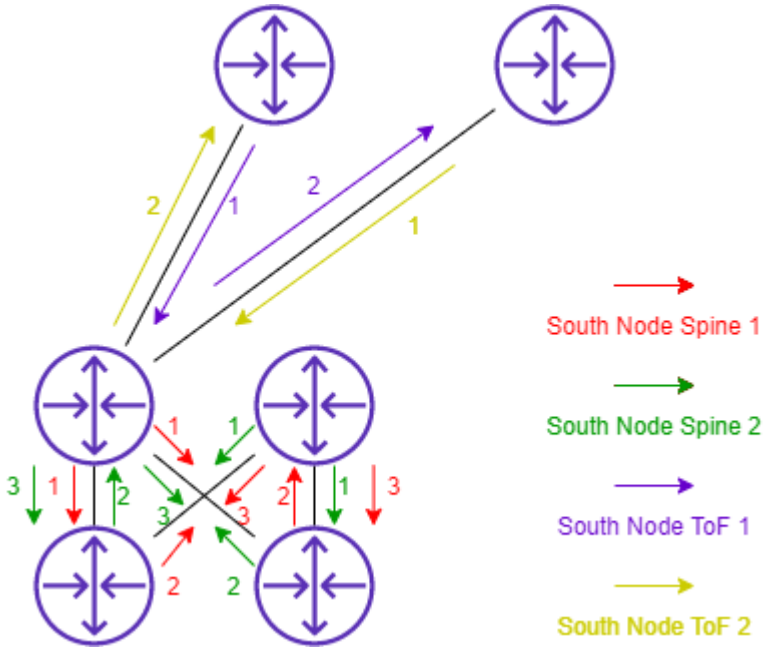


Figura 3.5: Propagación de South Node TIEs

$$= P * k * (\frac{k^2}{8} + \frac{S}{2N^2} + \frac{k^2}{4} - \frac{k}{2})$$

$$= P * k * (\frac{3k^2}{8} - \frac{k}{2} + \frac{S}{2N^2})$$

Calculando P, S y N en función de k se tiene:

$$M[Node\ S-TIE] = \frac{k}{R} * k * (\frac{3k^2}{8} - \frac{k}{2} + \frac{\frac{k^2}{4R}}{2 * (\frac{k}{2R})^2})$$

$$= \frac{k^2}{R} * (\frac{3k^2}{8} - \frac{k}{2} + \frac{R}{2})$$

$$= \frac{3k^4}{8R} - \frac{k^3}{2R} + \frac{k^2}{2}$$

En último lugar, se van a generar los flujos de mensajes a observar en la figura 3.6 para los South Prefix TIE, los cuales se describen a continuación.

- Los mensajes Prefix South-TIE van a ser enviados hacia el sur por los nodos Spine y ToF que los originen.
- Los Leaf y Spine van a reenviar el mensaje al nodo que lo originó.

Esta propagación de mensajes genera la siguiente expresión.

$$M[Prefix\ S-TIE] = M[Prefix\ S-TIE \text{ hacia el Sur}]$$

$$+M[Prefix\ S-TIE \text{ hacia el norte}]$$
$$= \frac{k}{2} * (P * \frac{k}{2}) + S * P * \frac{\frac{k}{2}}{N} + P * (k) * \frac{k}{2}$$
$$= P * (\frac{k}{2})^2 + P * S * \frac{k}{2N} + P * \frac{k^2}{2}$$
$$= P * k * (\frac{3k}{4} + \frac{S}{2N})$$

Calculando P, S y N en función de k se tiene:

$$M[Prefix\ S-TIE] = \frac{k}{R} * k * (\frac{3k}{4} + \frac{\frac{k^2}{4R}}{\frac{2k}{2R}}) = \frac{k^3}{R}$$

Finalmente el total es:

$$M[TIE] = 2 * (\frac{k^4}{4R} + \frac{k^3}{R}) + \frac{3k^4}{8R} - \frac{k^3}{2R} + \frac{k^2}{2} + \frac{k^3}{R}$$
$$= k^4 * (\frac{1}{2R} + \frac{3}{8R}) + k^3(\frac{2}{R} - \frac{1}{2R}) + \frac{k^2}{2} = \frac{7k^4}{8R} + \frac{3k^3}{2R} + \frac{k^2}{2}$$

Se puede determinar por lo tanto que una cota inferior de TIEs para la convergencia tendría un orden de $\Theta(k^4)$ mensajes.

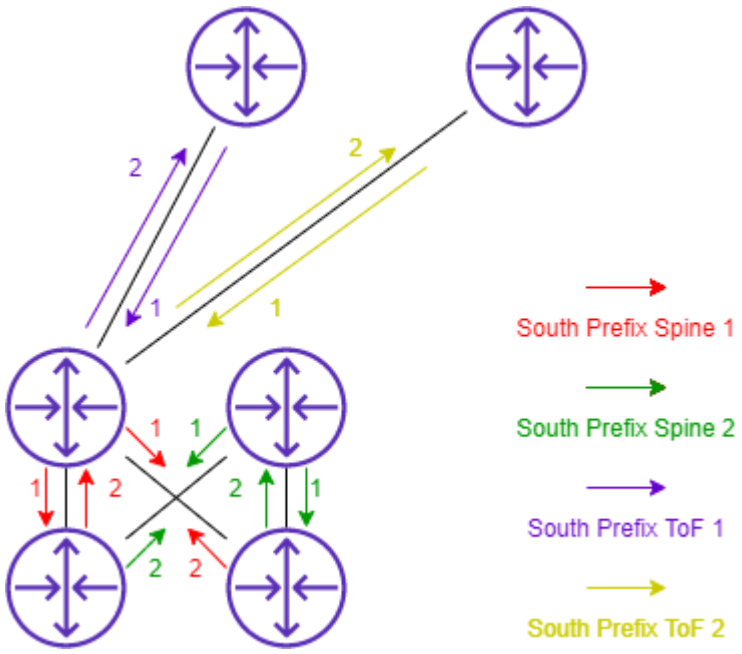


Figura 3.6: Propagación de South Prefix TIEs

3.2. Análisis de BGP

El protocolo BGP no realiza un proceso de descubrimiento de adyacencias como el caso de los mensajes LIE en RIFT o HELLO en OSPF por ejemplo. En su lugar, actúa a nivel de aplicación utilizando como protocolo de transporte TCP. Luego que se establece el *handshake* de TCP, los nodos envían un mensaje OPEN para establecer la conexión.

En la figura 3.7 se muestra la máquina de estados de BGP para el establecimiento de una conexión, especificada en [Rekhter y col., 2006], mientras que en la figura 3.8 se muestra una versión simplificada donde no se consideran los posibles errores. Como se puede ver en ambas máquinas de estado, cada nodo intercambia mensajes OPEN con sus pares BGP. Luego de que se confirma la conexión, se envía un primer mensaje KEEPALIVE y después de ello, se comienzan a enviar mensajes UPDATE. El análisis se centrará en estos últimos que se encargan de enviar la información necesaria de enrutamiento.

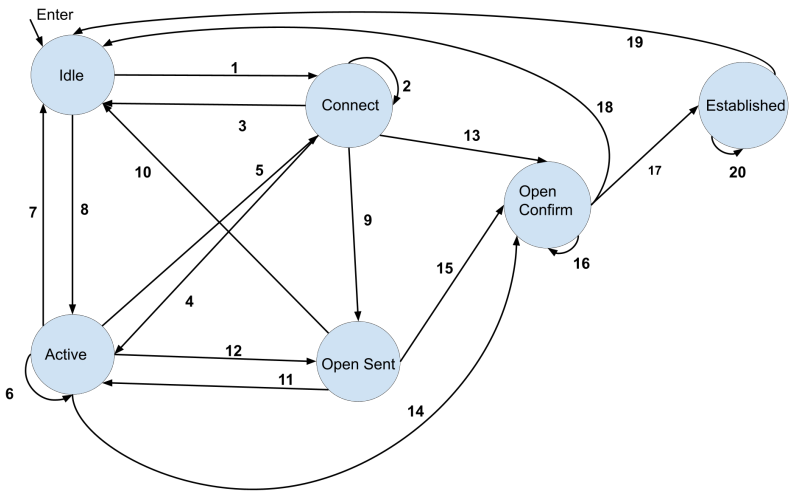


Figura 3.7: Máquina de estados de BGP

Se detallan las acciones de cada transición de la máquina de estados de BGP de la figura 3.7 a continuación. Dicha información se extrajo de [Rekhter y col., 2006] y la información sobre los distintos parámetros y contadores puede ser consultada en dicho estándar.

1. *Idle → Connect*

```
IF Manual Start or AutomaticStart
    Assign BGP Resources
    ConnectRetryCounter = 0
    Set(ConnectRetryTimer)
    Init TCP Connectoion with Peer
    Listen Incomming TCP connections
    from peer
```

2. *Connect → Connect*

```
IF ConnectRetryTimer_Expires
    Drop TCP Connection
    Restart ConnectRetryTimer
    Stop DelayOpenTimer
    Init TCP Connection with Peer
    Listen Incomming TCP connections
    from Peer
IF TCP connection success and
DelayOpen is TRUE
    stop(ConnectRetryTimer)
    ConnectRetryTimer = 0
    set(DelayOpenTimer)
```

3. *Connect → Idle*

```
IF Manual Stop
    Drop TCP connection
    Release BGP Resources
    ConnectRetryCounter = 0
    Stop(ConnectRetryTimer)
    ConnectRetryTimer = 0
IF TCP connection fails and
DelayOpenTimer stopped
    stop(ConnectRetryTimer)
    drop TCP connection
    Release BGP resources
IF header checking error or other
events*
    release BGP resources
    drops TCP connection
    actions for each event
```

4. *Connect → Active*

```
IF TCP connection fails and
DelayOpenTimer running
    restart(ConnectRetryTimer)
    stop(DelayOpenTimer)
    listen Incomming TCP connections from peer
```

5. *Active → Connect*

```
IF ConnectRetryTimer_Expires
    restart(ConnectRetryTimer)
    start TCP connection with peer
    listen incomming TCP
    connections from peer
```

6. *Active → Active*

```
IF TCP Connection success and
DelayOpen is True
    Stop(ConnectRetryTimer)
    ConnectRetryTimer = 0
```

7. *Active → Idle*

```
IF Manual Stop
    IF DelayOpenTimer running and
SendNotificationWithoutOpen is set
        Send(Notification) with a Cease
        release BGP resources
        drop TCP connection
        ConnectRetryCounter = 0
        stop(ConnectRetryTimer)
        ConnectRetryTimer = 0
IF TCP Connection fails
    restart(ConnectRetryTimer)
    stop(DelayOpenTimer)
    DelayOpenTimer = 0
    ConnectRetryCounter + 1
IF header checking error or other events*
    release BGP resources
    drops TCP connection
    actions for each event
```

8. *Idle → Active*

```
IF ManualStart_with_PassiveTcpEstablishment or
AutomaticStart_with_PassiveTcpEstablishment
    Assign Resources
    ConnectRetryCounter = 0
    Set(ConnectRetryTimer)
    Listen Incomming TCP
connections
    form Peer
```

9. *Connect → Open Sent*

```
IF DelayOpenTimer_Expires
    Send(OPEN)
    Set(HoldTimer)
IF TCP Connection success and
DelayOpen is FALSE
    Stop(ConnectRetryTimer)
    ConnectRetryTimer = 0
    BGP initialization
    Send(OPEN)
    set(HoldTimer)
```

10. *Open Sent → Idle*

```
IF Manual Stop or AutomaticStop
    Send(Notification)  with a Cease
    release BGP resources
    drop TCP connection
    ConnectRetryCounter = 0 or + 1
IF HolderTimeExpires
    Send(Notification)
    ConnectRetryTimer=0
    release BGP resources
    drop TCP connection
    ConnectRetryCounter + 1
IF header checking error or collision
or other events*
    release BGP resources
    drops TCP connection
    actions for each event
```

11. *Open Sent → Active*

```
IF TCP connection fails
    restart(ConnectRetryTimer)
    stop(DelayOpenTimer)
    listen Incomming TCP connections
    from peer
```

12. *Active → Open Sent*

```
IF DelayOpenTimer_Expires
    ConnectRetryTimer = 0
    stop(DelayOpenTimer)
    clear(DelayOpenTimer)
    Send(OPEN)
    Set(HoldValue)
IF TCP Connection success and
DelayOpen is  FALSE
    ConnectRetryTimer = 0
    BGP Initialization
    Send(OPEN)
    Set(HoldTimer)
```

13. *Connect → Open Confirm*

```
IF Received(OPEN) and DelayOpenTimer
is running
    stop(ConnectRetryTimer)
    ConnectRetryTimer = 0
    BGP initialization
    stop(DelayOpenTimer)
    DelayOpenTimer = 0
    Send(OPEN)
    Send(Keepalive)
    IF initial HoldTimer = 0
            reset(KeepaliveTimer)
            HoldTimer = 0
    ELSE
            start(KeepAliveTimer)
            reset(HoldTimer)
```

14. *Active → Open Confirm*

```
IF Received(OPEN) and DelayOpenTimer
is running
    stop(ConnectRetryTimer)
    ConnectRetryTimer = 0
    BGP initialization
    stop(DelayOpenTimer)
    DelayOpenTimer = 0
    Send(OPEN)
    Send(Keepalive)
    IF initial HoldTimer = 0
            reset(KeepaliveTimer)
            HoldTimer = 0
    ELSE
            start(KeepAliveTimer)
            reset(HoldTimer)
```

15. *Open Sent → Open Confirm*

```
IF Received(OPEN)
    DelayOpenTimer = 0
    ConnectRetryTimer = 0
    Send(KeepAlive)
    Set(KeepAliveTimer)
    Set(HoldTimer)
```

16. *Open Confirm → Open Confirm*

```
IF KeepaliveTimer_Expires
    Send(KeepAlive)
    restart(KeepAliveTimer)
```

17. *Open Confirm → Established*

```
Received(KeepAlive)
restart(HoldTimer)
```

18. *Open Confirm → Idle*

```
IF Manual Stop or AutomaticStop
    Send(Notification)    with a Cease
    release BGP resources
    drop TCP connection
    ConnectRetryCounter = 0 or + 1
    ConnectRetryTimer = 0
IF HolderTimeExpires before Keepalive
Mesage or (Received(OPEN) and
collision detected) or
Send(Notification)
    ConnectRetryTimer=0
    release BGP resources
    drop TCP connection
    ConnectRetryCounter + 1
IF header checking error or collision
or other events*
    release BGP resources
    drops TCP connection
    actions for each event
IF TCPConnectionFails or Received(Notification)
    ConnectRetryTimer = 0
    release BGP resources
    drop TCP Connection
    ConnectRetryCounter + 1
IF Received(Notification) with version error
    ConnectRetryTimer = 0
    release BGP resources
    drop TCP connection
```

19. *Established → Idle*

```
IF Manual Stop or Automatic Stop or
Collision or Received(Notification) or
Update error or other errors*
    Send(Notification)
    Drop TCP connection
    deletes all routes associated with
    this connection
    Release BGP Resources
    ConnectRetryCounter = 0
    Stop(ConnectRetryTimer)
    ConnectRetryTimer = 0 or + 1
IF HoldTimerExpires
    Send(Notification)
    ConnectRetryTimer = 0
    release BGP resources
    drop TCP connection
    ConnectrionRetryCounter + 1
```

20. *Established → Established*

```
IF KeepAliveTimer_Expires
    Send(KeepAlive)
    If HoldTIme != 0
            restart(KeepAliveTimer)
IF Received(KeepAlive)
    If HoldTIme != 0
            restart(KeepAliveTimer)
IF  Received(UPDATE)
        Process Message
    If HoldTIme != 0
            restart(KeepAliveTimer)
```

La máquina de estados simplificada de la figura 3.8 fue referenciada de la misma forma que la máquina de estados de la figura 3.7, conteniendo las transiciones descriptas anteriormente y necesarias para el establecimiento de la conexión BGP.

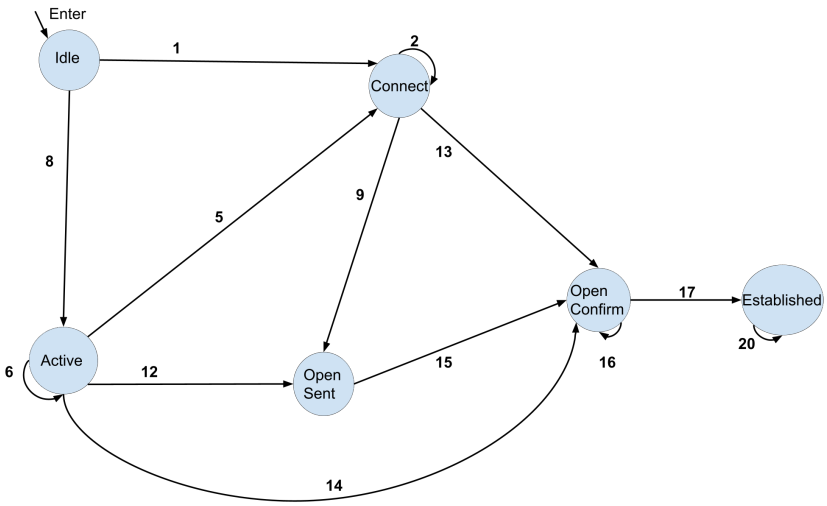


Figura 3.8: Máquina de estados de BGP simplificada

3.2.1. Análisis de mensajes UPDATE

En la figura 3.9 obtenida de [“Basic Configuration Concepts”, 2018] se puede apreciar el *pipeline* de operaciones que se aplica a los prefijos anunciados por un mensaje UPDATE. Como se mencionó anteriormente el proceso de decisión se va a basar en el caso de BGP en el Data Center en el atributo AS-Path y su largo.

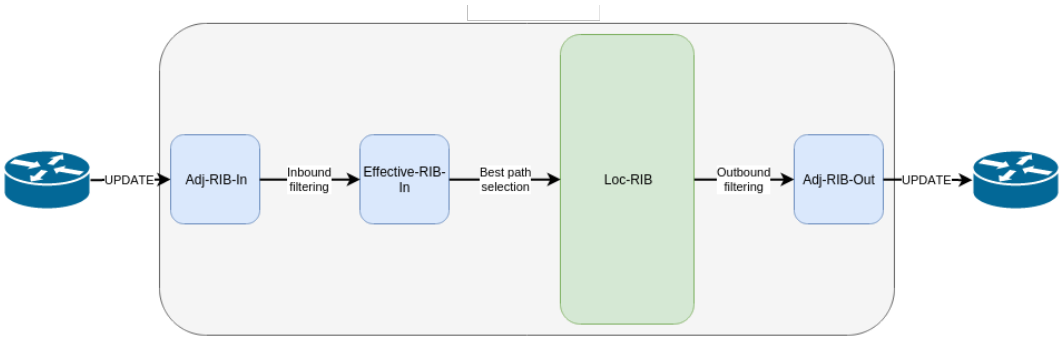


Figura 3.9: Pipeline de BGP [“Basic Configuration Concepts”, 2018]

Para analizar los mensajes UPDATE, primero se analizará el costo de anunciar un prefijo desde un nodo Leaf. En la figura 3.10 se pueden ver los mensajes UPDATE generados por cada nodo

1. Un Leaf que conoce un nuevo prefijo lo procesa y envía un UPDATE a sus vecinos, que son todos los Spine de su PoD. Estos son $\frac{k}{2}$ mensajes.

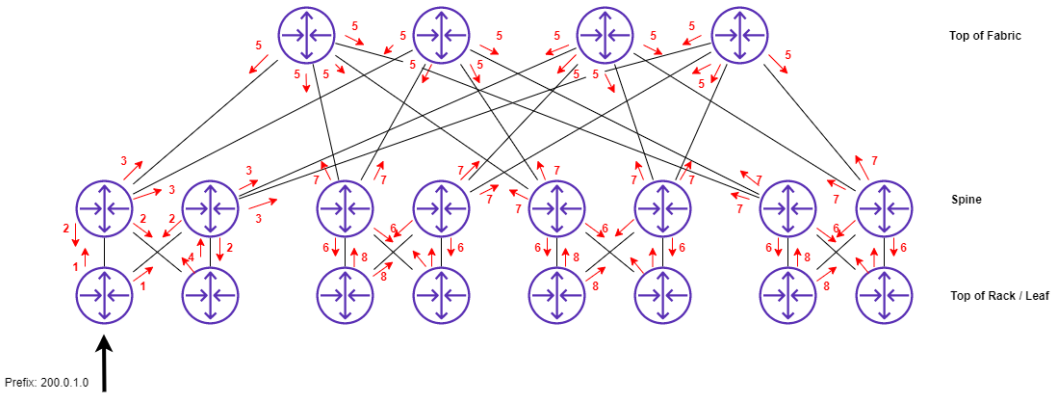


Figura 3.10: Propagación de anuncio de un prefijo

2. Los Spine de ese PoD van a procesar ese nuevo prefijo y si no lo conocían van a enviar un mensaje UPDATE a todos sus vecinos. En particular a los Leaf del mismo PoD. Por lo tanto se tienen $\frac{k^2}{4}$ mensajes, o $\frac{k}{2}$ mensajes por Spine.
3. Los Spine también van a enviar mensajes UPDATE a sus ToF, lo cual son $\frac{k^2}{4}$ mensajes o $\frac{k}{2}$ mensajes por Spine.
4. En el paso 2 los Spine enviaron mensajes a sus Leaf del mismo PoD. Aquellos que no conocían el prefijo van a procesarlo y enviarlo a todos sus Spine. Si se asume que es un prefijo nuevo en la topología, solo el Leaf que originó el prefijo lo conoce y no va a volver a enviarlo dado que se encuentra en el AS-Path del mismo. Por lo tanto se tienen $\frac{k}{2} - 1$ Leafs que envian mensajes a $\frac{k}{2}$ Spines.
5. Los ToF van a procesar el nuevo prefijo y si no lo conocen van a enviarlo a todos sus vecinos. Se tienen por lo tanto $S * Tr$ mensajes.
6. Los Spine de los otros PoD van a enviar a sus Leaf el nuevo prefijo luego de procesarlo. Los Spine del PoD donde se originó el mismo lo van a procesar y van a encontrar que su número AS está en el AS-Path, por lo cual no van a reenviar el prefijo. Por lo tanto se tienen $(P-1) * \frac{k^2}{4}$ mensajes.
7. Los Spine de los otros PoD además van a reenviar a sus ToF el prefijo luego de procesarlo, por lo cual se tienen $(P-1) * \frac{k^2}{4}$ mensajes.
8. Los Leaf de los otros PoD van a reenviar el prefijo a sus Spine luego de procesarlo. Se tienen por ello $(P-1) * \frac{k^2}{4}$ mensajes.

Sumando los flujos descriptos anteriormente se tiene la siguiente expresión.

$$M[prefijo] = \frac{k}{2}+\frac{k^2}{4}+\frac{k^2}{4}+(\frac{k}{2}-1)*\frac{k}{2}+S*Tr+(P-1)\frac{k^2}{4}+(P-1)\frac{k^2}{4}+(P-1)\frac{k^2}{4}$$

$$= \frac{3k^2}{4} + (P-1)*\frac{k^2}{4} + S*Tr$$

$$= \frac{3Pk^2}{4} + S*Tr$$

$$= \frac{3k^3}{4R} + \frac{k^2}{4R}*\frac{k}{R}*\frac{k}{2N}$$

$$= \frac{3k^3}{4R} + \frac{k^2}{4R}*\frac{k}{R}*\frac{k}{\frac{2k}{2R}}$$

$$= \frac{k^3}{4R} + \frac{k^3}{4R} = \frac{k^3}{2R}$$

Por lo tanto una cota inferior de UPDATES para enviar un prefijo sería de orden $\Theta(k^3)$. Ahora asumiendo que cada Leaf origina un prefijo tenemos:

$$M[UPDATE] = \#Leafs * \frac{k^3}{2R}$$

$$= P * \frac{k}{2} * \frac{k^3}{2R} = \frac{k^2}{2R} * \frac{k^3}{2R} = \frac{k^5}{4R^2}$$

La cota inferior de mensajes para que cada Leaf envíe un prefijo es de $\Theta(k^5)$ mensajes. En caso de que los Leaf envíen más prefijos el orden se mantiene, multiplicándose el conteo por una constante, por ejemplo p prefijos.

3.3. Conclusiones

En resumen, en este capítulo se detalló un análisis de complejidad de los dos protocolos de enrutamiento distribuidos para Data Center centrales en el presente estudio: BGP y RIFT. Este análisis refiere en particular a la situación donde ocurre el *bootstrap* de una topología Fat Tree. Para cada uno de estos protocolos, se obtuvieron cotas inferiores de cantidad de mensajes, teniendo en cuenta ciertas hipótesis planteadas al comienzo del capítulo de manera de representar lo más fielmente posible un "caso ideal" de funcionamiento de la topología.

Primero, se evaluó la cantidad de mensajes de dos etapas de RIFT: el

envío de mensajes LIE, que funcionan para descubrir la topología, y el envío de mensajes TIE, donde se distribuye la información relevante al enrutamiento propiamente dicho, esto es, los prefijos que existan. Para ello, se analizó y redujo la máquina de estados de LIE, teniendo como hipótesis la ausencia de mecanismo ZTP por simplicidad e ignorando las transiciones que no cumplieran las hipótesis planteadas al comienzo de la sección, o no fueran relevantes para el descubrimiento entre vecinos. Se repitió esto último para los mensajes TIE, pero esta vez se siguieron las reglas de envío de mensajes definidas en [Przygienda y col., 2020] para estos últimos.

En función de los parámetros definidos en la introducción de la sección, se pudo concluir que RIFT necesita de al menos un orden de $\Theta(k^3)$ mensajes LIE para que se descubran todos los nodos de la topología, y una cantidad adicional del orden de $\Theta(k^4)$ mensajes TIE, para que se diseminen p prefijos por cada Leaf a todos los nodos de la topología, donde se asumió que $p = 1$.

Por otro lado, se analizó la cantidad de mensajes de BGP, centrándose en los mensajes UPDATE, que son los mensajes que se encargan de diseminar los prefijos. No se evaluaron los mensajes OPEN, dada su poca relevancia estadística en la convergencia, debido a que BGP utiliza TCP como protocolo de transporte, lo que genera un envío reducido de este tipo de mensajes. Teniendo en cuenta el pipeline de BGP y la configuración específica que se utiliza para Data Centers, tomando las mismas hipótesis sobre la topología que para RIFT, se pudo concluir que se necesitan del orden de $\Theta(k^5)$ mensajes al menos para que BGP converja.

Para comprobar los resultados obtenidos, se debieron realizar pruebas sobre un entorno de emulación controlado. Para ello, se utilizó Wireshark para la disección de paquetes e implementaciones correspondientes de cada protocolo como se detallará en la sección 5.3. En el caso particular de RIFT, al ser un protocolo en etapa de desarrollo, se vió la necesidad de desarrollar un disector propio en Wireshark, dada la inexistencia de un disector del mismo dentro de los protocolos soportados oficialmente. Los detalles y decisiones sobre la implementación de este disector se explican en el capitulo 4.

Capítulo 4

Disector de RIFT

Los protocolos de red se diseñan para comunicar diferentes dispositivos de red, y definen mecanismos para identificar y establecer conexiones, reglas de formato y convenciones específicas para la transferencia de datos. Los analizadores de tráfico (también conocidos como *packet sniffers*) son herramientas de software específicas que interceptan y registran el tráfico de una red que atraviesa un enlace de red a través de captura de paquetes. Los paquetes capturados pueden luego ser analizados decodificando los datos en crudo de los paquetes y visualizarse exhibiendo varios campos para interpretar el contenido, usando por ejemplo la herramienta *Wireshark* [Orebaugh y col., 2007]

General header	Packet Header	Packet Data	Packet Header	Packet Data	...

Figura 4.1: Formato de archivo pcap

El análisis de paquetes de red es por lo general de gran ayuda para recolectar y reportar estadísticas de red, depurar el comportamiento de aplicaciones, y realizar *network forensics*, típicamente utilizando agregaciones de tráfico basadas en los headers TCP/IP; inspeccionar el payload de un paquete puede contribuir a la investigación adicional del comportamiento de aplicaciones, y es particularmente útil identificar (e interceptar) amenazas de seguridad, tales como DDoS[1]. Técnicas de DPI[2] son implementadas en middleboxes tales como los IDS[3].

[1]Distributed Denial of Service.
[2]Deep Packet Inspection.
[3]Intrusion Detection Systems.

El estándar de facto de formato de captura es *pcap*, implementado por la API de libpcap, originalmente desarrollada por el equipo de tcpdump ["TCPDUMP/LIBPCAP public repository", 2020]. Pcap es un formato binario, cuya estructura general comprende un header global que contiene un *magic number* (para identificar la versión del formato de archivo y el byte order), el offset GMT, la precisión de timestamp, el largo máximo de paquetes capturados (en octetos), y el tipo de enlace de datos. Esta información es seguida por cero o más registros de datos del paquete capturado. Cada paquete capturado comienza con el timestamp en segundos, el timestamp en microsegundos, el número de octetos del paquete guardado en el archivo, y el largo real del paquete. La estructura general se muestra en la Figura 4.1.

El sucesor de pcap es el Pcap Next Generation Capture File Format (pcapng), en desarrollo por la IETF [Tüxen y col., 2020]. Una guía completa sobre el uso práctico de Wireshark y tcpdump, incluyendo una discusión sobre formatos de tráfico y otras herramientas puede encontrarse en [Chapman, 2016; Sikos, 2020].

El disector de RIFT fue diseñado como un plugin de Wireshark e implementado en el lenguaje C, conforme a la API que provee la herramienta en dicho lenguaje.

4.1. Diseño

En la figura 4.2, se muestra el cabezal exterior de un paquete RIFT, que es transportado dentro de un payload UDP. El diseño propuesto sigue el envoltorio de seguridad (*Security Envelope*) para paquetes RIFT que se muestra en la figura. Se definen tres niveles a continuación:

1. *Outer Security Envelope Header*: aquí el disector debe identificar un conjunto específico de campos estáticos.
2. *TIE Origin Security Envelope Header*: al igual que en la etapa previa, aquí el disector debe identificar un conjunto específico de campos estáticos. La única diferencia entre esta etapa y la anterior es que este conjunto de campos está presente si y sólo si el tipo del paquete RIFT es un TIE.
3. *Serialized RIFT Model Object*: en esta etapa, el disector debe procesar

un conjunto de campos dinámicos, que conforman la codificación del protocolo Binary de Thrift [“Thrift Binary Protocol”, 2020] definido para RIFT.

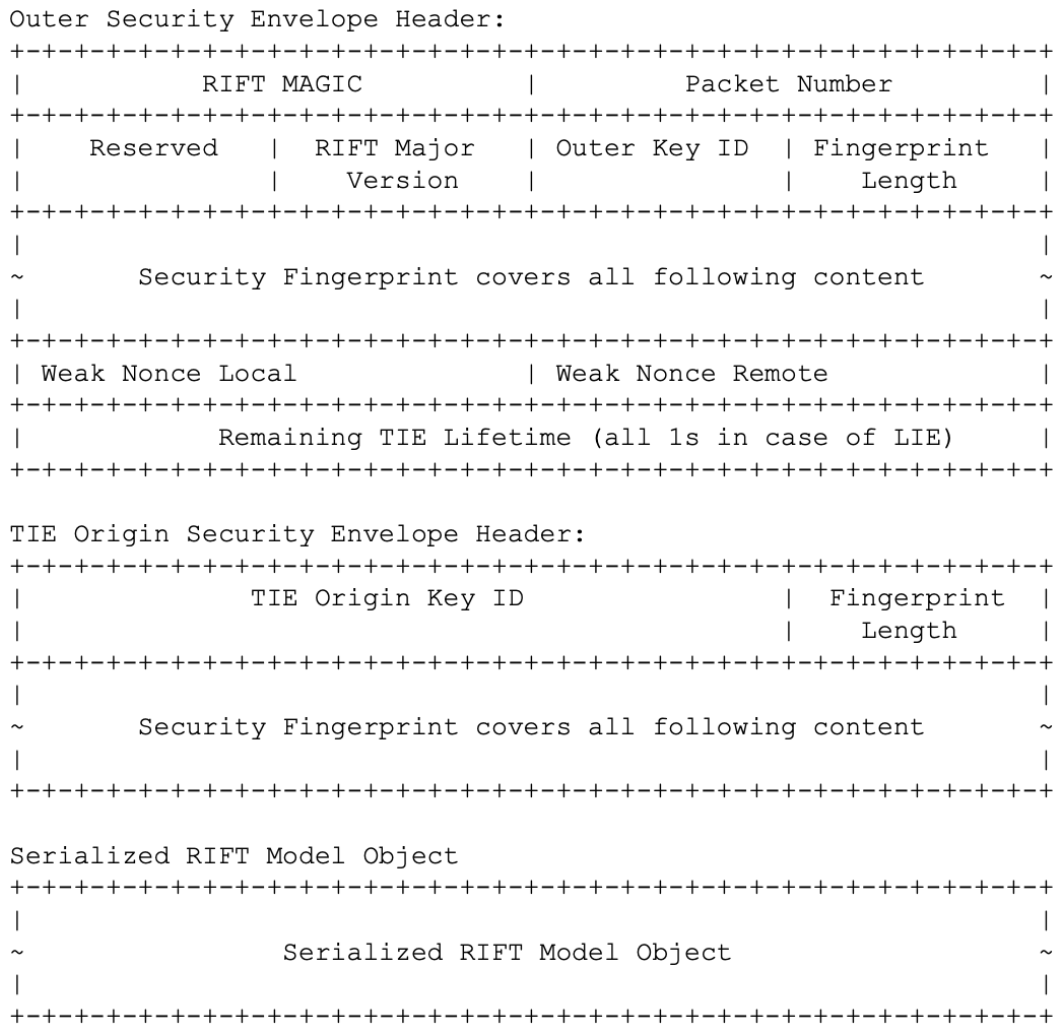


Figura 4.2: Security Envelope, extraído de Przygienda y col., 2020

Los primeros dos niveles están diseñados con un enfoque clásico, i.e., el disector debe seguir una definición estática que asigna un rango de bytes a un campo. El tercer nivel representa un cambio de paradigma. Thrift es un lenguaje de definición de interfaz y protocolo de comunicación binario utilizado para definir y crear servicios para numerosos lenguajes [Agarwal y col., 2007]. Thrift permite definir interfaces de servicio y data-types en un archivo de especificación. Lo último permite una operación dinámica de los servicios y aplicaciones, brindando al modelador la capacidad de extender los data-types

en una forma simple al modificar el archivo de definición. Este mecanismo provee un proceso rápido para agregación o modificado de datos.

Por consiguiente, con este nuevo enfoque, un paquete RIFT necesita ser decodificado conociendo con antelación cómo están especificados los datos en los archivos de definición de Thrift y cómo los data-types y estructuras son codificados. Es importante mencionar que existe un compilador de Thrift que genera un decodificador de Thrift basado en un modelo dado. Consecuentemente, se pudieron identificar dos opciones para diseñar la disección del payload serializado de RIFT:

1. Pasar esta parte del paquete binario al back-end de Thrift; o
2. Escribir el código C siguiendo la codificación definida en Thrift para los data-types involucrados. Esto necesita hacerse siguiendo el esquema para los elementos de información, cuya IDL[1] es Thrift.

La interfaz de usuario de Wireshark permite al usuario resaltar algunos campos en particular en el paquete decodificado. Además, resalta los bytes correspondientes en el hex dump[2] del paquete binario.

En ese sentido, el decodificador de Thrift que es utilizado en Wireshark requiere (*a*) tener conocimiento del orden preciso en el cual los campos fueron codificados en el mensaje binario, que podrían potencialmente no ser la misma secuencia que en el modelo; y (*b*) tener conocimiento de la correspondencia entre los bytes en el mensaje binario y los campos en el mensaje decodificado. Actualmente, el código generado por el compilador de Thrift no permite generar esta información, por lo que el disector desarrollado sigue la opción (*2*).

4.2. Implementación

Para implementar el disector como plugin de Wireshark, la herramienta brinda dos APIs distintas, una en el lenguaje C y otra en el lenguaje Lua. El disector desarrollado, fue escrito en el lenguaje C, debido a que es el lenguaje

[1] Interface Definition Language.
[2] Vista hexadecimal de los datos del paquete.

más utilizado para este fin, además de tener una notoria mejora en performance en comparación con Lua, y ser más fácil su distribución y su contribución al proyecto de Wireshark en caso de ser deseable. Todos los paquetes RIFT son transportados sobre UDP. Por lo tanto, la presente implementación hereda la disección del header UDP y construye una disección completa del payload UDP.

En primer lugar, se realiza la disección del Security Envelope Header, el cual está especificado en el draft de RIFT [Przygienda y col., 2020] y tiene el formato que se muestra en la figura 4.2. Esta implementación sigue una representación estática de los bits mapeados para cada campo, e.g, los primeros cuatro bytes representan el valor del *RIFT Magic*[1] para el paquete, seguido por los siguientes cuatro bytes que representan el valor del *Packet Number*.

Teniendo en cuenta que el protocolo no estaba asociado a ningún rango particular de puertos al momento de desarrollo del disector, éste se implementó como un disector heurístico de Wireshark, donde se corrobora que el comienzo del payload UDP contenga el valor adecuado de RIFT Magic (hexadecimal `0xA1F7`). Luego, el disector se encarga de decodificar todos los campos del header de seguridad, en el orden establecido (Packet Number, Reserved bytes, RIFT Major version, etc).

Finalizada esta etapa, luego de realizar algunos tests para verificar la correcta disección del header, se desarrolló el resto del disector, que se encarga del payload serializado con Thrift (como se muestra en la figura 4.2). Cabe destacar que este payload contiene dentro del mismo, codificado según Thrift, otro nuevo header y payload (PacketHeader y PacketContent, como se observa en la especificación que se encuentra disponible en [“encoding.thrift”, 2020]), donde se indica en el primer caso valores comunes a todos los paquetes (versión, node level, sender-ID) y en el segundo se encuentra el tipo de paquete propiamente dicho. Existen cuatro tipos de paquetes en RIFT: LIE, TIE, TIDE y TIRE. El disector realiza una disección completa de cualquiera de estos cuatro tipos de paquetes.

La capacidad de realizar una disección completa de los paquetes de tipo

[1] Identificador especial para reconocer un paquete RIFT.

TIE es de gran importancia, debido a que este tipo de mensajes se intercambian entre los nodos de RIFT para anunciar la topología de red (e.g. enlaces y prefijos). En particular, con la identificación y disección de todos los TIEs intercambiados entre los nodos RIFT, se pudo estudiar la convergencia del protocolo en una topología dada.
Por otro lado, con la disección de los mensajes LIE, equivalentes a los HELLOs en protocolos IGP, se puede observar en Wireshark, los mensajes intercambiados sobre todos los enlaces entre sistemas ejecutando RIFT para formar adyacencias three-way.

Con la implementación y decodificación completa de los mensajes TIDE y TIRE, el disector permite tener una visión completa de los mensajes intercambiados entre los nodos. Esta capacidad permite a ambos tener un análisis completo del comportamiento del protocolo y, dado que el protocolo RIFT aún se encuentra en etapa de desarrollo, servir potencialmente como herramienta de debug para los desarrolladores de RIFT.

En la figura 4.3, se muestra un ejemplo de cómo este tipo de estructuras son codificadas en Thrift, mostrando la manera en que una estructura dada definida con el IDL de Thrift se traduce en bytes. Como se observa en la figura, al decodificar una estructura, debe hacerse considerando que para cada campo, debe decodificarse un byte adicional que indica el tipo. Luego de eso, la decodificación continúa con los dos bytes que indican su identificador, y finalmente su valor. Por otra parte, un byte nulo indica que se terminó de recorrer esa estructura.

El disector desarrollado fue probado con dos implementaciones distintas. Por un lado se probó con la implementación de código abierto de Bruno Rijsman, rift-python [“Routing in Fat Trees implementation in Python”, 2018] en el ambiente Kathará. Particularmente para esta versión, los desarrolladores estaban implementando la desagregación negativa y la posibilidad de usar el disector fue de gran ayuda para la depuración de dicha funcionalidad. En la figura 4.4 se ve un ejemplo de una captura de desagregación negativa en Wireshark, que muestra los prefijos que se deben eliminar. Por otro lado, se experimentó con una versión prototipo de la implementación de Juniper. En la figura 4.5 se puede observar un caso de captura para dicha implementación.

```
Struct ::=  'struct' Identifier 'xsd_all'? '{' Field* '}

Binary protocol field header and field value:
+--------+--------+--------+--------+...+--------+
| type   | field id        | field value         |
+--------+--------+--------+--------+...+--------+

Binary protocol stop field:
+--------+
|00000000|
+--------+
```

Figura 4.3: Ejemplo de codificación de un struct usando el protocolo Binary de Thrift

Entre las distintas implementaciones se respetaron los tipos de datos obligatorios en la especificación de RIFT, pero se observó que algunos tipos opcionales se diferenciaban. Por lo tanto se tuvo en cuenta que los datos opcionales fueran contemplados pero no exigidos en en la disección de paquetes.

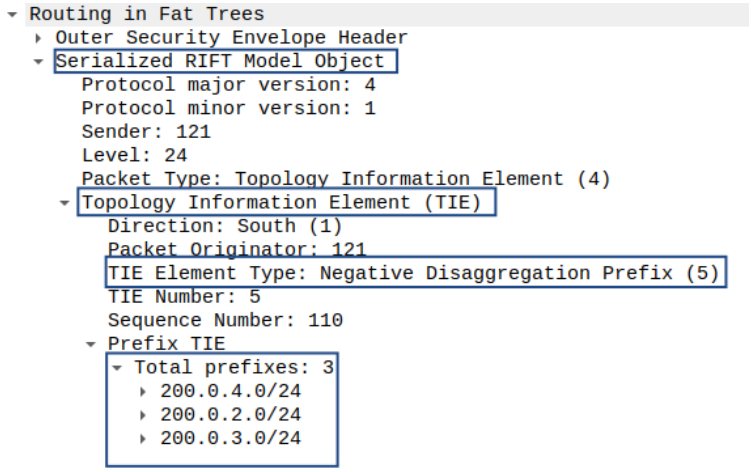


Figura 4.4: Captura de desagregación negativa en rift-python

Con el disector definido en el presente capítulo, se obtuvo la herramienta principal para la inspección de paquetes de RIFT usando la herramienta Wireshark. Con ella se puede distinguir los distintos tipos de mensajes que envía RIFT, brindando la posibilidad realizar el conteo de ocurrencias de cada uno de estos para el trabajo experimental que se realizó, el cual se especifi-

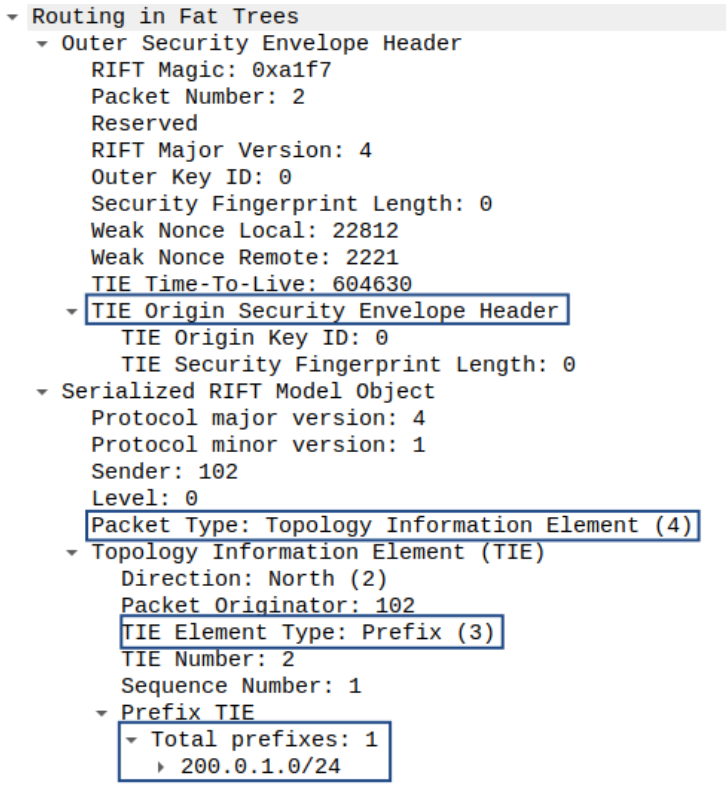


Figura 4.5: Captura de mensaje en implementación de RIFT para Juniper

ca en la sección 5.3. El disector se encuentra disponible de manera pública en ["rift-c-dissector", 2021], donde se tiene además un Dockerfile con la automatización del proceso de instalación de Wireshark y el *plugin* creado, así como una especificación del proceso de instalación. Para la implementación del disector se utilizó la documentación oficial de Wireshark, se puede leer en particular ["wireshark/wireshark: Read-only mirror of Wireshark's Git repository at https://gitlab.com/wireshark/wireshark." 2021] para más información.

Resulta de importancia destacar además que el trabajo descripto en este capítulo se presentó en conjunto con los supervisores a la conferencia *IFIP/IEEE International Symposium of Integrated Network Management (IM 2021)* ["IM 2021", 2021] bajo el título de *Routing in Fat Trees: a protocol analyzer for debug and experimentation* y fue aceptado como un *short paper*.

Capítulo 5

Desarrollo experimental

Se realizaron distintos experimentos con el fin de comprender algunas de las soluciones conocidas para enrutamiento en el Data Center. Para ello se utilizó como entorno de trabajo el framework Kathará [Bonofiglio y col., 2018].
Los objetivos propuestos para el desarrollo de experimentos fueron los siguientes:

1. Trabajar en un entorno de emulación de redes como el caso de Kathará y conocer sus características.
2. Desplegar la solución SDN Segment Routing sobre una topología del tipo Fat-Tree y comprobar su funcionamiento.
3. Proveer una metodología de experimentación eficiente que permita el análisis de los protocolos estudiados.
4. Obtener datos del plano de control sobre los protocolos BGP y RIFT para comparar los mismos con el resultado teórico obtenido.

5.1. Entorno de Trabajo

El desarrollo de nuevos protocolos y algoritmos de red, así como el análisis de comportamiento de protocolos existentes y modificaciones, requiere que se ejecuten experimentos. El uso de hardware específicamente utilizado para experimentos es costoso por lo cual generalmente el testeo se realiza sobre ambientes virtualizados. Además, los experimentos no pueden interferir con un ambiente productivo y deben ser aislados.

Por estas razones se han desarrollado distintas herramientas para testeo

que presentan al usuario un entorno virtual. Los entornos virtuales se clasifican básicamente en dos categorías, unos pueden reproducir los sistemas y su performance, mientras que otros reproducen las funcionalidades (configuraciones, arquitecturas y protocolos) pero no se tiene conocimiento sobre su performance real.

Resulta de interés principalmente probar la funcionalidad de algunos de los protocolos vistos, los cuales están disponibles para Kathará (como RIFT, BGP, OpenFabric, Segment Routing) y para Mininet (RIFT, OpenFabric y Segment Routing) [Mininet, 2018], entornos virtuales del segundo tipo.

En el caso de Kathará, se trata de un framework basado en el emulador Netkit, ampliamente utilizado para propósitos educativos y de investigación [Netkit, 2020]. Ambos comparten un mismo lenguaje de configuraciones, pero utilizan distintas tecnologías de virtualización. Netkit utiliza User-Mode Linux (UML) mientras que Kathará utiliza contenedores, específicamente Docker.
Por el lado de Mininet, se tiene un emulador flexible que utiliza código del kernel de Linux, de aplicaciones y OpenFlow, además de proveer una API para Python que permite extender protocolos y topologías.

Tanto Netkit como Mininet utilizan Linux como base y no pueden correr aplicaciones o switches no compatibles con Linux. Además se restringen a los recursos del servidor donde se encuentren instalados. Kathará en cambio con el uso de Docker, puede correr una gran variedad de sistemas operativos. La arquitectura de Kathará así como su comparación con otros emuladores se puede consultar en [Scazzariello, Ariemma y Caiazzi, 2020].

El modelo de Kathará 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.
- 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 do-

minios de colisión. Se representa mediante un directorio que contiene la configuración de la topología y de los distintos dispositivos.

Para poder distribuir Kathará en dos nodos físicos, se utilizó Megalos, una implementación distribuida de Kathará para utilizar en clústeres de potencialmente múltiples servidores físicos, diseñada por los mismos creadores de Kathará. Más en particular, Megalos es una arquitectura distribuida, escalable, y que conforma estándares de virtualización de funciones de red del ETSI[1] [Scazzariello, Ariemma, Battista y col., 2020]. Megalos soporta la implementación de dispositivos virtuales (VNFs), donde cada VNF puede tener varias interfaces de capa 2 asignadas a LANs virtuales. Utiliza contenedores de Docker para realizar VNFs y Kubernetes ["Kubernetes", 2021] para la gestión de los nodos en un cluster distribuido. Para poder distribuir Kathará con el gestor de Megalos, se utilizó un plugin CNI[2] de Kubernetes ["Megalos CNI", 2018], el cual permite crear una red overlay VXLAN[3] sobre el clúster de red de Kubernetes.

El ambiente consistió de dos servidores físicos provistos por el grupo MINA, cada uno con versiones de CentOS (7 y 8) como sistema operativo base. Utilizando máquinas virtuales KVM [KVM, 2016], se crearon 14 nodos de Kubernetes, uno de ellos tomando el rol de *master*, mientras que los 13 restantes los de *workers*. Cada uno de estos nodos tenía un sistema operativo Ubuntu 18.04 y Kubernetes instalado (el master en adición tenía Kathará instalado). El master disponía de 16 GBs de memoria RAM y 6 CPUs, mientras que los workers disponían de 4 GBs y 4 CPUs cada uno. Todos los nodos se encontraban conectados mediante una red bridge. En la figura 5.1 se puede observar la arquitectura utilizada para la experimentación.

5.2. Experimentación sobre SDN: Segment Routing

Se decidió desplegar una solución de Segment Routing en Kathará, con el propósito de comparar en un mismo entorno, un caso de algoritmo centralizado

[1] European Telecommunications Standards Institute.
[2] Container Network Interface.
[3] Virtual Extensible LAN.

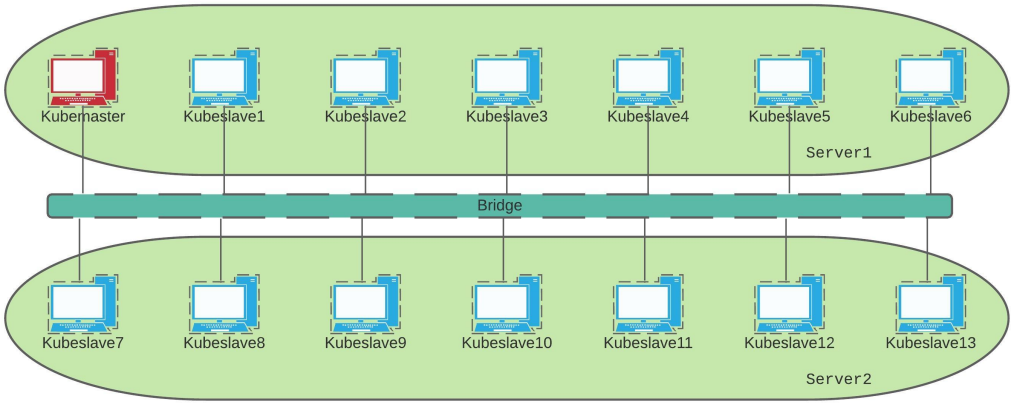


Figura 5.1: Entorno físico de experimentación

con los algoritmos distribuidos desarrollados hasta el momento, BGP y RIFT. En ["SDN para enrutamiento en el Datacenter", 2020] se realizó un despliegue de ONOS sobre una topología *Spine-Leaf*, o Fat-Tree de dos niveles en el entorno Kathará. Para ello se adaptó el script generador de topologías Fat-Tree para Kathará, diseñado por los desarrolladores del *framework*, cambiando la topología a un *Spine-Leaf* y añadiendo el controlador. Por lo cual se tomó este trabajo como base para adaptarlo a una topología Fat Tree de tres niveles. Se puede consultar en dicho trabajo sobre las configuraciones necesarias para la adaptación y configuración del entorno para dicho despliegue. A su vez se puede consultar sobre la arquitectura de la implementación de Segment Routing para ONOS en ["CORD: Leaf-Spine Fabric with Segment Routing", 2016].

Para verificar el funcionamiento de dicha solución en el entorno y la topología establecida se realizaron pruebas funcionales de la misma, en este caso de conectividad y falla de un enlace, pruebas previamente realizadas en el trabajo mencionado sobre topologías *Spine-Leaf*.
Para las pruebas se utilizó una topología Fat-Tree de múltiples planos con $k = 2$ y $R = 1$, con la idea de que fuera simple para verificar el correcto funcionamiento del algoritmo. Para ello se inició el laboratorio de Kathará, se configuró el controlador ONOS y se utilizaron herramientas como *ping* e *ip link* sobre los nodos. En la figura 5.2 se observa dicha topología en la interfaz web de ONOS.

Para probar la conectividad se ejecutó un ping entre dos servidores y se

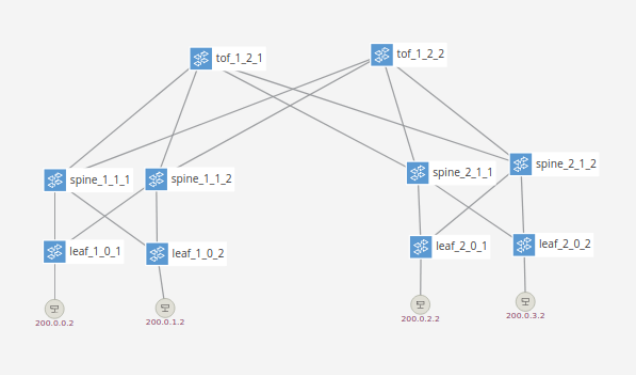


Figura 5.2: Fat-Tree generado y visualizado en interfaz de ONOS

siguió el flujo de los mensajes *echo ping request* y *echo ping reply*, observando la ruta generada por el controlador en cada nodo. Se puede ver por ejemplo en las figuras 5.3 y 5.4 como el flujo actual envía los mensajes por un grupo *Select* en el Leaf *leaf*_1_0_1, grupo de OpenFlow que permite replicar el comportamiento de ECMP conteniendo varias acciones posibles dentro de una misma regla, se puede consultar sobre el funcionamiento de estas pruebas en ["SDN para enrutamiento en el Datacenter", 2020], ya que se explican en mayor detalle las pruebas en una topología *Spine-Leaf*.

STATE	PACKETS	DURATION	FLOW PRIORITY	TABLE NAME	SELECTOR	TREATMENT	APP NAME
Added	1,994	3,891	48010	30	ETH_TYPE:ipv4, IPV4_DST:200.0.3.0/24	def[GROUP:0x70000010], transition:TABLE:60, cleared:false	*segmentrouting
Criteria: ETH_TYPE:ipv4, IPV4_DST:200.0.3.0/24							
Treatment Instructions: def[GROUP:0x70000010], transition:TABLE:60, cleared:false							

Figura 5.3: Flujos de Leaf 1 0 1

Se detectó que los mensajes entre niveles de Spine y ToF son enviados en un grupo del tipo *Indirect*, el cual no permite replicar el comportamiento de ECMP, no cumpliendo con requerimientos como el balanceo de carga y existencia de múltiples caminos en el Data Center. Como ejemplo se puede ver el flujo de mensajes de la figura 5.5 y sus correspondientes grupos en la figura 5.6.

Luego de realizar la prueba de conectividad, donde se pudo verificar el alcance entre los nodos pero se encontraron problemas en el balanceo a nivel de Spines y ToF, se experimentó con dos casos distintos de falla de un enlace. El primer caso se trata de una falla dentro de un PoD, mientras que el segundo caso contempla la caída de un enlace entre un Spine y un ToF. En teoría,

GROUP ID	APP ID	STATE	TYPE
0x70000006	org.onosproject.segmentrouting	ADDED	Select
Bytes: 0 Packets: 0 Actions: [GROUP:0x920000f3] Bytes: 0 Packets: 0 Actions: [GROUP:0x920000fd]			
0x7000000b	org.onosproject.segmentrouting	ADDED	Select
Bytes: 0 Packets: 0 Actions: [GROUP:0x920000f7] Bytes: 0 Packets: 0 Actions: [GROUP:0x920000ff]			
0x70000010	org.onosproject.segmentrouting	ADDED	Select
Bytes: 14798 Packets: 151 Actions: [GROUP:0x920000f4] Bytes: 0 Packets: 0 Actions: [GROUP:0x920000fc]			
0x920000f4	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 24696 Packets: 252 Actions: [VLAN_POP, MPLS_PUSH:mpls_unicast, MPLS_LABEL:202, GROUP:0x900000f1, VLAN_PUSH:vlan			
0x900000f1	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 27636 Packets: 282 Actions: [ETH_DST:02:00:00:ED:3D:5B, ETH_SRC:02:00:00:8A:C8:99, VLAN_ID:4094, GROUP:0xffe0001]			
0xffe0001	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 25284 Packets: 258 Actions: [VLAN_POP, OUTPUT:1]			

Figura 5.4: Grupos de Leaf 1 0 1

STATE	PACKETS	DURATION	FLOW PRIORITY	TABLE NAME	SELECTOR	TREATMENT	APP NAME
Added	327	2,150	100	24	ETH_TYPE:mpls_unicast, MPLS_LABEL:202, MPLS_BOS:true	def[DEC_MPLS_TTL, GROUP:0x95000081], transition:TABLE:60, cleared:false	*segmentrouting

Figura 5.5: Flujos de Spine 1 1 1

GROUP ID	APP ID	STATE	TYPE
0x95000081	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 44166 Packets: 433 Actions: [MPLS_LABEL:202, GROUP:0x9000007e]			
0x9000007e	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 48246 Packets: 473 Actions: [ETH_DST:02:00:00:50:B6:F5, ETH_SRC:02:00:00:ED:3D:5B, VLAN_ID:4094, GROUP:0xffe0003]			
0xffe0003	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 52224 Packets: 512 Actions: [VLAN_POP, OUTPUT:3]			

Figura 5.6: Grupos de Spine 1 1 1

debido a la topología y al funcionamiento de Segment Routing en ambos casos se debería mantener la conectividad, enviando los paquetes por otro camino.

En la figura 5.7 se visualiza la topología presentada anteriormente pero sin el enlace que conecta al Leaf $leaf\_1\_0\_1$ con el Spine $spine\_1\_1\_1$. Para simular dicha falla se ejecutó en el Leaf el comando *ip link set down eth1*. Mientras tanto se mantenía un *ping* entre los servidores. Luego de dar de baja el enlace se comprobó que la conectividad dentro del mismo PoD no se vió afectada, pero se perdió la conexión hacia los servidores en distintos PoD.

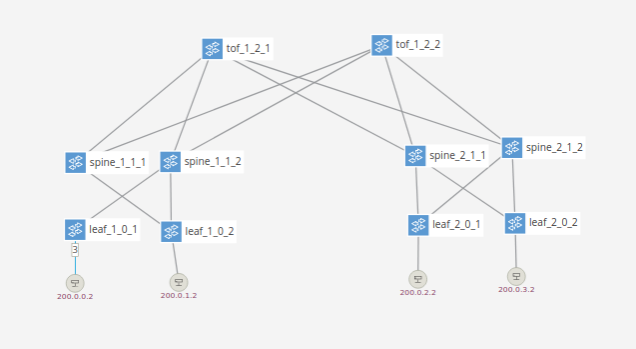


Figura 5.7: Falla de enlace dentro de un PoD

Se encontró el problema de pérdida de conexión analizando el flujo de los paquetes de respuesta *ping echo reply*. Particularmente el ToF $tof\_1\_2\_1$ no tiene un grupo que incluya a los dos Spine del primer PoD, sino que tiene una regla del tipo *Indirect* al nodo $spine\_1\_1\_1$. Debido a que el enlace entre el ToF y el Spine no falla, el segundo envía los paquetes al Spine. Sin embargo, el Spine no tiene conexión al Leaf y en el grupo *Select* no hay reglas a aplicar, lo cual se muestra en la figura 5.8. Al no tener una acción específica, los mensajes se descartan.

GROUP ID	APP ID	STATE	TYPE
0x700000cc	org.onosproject.segmentrouting	ADDED	Select
(No buckets for this group)			

Figura 5.8: Falla en Select de Spine 1 1 1

La figura 5.9 muestra un caso de falla de enlace entre un Spine y un ToF. Se ejecutó el comando *ip link set down eth3* en el Spine $spine\_1\_1\_2$, por el cual los paquetes *ping echo request* estaban siendo enviados entre los servidores. En este caso también se pierde la conectividad entre PoD distintos.

Analizando el camino de los paquetes nuevamente se encontró que en este caso el Spine que está recibiendo los paquetes, en este caso el nodo $spine\_1\_1\_2$ los procesa y envía por el enlace que tiene con el $tof\_1\_2\_1$, pero el mismo no está activo. Se debe también en este caso a que se tiene un grupo del tipo *Indirect* y no se detecta que existe otro camino, en este caso el ToF $tof\_1\_2\_2$. En las imágenes 5.10 y 5.11 se muestran el flujo y los grupos que procesan los paquetes recibidos en el nodo $spine\_1\_1\_2$.

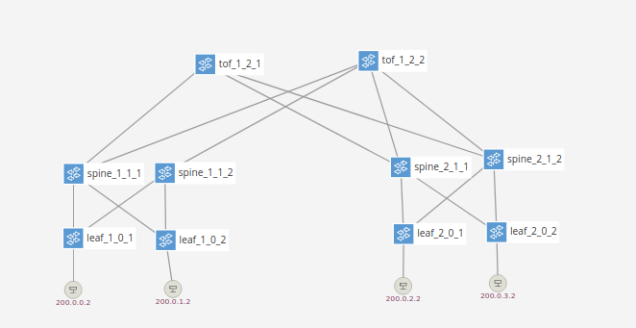


Figura 5.9: Falla de enlace entre un Spine y un ToF

STATE	PACKETS	DURATION	FLOW PRIORITY	TABLE NAME	SELECTOR	TREATMENT	APP NAME
Added	2,167	11,170	100	24	ETH_TYPE:mpls_unicast, MPLS_LABEL:202, MPLS_BOS:true	def[DEC_MPLS_TTL, GROUP:0x95000087], transition:TABLE:60, cleared:false	*segmentrouting

Figura 5.10: Flujo de Spine 1 1 2 ante falla de enlace

GROUP ID	APP ID	STATE	TYPE
0x95000087	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 223890 Packets: 2195 Actions: [MPLS_LABEL:202, GROUP:0x90000085]			
0x90000085	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 226848 Packets: 2224 Actions: [ETH_DST:02:00:00:50:B6:F5, ETH_SRC:02:00:00:81:DD:12, VLAN_ID:4094, GROUP:0xffe0003]			
0xffe0003	org.onosproject.segmentrouting	ADDED	Indirect
Bytes: 230520 Packets: 2260 Actions: [VLAN_POP, OUTPUT:3]			

Figura 5.11: Grupos de Spine 1 1 2 ante falla de enlace

Luego de investigar sobre las problemáticas detectadas y consultar a la comunidad de ONOS, se encontró que la aplicación Segment Routing fue diseñada restringida a topologías de dos niveles. Las limitantes específicas se pueden encontrar en [“Trellis - Supported Topology”, 2019]. Estas limitantes explican por qué se tiene resiliencia a nivel de cada PoD pero no entre ellos, así como el por qué no se tiene balanceo de carga a nivel de ToF.

Las limitaciones encontradas impidieron la posibilidad de comparar esta solución centralizada con los algoritmos distribuidos, por lo cual se decidió no profundizar en la misma. De todas formas, las pruebas realizadas fueron un gran aporte para comprender el funcionamiento de SDN, así como del enrutamiento basado en origen.

5.3. Enrutamiento distribuido: pruebas de Plano de Control

Se realizaron pruebas del plano de control de los protocolos RIFT y BGP sobre un Fat-Tree variando el tamaño de la topología según k, de modo de poder comparar los resultados teóricos obtenidos en el capítulo 3 con los datos procesados en los experimentos y a su vez, comparar ambos protocolos entre sí.

Tanto las herramientas utilizadas como el método de experimentación son importantes al momento de obtener y procesar datos de forma eficiente. Las herramientas consistieron en el framework Kathará como entorno virtual, el módulo *pyshark*, un *wrapper* de *tshark*, y el disector de RIFT implementado. En cuanto a la metodología, se introducirán con mayor detalle las decisiones tomadas en la siguiente sección.

5.3.1. Metodología de Experimentación

Se diseñó una metodología específica para las pruebas realizadas, valiéndose de distintas herramientas para las simulaciones, cada una con un rol independiente.
El framework pilar sobre el que se basó para realizar las simulaciones fue Kathará. El mismo funciona por un lado con contenedores Docker, que tienen todas las funcionalidades básicas del kernel de Linux, y además permitió personalizar los contenedores que utiliza cada nodo de manera de instalar otras herramientas que fueran necesarias en los mismos. Debido a la necesidad de escalar las pruebas, se utilizó el entorno Megalos del modo en que se especificó en la sección 5.1.

Utilizando Kathará, se creó una imagen en Docker donde además de tener las funcionalidades básicas del kernel de Linux, se instaló Wireshark con todas las dependencias necesarias para utilizar el disector desarrollado, de manera de poder clasificar los paquetes en el caso de las pruebas con RIFT, como se explica más adelante. En el caso de BGP, se creó una imagen de Docker que agrega la herramienta *tshark* a la imagen de FRRouting para BGP en Kathará, ya que BGP esta oficialmente soportado por Wireshark.

De esta manera, se obtuvo una manera de simular distintas topologías de manera simple utilizando Kathará, y una serie de scripts de Python creados por los mismos desarrolladores de Kathará para automatizar la generación de estas topologías, que facilitó el trabajo a la hora de generar las configuraciones necesarias para desplegarlas con algún demonio que permitiera funciones de enrutamiento como son Quagga o FRRouting.

Por otro lado, luego de tener una topología desplegada y en ejecución, era de interés poder clasificar los tipos de paquetes tanto en RIFT como en BGP, de manera de poder contar no solo la cantidad de paquetes totales sino el total de cada tipo de mensaje, y en particular de los mensajes relevantes para la generación de las tablas de enrutamiento, como son los TIE de RIFT o los UPDATE de BGP.

Para esto, en primera instancia se planteó la opción de dividir la experimentación en dos etapas distintas: una etapa de ejecución de la simulación y obtención de capturas con *tcpdump*, y otra etapa de análisis de los resultados obtenidos. Sin embargo, luego de realizar algunas pruebas de esta manera, y volviendo a evaluar la metodología en conjunto con los supervisores del proyecto, se encontró una manera más eficiente y rápida: aprovechando de mejor manera la disección de paquetes en Wireshark, se observó la posibilidad de realizar la clasificación de paquetes "en vivo" con la ayuda del módulo *pyshark* de Python, obteniendo resultados directos de cada una de las simulaciones.

Wireshark contiene una versión de línea de comandos llamada *tshark*, que permite utilizar en la misma todas las funcionalidades del mismo (inspección de paquetes, aplicación de filtros y etiquetas, estadísticas de tráfico, etc). Básicamente, *tshark* provee una especie de equivalente a *tcpdump*, pero con toda la potencia de Wireshark a la hora de depurar el contenido de los paquetes. En paralelo, existe *pyshark*, que permite realizar llamadas a cada una de estas funcionalidades de *tshark* a través de código Python, devolviendo los resultados en estructuras de datos válidas en este, y fáciles de manejar.

Utilizando las herramientas descritas anteriormente, se crearon scripts en Python que utilizaran las capacidades de filtrado de Wireshark, de manera

que cada uno de los nodos de una simulación ejecutara una instancia de este script. En estos scripts, lo que se hace es llevar a cabo el conteo en vivo de la cantidad total de paquetes, así como las cantidades acumuladas de cada tipo de paquetes. Esto fue de especial ayuda además para poder determinar en vivo la convergencia de cada uno de los protocolos en cada simulación de manera automatizada, sin la necesidad de tener que corroborarlo manualmente en los nodos. En ["rift-c-dissector", 2021] se publicó el script *sniffer-rift-bootstrap.py* en el cual se presenta dicho conteo para el caso de RIFT. El script para BGP es similar con sus respectivos mensajes.

5.3.1.1. Convergencia y criterio de parada

Es importante remarcar que los protocolos de enrutamiento analizados convergen pero no finalizan de manera explícita, esto es, no existe una cantidad exacta de mensajes para el cual se asuma que el protocolo convergió. Los protocolos necesitan de al menos cierta cantidad de mensajes, como se observa en el análisis teórico, pero pueden enviarse más mensajes por pérdida de mensajes, por delay, o simplemente porque el protocolo especifique que se deban reenviar mensajes periódicamente (no es el caso de TIEs y UPDATEs). En los datos obtenidos de los experimentos se puede observar este comportamiento, debido a que en cada instancia de un experimento para cierta topología se observan leves variaciones en los mensajes totales.

Se debió entonces definir un criterio de parada para poder determinar la finalización de cada experimento en la metodología desarrollada, el cual se acordó fuera una "ventana de tiempo", pero medida en base a la cantidad de mensajes enviados por cada protocolo. De esta manera se aseguró que el criterio fuera más representativo del concepto de "tiempo transcurrido desde el último mensaje" y no dependiera de los recursos físicos utilizados. Arbitrariamente se consideró que para cada protocolo, si luego de haberse observado el último TIE/UPDATE, se observaban al menos 1000 mensajes de otros tipos, se podía asumir la convergencia del protocolo en cada nodo.

5.3.2. Pruebas Realizadas

Utilizando la metodología presentada, se diseñó un plan de pruebas para ambos protocolos. Se especifican a continuación los casos de uso que se proba-

ron así como las topologías con las cuales se realizaron los experimentos.
En cuanto a los casos de uso, es de especial interés ejecutar el *bootstrap* de ambos protocolos, de forma de comparar la cantidad de mensajes TIE y UPDATE con los resultados teóricos de RIFT y BGP respectivamente. También es de interés poder comparar la mensajería total de ambos protocolos. El mismo consiste en:

1. Iniciar la fábrica
2. Para cada nodo contar los mensajes, diferenciados por tipos, hasta que el protocolo converge.
3. Detener la fábrica.

Se probó el evento sobre las topologías especificadas en la tabla 5.1. Se repitió el mismo cinco veces para cada topología con el objetivo de tener un muestreo estadístico de las ejecuciones. Esto se realizó para ambos protocolos y los resultados fueron posteriormente analizados para su comparación y representación gráfica.

Topología	k_leaf	k_leaf	R	N	P	Switches	Leafs	Spines	ToFs	Nodos
Fat Tree (K=4, R=1)	2	2	1	2	4	20	8	8	4	28
Fat Tree (K = 6, R = 1)	3	3	1	3	6	45	18	18	9	63
Fat Tree (K = 8, R = 1)	4	4	1	4	8	80	32	32	16	112
Fat Tree (K = 10, R = 1)	5	5	1	5	10	125	50	50	25	175
Fat Tree (K = 12, R = 1)	6	6	1	6	12	180	72	72	36	252

Tabla 5.1: Descripción de topologías utilizadas para *bootstrap*

5.3.3. Resultados Obtenidos

Luego de las pruebas realizadas se obtuvieron archivos de valores separados por comas o *csv* conteniendo para cada topología, evento y protocolo testeado, la cantidad de mensajes clasificados según el protocolo correspondiente. Debido a que cada experimento se repitió cinco veces, se buscó tener la media de cantidad de mensajes por tipo en cada uno de los mismos.

Se generaron scripts de Python utilizando las bibliotecas *pandas* y *matplotlib*, las cuales son comúnmente utilizadas para el análisis de datos y generación de gráficas. Para cada experimento resultó de interés comparar la

BGP	Open	% desviacion	Update	% desviacion	Notification	Desviacion	Keepalive	% desviacion	Total
k = 4, R = 1	256.0	20.3	563.7	13.6	130.4	48.8	10073.8	1,5	11023
k = 6, R = 1	426.8	4	3625.8	7.3	0.8	1.7	24688.6	0.8	28742
k = 8, R = 1	1031.2	2.5	13658.8	26.2	1.6	2.1	48301.8	0.07	62993
k = 10, R = 1	2009.4	1	39271.8	17.1	3.6	3.3	89915	10	131199
k = 12, R = 1	3425.8	1.7	97836.6	16.3	10.8	6	168928.0	14.8	270201
RIFT	LIE	% desviacion	TIDE	% desviacion	TIRE	% desviacion	TIE	% desviacion	Total
k = 4, R = 1	15054.3	4.5	6689.3	2.2	666.9	15.5	711.1	16	23121
k = 6, R = 1	40179.6	12,9	17637.2	7.8	3907.4	19.9	4825.9	16.4	66550

Tabla 5.2: Tabla de resultados exactos de bootstrap

cantidad total de mensajes entre los dos protocolos, la cantidad separando por tipos, y particularmente entre los tipos TIE y UPDATE. Se presentan los resultados para el evento de *bootstrap* junto con un análisis de los mismos y luego la comparación con los resultados teóricos calculados previamente.

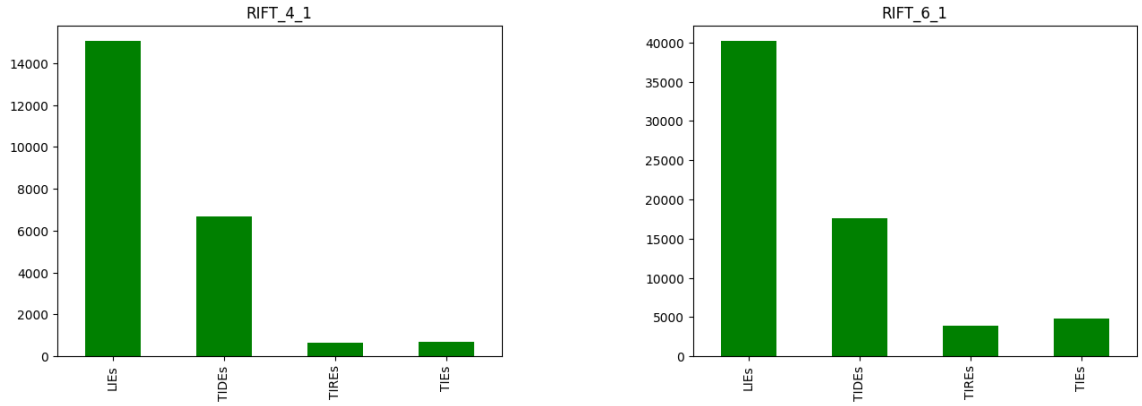
5.3.3.1. Bootstrap

En la tabla 5.2 se muestran los resultados obtenidos por tipo de mensajes, sus correspondientes desviaciones estándar y total de mensajes por experimento. Como se puede observar, se realizaron todos los experimentos planificados para el caso de BGP, sin embargo se obtuvieron solo los resultados de dos experimentos para RIFT. Esto se debe a que se observó inestabilidad en la implementación rift-python al escalar a topologías donde $k \geq 8$, como desagregaciones de todos los prefijos a nivel de los ToF en el *bootstrap*, sin llegar a observarse la convergencia del protocolo.

Se calculó la desviación estándar y su porcentaje en cada caso para verificar la diferencia entre los distintos experimentos. Inicialmente se generaron diez corridas para los dos experimentos de $k = 4$ y para RIFT en $k = 6$, sin embargo debido a que cuanto mayor era la topología mayores los tiempos de ejecución del entorno, y debido a que se disponía de un tiempo acotado de uso de dichos recursos al utilizar una infraestructura compartida para otros propósitos, se realizaron cinco ejecuciones para cada caso de BGP de $k = 6$ en adelante. Al observar que la desviación estándar se mantiene por debajo del 30 % en todos los casos y por debajo del 20 % en la mayoría de los mismos, se consideró viable mantener la cantidad de experimentos ejecutados en cinco y tomar la media como valor representativo.

En la figura 5.12 se puede ver una gráfica comparativa por tipos de mensajes de RIFT, donde claramente se nota que los mensajes LIE y TIDE son los

más abundantes y crecen en gran magnitud a medida que crece k. El protocolo necesita de los mensajes TIE para converger y los mismos son los de menor magnitud, por lo que se puede afirmar que el protocolo es ruidoso.



(a) k=4, R=1 **(b)** k=6, R=1

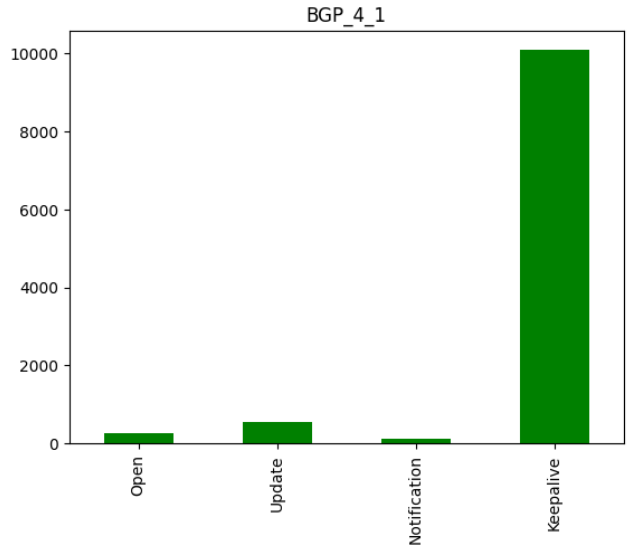
Figura 5.12: Conteo de mensajes por tipo para RIFT

Se observan en la figura 5.13 los experimentos realizados para BGP. En este caso se puede ver una gran cantidad de mensajes KEEPALIVE, lo cual es de esperar debido a que se envían cada cierto tiempo para comprobar la conexión. Por otro lado, se puede notar el crecimiento de los mensajes UPDATE a medida que crece la topología. Podemos decir que BGP es menos ruidoso, ya que los mensajes KEEPALIVE son mensajes simples y de menor tamaño en comparación a los mensajes LIE o TIDE de RIFT.

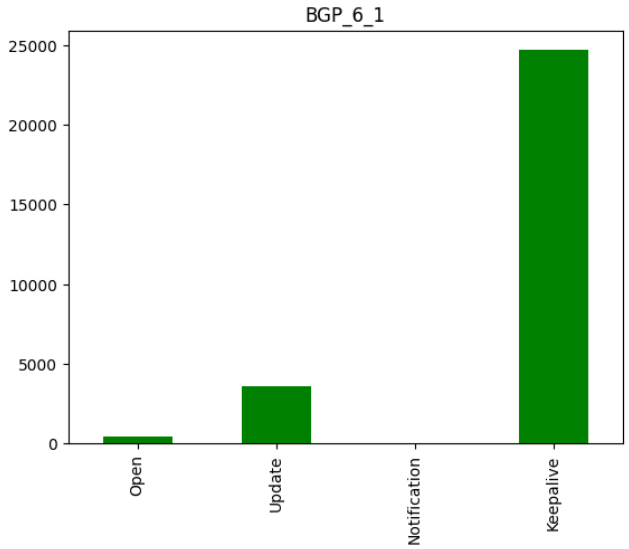
La diferencia entre los distintos tipos de mensaje resulta en que al comparar los totales entre BGP y RIFT, el segundo tenga una mayor cantidad. Dicha comparación se visualiza en la gráfica de puntos de la figura 5.14, donde se representa el total de mensajes de cada protocolo.

Por último, el interés de las pruebas realizadas se centra además de la comparación entre los dos protocolos, en la validación entre los resultados experimentales y los resultados teóricos analizados, que representan una cota inferior de mensajes. En la figura 5.15 se puede apreciar una gráfica de puntos en donde se comparan los mensajes de convergencia de cada protocolo, TIE o UPDATE según corresponda, con las cotas inferiores calculadas.

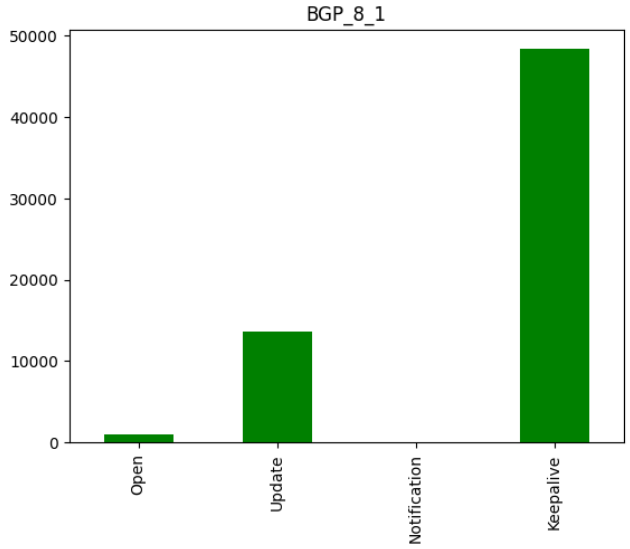
En el caso de RIFT, los dos casos que fue posible obtener su resultado coinciden de forma casi exacta, lo cual es un buen indicio, si bien no se pudo comprobar con topologías de mayor tamaño. En el caso de BGP, se puede observar que para topologías más chicas los resultados experimentales y teóricos son muy cercanos, cuanto más crece k, los mismos se alejan pero manteniendo el orden exponencial y con el resultado teórico esperado por debajo del resultado experimental obtenido, lo cual valida dicho análisis al tratarse de una cota inferior.



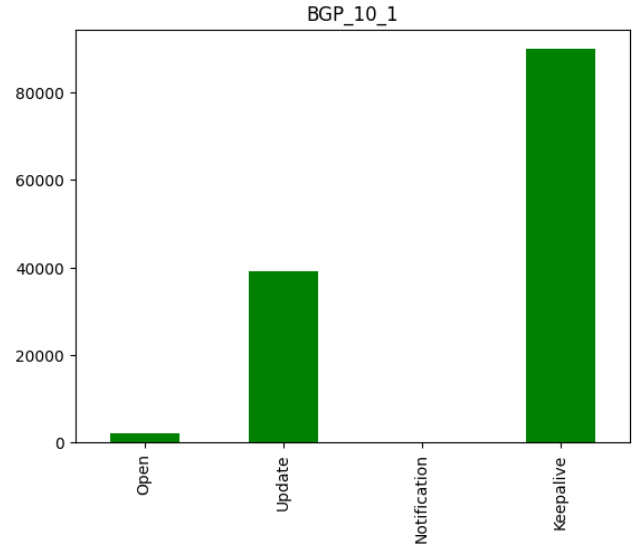
(a) k=4, R=1



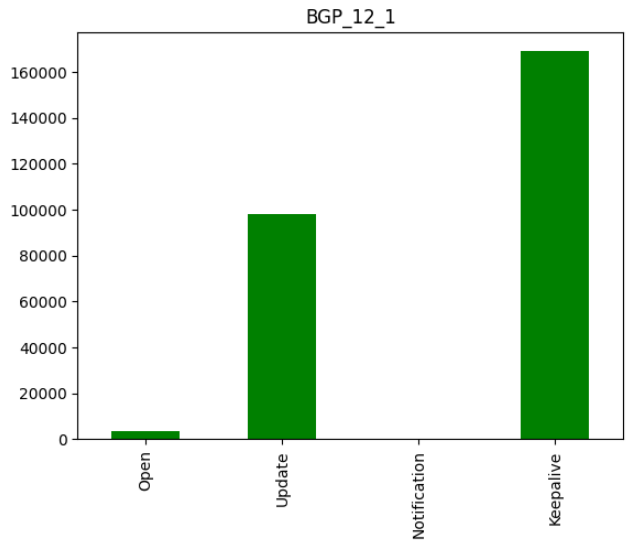
(b) k=6, R=1



(c) k=8, R=1



(d) k=10, R=1



(e) k=12, R=1

Figura 5.13: Conteo de mensajes por tipo para BGP

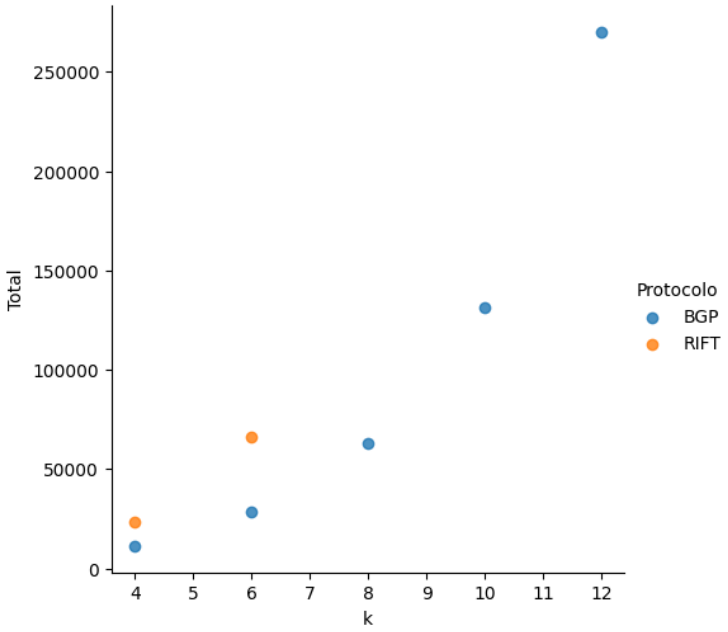


Figura 5.14: Comparación de mensajes totales entre RIFT y BGP

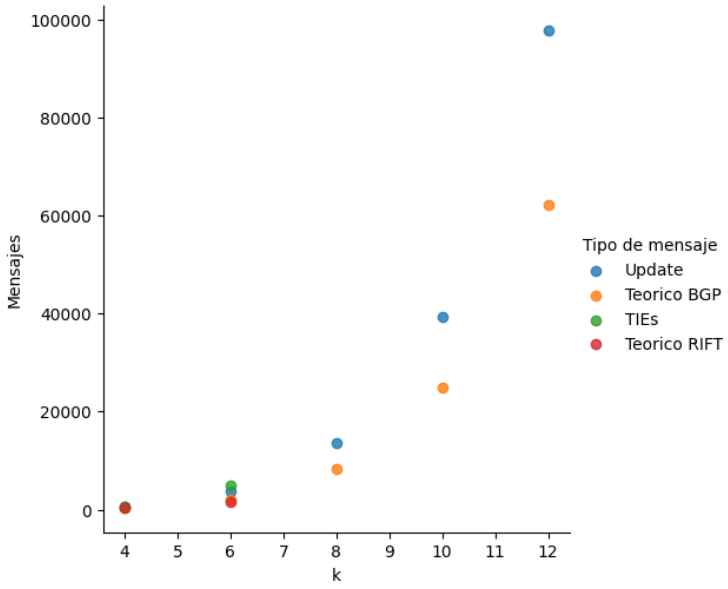


Figura 5.15: Comparación de convergencia teórica y experimentos

Capítulo 6

Conclusiones y trabajo a futuro

En el transcurso del proyecto, se realizó un estudio completo de los elementos y conceptos más relevantes en lo que refiere al enrutamiento dentro de un Data Center masivo. Se llevó a cabo un estudio del estado del arte, donde se relevaron:

- Los **requerimientos básicos** que existen para un el despliegue y correcto funcionamiento de un Data Center.
- Las **topologías** más comunes que se utilizan para desplegarlos.
- Los dos **paradigmas** existentes de enrutamiento (distribuido y centralizado), contextualizados a este tipo de topologías, y qué **protocolos** resuelven el enrutamiento en el Data Center.
- Cómo se resuelve el **forwarding** de paquetes, y en particular el modelo ECMP.
- El problema de la **gestión** dentro de un Data Center, en particular el modelo de gestión en el Cloud.

Luego de lograr construirse un panorama general teórico de los puntos mencionados, se decidieron realizar experimentos, acotando el universo de trabajo al enrutamiento en topologías Fat-Tree en particular, debido a ser las más utilizadas en la práctica en el Data Center. Investigando las implementaciones existentes, y teniendo en cuenta el alcance deseado del proyecto, se centró el trabajo en protocolos distribuidos, dado que son los protocolos más avanzados y de mayor utilización en la actualidad. En particular, se estudiaron los protocolos BGP y RIFT. Por otro lado, fue de interés poder probar alguna solución centralizada de manera de poder comparar con las soluciones distribuidas, por lo que se obtuvo una implementación de ONOS para el protocolo centralizado

Segment Routing, y se realizaron pruebas principalmente de funcionamiento del mismo.

Interesó realizar inicialmente un análisis funcional y cuantitativo del rendimiento de cada una de las implementaciones seleccionadas. Para ello, primero se realizó un análisis teórico de BGP y RIFT, donde se estimó una cota inferior para la cantidad de mensajes que cada uno envía, bajo ciertas hipótesis, y en particular para el *bootstrap* de una topología. Para RIFT, se realizó un análisis de la etapa de envío de mensajes LIE, así como la mensajes TIE, aprovechando la especifidad de la documentación oficial del protocolo para cada uno de estos, mientras que en BGP se focalizó el estudio en los mensajes UPDATE. Se obtuvieron cotas inferiores de $\Theta(k^3)$ y $\Theta(k^4)$ para LIEs y TIEs respectivamente, mientras que para los UPDATES de BGP se obtuvo una cota inferior de $\Theta(k^5)$ mensajes para la convergencia.

El siguiente paso a seguir fue corroborar las conclusiones obtenidas del estudio teórico experimentalmente. Se dispuso para esto de una infraestructura brindada por los supervisores, donde se desplegó un ambiente de trabajo basado en la herramienta de virtualización Kathará, y en particular en su versión distribuida Megalos, para realizar las pruebas correspondientes del plano de control de las soluciones distribuidas de BGP y RIFT. Al momento de diseñar las pruebas, se notó la necesidad de poder inspeccionar los paquetes para poder obtener resultados cuantitativos. Mientras que esto se pudo solucionar fácilmente utilizando Wireshark para el caso de BGP, en el caso de RIFT, se vió la necesidad de implementar un disector propio. Basándose en la especificación del protocolo en Thrift, y la documentación correspondiente de Wireshark para el desarrollo de un disector como plugin en C, se logró implementar exitosamente un disector de RIFT, el cual se utilizó en la etapa de experimentación posterior.

Además, se presentó el disector en conjunto con los supervisores del proyecto como parte de un artículo científico bajo el título *Routing in Fat Trees: a protocol analyzer for debug and experimentation*. Dicho artículo fue aceptado para su presentación en la conferencia "**IFIP/IEEE International Symposium of Integrated Network Management (IM) 2021**"[“IM 2021”, 2021] que se realizará en mayo de 2021; la serie de conferencias IM es, en conjunto

con la serie “Network Operations and Management Symposium (NOMS)”, la más prestigiosa en la comunidad académica de gestión de red. Por otro lado, el trabajo del disector también fue publicado en **Day One: Routing in Fat Trees (RIFT)** por parte de la empresa Juniper Networks [Aelmans y col., 2020], en el marco de un libro completo referente al protocolo RIFT.

Finalmente, se llevaron a cabo las simulaciones en el ambiente desplegado con Megalos, probando topologías Fat-Tree y escalando las dimensiones de dichas topologías para cada protocolo. Se diseñó una metodología específica, desarrollando scripts en Python que aprovecharan los disectores disponibles de BGP y RIFT, de manera de poder contabilizar paquetes “en vivo” y obtener resultados cuantitativos. En el caso de RIFT, los resultados obtenidos fueron coherentes con el análisis teórico realizado, aunque existieron problemas en topologías con valores de $k \geq 8$ en la implementación probada. Para BGP, se logró probar topologías hasta $k = 12$ inclusive, obteniendo resultados coherentes completamente con el análisis teórico.

Particularmente, el desarrollo de un análisis teórico del *bootstrap* de los dos protocolos estudiados, genera un aporte fundamental para el mayor entendimiento de los mismos, lo cual puede ser de ayuda para la comunidad científica. Por otro lado, la generación de un método de experimentación eficiente a partir de la disección de paquetes, presenta una herramienta de gran utilidad, la cual se puede adaptar para el estudio de protocolos nuevos como existentes. Estos desafíos fueron, en gran medida, los resultados que más se destacan del presente trabajo.

En paralelo a estas pruebas, se realizaron pruebas funcionales de la versión de Segment Routing de ONOS, utilizando el mismo ambiente físico, y una máquina virtual específica para desplegar el controlador. Se intentaron repetir las pruebas realizadas en [“SDN para enrutamiento en el Datacenter”, 2020], las cuales fueron realizadas en una topología Spine-Leaf, pero esta vez en una topología Fat-Tree. Mientras que se corroboró un funcionamiento correcto de la implementación al momento de realizar el bootstrap de una topología, se observó un comportamiento inesperado al dar de baja enlaces, perdiéndose la conectividad entre nodos en algunos casos. Investigando la documentación oficial y la información disponible en los foros de la comunidad de ONOS, se

llegó a la conclusión de que el problema era inherente a la implementación, que no aseguraba soporte para topologías Fat-Tree.

Algunos puntos que se podrían considerar relevantes como trabajo a futuro, y que quedaron por fuera del alcance del proyecto son:

- Profundizar experimentalmente en investigar soluciones e implementaciones existentes tanto en el área de **forwarding** de paquetes, así como en la **gestión** del Data Center, conceptos que se desarrollaron en la relevación de antecedentes pero quedaron por fuera de la etapa de experimentación.
- Corroborar los resultados obtenidos en el análisis teórico de las soluciones distribuidas con algún simulador de eventos.
- Adaptar el controlador ONOS, de código abierto, para obtener compatibilidad con las topologías Fat-Tree y comparar la solución con los protocolos distribuidos estudiados.
- Encontrar una forma de ejecutar los experimentos de RIFT para $k \geq 8$, sea utilizando otra implementación como la de Juniper, o encontrando los problemas que generan inestabilidad en rift-python al crecer la topología. De forma de poder validar el resultado teórico con topologías más grandes.
- Investigar la performance de los BGP y RIFT en cuanto a cantidad de mensajes en otros casos, como puede ser el funcionamiento normal (ausencia de eventos), así como la falla y recuperación de dispositivos y/o enlaces en una topología, además de investigar más la definiciíon de los protocolos de manera de poder realizar algún tipo de análisis previo a los experimentos, como sí se pudo hacer en el bootstrap.
- Se centró la experimentación en BGP y RIFT por un tema de alcance y material disponible de cada uno de ellos, aunque sería de interés expandir el universo de estudio a otros protocolos de enrutamiento (OpenFabric, LSVR, entre otros).

Referencias

¿Qué es un Data Center y cuál es su importancia? [Online: Accedido 13/3/2021]. (2020). https://www.optical.pe/blog/que-es-un-data-center-y-cual-es-su-importancia/

Aelmans, M., Vandezande, O., Rijsman, B., Head, J., Graf, C., Alberro, L., Mali, H. & Steudler, O. (2020). *Day One: Routing in Fat Trees (RIFT)*. Juniper Networks Books.

Agarwal, A., Slee, M. & Kwiatkowski, M. (2007). *Thrift: Scalable Cross-Language Services Implementation* (inf. téc.). Facebook. http://thrift.apache.org/static/files/thrift-20070401.pdf

Basic Configuration Concepts [Online: Accedido 18/12/2020]. (2018). https://docs.opendaylight.org/projects/bgpcep/en/latest/bgp/bgp-user-guide-config-concepts.html

Blanton, E., Paxson, D. V. & Allman, M. (2009). TCP Congestion Control. RFC Editor. https://doi.org/10.17487/RFC5681

Bonofiglio, G., Iovinella, V., Lospoto, G. & Di Battista, G. (2018). Kathará: A container-based framework for implementing network function virtualization and software defined networks, En *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. https://doi.org/10.1109/NOMS.2018.8406267

Brain-Slug: a BGP-Only SDN for Large-Scale Data-Centers [Online: Accedido 30/1/2021]. (2013). https://archive.nanog.org/sites/default/files/wed.general.brainslug.lapukhov.20.pdf

Build SDN Agilely [Online: Accedido 19/1/2021]. (2017). https://ryu-sdn.org/

Caiazzi, T. (2019). *Software Defined Data Centers: methods and tools for routing protocol verification and comparison*. UNIVERSITÀ DEGLI STUDI DI ROMA TRE.

Chapman, C. (2016). Chapter 7 - Using Wireshark and TCP dump to visualize traffic (C. Chapman, Ed.). En C. Chapman (Ed.), *Network*

Performance and Security. Boston, Syngress. https://doi.org/https://doi.org/10.1016/B978-0-12-803584-9.00007-X

Clausen, T. H., Jacquet, P., Adjih, C., Laouiti, A., Minet, P., Muhlethaler, P., Qayyum, A. & Viennot, L. (2003). Optimized link state routing protocol (OLSR).

Clos, C. (1953). A study of non-blocking switching networks. *The Bell System Technical Journal*, *32*(2), 406-424. https://doi.org/10.1002/j.1538-7305.1953.tb01433.x

CORD: Leaf-Spine Fabric with Segment Routing [Online: Accedido 1/1/2021]. (2016). https://wiki.onosproject.org/display/ONOS/CORD%5C%3A+Leaf-Spine+Fabric+with+Segment+Routing

Dutt, D. G. (2017). *BGP in the Data Center* (1.ª ed.). Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., O'Reilly Media, Inc.,

encoding.thrift. (2020). https://github.com/brunorijsman/rift-python/blob/master/rift/encoding.thrift

Filsfils, C., Previdi, S., Ginsberg, L., Decraene, B., Litkowski, S. & Shakir, R. (2018). Segment Routing Architecture. RFC Editor. https://doi.org/10.17487/RFC8402

Ginsberg, L., Previdi, S. & Chen, M. (2016). IS-IS Extensions for Advertising Router Information. RFC Editor. https://doi.org/10.17487/RFC7981

Greenberg, A., Hamilton, J., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D., Patel, P. & Sengupta, S. (2009). VL2: A Scalable and Flexible Data Center Network. *Communications of the ACM*, *54*, 95-104. https://doi.org/10.1145/1897852.1897877

Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y. & Lu, S. (2009). BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. *SIGCOMM Comput. Commun. Rev.*, *39*(4), 63-74. https://doi.org/10.1145/1594977.1592577

Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y. & Lu, S. (2008). DCell: A scalable and fault-tolerant network structure for data centers. https://doi.org/10.1145/1402958.1402968

Hopps, C. (2000). *RFC2992: Analysis of an Equal-Cost Multi-Path Algorithm*. USA, RFC Editor.

IM 2021 [Online: Accedido 25/1/2021]. (2021). https://im2021.ieee-im.org/

InfiniBand in the Enterprise Data Center [Online: Accedido 29/1/2021]. (2006). https://www.mellanox.com/pdf/whitepapers/InfiniBand__EDS.pdf

Kubernetes [Online: Accedido 24/2/2021]. (2021). https://kubernetes.io/es/

KVM. (2016). Main Page — KVM [[Online; accessed 24-February-2021]]. https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792

Landscape of the Data Center Industry [Online: Accedido 16/3/2021]. (2017). https://www.uschamber.com/sites/default/files/ctec_datacenterrpt_lowres.pdf

Lapukhov, P., Premji, A. & Mitchell, J. (2016). Use of BGP for Routing in Large-Scale Data Centers. RFC Editor. https://doi.org/10.17487/RFC7938

Leiserson, C. E. (1985). Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, *C-34*(10), 892-901. https://doi.org/10.1109/TC.1985.6312192

Litmanen, I. (2017). *Segment Routing*. Helsinki Metropolia University of Applied Sciences.

Medhi, D. & Ramasamy, K. (2018). Chapter 12 - Routing and Traffic Engineering in Data Center Networks (D. Medhi & K. Ramasamy, Eds.; Second Edition). En D. Medhi & K. Ramasamy (Eds.), *Network Routing (Second Edition)* (Second Edition). Boston, Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-0-12-800737-2.00014-4

Megalos CNI [Online: Accedido 24/2/2021]. (2018). https://github.com/KatharaFramework/Megalos-CNI

Mininet. (2018). Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet [[Online: accedido 25/2/2021]]. http://mininet.org/

Netkit. (2020). Netkit Official Site [[Online: accedido 25/2/2021]]. https://www.netkit.org/

ONOS Open Network Operating System [Online: Accedido 19/1/2021]. (2021). https://opennetworking.org/onos/

Open/R: Open routing for modern networks [Online: Accedido 23/1/2021]. (2017). https://engineering.fb.com/2017/11/15/connectivity/open-r-open-routing-for-modern-networks/

OpenDayLight [Online: Accedido 19/1/2021]. (2018). https://www.opendaylight.org/

OpenDaylight Pathman SR App [Online: Accedido 1/1/2021]. (2018). https://github.com/CiscoDevNet/pathman-sr#:~:text=OpenDaylight%20%28ODL%29%20is%20an%20open-source%20application%20development%20and,for%20forwarding%20packets%20across%20MPLS%20or%20IPv6%20networks.

OpenFlow - Open Networking Foundation [Online: Accedido 06/03/2021]. (2017). https://web.archive.org/web/20170314210531mp_/https://www.opennetworking.org/en/sdn-resources/openflow

Openstack [Online: Accedido 24/1/2021]. (2021). https://www.openstack.org/

Orebaugh, A., Ramirez, G., Beale, J. & Wright, J. (2007). *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing.

Patel, K., Lindem, A., Zandi, S. & Henderickx, W. (2020). *Shortest Path Routing Extensions for BGP Protocol* (Internet-Draft draft-ietf-lsvr-bgp-spf-11) [Work in Progress]. Internet Engineering Task Force. Work in Progress. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-lsvr-bgp-spf-11

Per-Flow and Per-Packet Load Balancing [Online: Accedido 4/2/2021]. (2021).

Pries, R., Jarschel, M., Schlosser, D., Klopf, M. & Tran-Gia, P. (2012). Power Consumption Analysis of Data Center Architectures. https://doi.org/10.1007/978-3-642-33368-2_10

Przygienda, T., Sharma, A., Thubert, P., Rijsman, B. & Afanasiev, D. (2020). *RIFT: Routing in Fat Trees* (Internet-Draft draft-ietf-rift-rift-12) [Work in Progress]. Internet Engineering Task Force. Work in Progress. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-rift-rift-12

Qin, Y., Li, Z., Wei, F., Dong, J. & Li, T. (2011). Purge Originator Identification TLV for IS-IS. RFC Editor. https://doi.org/10.17487/RFC6232

Quiroz Martiña, D. J. (2015). *Research on path establishment methods performance in SDN-based networks* (Tesis doctoral). UPC, Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels. http://hdl.handle.net/2117/80972

Rekhter, Y., Hares, S. & Li, T. (2006). A Border Gateway Protocol 4 (BGP-4). RFC Editor. https://doi.org/10.17487/RFC4271

rift-c-dissector. (2021). https://gitlab.com/fing-mina/datacenters/rift-dissector/-/tree/master/rift-c-dissector

Routing in fat trees [Online: Accedido 13/3/2021]. (2019). https://www.fing.edu.uy/owncloud/index.php/s/20cnCEViZxyrQHi

Routing in Fat Trees implementation in Python [Online: Accedido 31/1/2021]. (2018). https://github.com/brunorijsman/rift-python

Santoro, N. (2006). *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. USA, Wiley-Interscience.

Scazzariello, M., Ariemma, L., Battista, G. D. & Patrignani, M. (2020). Megalos: A Scalable Architecture for the Virtualization of Network Scenarios, En *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. https://doi.org/10.1109/NOMS47738.2020.9110288

Scazzariello, M., Ariemma, L. & Caiazzi, T. (2020). Kathará: A Lightweight Network Emulation System, En *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. https://doi.org/10.1109/NOMS47738.2020.9110351

SDN para enrutamiento en el Datacenter [Online: Accedido 13/3/2021]. (2020). https://www.fing.edu.uy/owncloud/index.php/s/p9IMEmD18lfCoOR

Shen, N. & McPherson, D. R. (2008). Dynamic Hostname Exchange Mechanism for IS-IS. RFC Editor. https://doi.org/10.17487/RFC5301

Sikos, L. F. (2020). Packet analysis for network forensics: A comprehensive survey. *Forensic Science International: Digital Investigation*, *32*, 200892. https://doi.org/https://doi.org/10.1016/j.fsidi.2019.200892

Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Liu, H., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S. & Vahdat, A. (2016). Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. *Commun. ACM*, *59*(9), 88-97. https://doi.org/10.1145/2975159

Singla, A., Hong, C.-Y., Popa, L. & Godfrey, P. (2011). Jellyfish: Networking Data Centers Randomly.

Son, J. & Buyya, R. (2018a). A Taxonomy of Software-Defined Networking (SDN)-Enabled Cloud Computing. *ACM Computing Surveys (CSUR)*, *51*, 1-36.

Son, J. & Buyya, R. (2018b). A Taxonomy of Software-Defined Networking (SDN)-Enabled Cloud Computing. *ACM Comput. Surv.*, *51*(3). https://doi.org/10.1145/3190617

srv6-sdn: An SDN ecosystem for SRv6 on Linux [Online: Accedido 7/1/2021]. (2020). https://netgroup.github.io/srv6-sdn/

TCPDUMP/LIBPCAP public repository. (2020, 2 de mayo). Recuperado el 23 de mayo de 2020, desde http://www.tcpdump.org/

Thaler, D. & Hopps, C. (2000). *RFC2991: Multipath Issues in Unicast and Multicast Next-Hop Selection*. USA, RFC Editor.

Thrift Binary Protocol [Online: Accedido 15/3/2021]. (2020). https://github.com/apache/thrift/blob/master/doc/specs/thrift-binary-protocol.md

Trellis [Online: Accedido 23/1/2021]. (2021). https://opennetworking.org/trellis/

Trellis - Production-ready Multi-purpose leaf-spine fabric [Online: Accedido 23/1/2021]. (2021). https://opennetworking.org/reference-designs/trellis/

Trellis - Supported Topology [Online: Accedido 1/1/2021]. (2019). https://docs.trellisfabric.org/1.12/supported-topology.html

Truong, N., Lee, G. M. & Ghamri-Doudane, Y. (2015). Software defined networking-based vehicular Adhoc Network with Fog Computing. *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*, 1202-1207. https://doi.org/10.1109/INM.2015.7140467

Tüxen, M., Risso, F., Bongertz, J., Combs, G., Harris, G. & Richardson, M. (2020). *PCAP Next Generation (pcapng) Capture File Format* (Internet-Draft draft-tuexen-opsawg-pcapng-01) [Work in Progress]. Internet Engineering Task Force. Work in Progress. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-tuexen-opsawg-pcapng-01

Using the POX SDN controller [Online: Accedido 19/1/2015]. (2017). http://www.brianlinkletter.com/using-the-pox-sdn-controller/

VMWare Cloud Director. (2021). https://www.vmware.com/latam/products/cloud-director.html

What is a Data Center? [Online: Accedido 13/3/2021]. (2021). https://searchdatacenter.techtarget.com/definition/data-center

White, R. & Zandi, S. (2018). *IS-IS Support for Openfabric* (Internet-Draft draft-white-openfabric-07) [Work in Progress]. Internet Engineering Task Force. Work in Progress. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-white-openfabric-07

wireshark/wireshark: Read-only mirror of Wireshark's Git repository at https://gitlab.com/wireshark/wireshark. [Online: Accedido 14/3/2021]. (2021). https://github.com/wireshark/wireshark/blob/master/doc/README.dissector

Zhang-Shen, R. & McKeown, N. (2008). Designing a Fault-Tolerant Network Using Valiant Load-Balancing, En *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*. https://doi.org/10.1109/INFOCOM.2008.305