UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE INGENIERIA

Extensión del simulador de redes ns-3 para ejecutar el software de enrutamiento FRRouting

Proyecto de grado

Autores:
Sara Azpiroz
Felipe Velázquez

Supervisores:
Eduardo Grampín
Matías Richart

Instituto de Computación, Facultad de Ingeniería
Universidad de la República

Diciembre 2021

Resumen

En el contexto de los datacenters modernos, el despliegue de la topología CLOS plegada (o fat tree) se ha mostrado como una alternativa a los diseños tradicionales, que resultaron poco adecuados a las necesidades de escala, resiliencia y comunicación. Por otro lado, la migración al uso del protocolo de ruteo BGP en el datacenter significó, entre otras cosas, una manera de aprovechar las ventajas de la conectividad de la topología fat tree.

Sin embargo, aún surgen motivaciones de estudiar el desempeño del protocolo en diferentes construcciones de fat trees. Dado que hacerlo sobre infraestructura real es inviable, se recurre a ambientes emulados/simulados. Basado en experiencias previas, surgió la iniciativa de utilizar el simulador de red `ns-3` que cuenta con un módulo `DCE` para incorporar código nativo en las simulaciones. De esta manera se origina el objetivo de realizar los cambios y extensiones necesarias que permitan ejecutar dentro de `ns-3 DCE` la familia de protocolos `FRR`, particularmente la implementación del protocolo BGP para su uso en el datacenter.

En este trabajo describimos los pasos realizados para lograr dicha extensión, lo que significó un estudio de la estructura y funcionamiento del simulador y su módulo `DCE`. Además, detallamos lo requerido para su instalación junto con las modificaciones y agregados que fueron necesarios para el funcionamiento de `FRR` en `ns-3 DCE`. Conseguido esto, se implementaron casos de prueba para verificarlo, así como para estudiar el comportamiento de BGP en el datacenter. Esto último consiste en estudiar la convergencia del protocolo para diferentes construcciones de fat trees, lo cual representa el objetivo final de todo el proyecto.

Como complemento, mencionamos limitantes encontradas durante la ejecución de las pruebas. En este sentido investigamos la posibilidad de optimizar el tiempo de ejecución y escala de las simulaciones mediante el uso de paralelismo. A tal fin, realizamos un análisis del problema junto con posibles estrategias de solución.

Palabras clave: Simulación, `ns-3`, Direct Code Execution (`DCE`), `FRR`, datacenter, fat tree, BGP.

Índice

1. Introducción

En este documento se presenta el proyecto de grado realizado por estudiantes de la *Facultad de Ingeniería UdelaR* [1] bajo la supervisión de profesores tutores. El proyecto se enmarca en el área de redes de computadoras, y consiste en estudiar y desarrollar los cambios y extensiones necesarias que permitan ejecutar la familia de protocolos de ruteo FRRouting (`FRR`) [2] dentro del simulador de redes `ns-3` [3]. A este proceso lo llamaremos porte de `FRR` a `ns-3 DCE`.

`FRR` es un conjunto de protocolos de enrutamiento de Internet de código abierto y gratuito para plataformas Linux y Unix. Implementa los protocolos BGP, OSPF, RIP, IS-IS, entre otros y tiene sus raíces en el proyecto `Quagga` [4], que también es un conjunto de software de enrutamiento. `FRR` fue iniciado por muchos desarrolladores de `Quagga` para mejorar las bases ya establecidas, dada la frustración de algunos de ellos con el ritmo de desarrollo del proyecto. El funcionamiento general es por tanto similar y `FRR` ha remplazado a `Quagga` en popularidad.

Por otro lado, `ns-3` es un simulador de redes de eventos discretos para sistemas de Internet, de amplio respaldo en la comunidad de redes. `ns-3` tiene un modo de ejecución denominado Direct Code Execution (`DCE`) [5], que permite utilizar código nativo (compilado adecuadamente) en las simulaciones. De esta manera es posible ejecutar dentro de `ns-3` implementaciones existentes de protocolos de red o aplicaciones.

De este modo, como primer paso buscamos realizar las implementaciones necesarias para que `ns-3` pueda dar soporte a `FRR`, con el fin de desarrollar simulaciones en `ns-3 DCE` que utilicen código de `FRR`. Si bien `FRR` implementa un conjunto de protocolos de red, nuestro alcance es dar soporte a la implementación provista para BGP. Para la implementación de las simulaciones, nos centraremos en un tipo específico de topología CLOS, denominada fat tree, que es ampliamente utilizada en los datacenters masivos. Se busca estudiar el comportamiento del protocolo BGP allí.

1.1. Antecedentes y motivación

Las redes de telecomunicaciones presentan desafíos crecientes, tanto en complejidad como en escala. A modo de ejemplo, los datacenters masivos con decenas de miles de nodos de cómputo y algunos miles de nodos de comunicaciones, superan en complejidad y escala a la Internet tradicional.

En este marco, para dar soporte a este y otros casos de uso, se han desarrollado nuevas técnicas de virtualización de red y nuevos protocolos de enrutamiento escalables. De todas maneras, para desarrollar y experimentar con nuevos protocolos y arquitecturas no es posible contar con infraestructuras físicas, las cuales resultan económicamente inaccesibles.

Surge pues la necesidad de contar con entornos de emulación y/o simulación que den soporte a estas iniciativas. En este sentido, los emuladores/simuladores de redes hacen posible la innovación y experimentación con nuevos protocolos y arquitecturas, obteniendo resultados representativos de la realidad y reduciendo significativamente los

costos.

Emulación se refiere a la capacidad de un software en un dispositivo para imitar otro programa o dispositivo. De este modo, los emuladores permiten ejecutar programas en una plataforma (sea una arquitectura de hardware o un sistema operativo) diferente de aquella para la cual fueron escritos originalmente, dado que imitan el hardware y el software del dispositivo de destino [6].

En los últimos dos años, en la facultad se ha estado trabajando con emulaciones centradas en el ya mencionado caso de uso del datacenter masivo, utilizando entornos de emulación tales como *Mininet* [7] y *Kathará* [8]. En estos entornos se han probado protocolos tales como *BGP en el datacenter*, *Routing in Fat Trees (RIFT)* y *Openfabric*, utilizando la implementación proporcionada por `FRR`.

Sin embargo, las emulaciones presentan algunos inconvenientes, por ejemplo no permiten analizar a fondo los aspectos temporales. Los mismos están afectados por factores externos al experimento emulado, por ejemplo, el sistema operativo que soporta al emulador.

Para superar estas limitaciones, se busca entonces la utilización de simulaciones. Una simulación es una imitación aproximada del funcionamiento de un proceso o sistema, que representa su funcionamiento en el tiempo. De este modo, mientras que un emulador trata de modelar de forma precisa el dispositivo, un simulador se enfoca en reproducir el comportamiento del programa o dispositivo [6].

La motivación de este proyecto es entonces contar con un simulador que pueda ejecutar código de `FRR`, para estudiar el comportamiento de BGP sobre fat trees.

Como comentario, el trabajo realizado previamente en emuladores y los resultados obtenidos (principalmente el trabajo del grupo MINA sobre el emulador Kathará), proporcionan un escenario de comparación y verificación para la migración a ambientes simulados.

1.2. Objetivos

Nuestro principal objetivo es trabajar con `ns-3` utilizando su modo de ejecución `DCE`. Buscamos con esto desarrollar simulaciones donde se ejecute el protocolo BGP, provisto por `FRR`, en una topología fat tree. Con esto en mente se proponen los siguientes objetivos:

1. Conocer detalladamente el funcionamiento de `ns-3` en modo `DCE`.
2. Estudiar y desarrollar los cambios y extensiones necesarias para que la implementación de BGP provista por `FRR` pueda ser ejecutado sobre `ns-3` en modo `DCE`.
3. Evaluar el porte realizado, validando el funcionamiento del mismo.
4. Desarrollar casos de prueba relevantes de BGP en fat trees incluyendo aspectos de performance y escala.
5. Validar los casos de prueba realizados.

1.3. Resultados esperados

La realización del proyecto proporcionará los siguientes entregables:

1. El presente informe, comprendiendo la descripción del trabajo de adaptación de `FRR` para que se pueda ejecutar en el simulador `ns-3` en modo `DCE`.
2. Sección experimental incluyendo descripción de los experimentos realizados y los resultados obtenidos.
3. Contribución al proyecto de código abierto `ns-3 DCE` con las extensiones realizadas durante el proyecto.
4. Repositorio de código que permita difundir y reutilizar el trabajo.

1.4. Organización del informe

Dado que nuestro objetivo es portar `FRR` a `ns-3 DCE` para probar BGP sobre fat trees, que como mencionamos, es una topología ampliamente utilizada en los datacenters modernos, en la sección 2 describimos la estructura de la topología. Además presentamos una breve reseña sobre los datacenters tradicionales en contraposición con los actuales, y la motivación que generó el despliegue de esta nueva topología. Conjuntamente explicamos el uso del protocolo BGP para el datacenter.

En la sección 3 nos adentramos en el simulador de red `ns-3` y explicamos su estructura y funcionamiento. Además explicamos con detalle el procedimiento que seguimos para instalar el mismo junto a su módulo `DCE`. De la misma manera explicamos los cambios y agregados para portar `FRR` a `ns-3 DCE`, así como facilidades para utilizar el porte.

La sección 4 detalla las simulaciones implementadas utilizando el porte, junto con las limitantes encontradas durante su ejecución y los datos obtenidos. Una de las principales limitantes observadas es el tiempo de ejecución, la demora para simulaciones con muchos nodos es muy alta. Se probaron distintas estrategias para reducir el impacto de esta limitante con buenos resultados.

Continuando con el problema de tiempos de ejecución muy lentos, en la sección 5 investigamos la posibilidad de optimizarlo mediante el uso de paralelismo en `ns-3 DCE`. A tal fin, se analizan y describen posibles formas de abordar el problema, las cuales pueden servir de punto de partida para trabajos futuros.

Finalmente, la sección 6 hace un recuento de las conclusiones del trabajo realizado y propuestas de trabajo a futuro.

2. Topología y ruteo en datacenters

Un datacenter es una infraestructura utilizada para alojar sistemas informáticos que puedan procesar, servir o almacenar datos. En este sentido, el datacenter aloja a los servidores necesarios para soportar estos servicios ofrecidos a los clientes. Mediante switches todos los servidores reciben y entregan información desde la red y hacia la red según la demanda y el trabajo al que estén destinados [9].

La gran cantidad de datos y actividad manejada por los datacenters en la actualidad ha motivado escalar su infraestructura. Por eso se busca construir la misma en base a una topología que sea eficiente a tal fin.

A medida que la infraestructura, datos y actividad aumentan, también lo hace la necesidad de comunicación entre componentes. En este sentido, el ruteo provee a los switches del datacener información sobre la red interna, para que los mismos sepan a qué componente enviar la información para que llegue a destino. Además permite que estos sean notificados de la ocurrencia de algún cambio en la red.

En esta sección introducimos la topología para datacenters que tenemos como objetivo desplegar en `ns-3` y su comparación con formas tradicionales. Además explicamos por qué BGP cobró relevancia como protocolo de ruteo en el despliegue de datacenters actuales, lo que motiva portar el mismo a `ns-3` para ser utilizado en las simulaciones.

2.1. Topología tradicional en datacenters

Los datacenters tradicionales implementaban una topología jerárquica [10] (también conocida como *tree-based*), que consiste en 3 niveles: una primera capa denominada central (*core*), una capa dos denominada de agregación (*aggregation*), y una capa tres denominada de acceso (*access*), como se muestra en la Figura 1.

Los principales problemas que enfrenta esta arquitectura incluyen escalabilidad, tolerancia a fallas, ancho de banda transversal y no menos importante, que utiliza dispositivos de red muy costosos y que consumen mucha energía [11].

Con respecto a la utilización del ancho de banda, por el diseño de esta topología cualquier tráfico de capa tres (capa de red en el modelo TCP/IP) debe salir del rack y alcanzar la capa de agregación antes de ser enrutado, incluso de regreso al mismo rack del que proviene. Esto da como resultado un exceso de consumo de ancho de banda hacia el nivel central de la red.

En la actualidad, como resultado de la virtualización de servidores, arquitectura de microservicios y la nube privada, entre otros, se incrementaron estos problemas a medida que el tráfico este-oeste[1] se volvió aún más frecuente. La virtualización esencialmente ubica las máquinas virtuales de manera aleatoria, y los servidores podrían estar en

[1]El tráfico este-oeste indica el flujo de datos entre servidores dentro de un mismo datacenter, mientras que el tráfico de norte-sur indica el flujo de datos que entra/sale del datacenter.

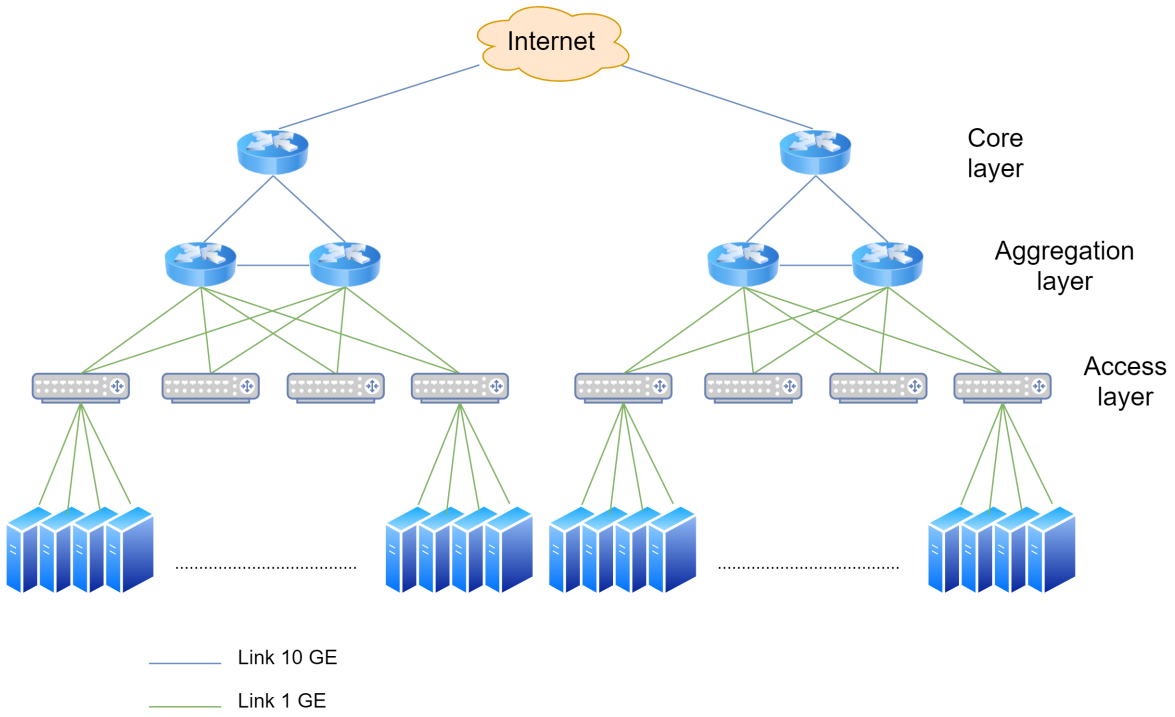

Figura 1: Topología tradicional en datacenters. En la capa uno (core) se encuentran routers de alto rendimiento, conectados a más routers en la capa dos (aggregation). La capa tres (access) se compone por racks de servidores, conectados a un switch (comunmente llamado top of rack).

cualquier lugar dentro de la infraestructura. Por lo tanto, el aumento de la comunicación este-oeste hace que el tráfico se dirija a la capa de agregación y con frecuencia al nivel central.

Existen problemas adicionales con las redes tradicionales, por ejemplo, una vez que se establece esta infraestructura de red jerárquica, escalarla es difícil. Otro rack de servidores no solo significa otro switch de acceso, sino posiblemente otro router de agregación o incluso más puertos en el router central. Además, a veces resulta muy difícil poder adquirir dispositivos de capa uno (core) con una densidad de puertos lo suficientemente grande como para escalar suficientemente la capa dos (aggregation). Finalmente, también se presenta el problema de que la visibilidad del tráfico es limitada y la depuración un desafío.

2.2. Necesidades de los datacenters modernos

Los centros de datos en la actualidad son mucho más grandes de lo que eran hace una década, con requisitos de aplicación (por ejemplo de servidor) muy diferentes de los del cliente tradicional, y con velocidades de implementación que son en segundos en lugar de días [12].

Entre las necesidades de los datacenters modernos se pueden encontrar:

- `Comunicación:` Como mencionamos antes, las aplicaciones de los datacenters implican mucha comunicación de servidor a servidor (este-oeste), principalmente las aplicaciones de búsqueda, microservicios y de nube.
- `Escala:` Los datacenters modernos van desde cientos hasta cientos de miles de servidores en un única ubicación física. El aumento de la comunicación servidor a servidor combinado con los requisitos de conectividad en tales escalas, obligan a repensar cómo se construyen tales redes.
- `Resiliencia:` El paradigma de hoy día en la construcción de aplicaciones e infraestructuras, no descansa sobre una red confiable, sino que se diseña para trabajar ante la presencia de fallos, afectando lo menos posible la experiencia del usuario final.

Por lo tanto las necesidades de las aplicaciones y la escala de la operación actual cambian la forma en que se diseñan y despliegan las redes para los datacenters modernos.

2.3. Topología CLOS para datacenters modernos

La topología CLOS fue diseñada dada la complejidad que estaban presentando los despliegues de otras topologías. Como mencionamos antes, necesitaban swtiches/routers de gran capacidad que resultaban costosos y complejos. Además los fallos repercutían a toda la red.

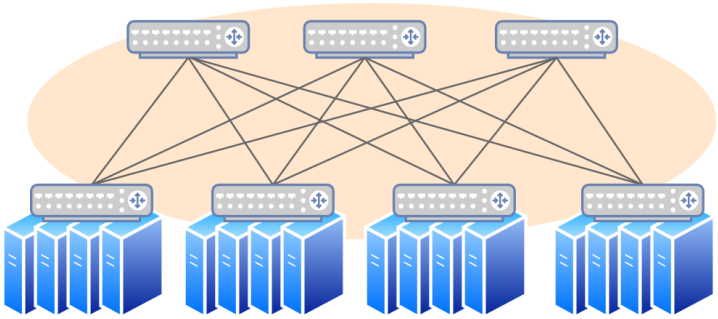

Figura 2: [*Elaboración propia en base a imagen de* [12]] Topología CLOS plegada de dos capas (fat tree de dos capas). Los nodos grises representan los switches y los azules los servidores, agrupados en racks.

Una opción común para una topología CLOS escalable horizontalmente, es una topología CLOS plegada, a veces llamada *fat tree*. En la Figura 2 se muestra un fat tree de dos capas en su forma más simple. La característica relevante es que todos los switches de la misma capa no están conectados entre sí. A su vez, cada switch de una capa se conecta con todos los de la capa siguiente.

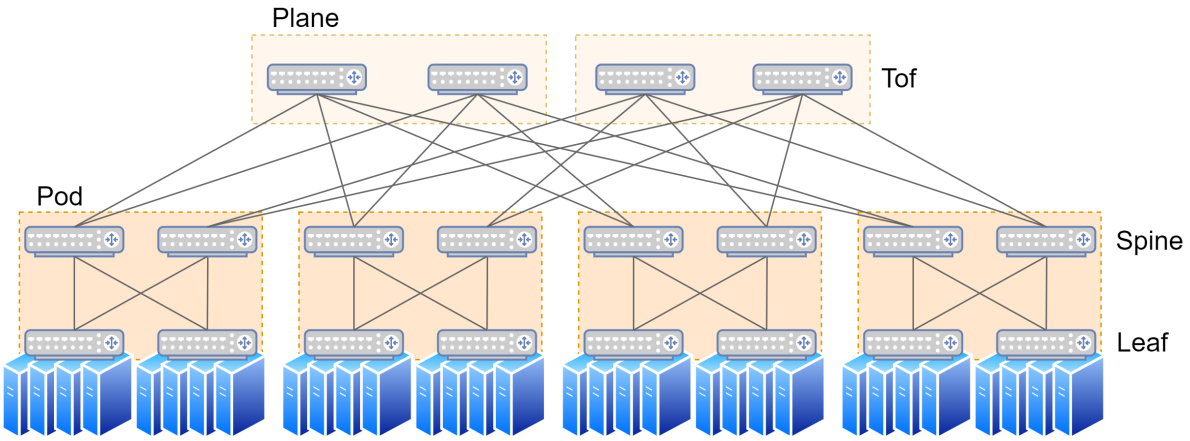

Figura 3: [*Elaboración propia en base a imagen de* [12]] Fat tree de tres capas. Consiste en conectar topologías de dos capas (Pods) agregando una capa de swtiches adicional que se agrupan en planos. Se construyó con parámetros $k\_leaf = 2$, $k\_top = 2$ y $redundancy = 1$.

Un fat tree de tres capas consiste en conectar topologías de dos capas (a las que llamamos *Pods*), donde además se añade un nivel más de switches. En la Figura 3 se muestra un ejemplo. Los switches del nivel superior son llamados *Tof nodes* (top of fabric), los del siguiente nivel *Spine nodes*, y los del nivel inferior *Leaf nodes* (también conocidos como top of rack).

De esta manera, cada servidor del rack se conecta a un único Leaf, y dentro de un Pod, cada Leaf se conecta a todos los Spines. Toda la funcionalidad está en los servidores, mientras que los Spines y Tofs actúan solo como conectores.

Se pueden construir diferentes instancias de fat trees variando los siguientes parámetros:

- `k_leaf`: Se corresponde con el número de puertos de un Leaf dividido dos. Es decir, el número de puertos de un Leaf apuntando hacia norte o sur.
- `k_top`: Se corresponde con el número de puertos de un Spine dividido dos. Es decir, el número de puertos de un Spine apuntando hacia norte o sur.
- `redundancy`: número de conexiones de un Tof a un Pod.
- `n_pods`: número de Pods, siendo el máximo:

$$\frac{k\_leaf + k\_top}{redundancy}.$$

De los parámetros detallados anteriormente, se derivan estos otros:

- `n_tofs_per_plane`: Los Tofs se agrupan en planos. El número de Tofs por plano resulta k_top.

- `n_planes`: El número de planos queda determinado por:

$$\frac{k\_leaf}{redundancy}.$$

- `n_tofs`: El número de Tofs queda determinado por:

$$k\_top \cdot \frac{k\_leaf}{redundancy}.$$

- `n_spines_per_tof`: El número de Spines dentro de un Pod, a las que un Tof se conecta queda determinado por:

$$redundancy \cdot n\_pods = k\_leaf + k\_top.$$

- `n_max_servers`: El número máximo de servidores que soporta la topología está dado por:

$$k\_leaf \cdot k\_top \cdot n\_pods = \frac{k\_leaf^2 \cdot k\_top + k\_leaf \cdot k\_top^2}{redundancy}.$$

  En el caso donde $k = k\_leaf = k\_top$, esta fórmula se reduce a:

$$\frac{2k^3}{redundancy}.$$

De esta manera la topología es fácilmente parametrizable, siendo suficiente proporcionar los parámetros k_leaf, k_top y $redundancy$ para su construcción. Lo que se destaca de esta topología es:

- Cada servidor generalmente está a tres saltos de cualquier otro servidor.
- Los nodos son homogéneos, esto es, todos los switches son iguales independientemente de la capa a la que pertenecen. Además, al no haber heterogeneidad en los switches, son más fáciles de mantener y reponer. Adicionalmente las fallas que se generan son más sencillas de identificar, lo que añade resiliencia a la red.
- La matriz de conexión es densa, lo que permite manejar de forma transparente las fallas. Por ejemplo, se logra proteger de desconexión total, ante la falla de un enlace, donde en otras topologías esto afectaba una fracción importante del ancho de banda. Por lo tanto, la densa interconexión existente reduce el radio de falla.
- El ancho de banda puede ser aumentado agregando, por ejemplo, más Spines.
- El objetivo de una configuración CLOS es ser simple y automatizada, lo que se ve reflejado en su diseño fractal. La automatización es posible porque las partes son repetitivas: piezas cada vez más grandes se ensamblan esencialmente a partir de los mismos bloques de construcción.

Por todo lo anterior, las redes CLOS permiten la construcción no solo de redes altamente escalables, pero también redes muy resistentes.

Además pueden añadirse mejoras, por ejemplo proveer dualidad conectando los servidores a otro Leaf adicional y haciendo que ambos Leaves sean uno lógicamente. Esto se conoce en capa dos como *link aggregation*. Esencialmente, el servidor cree que está conectado a un solo Leaf con un enlace, mientras en realidad son dos enlaces conectados a él. Esto proporciona la ilusión, desde un perspectiva de protocolo principalmente, que son uno solo enlace.

2.4. BGP en el datacenter

Border Gateway Protocol (BGP) se conoce desde hace décadas por ayudar a los sistemas conectados a Internet de todo el mundo a encontrarse unos con otros. Sin embargo, también se puso de manifiesto su utilidad dentro de los datacenters, y hoy día es el protocolo de enrutamiento más común utilizado en los mismos. Sabiendo esto, implementaciones de BGP como `FRR` tienen dos series de parámetros por defecto, siendo una de ellas la apropiada para el uso en datacenters.

Antes de explicar por qué BGP se usa en los datacenters y cómo adaptarlo, mencionamos que en las capas de agregación de las redes tradicionales (lo análogo a los Tofs de la topología CLOS), se solía usar bridging (enrutamiento de capa 2) en lugar de routing (enrutamiento de capa 3) [12].

Bridging utiliza protocolos como spanning tree protocol (STP), lo que rompe la poblada conectividad de la matriz de la red CLOS, creando un árbol sin ciclos. La construcción de un árbol como el generado por STP no solo reduce la conectividad, sino que conlleva ineficiencias ante la caída de un enlace.

Routing por otro lado, es capaz de utilizar todos los caminos, tomando ventaja de la rica matriz de conectividad de una red CLOS. Routing también puede tomar el camino más corto o programarse para tomar un camino más largo para una mejor utilización general del enlace. Utilizar routing en lugar de bridging también hace que se necesiten menos protocolos para hacer funcionar la red.

BGP utiliza routing y se puede ajustar para un uso eficiente en el datacenter, afirmándose como el protocolo de enrutamiento por elección. Es un protocolo simple de entender, permite el funcionamiento de Internet y existen un montón de implementaciones robustas y de código abierto (como `FRR`). Además cuenta con otras ventajas, como soporte a multiprotocolo.

De todas maneras, se necesita adaptar BGP a su uso en los datacenters, dado que difiere al uso extendido en las redes de proveedores de servicios de Internet, destacándose:

- En su uso común, BGP aprende rutas a través de protocolos internos de ruteo, pero en el datacenter BGP es el protocolo interno de ruteo.
- Hay mucha más conectividad en los datacenters en comparación a la conectividad entre sistemas autónomos (AS, del inglés *Autonomous System*).

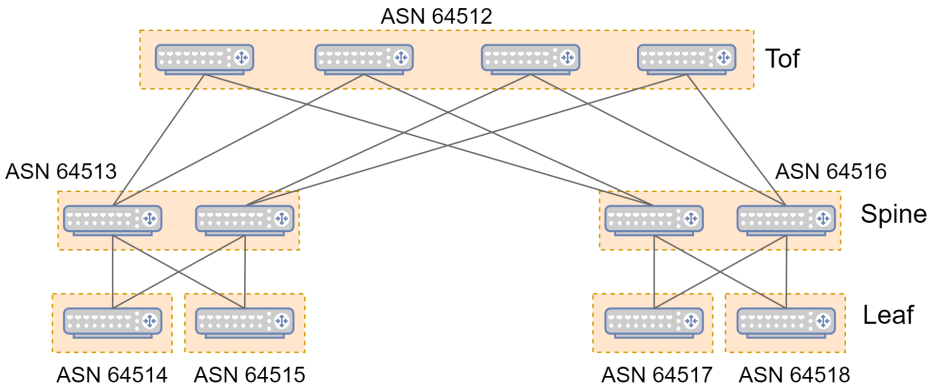

Figura 4: Ejemplo de como es la asignación de ASNs en un fat tree.

- Se necesita rápida actualización de los cambios, por lo que se prioriza la rápida convergencia en lugar de estabilidad.
- Se utiliza eBGP en lugar de iBGP, dado que es más simple de entender e implementar y cuenta con variedad de implementaciones robustas.
- Para los números de los sistemas autónomos (ASN, *Autonomous System Number*) se utiliza una numeración privada y se asigna de tal forma que todos los Tofs tengan el mismo número en la red, los Spines tengan el mismo número dentro del Pod (difiriendo la numeración entre Spines de Pods distintos), y todos los Leaves tengan diferente numeración. En la Figura 4 se muestra un ejemplo de esto.

En la Figura 5 se muestra un ejemplo de archivo de configuración de BGP para un nodo Tof de la topología CLOS. En la línea 14 se define el número de ASN siguiendo el criterio mencionado anteriormente.

Adicionalmente, se sugiere dejar por defecto las siguientes características en la implementación brindada por `FRR` para su uso en los datacenters:

- Se desactiva para la versión por defecto en el datacenter el requerimiento de política, permitiendo que se importen/exporten las rutas de los eBGP peers (línea 17).
- Permitir el multi-path en lugar del best-path.
- Se utiliza multi-path dentro de las rutas de igual costo, indicando que para esto solo se tome en cuenta el largo de las rutas, permitiendo que no contengan exactamente la misma lista de ASNs. Esto hace que si el largo del AS_PATH de dos fuentes distintas es el mismo, el algoritmo no verifica una coincidencia exacta de los ASNs en las listas, y sigue con el siguiente criterio (línea 18).
- Con respecto a los timers, se deja por defecto el valor advertisement interval en 0 segundo, keepalive en 3 segundos, hold timers en 9 segundos y connect timer en 60 segundos (línea 15).

```
1  frr version 7.7-dev-frr-ns3-dce
2  frr defaults datacenter
3  !
4  log stdout
5  !
6  debug bgp updates in
7  debug bgp updates out
8  !
9  ip prefix-list DC_LOCAL_SUBNET seq 5 permit 10.0.0.0/8 le 30
10 ip prefix-list DC_LOCAL_SUBNET seq 10 permit 200.0.0.0/8 le 24
11 route-map ACCEPT_DC_LOCAL permit 10
12  match ip address prefix-list DC_LOCAL_SUBNET
13 !
14 router bgp 64512
15  timers bgp 3 9
16  bgp router-id 192.168.0.25
17  no bgp ebgp-requires-policy
18  bgp bestpath as-path multipath-relax
19  bgp bestpath compare-routerid
20 !
21 neighbor fabric peer-group
22  neighbor fabric remote-as external
23  neighbor fabric advertisement-interval 0
24  neighbor fabric timers connect 5
25  neighbor sim0 interface peer-group fabric
26  neighbor sim1 interface peer-group fabric
27  neighbor sim2 interface peer-group fabric
28  neighbor sim3 interface peer-group fabric
29 !
30 address-family ipv4 unicast
31   neighbor fabric activate
32   maximum-paths 64
33 exit-address-family
```

Figura 5: Ejemplo de archivo de configuración de BGP para un nodo Tof de la topología CLOS.

Otras características relevantes del archivo de configuración, es que en la línea 21 se crea peer group denominado *fabric*. Este grupo contiene los Spines y los Tofs. Para el caso del grupo que contiene los Spines y los Leaves lo llamamos *TOR*. La línea 22 indica hacia donde se extiende la información de ruteo, indicando que se acepte el AS anunciado por el vecino, aunque este AS no esté configurado en la declaración de vecino de BGP.

Mencionamos que durante la generación de los archivos de configuración BGP para los nodos, se busca que estos sean automatizados. En este sentido, las líneas 25 a 28 le indican al nodo cuales son sus vecinos, los cuales son descubiertos simplemente indicando la interfaz por la cual están conectados. Esto facilita el automatismo en la generación de archivos de configuración.

3. Cambios y agregados para ejecutar FRR en ns-3

En esta sección detallamos el proceso realizado al portar FRR a ns-3, utilizando el módulo DCE. Esto involucró entre otras cosas estudiar e implementar los cambios y extensiones necesarias para que el código de FRR puede ser utilizado en simulaciones de ns-3. En este sentido identificamos seis etapas:

- Investigación de las herramientas y su funcionamiento.
- Preparación de entorno. En particular, la instalación de DCE fue compleja, dificultándose más por la poca documentación disponible.
- Agregados a DCE para poder ejecutar el código de FRR. Esto es, re-implementar algunas funciones de la biblioteca de C (glibc) que son usadas por DCE. También se corrigieron algunos *bugs* encontrados en el código existente.
- Modificaciones al código de FRR, que se realizaron a fin de solucionar de manera simple problemas que resultó difícil encontrarle otra solución.
- Implementación de una clase FrrHelper de forma que facilite la escritura de scripts que utilicen el porte.
- Realización de pruebas a fin de evaluar y validar el porte, lo que veremos en subsección 4.1.

3.1. Funcionamiento de DCE y experiencias previas

Simulador de redes ns-3

Como es mencionado por sus desarrolladores [3], ns-3 es un simulador de redes de eventos discretos para sistemas de Internet, dirigido principalmente para investigación y uso educativo. ns-3 es software libre con licencia GNU GPLv2, estando disponible para su uso.

El simulador ns-3 es de eventos discretos. Conceptualmente, el simulador hace un seguimiento de un número de eventos que están planificados para ejecutarse en un tiempo simulado. El trabajo del simulador es ejecutar los eventos secuencialmente en orden de tiempo. Una vez se completó un evento, el simulador se mueve al siguiente (o termina si no hay más eventos en la cola). Si, por ejemplo, un evento planificado para el tiempo simulado "100 segundos" se ejecuta y el siguiente evento no está planificado hasta "200 segundos", entonces el simulador salta inmediatamente desde los 100 segundos a los 200 segundos (de tiempo simulado) para ejecutar el siguiente evento. Esto es lo que significa un simulador de eventos discretos.

En el simulador de redes ns-3 tanto el núcleo (*core*) como los modelos están implementados en C++. Está construido como una biblioteca que puede ser utilizada tanto de forma estática como dinámica por un programa principal de C++, que define la

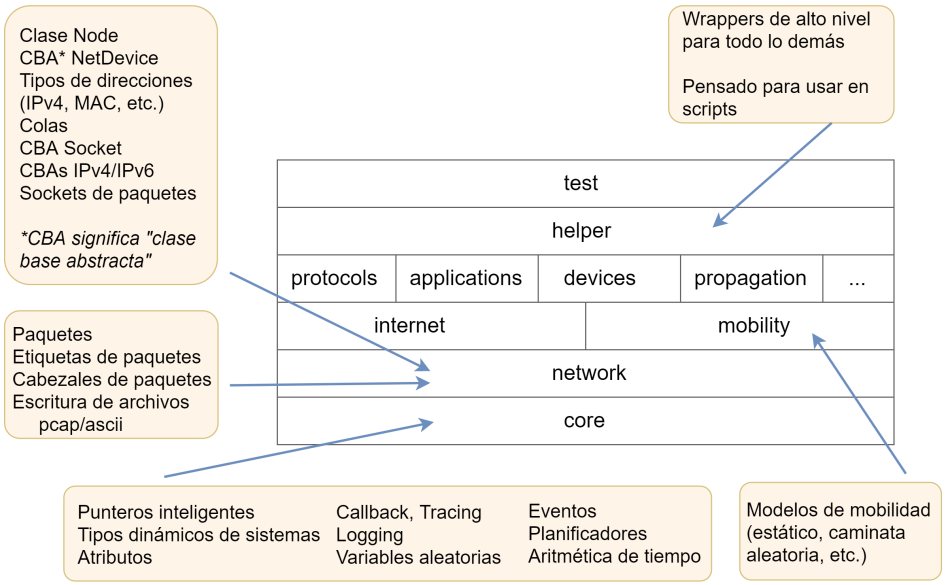

Figura 6: Organización de software en `ns-3` [*Elaboración propia en base a imagen de* [3]].

topología e inicia la simulación [13]. En la Figura 6 se muestra la organización modular del código del simulador.

Típicamente para ejecutar una simulación en `ns-3` creamos un programa de `C++`[2] (script en el lenguaje de `ns-3`) que define la topología y configuración de la simulación. Este programa incluye al final una llamada a la función `Run()` de la clase `Simulator` que dará comienzo a la simulación. Los conceptos básicos que define `ns-3` para usar en estos scripts son:

- **Nodo** (*Node*): En `ns-3` los nodos son instancias de la clase `Node`. Se puede pensar en un nodo como en una computadora, a la cual uno puede agregar dispositivos de red (tarjetas de red) y otros componentes incluyendo protocolos y aplicaciones. Un nodo tendrá entonces una lista de aplicaciones, una lista de dispositivos de red, una lista de protocoles y un ID único.
- **Aplicación** (*Application*): Una aplicación en `ns-3` (clase `Application`) representa la misma idea que tenemos de aplicaciones para una computadora. Una aplicación siempre está asociada a un nodo y representa un programa de software que ejecuta sobre él. Para acceder a lo que serían los recursos del sistema una aplicación hará uso de la API de `ns-3` para poder acceder a ellos (sockets, reloj del sistema, etc.).

[2]Mediante `python bindings` también se puede usar un script de `python`. Los `python bindings` para `ns-3` usan una herramienta llamada `PyBindGen` (*https://github.com/gjcarneiro/pybindgen*) para crear módulos de `python` a partir de las bibliotecas de `C++` construidas con `waf`. Los `python bindings` que usa `PyBindGen` son mantenidos en un directorio `bindings` en cada módulo y deben ser mantenidos para coincidir con la API del módulo `C++` de `ns-3`.

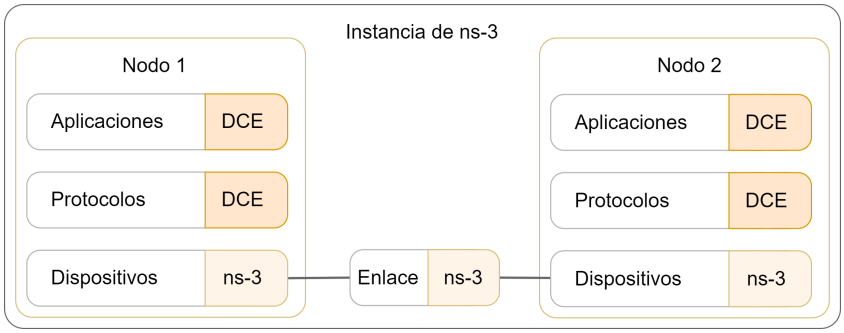

Figura 7: DCE es usado para ejecutar aplicaciones de Linux sin modificar su código, así como también para usar el stack de protocolos de redes de Linux. Los dispositivos de red (y el canal) son simulados solo con ns-3 mientras que aplicaciones y protocolos pueden usar DCE.

- **Dispositivo de red** (*NetDevice*): Los dispositivos de red en ns-3 (clase NetDevice) son a un nodo lo que una tarjeta de red (junto con su driver) es a una computadora. Así en ns-3 para poder conectar un nodo a través de un canal es necesario asociarle un dispositivo de red. Dependiendo de a que canal se realizará la conexión y el tipo de esta, existen diferentes subclases de dispositivo de red para usar.
- **Canal** (*Channel*): Un canal en ns-3 es una instancia de la clase Channel. Es un camino lógico sobre el cual fluye información. El camino puede ser simplemente una conexión cableada o algo más complejo como una conexión por Wifi. Para representar las diferentes posibilidades de canales existen en ns-3 diferentes subclases del mismo para simular distintas realidades. Estas subclases deben coincidir luego con las subclases de los dispositivos de red al cual se conectan.

Módulo DCE

Direct Code Execution (DCE) es un framework para ns-3 que provee facilidades para ejecutar, dentro de ns-3, implementaciones existentes de protocolos de red o aplicaciones sin modificaciones de código. Por ejemplo, en vez de usar la implementación de ns-3 de una aplicación tipo ping, se puede usar el ping real.

De esta manera, en una simulación de ns-3 que usa DCE, la topología de la red así como configuraciones del canal serán hechas en ns-3, mientras que las aplicaciones que ejecutan en los nodos pueden usar DCE y ejecutar cualquier aplicación de Linux. En la Figura 7 se muestra un esquema de esto.

Adicionalmente con DCE se pueden usar las implementaciones reales de protocolos de red, por ejemplo usar el TCP implementado en Linux en vez del TCP de ns-3. Esto proporciona dos modos de ejecutar DCE: *basic mode* y *advanced mode*. En *basic mode* se utiliza el stack de redes de ns-3, mientras que en *advanced mode* se utiliza el stack de redes de Linux. Esto último se realiza utilizando el kernel de Linux como una biblioteca.

En su origen DCE tenía los siguientes cinco requerimientos [14]:

- **Realismo en la experimentación:** Esto es, que el software utilizado sea el mismo que se utiliza en el mundo real, que el comportamiento temporal sea igual al real y que el tráfico de red se comporte de forma realista. Estas últimas dos características se consiguen gracias a `ns-3` mientras que la primera se consigue por `DCE`.
- **Flexibilidad en la topología:** Posibilidad de agregar parámetros a los experimentos de forma sencilla con el fin de simular cualquier tipo de topología.
- **Reproducibilidad:** Debe ser posible replicar un experimento. Esto se consigue gracias a que una ejecución de `ns-3` es completamente determinista.
- **Escalabilidad:** El rango de posibles escenarios de experimentación no debe estar limitado por los recursos de la máquina en donde se realizan los experimentos. Gracias a la dilatación del tiempo esto es fácilmente conseguible. Un minuto de tiempo simulado puede demorar más o menos en tiempo real pero siempre es posible ejecutar la simulación y obtener resultados correctos.
- **Facilidad de depuración:** Debe ser sencillo identificar los problemas que puedan ocurrir en el sistema y depurarlos fácilmente. Particularmente problemas de red que se den en sistemas distribuidos. Dado que `DCE` usa un esquema donde toda la simulación se ejecuta en un solo proceso, se facilita la depuración pudiendo usar *debugguers* tradicionales como `gdb`.

El diseño de `DCE` toma su idea de *library operating system* (*LibOS* [15]). `DCE` está estructurado alrededor de tres componentes: *core*, *kernel* y POSIX, como se muestra en la Figura 8. Primero en el nivel inferior se encuentra el módulo *core* que maneja la virtualización de la memoria: stack, heap y variables globales. Por encima se encuentra la capa *kernel* que toma ventaja de esos servicios para proveer un entorno de ejecución al stack de red de Linux dentro del simulador. Por último, la capa POSIX se construye sobre las capas *core* y *kernel* para re-implementar la API standard de sockets para su uso por aplicaciones simuladas.

`DCE` ejecuta cada proceso simulado en el mismo proceso host. Este modelo hace posible sincronizar y planificar cada proceso simulado sin tener que usar mecanismos de sincronización inter-procesos. Lo que es más, permite al usuario rastrear el comportamiento del experimento por diferentes procesos sin tener que usar un *debugger* distribuido que tienden a ser más complejos. Los hilos en cada proceso simulado son gestionados por un manejador de tareas, implementado en `DCE`, sincronizado con el host simulado y aislado de los otros hosts simulados.

Como el *loader* del sistema host tiene por objetivo asegurar que cada proceso no contenga más de una instancia de cada variable global, `DCE` debe proveer una implementación propia del *loader*. Por esto la virtualización de la memoria global es compleja. `DCE` provee un mecanismo de *loading* específico para instanciar cada variable global, una vez por instancia simulada.

La implementación de POSIX en `DCE` remplaza el uso de la biblioteca tradicional `glibc`. De esta forma cuando una aplicación ejecutando sobre `DCE` hace un llamado a `glibc`, `DCE` intercepta el llamado y ejecuta la función re-implementada. La mayoría de

esas funciones son simplemente un pasamano a la función correspondiente en la biblioteca de C del host. Sin embargo, llamadas que involucran recursos del sistema deben ser re-implementadas. Estas incluyen llamadas que involucran recursos de red, el reloj del sistema, manejo de memoria, etc..

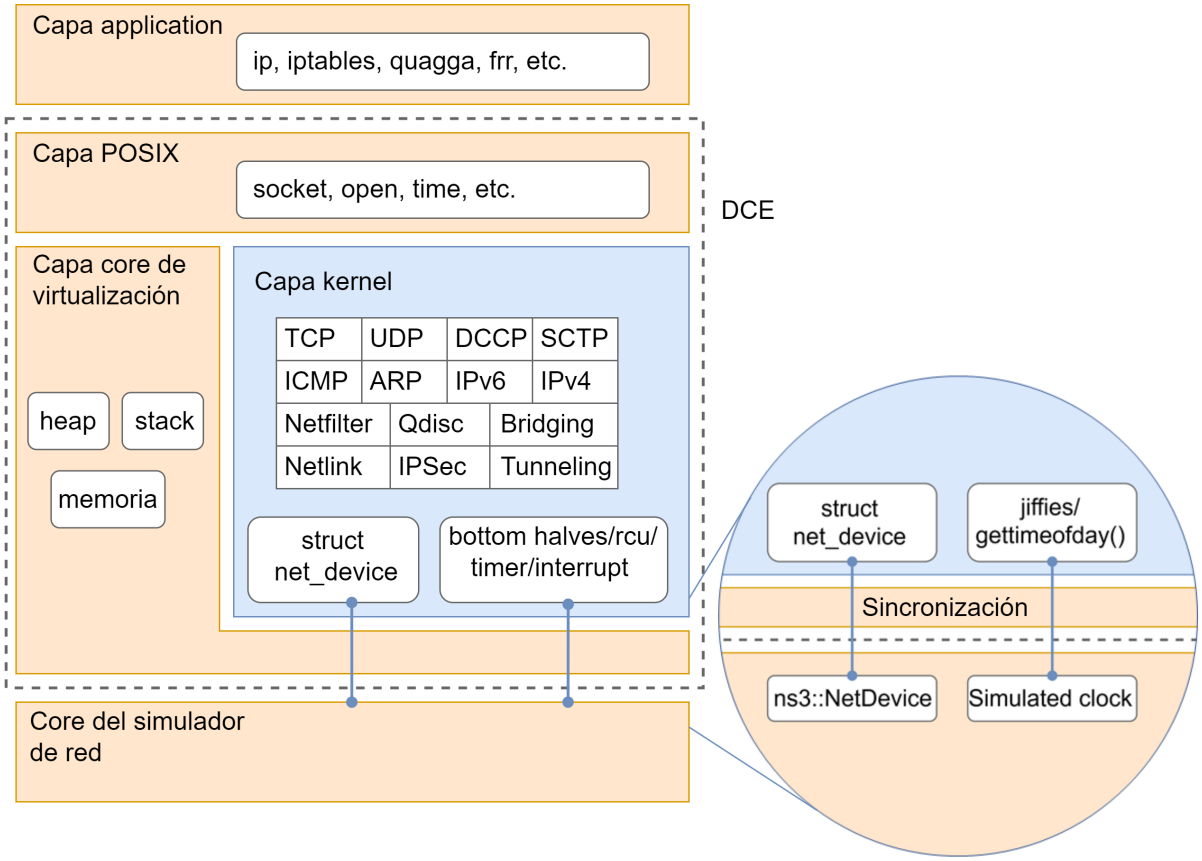

Figura 8: Arquitectura de DCE. En la capa application es donde se ejecutarán nuestros programas usando DCE para conectarse con el core del simulador de red (ns-3). DCE a su vez cuenta con tres componentes: una capa POSIX que re-implementa la API standard de Linux para su uso por aplicaciones, una capa kernel que ejecuta el stack de protocolos de red de Linux y una capa core que maneja la virtualización de la memoria entre otras cosas. Para el caso de Advanced Mode, DCE utiliza la implementación del kernel de Linux de protocolos de capa 3 y 4. Las capas 1 y 2 son simuladas con ns-3 y DCE se encarga de sincronizar, logrando que el kernel de Linux vea los dispositivos de red de ns-3 como si fueran dispositivos reales. Análogamente para otros recursos del sistema como el reloj, entre otros. [*Elaboración propia en base a imagen de* [16]]

Uno de los aspectos más complejos de esta implementación de POSIX es la función `fork()`. Tradicionalmente, implementaciones de POSIX con único espacio de direccionamiento solo proveen la implementación de la función `vfork()` ya que no es fácil hacer que dos procesos, compartiendo el mismo espacio de direccionamiento, vean diferentes valores para el mismo lugar de memoria. En contraste, DCE soporta ambas funciones para facilitar una mayor gama de aplicaciones. Esta funcionalidad se implementa regis-

Protocolo	Basic Mode (ns-3 stack)	Advanced Mode (ns-3 linux)	Observaciones
Rtadvd (zebra)	No muy bien	OK	
RIPv1/v2 (ripd)	No muy bien	OK	bind() falla
RIPng (ripngd)	No muy bien	OK	send() falla
OSPFv2 (ospfd)	OK	OK	
OSPFv3 (ospf6d)	No muy bien	OK	send() falla
BGP (bgpd)	OK	OK	
BGP+ (bgpd)	No muy bien	OK	

Cuadro 1: Funcionalidades soportadas por el porte de Quagga en DCE. [*Cuadro extraído de* [17]]

trando qué lugares de memoria son compartidos por qué procesos y luego guardando y restaurando estos lugares en cambios de contexto.

Porte de Quagga

Como experiencia previa al porte de FRR podemos citar un porte de Quagga[3] a DCE realizado en 2008 [17]. El módulo de Quagga en DCE permite usar la implementación de Quagga de protocolos de enrutamiento (RIPv1, RIPv2, RIPng, OSPFv2, OSPFv3, BGP, BGP+, RAadv) como modelos en la simulación de red. Esto permite ejecutar en la simulación código que ya está en uso en el mundo real, lo que hace a la simulación más fiable.

A la fecha ya no se mantiene el proyecto Quagga DCE activamente, siendo en 2012 la última vez que tuvo soporte. Pese a ello el proyecto sigue funcional y se puede ejecutar con DCE sin mayores inconvenientes. El soporte de Quagga en DCE no es completo. En el Cuadro 1 se muestra la disponibilidad de cada protocolo.

Para facilitar el uso de Quagga en las simulaciones, el proyecto proporciona una clase QuaggaHelper. Esta clase brinda métodos que pueden ser usados desde los scritps de la simulación para instalar un protocolo en un nodo y realizar la configuración del mismo. Durante el porte de FRR nos inspiramos fuertemente en esta clase para desarrollar un FrrHelper a fin de proporcionar facilidades similares. Adicionalmente el hecho de que no todas las funcionalidades funcionen con el stack de ns-3 (Basic Mode) nos motivó a concentrarnos en el stack de Linux (Advanced Mode) para el porte de FRR.

[3] FRR es un *fork* de Quagga realizado por la frustración de algunos desarrolladores con el ritmo de desarrollo de este. El funcionamiento general es por tanto similar y FRR ha remplazado a Quagga en popularidad.

3.2. Preparación de entorno

En esta sección explicamos lo necesario para la instalación de ns-3 DCE utilizando bake y la instalación de FRR, junto con sus dependencias. bake [18] es una herramienta escrita en python para hacer el build de software distribuido. Fue desarrollada para el proyecto ns-3, sin embargo su uso es general y podría usarse para otros proyectos de código abierto que requieran la integración de múltiples proyectos.

Destacamos que toda la instalación de ns-3 DCE, junto con sus dependencias y la descarga del código con los agregados y modificaciones introducidas por nosotros para hacer posible el porte, quedó automatizado mediante unos scripts en bash. Los mismos pueden consultarse en el Anexo 8.1. También se incluyen scripts para la instalación de FRR y sus dependencias. Este conjunto de scripts puede utilizarse como referencia para futuras instalaciones.

Instalación de ns-3 DCE

Según la documentación de DCE [16], este solo ha sido testeado en Ubuntu 16.04, debido a la versión de la biblioteca de libc (gcc-5.4, libc-2.23), por lo que utilizamos esta distribución. Existe una propuesta para utilizar DCE sobre Ubuntu 20.04, pero no se encontraba funcional al momento del inicio del proyecto.

Por lo tanto, la instalación inicial fue realizada sobre una máquina virtual con dicho sistema operativo, utilizando la herramienta bake mantenida por los desarrolladores de ns-3. Adicionalmente, configuramos DCE para su uso en Advanced Mode[4], esto es, usando el kernel de Linux.

Como mencionamos, la instalación de ns-3 DCE se realizó utilizando la herramienta bake. Para que el paso build de bake se realizara con éxito fue necesaria la instalación de muchas dependencias. Para ello se instalaron cerca de 60 paquetes a través del repositorio estándar de Ubuntu con apt[5], incluyendo los requeridos para la posterior instalación de FRR. Destacamos que algunas dependencias fueron indicadas por bake, mientras que otras resultaron de numerosos intentos fallidos de instalar ns-3 DCE, por lo que se instalaron a prueba y error.

Dado que Ubuntu 16.04 tiene python2 como versión por defecto, pero para casi todo lo demás se requiere python3, vimos conveniente que el entorno configurado utilice ambas versiones. Además fue necesario actualizar manualmente la versión de python3 a la 3.8[6] (por requerimiento de python bindings) dado que la que se encuentra en el repositorio no es lo suficientemente nueva. Utilizar dos versiones de python generó algunos problemas durante la instalación.

Además, antes de realizar el paso build de bake, nos posicionamos en un *commit*

[4]Línea 7 del script **download-dce** del Anexo 8.1
[5]Línea 7 del script **install-dependencies** del Anexo 8.1
[6]Script **update-python3** del Anexo 8.1

particular del código de ns-3 a fin de trabajar sobre código estable[7]. También realizamos un *fork* del proyecto DCE y lo integramos al código de ns-3 descargado[8]. Finalmente aplicamos un parche al componente elf-loader de DCE, debido a incompatibilidades con python[9].

Instalación de FRR

Al igual que para el caso de ns-3 DCE, la instalación de FRR requirió algunas dependencias. Algunas fueron instaladas utilizando el repositorio estándar de Ubuntu, sin embargo otras tuvieron que ser compiladas a partir de código fuente debido a usar versiones distintas a las que tiene el repositorio[10].

Para que un programa pueda ejecutar en DCE, este necesita reubicar el binario ejecutable en memoria. A su vez, estos archivos ejecutables necesitan construirse con opciones específicas para las etapas de compilación y *linking*. Según el manual de ns-3 DCE [16], para poder hacer esto se deben seguir los siguientes pasos:

- Compilar los objetos usando la *flag* de gcc -fPIC. Esto es para que DCE vea el programa compilado como una biblioteca compartida.
- (Opcional) Algunas aplicaciones requieren ser compiladas con la opción -U_FORTIFY_SOURCE para que le aplicación no use símbolos alternativos del estilo __chk (como memcpy__chk).
- Linkear el ejecutable usando las *flags* de gcc -pie y -rdynamic. Esto es para que DCE cargue la aplicación como una biblioteca dinámica.

Entre las dependencias requeridas por FRR se encuentra libyang. Para compilar esta desde código fuente, además de las *flags* requeridas por DCE, se utilizaron estas otras[11]:

- Para que no se utilicen funciones __str* se utiliza -O0.
- A fin de contar con los símbolos de *debugger* -ggdb.

Finalmente, para compilar FRR desde código fuente, se utilizaron las *flags* requeridas por DCE y las ya agregadas por libyang. Además de esto se compiló FRR con las siguientes opciones[12] [19]:

[7] Línea 14 del script **download-dce** del Anexo 8.1
[8] Línea 19 del script **download-dce** del Anexo 8.1
[9] Línea 24 del script **download-dce** del Anexo 8.1
[10] Scripts **swig-install**, **debhelper-install** y **libyang-install** del Anexo 8.1
[11] Script **libyang-install** del Anexo 8.1
[12] Script **install-frr** del Anexo 8.1

- Definir que se ejecute con usuario `root`. Cuando se compila `FRR` es necesario definir un usuario. Dado que sabemos que existe el usuario `root`, es cómodo usarlo y como `FRR` se ejecutará dentro de una simulación de `ns-3 DCE` no nos preocupa la parte de seguridad.
- Deshabilitar el uso de *capabilities* para que no use `libcap`. Como ya mencionamos en el porte no nos preocupa la parte de seguridad y la biblioteca `libcap` utiliza muchas llamadas al sistema que sería dificultoso re-implementar en `DCE`.
- Configurar `libfrr` para que se use como biblioteca estática. Que los binarios de `FRR` generados tengan la menor cantidad posible de dependencias de bibliotecas dinámicas, simplifica el porte hacia `ns-3 DCE`. Aunque no se puede dejar todas las bibliotecas de forma estática, `FRR` ofrece la posibilidad de compilar lo máximo posible de forma estática [20].

Ajustes al sistema operativo

Con lo anterior es suficiente para ejecutar simulaciones. Sin embargo, existen otras limitantes impuestas por el sistema operativo que afectan el tamaño máximo que las mismas pueden tener. Se pueden ajustar algunos parámetros en el sistema operativo (Ubuntu 16.04) a fin de superar estas limitantes[13].

Por un lado tenemos el límite del sistema operativo a la cantidad de archivos abiertos. Por defecto Linux limita a 1024 la cantidad de archivos que se pueden tener abiertos por proceso, sin embargo `ns-3 DCE` fácilmente puede exceder ese límite para simulaciones grandes. Por este motivo se aumenta la cantidad de archivos abiertos a 65536 como se describe en el manual de `DCE`.

De igual modo, el sistema operativo pone un límite a la cantidad de mappings que pueden ser utilizados por proceso. Este límite es por defecto 65536, pero es configurable. Simulaciones grandes (más de 800 nodos) fácilmente superan este límite cuando se utiliza `FRR`, terminando con error "Unable to protect bottom of stack space, errno=Cannot allocate memory".

Uso de *debuggers*

Es posible utilizar `gdb` para depurar scripts en `ns-3 DCE`, los cuales se ejecutan en un solo proceso. Esto permite poner *breakpoints* tanto en el código fuente de `DCE` como en los binarios importados por este. De este modo `FRR` y `libyang` fueron compilados habilitando el uso de *debuggers*, en particular `gdb`.

Cuando una simulación falla, `ns-3` no brinda información sobre el lugar de ocurrencia en el código y generalmente tampoco información que permita identificar el origen de la falla. De esta manera, el uso de `gdb` significó una herramienta de gran utilidad, la

[13] Script **limit-open-files** del Anexo 8.1

cual fue utilizada principalmente durante el porte de FRR a ns-3 DCE. Adicionalmente también se utilizó valgrind para ahondar más en errores en el manejo de memoria.

DCE usa para su funcionamiento la señal SIGUSR1. Esto debe ser tenido en cuenta al tener una sesión de *debugging* con gdb, de lo contrario gdb retorna el control al usuario cada vez que se produce esta señal. Para evitar esto, se puede usar el comando "handle SIGUSR1 nostop" al inicio de cada sesión de gdb. Alternativamente se puede iniciar la sesión de *debugging* con:

```
python3 waf --run <SCRIPT> --command-template="gdb -ex
'handle SIGUSR1 nostop noprint' --args %s".
```

3.3. Agregados a DCE

Como mencionamos antes, nos posicionamos en un *commit* estable del proyecto ns-3 DCE del cual hicimos un *fork*. En nuestro ambiente de trabajo, sea por configuración de la máquina virtual u otro motivo, el *commit* presentaba error de compilación el cual corregimos[14].

En el porte de FRR se presentaron múltiples errores por símbolos de función no reconocidos, dado que no estaban declaradas en DCE. DCE clasifica a las funciones de la biblioteca libc utilizando las macro DCE o NATIVE. Las primeras son funciones que son re-implementadas por DCE, mientras que las últimas se pasan a la biblioteca propia del sistema operativo.

Durante el porte de FRR nos encontramos con que este utiliza las siguientes funciones que no estaban declaradas en la capa POSIX de DCE. Por este motivo, y teniendo en cuenta su funcionamiento, decidimos implementarlas utilizando entonces la macro DCE:

- malloc_usable_size: Devuelve el tamaño de memoria usable que se solicitó previamente mediante malloc. La función malloc fue re-implementada por DCE y al hacerlo almacena el valor de cuanta memoria fue pedida originalmente. Este es el valor de retorno de la función re-implementada malloc_usable_size.
- localtime_r: Devuelve el tiempo en zona horaria UTC. Para hacerlo utiliza la función localtime re-implementada por DCE, cambiando la zona horaria.
- ppoll: Espera que uno o un conjunto de *file descriptors* estén listos para realizar operaciones de entrada/salida. Fue implementada copiando la implementación propia de glibc[15], pero utilizando las llamadas de DCE.

[14] Presentaba error en la línea "typedef PyAsyncMethods* cmpfunc;", el cual corregimos cambiando el tipo de la variable.
[15] *https://linux.die.net/man/2/ppoll*

- `openat`: Abre un archivo en un directorio utilizando una ruta relativa. Se busca un archivo con el *file descriptor* brindado en la ruta relativa, con la finalidad de llamar a la función `open` re-implementada por `DCE`.
- `pthread_setname_np`: Se utiliza para asignar un nombre único a un hilo, lo cuál puede ser útil para depurar aplicaciones multihilo. Se dejó su implementación vacía por no representar un requerimiento indispensable para el funcionamiento de `FRR`.
- `pthread_testcancel`: Crea un punto de cancelación dentro del hilo que llama a la función `pthread_testcancel`. Asumiendo que la cancelabilidad se encuentra deshabilitada, y en tal caso la función no tiene efecto, se deja su implementación vacía.
- `posix_fallocate`: Asegura que el espacio en disco es asignado para el archivo al que se hace referencia. Después de una llamada exitosa a `posix_fallocate`, se garantiza que las escrituras en bytes en el rango especificado no fallarán debido a falta de espacio en disco. Se deja su implementación vacía de forma que siempre se asume que no habrá fallos por falta de espacio en disco.

Por el contrario, las siguientes funciones se dejaron como `NATIVE` ya que su uso por `FRR` no involucra recursos del sistema:

- `fchownat`: Cambia el propietario de un archivo relativo, a un *file descriptor* de directorio.
- `getgrouplist`: Obtiene la lista de grupos a los que pertenece el usuario.
- `syscall`: Invoca una llamada de sistema cuya interfaz en lenguaje *assembler* tiene el número especificado.

Además de estas funciones agregadas, detectamos dos *bugs* en la gestión de memoria en funciones ya implementadas bajo el macro `DCE`. Uno de ellos fue en la función `closedir` en el archivo `dce-dirent.cc`. En la misma se llama a `internalClosedir`, donde hay una asignación de memoria de una región que ha sido liberada previamente. Esto causa un error de "corrupted double linked list", producto de `SEGFAULT`.

También se corrigió un *bug* en la implementación de `DCE` de `vasprintf`. Esta función llama al `vasprintf` del sistema, usando el valor de retorno `ret` para pedir memoria y copiar el *string* que devuelve `vasprintf`. El error se debía a que `ret` es la cantidad de bytes del *string* pero sin incluir el carácter de terminación `\0`. Por esto, al pedir memoria y copiar el *string* original es necesario utilizar "`ret + 1`" en lugar de solo `ret`.

Se realizaron dos *Pull Requests* en el proyecto `ns-3 DCE` por la corrección de estos *bugs*. Además se realizó otro *Pull Request* con las funciones necesarias ejecutar el código de `FRR`. Al momento de escribir este informe, los *Pull Requests* están pendientes de revisión.

3.4. Modificaciones a FRR

Además de los agregados a DCE que mencionamos en la sección anterior, para poder ejecutar FRR en DCE se realizaron cambios a este. Estos cambios se dan por motivos de practicidad, por la dificultad de adaptar DCE para ejecutar FRR en su forma original. Los cambios tuvieron dos formas: cambios en la forma de compilar y cambios en el código fuente.

Para empezar se cambió la flag de optimización en el Makefile de -O2 a -O0 (como ya se mencionó en la sección 3.2). Las optimizaciones del compilador muchas veces usan símbolos de funciones que DCE no implementa (como es __strndup). DCE tiene una macro especial DCE_WITH_ALIAS para estos símbolos de función con dos variantes func y __func. Sin embargo, no pudimos utilizar correctamente esta macro por lo que preferimos evitar este tipo de símbolo de función.

Otros cambios al modo de compilar se realizaron mediante modificaciones al archivo configure.ac. Este es un archivo de entrada para la herramienta autoconf que define algunos valores que luego son usados por macros en el código de FRR. Por ejemplo fijamos una condición frr_cv_pthread_condattr_setclock como false[16] para que FRR no use el símbolo pthread_condattr_setclock que no está implementado en DCE.

También definimos la condición needsync=false en configure.ac[17] para que FRR no utilice futex. Esta es una llamada al sistema operativo para sincronización entre hilos que es más eficiente que mutex. Sin embargo, como DCE no tiene al momento soporte para futex definimos la condición mencionada para que use mutex simplemente.

En el código de FRR hicimos cambios en el archivo lib/zlog.c para evitar el *log buffering*. Para mejorar la performance en el *logging*, FRR usa *buffering* por hilo. Esto es que cada hilo tiene su propio *buffer* para el *logging* y el *log* total es recompuesto luego (*logs* prioritarios igualmente van al *log* total). Para hacer esto FRR utiliza archivos temporales que mapea a memoria con mmap. Estas llamadas sin embargo terminaban el programa con señal SIGBUS al ejecutarse sobre DCE. Como el *log buffering* no es relevante para nuestro caso de uso optamos por simplemente no realizarlo modificando ligeramente el código de lib/zlog.c[18].

Otro cambio realizado en el código fue comentar usos de pthread_cleanup_push y pthread_cleanup_pop en el archivo bgpd/bgp_keepalives.c[19]. Estas funciones son en realidad macros de C donde se termina llamando al símbolo de función __pthread_register_cancel. No pudimos hacer que DCE reconociera estos símbolos. De todas formas como las llamadas pthread_cleanup_push y pthread_cleanup_pop son funciones de limpieza al terminar un hilo, en nuestro caso de uso no es problemático si no se llaman, ya que bgpd solo se ejecuta brevemente en la simulación.

Con estos cambios logramos ejecutar los demonios de FRR zebra y bgpd sobre DCE

[16]Línea 14 del script **install-frr** del Anexo 8.1
[17]Línea 12 del script **install-frr** del Anexo 8.1
[18]Líneas 51 y 52 del script **install-frr** del Anexo 8.1
[19]Línea 49 del script **install-frr** del Anexo 8.1

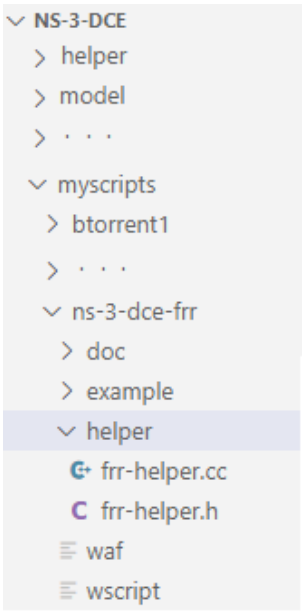

Figura 9: Estructura de archivos de `ns-3 DCE`, en particular el módulo *ns-3-dce-frr*. `FrrHelper` es una clase implementada para facilitar toda la configuración de los demonios de `FRR` portados: zebra, BGP y OSPF. Se encuentra en la carpeta `helper` sombreada en la imagen. En la carpeta `example` se encuentran ejemplos de simulaciones desarrollados usando el porte de `FRR`.

en un nodo de `ns-3`. Sin embargo, tuvimos problemas al intentar ejecutarlo en varios nodos simultáneamente. Este problema se debe a que el *loader* propio de `DCE` invoca solo una vez a la función `pre_init`. Quedan entonces cosas sin inicializar y la ejecución de `FRR` en el nodo siguiente falla. Para solucionar estos problemas optamos por una solución sencilla, de simplemente crear enlaces simbólicos a los ejecutables tantas veces como nodos haya. De esta manera, cada nodo usa un enlace simbólico distinto con nombre `bgpd<nodeId>` (por ejemplo) de forma que el *loader* ve cada enlace como un ejecutable distinto. Con esto nos evitamos los problemas con el *loader* de `DCE`.

3.5. Implementación de `FrrHelper`

Para ayudar en la creación de simulaciones que utilicen el porte de `FRR`, creamos un módulo *ns-3-dce-frr* dentro del proyecto `ns-3 DCE`. Este está basado en el módulo existente del porte de `Quagga` *ns-3-dce-quagga*.

El módulo *ns-3-dce-frr* incluye, entre otras cosas, ejemplos y el `FrrHelper`, como se muestra en la Figura 9. Este último contribuye a la configuración del entorno necesario para el despliegue de simulaciones. Como cualquier simulación de `ns-3`, para ejecutar un script se utiliza la herramienta `waf` que compila el código y ejecuta la función `main` del script. En nuestro módulo entonces tenemos un archivo de configuración *wscript* que es leído por `waf`. En este archivo, están definidas las dependencias del módulo, así como las declaraciones de los ejemplos. Al implementar un nuevo ejemplo es necesario agregarlo en el *wscript*.

Por un lado el `FrrHelper` implementa un método `Install`. Este puede ser invocado de tres maneras: con un conjunto de nodos, un nodo solo, o dado un identificador de nodo. Este método instala algunos de los demonios de `FRR` donde se le indique, quedando zebra instalado implícitamente. Además, se incluyen métodos para la configuración de los demonios portados: zebra, BGP y OSPF. Este último se incluye a fin de poder evaluar el funcionamiento del porte con otro protocolo de la familia `FRR`.

El `Install` es el método principal y más importante en `FrrHelper`, este se debe ejecutar siempre en los nodos que vayan a usar `FRR`. Crea directorios, archivos de configuración y carga los programas a ejecutar por los nodos. Todos los otros métodos del `FrrHelper` solo definen los parámetros que serán utilizados por el `Install`.

El `Install` crea un directorio `files-<idNodo>`, el que posteriormente es utilizado por `DCE` como sistema de archivos del nodo. Dentro de ese directorio se crean otros que son necesarios para el funcionamiento de `FRR` (por ejemplo, `/var/tmp`). De la misma manera se crean enlaces simbólicos apuntando hacia los módulos de la biblioteca `libyang`. Esto es para que durante la simulación puedan ser encontrados. Adicionalmente, dado que por defecto el `Install` instala zebra, se crea un archivo `zebra.conf` utilizado por zebra para la configuración.

Además del método `Install` se proveen implementaciones para la configuración de los demonios portados. Para el caso de zebra, estos métodos son:

- `void EnableZebraDebug(NodeContainer nodes)`: Habilita que zebra se ejecute en modo debug, donde se imprime más información en los *logs*. Esto se lleva a cabo escribiendo sobre el archivo `zebra.conf`.
- `void UseManualZebraConfig(NodeContainer nodes)`: Cuando se invoca este método, no se crea el archivo `zebra.conf` porque se asume que el usuario proveerá uno.

Para BGP los métodos son:

- `void EnableBgp(NodeContainer nodes)`: Primeramente se hacen las configuraciones necesarias de modo que al ejecutarse el `Install` con las configuraciones para zebra, también se instale BGP. Además se crea el archivo de configuración `bgpd.conf`.
- `void UseGivenBgpConfig(Ptr<Node> node, std::string file)`: Este método recibe una ruta por parte del usuario, que contiene un archivo de configuración, el que se utilizará posteriormente en el `Install`.
- `void FatTreeBgpConfig(Ptr<Node> node, int asn, int num_tor, int num_fabric, std::string network)`: Colabora en la creación del archivo `bgpd.conf`, incluyendo los parámetros recibidos para el despliegue del fat tree.
- `void BgpNodeFailure(Ptr<Node> node, Time at)`: Este método provoca que el demonio bgpd se detenga en un tiempo dado sobre el nodo provisto, simulando una falla sobre este.

- `void BgpDelayedStart(Ptr<Node> node, Time at)`: Este método retrasa el comienzo del demonio bgpd al tiempo `at`. Esto es útil para simular un caso donde un nodo tuvo una falla y posteriormente una recuperación.
- `void RunBgpWithoutZebra(NodeContainer nodes)`: Permite que se ejecute BGP sin zebra. Esto lo hace pasando `--no_zebra` como parámetro al ejecutar `bgpd`, y algunos ajustes al archivo `bgpd.conf`.

De forma análoga, se definen métodos para facilitar la ejecución de OSPF en los nodos:

- `void EnableOspf(NodeContainer nodes, const char *network)`: Añade las configuraciones necesarias al arhivo `ospfd.conf` para la ejecución del demonio OSPF.
- `void SetOspfRouterId(Ptr<Node> node, const char *routerid)`: Especifica el parámetro router-id necesario en OSPF.
- `void EnableOspfDebug(NodeContainer nodes)`: Habilita que OSPF se ejecute en modo debug, donde se imprime más información en los *logs*.

Además, se implementó un `frr-utils` con el fin de propiciar funciones útiles tanto para el `FrrHelper` como para las simulaciones:

- `void RunIp(Ptr<Node> node, Time at, std::string str)`: Ejecuta el comando `ip`, acompañado de los atributos brindados en `str` en el nodo `node` en el tiempo `at`. Esto es útil por ejemplo para: levantar/bajar una intefaz, asignar una dirección IP, listar la tabla de ruteo, etc.
- `void AddAddress(Ptr<Node> node, Time at, ...)`: Es un atajo para la función `RunIp(...)` en el caso donde se agrega una IP a una interfaz.
- `void AddAdressesList(Ptr<Node> n, std::vector<std::string> adr, ...)`: Es un atajo para levantar varias interfaces con sus IPs. Opcionalmente esta función ejecuta el comando "`ip -6 address flush dev sim<i>`" para que no se use IPv6 en la interfaz.
- `void RunPing(Ptr<Node> node, Time at, std::string str)`: Ejecuta el comando `ping`, acompañado de los atributos definidos en `str`, en el nodo `node` en el tiempo `at`.

4. Implementación de casos de prueba y evaluación de resultados

4.1. Evaluación funcional del porte de FRR a ns-3 DCE

Naturalmente el proceso de implementación de cambios y agregados a ns-3 DCE para permitir la ejecución de FRR, se da acompañado de verificaciones. En este sentido llevamos a cabo las mismas diseñando e implementando varios escenarios de simulación, donde seguimos un enfoque iterativo e incremental en el grado de complejidad de las mismas.

La arquitectura de FRR tiene como parte central al demonio zebra. Este administra el enrutamiento IP actualizando las tablas de ruteo del kernel del sistema operativo. Asimismo permite descubrir interfaces y redistribuir las rutas entre los diferentes protocolos de ruteo que ejecutan en el host [2].

De esta forma, para poder verificar el porte es necesario ejecutar zebra y BGP conjuntamente. Esto permite que las rutas aprendidas por BGP sean comunicadas a zebra para que actualice las tables de ruteo. Si bien BGP no necesita de zebra para ejecutarse, prescindir de zebra impide conexión real entre los nodos, dado que no se actualizan dichas tablas. Dado que el resultado de BGP es aprender nuevas rutas, disponer de las tablas de ruteo es fundamental para verificar su funcionamiento.

El método de verificación será entonces la comprobación de que las tablas de ruteo fueron actualizadas. Asimismo se realizarán pruebas de conectividad mediante el comando ping.

Ejecución de zebra y BGP en un nodo

La primera prueba consiste en lograr ejecutar zebra en una simulación compuesta por un solo nodo y que esta finalice sin error. Esto conlleva a la aparición de errores en tiempo de ejecución que deben identificarse y resolverse, como por ejemplo invocaciones a funciones que no están implementadas. La sección 3 detalla entre otras cosas la resolución de estos conflictos.

Luego de conseguir que zebra ejecute sin errores, se amplía la simulación de forma tal que el nodo también ejecute bgpd. Al igual que sucedió con zebra, al ejecutar código nuevo de FRR dentro de ns-3 DCE se producen nuevos errores en tiempo de ejecución que deben identificarse y resolverse. Una vez conseguida la ejecución exitosa de esta simulación, estamos en condiciones de ejecutar casos más complejos.

Ejecución de zebra y BGP en varios nodos conectados

Continuando con la verificación del porte de zebra y bgpd es necesario verificar el funcionamiento del protocolo BGP como tal. Esto es, ejecutar bgpd en varios nodos y comprobar que el comportamiento es el esperado (se actualizan las tablas de ruteo). Para

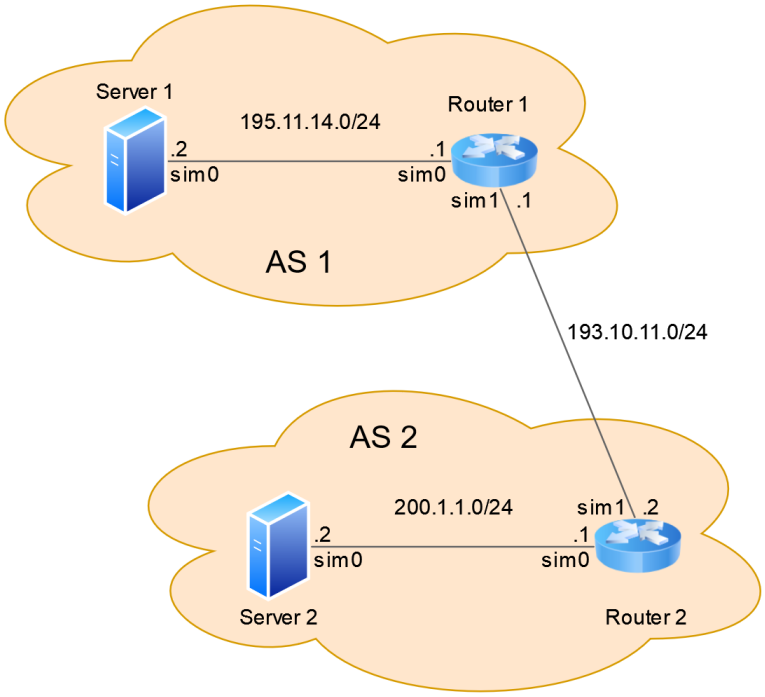

Figura 10: Laboratorio BGP Simple Peering de Kathará. Dos routers mediante BGP intercambian información de las subredes a las que están conectados.

esto implementamos algunos escenarios basados en laboratorios del proyecto Kathará [21]:

- BGP Simple Peering: Implementa una topología sencilla con simplemente dos routers (cada uno con una LAN propia) que intercambian información de sus propias rutas, como se muestra en la Figura 10.
- BGP Prefix Filtering: Este ejemplo es muy similar al anterior con la salvedad de que cada router tiene más de una LAN. El objetivo es configurar BGP de forma que se publiquen algunas rutas y otras no.
- BGP Multi-homed Stub: Es un ejemplo un poco más complejo donde los nodos tienen que publicar las rutas aprendidas desde otros nodos, entre otras cosas.

Con una ejecución correcta de estos escenarios, y habiendo comprobado que las tablas de ruteo se actualizan, podemos concluir que zebra y BGP están ejecutándose correctamente en `ns-3 DCE`. Esto verifica el trabajo realizado previamente en el porte.

Ejecución de otros protocolos

En principio todos los demonios de `FRR` podrían ejecutarse en `ns-3 DCE` ya que todos fueron compilados apropiadamente para ello. Sin embargo, por el limitado alcance

del proyecto, BGP fue el único que verificamos exhaustivamente.

De todas formas, se realizó una breve prueba del demonio OSPF en `ns-3 DCE`. Para esto realizamos la misma prueba que el Laboratorio BGP Simple Peering (Figura 10). Aquí verificamos que cada router aprende las subredes del otro.

Para este caso de OSPF no fue necesario hacer nada nuevo en el porte para que funcionara correctamente. Visto esto esperamos que otros protocolos también funcionen correctamente sin necesidad de hacer más modificaciones al porte. Esto es razonable dado que por la arquitectura de `FRR` la mayor complejidad se reserva al demonio zebra.

4.2. Diseño de pruebas para fat trees

Una vez verificada la correctitud del porte, comenzamos con las simulaciones que lo motivaron originalmente. Estas pruebas son en definitiva el uso de BGP en fat trees, topología que usan los datacenters en la actualidad.

Descripción de los casos de prueba

Se implementaron casos de prueba tomando como guía los descritos en el documento *Routing in data centers: Test features* [22]. Estos fueron diseñados por el proyecto Kathará, como se mencionó en la sección 1.1. Dado que contamos con los resultados obtenidos en esas emulaciones, resultó conveniente re-implementar los casos en `ns-3 DCE` a fin de poder comparar los resultados de ambos.

El objetivo de los casos de pruebas es registrar el número de Protocol Data Units (PDUs) intercambiado entre los nodos. En nuestro caso de estudio (solamente el protocolo BGP) esto equivale a la cantidad de mensajes BGP UPDATE.

Cuando se inicia una simulación y se comienza a ejecutar el protocolo BGP, los nodos comienzan a intercambiar mensajes UPDATE, hasta que se llega a la convergencia y los mensajes UPDATE cesan. La convergencia se da cuando toda la información de la topología ha sido distribuida. Cualquier cambio en la topología, o en la tabla de ruteo de un nodo, genera nuevamente intercambio de mensajes UPDATE hasta que la convergencia es alcanzada nuevamente.

De esta manera, una forma de determinar que se llegó a la convergencia es que ya no hay intercambio de mensajes UPDATE. Por esto es importante que las simulaciones ejecuten un tiempo suficiente para garantizar la convergencia.

De los casos definidos en [22], implementamos los siguientes:

- **Bootstrap**: El objetivo es estudiar el comportamiento estándar del protocolo en la topología cuando se le da inicio, sin ninguna falla.
- **Node Failure**: Este caso de prueba sirve tanto para verificar que BGP converge luego de la falla de un switch, como también para contar el número de PDUs que el protocolo intercambió a tal fin. La falla puede introducirse en cualquier tipo

de switch de la topología, es decir, Leaf, Spine o Tof. La misma se lleva a cabo apagando el demonio BGP en el switch dado.

- **Node Recovery**: En este caso de prueba el objetivo es contar la cantidad de PDUs intercambiados por los switches, luego de que uno de ellos falla y es reemplazado por uno nuevo. Al igual que el caso anterior, este caso puede ejecutarse en un Leaf, Spine o Tof. Este caso lo implementamos levantando la topología sin ejecutar BGP en el nodo en cuestión, esperamos que converja y luego iniciamos BGP. Esto es equivalente a ocasionar una falla en el nodo y luego iniciarlo nuevamente.
- **Link Failure**: Este caso también tiene dos objetivos. Por un lado, verificar la convergencia de BGP después de la falla de un enlace y por otro lado, contar la cantidad de PDUs a tal fin. La prueba puede ejecutarse tanto para el caso del enlace Leaf-Spine como para el enlace Spine-Tof, simplemente bajando una interfaz dada.
- **Link Recovery**: Este caso cuenta el número de PDUs luego de que se reemplaza un enlace que ha fallado. Esto es, se inicia la simulación y se espera que el protocolo converja. Luego se ocasiona la falla del enlace y se espera nuevamente que el protocolo converja. Finalmente se recupera el enlace y se espera la nueva convergencia. El número de PDUs que se toman en cuenta son los intercambiados en esta última fase.

Optamos por el criterio `x_y_z_caso-nivel` para nombrar los casos de prueba, en donde:

- `x` es el parámetro *k_leaf*.
- `y` es el parámetro *k_top*.
- `z` es el parámetro *redundancy*.
- `caso` representa el caso de prueba, pudiendo ser `link-failure`, `link-recovery`, `node-failure`, o `node-recovery`.
- `nivel` depende del `caso`:
  - Si el `caso` es `link-failure` o `link-recovery`, `nivel` puede ser `leaf-spine` o `spine-tof`, referenciando el nivel donde se produjo la falla o la recuperación del enlace.
  - Si el `caso` es `node-failure` o `node-recovery`, `nivel` puede ser `leaf`, `spine` o `tof`, referenciando el nivel donde se produjo la falla o la recuperación del nodo.

En la sección 4.3 describimos la forma de ejecución para estos casos de prueba y post-procesamiento. Asimismo, en la sección 4.4 se muestran y analizan los resultados obtenidos, y adicionalmente descripciones sobre la forma de verificación de los mismas.

Implementación de un generador de fat trees

Como se explicó en la sección 2.3, los fat trees se construyen según parámetros dados y siguiendo una estructura particular. A fin de poder ejecutar múltiples casos de prueba sobre fat trees diferentes, sin necesidad de implementarlos cada vez, se propone la construcción de un generador de fat trees (*vtfgen*, por virtual fat tree generator).

Esto facilita la automatización, creación y reproducibilidad de los casos de prueba. Para la creación de este generador se implementaron las utilidades `vftgen-utils` y `vftgen-classes` que se encargan de construir la topología. Esto es, según los parámetros indicados crean la cantidad adecuada de nodos de `ns-3`, los conectan según el fat tree correspondiente y les asigna IPs adecuadas.

A continuación haremos un recorrido de las diferentes utilidades con una breve descripción de las mismas. En `vftgen-classes` encontramos:

- Un struct `config` que almacena el valor de los parámetros para la construcción de la topología.
- Una clase `NodeVft` que representa un nodo cualquiera en la topología, y las clases `Tof`, `Spine`, `Leaf` y `Server`, las cuales heredan de `NodeVft`. Al momento de creación de estos nodos, se les asigna un identificador y un numéro de Pod o plano según corresponda. Además, cada nodo mantiene una lista con las IPs de sus interaces y una lista de sus vecinos.
- Una clase `Pod` tiene un identificador y listas de Spines, Leaves y servidores que pertenecen al Pod.
- Una `FatTree`, contiene toda la estructura del fat tree con las clases antes mencionadas. Además, provee métodos para facilitar su construcción.

Por otra parte, `vftgen-utils` proporciona funciones de utilidad como:

- `struct Config init_config(int k_leaf, int k_top, ...)`: Para inicializar el struct `config` según parámetros dados.
- `void CreateCollisionsDomains(...)`: Dado un fat tree crea los dominios de colision en `ns-3`. Se utilizaron dominios point-to-point para las conexiones entre switches, y para simplificar las conexiones entre los servidores con una Leaf, se utilizaron dominios csma[20].
- `void SetNetworkStack(FatTree *fat_tree, struct Config config)`: Le instala a cada nodo del fat tree el stack de red de Linux.

[20]Carrier-sense multiple access, equivalente a Ethernet en `ns-3`.

- `void IpConfig(FatTree *fat_tree, struct Config config)`: Asigna IPs a los nodos `ns-3` del fat tree. Estas IPs fueron generadas previamente y se encuentran almacenadas en las instancias de la clase `NodeVft`.
- `void ConfigFrr(...)`: Instala BGP en todos los nodos del fat tree. Para la configuración de BGP, este método le pasa al `FrrHelper` los parámetros del fat tree para que este la genere. Tiene la opción de configurar BGP sin zebra como se describe en la sección 4.5.
- `void RenameFolders(...)`: Durante las simulaciones se crea una carpeta `files-<nodeId>` por cada nodo de `ns-3`. Dicha carpeta actúa como el sistema de archivos del nodo y almacena archivos como pueden ser de configuración o *logs* de los programas ejecutados. Una vez finalizada la simulación, a fin de simplificar el análisis posterior de los resultados, este método renombra `files-<nodeId>` por

  `files-<leaf|spine|tof>_<nPlano|nPod>_<nivel>_<nNodo>.`

  Para el caso de los servidores

  `files-<server>_<nPod>_X_<nLeaf>_<nNodo>.`
- `void RenamePcapFiles(...)`: Análogo a lo anterior, renombramos las capturas de tráfico (archivos .pcap) generados durante la simulación por

  `<leaf|spine|tof>_<nPlano|nPod>_<nivel>_<nNodo>-<interface>.`
- `void GenerateGraphImg(...)`: Genera una imagen de la topología simulada, para facilitar el testeo y la visualización.

4.3. Plan de ejecución de pruebas y post-procesamiento de resultados

Uno de los objetivos del proyecto es poder utilizar el porte obtenido para ejecutar una variedad de casos de prueba. Con esto se espera validar la utilidad del trabajo realizado aplicado a la experimentación del uso de BGP en fat trees.

Para posibilitar esta tarea se escribieron dos scripts en `bash`: `run_examples.sh` y `run_analyze.sh`[21]. Con estos scripts se pueden encolar para ejecutar una gran cantidad de simulaciones así como hacer el posterior análisis de los datos de las mismas.

Ejecución de las pruebas

El script `run_examples.sh` permite definir una lista de ejemplos a ejecutar. Este ejecuta una simulación de `ns-3 DCE` para cada uno de los ejemplos provistos y luego

[21] Estos scripts se encuentran en el reposiorio *https://gitlab.fing.edu.uy/proyecto-2021/run-scripts*

mueve a otra carpeta los archivos necesarios para el post-procesamiento de resultados. El script posibilita la ejecución en el background de las simulaciones de forma de poder dejarlas ejecutando en un servidor para analizar luego los resultados.

Se debe tener cuidado con cuales archivos se guardan para el post-procesamiento de resultados. La simulaciones generan una gran cantidad de archivos incluyendo: capturas de tráfico, archivos de configuración de los nodos, archivos necesarios por `FRR` en cada nodo y *logs* de cada proceso simulado en los nodos. De conservar todos estos archivos para muchas simulaciones fácilmente se puede alcanzar el máximo número de inodos[22] permitido en el sistema de archivos.

Para ejemplificar esta situación consideremos el ejemplo `8_8_1_link-failure-leaf-spine`. La cantidad de inodos generados por esta simulación es de 44.928 y para simulaciones más grande este número aumenta aun más. Así una ejecución secuencial de cientos de simulaciones, sin eliminar carpetas y archivos generados, fácilmente puede superar la cantidad máxima de inodos del sistema de archivos (algo más de 6 millones en una instalación estándar de Ubuntu).

Procesamiento de resultados

Por como se ejecuta la simulación los estudios de convergencia se realizan a posteriori analizando las capturas, debido a que nos resultó la forma más sencilla y práctica[23]. El procesamiento posterior de los resultados involucra analizar las capturas de tráfico generadas por los nodos. Mencionamos que se genera una captura por interfaz, de modo que al crecer la cantidad de nodos y la conectividad, la cantidad de archivos por nodo aumenta. En este sentido, el post-procesamiento involucra potencialmente recorrer todos los archivos y seleccionar la información relevante.

Por esto, de la ejecución del script `run_examples.sh` guardamos únicamente los archivos `.pcap` de capturas de tráfico generadas en las simulaciones. Con esto, el script `run_analyze.sh` realiza el posterior procesamiento y almacena los resultados. Destacamos que este post-procesamiento puede ejecutarse simultáneamente con las simulaciones, de forma de optimizar el tiempo cuando se tienen muchos casos de prueba para ejecutar.

El objetivo en el procesamiento de datos es obtener el número de mensajes BGP UPDATE que hubo a partir de cierto evento en la simulación (Node Failure, Node Recovery, Link Failure o Link Recovery). Si bien este procesamiento no es parte de las simulaciones, igual es importante tener algunos aspectos en cuenta a fin de que sea eficiente y no se convierta en un cuello de botella al ejecutar muchas simulaciones.

Una forma ingenua de realizar el procesamiento sería la siguiente: generar capturas

[22] Un inodo contiene la información de un archivo o carpeta del sistema de archivos. De esta manera el sistema operativo mantiene un inodo por cada archivo o carpeta.

[23] Se podría programar la ejecución *callback*, de modo que se estudien los paquetes recibidos por los nodos de forma *live*. Esto haría que la simulación finalice cuando se determina que se ha alcanzado la convergencia. Este enfoque es similar a lo encontrado en el proyecto Kathará realizado por el grupo MINA.

de tráfico de toda la simulación y luego recorrer todas las capturas generadas contando mensajes BGP UPDATE (recordando luego dividir este número entre 2 ya que se está contando cada paquete dos veces, una cuando se envía y otra cuando se recibe). Sin embargo, esta no es la forma más eficiente de hacerlo. A continuación describimos dos importantes optimizaciones que se le pueden hacer.

La optimización más importante es reducir la cantidad de tráfico que contienen las capturas. Dado que solo nos interesa el tráfico a partir de cierto evento lo mejor es que las capturas incluyan solo el tráfico a partir de ese momento[24]. Esto es importante ya que la parte inicial de las simulaciones (donde se ejecuta el Bootstrap) es la que tiene más tráfico en la red. De este forma, evitando guardar capturas en ese momento se logran archivos `.pcap` mucho más chicos lo que hace el procesamiento más eficiente.

La otra optimización que vale la pena mencionar es sobre qué archivos realizar el procesamiento. Dado que cada paquete es registrado dos veces (una vez en el nodo donde se origina, y otra vez en el nodo donde se recibe), no es necesario analizar todos los archivos para realizar el conteo. Nosotros optamos por solo estudiar las capturas generados por los Leaves y Tofs, descartando las capturas de los Spines. Esto es suficiente para analizar todos los paquetes enviados ya que un paquete siempre está o en un enlace Leaf-Spine o en un enlace Spine-Tof.

En nuestro caso, implementamos un analizador de los resultados mediante la combinación de un script en `python` y un script en `bash` (`run_analyze.sh`). El primero combina los archivos a analizar en uno solo (utilizando la biblioteca `scapy`), y el útltimo abre ese archivo utilizando `tshark`[25] para contar la cantidad de mensajes BGP UPDATE. Lo realizamos de esta forma porque con `scapy` el procesamiento de paquetes defragmentados resulta complejo, por lo que resultó más práctico hacerlo con `tshark`.

4.4. Análisis de resultados de las pruebas sobre fat trees

Como mencionamos en la sección 4.2 para el diseño de los casos de prueba nos basamos en la propuesta del proyecto Kathará, encontrándose entre las ejecuciones realizadas casos link-failure, link-recovery, node-failure y node-recovery. Por otro lado, realizamos ejecuciones de BGP con zebra y sin zebra, sobre lo cual ampliaremos en la sección siguiente. Por ahora es suficiente saber que ejecuciones sin zebra tienen la ventaja de ser más rápidas, de forma que las incluimos para mejorar performance.

Para la ejecución de los casos de prueba descritos asignamos valores a los parámetros *k_leaf* y *k_top* de forma que sean iguales. Esto hace que switches de distintos niveles en la construcción de los fat trees sean homogéneos. Por otro lado, dado un valor de *k_leaf* y *k_top*, variamos el parámetro *redundancy*, siempre asegurando que divida a *k_top*. Adicionalmente en la ejecución de los casos de prueba fuimos variando el nivel donde se

[24]Para lograr esto en una simulación de `ns-3` lo que se puede hacer es agendar un evento y que la función de `ns-3 EnablePcapAll` sea llamada desde allí. De esta forma se puede hacer que solo se guarden capturas de tráfico a partir de un tiempo dado (el tiempo en que se agendó el evento).
[25]`tshark` es la contrapartida para consola del popular visualizador de capturas de tráfico `wireshark`.

produce la falla de forma de cubrir todas las posibilidades. Se encuentran en el Anexo 8.2 todos los casos de prueba ejecutados junto con sus resultados.

Comparación de resultados con y sin zebra

El conteo de mensajes BPG UPDATE para las ejecuciones con o sin zebra es el mismo en la gran mayoría de los ejemplos considerados. Sin embargo, en algunos ejemplos existe una pequeña diferencia en el número de mensajes BGP UPDATE dependiendo si el ejemplo se ejecutó con o sin zebra. Estas diferencias se dan ocasionalmente en los casos link-recovery-spine-tof y node-failure-leaf. Posiblemente esto se deba a que al ejecutar con zebra las simulaciones siguen un proceso diferente para la comunicación con los vecinos, ya que utilizan zebra para encontrar las interfaces. Esto provoca que los eventos se generen en tiempos distintos, lo que lleva a algunas diferencias en la ejecución de BGP.

Adicionalmente en los resultados de las ejecuciones con y sin zebra incluimos el tiempo de ejecución de cada simulación. Se puede apreciar en ellos que las ejecuciones sin zebra permiten obtener resultados en menor tiempo, necesitando aproximadamente la mitad del tiempo que las ejecuciones con zebra.

Los casos más grandes que se pudieron ejecutar sin zebra fueron los `14_14_1_*` (980 nodos) mientras que con zebra fueron los `8_8_1_*` (320 nodos). Esto quedó determinado por la cantidad de memoria RAM disponible en el equipo donde se ejecutaron las pruebas (98.4GB). Como era esperable las ejecuciones con zebra consumen más memoria que las sin zebra, especialmente teniendo en cuenta lo que se discutirá en la sección *Consideración sobre ejecuciones con zebra.*

Finalmente, dado que las ejecuciones sin zebra han demostrado ser más eficientes que las ejecuciones con zebra en cuanto al tiempo de ejecución y consumo de recursos, concluimos que, si el caso de estudio lo permite, es preferible utilizar las ejecuciones sin zebra.

Comparación de nuestros resultados con los del proyecto Kathará

Los tutores nos proporcionaron resultados de las ejecuciones sobre el emulador Kathará, las cuales incluimos para la comparación de resultados, como se muestra en el Anexo 8.2. Puede apreciarse que hay ejecuciones que presentan más de un resultado, lo que suele suceder en ambientes emulados. En nuestro caso, utilizar un simulador tiene la ventaja de que el comportamiento de las simulaciones es determinista, por lo que siempre se obtiene el mismo resultado.

Como se observa en el cuadro de resultados, los números obtenidos por los emulaciones con Kathará son muy cercanos a los obtenidos con nuestras simulaciones de `ns-3 DCE`. En particular podemos decir que las ejecuciones sin zebra presentan números muy similares a los de Kathará (en muchos casos exactamente iguales). Esto nos da una pauta de comparación para validar el trabajo realizado sobre `ns-3 DCE`, donde se buscó permitir la ejecución de código de `FRR`, así como también nos refuerza en la conclusión de que las ejecuciones sin zebra son preferibles para el estudio de BGP.

Los casos `10_10_1_node-failure-spine` y `10_10_1_node-failure-tof` son los únicos donde hubo diferencia significativa entre nuestros resultados y los del proyecto Kathará. Estos son los casos más grandes (500 nodos) de los cuales tenemos resultados del proyecto Kathará. En nuestros resultados el conteo de mensajes BGP UPDATE se muestra consistente en su aumento, en comparación con los demás resultados obtenidos. Por otro lado, visto que el proyecto Kathará solo pudo ejecutar con 500 nodos estos dos ejemplos, pensamos que la diferencia se debe a que en estos casos llegaron al límite posible de ejecución en el hardware del que disponían.

Como comentario, en etapas tempranas de ejecución de casos de prueba y análisis de resultados, se llevaron a cabo correcciones de errores de implementación. Para esto fue de gran utilidad contar con los resultados obtenidos por las emulaciones en Kathará. Estos resultados incluyen diagramas que ilustran el intercambio de paquetes producido por BGP para la convergencia. De esta manera, utilizando esa información junto con las capturas de tráfico producidas por nuestras simulaciones, se llevó a cabo un análisis paquete a paquete para algunos casos puntuales. Esto contribuyó tanto en la identificación y corrección de erores, así como para comprobar que la cadena de paquetes se produce adecuadamente según la lógica de BGP.

Consideración sobre ejecuciones con zebra

Al ejecutar la serie de ejemplos nos encontramos con que algunas de las ejecuciones con zebra terminan con error `SEGFAULT` lo que no nos ocurrió en las pruebas iniciales del porte, que fueron limitadas en su alcance. Las simulaciones donde se incluye zebra resultan muy sensibles a los tiempos establecidos para inicio de los demonios y tiempos de eventos generados. Para algunos casos, cambiando ligeramente estos tiempos se puede lograr una ejecución exitosa.

Utilizando `gdb` identificamos que el error `SEGFAULT` ocurre en el método `Free` de la clase `KernelSocketFdFactory`[26] de `DCE`. Los métodos de esta clase son invocados por la capa Kernel de `DCE` y no por programas ejecutándose sobre `DCE`. Esta capa contiene al kernel de Linux compilado a fin de usar la implementación del stack de redes propia de Linux. Esta clase contiene los métodos que utiliza la capa Kernel para el manejo de memoria. Asumimos es aquí donde se produce el error y no en el kernel de Linux, que es código que ha sido testeado exhaustivamente y por lo tanto más confiable.

Las condiciones que producen el error no son claras. Para el estudio de un caso particular, vimos que este ocurre una vez en más de un millón de invocaciones al método `Free` donde se realiza una liberación de memoria[27].

A fin de poder ejecutar las simulaciones que finalizan con error, optamos por comentar el código del método `Free`. Esto es posible porque este método solo libera memoria sin afectar comportamiento. El resultado de esto es que evitamos el error `SEGFAULT`

[26] El error se produce en línea 162 del archivo `model/kernel-socket-fd-factory.cc`.

[27] Para esto se realiza una lectura de un *buffer* para determinar cuanta memoria liberar, pero el mismo no puede ser accedido, posiblemente porque se liberó previamente.

pero a cambio se deja memoria colgada. Realizamos una prueba a fin de determinar cuanta memoria extra nos conlleva esto. Para el caso `12_12_3_link-failure-leaf-spine` de 240 nodos (donde no se produce el error) la ejecución sin modificar el método `Free` consume 43.98GB de memoria RAM. La ejecución del mismo ejemplo con el método comentado consume 52.35GB de memoria RAM. Esto significa un aumento en el uso de memoria de 19.03 %.

Como conclusión, si bien la implementación de esta alternativa permite ejecutar aquellos casos con zebra que no finalizan exitosamente, no resuelve el error de fondo, sino que lo evita. Por esto, no la incluimos como parte del porte de `FRR` a `ns-3 DCE` y la publicamos en una rama separada[28]. De todas maneras, de no utilizar la alternativa que comenta el método `Free`, igualmente es posible ejecutar los casos con errores cambiando su ejecución a sin zebra.

4.5. Limitantes en el rendimiento de las simulaciones y mejoras implementadas

Al comenzar con la ejecución de los casos de prueba descritos anteriormente nos enfrentamos a dificultades con respecto al rendimiento de las simulaciones en cuanto al tiempo de ejecución y el consumo de recursos. Con el objetivo de mejorar estas limitantes, se decidió analizar algunas de sus causas y proponer soluciones.

Las pruebas se realizaron en un entorno Ubuntu 16.04 con el ambiente instalado como se describió en la sección 3.2. Es una máquina virtual en un servidor de la facultad que tiene asignado 26 CPUs AMD Opteron 63xx class. Además cuenta con 98.4G de memoria RAM.

Mejoras en tiempo de ejecución

Por mencionar un ejemplo, la simulación `8_8_1_link-failure-leaf-spine` que tiene 320 nodos demora 1h15min en ejecutar 1min de tiempo simulado. Este tiempo es el obtenido ejecutando en la máquina ya descrita, la cual tiene buenos recursos.

Para mejorar este tiempo propusimos tres cambios:

- Deshabilitar IPv6.
- Minimizar tiempo simulado.
- Ejecutar el demonio BGP sin zebra.

Durante el análisis de resultados (observando las capturas de tráfico) notamos el envío de muchos paquetes *ICMPv6 Router Solicitation* e *ICMPv6 Router Advertisement*,

[28]Rama *sol_provisoria* del proyecto publicado en https://gitlab.fing.edu.uy/proyecto-2021/ns-3-dce.

entre otros paquetes IPv6. Esto se debe a que la instalación de nuestro ambiente utiliza `DCE` con el kernel de Linux, el cual por defecto utiliza IPv6 al levantar una interfaz.

Por lo tanto a fin de reducir la carga producida por los nodos, deshabilitamos este protocolo mediante la ejecución del comando "`ip -6 address flush dev <interfaz>`", inmediatamente cuando se levanta la interfaz.

Por otro lado, inicialmente a una simulación le asignamos 60 segundos. Sabemos a partir de varias ejecuciones de simulaciones bootstrap (sin fallas en enlaces o nodos) que la convergencia se alcanza siempre en menos de 10 segundos. Por tanto ajustamos los tiempos de modo que utilice 10 segundos para cada vez que se necesita converger:

- Para el caso de bootstrap le damos 10 segundos.
- Para los casos de node-failure y link-failure son necesarios 20 segundos, dado que el caso de prueba incluye iniciar la topología (bootstrap), esperar la convergencia, luego producir la falla en el enlace/nodo para finalmente esperar otra vez la convergencia.
- Para los casos de node-recovery y link-recovery se asignan 30 segundos, por una razón similar a la anterior.

Sin embargo, el principal cambio introducido para reducir el tiempo de simulación fue ejecutar el demonio BGP sin ejecutar el demonio zebra.

BGP por sí mismo no necesita de zebra para funcionar. Pero al ejecutar sin zebra no se actualizan las rutas del kernel. Esto significa que los nodos intercambiarán información de las rutas, pero estas no serán almacenadas. Como consecuencia, no existirá conectividad entre los nodos.

Por otro lado, ejecutar BGP sin zebra no permite el descubrimiento de interfaces. Por esto, en la configuración de BGP no se pueden referir a los *BPG peers* de forma genérica a través de la interfaz, como se explicó en la sección 2.4. En su lugar, se debe referir a los *BPG peers* a través de su IP.

Como nuestro propósito es simplemente estudiar características del plano de control, en particular la convergencia de BGP, las desventajas mencionadas no representan un inconveniente. Si se precisara alguna funcionalidad del plano de datos que requiera conectividad real, entonces no se podría prescindir del demonio zebra.

Para ejecutar BPG sin zebra se realizaron los siguientes ajustes:

- Ejecutar `bgpd` con la opción `--no_zebra`.
- Modificar el `FrrHelper` para que no levante zebra y se le pase la opción anterior cuando se utilice el método `RunBgpWithoutZebra`, mencionado en la sección 3.5.
- Modificar el archivo de configuración que se genera (`bgpd.conf`) para que se refiera a los vecinos (*BGP peers*) a través de la IP y no de la interfaz, aunque esto no es una buena práctica como se explicó en la sección 2.4

Caso	Tiempo	Memoria	% mejora de tiempo
Sin cambios	2m50.140s	705M	-
Deshabilitar IPv6	1m44.284s	705M	38.71 %
Minimizar tiempo simulado	2m18.452s	705M	18.62 %
Ejecutar sin zebra	1m29.786s	403M	47.23 %
Todos los cambios	**1m18.833s**	**403M**	**53.67 %**

Cuadro 2: Resultados obtenidos al ejecutar el ejemplo `6_6_6_link-failure-spine-tof`, aplicando cambios para mejorar performance.

Estas mejoras implementadas son opcionales. De esta manera es posible ejecutar las simulaciones aplicándolas en simultáneo (lo que llamamos ejecución sin zebra) o sin ninguna de ellas (ejecución con zebra).

A fin de ilustrar la mejora de perfomance que los cambios anteriores produjeron, se ejecutó el ejemplo `6_6_6_link-failure-spine-tof`. En el mismo se aplicaron los cambios de manera independiente, y luego todos juntos. El Cuadro 2 ilustra los resultados de tiempo de ejecución, memoria utilizada y porcentaje de mejora para el cambio introducido.

Consumo de memoria

El consumo de memoria limita la cantidad de nodos que pueden ser simulados. Una simulación con 320 nodos (como la `8_8_1_link-failure-leaf-spine` por ejemplo) usa 12GB de memoria y de quedarse sin memoria disponible la simulación termina con error. Esto limita los equipos donde se pueden ejecutar las simulaciones.

El cambio realizado anteriormente, ejecutando sin zebra, resultó en una reducción de memoria del 42.83 % como se ve en el Cuadro 2. Esto permite ejecutar ejemplos medianamente grandes en equipos de mediano porte.

5. Paralelismo en ns-3 DCE

Dadas las limitantes descritas en la sección 4.5, surgió la iniciativa de mejorar el tiempo de ejecución y posibilitar la escala de las simulaciones. En esta sección describimos la experiencia transitada a tal fin mediante la utilización de paralelismo, aunque no fue posible. Igualmente se consideró relevante documentar las acciones intentadas y posibles formas de abordar el problema.

5.1. Motivación

Como vimos, ns-3 DCE está implementado para ejecutar en un solo hilo. Introduciendo paralelismo se podría distribuir una misma simulación para que ejecute en varios hilos, sea en una misma máquina o utilizando varias. Esto podría reducir el tiempo de procesamiento, lo que significaba una limitante.

Por otro lado, cuantos más nodos se añaden a la simulación, más memoria esta consume. Eventualmente se alcanzarán los recursos de la máquina, lo que añade un límite superior a las simulaciones. Este límite se podría incrementar si la simulación estuviera divida en varias máquinas, de forma de distribuir la memoria requerida por la simulación.

5.2. Propuesta de paralelismo

Para poder paralelizar una simulación hay que diseñar una estrategia para separar el procesamiento entre distintas CPUs. Naturalmente se llega a que se debe ejecutar en un mismo proceso todas las aplicaciones de un mismo nodo. Llamamos Proceso Lógico (PL) a la parte de una simulación que ejecuta en el mismo proceso. Cuando un canal une dos nodos que están en el mismo PL entonces los paquetes que por allí pasan serán procesados también por ese PL. El problema ocurre cuando un canal une dos nodos en distintos PLs. En ese caso hay que diseñar una forma de compartir información entre procesos.

La decisión principal a tomar para diseñar una estrategia de paralelización pasa por como dividir los nodos entre los PLs. Lo que se busca es que los nodos queden distribuidos de forma equitativa entre los PLs a la vez que se busca disminuir la cantidad de canales que unen nodos en distintos PLs. Esto último es porque esta situación requiere un trabajo extra que genera una sobrecarga en la simulación.

Para nuestro caso de estudio que son los fat trees surge una división natural para los nodos en PLs. Esta división se basa en identificar los Pods del fat tree con PLs, así como los planos de la capa de agregación con PLs. Un ejemplo de esto se muestra en la Figura 11. En la misma se representan con líneas azules los canales que atraviesan PLs. Estos se corresponden con conexiones Point to Point entre los switches del fat tree. Para el caso del paralelismo estas conexiones son las que generarían sobrecarga al necesitar tener un método de intercambio de información (en este caso, paquetes intercambiados) entre distintos procesos.

Adicionalmente, para poder paralelizar una simulación es necesario resolver como

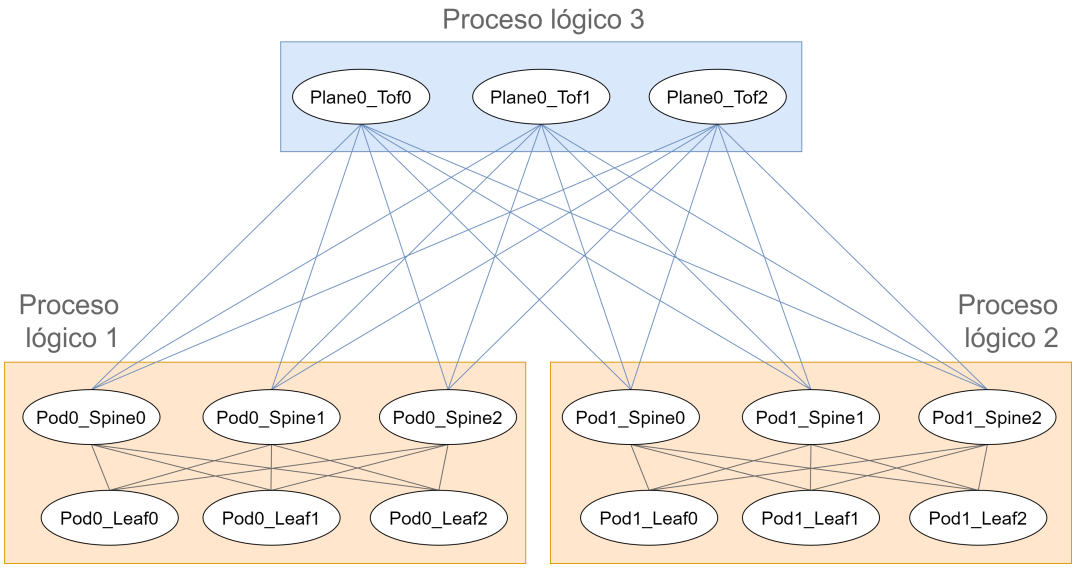

Figura 11: Estrategia de paralelismo. Identificamos cada Pod con un Proceso Lógico (PL). De igual forma cada plano lo identificamos con un PL. De esta forma buscamos disminuir la sobrecarga producida por la sincronización al paralelizar.

sincronizar el reloj entre los distintos PLs. Hay que tener en cuenta esto para que haya consistencia en los tiempos de envío-recepción de paquetes por parte de nodos que ejecutan en diferentes PLs.

5.3. ns-3 DCE con MPI

Un primer acercamiento a la búsqueda de proporcionar paralelismo a las simulaciones, es buscar herramientas ya existentes en ns-3. En este sentido ns-3 tiene un módulo a tal fin. *Message Passing Interface* (MPI) es un estándar de pasaje de mensajes diseñado para facilitar la computación paralela. Existen varias implementaciones del estándar, entre los que se encuentra OpenMPI [23], que es la utilizada en ns-3.

La simulación de eventos discretos en paralelo y distribuida permite la ejecución de un mismo programa de simulación en múltiples procesadores. Al dividir la simulación en Procesos Lógicos, cada PL puede ser ejecutado por un procesador diferente. Esta metodología de simulación permite simulaciones a muy gran escala aprovechando una mayor potencia de procesamiento y disponibilidad de memoria. Para garantizar la ejecución adecuada de una simulación distribuida, se requiere el pasaje de mensajes entre PLs. A tales efectos se utiliza MPI junto con una nueva clase de simulador distribuido. Como comentario, actualmente dividir una simulación para propósitos distribuidos en ns-3 solo puede ocurrir a través de enlaces Point to Point [24].

Lo antes mencionado aplica para ns-3, pero es necesario considerar su funcionamiento con DCE. Para el caso de ns-3 DCE, hubo una actualización del proyecto en 2012

para poder usar MPI también con DCE[29]. Este fue un agregado muy simple y no ha tenido actualizaciones desde entonces.

En nuestro caso, se intentó utilizar lo ya existente e importarlo a nuestro ambiente ya configurado. Sin embargo son necesarias algunas consideraciones para que ns-3 DCE reconozca MPI. En primera instancia es necesario realizar nuevamente el configure y el build tanto de ns-3 como de DCE. Para esto modificamos el archivo bakeconf.xml, agregando --enable-mpi[30]. Este cambio logró que ns-3 reconozca MPI, pero en nuestro caso no fue así para DCE. Por lo tanto ejecutamos el configure manualmente para conseguirlo[31]. Además, fue necesario modificar el wscript del módulo *ns-3-dce-frr* para incluir referencias a MPI en los ejemplos que lo requieran.

Luego de todo esto se consigue que ns-3 DCE reconozca MPI, pero al ejecutar tanto los propios ejemplos del proyecto citado del año 2012, como los ejemplos implementados por nosotros, la simulación da diversos errores en tiempo de ejecución. Entre ellos, errores de Segmentation Fault y fallas por no encontrar bibliotecas. Los errores son diferentes en cada ejecución (por la propia ejecución en hilos), lo que resulta difícil la utilización de *debuggers* para encontrarlos.

La resolución de estos problemas exceden el alcance del proyecto, por lo que luego de varios intentos, planteamos otras alternativas que se detallan en las secciones siguientes.

5.4. ns-3 DCE paralelismo con Tap Bridges

Como alternativa a paralelizar con MPI, buscamos opciones para paralelizar de manera manual. Esto fue basado en el proyecto publicado en el paper *Automating ns-3 Experimentation in Multi-Host Scenarios* [25]. En dicho proyecto se hace uso del framework NEPI junto con tuneles UDP para automatizar las simulaciones de ns-3 utilizando multihost.

La idea entonces es ejecutar varias simulaciones de ns-3 DCE e interconectarlas. ns-3 brinda una facilidad para que sus instancias puedan interactuar con el mundo real, a partir del uso de Tap Bridges [26].

Tap Bridge está diseñado para integrar hosts de Internet "reales" (o más precisamente, hosts que admiten dispositivos Tun / Tap) en simulaciones ns-3. El objetivo es hacer que nodos reales puedan interactuar con nodos dentro de una simulación de ns-3.

[29] El *commit* de este cambio se encuentra en *https://github.com/direct-code-execution/ns-3-dce/commit/2fce773b64da06ccc4a14690080a988c67d681e7*.

[30] Esto se realiza para que el configure agregue el módulo MPI al proyecto. A modo informativo, si se quisiera replicar lo actuado, se deben ejecutar los scripts hasta el 13, luego se realiza el cambio que habilita MPI y finalmente se retoma la ejecución de los scripts del 14 al 16.

[31] Comando `python3 waf configure --enable-mpi --prefix=/home/proyecto/bake/build --with-ns3=/home/proyecto/bake/build --with-elf-loader=/home/proyecto/bake/build/lib --with-libaspect=/home/proyecto/bake/build --enable-kernel-stack=/home/proyecto/bake/source/net-next-nuse-4.4.0/arch`.

Para que esto sea posible a un dispositivo de red de un nodo de ns-3 se le asocia una interfaz Tap del host que ejecuta la simulación.

Dado que, en esencia, estamos conectando las entradas y salidas de un dispositivo de red ns-3 a las entradas y salidas de un dispositivo de red Tap de Linux, llamamos a esta disposición un Tap Bridge.

La idea general es dividir un caso de estudio en PLs, como ya se vio en la Figura 11, y ejecutar todos los PLs en distintos procesadores o hosts. Luego, mediante Tap Bridges poder sacar los paquetes fuera de la simulación de ns-3 DCE para que entren en la simulación de otro PL. Hay varias posibilidades para conseguir esto que se detallan más adelante.

Adicionalmente, existe el problema de sincronizar los relojes entre los PLs. Una posibilidad es configurar las simulaciones para que todas ellas ejecuten bajo tiempo real[32]. Dado que en los primeros segundos de simulación se inicia el demonio BGP en todos los nodos, estos segundos son muy intensos en consumo de CPU. Por lo tanto puede que no sea posible la utilización de tiempo real. Como alternativa, se podría buscar otro enfoque utilizando algún método de comunicación IPC para sincronizar relojes.

Multihost con ghost

La prueba la realizamos para un escenario pequeño con miras a extenderlo en caso de éxito. Consiste en tener 4 nodos en línea: dos PCs y dos routers, ejecutando BGP en ellos. Se busca distribuir los nodos en diferentes hosts creando una única conexión mediante una Tap Bridge.

El uso habitual de Tap Bridge es creando un nodo *ghost* que reemplace la Tap en el host Linux. En la Figura 12 ilustramos el escenario objetivo. Se tienen dos hosts, cada uno ejecutando una instancia de ns-3. Cada instancia de ns-3 tiene una una PC y un router con la subred 192.168.X.0/24. Además tiene un router ghost que simula el router que se encuentra en el otro host. Los routers entre sí están conectados mediante la subred 10.1.1.0/24.

Los paquetes que el router 1 envía al router 2, se envían al ghost router 2. Como la interfaz de este ghost a donde llegan los paquetes es la Tap Bridge (tap0 en la imagen) estos son vistos por el sistema operativo. El objetivo es que mediante un túnel (ipip0 en la imagen) estos paquetes lleguen a la Tap Bridge del host 2, y de allí entren a la instancia de ns-3 que se ejecuta en el host 2.

El problema con el que nos encontramos es el siguiente. Un paquete que va desde el router 1 hacia el ghost router 2 tiene como dirección de destino 10.1.1.2. Esta dirección es una de las direcciones locales del host 1 (ya que es la dirección que tiene la interfaz tap0). Como consecuencia, el paquete no pasa por el túnel, dado que el host1 interpreta

[32]Para poder integrarse con stacks de redes reales y enviar/recibir paquetes, es necesario que ns-3 use un planificador de tiempo real para que el reloj simulado quede asociado al reloj real. ns-3 provee esta funcionalidad en el componente RealTime Scheduler.

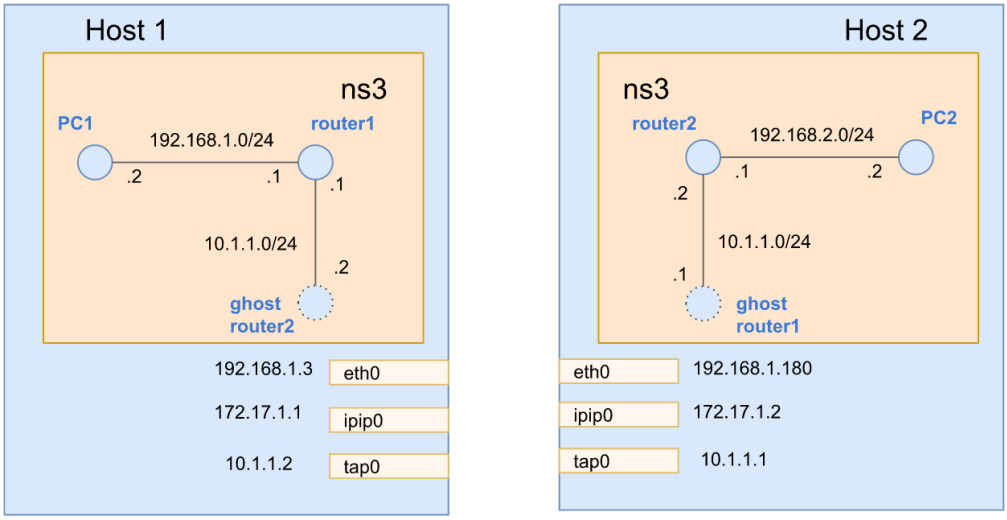

Figura 12: Intento de paralelizar utilizando Tap Bridges. Se tienen dos hosts, cada uno con una instancia de `ns-3`. El caso completo que se quiere simular consiste en cuatro nodos: dos routers y dos PCs. Cada instancia de la simulación contiene dos de estos nodos y además un nodo *ghost* que contiene la Tap Bridge. El nodo ghost de una instancia se corresponde con el router de la otra. La idea es que los paquetes que llegan al ghost puedan, a través de la Tap Bridge (tap0), pasar al sistema operativo y luego ir hacia el otro host e ingresar a la otra instancia de `ns-3`. Este paso se realizaría mediante el uso de túneles (ipip0) pero debido a problemas de ruteo no fue posible completar esta idea.

que llegó a destino, quedando en tap0.

Multihost sin ghost

Este caso es igual al anterior pero con la diferencia de que no usa nodos ghost en las simulaciones. La Tap Bridge del host 1 se corresponde con la interfaz con IP `10.1.1.1` del router 1. Por otro lado, la Tap Bridge del host 2 se corresponde con la interfaz con IP `10.1.1.2` del router 2.

La idea en este escenario es la siguiente. Un paquete sale del router 1 con destino `10.1.1.2`. Como ahora la Tap Bridge se encuentra en ese nodo, este paquete es visto por el sistema operativo en la interfaz tap0. En el host 1, mediante reglas de ruteo se define que los paquetes con destino `10.1.1.2` vayan al host 2. De esta forma, los paquetes llegarían a la Tap Bridge del host 2 y entrarían en la simulación del host 2.

El problema con este planteo es que el porte de `FRR` se realizó utilizando el Advanced Mode de `DCE`, como se vio en la sección 3. Esto hace que los routers que están ejecutando `FRR` utilicen el stack de redes de Linux. Al usar ese stack, antes de mandar un paquete con destino `10.1.1.2` el router 1 primero busca averiguar la MAC del nodo con esa IP, mediante el protocolo ARP. Como los paquetes ARP son de capa 2 no pueden pasar por un tunel ipip ni son dirigidos al host 2 por reglas de ruteo. Por consiguiente los paquetes ARP son dropeados y no se llega a establecer la conexión `TCP` entre los routers necesaria para que funcione `FRR`.

Multihilo en un mismo host

Visto los problemas del caso anterior buscamos realizar un caso menos ambicioso y paralelizar en un solo host. La propuesta es tener dos simulaciones de ns-3 DCE ejecutándose simultáneamente en la misma PC y mediante Tap Bridges lograr que se comuniquen.

Para esto buscamos ejecutar el mismo ejemplo que Multihost sin ghost solo que ejecutando ambas simulaciones en el mismo host. Al estar las simulaciones ejecutándose en el mismo host el túnel ipip resulta innecesario, pero tendremos dos interfaces Tap en el host (tap0 y tap1), una por cada simulación. La idea es hacer un bridge entre estas interfaces de forma que puedan intercambiar paquetes. De este modo, cuando el router 1 mande un paquete con destino 10.1.1.2, este llegue a la tap0 y mediante el bridge sea visto por la tap1. De esta forma se lograría la comunicación entre dos simulaciones distintas.

Al igual que el caso anterior el problema en este enfoque se dio por los paquetes ARP. Estos paquetes no pasan por el bridge y nos encontramos en la misma situación que en el caso anterior.

Como forma de solventar estos problemas, se consideró también la idea de ejecutar este mismo ejemplo pero con las dos simulaciones asociadas a la misma Tap Bridge. Sin embargo, esto no fue posible. Esto es porque una interfaz Tap puede estar asociada a una sola Tap Bridge de ns-3.

6. Conclusiones

Gracias al trabajo realizado contamos con una forma de ejecutar simulaciones en `ns-3` usando `FRR`. De esta manera contamos con las ventajas de ambos. Por un lado con `ns-3` estamos utilizando un simulador de código abierto, modular, cuyo ambiente es controlado y reproducible. Por otro lado, al mismo tiempo estamos utilizando el proyecto de código abierto `FRR`, diseñado para optimizar el stack de protocolos de enrutamiento. Como mencionamos, la utilización del código nativo de `FRR` dentro de `ns-3` se llevó a cabo gracias al módulo `DCE` de `ns-3`.

Dejamos público en el repositorio GitLab de facultad el trabajo realizado[33]. El mismo consiste en un grupo de tres proyectos. Por un lado, todos los pasos necesarios para la configuración de ambiente, así como modificaciones a `FRR`, pueden encontrarse en el proyecto *Scripts*. También lo incluimos en el Anexo 8.1, donde fue útil para facilitar referencias y explicaciones de la configuración.

Por otro lado, en el proyecto *ns-3-dce* se encuentra el *fork* del proyecto original `ns-3 DCE` sobre el que estuvimos trabajando. En el mismo se encuentran los agregados a `DCE` que permiten el porte y se encuentra el módulo *ns-3-dce-frr* con ejemplos y utilidades para ejecutar simulaciones utilizando el porte. En particular, se encuentran simulaciones parametrizables sobre fat trees.

Finalmente el proyecto *Run scripts* contiene scripts para la ejecución de simulaciones en cadena, que utilizan los ejemplos implementados en el módulo *ns-3-dce-frr*. Además se cuenta con otro script para el post-procesamiento de las mismas.

Adicionalmente, se realizaron *Pull Requests* al proyecto original de `ns-3 DCE`. Por un lado se realizó un *Pull Request* con los agregados a `DCE` que permiten el porte. Por otro lado se realizaron otros con dos *bugs* corregidos.

Sin dudas el porte de `FRR` a `ns-3 DCE` significa nuestro principal aporte a la comunidad. Esto contribuye al objetivo de estudiar el desempeño de BGP en topologías fat trees, como se venía realizando en otras oportunidades. Y más en general, contar con `FRR` sobre `ns-3` permite disponer de todos los protocolos de enrutamiento soportados por `FRR` para su uso en un ambiente simulado.

Como mencionamos en la sección 3, la instalación de DCE fue compleja debido a la falta de documentación. Por falta de referencias en problemas puntuales que tuvimos, algunas cosas se hicieron a prueba y error. Ejemplo de esto son las dependencias requeridas en la instalación de `DCE` y la utilización de la herramienta `waf` al momento de hacer el `build` con scripts implementados por nosotros.

Fue de gran utilidad el porte de `Quagga` realizado anteriormente, sobre el cual nos basamos para la implementación de utilidades como el `FrrHelper`. Adicionalmente, el uso de *debuggers* resultó una herramienta indispensable durante todo el porte, ayudando a la identificación de errores en tiempo de ejecución tanto en el código de `ns-3 DCE` como

[33] El repositorio se encuentra en *https://gitlab.fing.edu.uy/proyecto-2021*. Contiene los proyectos *ns-3-dce*, *Run scripts* y *Scripts*.

de FRR. Esto fue gracias al enfoque que toma DCE de facilitar la depuración. En particular, ejecutar todo en un solo hilo hace posible la utilización de *debuggers* tradicionales.

La implementación de casos de prueba fue otro de los objetivos del proyecto. Pudimos ejecutar más de 300 casos de pruebas en fat trees utilizando el protocolo BGP para el datacenter. Los resultados obtenidos para muchos de los casos ejecutados se encuentran en el Anexo 8.2. Para ayudar en la construcción de las simulaciones, fue de gran utilidad el proyecto VFTGen de Kathará que implementa un generador de fat trees virtuales. Nos basamos en el mismo tanto para comprender la generación parametrizable de los fat trees, como para implementar la nuestro propia.

Podemos concluir que se cumplieron los objetivos del proyecto y como resultado se cuenta con un simulador de redes que permite ejecutar FRR, con un generador de pruebas para topologías fat tree y un analizador de capturas para estudiar los resultados.

Como vimos en la sección 5, no conseguimos ejecutar en paralelo instancias de ns-3 DCE, pese a intentarlo con MPI y además de forma manual. Si fuera necesario continuar con la investigación, nos inclinamos fuertemente por el uso de MPI, dado que ya resuelve problemas de sincronismo. Con respecto a los intentos realizados de forma manual, realizando un análisis posterior creemos que no son eficientes y agregan complejidad.

De todas maneras, si se lograra el paralelismo, aun no podemos garantizar que esto efectivamente haga disminuir el tiempo de ejecución, dado que no sabemos si el sincronismo requerido puede significar una carga extra. No obstante, el paralelismo permitiría realizar simulaciones más grandes, dado que actualmente la cota en el tamaño de las simulaciones está dado por la memoria RAM del host donde se ejecutan.

Como trabajo a futuro sería bueno verificar el porte de los demás protocolos de la familia FRR que no llegamos a probar. Las pruebas durante el proyecto estuvieron enfocadas en la ejecución de BGP, que puede ejecutarse con o sin zebra. En este sentido, portar zebra fue lo que requirió mayor esfuerzo, dado que por la arquitectura de FRR la complejidad se traslada a zebra y no a los demonios que implementan protocolos. Por otro lado, portar BGP resultó comparativamente sencillo. Adicionalmente probamos OSPF, como mencionamos en la sección 4.1, en cuyo caso no fue necesario hacer nada nuevo en el porte para que funcionara correctamente. Visto esto esperamos que otros protocolos también funcionen correctamente sin necesidad de hacer más modificaciones.

Asimismo proponemos estudiar la falla en ejecuciones puntuales con zebra, que fue descrita en la sección *Consideración sobre ejecuciones con zebra*. Mencionamos que estos inconvenientes fueron detectados para casos grandes, cuya ejecución se dio en la etapa final del proyecto. Si bien mediante una alternativa[34] se pudieron ejecutar los casos con error, se está lejos de solucionar el problema de fondo.

Finalmente, sería bueno dedicar esfuerzos a mejorar la presentación de resultados en base a las capturas generadas por las simulaciones. Inspira dicha motivación lo realizado en el proyecto Kathará que cuenta con diversas visualizaciones de los datos.

[34]Publicada en la rama *sol_provisoria* del proyecto *ns-3-dce*.

En nuestro caso solo llegamos a presentar el conteo de mensajes BGP UPDATE y una imagen con el grafo generado según la topología de la prueba.

7. Referencias

[1] Facultad de Ingeniería UdelaR. https://www.fing.edu.uy/. [Último acceso Agosto 2021].

[2] Free Range Routing. https://frrouting.org. [Último acceso Octubre 2021].

[3] ns-3 Network Simulator. https://www.nsnam.org. [Último acceso Julio 2021].

[4] Quagga. https://www.quagga.net/. [Último acceso Noviembre 2021].

[5] ns-3 Direct Code Execution. https://www.nsnam.org/about/projects/direct-code-execution. [Último acceso Julio 2021].

[6] Emulador vs Simulador. https://ichi.pro/es/emulador-vs-simulador-135241110897. [Último acceso Agosto 2021].

[7] Mininet - An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/. [Último acceso Agosto 2021].

[8] G. Lospoto G. Bonofiglio, V. Iovinella and G. Di Battista. Kathará: A container-based framework for implementing network function virtualization and software defined networks. NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, Taipei, 2018, pp. 1-9, doi: 10.1109/NOMS.2018.8406267.

[9] Qué es un datacenter. https://www.datos101.com/blog/que-es-un-data-center/. [Último acceso Noviembre 2021].

[10] Use of BGP for Routing in Large-Scale Data Centers (rfc7938). https://datatracker.ietf.org/doc/html/rfc7938. [Último acceso Agosto 2021].

[11] The Traditional Network Infrastructure Model and Issues Associated with it. https://www.pluribusnetworks.com/blog/traditional-network-infrastructure-model-and-problems-associated-with-it/. [Último acceso Agosto 2021].

[12] Dinesh G. Dutt. BGP in the Data Center. O'Reilly Media, Inc.., 2017.

[13] ns-3 manual. https://www.nsnam.org/docs/release/3.34/manual/singlehtml/index.html. [Último acceso Agosto 2021].

[14] Emilio Mancini Mathieu Lacage Daniel Camara et al.. Hajime Tazaki, Frédéric Urbani. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. The 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2013.

[15] Engler D. R. Ganger G. R. Brice no H. M. Hunt R. Mazieres D. Pinckney T. Grimm R. Jannotti J. Kaashoek, M. F. and K. Mackenzie. Application performance and exibility on exokernel systems. In Proceedings of the sixteenth ACM symposium on Operating systems principles. (New York, NY, USA, 1997), SOSP '97, ACM, pp. 52-65.

[16] ns-3 Direct Code Execution (DCE) Manual. https://www.nsnam.org/docs/dce/manual/ns-3-dce-manual.pdf, 2020. [Último acceso Julio 2021].

[17] Dce quagga. https://www.nsnam.org/docs/dce/manual-quagga/html/getting-started.html. [Último acceso Agosto 2021].

[18] Bake. https://www.nsnam.org/docs/bake/tutorial/html/bake-tutorial.html. [Último acceso Julio 2021].

[19] FRRouting - Installation. https://docs.frrouting.org/en/latest/installation.html. [Último acceso Octubre 2021].

[20] Building FRR - Static Linking. http://docs.frrouting.org/projects/dev-guide/en/latest/static-linking.html. [Último acceso Octubre 2021].

[21] Kathara-Labs. https://github.com/KatharaFramework/Kathara-Labs. [Último acceso Septiembre 2021].

[22] Mariano Scazzariello Tommaso Caiazzi. Routing in data centers: Test features. February 3, 2020.

[23] Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/. [Último acceso Octubre 2021].

[24] MPI for Distributed Simulation. https://www.nsnam.org/docs/models/html/distributed.html. [Último acceso Octubre 2021].

[25] Thierry Turletti Walid Dabbous. Alina Quereilhac, Damien Saucez. Automating ns-3 Experimenta- tion in Multi-Host Scenarios. https://hal.inria.fr/hal-01141000/document, NS3 2015, May 2015, Barcelona, Spain. hal-01141000. [Último acceso Octubre 2021].

[26] Tap Bridge Model. https://www.nsnam.org/docs/release/3.9/doxygen/group__tap_bridge_model.html. [Último acceso Octubre 2021].

8. Anexos

8.1. Scripts de preparación de ambiente

Aquí se presentan los scripts utilizados para la preparación de ambiente a fin de utilizar `FRR` en simulaciones de `ns-3 DCE`. Estos scripts realizan toda la instalación de `ns-3 DCE`, `FRR`, junto a todas sus dependencias. Los mismos fueron escritos para Ubuntu 16.04 y deben ser ejecutados en el orden en que se muestran.

swig-install

```
#!/bin/bash
# Run this script as root
set -e

# Some frr dependencies
apt-get update
apt-get install cmake git curl autoconf automake libtool make libreadline-
    dev texinfo libpam0g-dev libjson-c-dev python3-pytest libc-ares-dev
    libsystemd-dev python-ipaddress install-info libsnmp-dev perl libcap-
    dev libelf-dev libpcre3-dev libboost-all-dev python3-apt flex bison
    python3-pip libpcre2-dev -y

# swig 3.0.12 - FRR dependency
cd ~
wget https://prdownloads.sourceforge.net/swig/swig-3.0.12.tar.gz
tar xvzf swig-3.0.12.tar.gz
cd swig-3.0.12/
./configure --prefix=$HOME --with-perl5=/usr/local/bin/perl --with-python
    =/usr/bin/python --with-ruby=/usr/bin/ruby
make
make check
make install
```

debhelper-install

```
#!/bin/bash
# Run this script as root
set -e

# debhelper 11 - FRR dependency
add-apt-repository ppa:jonathonf/debhelper-11 -y
apt-get update
apt-get install debhelper -y
```

libyang-install

```
#!/bin/bash
# Run this script as root
set -e

# libyang - FRR dependency
cd ~
git clone https://github.com/CESNET/libyang.git
cd libyang/
git checkout v1.0.225

# Use -O0 flag instead of -O3 so that it not uses __str* symbols
sed -i 's/-O3/-O0/' CMakeLists.txt

mkdir build
cd build
export CC=gcc
export CFLAGS='-ggdb -fPIC -U_FORTIFY_SOURCE'
export LDFLAGS='-pie'
cmake -DENABLE_LYD_PRIV=ON -DCMAKE_INSTALL_PREFIX:PATH=/usr -D
    CMAKE_BUILD_TYPE:String="Release" ..
make
make install
```

install-frr

```
#!/bin/bash
# Run this script as root
set -e

# frr
# http://docs.frrouting.org/projects/dev-guide/en/latest/building-frr-for-
    ubuntu1604.html
cd ~
git clone https://github.com/frrouting/frr.git frr
cd frr
git checkout frr-7.5.1
# To not use sys_futex, instead simply use mutex
sed -i 's/needsync=true/needsync=false/' configure.ac
# To not use pthread_condattr_setclock
sed -i 's/test "$frr_cv_pthread_condattr_setclock" = "yes"/false/'
    configure.ac
./bootstrap.sh
export CC=gcc
export CFLAGS='-ggdb -fPIC -U_FORTIFY_SOURCE'
export LDFLAGS='-pie'
./configure \
 --prefix=/usr \
 --includedir=\${prefix}/include \
```

```
 --enable-exampledir=\${prefix}/share/doc/frr/examples \
 --bindir=\${prefix}/bin \
 --sbindir=\${prefix}/lib/frr \
 --libdir=\${prefix}/lib/frr \
 --libexecdir=\${prefix}/lib/frr \
 --localstatedir=/var/run/frr \
 --sysconfdir=/etc/frr \
 --with-moduledir=\${prefix}/lib/frr/modules \
 --with-libyang-pluginsdir=\${prefix}/lib/frr/libyang_plugins \
 --enable-configfile-mask=0640 \
 --enable-logfile-mask=0640 \
 --enable-snmp=agentx \
 --enable-multipath=64 \
 --enable-user=root \
 --enable-group=root \
 --enable-vty-group=root \
 --enable-static  \
 --enable-static-bin  \
 --enable-shared  \
 --with-pkg-git-version \
 --with-pkg-extra-version=-frr-ns3-dce \
 --disable-capabilities \
 --enable-gcc-rdynamic

# Not use optimizations
sed -i 's/-O2/-O0/' Makefile
# Coment out the use of __pthread_register_cancel symbol
sed -i 's/pthread_cleanup_p/\/\/pthread_cleanup_p/' bgpd/bgp_keepalives.c
# To avoid log buffering
sed -i 's/if (mkdir(TMPBASEDIR, 0700)/goto out_warn; if (mkdir(TMPBASEDIR,
    0700)/' lib/zlog.c
sed -i 's/zlog_err(\"crashlog and per-thread/\/\/zlog_err(\"crashlog and
   per-thread/' lib/zlog.c

make install
```

limit-open-files

```
#!/bin/bash
# Run this script as root
set -e

# Limit of open files per process
echo "* hard nofile 65536" >> /etc/security/limits.conf
echo "* soft nofile 65536" >> /etc/security/limits.conf

# Limit of distinct mappings a process can have
echo "vm.max_map_count=250000" >> /etc/sysctl.conf
```

update-python3

```
#!/bin/bash
# Run this script as root
set -e

# Install python3.8
add-apt-repository ppa:deadsnakes/ppa -y
apt-get update
apt-get install python3.8-dev -y

# Change defautl python3
rm /usr/bin/python3
ln -ns /usr/bin/python3.8 /usr/bin/python3

# Update pip3 and setuptools
apt-get remove python3-pip -y
apt install python3.8-distutils -y
cd ~/Downloads
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python3.8 get-pip.py
apt-get remove python3-setuptools -y
pip3 install setuptools --upgrade
```

install-dependencies

```
#!/bin/bash
# Run this script as root
set -e

# Dependencies for ns3 - dce
apt-get update
apt-get install git vim curl openssh-server htop build-essential libpq-dev
    libssl-dev openssl libffi-dev zlib1g-dev mercurial cmake qt5-default
   libc6-dev libc6-dev-i386 libpcap-dev netanim libgraphviz-dev graphviz
   libgraphviz-dev pkg-config python-pip python3-pip python3-gi python3-gi
   -cairo python3 python3-dev python3-setuptools python-gi python-gi-cairo
    python3-pygraphviz python-pygraphviz ipython3 python3-sphinx python-
   virtualenv libdb-dev flex bison expect python3.8-distutils gawk indent
   libsysfs-dev -y
pip3 install distro
```

download-bake

```
#!/bin/bash

cd ~
```

```
4 git clone https://gitlab.com/nsnam/bake.git
```

download-dce

```
1  #!/bin/bash
2
3  cd ~/bake
4  export PATH=$PATH:`pwd`/build/bin:`pwd`/build/bin_dce
5  export PYTHONPATH=$PYTHONPATH:`pwd`/build/lib
6  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`/build/lib
7  python3 bake.py configure -e dce-linux-dev
8  python3 bake.py check
9  python3 bake.py show
10 python3 bake.py download
11
12 # Fix ns-3 commit to use stable code
13 cd ~/bake/source/ns-3-dev/
14 git checkout 8d09314e86c70a5b282befee8c6f516aff5e417a
15
16 # Download our DCE fork
17 cd ~/bake/source
18 rm -rf ns-3-dce
19 git clone https://gitlab.fing.edu.uy/proyecto-2021/ns-3-dce.git
20
21 # Patch elf-loader
22 cd ~/bake/source/elf-loader
23 curl -L -o elf-loader-ns3-dce-ubuntu-16-4.patch https://gist.
      githubusercontent.com/bullekeup/727cf9cce293f035569acbdc60e1803e/raw/
      f2862da25c819c247f5dd450754133d3fc76723e/elf-loader-ns3-dce-ubuntu
      -16-4.patch
24 patch < elf-loader-ns3-dce-ubuntu-16-4.patch
```

download-dce-quagga

```
1  #!/bin/bash
2
3  cd ~/bake
4  export PATH=$PATH:`pwd`/build/bin:`pwd`/build/bin_dce
5  export PYTHONPATH=$PYTHONPATH:`pwd`/build/lib
6  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`/build/lib
7  python3 bake.py configure -e dce-linux-dev -e dce-quagga-dev
8  python3 bake.py check
9  python3 bake.py show
10 python3 bake.py download
```

build-dce

```
1 #!/bin/bash
2
3 cd ~/bake
4 export PATH=$PATH:`pwd`/build/bin:`pwd`/build/bin_dce
5 export PYTHONPATH=$PYTHONPATH:`pwd`/build/lib
6 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`/build/lib
7 unbuffer python3 bake.py build -vvv 2>&1 | tee ~/build-$(date '+%F_%H:%M')
    .txt
```

8.2. Resultados de casos de prueba

Caso de prueba	Número de nodos	Tiempo de ejecución sin zebra	PDUs `ns-3` sin zebra	Tiempo de ejecución con zebra	PDUs `ns-3` con zebra	PDUs Kathará
2_2_1_link-failure-leaf-spine	20	44.23s	44	1m34.81s	44	-
2_2_1_link-failure-spine-tof	20	44.95s	45	1m28.90s	45	-
2_2_1_link-recovery-leaf-spine	20	48.82s	69	1m50.76s	69	-
2_2_1_link-recovery-spine-tof	20	45.97s	72	1m45.56s	72	-
2_2_1_node-failure-leaf	20	46.62s	60	1m33.48s	60	60
2_2_1_node-failure-spine	20	50.79s	56	1m34.34s	56	-
2_2_1_node-failure-tof	20	46.70s	72	1m33.65s	72	-
2_2_1_node-recovery-leaf	20	50.76s	96	1m34.14s	96	-
2_2_1_node-recovery-spine	20	54.21s	160	1m40.14s	160	-
2_2_1_node-recovery-tof	20	50.82s	168	1m38.39s	168	-
2_2_2_link-failure-leaf-spine	10	24.68s	16	47.64s	16	-
2_2_2_link-failure-spine-tof	10	22.90s	12	47.80s	12	-
2_2_2_link-recovery-leaf-spine	10	24.96s	29	56.91s	29	-
2_2_2_link-recovery-spine-tof	10	24.95s	26	55.48s	26	-
2_2_2_node-failure-leaf	10	25.22s	28	48.84s	28	28
2_2_2_node-failure-spine	10	25.80s	18	51.28s	18	-
2_2_2_node-failure-tof	10	26.56s	24	50.18s	24	-
2_2_2_node-recovery-leaf	10	28.64s	48	48.99s	48	-
2_2_2_node-recovery-spine	10	30.07s	68	55.52s	68	-
2_2_2_node-recovery-tof	10	30.19s	72	50.50s	72	-
4_4_1_link-failure-leaf-spine	80	4m15.98s	312	7m56.74s	312	-
4_4_1_link-failure-spine-tof	80	4m06.16s	427	7m38.23s	427	-
4_4_1_link-recovery-leaf-spine	80	4m25.69s	409	9m25.13s	409	-
4_4_1_link-recovery-spine-tof	80	4m10.76s	542	9m00.16s	542	-
4_4_1_node-failure-leaf	80	4m20.82s	504	8m11.00s	504	504
4_4_1_node-failure-spine	80	4m14.50s	904	7m56.27s	904	-
4_4_1_node-failure-tof	80	4m09.03s	1568	7m51.87s	1568	-
4_4_1_node-recovery-leaf	80	4m39.95s	768	8m19.27s	768	-
4_4_1_node-recovery-spine	80	4m36.30s	1808	8m18.78s	1808	-
4_4_1_node-recovery-tof	80	4m23.41s	2320	7m45.67s	2320	-
4_4_2_link-failure-leaf-spine	40	2m02.71s	96	3m36.06s	96	-
4_4_2_link-failure-spine-tof	40	2m00.35s	112	3m36.94s	112	-
4_4_2_link-recovery-leaf-spine	40	2m05.39s	145	4m09.21s	145	-
4_4_2_link-recovery-spine-tof	40	2m02.69s	162	4m08.26s	162	-
4_4_2_node-failure-leaf	40	2m08.18s	248	3m52.32s	248	248
4_4_2_node-failure-spine	40	2m05.47s	292	3m44.67s	292	-
4_4_2_node-failure-tof	40	2m04.07s	672	3m45.93s	672	-
4_4_2_node-recovery-leaf	40	2m18.31s	384	3m50.94s	384	-
4_4_2_node-recovery-spine	40	2m09.81s	640	3m43.03s	640	-
4_4_2_node-recovery-tof	40	2m09.73s	1040	3m57.96s	1040	-

Caso de prueba	Número de nodos	Tiempo de ejecución sin zebra	PDUs `ns-3` sin zebra	Tiempo de ejecución con zebra	PDUs `ns-3` con zebra	PDUs Kathará
4_4_4_link-failure-leaf-spine	20	53.09s	72	1m39.88s	72	-
4_4_4_link-failure-spine-tof	20	53.04s	56	1m37.98s	56	-
4_4_4_link-recovery-leaf-spine	20	5605s	97	1m53.58s	97	-
4_4_4_link-recovery-spine-tof	20	53.57s	78	1m52.30s	78	-
4_4_4_node-failure-leaf	20	56.22s	120	1m46.85s	120	120
4_4_4_node-failure-spine	20	57.73s	196	1m47.25s	196	-
4_4_4_node-failure-tof	20	55.19s	224	1m47.55s	224	-
4_4_4_node-recovery-leaf	20	1m01.36s	192	1m48.51s	192	-
4_4_4_node-recovery-spine	20	1m06.83	384	1m51.57s	384	-
4_4_4_node-recovery-tof	20	1m04.22s	400	1m52.50s	400	-
6_6_1_link-failure-leaf-spine	180	10m46.19s	996	19m57.98s	996	996
6_6_1_link-failure-spine-tof	180	10m37.20s	1529	19m54.56s	1529	1529, 1577
6_6_1_link-recovery-leaf-spine	180	11m21.15s	1213	20m42.85s	1213	1069, 1238
6_6_1_link-recovery-spine-tof	180	10m55.64s	1796	20m04.66s	1730	1657, 1796
6_6_1_node-failure-leaf	180	11m09.37s	1716	20m13.76s	1716	1716, 1734
6_6_1_node-failure-spine	180	10m48.70s	4704	19m59.83s	4704	4704, 4726
6_6_1_node-failure-tof	180	10m38.58s	8712	19m31.38s	8712	8712, 8808
6_6_1_node-recovery-leaf	180	12m04.08s	2592	21m42.40s	2592	2592, 3444
6_6_1_node-recovery-spine	180	11m25.12s	7872	20m44.13s	7872	8686, 9362
6_6_1_node-recovery-tof	180	10m53.93s	11256	19m12.33s	11256	11190, 11316
6_6_2_link-failure-leaf-spine	90	5m12.42s	288	9m08.50s	288	288, 358
6_6_2_link-failure-spine-tof	90	5m16.70s	396	9m18.83s	396	396, 456
6_6_2_link-recovery-leaf-spine	90	5m16.91s	397	10m50.93s	397	325, 397
6_6_2_link-recovery-spine-tof	90	5m13.41s	500	10m47.11s	476	439, 506
6_6_2_node-failure-leaf	90	5m25.74s	852	9m58.10s	852	852, 858
6_6_2_node-failure-spine	90	5m20.58s	1446	9m43.56s	1446	1446
6_6_2_node-failure-tof	90	5m10.79s	3960	9m25.19s	3960	3960
6_6_2_node-recovery-leaf	90	5m59.04s	1296	10m09.28s	1296	1296, 1926
6_6_2_node-recovery-spine	90	5m34.60s	2580	9m58.72s	2580	2916, 3247
6_6_2_node-recovery-tof	90	5m42.00s	5208	10m02.72s	5208	5208, 5310
6_6_3_link-failure-leaf-spine	60	3m18.67s	228	6m13.70s	228	228
6_6_3_link-failure-spine-tof	60	3m14.53s	264	6m19.88s	264	264
6_6_3_link-recovery-leaf-spine	60	3m24.92s	301	7m10.39s	301	253, 301
6_6_3_link-recovery-spine-tof	60	3m22.91s	332	7m10.26s	320	295, 338
6_6_3_node-failure-leaf	60	3m32.41s	564	6m44.79s	564	562, 564
6_6_3_node-failure-spine	60	3m23.68s	1086	6m13.04s	1086	1086
6_6_3_node-failure-tof	60	3m20.67s	2376	6m22.33s	2376	2376, 2942
6_6_3_node-recovery-leaf	60	3m39.39s	864	6m42.36s	864	864, 1002
6_6_3_node-recovery-spine	60	3m34.06s	1860	6m18.54s	1860	1836, 2413
6_6_3_node-recovery-tof	60	3m33.75s	3192	6m45.90s	3192	3354, 3450
6_6_6_link-failure-leaf-spine	30	1m25.47s	168	2m31.79s	168	168, 541
6_6_6_link-failure-spine-tof	30	1m26.67s	132	2m26.76s	132	132
6_6_6_link-recovery-leaf-spine	30	1m30.17s	205	3m01.47s	205	181, 205
6_6_6_link-recovery-spine-tof	30	1m25.80s	164	2m53.77s	164	151, 170
6_6_6_node-failure-leaf	30	1m33.04s	276	2m42.43s	276	276
6_6_6_node-failure-spine	30	1m31.11s	726	2m41.95s	726	726
6_6_6_node-failure-tof	30	1m32.45s	792	2m48.15s	792	792
6_6_6_node-recovery-leaf	30	1m41.34s	432	2m53.86s	432	432
6_6_6_node-recovery-spine	30	1m42.44s	1140	2m53.61s	1140	1104, 1182
6_6_6_node-recovery-tof	30	1m41.56s	1176	3m00.93s	1176	1176

Caso de prueba	Número de nodos	Tiempo de ejecución sin zebra	PDUs `ns-3` sin zebra	Tiempo de ejecución con zebra	PDUs `ns-3` con zebra	PDUs Kathará
8_8_1_link-failure-leaf-spine	320	21m42.99s	2288	38m53.64s	2288	2288
8_8_1_link-failure-spine-tof	320	21m09.46s	3735	38m53.40s	3735	3743
8_8_1_link-recovery-leaf-spine	320	23m13.51s	2673	48m10.13s	2673	2417
8_8_1_link-recovery-spine-tof	320	21m57.73s	4218	45m23.74s	4098	4218
8_8_1_node-failure-leaf	320	22m30.09s	4080	40m42.60s	4080	4076, 4080, 4088
8_8_1_node-failure-spine	320	22m08.93s	15152	39m13.09s	15152	15152, 15197
8_8_1_node-failure-tof	320	21m30.22s	28800	58m00.13s	28800	28958, 29058
8_8_1_node-recovery-leaf	320	24m16.70s	6144	39m42.12s	6144	8172, 10228
8_8_1_node-recovery-spine	320	23m24.80s	22816	43m23.44s	22816	22848
8_8_1_node-recovery-tof	320	22m28.49s	34848	55m26.64s	34848	34622, 35076
8_8_2_link-failure-leaf-spine	160	10m22.76s	640	18m32.54s	640	640
8_8_2_link-failure-spine-tof	160	10m27.58s	960	19m02.41s	960	960
8_8_2_link-recovery-leaf-spine	160	10m45.17s	833	19m11.29s	833	705, 855
8_8_2_link-recovery-spine-tof	160	10m35.79s	1146	19m05.76s	1098	1033, 1154
8_8_2_node-failure-leaf	160	11m01.99s	2032	19m38.43s	2032	2028, 2032
8_8_2_node-failure-spine	160	10m28.81s	4488	19m10.55s	4488	4488, 4510
8_8_2_node-failure-tof	160	10m38.06s	13440	19m16.64s	13440	13440, 13680
8_8_2_node-recovery-leaf	160	12m10.75s	3072	20m45.33s	3072	4584
8_8_2_node-recovery-spine	160	10m50.02s	7136	19m20.54s	7136	7104, 10439
8_8_2_node-recovery-tof	160	11m05.84s	16416	19m43.11s	16416	16599, 16832
8_8_4_link-failure-leaf-spine	80	5m00.48s	416	9m00.14s	416	416, 447
8_8_4_link-failure-spine-tof	80	4m57.10s	480	9m05.06s	480	480
8_8_4_link-recovery-leaf-spine	80	4m14.44s	513	10m55.01s	513	449, 513
8_8_4_link-recovery-spine-tof	80	5m07.28s	570	10m48.89s	554	521, 578
8_8_4_node-failure-leaf	80	5m29.00s	1008	9m33.45s	1008	1008, 1024
8_8_4_node-failure-spine	80	5m14.87s	2696	9m29.02s	2696	2696
8_8_4_node-failure-tof	80	5m08.02s	5760	9m38.15s	5760	5760
8_8_4_node-recovery-leaf	80	5m55.11s	1536	10m02.02s	1536	1784, 2278
8_8_4_node-recovery-spine	80	5m26.83s	4064	9m36.43s	4064	4400, 4802
8_8_4_node-recovery-tof	80	5m34.86s	7200	9m49.72s	7200	7200
8_8_8_link-failure-leaf-spine	40	2m16.02s	304	4m24.74s	304	304
8_8_8_link-failure-spine-tof	40	2m09.70s	240	4m17.70s	240	240
8_8_8_link-recovery-leaf-spine	40	2m24.72s	353	5m08.99s	353	321, 353
8_8_8_link-recovery-spine-tof	40	2m18.88s	282	5m04.52s	282	265, 290
8_8_8_node-failure-leaf	40	2m26.75s	496	4m35.22s	496	496
8_8_8_node-failure-spine	40	2m19.87s	1800	4m44.34s	1800	1800
8_8_8_node-failure-tof	40	2m23.88s	1920	4m38.25s	1920	1920
8_8_8_node-recovery-leaf	40	2m43.20s	768	4m39.58s	768	768, 888
8_8_8_node-recovery-spine	40	2m35.04s	2528	4m43.29s	2528	2613, 2856
8_8_8_node-recovery-tof	40	2m33.62s	2592	4m40.12s	2592	2592, 2960
10_10_1_link-failure-leaf-spine	500	38m49.47s	4380	Sin memoria		-
10_10_1_link-failure-spine-tof	500	37m04.71s	7429	Sin memoria		-
10_10_1_link-recovery-leaf-spine	500	39m41.84s	4981	Sin memoria		-
10_10_1_link-recovery-spine-tof	500	38m17.07s	8192	Sin memoria		-
10_10_1_node-failure-leaf	500	39m04.37s	7980	Sin memoria		-
10_10_1_node-failure-spine	500	38m24.18s	37480	Sin memoria		18390
10_10_1_node-failure-tof	500	37m47.82s	72200	Sin memoria		36072
10_10_1_node-recovery-leaf	500	42m29.05s	12000	Sin memoria		-
10_10_1_node-recovery-spine	500	40m12.98s	52640	Sin memoria		-
10_10_1_node-recovery-tof	500	40m04.27s	84040	Sin memoria		-

Caso de prueba	Número de nodos	Tiempo de ejecución sin zebra	PDUs `ns-3` sin zebra	Tiempo de ejecución con zebra	PDUs `ns-3` con zebra	PDUs Kathará
10_10_2_link-failure-leaf-spine	250	17m40.56s	1200	32m48.21s	1200	1200
10_10_2_link-failure-spine-tof	250	17m44.08s	1900	32m54.79s	1900	1900
10_10_2_link-recovery-leaf-spine	250	18m00.51s	1501	34m44.02s	1501	1301, 1501
10_10_2_link-recovery-spine-tof	250	17m46.80s	2192	34m28.32s	2112	2011, 2202
10_10_2_node-failure-leaf	250	18m42.95s	3980	34m48.54s	3980	3980, 3996
10_10_2_node-failure-spine	250	17m30.75s	10810	33m54.21s	10810	10810, 10833
10_10_2_node-failure-tof	250	17m35.49s	34200	33m42.54s	34200	34527, 34788
10_10_2_node-recovery-leaf	250	20m46.06s	6000	35m49.45s	6000	6000, 7976
10_10_2_node-recovery-spine	250	18m30.37s	15940	44m37.44s	15940	15661, 19401
10_10_2_node-recovery-tof	250	18m26.49s	40040	34m00.97s	40040	41188, 41667
10_10_5_link-failure-leaf-spine	100	6m32.38s	660	11m00.54s	660	660
10_10_5_link-failure-spine-tof	100	6m26.11s	760	10m54.41s	760	760, 820
10_10_5_link-recovery-leaf-spine	100	6m50.84s	781	12m04.48s	781	701, 781
10_10_5_link-recovery-spine-tof	100	6m34.77s	872	11m51.21s	852	811, 882
10_10_5_node-failure-leaf	100	6m55.75s	1580	12m32.87s	1580	1580
10_10_5_node-failure-spine	100	6m31.14s	5410	11m34.54s	5410	5410
10_10_5_node-failure-tof	100	6m47.49s	11400	11m54.00s	11400	11544, 11550
10_10_5_node-recovery-leaf	100	7m48.76s	2400	13m12.67s	2400	3180, 2790
10_10_5_node-recovery-spine	100	6m57.20s	7540	12m49.14s	7540	7510, 7490
10_10_5_node-recovery-tof	100	7m14.20s	13640	13m09.53s	13640	14798, 15178
10_10_10_link-failure-leaf-spine	50	2m59.37s	480	5m22.45s	480	480
10_10_10_link-failure-spine-tof	50	2m59.86s	380	5m28.12s	380	380
10_10_10_link-recovery-leaf-spine	50	3m08.51s	541	6m08.82s	541	501, 541
10_10_10_link-recovery-spine-tof	50	3m02.83s	432	6m02.63s	432	411, 442
10_10_10_node-failure-leaf	50	3m07.27s	780	5m41.16s	780	780
10_10_10_node-failure-spine	50	3m05.43s	3610	5m48.46s	3610	3610
10_10_10_node-failure-tof	50	3m05.12s	3800	5m49.68s	3800	3800
10_10_10_node-recovery-leaf	50	3m32.97s	1200	6m01.74s	1200	1580, 1582
10_10_10_node-recovery-spine	50	3m19.45s	4740	5m57.94s	4740	4670, 4710
10_10_10_node-recovery-tof	50	3m29.01s	4840	5m59.03s	4840	4840, 5540
12_12_1_link-failure-leaf-spine	720	1h44m55s	7464	Sin memoria		-
12_12_1_link-failure-spine-tof	720	1h55m54s	12995	Sin memoria		-
12_12_1_link-recovery-leaf-spine	720	2h16m37s	8329	Sin memoria		-
12_12_1_link-recovery-spine-tof	720	2h14m49s	14102	Sin memoria		-
12_12_1_node-failure-leaf	720	2h11m32s	17244	Sin memoria		-
12_12_1_node-failure-spine	720	2h01m16s	78456	Sin memoria		-
12_12_1_node-failure-tof	720	2h16m34s	152352	Sin memoria		-
12_12_1_node-recovery-leaf	720	2h15m40s	20736	Sin memoria		-
12_12_1_node-recovery-spine	720	2h06m35s	104880	Sin memoria		-
12_12_1_node-recovery-tof	720	2h02m28s	172848	Sin memoria		-
12_12_3_link-failure-leaf-spine	240	26m33.30s	1488	45m23.74s	1488	1488
12_12_3_link-failure-spine-tof	240	22m29.29s	2208	40m39.63s	2208	2208
12_12_3_link-recovery-leaf-spine	240	22m22.99s	1777	48m54.45s	1777	1585, 1777
12_12_3_link-recovery-spine-tof	240	22m30.69s	2486	45m46.89s	2414	2317, 2498
12_12_3_node-failure-leaf	240	23m00.82s	5724	46m12.83s	4584	4580, 4584, 4628
12_12_3_node-failure-spine	240	22m17.87s	15852	45m42.76s	15852	15929, 16002
12_12_3_node-failure-tof	240	21m58.50s	46368	40m59.92s	46368	47227, 57292
12_12_3_node-recovery-leaf	240	25m03.87s	6912	49m06.91s	6912	8052, 11514
12_12_3_node-recovery-spine	240	23m06.23s	21792	43m50.62s	21792	21797, 23710, 29768
12_12_3_node-recovery-tof	240	23m13.26s	53040	40m49.30s	53040	53926, 54024

Caso de prueba	Número de nodos	Tiempo de ejecución sin zebra	PDUs `ns-3` sin zebra	Tiempo de ejecución con zebra	PDUs `ns-3` con zebra	PDUs Kathará
12_12_4_link-failure-leaf-spine	180	15m45.54s	1224	28m45.12s	1224	1224
12_12_4_link-failure-spine-tof	180	15m56.67s	1656	28m59.42s	1656	1656, 1680
12_12_4_link-recovery-leaf-spine	180	16m16.61s	1441	30m07.54s	1441	1297, 1441
12_12_4_link-recovery-spine-tof	180	16m17.35s	1862	29m42.24s	1814	1741, 1874
12_12_4_node-failure-leaf	180	16m32.19s	3852	29m54.12s	3432	3430, 3432
12_12_4_node-failure-spine	180	16m22.04s	12684	30m46.41s	12684	12763, 12775
12_12_4_node-failure-tof	180	16m01.50s	33120	29m04.38s	33120	33930, 34800
12_12_4_node-recovery-leaf	180	18m03.52s	5184	30m42.63s	5184	5184
12_12_4_node-recovery-spine	180	17m04.79s	17184	31m12.87s	17184	-
12_12_4_node-recovery-tof	180	16m49.52s	38064	29m52.92s	38064	38979, 39312
12_12_6_link-failure-leaf-spine	120	10m40.88s	960	18m54.38s	960	960, 983
12_12_6_link-failure-spine-tof	120	10m32.16s	1104	18m23.65s	1104	1104
12_12_6_link-recovery-leaf-spine	120	11m02.66s	1105	19m59.26s	1105	1009, 1105
12_12_6_link-recovery-spine-tof	120	10m50.77s	1238	18m21.71s	1214	1165, 1250
12_12_6_node-failure-leaf	120	11m08.53s	2700	19m02.25s	2280	2278, 2280, 2292
12_12_6_node-failure-spine	120	10m55.35s	9516	19m28.75s	9516	9516, 9589
12_12_6_node-failure-tof	120	11m07.92s	19872	19m41.38s	19872	20580, 20712
12_12_6_node-recovery-leaf	120	12m05.47s	3456	20m10.93s	3456	3456, 5146
12_12_6_node-recovery-spine	120	11m36.57s	12576	20m21.34s	12576	12514, 15936
12_12_6_node-recovery-tof	120	11m46.11s	23088	20m37.42s	23088	23352, 23625
12_12_12_link-failure-leaf-spine	60	4m23.20s	696	7m34.12s	696	696
12_12_12_link-failure-spine-tof	60	4m21.28s	552	7m22.85s	552	552
12_12_12_link-recovery-leaf-spine	60	4m40.15s	769	8m02.89s	769	721, 769
12_12_12_link-recovery-spine-tof	60	4m33.03s	614	7m54.73s	614	589, 626
12_12_12_node-failure-leaf	60	4m53.51s	1548	8m12.02s	1128	1128
12_12_12_node-failure-spine	60	4m36.18s	6348	8m08.54s	6348	6348
12_12_12_node-failure-tof	60	4m35.58s	6624	8m22.39s	6624	6624
12_12_12_node-recovery-leaf	60	5m11.81s	1728	8m00.45s	1728	1728
12_12_12_node-recovery-spine	60	5m10.84s	7968	7m59.02s	7968	8162, 9469
12_12_12_node-recovery-tof	60	5m07.00s	8112	7m45.12s	8112	8112, 8676
14_14_1_link-failure-leaf-spine	980	3h48m26s	11732	Sin memoria		-
14_14_1_link-failure-spine-tof	980	4h07m34s	20817	Sin memoria		-
14_14_1_link-recovery-leaf-spine	980	5h16m00s	12909	Sin memoria		-
14_14_1_link-recovery-spine-tof	980	4h53m55s	22332	Sin memoria		-
14_14_1_node-failure-leaf	980	4h50m58s	27398	Sin memoria		-
14_14_1_node-failure-spine	980	4h49m53s	146384	Sin memoria		-
14_14_1_node-failure-tof	980	5h00m02s	285768	Sin memoria		-
14_14_1_node-recovery-leaf	980	4h45m53s	32928	Sin memoria		-
14_14_1_node-recovery-spine	980	4h58m10s	188608	Sin memoria		-
14_14_1_node-recovery-tof	980	4h58m17s	318360	Sin memoria		-
14_14_2_link-failure-leaf-spine	490	1h13m18s	3136	Sin memoria		-
14_14_2_link-failure-spine-tof	490	1h08m05s	5292	Sin memoria		-
14_14_2_link-recovery-leaf-spine	490	1h09m45s	3725	Sin memoria		-
14_14_2_link-recovery-spine-tof	490	1h09m51s	5882	Sin memoria		-
14_14_2_node-failure-leaf	490	1h11m07s	14070	Sin memoria		-
14_14_2_node-failure-spine	490	1h08m54s	40782	Sin memoria		-
14_14_2_node-failure-tof	490	1h09m27s	137592	Sin memoria		-
14_14_2_node-recovery-leaf	490	1h15m02s	16464	Sin memoria		-
14_14_2_node-recovery-spine	490	1h10m49s	54740	Sin memoria		-
14_14_2_node-recovery-tof	490	1h13m18s	153720	Sin memoria		-

Caso de prueba	Número de nodos	Tiempo de ejecución sin zebra	PDUs `ns-3` sin zebra	Tiempo de ejecución con zebra	PDUs `ns-3` con zebra	PDUs Kathará
14_14_7_link-failure-leaf-spine	140	14m41.96s	1316	26m36.23s	1316	1336
14_14_7_link-failure-spine-tof	140	14m37.25s	1512	26m23.38s	1512	1512, 1635
14_14_7_link-recovery-leaf-spine	140	15m15.71s	1485	26m58.23s	1485	1373, 1485
14_14_7_link-recovery-spine-tof	140	15m09.47s	1668	26m50.92s	1640	1583, 1682
14_14_7_node-failure-leaf	140	15m34.54s	3878	27m34.72s	3108	3108, 3150
14_14_7_node-failure-spine	140	15m08.11s	15302	26m53.14s	15302	15374
14_14_7_node-failure-tof	140	14m47.13s	31752	26m23.54s	31752	32988, 32990
14_14_7_node-recovery-leaf	140	16m45.26s	4704	27m15.82s	4704	5470, 7014
14_14_7_node-recovery-spine	140	16m02.32s	19460	27m18.14s	19460	21484, 24662
14_14_7_node-recovery-tof	140	15m44.28s	36120	27m46.19s	36120	37967, 39645
14_14_14_link-failure-leaf-spine	70	6m29.86s	952	10m34.12s	952	952
14_14_14_link-failure-spine-tof	70	6m32.31s	756	10m54.74s	756	756
14_14_14_link-recovery-leaf-spine	70	6m54.24s	1037	11m28.35s	1037	981, 1037
14_14_14_link-recovery-spine-tof	70	6m46.29s	828	10m41.73s	828	799, 842
14_14_14_node-failure-leaf	70	6m58.19s	2114	11m18.52s	1540	1540, 1552, 1554
14_14_14_node-failure-spine	70	6m50.53s	10206	10m58.12s	10206	10206, 10308, 10350
14_14_14_node-failure-tof	70	6m56.14s	10584	11m09.79s	10584	10570, 10584
14_14_14_node-recovery-leaf	70	7m43.53s	2352	12m48.42s	2352	2730
14_14_14_node-recovery-spine	70	7m29.55s	12404	11m24.18s	12404	12580, 13024
14_14_14_node-recovery-tof	70	7m41.30s	12600	11m39.76s	12600	14406, 14574
16_16_1_link-failure-leaf-spine	1280	Sin memoria		Sin memoria		-
16_16_1_link-failure-spine-tof	1280	Sin memoria		Sin memoria		-
16_16_1_link-recovery-leaf-spine	1280	Sin memoria		Sin memoria		-
16_16_1_link-recovery-spine-tof	1280	Sin memoria		Sin memoria		-
16_16_1_node-failure-leaf	1280	Sin memoria		Sin memoria		-
16_16_1_node-failure-spine	1280	Sin memoria		Sin memoria		-
16_16_1_node-failure-tof	1280	Sin memoria		Sin memoria		-
16_16_1_node-recovery-leaf	1280	Sin memoria		Sin memoria		-
16_16_1_node-recovery-spine	1280	Sin memoria		Sin memoria		-
16_16_1_node-recovery-tof	1280	Sin memoria		Sin memoria		-
16_16_8_link-failure-leaf-spine	160	21m56.72s	1728	41m54.69s	1728	1728
16_16_8_link-failure-spine-tof	160	21m27.73s	1984	41m55.50s	1984	1984
16_16_8_link-recovery-leaf-spine	160	22m54.44s	1921	42m34.61s	1921	1793, 1991
16_16_8_link-recovery-spine-tof	160	22m28.15s	2162	42m52.36s	2130	2065, 2178
16_16_8_node-failure-leaf	160	22m49.06s	5584	42m28.83s	4064	4062, 4064
16_16_8_node-failure-spine	160	22m26.64s	23056	42m25.36s	23056	23262
16_16_8_node-failure-tof	160	22m15.73s	47616	42m52.26s	47616	48543, 48807
16_16_8_node-recovery-leaf	160	24m50.96s	6144	44m15.59s	6144	8230, 10174
16_16_8_node-recovery-spine	160	23m39.33s	28480	43m56.16s	28480	29729, 30013
16_16_8_node-recovery-tof	160	23m27.73s	53312	43m52.23s	53312	54467, 56962
16_16_16_link-failure-leaf-spine	80	9m18.42s	1248	17m42.73s	1248	1248, 1310
16_16_16_link-failure-spine-tof	80	9m04.46s	992	17m29.28s	992	992
16_16_16_link-recovery-leaf-spine	80	9m47.55s	1345	17m53.91s	1345	1281, 1345
16_16_16_link-recovery-spine-tof	80	9m40.63s	1074	17m49.14s	1074	1041, 1090
16_16_16_node-failure-leaf	80	10m14.61s	2768	18m31.46s	2016	2016
16_16_16_node-failure-spine	80	9m35.26s	15376	17m39.17s	15376	15582, 15596, 15599
16_16_16_node-failure-tof	80	9m24.25s	15872	17m14.04s	15872	15376, 15840, 15872
16_16_16_node-recovery-leaf	80	10m40.29s	3072	18m39.23s	3072	3070, 4064
16_16_16_node-recovery-spine	80	10m23.09s	18240	18m41.98s	18240	18016, 19074
16_16_16_node-recovery-tof	80	10m19.44s	18496	18m43.41s	18496	19328, 19776