UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Control de tráfico en el data center

Un enfoque experimental

Leonardo Alberro

Programa de Posgrado de Maestría en Informática
Facultad de Ingeniería
Universidad de la República - PEDECIBA

Montevideo – Uruguay
Enero de 2023

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Control de tráfico en el data center

Un enfoque experimental

Leonardo Alberro

Tesis de Maestría presentada al Programa de Posgrado de Maestría en Informática, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magíster en Maestría en Informática.

Directores:
Dr. Prof. Eduardo Grampín
Dr. Prof. Alberto Castro

Director académico:
Dr. Prof. Eduardo Grampín

Montevideo – Uruguay
Enero de 2023

Alberro, Leonardo

Control de tráfico en el data center / Leonardo Alberro. - Montevideo: Universidad de la República, PEDECIBA, Facultad de Ingeniería, 2023.

XII, 88 p. 29, 7cm.

Directores:

Eduardo Grampín

Alberto Castro

Director académico:

Eduardo Grampín

Tesis de Maestría – Universidad de la República, Programa de Maestría en Informática, 2023.

Referencias bibliográficas: p. 78 – 88.

1. data center, 2. fat-tree, 3. emulación, 4. enrutamiento, 5. BGP. I. Grampín, Eduardo, Castro, Alberto, . II. Universidad de la República, Programa de Posgrado de Maestría en Informática. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

_____

D.Sc. Prof. Nombre del 1er Examinador Apellido

_____

Ph.D. Prof. Nombre del 2do Examinador Apellido

_____

D.Sc. Prof. Nombre del 3er Examinador Apellido

_____

Ph.D. Prof. Nombre del 4to Examinador Apellido

_____

Ph.D. Prof. Nombre del 5to Examinador Apellido

Montevideo – Uruguay
Enero de 2023

Agradecimientos

En primer lugar quisiera agradecer a todos los que de una u otra manera me brindaron apoyo académico durante el desarrollo de esta Maestría. En especial a mis tutores Eduardo Grampín y Alberto Castro, y a todos los integrantes del grupo de investigación MINA de la Universidad de la República.

A la Agencia Nacional de Investigación e Innovación por el apoyo financiero para poder llevar a cabo este trabajo. A la Universidad de la República y al Programa de Desarrollo de las Ciencias Básicas (PEDECIBA) por permitirme seguir este camino.

Finalmente, a mis afectos: amigas, amigos, compañera, madre y un largo etcétera, por el apoyo que siempre me dan y por siempre acompañarme en mis decisiones.

RESUMEN

Los centros de datos de escala masiva (MSDC, por sus siglas en inglés) se han convertido en un componente clave de la arquitectura de Internet. Con escalas de hasta cientos de miles de servidores, transportar tráfico dentro de estas infraestructuras requiere recursos de conectividad mucho mayores que en las redes de tránsito tradicionales de Internet. Usualmente, los MSCD utilizan topologías de tipo fat-tree, que brindan diversidad de caminos (o rutas múltiples) y un ancho de banda de bisección constante entre servidores. Para explotar las potenciales ventajas de estas topologías, se necesitan protocolos de enrutamiento específicos, con soporte para rutas múltiples y con baja carga de mensajería del plano de control. Estas infraestructuras tienen un alto costo, y por tanto, no es posible utilizarlas para experimentar con nuevos protocolos. Por esta razón, se necesitan entornos de emulación/simulación escalables y realistas que permitan experimentar emulando o simulando el comportamiento de estas grandes infraestructuras. En este trabajo, se revisaron y desarrollaron distintos ambientes de experimentación, tanto de un único host como distribuidos, que permiten analizar las ventajas y desventajas de las diferentes soluciones. Además, centrados en el Border Gateway Protocol (BGP), se ejecutó un conjunto de experimentos del plano de control sobre topologías fat-tree en entornos simulados y emulados. La validación de los resultados experimentales se complementó con un análisis teórico del comportamiento de BGP sobre un conjunto de escenarios seleccionados. Estos resultados experimentales y teóricos, así como la evaluación y desarrollo de los ambientes de experimentación, fueron oportunamente difundidos a la comunidad científica por medio de publicaciones científicas referadas. Todas los desarrollos de los entornos de experimentación realizados en el marco de este trabajo están disponibles y son reproducibles.

Palabras claves:
data center, fat-tree, emulación, enrutamiento, BGP.

ABSTRACT

Massive Scale Data Centers (MSDC) have become a key component of nowadays content-centric Internet architecture. With scales up to hundred thousands servers, conveying traffic inside these infrastructures requires much larger connectivity resources than traditional broadband Internet transit networks. MSCD uses fat-tree type topologies, which ensure multipath connectivity and constant bisection bandwidth between servers. To properly use the potential advantages of these topologies, specific routing protocols are needed, with multipath support and low control messaging load. These infrastructures are hugely expensive, and therefore, it is not possible to use them to experiment with new protocols. This rises the need for scalable and realistic emulation/simulation environments. In this work, we reviewed and developed several experimental environments, both single-host and distributed, analyzing the benefits and drawbacks of the different solutions. Also, focused on the Border Gateway Protocol (BGP), we ran a comprehensive set of control plane experiments over fat-tree topologies in both simulated and emulated environments. The validation of the experimental results was complemented with a theoretical analysis of BGP behavior over selected scenarios. These experimental and theoretical results, as well as the evaluation and development of the experimental environments, were opportunely disseminated to the scientific community through peer-reviewed scientific publications. All the development of the experimentation environments are available and can be reproduced.

Keywords:
data center, fat-tree, emulation, routing, BGP.

Lista de siglas

Lista de siglas y acrónimos

ARP Address Resolution Protocol 47, 48, 49

AS Sistema Autónomo 13

ASN Autonomous System Number 13, 14, 65

BGP Border Gateway Protocol 3, 4, 5, 12, 13, 14, 29, 31, 34, 42, 43, 44, 46, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 63, 67, 68, 72, 74, 75, 76

CDN Content Delivery Network 1

CORE Common Open Research Emulator 23, 46, 52, 54, 55, 58, 59, 72

DCE Direct Code Execution 25, 45, 46, 54, 76

DCN Data Center Network 3, 5, 6, 7, 8, 10, 11, 12, 13, 16, 18, 21, 22, 27, 53, 56, 57, 58, 73, 74, 76, 77

DPID Datapath ID 47

ECMP Equal Cost Multi-Path 2, 3, 8, 10, 13, 14, 33, 74, 76

EVPN Ethernet Virtual Private Networks 16, 21, 57

FRR Free Range Routing 21, 26, 43, 44, 46, 47, 53, 54, 56, 57, 58, 59, 76

GUI Graphical User Interface 23, 24, 58

IETF Internet Engineering Task Force 3, 5, 12, 16

IGP Interior Gateway Protocol 11, 14

IP Internet Protocol 8, 12, 14, 16, 26, 43, 44, 47, 53, 56

IS-IS Intermediate System to intermediate System 3, 5, 14, 17, 26, 29, 34, 42, 43, 46, 53, 54, 74

ISP Internet Service Provider 14

LAN Local Area Network 21

LSA Link State Advertisement 11, 14

LSP Link State PDU 34

LSVR Link State Vector Routing 3

MAC Media Access Control 47

MPLS Multiprotocol Label Switching 12

MSDC Massive Scale Data Centers 1, 3, 4, 5, 13, 27, 32, 36, 50, 51, 53, 57, 58, 72, 73, 74, 76

OSPF Open Shortest Path First 14, 17, 26, 57

PoD Point of Delivery 9, 10, 13, 31, 57, 58, 65, 67, 68

REST Representational State Transfer 45

RIFT Routing in Fat Trees 3, 5, 10, 12, 15, 16, 26, 29, 31, 34, 42, 43, 46, 53, 54

SDN Software-Defined Networking 3, 12, 17, 18, 42, 54, 72

STP Spanning Tree Protocol 2, 8, 11

TCP Transmission Control Protocol 76

TIE Topology Information Element 34

ToF Top of Fabric 8, 9

ToR Top of Rack 2, 6

VM Virtual Machine 57

VPCS Virtual PC Simulator 57

VPN Virtual Private Network 12

VXLAN Virtual Extensible LAN 16, 21, 57

WAN Wide Area Network 1, 6

eBGP External BGP 13

iBGP Internal BGP 13

vCPU Virtual CPU 52, 55, 56

vRAM Virtual RAM 52, 55, 56, 59, 74

Tabla de contenidos

Capítulo 1

Introducción

En la última década, Internet ha ido experimentando una transición gradual de una red jerárquica de redes, con proveedores de tránsito de nivel 1 y nivel 2, a una red basada en contenido [1]. Las redes de distribución de contenido o Content Delivery Network (CDN), buscan replicar el contenido lo más cerca posible del usuario para mejorar la calidad de la experiencia. Los proveedores de CDN y Over The Top (OTT) tales como Netflix o WhatsApp, entre otros, tienen presencia en todo el mundo e intercambian dos tipos de tráfico: el tráfico público para sus usuarios, y el tráfico privado, principalmente relacionado con la replicación de contenido. Estos proveedores, en conjunto con los proveedores de servicios en la nube, han impulsado el despliegue global de los denominados data centers de escala masiva o Massive Scale Data Centers (MSDC). Estos data centers pueden alojar a cientos de miles de servidores físicos, que a su vez pueden desplegar una gran cantidad de contenedores y máquinas virtuales. Las funcionalidades básicas de los MSDC incluyen la computación, el almacenamiento de datos y la replicación de datos, que se realiza mediante el intercambio de mensajes entre servidores utilizando la infraestructura de red disponible.

Transportar paquetes en este tipo de infraestructuras requiere una comunicación de gran escala, y un ancho de banda significativamente mayor al requerido por el tráfico de la histórica Wide Area Network (WAN) de Internet. En este sentido, es necesario desarrollar soluciones de reenvío, enrutamiento y transporte de paquetes que estén diseñadas específicamente para este caso de uso.

El tráfico del data center generalmente se clasifica como Este-Oeste y Norte-

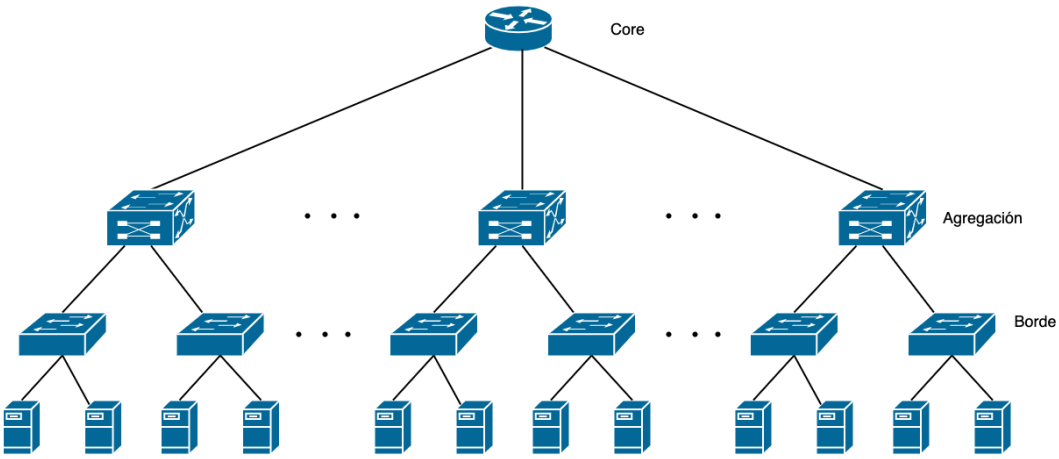


Figura 1.1: Una topología de red tradicional para data centers basada en un árbol de 3 niveles.

Sur. El tráfico Este-Oeste refiere al tráfico entre servidores de un mismo data center, mientras que el tráfico Norte-Sur se refiere al tráfico entre las aplicaciones que se ejecutan en el data center y los usuarios de Internet. Cabe destacar que el tráfico Este-Oeste representa casi el 85 % del tráfico total dentro de un data center [2].

Tradicionalmente, los data centers desplegaban una topología de árbol básica como la que se puede ver en la Figura 1.1 donde generalmente se empleaban dos o tres niveles: el nivel de Borde, el nivel de Agregación (en caso de ser necesario) y el nivel Core. Los servidores, son hojas del árbol que están conectados a conmutadores Top of Rack (ToR).

La búsqueda de un ancho de banda de bisección constante, es decir, la misma capacidad disponible para cualquier comunicación entre servidores, impulsó el resurgimiento de las redes no bloqueantes Clos [3]. Una topología fat-tree [4] es un caso particular de una red Clos, en donde se logra un alto ancho de banda de bisección mediante la interconexión switches con funcionalidades básicas. Esta topología será presentada en detalle más adelante en este documento; por más información consultar [5]-[7].

Las topologías de tipo fat-tree proveen una gran redundancia de caminos (o rutas), donde el ancho de banda combinado de los enlaces es potencialmente más alto que el de la ruta más corta tradicional. Para poder aprovechar esta diversidad de caminos, se deben implementar soluciones que contemplen Equal Cost Multi-Path (ECMP) [8]. Sin embargo, ECMP no se puede implementar sobre redes puras de capa 2 con Spanning Tree Protocol (STP) [9] (pues este

protocolo elimina las rutas redundantes "podando" el árbol), en cambio, se deben contemplar soluciones con enrutamiento, es decir, redes de capa 3 donde esto sea posible.

En el último tiempo, se han propuesto una variedad de protocolos de enrutamiento específicos para las topologías fat-tree, como lo son Border Gateway Protocol (BGP) para el data center [10] u Openfabric (Intermediate System to intermediate System (IS-IS) con reducción de inundación de paquetes) [11]. Además, existen nuevas propuestas que están bajo desarrollo y estandarización por la Internet Engineering Task Force (IETF) como lo son Routing in Fat Trees (RIFT)[12], [13] y Link State Vector Routing (LSVR) [14]. Estos últimos buscan combinar las ventajas de los algoritmos de tipo estado de enlaces y vector de distancias, junto con el conocimiento de la topología subyacente, para tener un mejor rendimiento dentro del data center.

Apartándose del enfoque de una red tradicional, donde se despliegan protocolos de enrutamiento distribuido y se explotan los múltiples caminos presentes en las Data Center Network (DCN)s mediante el uso de protocolos con soporte ECMP, se tiene a las Software-Defined Networking (SDN) [15]. La principal diferencia entre una SDN y una red tradicional es que en SDN el plano de control y el plano de datos están separados. Mientras el plano de datos (encargado del reenvío) está en los conmutadores, el plano de control (encargado del enrutamiento) está centralizado en un controlador SDN. Mientras que ECMP realiza la división del tráfico estáticamente en función de la información de los encabezados de los paquetes, en particular, este algoritmo no considera requisitos de ancho de banda de los flujos, con el enfoque SDN se puede lograr un balanceo de carga eficiente sobre la red, considerando múltiples factores imposibles a tener en cuenta por ECMP, como por ejemplo, la condición del tráfico de toda la red o el ancho de banda de los flujos [16] [17].

Para depurar las implementaciones de los protocolos de enrutamiento, introducir nuevas funcionalidades o experimentar con diferentes configuraciones de topologías y escenarios, es necesario contar con ambientes de desarrollo y pruebas. Entre otras razones, debido a las dimensiones de la infraestructura, costo y disponibilidad, sería impracticable experimentar con las instalaciones de un MSDC existente; además, los investigadores no pueden acceder a infraestructuras reales (p.e., el data center de Facebook [18]) con la escalabilidad necesaria para experimentar con los protocolos de enrutamiento mencionados anteriormente. Por lo tanto, es necesario contar con otros enfoques como

pueden ser la verificación basada en modelos o los entornos de emulación o simulación.

En este sentido, el aporte de esta tesis tiene dos dimensiones. Por un lado, se exploran diferentes soluciones de enrutamiento en el data center, presentando los distintos protocolos y las implementaciones disponibles al momento. Por otro, se analizan y construyen diferentes entornos de experimentación adecuados para topologías fat-tree típicas en los MSDC. Ambas dimensiones convergen en un análisis experimental en diversos entornos de simulación/emulación del enrutamiento en redes fat-tree.

Los objetivos específicos de esta tesis son: 1) estudio del estado del arte en soluciones de enrutamiento en el data center, 2) análisis experimental, desarrollo y despliegue de entornos de experimentación adecuados para topologías fat-tree 3) análisis experimental del enrutamiento en redes fat-tree sobre los ambientes de experimentación desplegados, tomando al protocolo BGP como caso de estudio, y 4) estudio del comportamiento teórico de BGP para la validación de los resultados experimentales obtenidos.

Este documento se organiza de la siguiente manera: en el capítulo 2 se presentan los fundamentos teóricos de esta tesis, exponiendo en detalle la topología fat-tree, el enrutamiento dentro del data center y presentando distintos emuladores y simuladores que luego son el centro de los ambientes de experimentación aquí utilizados. En el capítulo 3, la metodología de trabajo es mostrada. Esta metodología incluye: casos de uso típicos para testear protocolos de enrutamiento, criterios de paradas para la convergencia de estos protocolos, métricas independientes del tiempo para comparar el desempeño de estos protocolos sin perder precisión, y, finalmente, se presentan los distintos ambientes de experimentación utilizados. Luego, en el capítulo 4, se presentan los resultados de la experimentación. En este sentido, se presentan los resultados de la evaluación de los distintos ambientes de experimentación y se realiza una evaluación experimental del protocolo BGP para el data center. Los resultados experimentales de la evaluación de este protocolo son validados con un análisis teórico que acompaña la presentación de los resultados. Finalmente en el capítulo 5, se exponen las principales conclusiones de este trabajo.

Capítulo 2

Fundamentos teóricos

La búsqueda de un ancho de banda de bisección constante, es decir, la misma capacidad disponible para cualquier comunicación entre servidores dentro de la DCN, impulsó el resurgimiento de las redes Clos y, en particular, de la adopción de las topologías fat-tree por parte de los MSDC. Las topologías fat-tree proveen una gran redundancia de caminos (o rutas), donde el ancho de banda combinado de los enlaces es potencialmente más alto que el de la ruta más corta tradicional [5]-[7].

En el último tiempo, se han propuesto una variedad de protocolos de enrutamiento para este tipo de topologías. Por ejemplo, BGP para el data center [10] y Openfabric (IS-IS con reducción de inundación) [11]. Además, nuevas propuestas están bajo desarrollo y estandarización por la IETF, como por ejemplo, el protocolo RIFT [12], [13].

Además, para depurar las implementaciones de los protocolos de enrutamiento, introducir nuevas funcionalidades o experimentar con diferentes configuraciones de topologías y escenarios, es necesario contar con ambientes de desarrollo y testeo.

En este capítulo se introducirán los principales conceptos de este trabajo, así como los protocolos y tecnologías utilizadas a lo largo de esta tesis.

2.1. Redes de Data Center

A diferencia de una arquitectura de red troncal o *backbone* tradicional, donde los dispositivos están conectados en malla de manera arbitraria, en una arquitectura de data center moderna, la topología está formada de capas regu-

lares y altamente interconectadas de dispositivos de red. A su vez, a diferencia de las arquitecturas tradicionales de varios niveles que generalmente se usan en las redes WAN, la estructura del data center aplana la arquitectura de la red, reduciendo la distancia entre los puntos finales de la DCN. Este diseño permite una eficiencia de ancho de banda y baja latencia, proporcionando múltiples rutas redundantes y ofreciendo un mayor rendimiento total en comparación con una malla irregular [19].

La Figura 1.1 muestra una DCN basada en una topología de árbol simple. En la parte inferior, se ubican los racks de servidores a los que se conectan los hosts físicos. Cada host físico a su vez, puede alojar varias máquinas virtuales o servidores virtuales. Los hosts están conectados a los conmutadores denominados Top-of-Rack(ToR) (omisos en la figura). Luego, estos conmutadores ToR se conectan a un conmutador de borde. Varios conmutadores de borde se conectan a un conmutador de agregación para la agregación de tráfico. Finalmente, los conmutadores de agregación están conectados a uno o más conmutadores Core. En este enfoque, el conmutador Core tiene un enlace saliente para conectarse a Internet. Notar que puede haber más de un conmutador Core y se pueden instalar múltiples enlaces paralelos para la conectividad a Internet. Este tipo de topologías de árbol tiene múltiples puntos de falla. Por ejemplo, si falla un enlace entre un conmutador de agregación y uno de borde, los hosts que forman parte de este último perderán la conectividad.

En los años cincuenta, cuando el crecimiento de las conexiones cruzadas cableadas se volvió físicamente imposible, y por lo tanto también el de los costos, Charles Clos diseñó las redes Clos [20] de tres etapas (ingreso, intermedia, egreso). Esta topología de red ofrece altos niveles de ancho de banda para muchos dispositivos finales mediante la interconexión de conmutadores básicos pequeños. Este tipo de redes son utilizadas en la gran mayoría de las arquitecturas de los data center actuales [19]. La topología Clos proporciona una capacidad de no bloqueo al dimensionar adecuadamente la etapa intermedia. Otra ventaja es que puede escalar horizontalmente agregando más capacidad a las diferentes etapas, o verticalmente agregando más etapas. La topología fat-tree, no es más que una Clos plegada de tres etapas.

2.1.1. Tipos de tráfico

En una DCN se pueden visualizar dos tipos de tráfico: tráfico este-oeste y tráfico norte-sur. El tráfico este-oeste se refiere al tráfico entre servidores que tiene lugar debido a aplicaciones internas que requieren transferencias de datos, como por ejemplo, el cálculo de MapReduce para la indexación [21] o el movimientos de datos de almacenamiento entre servidores, entre otros. Si las aplicaciones del data center realizan, por ejemplo, muchos cálculos de MapReduce, es posible que se tenga principalmente tráfico de este-oeste. El tráfico norte-sur se refiere al tráfico como resultado de solicitudes externas de Internet que llegan al data center, las cuales son respondidas por los servidores. Por ejemplo, cuando los usuarios finales solicitan contenido web alojado en servidores del data center, este tráfico se clasifica como tráfico norte-sur. Según el propósito comercial específico de un data center, la cantidad de tráfico este-oeste y norte-sur puede variar ampliamente [6], aunque predicciones afirman que, en general, el tráfico este-oeste representa alrededor del 85 % del tráfico asociado con los data center [2].

2.1.2. Requisitos de una DCN

Un requisito importante de una DCN es lograr escalabilidad y flexibilidad con fácil administración. La escalabilidad y la flexibilidad refieren a la capacidad de hacer crecer el data center sin requerir reorganizaciones significativas respecto a, por ejemplo, el direccionamiento de la topología. Este direccionamiento también debe ser manejable o de fácil administración. Además, debido a la naturaleza de las aplicaciones y servicios que se alojan en un data center, una DCN debe ser tolerante a fallas. A continuación, se enumeran algunos requisitos extraídos de [6], que pueden clasificarse en términos generales, como relacionados con el enrutamiento en una DCN:

1. Tener una forma de comunicación eficiente entre hosts finales.

2. Evitar los bucles de reenvío.

3. Rápida detección de fallas.

4. Utilizar la red de manera eficiente.

5. Las máquinas virtuales, así como los servicios, se deben poder mover entre dos máquinas físicas sin necesidad de cambiar sus direcciones Internet Protocol (IP).

Estos requisitos han moldeado las DCN en cuanto a topología, direccionamiento y tecnologías. Por ejemplo, el requerimiento 2 puede satisfacerse mediante la elección de una estructura topológica adecuada, precisamente, en una topología basada en árbol como la topología fat-tree, los bucles de reenvío no existen. Por otro lado, dada la escala de estas estructuras y en particular el requisito 4, un direccionamiento de capa 2 no es factible. Es decir, este tipo de redes con, por ejemplo, el protocolo STP no aprovechan la diversidad de caminos de la topología, desperdiciando el ancho de banda de la DCN. En consecuencia, las DCN se implementan con enrutamiento y direccionamiento de capa de red, estas soluciones son conocidas como "IP Fabrics" [19]. Luego, con el objetivo de aprovechar la diversidad de caminos de la topología subyacente, se deben implementar soluciones de enrutamiento que contemplen ECMP. Notar que con un enfoque de capa 3, el requisito 5 no es posible, dado que mover una máquina virtual requeriría asignarle una nueva dirección IP perteneciente a la nueva subred en la que se aloja. En este sentido, y sabiendo que un enfoque de capa 2 no es escalable, surge la necesidad de una capa superpuesta u *overlay* de capa 2.

2.2. Topologías fat-tree

Las topologías de árbol son la clase de topologías más representativas para las DCNs, entre ellas, la topología de árbol básica, la fat-tree y la Clos de 5 etapas. Una topología fat-tree es un caso particular de una red Clos [3] donde se logra un alto ancho de banda de bisección mediante la interconexión de conmutadores básicos. La idea de la topología fat-tree se propuso inicialmente para la supercomputación [4] y luego se adaptó para redes de data centers [5], [6].

Topológicamente, las redes fat-tree pueden verse como grafos parcialmente ordenados, donde "nivel"denota el conjunto de nodos a la misma altura. En la Figura 2.1, se puede observar un fat-tree de tres niveles. El nivel superior se llama Top of Fabric (ToF) o Core y comprende conmutadores Core. El nivel inmediatamente inferior (nivel de agregación) se compone de conmutadores

Cantidad de PoDs	**k**
Switches Core	$(k/2)^2$
Switches Spine	$k^2/2$
Switches Leaf	$k^2/2$
Switches Totales	$5k^2/4$
Cantidad de Links	$k^3/2$
Hosts soportados	$k^3/4$

Tabla 2.1: Descripción de la topología fat-tree en base al factor k [6].

Spine. Finalmente, en el nivel 0 (nivel de borde o Edge), hay conmutadores Leaf. Un Point of Delivery (PoD) es un subconjunto del fat-tree que generalmente contiene solo nodos de tipo Leaf y Spine completamente interconectados entre sí. Un nodo en un PoD se comunica con nodos en otros PoD a través del nivel ToF.

La topología fat-tree consta de k PoDs, numerados de izquierda a derecha de PoD-0 a PoD-(k-1), con tres capas de conmutadores: Leaf, Spine, y Core. Por lo tanto, en una topología fat-tree de k PoDs, hay k conmutadores (cada uno con k puertos) en cada PoD, dispuestos en dos niveles de $k/2$ conmutadores, un nivel para conmutadores de tipo Leaf y otro para Spines. En cada uno de estos PoD, cada Leaf está conectado a los $k/2$ Spine. Hay $(k/2)^2$ conmutadores de tipo Core, cada uno de los cuales se conecta a k PoDs. Con esta descripción de topología, el factor k se puede utilizar para calcular el número de nodos en cada nivel del fat-tree. La Tabla 2.1 adoptada de [6] resume la información topológica de un fat-tree en términos del factor k.

Las topologías fat-tree se pueden clasificar en dos tipos: single-plane o multi-plane. En una topología multi-plane (el tipo de fat-tree descrito anteriormente), cada nodo Core se conecta a k PoDs. Por el contrario, en un single-plane cada nodo Core se conecta a todos los Spine. Con esta configuración, incluso si todos los Core menos uno están caídos, la conectividad entre los Leaf está garantizada. Sin embargo, presenta un inconveniente importante: la cantidad de puertos necesarios para cada Core aumenta significativamente, lo que podría ser inviable si k es demasiado grande. A lo largo de esta tesis, cuando se refiere a una topología fat-tree descrita mediante un único parámetro (el factor k) se alude a un fat-tree multi-plane como el que se muestra en la Figura 2.1.

Existen otras terminologías para describir los fat-tree, por ejemplo, la uti-

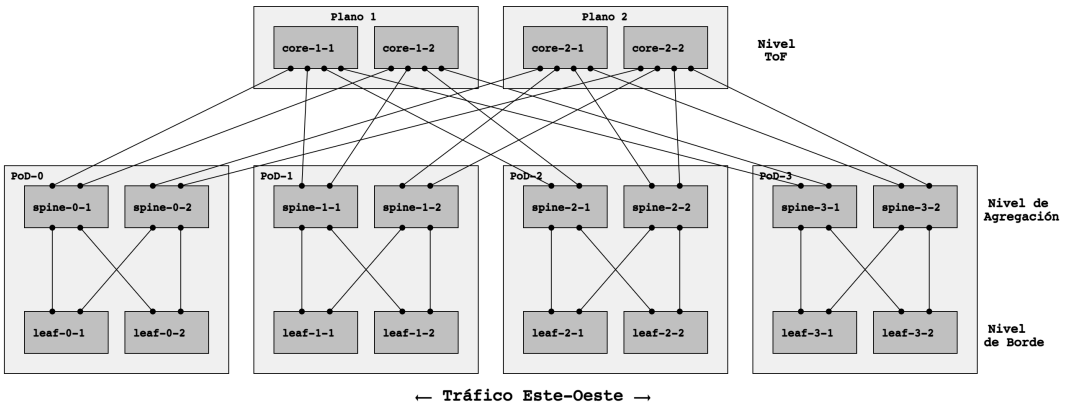


Figura 2.1: Una topología fat-tree multi-plane con $k = 4$: cuatro PoDs, con cuatro switches de cuatro puertos cada uno.

lizada en la especificación del protocolo RIFT [12]. Esa terminología especifica tres parámetros: K_{LEAF}, que describe la cantidad de puertos que apuntan al norte o al sur para los nodos de tipo Leaf, y K_{TOP}, que describe lo mismo para los nodos Spine. Finalmente, se denota por R el factor de redundancia, es decir, la cantidad de enlaces de un Core a cada PoD. Siguiendo esta notación, la Figura 2.2 representa a un fat-tree single-plane que se puede describir como $K_{LEAF} = 2$, $K_{TOP} = 2$, $R = 2$. A lo largo de esta tesis, cuando se refiere a una topología fat-tree descrita mediante esta notación extendida, se refiere a un fat-tree single-plane. Notar que si se quiere describir una topología multi-plane como la descrita anteriormente con esta notación, el factor de redundancia R debería de cumplir la condición $R = 1$. La Figura 2.1 entonces se puede describir de la siguiente manera: $K_{LEAF} = 2$, $K_{TOP} = 2$, $R = 1$.

La implementación de una topología multi-plane con estas propiedades tiene ciertos beneficios para el data center. Primero, todos los conmutadores son del mismo tipo y con la misma cantidad de puertos, lo que minimiza los períodos de inactividad y reduce los costos operativos (OPEX), pues un conmutador puede ser reemplazado con facilidad. Además, y generalizando a las single-plane, existen múltiples caminos entre cualquier par de hosts. En particular, en una topología fat-tree con factor k, hay $k/2$ rutas entre dos nodos Leaf dentro de un mismo PoD (intra-pod), y hay k rutas entre cualquier par de nodos Leaf que están en distintos PoDs (inter-pod). Esta solución de múltiples rutas inspira a explorar la estrategia de ECMP, que permite dividir la carga de tráfico de la DCN de manera más eficiente.

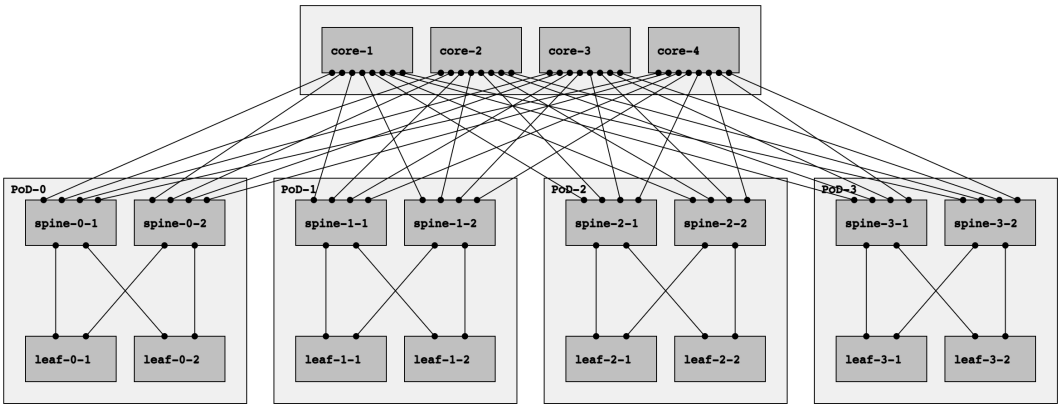


Figura 2.2: Una topología fat-tree single-plane con $K_{LEAF} = 2$, $K_{TOP} = 2$ y $R = 2$.

2.3. Enrutamiento en el data center

Para tener un adecuado enrutamiento y reenvío de paquetes en el data center, es importante aprovechar todas las características topológicas que brindan los fat-trees. Como se mencionó anteriormente en este capítulo, en una DCN se deben satisfacer algunos requerimientos básicos como: evitar bucles de reenvío, contar con una rápida detección de fallas, utilizar eficientemente la red (p.e., las soluciones con STP no son aceptables debido a que no se explotaría la diversidad de caminos de igual costo), contar con un enrutamiento que escale (además de la escalabilidad física), entre otros. Debido a estos requerimientos, una red de capa 2 no escalará y, por lo tanto, se deben adoptar soluciones de capa 3 (p.e., enrutamiento). A su vez, debido al requerimiento que refiere a la necesidad de migrar máquinas virtuales y contenedores entre servidores manteniendo la misma dirección IP, el data center deberá estar conectado por una estructura transparente de capa 2. Esto se implementa como una red superpuesta (*Overlay network*) sobre la solución de capa 3 [6].

Al ser el data center un único dominio administrativo, es natural pensar que los candidatos de llevar a cabo la tarea del enrutamiento son los clásicos Interior Gateway Protocol (IGP)s basados en estado de enlaces. Sin embargo, estos han sido diseñados para funcionar sobre topologías arbitrarias, y dada la escala y topologías de los data center, estos pueden conducir a problemas de escala debido a la inundación de Link State Advertisement (LSA)s. Por lo tanto, las posibles soluciones deben considerar la reducción de mensajes de inundación, explotar el conocimiento de la topología o utilizar otros algoritmos

de enrutamiento. En este sentido se presentarán los siguientes protocolos de enrutamiento: BGP con la configuración específica para el data center, algoritmos de estado de enlace con reducción de inundación (*flooding reduction*), y un protocolo en vías de estandarización por la IETF, el protocolo RIFT. Esta nueva propuesta aprovecha las ventajas de los protocolos estado de enlace y de vector de distancia para diseñar un algoritmo de enrutamiento específico para el data center.

Además de las soluciones mencionadas, que son aplicables en el paradigma de un plano de control distribuido (i.e., protocolos de enrutamiento), se presentará una solución adecuada para un ambiente SDN, es decir, donde el plano de control está lógicamente centralizado en un controlador SDN.

2.3.1. BGP en el data center

La elección de BGP [22] como protocolo de enrutamiento para los data centers está motivada por: 1) la existencia de implementaciones robustas, 2) la reducción de inundación de los paquetes del plano de control respecto a los protocolos de estado de enlaces, 3) el soporte nativo para una gran cantidad de protocolos como IPv4, IPv6, Multiprotocol Label Switching (MPLS) o Virtual Private Network (VPN)s, y, 4) el soporte para múltiples caminos (multi camino).

BGP fue inicialmente diseñado para el enrutamiento entre sistemas autónomos con un enfoque de ruta única y, por lo tanto, debe configurarse específicamente para el enrutamiento en el data center, donde existen múltiples rutas que deben ser consideradas. De hecho, la densa conectividad de la red del data center es muy diferente de la conectividad poco redundante entre sistemas autónomos de Internet.

En el enrutamiento inter dominio, es preferible estabilidad a tener rápidas notificaciones de los cambios. Es por esto que los transmisores BGP (o BGP speakers) generalmente posponen el envío de notificaciones sobre cambios por un tiempo determinado. Por el contrario, los operadores de los data centers quieren que los cambios en el enrutamiento en la DCN se propaguen lo más rápido posible. Además, debido a su comportamiento como protocolo de vector de caminos (*path-vector protocol*), cualquier falla en un enlace puede provocar que una gran cantidad de mensajes BGP pasen por todos los nodos de la red, una situación que debe evitarse en una DCN. Por último, cuando se aprende

un prefijo de muchos sistemas autónomos diferentes, los transmisores BGP construyen, de manera predeterminada, una única mejor ruta, mientras que en las DCN se necesita la elección de múltiples rutas.

Por estas razones, es evidente que la configuración de BGP debe ser ajustada de manera especial para poder alcanzar un rendimiento aceptable en el data center. En primer lugar, se prefiere External BGP (eBGP) sobre Internal BGP (iBGP) debido a su simplicidad para entender y configurar, en particular, las necesidades para el soporte multi camino. Por otra parte, la numeración de Sistema Autónomo (AS) en el data center es diferente a la numeración tradicional. En este sentido, solo se utilizan Autonomous System Number (ASN)s privados y, para admitir más de 1024 nodos (un escenario típico de un MSDC), se utiliza el espacio de direcciones ASN de 4 bytes [23]. Sin embargo, aunque el enfoque más sencillo para la asignación de ASNs es que a cada enrutador se le asigne un ASN diferente, este conduce al problema de *path hunting*, que es una variación del problema de conteo a infinito sufrido por los protocolos de vector de distancias [24]. Para evitar este problema, se puede seguir la siguiente práctica para la asignación de ASNs en topologías fat-tree: (1) A cada nodo Leaf se le asigna un ASN distinto; (2) Los Spine de un mismo PoD tienen el mismo ASN, que es diferente en cada PoD; (3) Todos los Core comparten el mismo ASN. Sin embargo, este modelo de asignación presenta el inconveniente de que la agregación de rutas[1] no es posible, dado que puede conducir a un *black-holing*[2]. Por esta razón, se necesita como ajustes adicionales, que el único atributo a considerar en el proceso de decisión de BGP sea el AS_PATH y que, a fin de soportar ECMP, un grupo de rutas para un destino dado se considere igual si la longitud del AS_PATH es la misma, relajando así el criterio que indica que los ASNs en los AS_PATH deben coincidir de forma exacta. Estas especificaciones están estandarizadas y pueden consultarse en [10] y [25] para obtener una descripción más detallada. La Figura 2.3 muestra un ejemplo de asignación de números de sistema autónomo para el fat-tree de ejemplo, siguiendo las reglas mencionadas.

Por último, el tipo de mensaje más importante en BGP es el mensaje de tipo Update, que se utiliza para enviar información detallada sobre rutas entre dispositivos BGP. Estos mensajes utilizan una estructura compleja que permite

[1] Método para minimizar el tamaño de las tablas de enrutamiento en una red IP mediante la consolidación de múltiples rutas en una sola.

[2] Lugares de la red donde el tráfico entrante o saliente se descarta sin informar al origen que los paquetes no llegaron a destino.

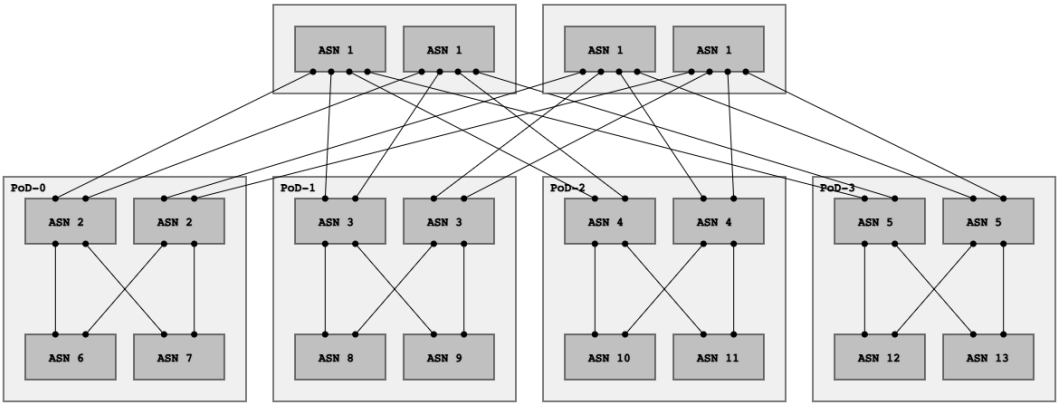


Figura 2.3: Ejemplo de numeración de sistema autónomo (ASN) para una topología fat-tree multi-plane con $k = 4$.

que un transmisor BGP especifique eficientemente nuevas rutas, actualice las existentes y retire rutas que ya no son alcanzables. Un mensaje Update puede anunciar como máximo una ruta, que puede estar descrita por varios atributos de ruta. A su vez, puede anunciar múltiples rutas que se retirarán (también llamado BGP Withdraw), cada una identificada por su prefijo IP.

2.3.2. Enrutamiento de tipo estado de enlaces

Los protocolos de enrutamiento de tipo estado de enlaces como Open Shortest Path First (OSPF) [26] o IS-IS [27] son, desde hace muchos años, el estado del arte de los IGPs que utilizan los Internet Service Provider (ISP)s en sus redes troncales o *backbones*. Sin embargo, a pesar de que son protocolos jerárquicos por diseño, una red enrutada con este tipo de protocolos no puede escalar con facilidad más allá de los mil enrutadores, debido principalmente a la inundación de los LSAs [28]. Hoy en día, los data centers fácilmente alojan decenas de miles de servidores, lo que se traduce a desplegar algunos miles de conmutadores/enrutadores. Por ejemplo, para cincuenta mil servidores, con una topología fat-tree, se necesitan más de cuatro mil enrutadores [6]; redes Clos más eficientes [29] demandan alrededor de 2400 conmutadores para cien mil servidores físicos. En los últimos años, la industria y la academia han estado trabajando en formas eficientes de reducir la inundación en este tipo de protocolos; por ejemplo, Openfabric es una implementación de *IS-IS Optimal Distributed Flooding for Dense Topologies* [11]. Esta modificación, junto con el soporte para ECMP que brinda este protocolo, hace factible la opción de desplegar este protocolo de estado de enlaces en un data center de gran escala.

2.3.3. RIFT: Routing in Fat Trees

El protocolo RIFT es parte del desarrollo de protocolos de enrutamiento específicos para el data center, donde el conocimiento de la topología fat-tree subyacente se puede utilizar como una ventaja para reducir la inundación de mensajes de control. Para cumplir este fin, RIFT combina los algoritmos de estado de enlaces y vector de distancias.

En este sentido, RIFT inunda con información de tipo estado de enlaces hacia el Norte, de esta manera, cada nivel de la topología conoce en su totalidad lo que tiene al Sur. Sin embargo, esta información de estado del enlace, salvo algunas excepciones, nunca se inunda de Este a Oeste o de vuelta al Sur. Esta característica define a RIFT como un protocolo donde la información no se distribuye uniformemente, sino que se resume a lo largo del gradiente Norte-Sur, donde los nodos no reciben la misma información desde múltiples direcciones en simultáneo.

Por otra parte, en la dirección Sur, el protocolo funciona como un protocolo de vector de distancias. La información se propaga un salto hacia el sur y es "re-anunciada"por nodos en el siguiente nivel inferior (normalmente solo la ruta predeterminada).

Luego de la convergencia, cada nivel de la topología tiene la vista completa de los niveles al Sur y una ruta predeterminada al nivel superior. Esto permite una buena agregación de rutas, por ejemplo, un nodo Leaf en su tabla de enrutamiento tendrá solo entradas para la ruta por defecto (una por cada Spine directamente conectado). Sin embargo, sin los recaudos necesarios se pueden generar *black holes* de tráfico, o incluso una partición parcial de la red debido a fallas o a una configuración inadecuada. RIFT aborda estos problemas implementando una desagregación automática de prefijos en caso de fallas de enlaces o nodos. Este mecanismo se basa en lo que se denomina la desagregación positiva no transitiva y la desagregación negativa transitiva. La desagregación positiva la utiliza un nodo que detecta que su prefijo por defecto cubre uno o más prefijos a los que se puede acceder a través de él, pero no a través de uno o más nodos del mismo nivel. Por lo tanto, este debe anunciar esos prefijos individualmente hacia el sur. Por ejemplo, en la Figura 2.2 el nodo *spine-0-2* puede disparar este mecanismo en caso de que el enlace entre el *spine-0-1* y el *leaf-0-1* esté caído, previniendo que tráfico con origen *leaf-0-2* y destino *leaf-0-1* termine en un *black hole* al elegir como primer salto el *spine-0-1*. Esta

desagregación no es transitiva porque los efectos siempre están contenidos en un solo nivel de la topología.

Por otra parte, la desagregación negativa transitiva, es utilizada por un nodo Core cuando descubre que no puede alcanzar un Leaf en una topología multi-plane. En este caso, este nodo tiene que desagregar todos los prefijos de dicho Leaf, enviándolos hacia el sur. Este tipo de desagregación es transitiva porque si un nodo recibe una desagregación negativa para un prefijo específico de un nivel inmediatamente superior, tiene que propagar dicha desagregación hacia el sur para llegar a los Leaf. Esto es necesario ya que los nodos Leaf son quienes ingresan tráfico al fat-tree, y al estar conectados a varios planos, deben poder elegir el plano correcto para evitar *black holes* en el tráfico. Estos mecanismos y otros conceptos de RIFT, como *South Reflection*, *Flood Repeater*, *Bandwidth Adjusted Distance*, entre otros, pueden consultarse en la especificación del protocolo (al momento de la escritura de este manuscrito, en vías de estandarización por la IETF) [12].

Existe una implementación comercial de este protocolo que se puede integrar al sistema operativo Junos OS de la empresa Juniper Networks[1]. Además, está disponible en [30] una implementación de software libre desarrollada en el lenguaje de programación Python, en adelante rift-python.

2.3.4. Red superpuesta de capa 2

Además del caso de uso del enrutamiento, un requisito importante de las DCN es la migración de máquinas virtuales entre diferentes servidores físicos manteniendo sus direcciones IP sin cambios (es decir, manteniendo su conexión a una subred de capa 3 determinada). Este es un requisito importante desde el punto de vista de la aplicación, que es completamente inconsciente de la conectividad subyacente. Como se mencionó en la sección 2.1 esto implica que el data center está conectado por una de red superpuesta de capa 2. Este requisito, se puede implementar utilizando soluciones como Virtual Extensible LAN (VXLAN) [31] o Ethernet Virtual Private Networks (EVPN) [32].

[1] https://www.juniper.net/us/en.html

2.4. Paradigmas del plano de control

En el mundo de las redes de computadoras actuales existen dos enfoques bien diferenciados acerca de donde se implementa la lógica del plano de control. Por un lado se tiene el clásico enfoque distribuido, donde toda la lógica del plano de control está distribuida en todos los dispositivos de la red. De esta manera, la lógica que determina cómo se reenvía un datagrama de un enrutador a otro a lo largo de una ruta se encuentra dentro de cada enrutador. Esto se puede llevar a cabo con protocolos de enrutamiento distribuido (p.e., OSPF o IS-IS) que ejecutan instancias independientes en cada enrutador de la red. Por otro lado, en el último tiempo, el enfoque centralizado, también llamado SDN ha ido ganando terreno. Con este enfoque, las tareas del plano de control como el enrutamiento y la gestión de los servicios y componentes de la capa de red se realizan de manera centralizada, típicamente en un controlador SDN remoto [33]. La Figura 2.4 muestra la separación de los planos en donde el control del enrutamiento se encuentra lógicamente centralizado en un controlador SDN.

Este controlador es quien decide de manera centralizada el camino que siguen los paquetes en la red. Para realizar esta tarea, pueden valerse de cualquier algoritmo conocido (p.e., Dijkstra) o personalizado (p.e., tomando en cuenta la utilización de los enlaces), agregando una gran flexibilidad y programabilidad a la red.

2.4.1. Controladores SDN

En la teoría, un controlador SDN proporciona servicios que pueden implementar un plano de control distribuido y apoyar en las tareas de gestión y centralización del estado de la red. En realidad, cualquier instancia de un controlador proporcionará una porción o subconjunto de estas funcionalidades, así como su propia versión de estos conceptos [34]. Dentro de la oferta de controladores SDN se encuentran POX [35] y Ryu [36].

Ryu es un framework de código abierto basado en componentes e implementado completamente en Python. Este controlador proporciona componentes de software, con APIs bien definidas, que facilitan la creación de nuevas aplicaciones de control y administración de redes. Este enfoque de componentes flexibiliza la personalización de implementaciones existentes, pudiendo adaptarlas a necesidades específicas. Es compatible con los protocolos de administración

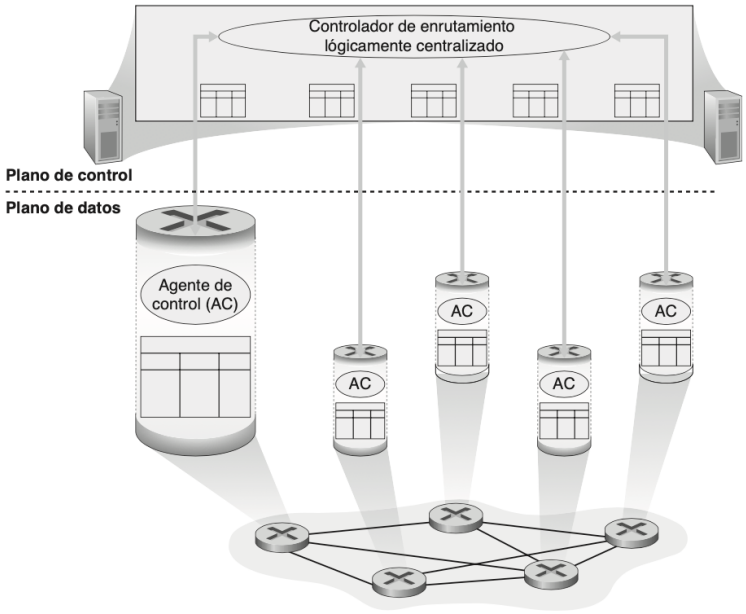


Figura 2.4: Plano de control lógicamente centralizado: un controlador remoto interactúa con agentes de control (AC) locales a cada nodo [33].

de red NETCONF [37] y OF-config, así como con OpenFlow [38], el estándar de comunicación SDN más desplegado [34].

Por otra parte, POX es un controlador SDN de código abierto basado en Python. Soporta Openflow 1.0 y es utilizado como una manera fácil de ejecutar experimentos OpenFlow/SDN [39], de hecho, este controlador viene preinstalado con la máquina virtual provista por el emulador Mininet [40].

La arquitectura basada en componentes hace que Ryu sea más robusto que POX, y por lo tanto, que el rendimiento sea mejor. Una evaluación del rendimiento de ambos controladores utilizando Mininet puede consultase en [41].

2.5. Virtualización de la red

La comunidad de redes ha ido adoptando progresivamente el modelo NetDevOps, que implica incorporar los conceptos de desarrollo e integración continuos de la industria del software al ciclo de vida de la gestión de redes y, en particular, de las DCN a gran escala. De hecho, varios actores industriales han desarrollado entornos de emulación para experimentar con nuevas tecnologías antes de incorporarlas a la operación, incluido Nokia [42], Cisco [43], Nvidia/Cumulus [44], entre otros. Además, algunos de estos entornos se han presentado a la academia, como Microsoft CrystalNet [45] y Huawei Net-

Graph [46], pero, lamentablemente, son software propietario y por lo tanto, los resultados no se pueden reproducir.

Es importante aclarar las diferencias entre virtualización, emulación y simulación de redes. Mientras que la virtualización de red consiste en usar un dispositivo virtual en una red de producción como reemplazo de un dispositivo físico, la emulación de red consiste en emular un dispositivo de producción con un equivalente virtual, ejecutando las mismas imágenes de software que el dispositivo de producción. Por otro lado, la simulación busca imitar la funcionalidad del dispositivo de producción implementado al mismo en un entorno completamente diferente, por ejemplo, en un simulador de eventos discretos. Es importante señalar que, aunque la emulación ejecuta la misma imagen que el software de producción, los recursos subyacentes son inherentemente menos poderosos; por ejemplo, mientras que el software del plano de control (como los protocolos de enrutamiento y señalización) generalmente se ejecutan en CPUs de propósito general, el reenvío del plano de datos se implementa en un circuito integrado para aplicaciones específicas (o ASIC por sus siglas en inglés) especializados, que no se puede emular. Por lo tanto, mientras que la emulación es una técnica muy atractiva para probar y depurar el plano de control, los resultados de la emulación del plano de datos solo se pueden tener en cuenta como una aproximación al comportamiento de la red de producción, pero no es posible usarlos para medir desempeño.

Además de la emulación y la simulación, la depuración del plano de control se puede realizar mediante la verificación basada en modelos [47]. Aunque estos métodos teóricamente pueden explorar todas las rutas de ejecución posibles, no pueden considerar el impacto de los errores de software, ya que realizan análisis de configuración estática y asumen implementaciones correctas y sin errores. Según [48], los errores de software tienen un impacto significativo en las fallas de los data centers, y representan el 12 % de las fallas en los data centers de Facebook. Por otro lado, en las pruebas basadas en emulación, los dispositivos de red ejecutan firmware de red sin modificar y, por lo tanto, están expuestos a errores de software reales.

Los operadores de red, desarrolladores e investigadores utilizan con frecuencia simuladores de red populares como ns-3 [49] u OMNeT++ [50], que tienen frameworks robustos para la simulación de redes, para por ejemplo, evaluar, probar y replicar propuestas de protocolos, implementaciones de servicios y arquitecturas.

Sin embargo, las capacidades de las simulaciones no son suficientes para cubrir algunos requisitos impuestos por los actores, como el despliegue de elementos de red heterogéneos, la inclusión de tecnologías emergentes o el comportamiento de los usuarios. La simulación también se considera un método débil para probar nuevas propuestas, porque en general no admite la ejecución de aplicaciones no modificadas (usualmente, se debe programar el comportamiento de la aplicación en el lenguaje del simulador). Además, no proporciona resultados cercanos al comportamiento real de las redes actuales, que son más complejas y no deterministas [51], [52].

En consecuencia, a menudo se considera que las plataformas de emulación superan las limitaciones de la simulación. La emulación tiene como objetivo mejorar la fidelidad de la experimentación al permitir la reutilización de protocolos y aplicaciones del mundo real dentro de una red virtual. Existe una gran variedad de emuladores de redes, cada uno con sus tecnologías, ventajas y desventajas.

A pesar de las ventajas descritas de los emuladores sobre los simuladores, es importante señalar el principal inconveniente de los emuladores: la complejidad de los sistemas emulados limita su escalabilidad. Esto se manifiesta en limitaciones de soporte para redes de gran tamaño y velocidades de enlace en tiempo real.

La elección de un entorno de experimentación depende de múltiples factores, y puede ser diferente para operadores de red, investigadores u otros actores. Esta decisión puede involucrar simuladores, emuladores y otras herramientas de red. Las siguientes secciones presentan las herramientas de red y los entornos de emulación y simulación relevantes para este trabajo.

2.5.1. Kathará

Kathará [53], [54] es un sistema de emulación de red. Emula dispositivos de red con contenedores Docker [55] y los interconecta de forma flexible, lo cual permite que Kathará se ejecute en sistemas operativos convencionales como Windows, Linux y macOS. Además, el uso de este tipo de contenedores permite que los dispositivos usen diferentes imágenes de sistemas de archivos en el mismo escenario de red.

Esta flexibilidad habilita a que los nodos de la red emulada tengan diferentes comportamientos. Por ejemplo, un nodo puede ser configurado para

utilizar los protocolos de enrutamiento proporcionados por Quagga [56] o Free Range Routing (FRR) [57], [58], simplemente seleccionando la imagen Docker apropiada. Además, permite configurar los nodos para que utilicen una imagen Docker personalizada, con, por ejemplo, todos los protocolos de enrutamiento y las herramientas de red necesarias para experimentar en una DCN. En este sentido, en [59] se ofrece un conjunto de imágenes prediseñadas que pueden ser de utilidad para ejecutar un conjunto variado de escenarios.

La orquestación de la emulación en Kathará está completamente basada en configuraciones. Integra herramientas que automatizan estas configuraciones pero no ofrece ninguna programabilidad incorporada para esta tarea. Esto quiere decir que, a la hora de utilizar este emulador para ejecutar un experimento, se deberán generar configuraciones previas o interactuar directamente con el programa mediante los comandos disponibles para por ejemplo comenzar, configurar los nodos, y finalmente, terminar la emulación.

2.5.2. Megalos

Megalos [60] es el nombre que se le da a Kathará cuando cuando se utiliza Kubernetes [61] como administrador de la virtualización, es decir, se encarga de distribuir y administrar los contenedores Docker (que emulan a los dispositivos) y sus conexiones, entre un conjunto de hosts indicados.

Al igual que Kathará, Megalos implementa escenarios de redes virtuales que consisten en dispositivos virtuales, en este contexto denominados Funciones de red virtual (VNF), donde cada uno puede tener varias interfaces L2 asignadas a Local Area Network (LAN)s virtuales. Al explotar la tecnología VXLAN/EVPN Megalos garantiza la segregación del tráfico de cada una de estas LANs virtuales de otras LAN y del tráfico de Internet. Además, esta característica permite distribuir VNFs (es decir, dispositivos) en un clúster de servidores, donde la superposición L2 permite separar el tráfico de emulación del tráfico del clúster. La distribución de la emulación en múltiples nodos es lo que permite a Megalos emular infraestructuras de red a gran escala.

2.5.3. Mininet

Mininet [40], [62] se define como un sistema de orquestación de emulación de red. Puede emular hosts finales, conmutadores, enrutadores y enlaces en un solo núcleo Linux. Utiliza una virtualización “ligera”, que permite emular la

infraestructura de red en un solo host, ejecutando el mismo núcleo, sistema y código de usuario. Mininet es de código abierto, fácil de desplegar y proporciona una interfaz programable para definir y crear configuraciones de red.

El mecanismo de contenedores Linux utilizado por Mininet permite que los grupos de procesos que se ejecutan en el mismo kernel, tengan espacios de nombres independientes para los recursos del sistema (por ejemplo, interfaces de red y sistemas de archivos). Para cada host virtual, Mininet crea un contenedor asociado a un Linux namespace. Cada Linux namespace tiene una interfaz de red virtual conectada a un conmutador de software a través de enlaces Ethernet virtuales.

Mininet logra un alto rendimiento en servidores básicos, que satisface las demandas de las pruebas de red usuales. También, alcanza altos anchos de banda agregados y tiempos medios de ida y vuelta (RTT) bajos, incluso en escenarios exigentes donde se incluyen más de mil hosts en las topologías emuladas [51].

Otra característica importante de Mininet es su programabilidad. En este sentido, Mininet integra en su núcleo una API de Python. Por lo tanto, Python se usa para la orquestación, pero la emulación se realiza mediante código C compilado. Esta API cuenta con buena documentación, soporte de la comunidad, es fácil de usar y debido a que está integrada con el núcleo de Mininet, todo se puede construir con ella.

Este emulador también integra varios elementos SDN. Por ejemplo, en [63], se combina Mininet y el controlador ONOS [64] para presentar un análisis de escalabilidad. Por todas las características anteriormente mencionadas, Mininet se ha convertido en una de las herramientas de emulación de referencia.

2.5.4. MaxiNet

MaxiNet [65] es una extensión de Mininet que permite distribuir la emulación en varios hosts. Esto hace posible escalar la emulación de un único host que realiza Mininet, logrando emular redes con topologías más densas y con una gran cantidad de nodos como lo son las DCN.

Se puede ver a MaxiNet como una capa de abstracción que conecta múltiples instancias de Mininet que se ejecutan en diferentes hosts y que pueden ser accedidas desde una API centralizada. Además, se utilizan túneles GRE (Generic Routing Encapsulation) para conectar los nodos emulados que ten-

gan conexión entre si y que sus emulaciones tengan lugar en diferentes hosts. MaxiNet funciona como una interfaz que configura todas las instancias de Mininet, invoca comandos en los nodos y configura los túneles necesarios para una conectividad adecuada.

Para particionar una red virtual en varios hosts, utiliza la biblioteca de particionamiento de grafos METIS [66]. Para n hosts, METIS calcula n particiones de casi el mismo peso. El objetivo del proceso de partición es confinar la mayor parte del tráfico emulado localmente a cada host.

Al igual que con Mininet, mediante la API Python de MaxiNet se puede configurar, controlar y terminar una emulación de red. Esta API está diseñada para ser muy cercana, en términos de usabilidad, de la API de Mininet, facilitando así el uso de MaxiNet cuando ya se está familiarizado con Mininet.

2.5.5. CORE

Common Open Research Emulator (CORE) [67] es un emulador de red en tiempo real. Emula enrutadores, PCs y otros hosts y simula los enlaces de red entre ellos. Permite que las redes emuladas se pueden conectar en tiempo real a redes físicas y enrutadores. Este tipo de funcionalidad no es común en los emuladores disponibles y permite, por ejemplo, aumentar el tamaño de las redes físicas de prueba. CORE también puede distribuir la emulación en una o más máquinas Linux y proporciona un entorno para ejecutar aplicaciones y protocolos reales, aprovechando las herramientas proporcionadas por el sistema operativo Linux.

La arquitectura clásica consta de una Graphical User Interface (GUI) para dibujar topologías, una capa de servicios que crea instancias de máquinas virtuales "livianas" y una API para utilizarlas en conjunto. En la arquitectura actual, la virtualización se puede realizar mediante FreeBSD jails, Linux OpenVZ y Linux Namespaces (al igual que Mininet). En [68], estas diferentes plataformas se comparan en términos de características de virtualización y rendimiento.

Recientemente, CORE incorporó una API de Python como la disponible para Mininet. Escribir scripts de Python ofrece un rico entorno de programación con control completo sobre todos los aspectos de la emulación. En general, un script de Python CORE no se conecta al demonio CORE; de hecho, el *core-daemon* es solo otro script de Python que usa los módulos CORE de Python

e intercambia mensajes con la GUI. Esta API proporciona muchas funcionalidades, pero carece de documentación y, por lo tanto, usar la API tiene una curva de aprendizaje empinada.

2.5.6. GNS3

GNS3 [69] es un software de código abierto para emular, configurar y probar redes virtuales y reales. Permite ejecutar topologías pequeñas y complejas en un solo host. GNS3 admite dispositivos emulados y simulados. Para tener una emulación completa, se debe proporcionar una imagen (por ejemplo, Cisco IOS) y luego GNS3 emulará el hardware para el dispositivo seleccionado. Por otro lado, puede simular las características y funcionalidades de un conjunto de dispositivos (por ejemplo, un conmutador de capa 2 o un host).

GNS3 consta de dos componentes de software: el software "todo en uno" (GUI) y la máquina virtual GNS3. La primera es la parte "cliente" y puede ejecutarse sobre todos los sistemas operativos usuales. Este componente permite crear topologías, pero los dispositivos creados deben ser alojados y ejecutados por un proceso de "servidor". GNS3 permite asignar la parte servidor (i) localmente en la misma PC, (ii) localmente en una Máquina Virtual (VM), o (iii) remotamente en una VM.

Inicialmente, esta herramienta solo podía emular dispositivos Cisco usando un software llamado Dynamips[1], pero hoy en día, es compatible con muchos dispositivos de múltiples proveedores de redes, incluidos conmutadores virtuales Cisco, routers virtuales Brocade, conmutadores Cumulus Linux, instancias de Docker, múltiples dispositivos Linux, entre muchos otros.

Como emulador de red "gráfico", la orquestación de los experimentos se realiza completamente sobre la GUI proporcionada. Si bien esta función destaca la usabilidad, por otro lado, impide la programabilidad.

2.5.7. ns-3

ns-3 [49] es un simulador de eventos discretos para redes. Es un software de código abierto y está destinado principalmente a la investigación y el uso educativo. Como en la mayoría de las herramientas de simulación de redes, el simulador ns–3 tiene modelos para los diversos elementos que componen

[1] https://github.com/GNS3/dynamips

una red de computadoras, es decir, nodos y dispositivos de red, canales de comunicación, protocolos de comunicación, paquetes, entre otros.

Este simulador está completamente desarrollado y distribuido en el lenguaje de programación C++, aunque provee vínculos Python (*Python bindings*) para permitir que el código C++ se llame desde Python. Para construir una simulación usando ns–3, se debe escribir un programa principal en C++ que construya los diversos elementos necesarios para describir la red que se está simulando y la actividad deseada para esa red. Luego, este programa se compila y se vincula con la biblioteca de modelos de red de ns–3 [70].

Direct Code Execution (DCE) es un framework para ns-3 que proporciona funciones para ejecutar, dentro de ns-3, las implementaciones existentes de los protocolos o aplicaciones de red sin cambios en el código fuente. Por ejemplo, en lugar de utilizar o programar el comportamiento de un protocolo de enrutamiento en la simulación, se puede utilizar la implementación real de este protocolo compilada adecuadamente para utilizar con este framework. Esta funcionalidad, permite paliar una de las desventajas que en general presentan los simuladores por sobre los emuladores, permitiendo ejecutar implementaciones de protocolos y aplicaciones del mundo real, en el simulado.

2.6. Herramientas de análisis de tráfico

Los analizadores de tráfico son herramientas de software específicas que interceptan y registran el tráfico que atraviesa un enlace de red mediante la captura de paquetes. Estos paquetes capturados se pueden analizar y visualizar con distintas herramientas. La visualización de una gran variedad campos que permiten interpretar el contenido (cabezales, protocolos, carga útil, entre otros), se puede realizar, por ejemplo, con la herramienta gráfica Wireshark [71] o con el clásico analizador de paquetes de línea de comandos Tcpdump [72]. En su comportamiento por defecto, estas herramientas utilizan la biblioteca pcap [72] para capturar el tráfico de la primera interfaz de red disponible y despliegan una línea con el resumen de cada paquete recibido.

Las funcionalidades de Wireshark también están disponibles en una herramienta de línea de comandos llamada TShark [73]. Al igual que Wireshark, TShark permite capturar paquetes de una red en tiempo real o leer paquetes de un archivo de captura guardado previamente. TShark funciona de manera muy similar a Tcpdump, siendo la principal diferencia en la variedad de opcio-

nes soportadas para cambiar su comportamiento. Una de las grandes ventajas de TShark sobre Tcpdump es la posibilidad de usar disectores Wireshark para el filtrado en tiempo real. La disección de un paquete se compone de tantos disectores como protocolos involucrados, por ejemplo, un paquete HTTP puede involucrar un disector por capa: Ethernet, IP, TCP y HTTP. Cada disector decodifica su parte del protocolo (en general centrado en una capa) y luego entrega la decodificación a disectores posteriores para que decodifiquen las siguientes capas (asumiendo una arquitectura de capas). Cada disección comienza con el disector Frame, que disecciona los detalles del propio archivo de captura (por ejemplo, marcas de tiempo). Después de eso, el proceso continúa como el ejemplo de disección de paquetes HTTP.

Los disectores pueden integrarse en Wireshark o escribirse como un complemento (una biblioteca compartida o DLL). Esta funcionalidad es clave para, por ejemplo, la incorporación de disectores a medida para protocolos aún no estandarizados o en desarrollo (p.e., RIFT). Además, Tshark y todas sus funcionalidades se pueden usar con Python a través de un contenedor llamado Pyshark [74].

En [75], [76] se puede consultar una guía completa sobre el uso práctico de Wireshark y Tcpdump, incluida una discusión sobre los formatos de tráfico y otras herramientas.

2.7. Suite de Protocolos FRR

La suite de protocolos Free Range Routing (FRR) [57][58] es una bifurcación del proyecto Quagga [56] que proporciona servicios de enrutamiento IP. Su función en la red incluye: intercambiar información de enrutamiento entre enrutadores, tomar decisiones y establecer políticas de enrutamiento e informar a otras capas sobre estas decisiones. En un escenario usual, FRR instala reglas de enrutamiento en el núcleo del sistema operativo, lo que permite que la pila de red del núcleo cree las reglas de reenvío correspondientes. FRR provee una amplia gama de configuraciones de capa 3 (L3) tales como enrutamiento dinámico, configuración de rutas estáticas, direccionamiento, entre otras. En particular, esta suite de protocolos provee implementaciones para BGP (incluida la configuración específica para el data center), OSPF, Openfabric e IS-IS.

2.8. Consideraciones finales

En este capítulo se plasmaron los fundamentos teóricos en términos de: la descripción y principales requisitos de una DCN, la topología fat-tree típica de estas redes, los protocolos de enrutamiento relevantes para el caso de uso MSDC, así como otras cuestiones del enrutamiento en el data center. Finalmente, se presentaron emuladores, simuladores y herramientas de red que serán una parte central de los ambientes de experimentación utilizados en este trabajo.

Capítulo 3

Metodología

La metodología definida responde a un análisis experimental en diversos entornos de simulación/emulación del enrutamiento en redes fat-tree. En este sentido, se ha llevado a cabo un conjunto exhaustivo de experimentos utilizando los entornos y herramientas de experimentación presentados en la sección 2. El flujo de trabajo seguido para realizar la experimentación incluye:

- Preparación del ambiente de emulación/simulación: en esta etapa se estudia el emulador o simulador a utilizar, se instala y despliega en una infraestructura adecuada para la escala de los experimentos, y por último, de ser posible y necesario, se adaptan o agregan funcionalidades personalizadas.

- Generación de fat-trees: en esta etapa se programan o utilizan herramientas que permitan la auto generación de topologías de tipo fat-tree arbitrarias, para el entorno de experimentación.

- Estudio y configuración de protocolos de enrutamiento: se debe estudiar los protocolos a desplegar, comprendiendo tanto su funcionamiento como su despliegue y configuraciones necesarias. Debido a la escala de los experimentos (cientos o miles de nodos de red), estas configuraciones deben ser automatizadas en la generación de la topología para cada entorno de experimentación.

- Establecer criterios de parada para el experimento: pueden ser independientes del entorno y protocolo con el que se esté experimentando (como por ejemplo el tiempo) o ser dependientes del entorno o protocolo.

- Establecimiento de métricas: para un correcto análisis de los protocolos de enrutamiento sobre entornos emulados, se debe contar con métricas independientes del tiempo que permitan obtener información de alto nivel sobre el comportamiento o desempeño de estos protocolos. Por ejemplo, la cantidad de paquetes de control inyectados por un protocolo ante un evento de falla.

- Ejecución de casos de uso, recolección y análisis de resultados: en esta etapa se adaptan los programas de experimentación para que se guarden los datos necesarios que permitan la validación de la ejecución de los casos de uso, y a su vez, permitan utilizar las métricas definidas. Finalmente, se ejecutan los casos de uso, se recolectan y analizan los resultados.

La metodología propuesta para la evaluación de los protocolos de enrutamiento [77], se basa en las siguientes premisas:

1. Los fat-trees son topologías regulares. Esto permite realizar pruebas enfocadas en elementos específicos de la topología para obtener resultados con un valor general.

2. En los entornos emulados, la realización de pruebas que midan los tiempos reales no es fiable, ya que los tiempos dependen en gran medida del hardware utilizado. Además, suponer que los relojes del sistema de los nodos emulados están perfectamente sincronizados no es realista. Por lo tanto, se evalúan las implementaciones sin considerar los tiempos reales.

3. Los protocolos de enrutamiento pueden ser conceptualmente muy diferentes. Por esta razón, se debe adoptar un método de caja negra, ignorando, tanto como sea posible, las particularidades internas de las implementaciones.

En todos los entornos emulados, se desplegaron los protocolos de enrutamiento para el data center presentados en la sección 2, es decir, BGP, IS-IS, Openfabric y RIFT. Por otra parte, en el simulado se desplegó únicamente BGP.

En el resto del capítulo se definen: los casos de uso para probar el funcionamiento de los protocolos de enrutamiento, los criterios de parada para el experimento, y las métricas independientes del tiempo para el correcto análisis de los protocolos de enrutamiento. Luego, se presenta la metodología a seguir

para el análisis de los resultados, tanto de la validación de los entornos de experimentación, como del comportamiento de los protocolos de enrutamiento. Finalmente, se presentan ambientes de experimentación adecuados para llevar a cabo los experimentos.

3.1. Casos de uso

Los casos de uso definen los eventos considerados para probar el funcionamiento de los protocolos de enrutamiento en circunstancias operativas usuales en los data centers [78], [79].

- **Bootstrap**: El objetivo de este caso de uso es verificar el comportamiento del protocolo cuando este inicia. Todos los elementos de la topología están configurados para funcionar de manera correcta.

- **Node Failure**: Esta prueba verifica la capacidad de un protocolo de converger luego de la falla de un único nodo de la topología. Para esto, se considera la falla de un nodo de cada nivel del fat-tree por prueba, es decir, un nodo de tipo Leaf, uno de tipo Spine y finalmente uno de tipo Core. La implementación de la falla se puede resolver de múltiples maneras, por ejemplo, parando la emulación o simulación del nodo elegido para la falla, aislando el nodo del resto de la topología (eliminando todos sus enlaces) o deteniendo el proceso que ejecuta el demonio que se encarga del enrutamiento en dicho nodo.

- **Node Recovery**: El objetivo de esta prueba es verificar la capacidad que tiene un protocolo de retornar a la operativa normal, luego de la recuperación de un nodo Leaf, Spine o Core.

- **Link Failure**: El objetivo de esta prueba es verificar la capacidad que tiene un protocolo para converger luego de la falla de un único enlace de la topología. Para esto, se considera la falla de un enlace de cada nivel del fat-tree por prueba, es decir, un enlace Leaf-Spine y otro Spine-Core.

- **Link Recovery**: El objetivo de esta prueba es verificar la capacidad que tiene un protocolo de retornar a la operativa normal, luego de la recuperación de un enlace Leaf-Spine o Spine-Core.

- **Partitioned Fabric**: El objetivo de esta prueba es verificar la capacidad de respuesta que tiene un protocolo luego de que un nodo de tipo Core no puede alcanzar a ningún prefijo de un PoD debido a múltiples fallas de enlace.

- **Fallen Leaf**: Este caso sucede cuando debido a algún problema de conectividad, se puede acceder a un nodo Leaf solamente a través de un subconjunto de los nodos Core, pero no desde todos ellos. Este problema está definido en la especificación del protocolo RIFT como "*Fallen Leaf Problem*"dado que RIFT implementa un mecanismo específico para recuperarse de esta falla.

Para cada caso de uso, las operaciones típicas incluyen: 1) iniciar la emulación o simulación, 2) capturar los mensajes del plano de control intercambiados, 3) comprobar la convergencia, 4) detener la emulación , y 5) analizar las capturas y datos recolectados.

El propósito de comprobar la convergencia es verificar que, luego de que se produce un evento (modelado con un caso de uso), las tablas de reenvío del plano de datos de los nodos alcanzan el estado esperado (p.e., luego del *Bootstrap* todos los nodos Leaf deben tener alcanzabilidad a los demás). Como se detalla en la sección 3.2, esta verificación de convergencia se puede implementar de varias maneras.

Finalmente, notar que este enfoque tiene el objetivo de analizar el comportamiento de los protocolos de enrutamiento. Para validar los entornos de emulación/simulación, el proceso definido consiste en ejecutar el caso de uso denominado *Bootstrap* con el protocolo BGP, para una variedad significativa de topologías fat-trees. Esto se puede lograr variando el factor k que describe a estas topologías, generando además, una noción de escalabilidad para estos entornos.

3.2. Criterios de parada para la convergencia de protocolos de enrutamiento

Independientemente del entorno o del caso de uso particular, experimentar con protocolos de enrutamiento presenta un desafío peculiar: determinar

cuándo ha convergido un protocolo. Abordar este problema es crucial para tener criterios de parada precisos para un conjunto de casos de uso, por ejemplo, verificar el comportamiento esperado de los protocolos en el arranque (*Bootstrap*) o determinar cuándo un protocolo se ha recuperado de una falla (*Node Recovery*). Esta tarea se puede realizar de múltiples formas, algunas dependientes de los mensajes de control de cada protocolo y otras independientes del protocolo. Por supuesto, existe una forma intuitiva de resolver el problema: esperar un tiempo suficiente para estar seguro de la convergencia del protocolo. Esto podría no ser un problema para topologías no densas o incluso fat-trees con pocos nodos. Sin embargo, si se va a realizar la emulación de un MSDC (eventualmente en un entorno distribuido), es difícil determinar cuánto tiempo sería suficiente esperar.

En este trabajo se han explorado tres formas de resolver este problema: 1) calcular previamente las tablas de enrutamiento de cada nodo, 2) alimentar una bandera de parada y 3) mover una ventana deslizante.

Debido a la naturaleza de una emulación, donde un programa centralizado es responsable de iniciar y finalizar el experimento, todas estas soluciones son "conscientes" de una entidad centralizada. Esta entidad, en adelante controlador, comprende el conocimiento de toda la topología, y puede enviar y recibir mensajes de cada nodo emulado/simulado.

Además de este análisis con enfoque en los protocolos y sus comportamientos, existen entornos donde se desarrolla la experimentación que pueden contar con herramientas que ayuden a solucionar este problema. Las mismas serán descritas en la sección 3.2.4. Finalmente, cabe mencionar que para el caso simulado el controlador se puede asociar tanto al programa principal como a un nodo de la simulación.

3.2.1. Tablas previamente calculadas

El primer método para determinar la convergencia de enrutamiento se implementa mediante el Algoritmo 1, el cual está inspirado en la herramienta Sibyl RT Calculator (parte del framework Sibyl). Este algoritmo centraliza el cómputo en el controlador, quien puede interactuar con los nodos de la topología para, por ejemplo, solicitarle información. Por lo tanto, el algoritmo es bastante simple: en primer lugar, el controlador, como entidad omnisciente, calcula previamente las tablas de reenvío de todos los nodos de la topología

(por ejemplo, utilizando un algoritmo de Dijkstra compatible con ECMP) y, en segundo lugar, solicita la tabla de reenvío real (actual) a cada nodo de la emulación, comparando las tablas recibidas con las esperadas (pre calculadas).

Algorithm 1 Convergencia con tablas precalculadas (enfoque centralizado)

Require: k: factor; T: Topología
Result: $True$
  for $i = 1$; $i = 5k^2/4$ **do**
    $x_i \leftarrow f(T, i, k)$ ▷ f puede ser Dijkstra
  end for
  $y_i \leftarrow obtenerTabla(i)$
  while $y_i \neq x_i$ **do**
    $y_i \leftarrow obtenerTabla(i)$
  end while

Con un enfoque descentralizado, es decir, donde el cómputo se distribuye entre todos los nodos, cada uno de estos puede comparar las tablas esperadas con las reales y notificar al controlador solo cuando son iguales, es decir, cuando él convergió.

Finalmente, se puede implementar una variación híbrida de esta estrategia, donde el controlador calcula previamente las tablas, se las envía a todos los nodos y luego espera un mensaje de finalización de cada nodo. Por supuesto, cuando el controlador es consciente de que todos los nodos han llegado a la tabla esperada (convergieron), el experimento debe finalizar.

Para implementar este criterio de parada independiente del entorno, es necesario construir un grafo representativo del fat-tree. Luego, se puede ejecutar sobre este grafo un algoritmo de enrutamiento de múltiples caminos (como un Dijkstra compatible con ECMP) para obtener las tablas de reenvío esperadas de cada nodo. Esta metodología se puede implementar utilizando la biblioteca NetworkX[1] disponible para el lenguaje de programación Python. Esta biblioteca también incluye una función que calcula todas las rutas más cortas entre cualquier par origen-destino. Por otro lado, para obtener las tablas de reenvío reales de los nodos del experimento, se debe interactuar con el emulador o simulador, por ejemplo utilizando la API para Python proporcionada por el emulador Mininet. Eventualmente, cuando todas las tablas precalculadas son iguales a las tablas reales, el programa principal que realiza la emulación detecta la convergencia del protocolo de enrutamiento y lleva a cabo la acción

[1] https://networkx.org/

correspondiente, que usualmente, consiste en detener la emulación o simulación.

3.2.2. Bandera de parada

Contrariamente al criterio de parada anterior, el criterio de bandera de parada es dependiente de cada protocolo de enrutamiento. La idea principal es que cada nodo genere una bandera cada vez que se recibe un paquete de control que esté relacionado con la convergencia del protocolo. El Algoritmo 2 abstrae la dependencia del protocolo en una función llamada *receivedControlPacket*. Esta función debe devolver el valor *True* solo cuando el nodo recibe un paquete de control relacionado con la convergencia. Estos paquetes de control varían según el protocolo utilizado de la siguiente manera:

- BGP: la función debe considerar solamente los paquetes de tipo Update.
- RIFT: solo se deben considerar los paquetes de tipo Topology Information Element (TIE).
- IS-IS, Openfabric: los paquetes considerados para estos protocolos deben ser los Link State PDU (LSP).

Observar que el Algoritmo 2 tiene una vista local de cada nodo, es decir, cada nodo de la topología debe ejecutar este algoritmo, y a su vez, debe existir un controlador global esperando por la notificación de convergencia de cada nodo. También, debe tenerse en cuenta que la eficiencia y eficacia de esta heurística depende del valor seleccionado para el umbral. Este umbral está predefinido, y representa el tiempo máximo dispuesto a esperar por otro paquete de control antes de asumir la convergencia. Si no llega ningún paquete de control durante este tiempo, la tabla de reenvío del nodo se considera estable y se notifica al controlador.

3.2.3. Ventana deslizante

El concepto de ventana deslizante representa la idea de recorrer un conjunto de paquetes de control con un tamaño de paso fijo. Determinar cuándo un protocolo ha convergido usando ventanas deslizantes hereda algunas ideas del Algoritmo 2. En primer lugar, también es dependiente del protocolo. Esto

Algorithm 2 Convergencia con bandera de parada

Require: t: threshold;
  $flag \leftarrow True$
  while flag **do**
    sleep(t)
    $flag \leftarrow receivedControlPacket()$
  end while
  notifyEnd()

se representa mediante el uso de la función *countRelevant* presente en el Algoritmo 3. Esta función tiene las mismas consideraciones mencionadas para el método anterior, detallado en la sección 3.2.2 para la función *receiveControlPacket*, es decir, su comportamiento depende del protocolo de enrutamiento que se esté considerando. En segundo lugar, se utiliza el mismo concepto de umbral. En este caso, es el promedio de los paquetes de control relacionados con la convergencia sobre todos los paquetes de control recibidos. En consecuencia, definiendo este umbral se establece el número mínimo de paquetes de control relacionados con la convergencia requeridos en cada ventana (paso) para determinar si el protocolo convergió.

Algorithm 3 Convergencia con ventana deslizante

Require: s: window size; t: threshold;
  while $average > t$ **do**
    $x \leftarrow countRelevant(receiveControlPacket(s))$
    $average \leftarrow x/s$
  end while
  notifyEnd()

3.2.4. Otros enfoques

Más allá de las heurísticas y técnicas mostradas anteriormente para determinar cuando un protocolo de enrutamiento ha convergido, cada entorno de emulación puede proveer de herramientas clásicas para el diagnóstico de redes (p.e., ping) que, combinadas con estrategias de configuración sobre los nodos emulados, pueden ayudar en la tarea.

En definitiva, se debe definir una matriz de alcance entre todos los prefijos difundidos por el protocolo de enrutamiento elegido. Luego, utilizando la herramienta de red ping, se debe verificar el alcance entre todas las com-

binaciones representadas en la matriz antes mencionada. Este enfoque está presente en Mininet bajo la utilidad de *pingall*, donde la matriz contiene las combinaciones de alcance entre todos los nodos de tipo "host" definidos en la emulación. Desafortunadamente, debido a la escala que requieren las emulaciones de topologías fat-tree en el entorno MSDC, se suele emular solo la topología de red, evitando destinar recursos para emular los hosts. En este sentido, un enfoque práctico es definir en todos los nodos de tipo Leaf una interfaz de *loopback*, distribuirla oportunamente con el protocolo de enrutamiento, construir una matriz de alcance, y finalmente, ejecutar la herramienta ping para cada combinación presente en la misma.

Si bien este enfoque es más pragmático y con menos garantías que los anteriores, puede ser un enfoque útil en etapas preliminares de la experimentación o incluso cuando se trabaja a escalas pequeñas.

3.3. Métricas independientes del tiempo

Medir el tiempo de convergencia no es una métrica válida en un entorno emulado, ya que cada ejecución depende no solo de la emulación, sino también de procesos en segundo plano (p.e., sistema operativo, hipervisor), que pueden alterar el tiempo total transcurrido. Una buena métrica para comparar diferentes protocolos de enrutamiento es considerar la cantidad de tráfico del plano de control intercambiado hasta la convergencia. Contar con una metodología diseñada para probar las implementaciones de protocolos de enrutamiento en topologías fat-tree adoptando un método independiente del tiempo, permite normalizar los resultados de diferentes protocolos e implementaciones de enrutamiento, independientemente del entorno de ejecución (es decir, del hardware subyacente).

Las métricas adoptadas en esta metodología [77] buscan agregar información de alto nivel acerca del comportamiento de las implementaciones de los protocolos de enrutamiento siendo independientes, en la medida de lo posible, del tiempo real y de las particularidades de cada implementación. De esta manera, se definen como métricas: la carga de mensajería de control inyectada por un protocolo ante los eventos definidos en los casos de uso (sección 3.3.1), una medida de localidad para poder evaluar que tan lejos viajan las "noticias" luego de un evento (sección 3.3.2) y, finalmente, una métrica que da una idea de la cantidad de iteraciones (o rondas) que necesita una implementación para

volver a un estado estable luego de un evento dado (sección 3.3.3).

3.3.1. Carga de la mensajería

Medir la carga de paquetes de control inyectada por cada protocolo de enrutamiento es un punto de comparación interesante a la hora de evaluar su comportamiento. Notar que esta medida es independiente del tiempo y no se ve afectada por el entorno de emulación o simulación.

El propósito de esta métrica es determinar si la sobrecarga de paquetes de control introducida por cada protocolo crece de manera suave con respecto a las características principales de la topología: los valores de k (y eventualmente R si se trabaja con topologías con redundancia) y el número de nodos e interfaces. Para medir la sobrecarga inyectada por la implementación de un protocolo se pueden adoptar dos métricas diferentes: 1) contar los paquetes originados por una prueba, y 2) calcular su tamaño agregado. Al calcular el tamaño agregado, solo se debe considerar la carga útil del protocolo.

3.3.2. Localidad

La localidad es una métrica que permite evaluar la capacidad de una implementación para limitar el "radio de explosión" o "*blast radius*"de una falla o una recuperación. Para calcular la localidad se debe seguir el siguiente algoritmo:

1. Para cada enlace de la red, se calcula el número de paquetes recibidos por las interfaces de ese enlace.

2. Se asigna una distancia topológica a cada enlace de la red, calculada como la distancia desde el evento (falla o recuperación). Si el evento involucra a un nodo, todos los enlaces incidentes al mismo tienen una distancia topológica de 0. Por el contrario, si el evento involucra un enlace entre el nodo u y el nodo v, donde el nodo u es el de menor nivel en la topología, entonces todos los enlaces incidentes a u, incluido el enlace fallido/recuperado, tienen una distancia topológica 0. En ambos casos, para todos los demás enlaces, la distancia topológica es el número mínimo de enlaces que deben atravesarse para llegar a un enlace con una distancia topológica 0.

3. Se calcula el número total de paquetes recibidos en todas las interfaces que tienen una cierta distancia topológica. El resultado es un vector L, donde $L[i]$ con $i = 0, 1, 2, 3$ es el número de paquetes capturados a la distancia i.

Finalmente, se convierte la salida L en un valor escalar (llamado *blast radius*) haciendo el producto escalar entre L y el vector W donde $W[i] = i + 1$ con $i = 0, 1, 2, 3$, dándole un mayor peso a los paquetes que están lejos del evento.

3.3.3. Rondas

Si bien calcular la Localidad permite medir el "radio de explosión" de un evento, por la forma en que se calcula la misma, brinda información incompleta sobre cómo se propaga la "ola de la explosión". Por ejemplo, supongamos que los enlaces a cierta distancia topológica reciben una cantidad total de m paquetes, la Localidad presentada anteriormente no brinda ninguna información sobre la cantidad de "rondas" en las que esto sucede: los m paquetes podrían recibirse todos al mismo tiempo, o en diferentes ciclos ("olas").

Sin embargo, considerar la sucesión de olas ignorando, en la medida de lo posible, el tiempo real, es gran un desafío. Para esto, la metodología definida explota un grafo nodo-estado, que describe cómo se sincronizan los estados de los nodos mediante los paquetes de control de los protocolos de enrutamiento.

Supongamos que tenemos un grafo dirigido G, relacionado con un experimento, tal que un vértice de G es un par $<v_i, s_j>$ que representa a un nodo v_i del la topología en un estado s_j. Una arista de un vértice $<v_i, s_j>$ a otro vértice $<v_h, s_k>$ representa el hecho de que un paquete que sale del nodo v_i cuando está en el estado s_j contribuye a cambiar el estado del nodo v_h a s_k. Con esta representación, si después del experimento la implementación converge, entonces G es acíclico.

A diferencia de la Localidad, el grafo G construido con esta metodología daría una descripción clara de las relaciones causa-efecto entre los cambios de estado de los nodos. Además, se podría extraer información útil de G: por ejemplo, al calcular el número de sus vértices, se podría estimar el número de cambios de estado inducidos por un evento. Además, calcular la ruta más larga de G (factible dado que G es acíclico) daría el número de rondas que una implementación tardó en converger después de un evento, logrando una nueva

medida de eficiencia que es independiente del tiempo.

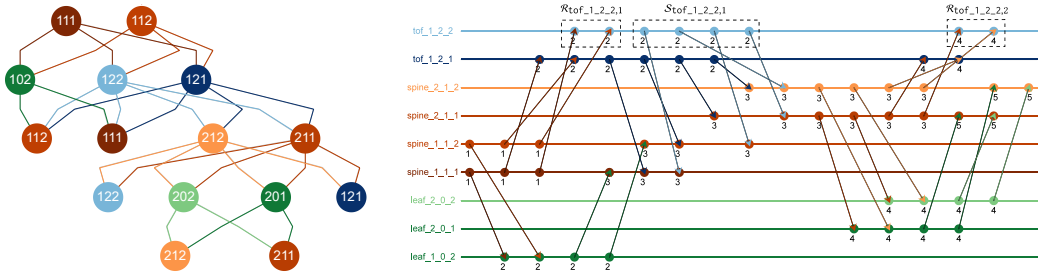
Desafortunadamente, computar G de la manera descrita, requiere tener como entrada características de los protocolos y sus implementaciones; algo que, en la medida de lo posible, se quiere evitar. En consecuencia, se computa G "desde afuera", es decir, observando los paquetes intercambiados por los nodos. Para esta tarea, se sigue el siguiente algoritmo:

1. Se utiliza el reloj del sistema local a cada nodo para ordenar los paquetes enviados/recibidos por/desde ese nodo. Sea n el número de nodos de la topología. Para cada nodo v_i con $i = 1 \dots n$, se ordenan los paquetes que entran o salen de v_i según la hora local en la que se registraron, obteniendo una lista ordenada λ_i. La Figura 3.1(b) muestra las listas ordenadas que llevaron al cómputo del grafo mostrado en la Figura 3.1(a). Este tipo de diagrama se lo denominó línea de tiempo nodo-estado o *node-state timeline*. Cada lista λ_i es una secuencia de puntos a lo largo de la línea horizontal que representa al tiempo asociado con el nodo v_i correspondiente. Cada paquete p está representado por: I) un punto en la línea del nodo que envía p, II) un punto en la línea del nodo que recibe p, y III) una flecha conectando esos dos puntos. Si existen, se llama $pred(p, \lambda_i)$ al predecesor del paquete p en la lista λ_i y $succ(p, \lambda_i)$ al sucesor.

2. Se divide cada λ_i en listas de conjuntos máximos de paquetes entrantes o salientes consecutivos, llamados *grupos de recepción* y *grupos de envío* de v_i. Por lo tanto, se puede escribir $\lambda_i = \mathcal{R}_{i,1}\mathcal{S}_{i,1} \dots \mathcal{R}_{i,h_i}\mathcal{S}_{i,h_i}$, donde $\mathcal{R}_{i,j}$ y $\mathcal{S}_{i,j}$ $(j = 1, \dots, h_i)$ son los grupos de recepción y envío de v_i respectivamente, y donde $\mathcal{R}_{i,1}$ y $\mathcal{S}_{i,h_i}$ pueden estar vacíos.

   Por ejemplo, en la Figura 3.1(b), el nodo `tof_1_2_2` tiene un grupo receptor $\mathcal{R}_{\texttt{tof\_1\_2\_2},1}$ que contiene los dos primeros paquetes, un grupo emisor $\mathcal{S}_{\texttt{tof\_1\_2\_2},1}$ con los siguientes cuatro paquetes y un grupo receptor $\mathcal{R}_{\texttt{tof\_1\_2\_2},2}$ que contiene los últimos dos paquetes. En este caso se puede observar que $\mathcal{S}_{\texttt{tof\_1\_2\_2},2} = \emptyset$. Por otra parte, un ejemplo de nodo cuyo primer grupo receptor está vacío es `spine_1_1_1` que tiene $\mathcal{R}_{\texttt{spine\_1\_1\_1},1} = \emptyset$.

3. Se asocia a cada lista λ_i una secuencia de estados: (1) Si el primer grupo de λ_i es un grupo emisor $(\mathcal{S}_{i,1})$ se le asocia un estado s_1 a v_i. (2) Para cada par $\mathcal{R}_{i,j}$ y $\mathcal{S}_{i,j}$, se le asocia un estado s_j a v_i. (3) Si el último grupo de λ_i es un grupo receptor $(\mathcal{R}_{i,h_i})$ se le asocia un estado s_{h_i} a v_i. Como



(a) Grafo nodo-estado.

(b) Línea de tiempo nodo-estado. Muestra los paquetes intercambiados durante el experimento.

Figura 3.1: Reacción de la implementación de BGP ante una falla de un nodo Leaf, nombrado `leaf_1_0_1`, en una topología fat-tree con $K_{LEAF} = 2$, $K_{TOP} = 2$ y $R = 2$ [77].

ejemplo, el nodo `tof_1_2_2` tiene dos estados, s_1 asociado con $\mathcal{R}_{\texttt{tof\_1\_2\_2},1}$ y $\mathcal{S}_{\texttt{tof\_1\_2\_2},1}$, y s_2 asociado solo con $\mathcal{R}_{\texttt{tof\_1\_2\_2},2}$.

4. Con los estados definidos anteriormente, se construye un grafo dirigido G de la siguiente manera: Cada vértice de G es un par $\langle v_i, s_j \rangle$, donde el primer elemento es un nodo de la topología y el segundo, un estado.

   Para todos los $\mathcal{S}_{i,j}$, se tiene que cada paquete p es enviado desde un nodo v_i a uno v_k de la topología. G tiene una arista $(\langle v_i, s_j \rangle, \langle v_k, s_g \rangle)$ donde g es el índice del grupo receptor correspondiente $\mathcal{R}_{k,g}$ de v_k que contiene p (es decir, $p \in \mathcal{R}_{k,g}$).

   Como ejemplo, el estado s_1 asociado al nodo `tof_1_2_2` que corresponde al vértice $\langle \texttt{tof\_1\_2\_2}, s_1 \rangle$, tiene dos aristas entrantes, que representan los paquetes de $\mathcal{R}_{\texttt{tof\_1\_2\_2},1}$, y cuatro salientes que representan los paquetes de $\mathcal{S}_{\texttt{tof\_1\_2\_2},1}$. Por otra parte, el vértice $\langle \texttt{tof\_1\_2\_2}, s_2 \rangle$ tiene dos aristas entrantes y ninguna saliente, ya que el grupo receptor $\mathcal{R}_{\texttt{tof\_1\_2\_2},2}$ es el último grupo en $\lambda_{\texttt{tof\_1\_2\_2}}$.

   Este ejemplo se puede observar en la Figura 3.1(a), donde los nodos $\langle \texttt{tof\_1\_2\_2}, s_1 \rangle$ y $\langle \texttt{tof\_1\_2\_2}, s_2 \rangle$ se corresponden a los dos vértices color celeste etiquetados con `122`. A este grafo G, se le denomina *grafo nodo-estado* o *node-state graph*.

Ahora, se tiene el cálculo del número de rondas para cada vértice v de G, como la longitud del camino más largo desde un origen de G a v. Esto representa el número de cambios de estado que llevaron a v y se lo denomina

como el *valor ronda-estado* de v. Además, considerando todos los vértices de G, se puede calcular el máximo del valor ronda-estado, el cual se denomina *valor ronda-estado* de G.

En la Figura 3.1(a), la longitud del camino más largo y, por lo tanto, valor ronda-estado del grafo es 4. La Figura 3.1(b) muestra el resultado del cálculo realizado por el algoritmo. La etiqueta de cada paquete p es el número de ronda asignado. Observar que el valor de ronda máximo es 5, ya que se calcula considerando el número de vértices en lugar del número de aristas.

Si bien el cálculo de esta métrica tiene la independencia deseada en cuanto al tiempo y las particularidades de cada implementación, hay que tener en cuenta que los pasos que toma una implementación para converger y el conjunto de paquetes que intercambian los nodos pueden cambiar según los parámetros de la red, el tiempo de procesamiento de los nodos y la presencia de temporizadores en los protocolos. Eso se traduce a que, para un experimento dado, el grafo nodo-estado y los valores ronda-estado no están definidos unívocamente y pueden depender del momento en que se den los eventos. En este sentido, para analizar el comportamiento de una implementación utilizando esta métrica, se deberán realizar varias ejecuciones del mismo experimento.

3.4. Análisis de resultados

Naturalmente, marcado por el objetivo de esta tesis de tener un análisis experimental en diversos entornos de emulación del enrutamiento en redes fat-tree, el análisis de resultados tiene también, dos dimensiones.

Por un lado, se deben analizar los resultados de la validación de los entornos de emulación/simulación. Este análisis cuando el resultado es favorable (el protocolo convergió) o a escalas pequeñas de la topología ($k <= 8$) es manejable o incluso trivial, pero a grandes escalas, deriva en un análisis de *logs* y capturas de tráfico con grandes volúmenes de datos. Para estos casos la metodología es seguir un análisis *post mortem*, creando programas para analizar los resultados obtenidos luego de terminar la emulación: *logs* de los demonios de los protocolos de enrutamiento, estado de las tablas de reenvío de cada nodo, interfaces configuradas y sus numeraciones, configuración del protocolo, capturas de tráfico, revisión de código, entre otras.

Por otra parte, para analizar el comportamiento de los protocolos ante los eventos definidos en los casos de uso, la etapa de análisis de resultados comienza

luego de verificar que, una vez de ejecutado el caso de uso seleccionado, la red alcanza el estado esperado, es decir, el protocolo logró la convergencia. Luego de esto, se utilizan las métricas definidas anteriormente para generar resultados. Como base, se mide la carga de mensajería inyectada en la red, contando los paquetes de control relacionados a la convergencia (sección 3.3.1). Luego, si se quiere obtener resultados que reflejen que tan "local" a donde ocurrió el evento es la sobrecarga de dicha mensajería, se puede calcular la Localidad (sección 3.3.2). Finalmente, se puede determinar el número de Rondas usadas por la implementación para converger (sección 3.3.3). Para ser independiente de las operaciones internas de las implementaciones, los análisis de carga de mensajería, localidad y número de rondas se basan únicamente en los paquetes de señalización del plano de control capturados durante los experimentos. Dichos paquetes tienen diferentes características. Mientras que algunos se inyectan en la red como consecuencia de un evento (p.e., paquetes de tipo Update de BGP), otros se intercambian independientemente de los eventos que ocurren en la red (p.e., paquetes de tipo Keepalive de BGP). Para analizar la reacción de un protocolo de enrutamiento a un evento específico, el análisis de esta metodología solo considera los paquetes que se intercambian como consecuencia del mismo, es decir, que estén relacionados a la convergencia del protocolo. En la sección 3.2.2 se especifican los tipos de paquetes a considerar para los protocolos analizados en esta tesis, es decir, BGP, RIFT, IS-IS y OpenFabric.

3.5. Ambientes de experimentación

Los entornos de emulación y simulación donde se lleva a cabo la experimentación son los presentados en la sección 2. Los ambientes de experimentación utilizan estos emuladores/simuladores en conjunto con herramientas, para poder llevar acabo la metodología de experimentación planteada. Es decir, generar topologías de tipo fat-tree, desplegar los protocolos de enrutamiento, ejecutar los casos de uso diseñados y recolectar los resultados para su análisis.

Además, se presentan los ambientes de experimentación para emular un fat-tree como una red de tipo SDN, en la que el papel que juegan los protocolos de enrutamiento aquí estudiados, son sustituidos por un controlador SDN centralizado.

En el resto de esta sección se presenta como metodología la preparación y adecuación necesaria de los ambientes requerida para realizar la experimenta-

ción.

3.5.1. Mininet

Para construir un ambiente de experimentación que utilice a Mininet como emulador de la red, lo primero que se debe hacer es instalar en un servidor este emulador en conjunto con las herramientas y protocolos de enrutamiento a utilizar. Dado que Mininet está desarrollado para ambientes Linux, para contar con implementaciones de BGP (con sus configuraciones para el data center), IS-IS y Openfabric, se instala la suite de protocolos FRR. Además, para contar con una implementación de RIFT se instala la implementación de código abierto rift-python [30].

Por otra parte, se desarrolló un generador de topologías fat-tree que toma como entrada el valor del parámetro k y genera los nodos emulados para la topología, sus conexiones y realiza una numeración IP acorde. La nomenclatura de los nodos de la emulación sigue la siguiente convención: `r{nivel}0{índice}`, así, el primer nodo Leaf generado será el `r30001`, el primer Spine el `r20001` y el primer Core el `r10001`. Para la numeración IP de los enlaces punto a punto entre los nodos de red (routers) se utilizan prefijos /30 del rango 10.0.0.0/8. Mientras que para los enlaces Leaf-Host, de estar presentes, se utilizan prefijos /24 del rango 200.0.0.0/8. Este generador está basado en el desarrollado en [80] y se desarrolló utilizando la API de Python [81] que brinda Mininet.

Si bien con el generador de la topología, y la suite de protocolos instaladas ya es posible emular un fat-tree y manualmente configurar y ejecutar los protocolos, esto resulta impracticable. En este sentido, se incorporó al generador una capa de configuración que toma toda la información disponible en la estructura de datos de Mininet (nomenclatura de nodos, interfaces, numeración IP, entre otras) para generar un archivo de configuración para cada nodo de la emulación. Esta capa de configuración es dependiente del protocolo por lo que se agregó una capa por cada protocolo a desplegar.

Finalmente, para completar el ambiente de experimentación se programa el inicio automatizado de los demonios perteneciente a los protocolos de enrutamiento, la captura de tráfico y la ejecución de los casos de uso mencionados. En este ambiente además, se desarrolló la propuesta de criterio de parada para la convergencia del protocolo BGP presentada en la sección 3.2.1.

3.5.2. MaxiNet

Como se detalló en la sección 2, se puede ver a MaxiNet como una versión distribuida de Mininet. Por esta razón, la API para desarrollar programas para MaxiNet tiene similitudes y correspondencias con la API de Mininet antes presentada. En este sentido, se desarrolló un generador de topologías fat-tree similar al desarrollado para Mininet, cambiando las llamadas a la API de Python y agregando particularidades en la configuración. Análogamente se implementaron la automatización de la configuración de protocolos de enrutamiento (en este caso para BGP), los casos de uso y la recolección de datos para el análisis. La nomenclatura de los nodos emulados y la numeración IP, heredan las convenciones tomadas para Mininet.

En cuanto al entorno de ejecución, se instaló FRR y una versión modificada de UltraViolet (la versión actualizada de MaxiNet). Esta versión modificada del entorno fue desarrollada en el marco de esta tesis, y agrega al programa original la capacidad para guardar meta datos acerca de la distribución de la emulación y de la creación de túneles, así como comandos para consultar las tablas de reenvío de los nodos emulados. Estos meta datos son necesarios para la automatización de la configuración de los protocolos de enrutamiento, ya que los enlaces virtuales que antes utilizaba Mininet, en el caso en que dos nodos sean emulados en distintos servidores, son sustituidos por túneles (y sus interfaces renombradas).

3.5.3. Framework Sibyl

Para tener un ambiente de experimentación que utilice como emulador a Kathará o Megalos, basta con instalar los mismos en un ambiente adecuado y utilizar el framework Sibyl [82][77].

Sibyl aprovecha un conjunto de herramientas para implementar y ejecutar automáticamente las pruebas propuestas en la sección 3.1. Durante cada experimento, se realizan los siguientes pasos: 1) se genera una topología fat-tree; 2) se despliega un conjunto de contenedores y de enlaces virtuales que representan la topología; 3) en cada nodo emulado, se capturan todas las PDU generadas en todas las interfaces; 4) se inician los demonios de los protocolos de enrutamiento en todos los nodos con la configuración adecuada; 5) se realiza la comprobación de convergencia; 6) se realizan las acciones requeridas para ejecutar un caso de uso concreto (p.e., link failure); 7) se vuelve a realizar la

comprobación de convergencia; 8) se termina la emulación; y 9) se realiza el análisis y se generan los resultados.

A diferencia de Mininet y MaxiNet, Kathará (o su versión distribuida Megalos) no cuenta con una API de programación que permita controlar el experimento desde un programa centralizado. Es por esto que el framework Sibyl combina una variedad de herramientas que realizan estas tareas interactuando con la emulación (o generando los elementos necesarios para la misma) mediante los comandos disponibles en el emulador. Estas herramientas son:

- *VFTGen* [83]: genera y configura automáticamente topologías fat-tree single-plane y multi-plane. Toma como entrada los parámetros (K_{LEAF}, K_{TOP} y R) y genera como salida un directorio que contiene todos los archivos necesarios para ejecutar la topología correspondiente en Kathará.

- *RTcalculator*: es una herramienta para generar las tablas de reenvío de un fat-tree en base a información de la topología. Toma como entrada una topología, un protocolo de enrutamiento y un tipo de prueba, y devuelve la información de enrutamiento para cada nodo del fat-tree.

- *Sibyl Agent*: es un servicio Representational State Transfer (REST) compuesto por un controlador y varios agentes. Se implementa una instancia de tipo agente en cada nodo del fat-tree. El agente del controlador puede realizar acciones en los nodos a través de los agentes de los nodos (por ejemplo, provocar una falla en una interfaz).

- *Sibyl Analyzer*: es una herramienta para analizar y graficar los resultados de los experimentos. Toma como entrada las capturas que contienen los paquetes intercambiados por los nodos durante un experimento. Calcula las métricas presentadas en la sección 3.3 y las guarda en un archivo `.json`.

3.5.4. Ambiente ns-3

Este ambiente de experimentación está construido en el simulador ns-3 presentado en la sección 2. Si bien este ambiente, a diferencia del resto de los presentados en este trabajo, es un ambiente simulado, utiliza el modo DCE que provee el simulador ns-3. Utilizar el modo DCE permite poder proveer al entorno de implementaciones reales de los protocolos de enrutamiento, pudiendo experimentar con el mismo código que en un emulador. Esto logra, por

ejemplo, tener un ambiente para experimentar con protocolos de enrutamiento de la suite FRR, pero con las ventajas de utilizar un simulador por sobre un emulador.

Para poder utilizar una implementación de los protocolos de enrutamiento de FRR sobre ns-3 DCE, se debe realizar un porte de dicha implementación, ya que su pasaje no es directo. Este porte fue implementado por estudiantes de grado en el marco de un proyecto de fin de carrera (acotado a la validación de la implementación del protocolo BGP), el cual se puede consultar en [84]. En dicho proyecto además, inspirados en la metodología antes presentada, se desarrolló un ambiente de experimentación adecuado que incluye: un generador de fat-trees, configuración automatizada del protocolo de enrutamiento, recolección y análisis de los resultados. Este ambiente, junto con el porte, están disponibles con acceso libre en [85]. En el marco de esta tesis se desarrolló e integró a este ambiente el caso de uso de *Bootstrap*, utilidades para hacer más eficiente el análisis de paquetes, un chequeo de convergencia *post mortem* inspirado en el método presentado en la sección 3.2.3, y mejoras en los tiempos de simulación.

3.5.5. Ambiente sobre CORE

Al igual que Mininet, CORE está desarrollado para sistemas Linux, por lo que el primer paso para tener un ambiente de experimentación que utilice a este emulador es instalarlo en un ambiente de ejecución Linux junto con la suite de protocolos FRR y rift-python.

Si bien CORE brinda una interfaz gráfica lista para diseñar y configurar un fat-tree, dada la escala de los experimentos, un ambiente con estas características resulta de poca utilidad. Para poder escalar los experimentos y automatizar la configuración de los protocolos de enrutamiento se utiliza la *Core Python API* [86]. Con este enfoque, la metodología es similar a la explicada para Mininet, es decir, la topología y las configuraciones para la experimentación siguen un código Python. Este código genera una topología fat-tree basada en un k determinado, genera todas las configuraciones para los demonios de los protocolos de enrutamiento (BGP, IS-IS, Openfabric o RIFT) y orquesta la emulación, es decir, lanza todos los procesos necesarios en los nodos, captura los paquetes y ejecuta los casos de uso.

A la fecha del desarrollo de este entorno de experimentación, el uso de esta

API estaba poco documentado. De todas maneras, se logró tener un ambiente de experimentación que cumple con la generación de fat-trees en base al parámetro k y el despliegue de los protocolos de la suite FRR así como de la implementación rift-python.

3.5.6. Ambientes SDN

Para el caso de un ambiente de experimentación con una red de tipo SDN se utiliza como emulador base a Mininet y como controladores a POX y Ryu (descritos en la sección 2). En este caso al no tener que desplegar un protocolo de enrutamiento en los nodos, el ambiente de ejecución solo necesita tener instalado Mininet (que va a emular la topología) y el controlador SDN. El ambiente de experimentación en este caso busca validar el despliegue de topologías fat-tree, donde las rutas entre los hosts y tablas de reenvío se calculen centralizadas en un controlador.

En este sentido, se debe programar un generador de la topología como el descrito para Mininet sin agregar la capa de configuración de los protocolos de enrutamiento. Esto es independiente del controlador, salvo algunas modificaciones particulares que le indican a la emulación que se utilizará un controlador remoto y que los routers emulados deben "hablar" la versión de Openflow [38] adecuada. Luego, se debe programar el controlador con las funcionalidades necesarias para lograr conectividad de capa 3 en el fat-tree.

3.5.6.1. Ambiente con controlador POX

Para el despliegue del controlador POX, se desarrolló un generador y un par de controladores inspirados en el proyecto disponible en [87]. Si bien es factible utilizar el generador de fat-trees desarrollado para Mininet con algunas modificaciones, el desarrollado contiene funciones de utilidad para asignar/obtener Datapath ID (DPID)s, direcciones IP y direcciones Media Access Control (MAC) que ayudan a la tarea del controlador. También se le agregó la generación de una representación con grafos de la topología. La numeración IP solo de realiza a nivel de los servidores y se utiliza el rango 10.0.0.0/8.

El controlador cuenta con los siguientes módulos:

- Un módulo denominado *fakeARP* tomado de [87] que se encarga de atender las consultas Address Resolution Protocol (ARP) que se generen.

- Un módulo denominado *controller install*, que toma la representación en un grafo de la topología, calcula todas las rutas utilizando Dijkstra multi-camino y finalmente, cuando un switch se conecta al controlador, le instala su tabla previamente calculada.

- Un módulo denominado *controller route*, que toma la representación en un grafo de la topología, calcula y almacena todas las rutas utilizando Dijkstra multi-camino y luego resuelve el reenvío paquete a paquete, es decir, cada switch de la topología va a tener su tabla de flujo (o reenvío) vacía y le va a consultar al controlador cada vez que reciba un paquete qué acción tomar.

Notar que para tener un ambiente de experimentación funcional basta con utilizar el generador de la topología, el módulo *fakearp* y uno entre el módulo *controller install* y el *controller route*.

Si bien, el objetivo con este enfoque es validar el ambiente de experimentación, estos controladores fueron provistos además, con una función que atiende las caídas de los enlaces, para poder implementar los casos de uso de la metodología.

El diagrama mostrado en la Figura 3.2 muestra el flujo de la comunicación entre un nodo de tipo Leaf con el controlador cuando este utiliza el módulo *controller install*. En este caso, la interacción 1 corresponde al aviso de llegada del nodo al controlador, la 2 y 3 corresponden a los módulos instalándole al nodo entradas en su tabla de flujo. Con 2 se le indica que reenvíe al controlador cualquier mensaje de tipo ARP y con la 3 se le instala la tabla de reenvío previamente calculada por el módulo. Finalmente, y por el resto del experimento, cada vez que a un nodo de tipo Leaf le llegue un paquete ARP proveniente de un host, va a interactuar con el módulo *fakeARP* para resolver la consulta.

Notar que los nodos de otros niveles de la topología ignoran la interacción 5 y 6 debido a que, al no generar tráfico hacia los hosts, no deben interactuar con el módulo que resuelve el ARP. Finalmente, notar que la interacción 3 puede darse de manera espontánea desde el controlador, cuando existan eventos de fallas (enlaces o nodos).

Por otra parte, la Figura 3.3 muestra el mismo flujo descrito anteriormente, pero esta vez cuando el controlador utiliza el módulo *controller route*. En este caso la interacción 1 y 2 son análogas a la del caso anterior, luego como el controlador no le instala una tabla de flujo, cada paquete entrante al nodo va a ser

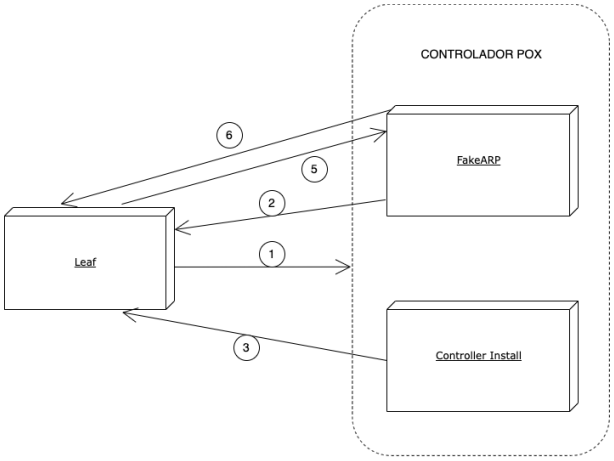


Figura 3.2: Interacción de un nodo de tipo Leaf de la emulación con el controlador POX utilizando el módulo *controller install*.

reenviado al controlador y este le va a indicar que acción tomar (interacciones 5 y 6). Las interacciones 3 y 4 también son análogas al caso anterior. Una vez que las tablas ARP de los hosts debajo del Leaf estén pobladas, la interacción 3 y 4 no se va a ejecutar continuamente. Finalmente, notar que en este caso un evento de falla no genera una interacción del controlador hacia los nodos debido a que es el controlador el que tiene las tablas de reenvío de todos los nodos.

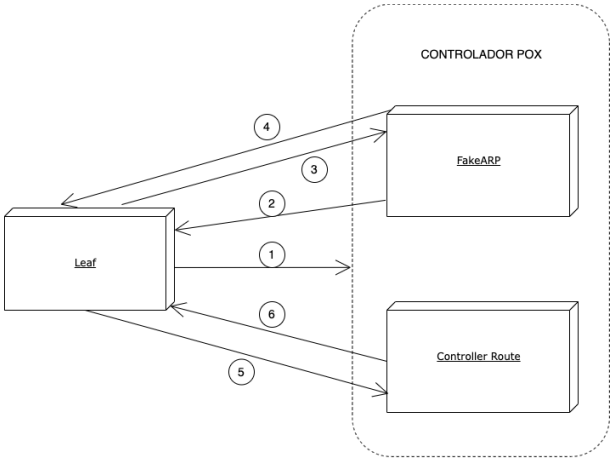


Figura 3.3: Interacción de un nodo de tipo Leaf de la emulación con el controlador POX utilizando el módulo *controller route*.

3.5.6.2. Ambiente con controlador Ryu

Para el despliegue con el controlador Ryu se utilizó el proyecto PureSDN [88] con modificaciones mínimas para su despliegue en ambientes de ejecución actualizados. Este proyecto brinda la configuración necesaria para desplegar un fat-tree con $K = 4$ y otro con $K = 8$. El controlador brindado por este proyecto calcula e instala todas las rutas. Además, incluye un conjunto de aplicaciones Ryu que recopilan información básica de la red, como la topología y el ancho de banda libre de los enlaces. PureSDN puede lograr que el reenvío se realice utilizando el enfoque tradicional de la ruta más corta o basado en el ancho de banda disponible.

3.6. Consideraciones finales

En este capítulo se presentó la metodología de trabajo y los ambientes de experimentación que implementan dicha metodología. Esta metodología define los principales casos de uso, criterios de parada necesarios para determinar la convergencia de un protocolo de enrutamiento, métricas independientes de tiempo para tener resultados fiables sobre ambientes emulados y la definición del análisis de los resultados. Los ambientes de experimentación se basan en emuladores/simuladores de red para poder llevar acabo la metodología planteada. Para esto se definieron seis ambientes de experimentación diferentes que logran generar topologías fat-trees, desplegar protocolos de enrutamiento, ejecutar los casos de uso y recolectar los resultados. Cada uno de estos ambientes se vale de distintos emuladores/simuladores y tecnologías para poder experimentar con redes fat-trees arbitrarias representativas de un MSDC. Construir cada uno de ellos requiere resolver importantes desafíos ingenieriles/tecnológicos.

Capítulo 4

Experimentación y presentación de resultados

La experimentación realizada cubre las dos dimensiones de este trabajo. Por un lado, se realizó una evaluación exhaustiva de los ambientes de experimentación tomando como base la metodología presentada y considerando dos aspectos clave: la validación y la escalabilidad. En esta etapa además, se presentan resultados de aplicabilidad y se muestra un caso de uso aplicado.

Por otra parte, se evaluó el protocolo BGP en base a la metodología presentada. Una evaluación más amplia que incluye al resto de los protocolos de enrutamiento presentados se puede consultar en [77]. Los resultados de esta etapa se obtuvieron experimentando con el framework Sibyl (emulado) y el entorno simulado sobre ns-3. Para Sibyl, se utilizaron los resultados experimentales de [77] disponibles de forma libre [1].

Finalmente, se realiza una validación de los resultados experimentales en base a un análisis teórico del comportamiento de BGP. Este análisis utiliza la regularidad de los fat-tree multi-plane que permite describir la topología en base al parámetro k (Tabla 2.1), y el comportamiento del protocolo bajo las configuraciones necesarias para su despliegue en los MSDC (sección 2.3.1).

Cabe señalar que la elección de BGP como protocolo a estudiar se justifica por la robustez y madurez de la implementación, producto de su largo camino en el ambiente industrial.

[1] https://gitlab.com/uniroma3/compunet/networks/sibyl-framework/sibyl-results

4.1. Ambiente de ejecución

Trabajar con los ambientes de experimentación descritos anteriormente requiere como base emular o simular topologías fat-tree. Esta tarea necesita, no solo de una gran cantidad de nodos emulados/simulados, sino también de una gran densidad en la conectividad. Para el caso de los ambientes emulados, además de los recursos básicos como CPU y RAM, esto se traduce en una gran cantidad de interfaces y enlaces virtuales, entre otros.

Los distintos ambientes de experimentación (y escalabilidad) se desplegaron sobre distintos ambientes de ejecución:

- Ambiente con Sibyl: Para emular topologías de hasta 320 nodos y alrededor de 4500 interfaces, se utilizó un clúster virtual local compuesto por 22 máquinas virtuales, cada una con 2 Virtual CPU (vCPU) y 8GB de Virtual RAM (vRAM). Para topologías más densas, se utilizó *Azure Cloud* con un clúster compuesto por 160 máquinas virtuales, cada una con 4 vCPU y 8GB de vRAM [77].

- Ambiente con MaxiNet: Se utilizó un clúster local compuesto por 3 máquinas virtuales, dos de ellas con 24 vCPU y 120GB de vRAM (*workers*) y otra con 4 vCPU y 4GB (principal).

- Ambientes de un único host: Mininet, Ambiente ns-3, Ambiente CORE y Kathará: Se utilizó una máquina virtual con 32 vCPU y 244GB de vRAM.

4.2. Evaluación de los ambientes de experimentación

Se realizó una evaluación exhaustiva de los ambientes de experimentación tomando como base la metodología presentada. Se consideraron dos aspectos principales: la validación, es decir, que el ambiente sea capaz de emular o simular topologías fat-tree donde la conectividad a nivel de capa de red esté dada por protocolos de enrutamiento adecuados para el data center, y, en segundo lugar, la escalabilidad, buscando los límites de cada entorno sobre el hardware subyacente. En conjunto, ambos aspectos permiten validar la construcción de

un prototipo de data center emulado con conectividad de capa de red resuelta por implementaciones reales de protocolos de enrutamiento.

Además, se discute la aplicabilidad de cada ambiente, y se presenta un caso de uso aplicado que va más allá de la infraestructura de enrutamiento: implementar una red superpuesta u *Overlay* de capa 2 sobre la red enrutada. Este caso de uso cubre uno de los requerimientos claves que tienen las DCN: la migración de máquinas virtuales conservando la numeración IP y sin perder conectividad.

4.2.1. Validación

Se definió como validado a un ambiente para el contexto data center si logra: 1) generar de manera automatizada topologías fat-tree en base a los parámetros que la describen 2) ejecutar y configurar de manera automatizada implementaciones existentes de los protocolos BGP, IS-IS, RIFT y OpenFabric 3) orquestar un experimento de manera automatizada, es decir, iniciar la emulación y todos los programas necesarios (demonios de enrutamiento, captura de tráfico, etc.), ejecutar casos de prueba y, finalmente, detener la emulación.

Por último, como requisito adicional se debe considerar el poder alcanzar una escala razonable (en cantidad de nodos y densidad de las topologías emuladas) que se asemeje a la escala de los MSDC.

- Ambiente Sibyl: puede emular topologías fat-tree single-plane y multi-plane sobre Kathará/Megalos, utilizando los protocolos de enrutamiento mencionados y distribuyendo los nodos emulados en múltiples hosts. Además, este framework genera resultados para la evaluación de los protocolos de enrutamiento utilizando las métricas presentadas en la sección 3.3 y los casos de uso presentados en la sección 3.1.

- Ambiente Mininet: el ambiente que utiliza a Mininet como emulador de la red, puede emular topologías fat-tree multi-plane en un único host. En este ambiente se pueden utilizar los protocolos implementados en la suite FRR así como cualquier implementación compatible con las distribuciones de Linux que soportan al emulador, en particular se validó rift-python. También, se validó la implementación desarrollada en este trabajo que modela los casos de uso presentados en la sección 3.1, así

como el criterio de parada basado en tablas previamente calculadas (sección 3.2.1).

- Ambiente MaxiNet: el ambiente que utiliza una extensión de Mininet como emulador de la red, puede emular topologías fat-tree multi-plane en múltiples host. En este ambiente se pueden utilizar los protocolos implementados en la suite FRR así como cualquier implementación compatible con las distribuciones de Linux que soportan al emulador, en particular se validó rift-python. También se validó la implementación de los casos de uso presentados en la sección 3.1.

- Ambiente CORE: en este ambiente se validó la emulación de topologías fat-tree multi-plane en un único host. En este ambiente se pueden utilizar los protocolos implementados en la suite FRR así como cualquier implementación compatible con las distribuciones de Linux que soportan al emulador, en particular se validó rift-python.

- Ambientes SDN: estos ambientes utilizan a Mininet como emulador de la red. En este sentido pueden emular topologías fat-tree multi-plane en un único host como en múltiples (utilizando MaxiNet). Para estos ambientes se validó la utilización de los controladores POX y Ryu para darle conectividad al fat-tree. Además, en el caso del ambiente con POX se validó la implementación de los casos de uso presentados en la sección 3.1. Notar que al cambiar el paradigma, el punto 2 considerado para validar un ambiente, se sustituye por la presencia de un controlador SDN.

- Ambiente ns-3: en este ambiente de simulación, se pueden simular topologías fat-tree single-plane y multi-plane en un único host. El porte actual de la suite FRR en ns-3 DCE contempla la correcta ejecución y auto configuración de BGP, y la incorporación de IS-IS y OpenFabric. Por otro lado, si se quiere experimentar con RIFT u otras implementaciones ajenas a la suite FRR se deberían evaluar sus portes al ambiente.

4.2.2. Escalabilidad

Las pruebas de escalabilidad de los ambientes de experimentación sobre los ambientes de ejecución se realizaron utilizando BGP como protocolo de enrutamiento. Esto es debido a la escalabilidad a nivel industrial que tienen

las implementaciones de este protocolo, las cuales superan al resto de los presentados.

En este sentido, en cada entorno se ejecutó el caso de uso denominado *Bootstrap*, con BGP como protocolo de enrutamiento para crecientes (en k) configuraciones de fat-trees. En el caso que, independientemente del método utilizado (tiempo, tablas previamente calculadas, etc) se detecta que el protocolo logra la convergencia, el ambiente se considera válido para la escala de fat-tree ejecutada. La Tabla 4.1 resume los resultados, cuantificando la cantidad de nodos e interfaces emulados con cada entorno en función del k seleccionado.

Ambiente	**k**	**Nodos**	**Interfaces**
Mininet	24	720	17280
CORE	12	180	2160
Kathará	12	180	2160
Megalos	32	1280	40960
MaxiNet	28	980	27440
ns-3	32	1280	40960

Tabla 4.1: Máxima escala obtenida de los ambientes de experimentación, utilizando BGP como protocolo de enrutamiento.

La primera observación es que la emulación distribuida supera a la de un solo host en términos de escalabilidad. Así lo demuestra el hecho de que MaxiNet y Megalos emulan con éxito más nodos que cualquiera en los que la emulación tiene lugar en un único host. Para el caso de Megalos, que supera los mil doscientos nodos, este resultado es esperable ya que éste se beneficia de la escalabilidad horizontal. Se debe considerar el compromiso entre la escalabilidad y el costo (es decir, el costo de la cantidad de servidores necesarios). Por otro lado, el entorno de ejecución donde se desplegó MaxiNet (detallado en la sección 4.1) que es notoriamente inferior en términos de recursos al anterior (Megalos: 640 vCPUs y 1280GB de vRAM versus MaxiNet: 52 vCPUs y 244GB de vRAM), logró una escala de 980 nodos emulados.

En contraste, en el lado de emulaciones de un único host, Mininet escaló hasta setecientos veinte nodos y diecisiete mil interfaces virtuales, superando ampliamente a CORE y Kathará. Esta diferencia sustancial entre Kathará y Mininet valida los resultados observados en la escala de MaxiNet (múltiples instancias de Mininet) y Megalos (múltiples instancias de Kathará) teniendo

en cuenta los recursos necesarios para lograr las escalas observadas.

Por otro lado, el único ambiente simulado logró una escala mayor a los ambientes emulados de un único host, mientras que igualó al ambiente distribuido de Megalos pero con una cantidad significativamente menor de recursos (1280GB versus 244GB de vRAM). Si bien esta comparación es válida, porque ambos ambientes producen los mismos resultados en términos de la evaluación del protocolo BGP tal como se presenta en la sección 4.3, cabe destacar que el ambiente que utiliza ns-3 está optimizado en términos de procesos que ejecuta cada nodo. Precisamente, este ambiente ejecuta el demonio FRR de BGP pero no ejecuta el de Zebra. También, notar que solo se hacen apreciaciones contemplando el recurso vRAM ya que la simulación corre en un único hilo de ejecución, y por lo tanto, utiliza una única vCPU.

Si bien en los ambientes distribuidos, los recursos de los ambientes de ejecución en términos de vRAM y vCPU limitan la escalabilidad, en ambientes de un único host, la limitante se puede dar por otros recursos (por ejemplo, la máxima cantidad de archivos abiertos por el sistema operativo) o por limitantes de la implementación de la emulación (por ejemplo, cantidad de contenedores Docker). En la Tabla 4.2 se resumen los ambientes validados, y en particular, se especifica si la limitante detectada fue en términos de "Recursos" (vRAM y vCPU) u "Otro".

4.2.3. Red superpuesta de capa de enlace en GNS3

Construir un ambiente de experimentación utilizando como simulador/emulador a GNS3 no cumple con los requisitos de validación mencionados anteriormente, pues, al momento de la escritura de esta tesis, carece de herramientas para automatizar la creación y configuración de las topologías fat-tree. Esto tiene sentido debido a que GNS3 se define como un simulador gráfico. A pesar de esto, en este trabajo se muestra como GNS3 es un entorno adecuado para enseñar o demostrar escenarios complejos de redes DCN a pequeña escala. En particular, se construyó un prototipo de data center donde la DCN cubre el requisito presentado en la sección 2.1: "Las máquinas virtuales se pueden migrar entre dos máquinas físicas sin necesidad de cambiar sus direcciones IP".

En este sentido, se construyó un fat-tree representativo para un data center de pequeña escala. Este modelo de data center se construyó utilizando imágenes de enrutadores con la suite FRR instalada y hosts simulados, lo que en la

jerga de GNS3 se denominan Virtual PC Simulator (VPCS). Para simplificar en términos de configuración, y debido a que la escala lo permite, en los nodos se utilizó la implementación de FRR para el protocolo OSPF, con el objetivo de proporcionar conectividad de capa 3 con soporte multi camino a la DCN.

Luego, y con el objetivo de proveer a esta DCN de una red superpuesta de capa 2, se desplegó la tecnología EVPN\VXLAN configurando adecuadamente la implementación de BGP presente en FRR. Con esta configuración se logró realizar con éxito las migraciones de Virtual Machine (VM)s entre PoDs en la DCN, esto es, manteniendo sus direcciones IP sin cambios. Este experimento es totalmente reproducible y está disponible en [89]. La Figura 4.1 muestra este prototipo de data center visto desde la interfaz gráfica de GNS3.

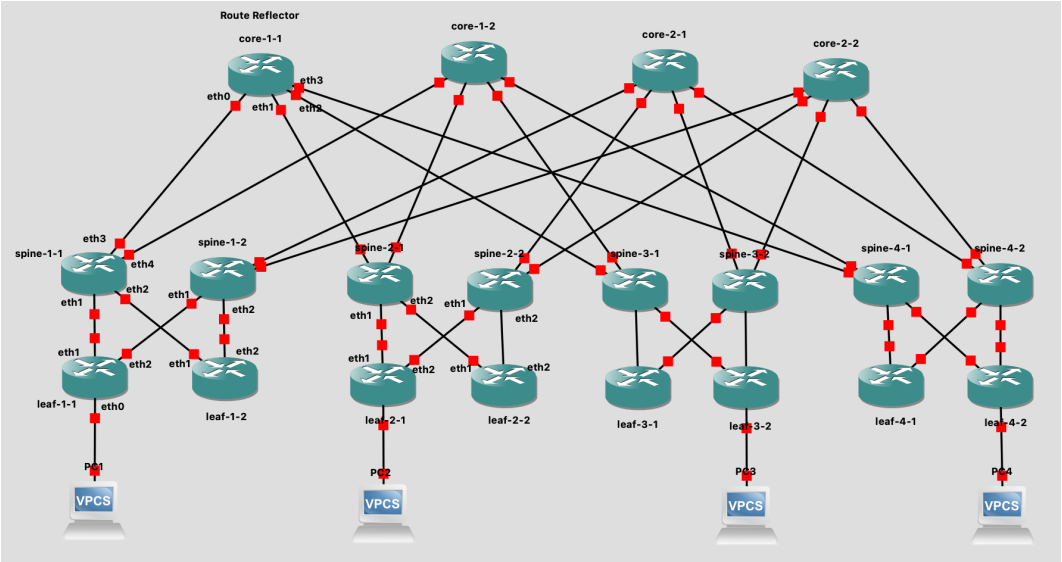


Figura 4.1: fat-tree construido en el simulador gráfico GNS3.

4.2.4. Aplicaciones

Si bien los ambientes fueron validados para la experimentación con fat-trees y protocolos de enrutamiento, cada uno puede ser más o menos conveniente dependiendo del escenario general.

Por un lado, si se quiere un ambiente emulado de gran escala y se tienen recursos limitados, el ambiente que utiliza a Mininet es el adecuado. Con este ambiente se logra una escala casi representativa para un MSDC y además cuenta con una gran flexibilidad para extenderlo y personalizarlo con gran

facilidad. El resto de los emuladores de un único host (Kathará y CORE) pueden ser adecuados para validar los experimentos a pequeña escala.

Sin embargo, Kathará con su versión distribuida Megalos, son recomendados cuando los recursos no son una limitante y se quiere tener representaciones emuladas de grandes MSDC. Además, incorporando al framework Sibyl se convierte en un ambiente sumamente adecuado para fácilmente evaluar protocolos de enrutamiento en base a la metodología propuesta.

Por otro lado, el ambiente distribuido con MaxiNet es adecuado cuando se quiere tener una gran escala, representativa para grandes MSDC con recursos limitados. Este ambiente cuenta con las mismas flexibilidades que el de Mininet.

Con respecto al ambiente simulado, este es quien logra mejor rendimiento en términos de recursos, sin embargo si se quiere experimentar con nuevos protocolos o incorporar casos de uso se debe extender este ambiente. Esto podría no llegar a ser tan flexible como en los ambientes emulados, y por esto, podría ser de más utilidad en escenarios donde se quiera evaluar en profundidad un protocolo en particular, donde se maximice el costo/beneficio de preparar el ambiente.

Finalmente, los ambientes con soporte de GUI (CORE, GNS3) son un entorno adecuado para enseñar o demostrar escenarios complejos de DCNs a pequeña escala. En la Tabla 4.2 se resumen estos aspectos de aplicabilidad en conjunto con los métodos de virtualización y limitantes de escala detectados para cada ambiente.

4.3. Evaluación del protocolo de enrutamiento BGP

La evaluación de la implementación de BGP de la suite FRR se realizó tomando tres casos de uso representativos: *Bootstrap*, *Node Failure* y *Node Recovery*. Para el caso de la falla de un nodo, se presentan los resultados experimentales para un nodo de cada nivel dentro de un PoD, es decir, se analiza la falla de un Leaf y de un Spine. Estos resultados experimentales, son comparados y validados con la realización de un análisis teórico del comportamiento del protocolo. Los escenarios de falla validados teóricamente fueron intencionalmente elegidos debido a diferencias obtenidas en los resultados experimentales.

Ambiente	Virtualización	Protocolos	Aplicabilidad	Límite
Mininet	Linux namespaces	FRR rift-python	Escala en un único host	Recursos
Kathará	Contenedores Docker	FRR rift-python	Rápida configuración y despliegue	Otros
Megalos	Kubernetes y Docker	FRR rift-python	Escala horizontal y distribuida	Recursos
MaxiNet	Linux namespaces	FRR rift-python	Escala horizontal y distribuida	Recursos
CORE	Linux namespaces	FRR rift-python	Emulación con soporte de GUI	Otros
ns-3	Simulada	FRR	Evaluación de implementaciones	Recursos (vRAM)

Tabla 4.2: Resumen de los ambientes validados en términos de: la virtualización utilizada, los protocolos validados, una aplicabilidad recomendada y el límite detectado.

Para la evaluación se tomó como métrica la carga de la mensajería (sección 3.3.1). De esta manera, se utilizará el término "paquete" para referirse a los paquetes de control tomados en cuenta por la métrica, es decir, mensajes BGP de tipo Update (en adelante, Updates). A su vez, se referirá a los mensajes BGP de tipo Update que contengan rutas a desagregar como Withdraw.

4.3.1. Bootstrap con BGP

En esta sección, el objetivo es analizar el comportamiento de la implementación de FRR para el protocolo BGP cuando este inicia. El experimento se realizó utilizando el ambiente simulado con ns-3 y como métrica la carga de la mensajería (sección 3.3.1). La Tabla 4.3 resume los resultados obtenidos en el experimento en términos del k que describe al fat-tree, la cantidad de prefijos difundidos (uno por Leaf, es decir, $k^2/2$) y la cantidad total de Updates intercambiados por todos los nodos de la topología desde el inicio de la simulación, hasta que el protocolo convergió. Si bien la finalización de la simulación está fijada por configuración, es decir, el simulador no es consciente del estado de convergencia del protocolo, se verificó la convergencia *post mortem* en base a las capturas de tráfico generadas durante el experimento. También, notar que una vez que el protocolo convergió, los nodos intercambiarán únicamente

k	Prefijos	Updates
4	8	576
8	32	22392
12	72	158364
16	128	669872
20	200	1898970
24	288	7247899
28	392	22189258
30	450	35139287
32	512	54880110

Tabla 4.3: Resultados del conteo de mensajes Update para el caso de uso *Bootstrap* sobre el ambiente ns-3.

mensajes BGP de tipo Keepalive, por lo que terminar la simulación por configuración (en base a esperar un tiempo suficiente) no altera los resultados de este análisis.

La Figura 4.2 muestra la cantidad de mensajes Update en función del k. Según [90], el orden de los mensajes Update intercambiados durante el *Bootstrap* es de $\Theta(k^5)$. Por este motivo, la Figura 4.2 incluye además, la curva que describe a este polinomio, validando que el orden experimental obtenido para los mensajes Update se corresponde con el resultado teórico.

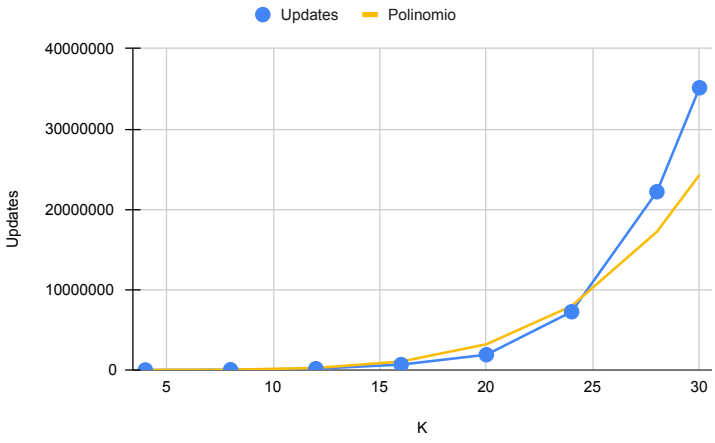


Figura 4.2: Crecimiento de mensajes Update en función del factor k obtenidos con el ambiente ns-3, en comparación con el polinomio k^5.

Por otro lado, la Figura 4.3 muestra la tendencia de estos mensajes a escala logarítmica. Dado el amplio rango de valores, esta visualización permite observar la tendencia del crecimiento de los mensajes Update sin perder noción

de las topologías con menos nodos.

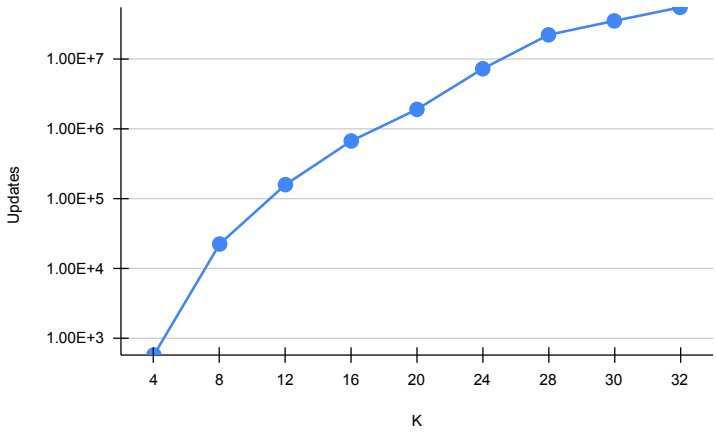


Figura 4.3: Tendencia a escala logarítmica de los mensajes Update en función del factor k.

4.3.2. Falla de un nodo Leaf con BGP

El objetivo de esta sección es analizar el comportamiento de BGP ante la falla de un nodo de tipo Leaf. Para generar los resultados se utilizó el ambiente emulado que utiliza el framework Sibyl y el ambiente simulado en ns-3.

La ejecución del experimento transita varias etapas, primero, se levanta la emulación y se espera a que el protocolo alcance la convergencia, es decir, realice el *Bootstrap*; luego se elimina un nodo de tipo Leaf, en este caso el *leaf-0-1* (primero desde la izquierda) y se captura el tráfico intercambiado por todas las interfaces de todos los nodos de la topología hasta determinar que el protocolo nuevamente convergió.

Tomando como métrica el conteo de paquetes, la Figura 4.4 muestra la tendencia de aumento que siguen los mismos respecto al k que define a la topología. Esta tendencia se muestra comparativamente para los resultados experimentales obtenidos con Sibyl y ns-3. Una observación apreciable en la figura es la diferencia de los resultados obtenidos en ambos ambientes.

4.3.2.1. Métricas aplicadas

Continuando con la evaluación de BGP en el escenario de la falla de un nodo Leaf, se utilizaron los resultados de la experimentación con el framework

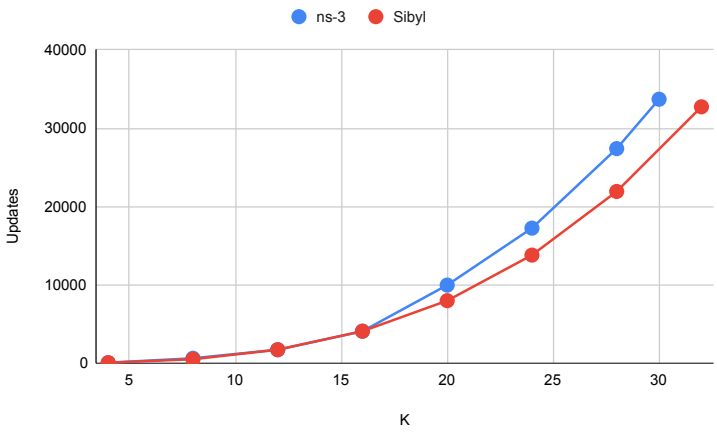


Figura 4.4: Evolución del conteo de paquetes en función del k, para los experimentos con ns-3 y Sibyl en el escenario de falla de un Leaf.

Sibyl para mostrar en acción todas las métricas independientes del tiempo presentadas en la sección 3.3. En este sentido, y con el objetivo de tener resultados con representaciones visuales y coherentes con el fat-tree que se ha mostrado a lo largo de este documento, se escogió estudiar un fat-tree con $k = 4$.

Siguiendo el orden de las métricas presentadas en la sección 3.3, en primer lugar se obtiene como resultado la carga de la mensajería. Este resultado está presente en la Figura 4.4 pero por razones de escala no es visible. En definitiva, para $k = 4$ se contaron un total de 60 paquetes.

En segundo lugar, se analiza la Localidad (sección 3.3.2), teniendo como salida el *blast radius* de los mensajes luego de la falla. Recordar que el *blast radius* es un valor escalar resultado del producto entre el vector L, que contiene el número de paquetes capturados para cada distancia topológica considerada (de 0 a 3), y el vector W, donde $W[i] = i + 1$ con $i = 0, 1, 2, 3$ que es el encargado de darle mayor peso a los paquetes más lejanos al nodo de la falla. La Figura 4.5 generada con el framework Sibyl y adaptada a la notación aquí utilizada, presenta, para cada enlace de la topología, la cantidad de paquetes intercambiados. Como se puede observar en la figura todos los enlaces de la topología, sin importar lo distante al evento, intercambian la misma cantidad de paquetes. Esto adelanta una idea del comportamiento del protocolo en cuanto a la localidad de los eventos. Si se calcula el *blast radius* siguiendo la definición se tiene que:

- Para cada enlace de la red, se tiene un total de 2 paquetes recibidos por

las interfaces de ese enlace.

- Se asigna una distancia topológica a cada enlace de la red: todos los enlaces incidentes al *leaf-0-1* tienen un distancia topológica de 0. Luego, la distancia topológica del resto de los enlaces es el número mínimo de enlaces que deben atravesarse para llegar a un enlace incidente al *leaf-0-1*.

- Se obtiene el vector $L = (0, 12, 24, 24)$ donde $L[i]$ es la cantidad de paquetes capturados a la distancia topológica i.

- Se calcula el producto escalar entre L y $W = (1, 2, 3, 4)$ obteniendo el *blast radius* de la falla, en este caso $0 \times 1 + 12 \times 2 + 24 \times 3 + 24 \times 4 = 192$.

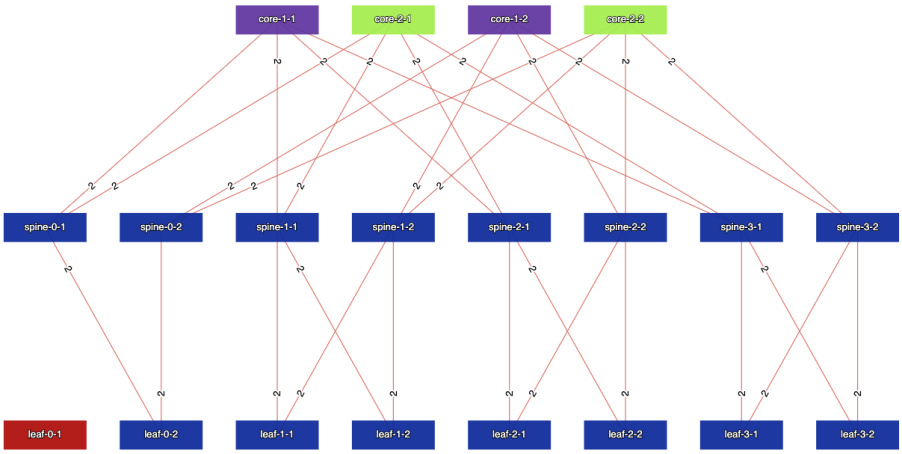


Figura 4.5: Cantidad de mensajes Update que atraviesan cada enlace de una topología fat-tree con $k = 4$ luego de la falla de un nodo Leaf.

Finalmente, los resultados que arroja el framework utilizando como métrica la cantidad de rondas (sección 3.3.3) son: la línea de tiempo nodo-estado y el grafo nodo-estado. La Figura 4.6 muestra la línea de tiempo nodo-estado resultado del experimento. Este gráfico muestra los paquetes intercambiados durante el experimento con información extra basada en los tiempos locales de cada nodo, lo que permite ordenar los paquetes cronológicamente. Con esta información se puede observar como, al tener que difundir información sobre un único prefijo, todos los nodos de la topología envían y reciben exactamente un paquete por interfaz, o análogamente, se observan dos paquetes por enlace. Este resultado es coherente con el comportamiento esperado de BGP bajo este escenario. Los detalles se muestran en la siguiente sección.

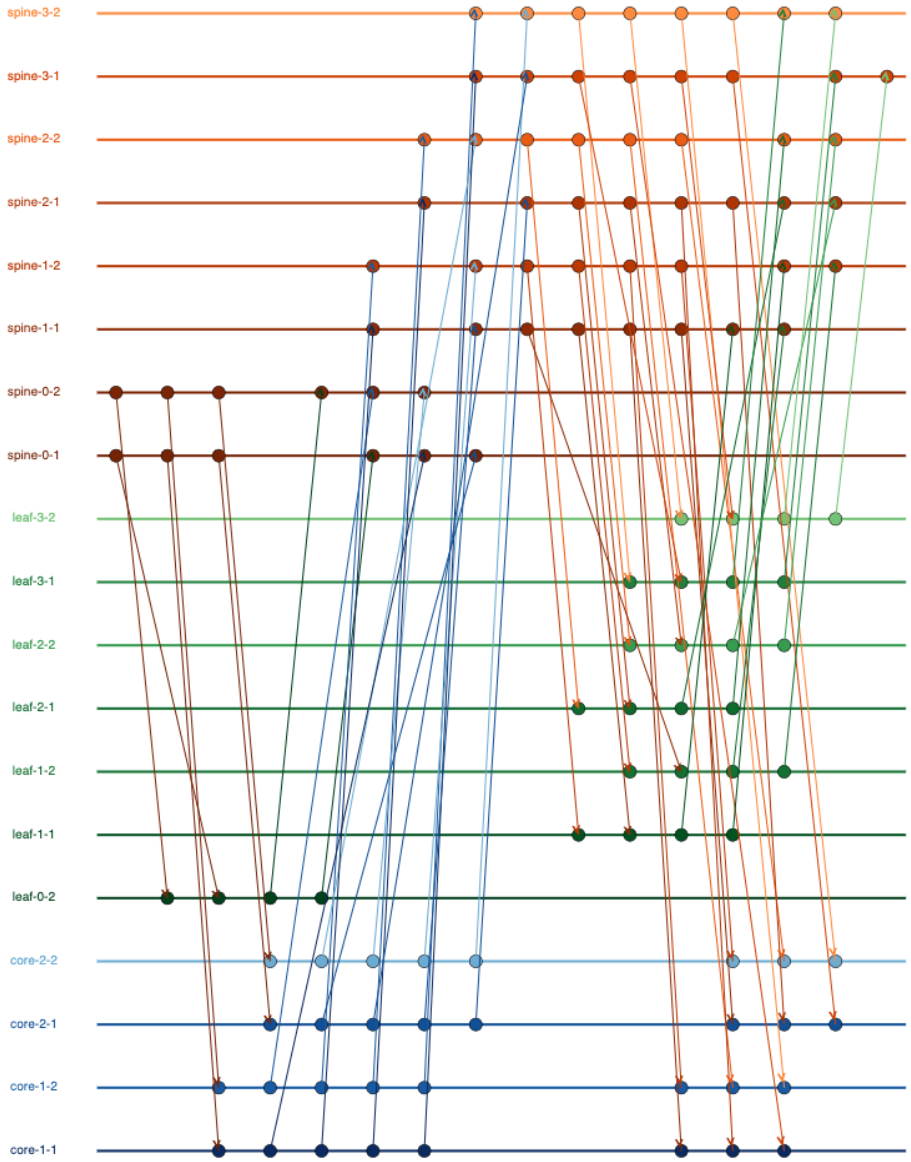


Figura 4.6: Línea de tiempo nodo-estado generada luego de la falla del nodo *leaf-0-1* en la topología fat-tree de referencia.

4.3.2.2. Verificación teórica

Cuando falla un nodo de tipo Leaf, el efecto a nivel de enrutamiento es que un prefijo de la topología ya no es alcanzable por el resto, y que los enlaces adyacentes al nodo de la falla ya no estarán disponibles para el intercambio de paquetes de control. Bajo este escenario, cada nodo BGP debe recibir y enviar por todas sus interfaces un Withdraw conteniendo el prefijo que ya no es alcanzable. Notar que quienes inician el aviso son los Spine adyacentes al Leaf de la falla, quienes dejarán de recibir los mensajes BGP de tipo Keepalive

desde él y asumirán que el Leaf, y por lo tanto el prefijo alcanzable a través de él, ya no están disponibles.

En términos del conteo de paquetes intercambiados luego de la falla, esto implica que en cada enlace de la topología se verán dos paquetes. El total de enlaces antes de la falla corresponde a la expresión $k^3/2$ (Tabla 2.1). Cada Leaf está conectado a $k/2$ Spines, por lo que luego de la falla, estos enlaces no serán considerados, quedando un total de $k^3/2 - k/2$ enlaces luego de la falla. Así, la cantidad de Updates intercambiados luego de la falla se puede representar con la expresión $2 \times (k^3/2 - k/2)$ o equivalentemente:

$$k^3 - k \tag{4.1}$$

Notar que esta expresión es una cota mínima. Cuando un nodo Leaf recibe el Withdraw mencionado desde un Spine, su tabla de enrutamiento sigue teniendo una entrada para el prefijo desagregado, que es alcanzado a través de otro Spine. Esta condición de carrera hace creer al Leaf que todavía puede alcanzar a este prefijo, y reaccionar al Withdraw con un Update anunciando la (inexistente) ruta al prefijo a los Spines conectados a él. Debido a la específica numeración de los ASN en el fat-tree (sección 2.2), cada prefijo anunciado en dichos Update contienen en su AS-PATH el ASN del Spine que lo recibe, y es por esta razón que luego de recibirlo, lo descarta. Esto evita que la inexistente ruta se siga propagando al resto de la topología, y parando el *path hunting* en una etapa temprana. Una forma simple de encontrar una expresión que modele ese comportamiento es agregar un paquete más para cada enlace Leaf-Spine a la expresión anterior (4.1). Para determinar esta cantidad de enlaces, primero contamos el número de enlaces dentro de un PoD normal, es decir, $k/2 \times k/2$ y lo multiplicamos por la cantidad de PoDs normales $(k-1)$. Luego, el número de enlaces en el PoD de la falla es $k/2 \times (k/2-1)$. En consecuencia, la cantidad total de paquetes BGP que modelan este comportamiento se puede expresar con el polinomio:

$$\frac{5k^3}{4} - \frac{3k}{2} \tag{4.2}$$

La Figura 4.7 compara el crecimiento de los paquetes en función del k que describe a la topología, para los resultados obtenidos de la experimentación con ns-3 y Sibyl, y los polinomios hallados. Notar que mientras los resultados de Sibyl siguen al polinomio (4.1), los de ns-3 siguen a (4.1) y (4.2) de manera

alternada. De todas maneras, los resultados son correctos ya que ambos comportamientos pueden ocurrir debido a la naturaleza de BGP y la sincronización de los paquetes del plano de control.

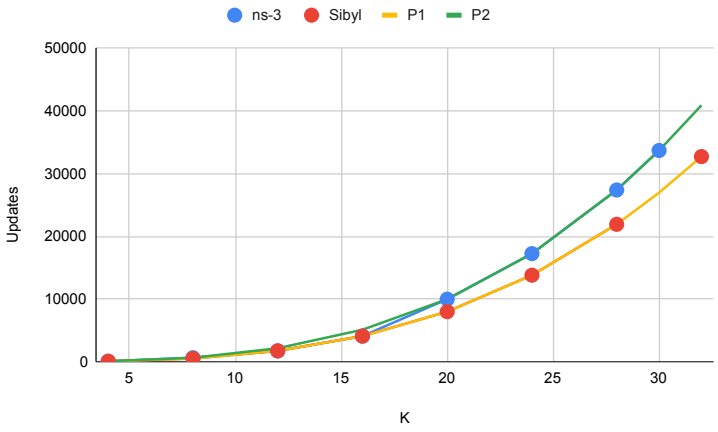


Figura 4.7: Evolución del conteo experimental de paquetes para el escenario de falla de un nodo Leaf en comparación con los polinomios $P1 = k^3 - k$ y $P2 = \frac{5k^3}{4} - \frac{3k}{2}$.

4.3.3. Falla de un nodo de tipo Spine con BGP

El segundo caso de la falla de un nodo a analizar, es cuando la falla se presenta al nivel de los nodos Spine. La Figura 4.8 muestra la evolución de los paquetes intercambiados luego de la falla de un nodo Spine en los entornos ns-3 y Sibyl.

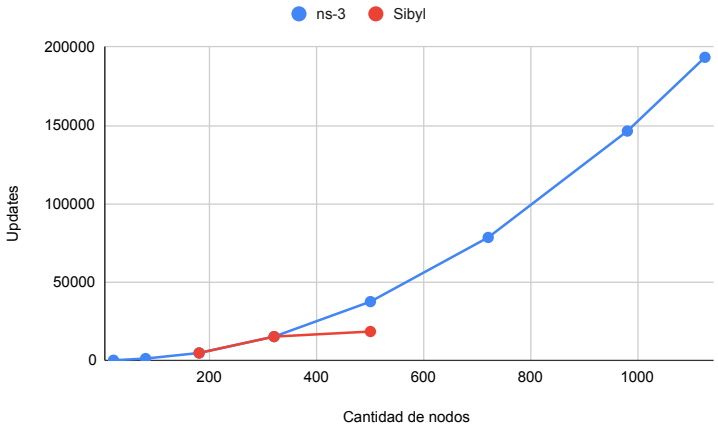


Figura 4.8: Evolución del conteo de paquetes en función de la cantidad de nodos para la experimentación con ns-3 y Sibyl en el escenario de falla de un Spine.

En este caso, los resultados generados por Sibyl para el escenario $k = 20$ arrojan un conteo de 18390 paquetes, mientras que los generados en ns-3 un total de 37480 paquetes (más del doble, ver Tabla 4.4). Desafortunadamente, para este caso de uso no se escaló en los resultados generados con Sibyl a configuraciones mayores a este k. Esta diferencia será analizada con la validación teórica desarrollada a continuación.

4.3.3.1. Verificación teórica

Para analizar el comportamiento de BGP cuando falla un nodo Spine, se dividió el problema en tres subproblemas: 1) el PoD de la falla, 2) los PoDs sin falla y 3) los enlaces Spine-Core. Dado que el objetivo es encontrar una expresión que modele el número total de paquetes intercambiados luego de la falla, dividir el problema en subproblemas es equivalente a dividir dicha expresión en sumas.

1. Primero, notar que cada Leaf necesita información de conectividad para los $k^2/2$ prefijos de la topología; mientras que $k^2/2 - 1$ prefijos son "externos", el restante está directamente conectado a él. Cuando el Spine falla, en el PoD donde ocurre hay $k/2$ nodos Leaf conscientes de la falla (conectados al Spine caído). El proceso BGP de cada uno de estos nodos Leaf recalculará las rutas y notará que por cada prefijo conocido, falta un posible próximo salto. En consecuencia, enviará, por cada prefijo conocido, un Update con los prefijos actualizados. Por lo tanto, tenemos $k/2$ nodos Leaf que envían $k^2/2 - 1$ paquetes (el número total de prefijos en la topología que han perdido un siguiente salto) a través de sus $k/2 - 1$ enlaces. En consecuencia, la cantidad total de paquetes BGP en el PoD de la falla es igual a $k/2 \times (k/2 - 1) \times (k^2/2 - 1)$.

2. El resto de los PoDs conocen la falla a través del Spine conectado al mismo plano que el Spine caído. Este Spine, luego de recibir el aviso de la falla, envía a todos sus vecinos ($k/2$ nodos Leaf en este sub problema), un Withdraw que contiene todos los prefijos que ya no se pueden alcanzar a través del nodo Core correspondiente (todos los prefijos dentro del PoD de la falla, es decir, $k/2$). Luego, cada Leaf vuelve a calcular sus rutas y observa que cada prefijo recibido en el Withdraw ya no es alcanzable a través de uno de sus próximos saltos. En consecuencia, enviará, para cada

uno de estos $k/2$ prefijos, un Update a todos sus vecinos ($k/2$ Spines). Así, la cantidad total de paquetes BGP en un PoD sin falla es igual a $(k/2) + (k/2) \times (k/2) \times (k/2)$.

3. El Spine de la falla estaba conectado a $k/2$ nodos Core. Debido a la característica de la topología fat-tree multi-plane, estos Core tienen exactamente un enlace con cada PoD. Luego de la falla, cada Core conectado con el Spine de la falla ya no tiene alcance a los prefijos del PoD correspondiente, y debe entonces enviar un Withdraw con los prefijos de dicho PoD a todos sus vecinos ($k-1$ Spines). Luego de eso, cuando un Spine conectado a estos Core reciba el Withdraw mencionado, notará que ya no tiene alcance a los prefijos anunciados, y enviará el Withdraw correspondiente de nuevo hacia los Core; por lo tanto, un total de dos paquetes atraviesan cada enlace Core-Spine. En resumen, se tienen $k/2$ Core que envían y reciben un Withdraw a través de todas sus interfaces disponibles ($k-1$). En consecuencia, la cantidad total de paquetes BGP en los enlaces Core-Spine es igual a $2 \times (k-1) \times k/2 = (k-1) \times k$ paquetes.

Para llegar a la expresión final, basta con sumar las expresiones anteriores y multiplicar la expresión de 2 por la cantidad total de PoDs sin falla ($k-1$). Así, se tiene un total de $k/2 \times (k/2-1) \times (k^2/2-1) + (k-1) \times ((k/2) + (k/2) \times (k/2) \times (k/2)) + (k-1) \times k$ paquetes. Simplificando, se obtiene el polinomio

$$\frac{k^4}{4} - \frac{3k^3}{8} + \frac{5k^2}{4} - k \tag{4.3}$$

La Figura 4.9 compara el crecimiento de paquetes en función del número de nodos para los resultados de ns-3, Sibyl y la expresión polinomial. Se observa que los resultados de ns-3 se ajustan exactamente al polinomio. Por otro lado, los resultados de Sibyl muestran una desviación del polinomio. Esto indica que los resultados de la emulación representan un valor atípico en su último tramo, que podrían deberse a que el experimento no finalizó correctamente por falta de recursos u otro motivo.

k	Conteo ns-3	Polinomio(k)
4	56	56
8	1128	904
12	4704	4704
16	15152	15152
20	37480	37480
24	78456	78456
28	146384	146384
30	193470	193470

Tabla 4.4: Conteo de mensajes de tipo Update con el ambiente ns-3 para el caso de uso *Node failure* aplicado a un nodo de tipo Spine comparado con los valores asociados al polinomio hallado.

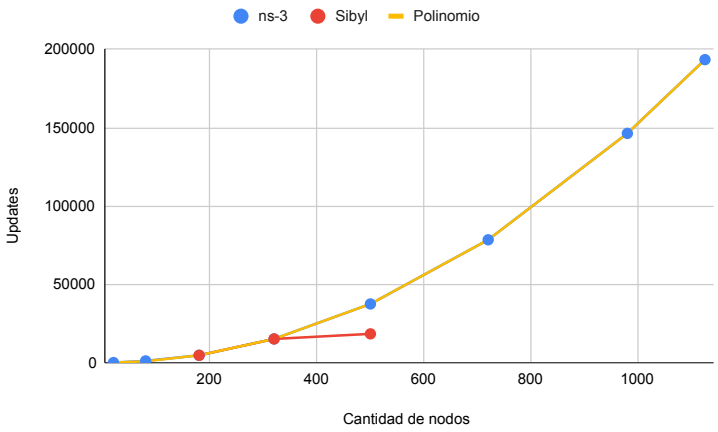


Figura 4.9: Evolución de los resultados de ns-3 y Sibyl para el escenario de falla de un Spine, en comparación con la evolución del polinomio $\frac{k^4}{4} - \frac{3k^3}{8} + \frac{5k^2}{4} - k$.

4.3.4. Recuperación de la falla de un nodo con BGP

Cuando un nodo Leaf se recupera de una falla deben suceder dos cosas, la primera es que toda la estructura debe volver a alcanzar al nuevo prefijo (recordar que cada Leaf anuncia un prefijo distinto). Para esto, el nodo debe anunciar a sus $k/2$ vecinos este nuevo prefijo. En segundo lugar, sus $k/2$ vecinos le deben anunciar todos los prefijos que conocen, o equivalentemente la cantidad de nodos de tipo Leaf previos a la recuperación: $k^2/2 - 1$.

1. Enlaces del nodo recuperado: cuando el nodo Leaf se recupera de una falla, este envía un Update a sus $k/2$ vecinos con su prefijo. A su vez,

estos $k/2$ vecinos reconocen al prefijo como nuevo, y también lo anuncian a todos sus vecinos. En particular, si consideramos los enlaces del nodo recuperado, esto tiene un costo de $k/2 \times 2 = k$ paquetes. Por otra parte, los vecinos del Leaf recuperado, le deben anunciar a este todos los prefijos que conocen, es decir, $k^2/2 - 1$. A su vez, el Leaf va a reconocer a estos prefijos como nuevos, y anunciarlos a todos sus vecinos ($k/2$). Esto agrega al costo mencionado $(k^2/2 - 1) \times 2 \times k/2$ paquetes, sumando un total de $(k^2/2 - 1) \times k + k$.

2. Resto de los enlaces: en el resto de los enlaces de la topología solo son necesarios dos paquetes por enlace, es decir, cada nodo de la topología va a recibir y anunciar el nuevo prefijo alcanzable, por todos sus enlaces. Una forma de expresar esta cantidad, en función del k, es restándole a la cantidad de enlaces totales los adyacentes al nodo recuperado, y multiplicando esto por la cantidad de paquetes intercambiados en cada enlace. Esto nos deja la expresión $((k^3/2) - (k/2)) \times 2$.

Si se suman las expresiones y se simplifica el resultado, el polinomio que caracteriza al crecimiento de paquetes en función del k que describe a la topología es:

$$\frac{3k^3}{2} - k \tag{4.4}$$

La Figura 4.10 muestra comparativamente la evolución de este polinomio y los resultados de la ejecución del caso de uso en ns-3.

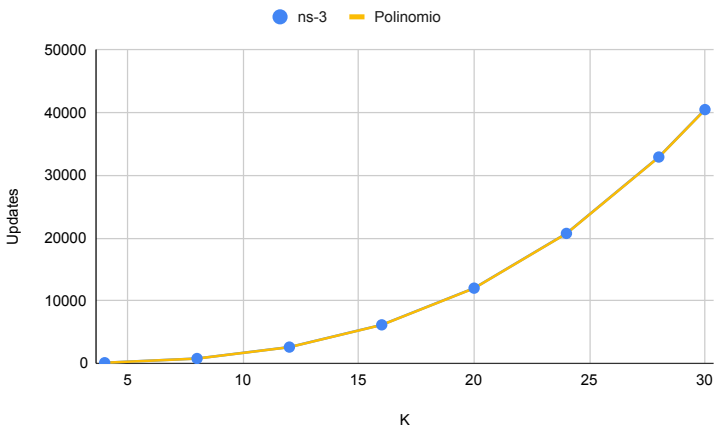


Figura 4.10: Evolución de los resultados de ns-3 para el escenario de recuperación de un nodo Leaf, en comparación con la evolución del polinomio $\frac{3k^3}{2} - k$.

En esta figura, se puede observar como los resultados experimentales son verificados por la expresión teórica hallada. Si bien a la escala observada en la figura los resultados son exactos, existe una pequeña diferencia constante entre el polinomio y los resultados experimentales. Como se mencionó anteriormente, las expresiones analíticas son una cota inferior y durante los experimentos, las implementaciones o condiciones de carrera de la ejecución pueden hacer que el conteo difiera de esta cota. En este caso, se analizaron en detalle los resultados experimentales, observando dos paquetes Update adicionales en los enlaces que conectan al nodo recuperado. Estos Update (uno del nodo que se recuperó hacia sus vecinos y la respuesta desde los mismos) están vacíos, es decir, no contienen nuevas rutas ni prefijos a desagregar. Este comportamiento puede ser natural de la implementación o un bug existente.

Finalmente, en la Figura 4.11 se presenta comparativamente para los casos de uso presentados y verificados teóricamente, la evolución del conteo de paquetes en función del k. En esta figura se puede observar como de los tres casos analizados, el más costoso en términos de cantidad de Updates inyectados es cuando falla de un nodo de tipo Spine, mientras que el menos costoso es la falla de un nodo Leaf.

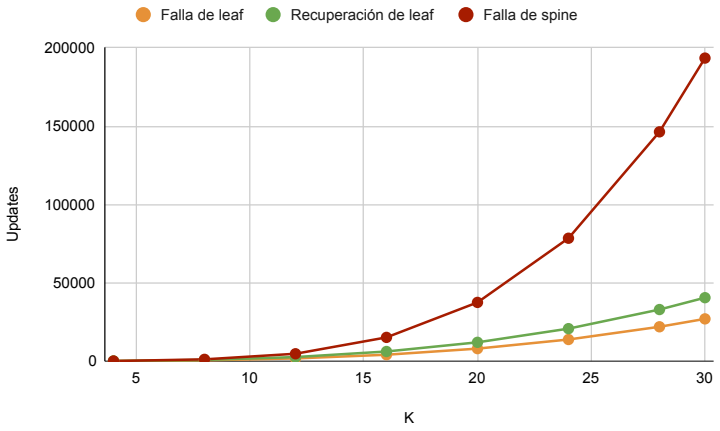


Figura 4.11: Comparación de la evolución del conteo de paquetes en función del k, para los casos de: falla de un Leaf, recuperación de un Leaf y falla de un Spine.

4.4. Consideraciones finales

En este capítulo se presentó la evaluación de los ambientes de experimentación en términos de validación, escalabilidad y posibles aplicaciones. En térmi-

nos de escalabilidad sobre el ambiente de ejecución utilizado (y por lo tanto restringidos a los recursos de los mismos) se encontró que los ambientes con Mininet, MaxiNet, Megalos y ns-3 son propicios para trabajar a escalas de un MSDC.

Es importante mencionar que si bien los desarrollos realizados en el marco de esta tesis (Mininet, MaxiNet, CORE y los ambientes SDN) logran validar los puntos más importantes para poder experimentar con fat-trees enrutados, pueden ser extendidos y personalizados fácilmente. Por ejemplo, se podrían implementar todas las métricas independientes del tiempo (tal como lo hace el framework Sibyl), o nuevos casos de uso que validen otros aspectos de los protocolos de enrutamiento.

Por otra parte, se realizó una evaluación del protocolo de enrutamiento BGP bajo los escenarios de: *Bootstrap*, falla y recuperación de un nodo Leaf y falla de un nodo Spine. Los resultados de los experimentos muestran, entre otras cosas, como bajo estos escenarios la falla de un Spine inyecta más Updates que, por ejemplo, la falla de un Leaf. Estos resultados fueron validados con un análisis teórico del comportamiento de BGP, donde se obtuvieron una serie de polinomios que modelan la cantidad de Updates inyectados por el protocolo en función del k que describe a la topología para los casos de uso mencionados.

Paralelamente a la presentación y validación de los resultados, en este capítulo se fue mostrando el uso de los ambientes de experimentación de Sibyl y ns-3, presentando los resultados obtenidos de uno y/u otro con el objetivo de mostrar sus potenciales.

Capítulo 5

Consideraciones finales

Los entornos experimentales son necesarios para desarrollar y probar nuevos protocolos y servicios de red. Esto es particularmente cierto en el caso de topologías fat-tree de gran densidad y dimensiones, representativas para los data centers de escala masiva. Durante el desarrollo de esta tesis se ha revisado y experimentado con una variedad de entornos experimentales que se valen de emuladores y simuladores de red, desarrollando utilidades y configuraciones específicas para el caso de uso del MSDC. Estos entornos de experimentación, en conjunto con la metodología definida para llevar a cabo un análisis experimental de protocolos de enrutamiento sobre los mismos, son un aporte fundamental de este trabajo y sientan los cimientos para poder realizar experimentos representativos para un MSDC.

Por otra parte, se realizó un análisis experimental del enrutamiento en redes fat-tree. En este sentido, se presentó una metodología de trabajo necesaria para llevar a cabo esta tarea, donde la adaptación de herramientas disponibles, la automatización de la generación de las topologías fat-tree y de las configuraciones de los protocolos de enrutamiento, y el análisis de los resultados utilizando métricas independientes del tiempo, son los pilares fundamentales para lograr esta tarea. Respecto a estos entornos de experimentación, con el entorno emulado que utiliza a Mininet como emulador de la red se ha logrado experimentar con un data center enrutado de hasta 720 nodos de red sobre un único host físico. Por otro lado, el ambiente provisto por el framework Sibyl, que utiliza a Megalos como emulador de la red, ha logrado emular una DCN de hasta 1280 nodos de red en un ambiente distribuido. Con una cantidad considerable menor de recursos que Megalos, MaxiNet logró emular hasta 980

nodos de red en un ambiente distribuido. Finalmente, el ambiente simulado sobre ns-3 logró simular hasta 1280 nodos de red con 5.2 veces menos de recursos (vRAM) que Megalos. Las escalas logradas son representativas para el caso de uso del MSDC mencionado.

Con respecto a la evaluación de los protocolos de enrutamiento, todas las implementaciones probadas, es decir, BGP, IS-IS con reducción de inundación y rift-python, implementan correctamente el enrutamiento ECMP. Sin embargo, difieren en términos de escalabilidad, siendo BGP el preeminente, resultado obtenido en el trabajo sobre el framework Sibyl [77]. Tomando a este protocolo como caso a analizar, en este trabajo se realizó una evaluación experimental del mismo, tomando como métrica principal la cantidad de mensajes del plano de control intercambiados bajo un conjunto de casos de uso de interés. Los resultados experimentales fueron validados con un análisis teórico del comportamiento del protocolo bajo los escenarios mencionados. Este análisis teórico es un aporte en si mismo, que permite tener una expresión teórica de la cantidad de tráfico de control inyectada en la DCN por el protocolo BGP, luego de la ocurrencia de ciertos eventos de interés.

5.1. Contribuciones

La mayoría del trabajo detallado en este documento se ha difundido oportunamente en una serie de artículos científicos indexados y con revisión de pares:

- **Alberro L.**, Castro A., Grampin E. Experimentation Environments for Data Center Routing Protocols: A Comprehensive Review. Future Internet. 2022; 14(1):29. https://doi.org/10.3390/fi14010029. [91]

- Caiazzi T., Scazzariello M., **Alberro L.**, Ariemma L., Castro A., Grampin E. y Battista G. D., "Sibyl: a Framework for Evaluating the Implementation of Routing Protocols in Fat-Trees,"NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, 2022, pp. 1-7. doi: 10.1109/NOMS54207.2022.9789876. [77]

- **Alberro L.**, Velázquez F., Azpiroz S., Grampin E., Richart M. Experimenting with Routing Protocols in the Data Center:

An ns-3 Simulation Approach. Future Internet. 2022; 14(10):292. https://doi.org/10.3390/fi14100292. [92]

Respecto a la evaluación de protocolos de enrutamiento con el framework Sibyl y a la definición de las métricas independientes del tiempo, los resultados principales fueron publicados en [77]. Este autor trabajó en la definición de la metodología y la experimentación. El software denominado Sibyl, framework que implementa la metodología definida, fue desarrollada por otros miembros del equipo de trabajo.

Por otra parte, una evaluación de entornos de experimentación para protocolos de enrutamiento en el data center, fue presentada en [91]. Este trabajo fue desarrollado en su mayoría por este autor, recibiendo la colaboración de Eduardo Grampín y Alberto Castro en la conceptualización, preparación del borrador original y escritura de la revisión y edición, tal como se especifica al final de dicho artículo.

Finalmente, en [92] se presenta una experimentación con protocolos de enrutamiento realizada sobre el simulador ns-3. En este trabajo se validó el porte de la implementación de FRR para BGP comparando los resultados obtenidos con los del framework Sibyl. Además, se presentó la verificación teórica de los resultados experimentales para los casos de uso de falla de un nodo Leaf y Spine. El porte de software presentado en dicho artículo fue desarrollado por otros miembros del equipo de trabajo. Este autor lideró el análisis y validación teórica de los resultados, y tuvo una fuerte participación en la experimentación y presentación de los resultados.

5.2. Disponibilidad del software

Todos los ambientes de experimentación desarrollados o adaptados en el marco de esta tesis están disponibles de forma libre en [85], [89], [93].

- [89]: En este repositorio se encuentran los entornos, junto con las herramientas desarrolladas para experimentar con Mininet, Core y MaxiNet. Es decir, generadores de topologías, configuración de los protocolos, programas principales con la programación de los casos de uso, recolección de resultados, y scripts de soporte para el análisis *post mortem* para algunos de los entornos. Además, en este repositorio se encuentra el proyecto GNS3 del experimento presentado en la sección 4.2.3.

- [93]: Rama modificada de Ultraviolet [94] donde se desarrolló la capacidad de guardar meta datos acerca de la distribución de la emulación y de la creación de túneles en MaxiNet, así como comandos para consultar las tablas de reenvío de los nodos emulados. Estos meta datos son necesarios para la automatización de la configuración de los protocolos de enrutamiento.

- [85]: Este es el repositorio del porte de FRR BGP a ns-3 DCE. El porte fue desarrollado por estudiantes de grado en el marco de un proyecto de fin de carrera. Este autor hizo aportes menores como la incorporación del caso de uso *Bootstrap*.

Los entornos mencionados, junto con Sibyl [82], permiten la fiel reproducción de los experimentos que se presentaron en este trabajo.

5.3. Trabajo a futuro

Si bien este trabajo analiza las DCN centrado en el enrutamiento ECMP, cabe mencionar que para lograr una DCN que maximice la utilización del ancho de banda y minimice la latencia, no alcanza con tener múltiples caminos disponibles a nivel de enrutamiento, sino que los protocolos de transporte deben ser capaces de enviar tráfico de manera eficiente por estos múltiples caminos. En efecto, Transmission Control Protocol (TCP), el protocolo de transporte que utiliza Internet para el transporte confiable, tiene un rendimiento muy poco satisfactorio en el entorno MSDC. Por ejemplo, dos problemas conocidos que sufre este protocolo en el data center son los denominados TCP Incast y TCP Outcast [95]. Es por esta razón que se deben buscar soluciones específicas de transporte en este entorno, como lo son NDP [8], Homa [96] o SCDP [97], que además, pueden necesitar ser coordinadas con los mecanismos de reenvío de los conmutadores de la DCN para lograr sus objetivos. Puede ser de interés analizar estas soluciones y evaluar, por ejemplo, si estos protocolos de transporte pueden valerse de la regularidad de la topología fat-tree para mejorar la latencia y el ancho de banda en una DCN.

Por otro lado, agregando la capa de aplicación al análisis, el modelo de computación en la nube ("Cloud Computing") domina ampliamente las estrategias de despliegue de aplicaciones en los MSDC. Los gestores de este modelo

asignan recursos en el data center intentando distribuir la carga, lo que frecuentemente implica mover máquinas virtuales entre servidores, y que desde el punto de vista del transporte resulta en fuentes de tráfico que se mueven dentro de la DCN [98]. En este sentido, las funciones objetivo de negocio que determinan la asignación de recursos a alto nivel no necesariamente están correlacionadas con los mecanismos de transporte del tráfico; en general, los procesos de optimización de Cloud Computing buscan mejorar la eficiencia energética (p.e., "prendiendo y apagando" recursos tales como CPUs/GPUs), y el balance de carga de recursos de red y CPU, lo que implica mover recursos virtuales. Es por esta razón que para lograr una DCN eficiente, una línea a futuro es idear modelos de optimización que contemplen el diseño de la DCN y la asignación de recursos de red para minimizar la latencia de las aplicaciones, entre otros objetivos posibles.

Bibliografía

[1] Huston G., *The death of transit? APNIC Blog*, https://blog.apnic.net/2016/10/28/the-death-of-transit/, (Accessed on September 2022), 2016.

[2] Cisco, «Cisco Global Cloud Index: Forecast and Methodology, 2016–2021,» White Paper, 2018. dirección: https://virtualization.network/Resources/Whitepapers/0b75cf2e-0c53-4891-918e-b542a5d364c5_white-paper-c11-738085.pdf.

[3] Clos C., «A study of non-blocking switching networks,» *The Bell System Technical Journal*, vol. 32, n.º 2, págs. 406-424, mar. de 1953, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

[4] Leiserson C. E., «Fat-trees: Universal networks for hardware-efficient supercomputing,» *IEEE Transactions on Computers*, vol. C-34, n.º 10, págs. 892-901, oct. de 1985, ISSN: 2326-3814. DOI: 10.1109/TC.1985.6312192.

[5] Al-Fares M., Loukissas A. y Vahdat A., «A Scalable, Commodity Data Center Network Architecture,» en *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ép. SIGCOMM '08, Seattle, WA, USA: ACM, 2008, págs. 63-74, ISBN: 978-1-60558-175-0. DOI: 10.1145/1402958.1402967. dirección: http://doi.acm.org/10.1145/1402958.1402967.

[6] Medhi D. y Ramasamy K., *Network Routing, Second Edition: Algorithms, Protocols, and Architectures*, 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128007370, 9780128007372.

[7] Bari M. F., Boutaba R., Esteves R., Granville L. Z., Podlesny M., Rabbani M. G., Zhang Q. y Zhani M. F., «Data Center Network Virtualization: A Survey,» *IEEE Communications Surveys Tutorials*, vol. 15, n.º 2,

págs. 909-928, feb. de 2013, ISSN: 2373-745X. DOI: 10.1109/SURV.2012.090512.00043.

[8] Handley M., Raiciu C., Agache A., Voinescu A., Moore A. W., Antichi G. y Wójcik M., «Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance,» en *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ép. SIGCOMM '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, págs. 29-42, ISBN: 9781450346535. DOI: 10.1145/3098822.3098825. dirección: https://doi.org/10.1145/3098822.3098825.

[9] IEEE, *Spanning Tree Protocols*, https://www.ieee802.org/1/files/public/docs2009/aq-seaman-merged-spanning-tree-protocols-0509.pdf, (Accessed on October 2022), 2009.

[10] Lapukhov P., Premji A. y Mitchell J., «Use of BGP for Routing in Large-Scale Data Centers,» RFC Editor, RFC 7938, ago. de 2016.

[11] White R., Hegde S. y Zandi S., «IS-IS Optimal Distributed Flooding for Dense Topologies,» IETF Secretariat, Internet-Draft draft-white-distoptflood-04, jul. de 2020. dirección: https://datatracker.ietf.org/doc/pdf/draft-white-distoptflood-04.pdf.

[12] Przygienda T., Sharma A., Thubert P., Rijsman B., Afanasiev D. y Head J., «RIFT: Routing in Fat Trees,» IETF Secretariat, Internet-Draft draft-ietf-rift-rift-16, sep. de 2022.

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

[14] Patel K., Lindem A., Zandi S. y Henderickx W., «BGP Link-State Shortest Path First (SPF) Routing,» IETF Secretariat, Internet-Draft draft-ietf-lsvr-bgp-spf-13, feb. de 2021. dirección: https://www.ietf.org/archive/id/draft-ietf-lsvr-bgp-spf-13.txt.

[15] Haleplidis E., Pentikousis K., Denazis S., Salim J. H., Meyer D. y Koufopavlou O., «Software-Defined Networking (SDN): Layers and Architecture Terminology,» RFC Editor, RFC 7426, ene. de 2015, http://www.rfc-editor.org/rfc/rfc7426.txt. dirección: http://www.rfc-editor.org/rfc/rfc7426.txt.

[16] Jo E., Pan D., Liu J. y Butler L., «A simulation and emulation study of SDN-based multipath routing for fat-tree data center networks,» en *Proceedings of the Winter Simulation Conference 2014*, 2014, págs. 3072-3083. DOI: 10.1109/WSC.2014.7020145.

[17] Askar S., «SDN-Based Load Balancing Scheme for Fat-Tree Data Center Networks,» *Al-Nahrain Journal for Engineering Sciences*, vol. 20, n.º 5, págs. 1047-1056, nov. de 2017. dirección: https://www.nahje.com/index.php/main/article/view/335.

[18] Meta E. at, *Introducing data center fabric, the next-generation Facebook data center network*, https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, (Accessed on October 2022), 2014.

[19] Przygienda T. y Vandezande O., *The Rise of IP Fabrics and Corresponding Novel Routing Paradigms – Part 1*, https://www.comsoc.org/publications/ctn/rise-ip-fabrics-and-corresponding-novel-routing-paradigms-part-1, (Accessed on October 2022), 2022.

[20] Clos C., «A study of non-blocking switching networks,» *The Bell System Technical Journal*, vol. 32, n.º 2, págs. 406-424, mar. de 1953, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

[21] Dean J. y Ghemawat S., «MapReduce: Simplified Data Processing on Large Clusters,» *Commun. ACM*, vol. 51, n.º 1, págs. 107-113, ene. de 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. dirección: https://doi.org/10.1145/1327452.1327492.

[22] Rekhter Y., Li T. y Hares S., «A Border Gateway Protocol 4 (BGP-4),» RFC Editor, RFC 4271, ene. de 2006. dirección: http://www.rfc-editor.org/rfc/rfc4271.txt.

[23] Vohra Q. y Chen E., «BGP Support for Four-octet AS Number Space,» RFC Editor, RFC 4893, mayo de 2007.

[24] Oliveira R., Zhang B., Pei D. y Zhang L., «Quantifying Path Exploration in the Internet,» *IEEE/ACM Transactions on Networking*, vol. 17, n.º 2, págs. 445-458, 2009. DOI: 10.1109/TNET.2009.2016390.

[25] Dutt D. G., *BGP in the Data Center*. O'Reilly, 2017, ISBN: 9781491983409.

[26] Moy J., «OSPF Version 2,» IETF, RFC 2328, abr. de 1998. dirección: http://tools.ietf.org/rfc/rfc2328.txt.

[27] Shen N., Ginsberg L. y Thyamagundalu S., «IS-IS Routing for Spine-Leaf Topology,» IETF Secretariat, Internet-Draft draft-shen-isis-spine-leaf-ext-07, oct. de 2018. dirección: http://www.ietf.org/internet-drafts/draft-shen-isis-spine-leaf-ext-07.txt.

[28] Martey A., Sturgess S. y Martey A., *IS-IS Network Design Solutions*, 2002.

[29] Zandi S., *LinkedIn OpenFabric Project*, https://www.slideshare.net/shawnzandi/linkedin-openfabric-project-interop-2017, (Accessed on October 2022), 2017.

[30] Rijsman B., *Routing In Fat Trees (RIFT)*, https://github.com/brunorijsman/rift-python, (Accessed on October 2022), 2019.

[31] Mahalingam M., Dutt D., Duda K., Agarwal P., Kreeger L., Sridhar T., Bursell M. y Wright C., «Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks,» IETF, RFC 7348, ago. de 2014. dirección: http://tools.ietf.org/rfc/rfc7348.txt.

[32] Sajassi A., Drake J., Bitar N., Shekhar R., Uttaro J. y Henderickx W., «A Network Virtualization Overlay Solution Using Ethernet VPN (EVPN),» IETF, RFC 8365, mar. de 2018. dirección: http://tools.ietf.org/rfc/rfc8365.txt.

[33] Kurose J. F. y Ross K. W., *Computer networking: a top-down approach*. United States of America: Pearson, 2017.

[34] Nadeau T. y Gray K., *SDN: Software Defined Networks*. O'Reilly, 2013, ISBN: 9781449342302.

[35] NOX, *POX Controller*, https://github.com/noxrepo/pox, (Accessed on October 2022), 2020.

[36] Community R. S. F., *RYU SDN*, https://ryu-sdn.org/, (Accessed on October 2022), 2017.

[37] Enns R., Bjorklund M., Schoenwaelder J. y Bierman A., «Network Configuration Protocol (NETCONF),» RFC Editor, RFC 6241, jun. de 2011. dirección: http://www.rfc-editor.org/rfc/rfc6241.txt.

[38] Foundation O. N., *OpenFlow Switch Specification*, https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/openflow-spec-v1.3.2.pdf, (Accessed on October 2022), 2013.

[39] Kaur S., Singh J. y Ghumman N., «Network Programmability Using POX Controller,» ago. de 2014. DOI: 10.13140/RG.2.1.1950.6961.

[40] Team M., *Mininet an instant virtual network on your laptop (or other PC)*, http://mininet.org/, (Accessed on October 2022), 2018.

[41] Kazi N. M., Suralkar S. R. y Bhadade U. S., «Evaluating the Performance of POX and RYU SDN Controllers Using Mininet,» en *Data Science and Computational Intelligence*, Venugopal K. R., Shenoy P. D., Buyya R., Patnaik L. M. e Iyengar S. S., eds., Cham: Springer International Publishing, 2021, págs. 181-191, ISBN: 978-3-030-91244-4.

[42] Nokia, *Nokia SR Linux goes public*, https://netdevops.me/2021/nokia-sr-linux-goes-public/, (Accessed on October 2022), 2021.

[43] Cisco, *NetDevOps - Cisco DevNet*, https://developer.cisco.com/netdevops/, (Accessed on October 2022), 2021.

[44] Maor I., *Make a Digital Twin of your Data Center*, https://developer.nvidia.com/blog/make-a-digital-twin-of-your-data-center-with-sonic-running-on-nvidia-air/, (Accessed on October 2022), 2021.

[45] Liu H., Zhu Y., Padhye J., Cao J., Tallapragada S., Lopes N., Rybalchenko A., Lu G. y Yuan L., «CrystalNet: Faithfully Emulating Large Production Networks,» en *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*, ACM, oct. de 2017, págs. 599-613, ISBN: 978-1-4503-5085-3. dirección: https://www.microsoft.com/en-us/research/publication/crystalnet-faithfully-emulating-large-production-networks/.

[46] Hong H., Wu Q., Dong F., Song W., Sun R., Han T., Zhou C. y Yang H., «NetGraph: An Intelligent Operated Digital Twin Platform for Data Center Networks,» en *Proceedings of the ACM SIGCOMM 2021 Workshop on Network-Application Integration*, ép. NAI'21, Virtual Event, USA: Association for Computing Machinery, 2021, págs. 26-32, ISBN: 9781450386333. DOI: 10.1145/3472727.3472802. dirección: https://doi.org/10.1145/3472727.3472802.

[47] Li Y., Yin X., Wang Z., Yao J., Shi X., Wu J., Zhang H. y Wang Q., «A Survey on Network Verification and Testing With Formal Methods: Approaches and Challenges,» *IEEE Communications Surveys Tutorials*, vol. 21, n.º 1, págs. 940-969, 2019. DOI: 10.1109/COMST.2018.2868050.

[48] Meza J., Xu T., Veeraraghavan K. y Mutlu O., «A Large Scale Study of Data Center Network Reliability,» en *Proceedings of the Internet Measurement Conference 2018*, ép. IMC '18, Boston, MA, USA: Association for Computing Machinery, 2018, págs. 393-407, ISBN: 9781450356190. DOI: 10.1145/3278532.3278566. dirección: https://doi.org/10.1145/3278532.3278566.

[49] nsnam, *ns-3 Network Simulator*, https://www.nsnam.org/, (Accessed on October 2022), 2022.

[50] Ltd. O., *OMNeT++ - Discrete Event Simulator*, https://omnetpp.org/, (Accessed on October 2022), 2022.

[51] Muelas D., Ramos J. y Vergara J. E. L. d., «Assessing the Limits of Mininet-Based Environments for Network Experimentation,» *IEEE Network*, vol. 32, n.º 6, págs. 168-176, 2018. DOI: 10.1109/MNET.2018.1700277.

[52] Pediaditakis D., Rotsos C. y Moore A. W., «Faithful reproduction of network experiments,» en *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014, págs. 41-52.

[53] Bonofiglio G., Iovinella V., Lospoto G. y Di Battista G., «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*, 2018, págs. 1-9. DOI: 10.1109/NOMS.2018.8406267.

[54] Scazzariello M., Ariemma L. y Caiazzi T., «Kathará: A Lightweight Network Emulation System,» en *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, págs. 1-2. DOI: 10.1109/NOMS47738.2020.9110351.

[55] Merkel D., «Docker: lightweight linux containers for consistent development and deployment,» *Linux journal*, vol. 2014, n.º 239, pág. 2, 2014.

[56] Jakma P. y Lamparter D., «Introduction to the quagga routing suite,» *IEEE Network*, vol. 28, n.º 2, págs. 42-48, 2014. DOI: 10.1109/MNET.2014.6786612.

[57] FRRouting, *FRRouting*, https://frrouting.org/, (Accessed on October 2022), 2021.

[58] Lumbis P., *FRR: the most popular network router you've never heard of*, https://www.nextplatform.com/2020/10/26/frr-the-most-popular-network-router-youve-never-heard-of/, (Accessed on October 2022), 2021.

[59] Kathará, *Kathará*, https://github.com/KatharaFramework/Kathara, (Accessed on October 2022), 2021.

[60] Scazzariello M., Ariemma L., Battista G. D. y Patrignani M., «Megalos: A Scalable Architecture for the Virtualization of Network Scenarios,» en *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, págs. 1-7. DOI: 10.1109/NOMS47738.2020.9110288.

[61] Kubernetes, *Kubernetes*, https://kubernetes.io/, (Accessed on October 2022), 2021.

[62] Lantz B., Heller B. y McKeown N., «A Network in a Laptop: Rapid Prototyping for Software-Defined Networks,» en *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ép. Hotnets-IX, Monterey, California: Association for Computing Machinery, 2010, ISBN: 9781450304092. DOI: 10.1145/1868447.1868466. dirección: https://doi.org/10.1145/1868447.1868466.

[63] Ruslan R., Othman N. B., Fuzi M. M. y Ghazali N., «Scalability Analysis in Mininet on Software Defined Network using ONOS,» en *2020 Emerging Technology in Computing, Communication and Electronics (ETC-CE)*, 2020, págs. 1-6. DOI: 10.1109/ETCCE51779.2020.9350892.

[64] Foundation O. N., *Open Network Operating System (ONOS*, https://opennetworking.org/onos/, (Accessed on October 2022), 2022.

[65] Wette P., Dräxler M., Schwabe A., Wallaschek F., Zahraee M. H. y Karl H., «MaxiNet: Distributed emulation of software-defined networks,» en *2014 IFIP Networking Conference*, 2014, págs. 1-9. DOI: 10.1109/IFIPNetworking.2014.6857078.

[66] Karypis Lab U. o. M., *METIS*, https://github.com/KarypisLab/METIS, (Accessed on October 2022), 2020.

[67] Ahrenholz J., Danilov C., Henderson T. R. y Kim J. H., «CORE: A real-time network emulator,» en *MILCOM 2008 - 2008 IEEE Military Communications Conference*, 2008, págs. 1-7. DOI: 10.1109/MILCOM.2008.4753614.

[68] Ahrenholz J., «Comparison of CORE network emulation platforms,» en *2010 - MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*, 2010, págs. 166-171. DOI: 10.1109/MILCOM.2010.5680218.

[69] Technologies G., *GNS3: The software that empowers network professionals*, https://www.gns3.com/, (Accessed on October 2022), 2021.

[70] Riley G. F. y Henderson T. R., «The ns-3 Network Simulator,» en *Modeling and Tools for Network Simulation*, Wehrle K., Güneş M. y Gross J., eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, págs. 15-34, ISBN: 978-3-642-12331-3. DOI: 10.1007/978-3-642-12331-3_2. dirección: https://doi.org/10.1007/978-3-642-12331-3_2.

[71] Orebaugh A., Ramirez G., Beale J. y Wright J., *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing, 2007, ISBN: 1597490733.

[72] Group T. T., *Tcpdump & Libpcap*, https://www.tcpdump.org/, (Accessed on October 2022), 2022.

[73] Wireshark, *tshark Manual Page*, https://www.wireshark.org/docs/man-pages/tshark.html, (Accessed on October 2022), 2022.

[74] KimiNewt, *Pyshark*, https://github.com/KimiNewt/pyshark, (Accessed on October 2022), 2021.

[75] Sikos L. F., «Packet analysis for network forensics: A comprehensive survey,» *Forensic Science International: Digital Investigation*, vol. 32, pág. 200 892, 2020, ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2019.200892. dirección: http://www.sciencedirect.com/science/article/pii/S1742287619302002.

[76] Chapman C., «Chapter 7 - Using Wireshark and TCP dump to visualize traffic,» en *Network Performance and Security*, Chapman C., ed., Boston: Syngress, 2016, págs. 195-225, ISBN: 978-0-12-803584-9. DOI: https://doi.org/10.1016/B978-0-12-803584-9.00007-X. dirección: http://www.sciencedirect.com/science/article/pii/B978012803584900007X.

[77] Caiazzi T., Scazzariello M., Alberro L., Ariemma L., Castro A., Grampin E. y Battista G. D., «Sibyl: a Framework for Evaluating the Implementation of Routing Protocols in Fat-Trees,» en *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, págs. 1-7. DOI: 10.1109/NOMS54207.2022.9789876.

[78] Gill P., Jain N. y Nagappan N., «Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications,» ép. SIGCOMM '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, págs. 350-361, ISBN: 9781450307970. DOI: 10.1145/2018436.2018477. dirección: https://doi.org/10.1145/2018436.2018477.

[79] Caiazzi T., «Software Defined Data Centers: methods and tools for routing protocol verification and comparison,» Tesis de mtría., Università degli studi Toma Tre, 2019.

[80] panandr, *mininet-fattree*, https://github.com/panandr/mininet-fattree, (Accessed on October 2022), 2016.

[81] Mininet, *Mininet Python API Reference Manual*, http://mininet.org/api/annotated, (Accessed on October 2022), 2021.

[82] Caiazzi T., Scazzariello M., Alberro L., Ariemma L., Castro A., Grampin E. y Battista G. D., *Sibyl: a Framework for Evaluating the Implementation of Routing Protocols in Fat-Trees*, https://gitlab.com/uniroma3/compunet/networks/sibyl-framework/sibyl, (Accessed on October 2022), 2021.

[83] Caiazzi T., Scazzariello M. y Ariemma L., «VFTGen: a Tool to Perform Experiments in Virtual Fat Tree Topologies,» en *IM 2021 - 2021 IFIP/IEEE International Symposium on Integrated Network Management*, 2021, págs. 1-2.

[84] Azpiroz S y Velázquez F., «Extensión del simulador de redes ns-3 para ejecutar el software de enrutamiento FRRouting,» Tesis de Grado, Universidad de la República, 2021.

[85] Azpiroz S. y Velázquez F., *FRR ns-3 DCE*, https://gitlab.com/fing-mina/datacenters/frr-ns3, (Accessed on October 2022), 2021.

[86] Emulator C. O. R., *Common Open Research Emulator - Python API*, http://coreemu.github.io/core/python, (Accessed on October 2022), 2021.

[87] rubiruchi, *Fat Tree Topology in Mininet and POX*, https://github.com/rubiruchi/252-project, (Accessed on October 2022), 2019.

[88] Huangmachi, *PureSDN*, https://github.com/Huangmachi/PureSDN, (Accessed on October 2022), 2107.

[89] Alberro L., *Experimentation environments for routing protocols in the datacenter*, https://gitlab.com/fing-mina/datacenters/fat-tree-emulators, (Accessed on October 2022), 2021.

[90] Lucero M y Parnizari A., «Escalabilidad del enrutamiento en data centers masivos,» Tesis de Grado, Universidad de la República, 2021.

[91] Alberro L., Castro A. y Grampin E., «Experimentation Environments for Data Center Routing Protocols: A Comprehensive Review,» *Future Internet*, vol. 14, n.º 1, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14010029. dirección: https://www.mdpi.com/1999-5903/14/1/29.

[92] Alberro L., Velázquez F., Azpiroz S., Grampin E. y Richart M., «Experimenting with Routing Protocols in the Data Center: An ns-3 Simulation Approach,» *Future Internet*, vol. 14, n.º 10, 2022, ISSN: 1999-5903. DOI: 10.3390/fi14100292. dirección: https://www.mdpi.com/1999-5903/14/10/292.

[93] Alberro L., *UltraViolet*, https://github.com/leoalb/UltraViolet, (Accessed on October 2022), 2022.

[94] bluesaiyancodes, *UltraViolet*, https://github.com/bluesaiyancodes/UltraViolet, (Accessed on October 2022), 2021.

[95] Qin Y., Yang W., Ye Y. y Shi Y., «Analysis for TCP in data center networks: Outcast and Incast,» *Journal of Network and Computer Applications*, vol. 68, págs. 140-150, 2016, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2016.04.014. dirección: https://www.sciencedirect.com/science/article/pii/S1084804516300649.

[96] Montazeri B., Li Y., Alizadeh M. y Ousterhout J., «Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities,» en *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ép. SIGCOMM '18, Budapest, Hungary: Association for Computing Machinery, 2018, págs. 221-235, ISBN: 9781450355674. DOI: 10.1145/3230543.3230564. dirección: https://doi.org/10.1145/3230543.3230564.

[97] Alasmar M., Parisis G. y Crowcroft J., «SCDP: Systematic Rateless Coding for Efficient Data Transport in Data Centers,» *IEEE/ACM Transactions on Networking*, vol. 29, n.º 6, págs. 2723-2736, 2021. DOI: 10.1109/TNET.2021.3098386.

[98] BRAIKI K. y YOUSSEF H., «Resource Management in Cloud Data Centers: A Survey,» en *2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC)*, 2019, págs. 1007-1012. DOI: 10.1109/IWCMC.2019.8766736.