UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Ambiente para simulación de tráfico realista en redes con topologías Fat-Tree

Informe de Proyecto de Grado presentado por

Sebastián Cabrera

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Leonardo Alberro
Eduardo Grampín

Montevideo, 22 de abril de 2023

Agradecimientos

A todos los que de una forma u otra me han acompañado durante este proceso turbulento, que muchas veces pareció interminable, estando a mi lado e impulsándome a seguir hasta alcanzar esta instancia.

No quiero dejar pasar la oportunidad de agradecer a aquellos grandes docentes que la facultad ha puesto en mi camino. En esta última etapa, tuve la suerte de cruzarme con Leonardo y Eduardo que han sido excelentes técnica y humanamente.

Por último, agradecer a la UdelaR como institución, a la educación pública como sistema, que con sus infinitos defectos, es siempre garante de intercambio y movilidad social.

Resumen

En este trabajo se detalla sistemáticamente el proceso de desarrollo de un ambiente de simulación de tráfico realista para distintos tipos de topologías fat-tree.

Se integraron herramientas de generación de tráfico con entornos de simulación y modelado de redes, para obtener métricas que se asemejen al comportamiento real de los nodos dentro de una infraestructura del tipo data center.

Este trabajo considera que los nodos están conectados mediante una topología del tipo fat-tree. Por lo que en una primera instancia, se presenta con detalle la misma y se analizan sus características principales.

También, se abordan algunas de las problemáticas más comunes para este tipo de infraestructuras, detallando la situación en cada caso y presentando alternativas para mitigarlas.

Además, se presentan algunos trabajos que sirvieron como referencia para abordar el desarrollo del ambiente propuesto, gracias a los cuales se pudo agregar funcionalidades importantes para el análisis del comportamiento en la simulación.

El fruto del presente trabajo brinda al usuario un ambiente de simulación que le permite investigar sobre el comportamiento de redes de data centers bajo la influencia de diferentes tipos de tráfico que se asemejan a la realidad.

El entorno construido permite simular trazas de tráfico realistas de diferentes tipos de data centers (empresariales, académicos, cloud y redes sociales) con redes de topología fat-tree de tamaños arbitrarios. A su vez, dicho ambiente contará con la posibilidad de de parametrizar los aspectos principales de la red, por nombrar alguno de ellos: cantidad de PoDs (Points of Delivery) del fat-tree, ancho de banda de los enlaces, tráfico a generar por parte de las aplicaciones de los servidores, tiempos de inicio de los flujos, entre otros.

Palabras clave: Data center, OMNeT++, Simulación, Fat-tree, Tráfico, Transporte

Índice general

Capítulo 1

Introducción

Múltiples aspectos de nuestra sociedad se encuentran basados en las infraestructuras de las redes de telecomunicaciones actuales. Esto es posible dado el grado de avance que las mismas poseen.

Estas infraestructuras, conocidas como DCN, pueden resumirse como instalaciones donde conviven decenas de miles de nodos de cómputo y comunicaciones, y en los cuales se supera la complejidad de la internet tradicional.

Este tipo de instalaciones necesitan contar con fuentes de energía constantes y estables para su operativa, así como para brindar la refrigeración necesaria para mantener los nodos dentro de los rangos de temperaturas de trabajo. Dada la complejidad de este contexto, no es posible contar con infraestructuras físicas para el desarrollo y la experimentación de nuevas soluciones, por los costes asociados a la instalación y mantenimiento de las mismas. Por lo tanto, los simuladores y emuladores se vuelven imprescindibles para dar soporte a este tipo de redes.

La dependencia de estas infraestructuras plantea desafíos crecientes dadas sus características particulares. Para dar soporte a esta realidad han surgido desarrollos enfocados en el enrutamiento escalable, como por ejemplo: Routing in Fat Trees (RIFT) (A. Przygienda, 2021) y Link State Vector Routing (LSVR) (*LSVR Working Group*, s.f.).

El objetivo general de este trabajo es la creación de un ambiente que permita simular redes de datos con sus nodos interconectados mediante una topología fat-tree (en particular data centers) utilizando diferentes tipos de trazas de tráfico realistas. En este tipo de arquitectura de red los nodos de cómputo se dividen dentro de diferentes PoDs que luego son interconectados a través de tres capas de switches: edge, aggregation y core switches (desde la capa más cercana a los hosts hacia la más lejana).

La herramienta de simulación utilizada es el simulador de eventos discretos OMNeT++ (OMNeT++, 2022), el cual es un software extensible, basado en módulos implementados en C++, para la simulación de redes de diversos tipos. A su vez, ofrece un Integrated Development Environment (IDE) basado

en Eclipse [1] para el desarrollo de las simulaciones por parte de los usuarios y múltiples ambientes de ejecución (gráficos y por línea de comandos).

Dado que OMNeT++ cuenta con un motor de simulación de redes a un nivel más "general" es necesario integrarlo con un framework especifico como INET (*INET Framework*, s.f.) para la simulación de redes de datos. El cual es una librería open-source que provee soporte para los principales protocolos, agentes y demás componentes necesarios para el diseño de las simulaciones de redes de datos, todo esto lo hace particularmente útil para el diseño y la validación de nuevos escenarios dentro del ambiente de las DCN.

INET brinda soporte a nivel de módulos para los principales componentes de todo el stack de internet (Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Internet Protocol (IP) versión 4, IP versión 6, Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), etc.), así como también para los protocolos a nivel de capa de enlace (Ethernet, Point to Point Protocol (PPP), 802.11, etc.). Este framework está construido basado en módulos que intercambian información a través del pasaje de mensajes, es posible combinar múltiples módulos para generar elementos más complejos como: servidores, routers, switches, etc. Además, es posible realizar personalizaciones sobre cualquiera de ellos o generar nuevos.

En conjunto con las herramientas de simulación anteriores, es clave para el correcto abordaje de este escenario, trabajar sobre trazas de tráfico realistas para comprender de mejor manera lo que ocurre dentro del data center, así como las problemáticas que se presentan durante los distintos escenarios operativos de estas infraestructuras.

Para generar estas trazas de tráfico realista se integró TrafPy (*Overview of TrafPy*, s.f.) a la arquitectura del ambiente de simulación construido. Con esta herramienta se logra replicar tráfico sin contar con acceso a las fuentes de datos, que en múltiples ocasiones no están disponibilizadas por parte de los propietarios de las DCN.

El ambiente construido como resultado de este trabajo es parametrizable en la cantidad de servidores, así como también en el ancho de banda de los enlaces y demás características típicas de estas infraestructuras. Se busca que el ambiente resultante de este trabajo presente la mayor adaptabilidad y extensibilidad posible, permitiendo agregar múltiples guiones de simulación para diversos escenarios en el futuro. Así como también diseñar nuevas instancias de redes a evaluar.

Este documento se encuentra organizado de la siguiente manera: un primer capítulo encargado de presentar el trabajo desarrollado por otros autores sobre el abordaje y mitigación de las problemáticas encontradas en el tráfico dentro del data center, la presentación de la topología utilizada en el modelado de los data center considerados, además de desarrollar las herramientas utilizadas para la construcción del ambiente de simulación.

Luego, un capítulo enfocado en el ambiente construido, presentando el diseño y la arquitectura del mismo, las métricas a tomar en cuenta y una guía de

[1] `https://www.eclipse.org/`

uso sobre el escenario de simulación. Le sigue un capítulo dedicado a la presentación de las ejecuciones sobre el ambiente y mostrando el comportamiento del tráfico simulado. Más tarde, se presentan en un siguiente capítulo las limitantes y problemáticas detectadas en el desarrollo de este trabajo.

Por último, se agrega capítulo final a modo de resumen de las conclusiones alcanzadas y agregando consideraciones sobre posible trabajo futuro que quedó fuera del alcance del proyecto.

Capítulo 2

Estado del arte

Las redes de computadoras han ido aumentando su complejidad a lo largo del tiempo conforme avanzó el desarrollo de internet. En la actualidad soportan una gran diversidad de aplicaciones con múltiples requerimientos.

Estas DCN están formadas por un gran número de nodos terminales (servidores) y switches, distribuidos según algún tipo de arquitectura, debiendo soportar múltiples caminos de enrutamiento entre pares de servidores. Además, deben contar con un gran ancho de banda en sus enlaces y una muy baja latencia (Medhi y Ramasamy, 2018).

Las DCN brindan el soporte para los servicios principales de la internet actual, como por ejemplo los servicios de búsqueda de Google[1], los servicios cloud como Amazon[2], las redes sociales en el ambiente de Meta[3] o el streaming de video por parte de Netflix[4]

Esta realidad hace que sea de suma importancia contar con protocolos y mecanismos de transporte que garanticen un alto rendimiento para el vasto conjunto de aplicaciones y servicios que son soportados en las DCN. En particular, se ha vuelto clave conseguir altas velocidades de goodput[5] y baja latencia.

Durante el este capítulo se procederán a desarrollar conceptos básicos para el entendimiento del presente trabajo. Se desarrollarán algunas de las problemáticas más comunes en los ambientes de DCN así como también alternativas para mitigarlas. Se presentará la topología de modelado de DCN utilizada como referencia en todo el presente trabajo. Por último, se abordarán las diferentes herramientas de simulación y generación de tráfico utilizadas para la construcción del ambiente.

[1] https://www.google.com
[2] https://aws.amazon.com/ec2
[3] https://about.meta.com
[4] https://www.netflix.com
[5] Se entiende por goodput la medida a nivel de capa de aplicación del throughput, es decir, la cantidad de bits útiles que se transmiten por la red en una unidad de tiempo entre el origen y el destino. Excluyendo los agregados de cabezales y etc. por parte de los protocolos así como la retransmisión.

2.1. Tipos de tráfico en ambientes de DCN

Una parte importante de la información que se transmite a través de las DCNs es generada por aplicaciones y servicios, con la finalidad de ser más robustas (Alasmar, 2019). Por ejemplo, los sistemas de almacenamiento distribuido replican bloques de información a través de múltiples nodos del data center con la finalidad de soportar mejor eventuales fallos. Este tipo de tráfico tiene un único origen y múltiples destinos, denominado como *"one-to-many"*.

En la actualidad los protocolos de transporte para DCN no son óptimos en términos de uso de la red y los servidores para los escenarios *"one-to-many"* (Alasmar, 2019), dado que estos tipos de tráfico se implementan, por parte de los protocolos de transporte, como múltiples flujos unicast. Esto implica que copias de los mismos datos son transmitidos más de una vez a través de la red.

Este comportamiento termina degradando el funcionamiento general, ya que existen desperdicios en las capacidades de transmisión de los enlaces. En particular se ven afectados los flujos de datos pequeños que son particularmente sensibles a la latencia (Alasmar, 2019).

Existen estrictos requerimientos respecto a la baja latencia para las aplicaciones "cloud" desplegadas en las DCN (Alizadeh y cols., s.f.) (Munir y cols., s.f.), los cuales se suman a la alta utilización de las capacidades de red por parte de los servicios de fondo que a menudo se encuentran desplegados también en las DCN (Kheirkhah, Wakeman, y Parisis, s.f.) (Raiciu y cols., 2011).

Esto ha servido como motivación para desarrollar varios protocolos y mecanismos de transporte con sus ventajas y desventajas en función de los tipos de tráfico empleados y otras variables. A continuación, se presenta un breve resumen de dichos enfoques:

- Dando soporte para múltiples caminos de igual costo entre todo par de servidores (Handley y cols., 2017) (Raiciu y cols., 2010)
- Utilizando hardware específico que garantiza baja latencia (Montazeri, Li, Alizadeh, y Ousterhout, 2018)
- Eliminando los problemas de Incast y Outcast (detallados en la sección 2.3.1)
- Diferenciando los tipos de tráfico dentro del data center (flujos cortos / largos)

De cualquier manera, ninguno de estos protocolos soporta de manera eficiente el tráfico "one-to-many" ni el "many-to-one"

2.2. Topología fat-tree

La topología fat-tree para redes data center es un caso especial de red Clos (Clos, 1953) (diseño teórico de red multinivel). Se presenta por primera vez en (Leiserson, 1985) para el ambiente de las súper computadoras y luego es adaptado para el entorno de las DCN.

Número de PoDs	k
Número de core switches	$(k/2)^2$
Número de aggregation switches	$k^2/2$
Número de edge switches	$k^2/2$
Número de switches, N (todos los tipos)	$5k^2/4$
Número de links, L	$k^3/2$
Número de hosts soportados	$k^3/4$

Cuadro 2.1: Cardinalidad de los componentes de un fat-tree (Medhi y Ramasamy, 2018).

Una instancia de fat-tree es indicada típicamente a partir de su número de *PoDs* (Medhi y Ramasamy, 2018), los cuales se numeran de izquierda a derecha desde el PoD-0 al PoD-(k-1). El diseño de la topología consiste en tres capas de switches: edge, aggregation y core switches (desde la capa más cercana a los hosts hacia la más lejana).

En la figura 2.1, se puede observar un diseño completo para un fat-tree de tamaño $k = 4$ (tamaños mayores se omiten por dificultad en la visualización).

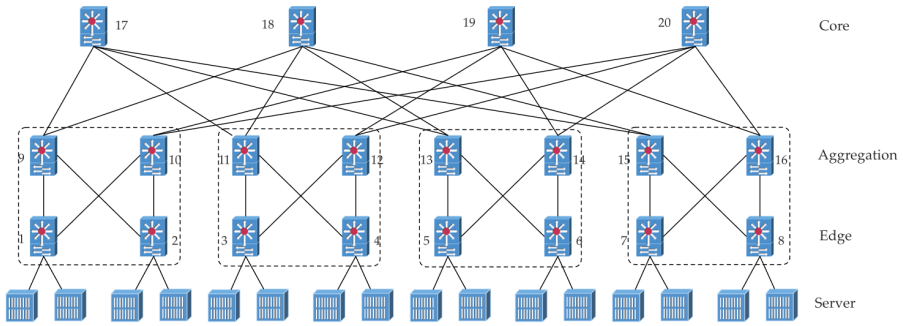

Figura 2.1: Ejemplo de 4-PoD fat-tree (Medhi y Ramasamy, 2018).

En el cuadro 2.1, se puede observar la cantidad de componentes de la topología para un fat-tree de tamaño k

La adopción de una topología del tipo fat-tree trae ciertos beneficios: normaliza los switches dentro de la red, ya que todos son del mismo tipo, con las mismas cantidades de puertos y la misma velocidad, permitiendo que los hosts tengan todos la misma velocidad de conexión y facilitando el reemplazo ante mal funcionamiento de cualquiera de los switches. Además, este diseño permite múltiples caminos entre distintos pares de hosts lo cual permite una mayor fiabilidad y mejora el balanceo de carga en la red [6].

[6]La excepción a esto se da cuando los hosts están conectados al mismo edge switch, allí el camino es único

Como contra partida, este diseño no permite ampliar la red de una forma directa (*plug and play*), sino que por el contrario, es necesario agregar enlaces en los niveles superiores (core - aggregation) si se desean introducir nuevos PoDs al data center, además de reservar espacio en las subredes IP utilizadas para el direccionamiento de los hosts.

2.3. TCP

El protocolo TCP es el estándar de facto para el transporte de datos en Internet (Alasmar, 2019). Esto se debe fundamentalmente a sus características principales, como lo son la entrega ordenada de los paquetes en los extremos de la conexión y la garantía de fiabilidad en la entrega de los datos. TCP es capaz de brindar estas funcionalidades ya que aplica corrección de errores y control de flujo a través de la conexión extremo a extremo, además de brindar soporte ante la pérdida de paquetes debido a la congestión de la red o a errores de transmisión.

Este protocolo cuenta con varias implementaciones de algoritmos de control de congestión que permiten regular al emisor de datos cuando la red se encuentra congestionada. Los más utilizados son: Tahoe (*RFC 5681 - TCP Congestion Control*, s.f.), Reno (*RFC 5681 - TCP Congestion Control*, s.f.), Vegas (Brakmo y Peterson, 1995), New Reno (*RFC 6582 - The NewReno Modification to TCP's Fast Recovery Algorithm*, s.f.)

En el año 2016 Google añadió Bottleneck Bandwidth and Round-trip propagation time (BBR), un nuevo algoritmo de control de transmisión para TCP (Cardwell, Cheng, Gunn, Yeganeh, y Jacobson, 2017), al kernel de Linux (*Linux kernel source tree*, s.f.).

El uso de dicho algoritmo en la infraestructura de Google ha representado un aumento en el throughput, un descenso en la latencia y un aumento de la calidad de servicio en general para los usuarios de estas infraestructuras, respecto a los algoritmos de control de congestión anteriores (*TCP BBR congestion control comes to GCP – your Internet just got faster — Google Cloud Blog*, s.f.).

La idea detrás de BBR está dada por el cambio de enfoque respecto a no sobre reaccionar respecto a la pérdida de paquetes en la red, evitando reducir la velocidad de envío de datos cuando esto sucede. Ya que en el contexto actual, enlaces con anchos de banda de Giga bit por segundo (Gbps) sumados a switches y routers con altas capacidades de procesamiento, esta pérdida no es sinónimo de congestión en las DCN como anteriormente sucedía.

Para introducirlo en las DCN actuales no es necesario ninguna actualización del lado de los clientes, ya que BBR está soportado por defecto en el Kernel de Linux desde la versión 4.9.

2.3.1. Limitantes de TCP en entornos de data centers

En la actualidad se considera que el protocolo TCP es inapropiado para el transporte dentro de las DCN (Handley y cols., 2017), donde se tienen fuertes

requerimientos de baja latencia y alto ancho de banda.

En las siguientes secciones se detallarán dos problemas que degradan fuertemente el rendimiento de TCP bajo este escenario: Incast (Chen, Griffith, Liu, Katz, y Joseph, 2009) y Outcast (Wieder, Bhatotia, Post, y Rodrigues, 2012).

Incast

El fenómeno de Incast implica una enorme degradación del throughput a nivel de aplicación en el contexto de un data center que utilice TCP como protocolo de transporte (Chen y cols., 2009).

Se genera cuando una gran cantidad de flujos de tráfico cortos y sincronizados envían peticiones TCP a un mismo receptor. Esto puede generar la sobrecarga de la cola de procesamiento para alguno de los switches dentro de la DCN conectado directamente al receptor (tráfico "many-to-one"), se puede ver en detalle en la figura 2.2.

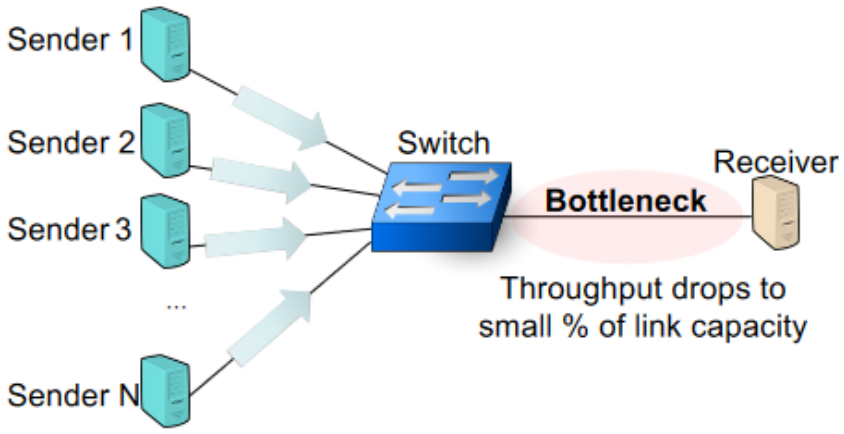

Figura 2.2: Fenómeno de Incast (Alasmar, 2019).

Bajo esta situación de alta congestión en la red, el porcentaje de pérdida de paquetes aumenta fuertemente. La retransmisión de dichas pérdidas se da luego de cumplido el Retransmit TimeOut (RTO). En general, el RTO es de una magnitud de cientos de milisegundos, mientras que el tiempo de ida y vuelta Round Trip Time (RTT) de un segmento TCP dentro de la DCN es de una magnitud bastante menor, cientos de microsegundos (Alasmar, 2019).

Por lo tanto, se cumple que el RTO $\gg$ RTT y el emisor deberá esperar al menos un RTO antes de recibir nuevos segmentos, lo cual implica dejar enlaces de la red sin uso (Chen y cols., 2009).

Esta situación degrada fuertemente la performance general de la red. En particular, impacta sobre el throughput a nivel de aplicación e implica mayores

tiempos de respuesta para los usuarios finales.

En algunos trabajos (Alizadeh y cols., 2010) (Handley y cols., 2017) se han propuesto técnicas para intentar mitigar este fenómeno. Desafortunadamente estos enfoques no soportan tráfico del tipo "one-to-many" ni "many-to-one". También existen estudios sobre el impacto de diferentes algoritmos de control de congestión de TCP en este escenario (NewReno, Selective Acknowledgments (SACK), etc.), sin reportar grandes mejoras de throughput (Chen y cols., 2009).

Outcast

Este fenómeno tiene como consecuencia un reparto inequitativo del ancho de banda disponible en los enlaces de las DCN al momento de transportar los segmentos TCP.

El mismo se presenta al momento de que un flujo de tráfico pequeño comparte un enlace congestionado de la red con otro de mayor cantidad de tráfico. El Outcast en TCP suele aparecer en situaciones de tráfico del tipo "many-to-one" donde los flujos compiten a nivel de puertos de entrada en los switches (Wieder y cols., 2012).

En particular, los data center tienden a utilizar topologías del tipo árbol. Esto implica que los switches de nivel más bajo (más cercanos a los hosts) reciben múltiples flujos por puertos independientes, pero cuentan con un único puerto de salida común, ver figura 2.3.

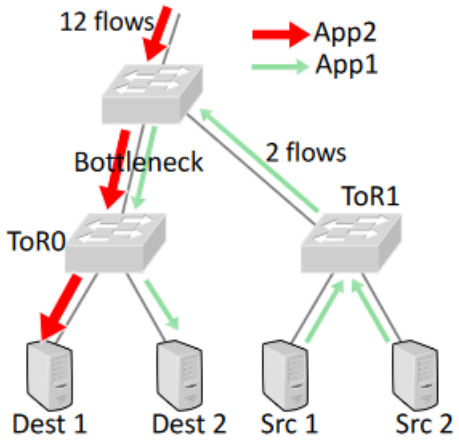

Figura 2.3: Fenómeno de Outcast (Alasmar, 2019).

En este contexto los flujos más cortos obtienen menos recursos del enlace congestionado en la red, lo cual termina degradando el throughput en comparación a aquellos flujos de tráfico de mayor duración (Wieder y cols., 2012).

El origen de esta situación, por la distribución de puertos de los switches inferiores comentada anteriormente, está en que los segmentos TCP que ingresan

a los mismos (desde múltiples orígenes, por formar parte de diferentes flujos) deben competir por el único puerto de salida asociado a cada host.

A lo anterior hay que agregar que estos switches utilizan colas de procesamiento del tipo *tail-drop*, descartando los últimos segmentos que arriban una vez que la cola está llena. Este tipo de comportamiento se conoce como apagón de puertos (port blackout). La figura 2.4 muestra un ejemplo de esta situación. El switch en este caso recibe paquetes en los puertos de entrada A y B, donde ambos flujos compiten por el puerto de salida C. En este caso todos los paquetes del puerto A son descartados debido a que la cola de procesamiento se encuentra llena.

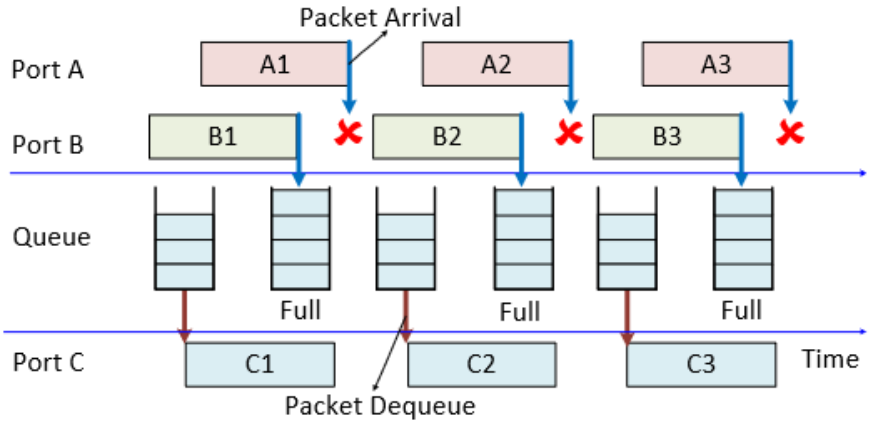

Figura 2.4: Apagón de puerto (Alasmar, 2019).

Es importante para la existencia del apagón de puertos que el tamaño de los paquetes que forman parte de los diferentes flujos sea el mismo. Contrario a la intuición, cuando ocurre el fenómeno de Outcast los flujos con mayores tiempos de RTT son los que obtienen un mejor desempeño en cuanto a throughput.

En (Wieder y cols., 2012) los autores plantean algunas soluciones para sobrellevar el problema del Outcast en TCP. En particular el uso de un tipo de cola diferente en los switches conocida como Stochastic Fair Queuing (SFQ) (McKenney, 1990) y el uso de gateways para evitar la congestión mediante Random Early Detection (RED) (Floyd y Jacobson, 1993).

2.4. ECMP

Equal-cost multi-path (ECMP) es un algoritmo de ruteo para transportar paquetes a lo largo de diferentes caminos de igual costo en una red (Hopps, 2000).

En el ambiente de las DCN modernas es clave poder explotar el hecho de que las topologías de red brindan múltiples caminos de igual coste entre pares

de nodos. Esto permite una mejora en el balance de tráfico en la red, así como la mitigación de los "puntos calientes" en la misma y los cuellos de botella (Alasmar, 2019). Lo cual se traduce en mejoras de rendimiento desde el punto de vista de baja latencia y alto throughput.

Existen dos versiones en la implementación de ECMP: Per-packet ECMP y Per-flow ECMP.

Para la versión *per-flow* los paquetes en la red se agrupan en flujos, mediante la aplicación de una función de hash sobre los mismos a partir de una 5-tupla compuesta por: IP de origen, IP de destino, número de protocolo, puerto de origen y puerto de destino. Los paquetes de un mismo flujo viajan sobre los mismos enlaces como puede observarse en la figura 2.5.

Dada la construcción del hash para el armado del flujo, se asegura que el arribo de los paquetes a destino se da en orden. Por otro lado, no es óptimo en cuanto a utilización de los enlaces, ya que pueden existir colisiones de flujos de tráfico grandes sobre los mismos enlaces mientras que otros caminos no son utilizados, lo cual termina impactando negativamente en el throughput.

Por lo tanto, en la versión *per-flow* sigue existiendo un reparto no equitativo de la carga en los enlaces de la red y se tiene la presencia de zonas calientes dentro de la topología (Alasmar, 2019).

En la versión *per-packet* el reenvío de los paquetes se maneja aleatoriamente sobre todos los caminos de igual coste permitiendo el balance óptimo de carga, ver figura 2.5.

Por contrapartida, esta versión de ECMP genera que los paquetes lleguen a destino fuera de orden, por lo que no es recomendable utilizarlo en conjunto con protocolos de transporte sensibles al reordenamiento de paquetes (por ejemplo TCP).

En la implementación del ambiente de simulación de tráfico que se presentará en secciones posteriores se utilizó la implementación de ECMP para INET descrita en (Alasmar, 2019) en su versión *per-flow*. Además, para obtener el camino más corto (o los caminos más cortos) entre pares de nodos, se utiliza una implementación del algoritmo de Dijkstra el cual se detalla a continuación.

2.4.1. Algoritmo de Dijkstra

Fue presentado por primera vez en (Dijkstra, 1959). La idea principal de este algoritmo consiste en ir explorando todos los caminos de menor coste que parten del nodo origen hacia cada nodo destino. Una vez obtenido este árbol de caminos mínimos la ejecución se detiene.

Tiene el objetivo de determinar el camino más corto dentro de un grafo con costes en sus aristas, a partir de un nodo de origen, hacia todos los demás.

El orden de complejidad del mismo es $O(|V|^2 + |E|) = O(|V|^2)$ siendo V el conjunto de todos los nodos del grafo y E el conjunto de las aristas del mismo.

Un pseudo código posible para el algoritmo es el presentado en 2.1

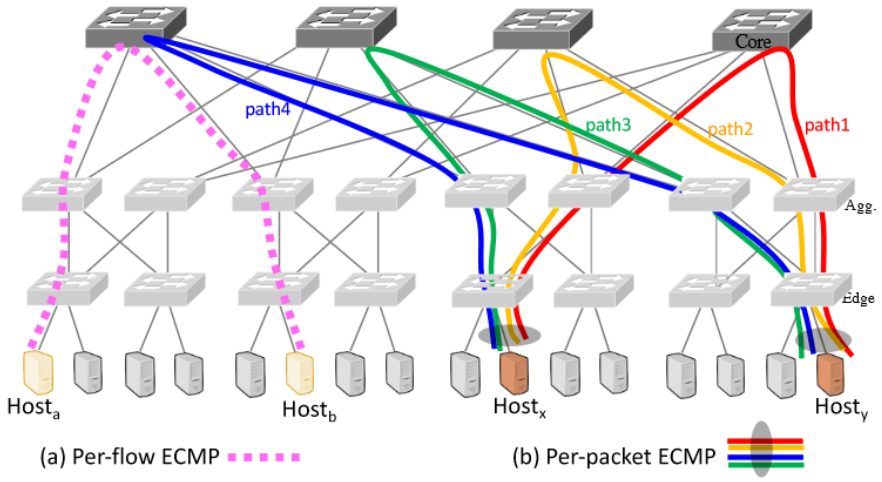

Figura 2.5: ECMP: *Per-flow* - *Per-packet* en topología fat-tree $k = 4$ (Alasmar, 2019).

```
DIJKSTRA (Grafo G, nodo_fuente s)
    para u en V[G] hacer
       distancia[u] = INFINITO
       padre[u] = NULL
       visto[u] = false
    distancia[s] = 0
    adicionar (cola, (s, distancia[s]))
    mientras que: cola ! vacia hacer:
       u = extraer_minimo(cola)
       visto[u] = true
       para todos v en adyacencia[u] hacer
           si ! visto[v]
               si distancia[v] > distancia[u] + peso (u, v) hacer
                   distancia[v] = distancia[u] + peso (u, v)
                   padre[v] = u
                   adicionar(cola,(v, distancia[v]))
```

Listing 2.1: Pseudo código para el algoritmo de Dijkstra.

2.5. DCTCP

El protocolo Data Center TCP (DCTCP) (Alizadeh y cols., 2010) surge de la necesidad de mejorar el mecanismo de control de congestión de TCP para las DCN.

Se basa en el uso de la opción de la Explicit Congestion Notification (ECN) que brindan los switches modernos en los entornos de DCN (Ramakrishnan, Floyd, y Black, 2001).

El objetivo de dicho protocolo es mejorar el deterioro de performance producido en TCP cuando se presentan los fenómenos de Incast, Outcast y otros.

La idea es que DCTCP sigue un enfoque pro-activo para intentar evitar la pérdida de paquetes en vez de esperar que dichas pérdidas se efectivicen en la red, utilizando un esquema de marcado de los paquetes que arriban a cada uno de los switches. Cuando estos reciben un paquete y su porcentaje de ocupación está por encima de una constante K previamente definida, él mismo es marcado con el código Congestion Experienced (CE).

Los resultados del análisis experimental de DCTCP muestran que los flujos de tráfico cortos muestran baja latencia, mientras que los flujos de mayor tamaño alcanzan mejor throughput. Además, DCTCP logra mejoras significativas en convergencia y reparto equitativo de los enlaces (Alizadeh y cols., 2010).

Por otro lado, en (Grosvenor y cols., 2015) se identifican dos problemas asociados a DCTCP. En primer lugar, este protocolo no convive bien junto a implementaciones tradicionales de TCP. Se observa que en DCN donde se ejecuta DCTCP algunos flujos TCP detienen su transmisión de datos por completo.

En segundo lugar, DCTCP no permite el uso de ECN Capable Transport (ECT) en los paquetes SYN y SYN/ACK (Ramakrishnan y cols., 2001). Lo cual implica que estos paquetes marcados con ECT son descartados en los escenarios de congestion (cuando se supera el K definido), y esto reduce la probabilidad de establecer una nueva conexión DCTCP rápidamente.

Como comentario adicional, DCTCP no aprovecha el hecho de que en las DCN modernas (dada su topología) existen múltiples caminos de igual costo entre todo par de hosts.

2.6. Herramientas de simulación

En las siguientes secciones se presentarán en detalle las herramientas utilizadas durante el desarrollo de este trabajo para el diseño y la ejecución de las simulaciones.

Se eligió OMNeT++ como motor de simulación dadas sus capacidades de extensión e integración con otras herramientas para el modelado especifico de redes de datos, en concreto el framework INET, así como también la existencia de una comunidad de usuarios y desarrolladores que brindan soporte ante los problemas que puedan ocasionarse.

2.6.1. OMNeT ++

OMNeT++ (OMNeT++, 2022) es un framework de simulación, basado en eventos discretos. Esta diseñado para ser modular y extensible, permitiendo la integración de sus componentes para permitir la construcción de simulaciones

de múltiples tipos de redes. Además, permite integrarse con otros proyectos especificos para ampliar sus funcionalidades de simulación (por ejemplo INET).

A pesar de que OMNeT++ no es un simulador de redes en si mismo (OMNeT++, 2022), posee amplia popularidad en las comunidades académicas e industriales por su gran adaptabilidad y ha generado una comunidad de usuarios importante que es muy útil para el soporte durante el desarrollo de las simulaciones.

Este software brinda una arquitectura basada en componentes (módulos), los cuales se desarrollan en C++ y permiten integrarse dentro de otros de mayor complejidad, esto se realiza mediante el NEtwork Description language (NED) que permite descripciones a alto nivel. Dado este tipo de diseño en componentes se cuenta con posibilidades de reusabilidad de los diversos módulos y esto permite que los mismos puedan ser integrados con facilidad a otras aplicaciones.

OMNeT++ se distribuye mediante una suite de herramientas que permiten simplificar la operativa al usuario respecto al desarrollo de las simulaciones que se deseen construir, dichas herramientas son:

- Kernel de simulación, entregado como librería C++.
- Especificación del lenguaje de descripción de topologías NED.
- IDE para la simulación basado en la plataforma Eclipse.
- Ambiente gráfico para ejecución de simulaciones - Qtenv.
- Ambiente de ejecución en línea de comandos - Cmdenv.
- Herramientas de soporte (makefile, creation tool, opp_run, opp_runall, etc).
- Documentación y ejemplos.

Las simulaciones desarrolladas para OMNeT++ están definidas por los archivos *.ini* donde se les coloca valor a todos los parámetros relevantes para dicha instancia, así como también se modela el comportamiento de los componentes en cuanto a múltiples variables como pueden ser: conexión / desconexión de un módulo, aumento de tráfico generado en determinado punto, etc. La figura 2.6 muestra el soporte para la edición de archivos *.ini* dentro del IDE para OMNeT++.

Desde el IDE también es posible personalizar los parámetros de ejecución más relevantes: La paralelización en múltiples CPUs, el tipo de interfaz de usuario a utilizar (Qtenv - Cmdenv) y variables de ambiente. La figura 2.7 muestra esta vista de configuración.

Luego de cada ejecución OMNeT++ genera archivos con las estadísticas recolectadas durante la misma, las cuales es posible analizar a través de los archivos *ANalytic File (anf)*. Durante el procesamiento de estos datos de salida es sencillo, a través del IDE, graficar dichos resultados para visualizar el comportamiento de los módulos involucrados. La figura 2.8 muestra un ejemplo de esto.

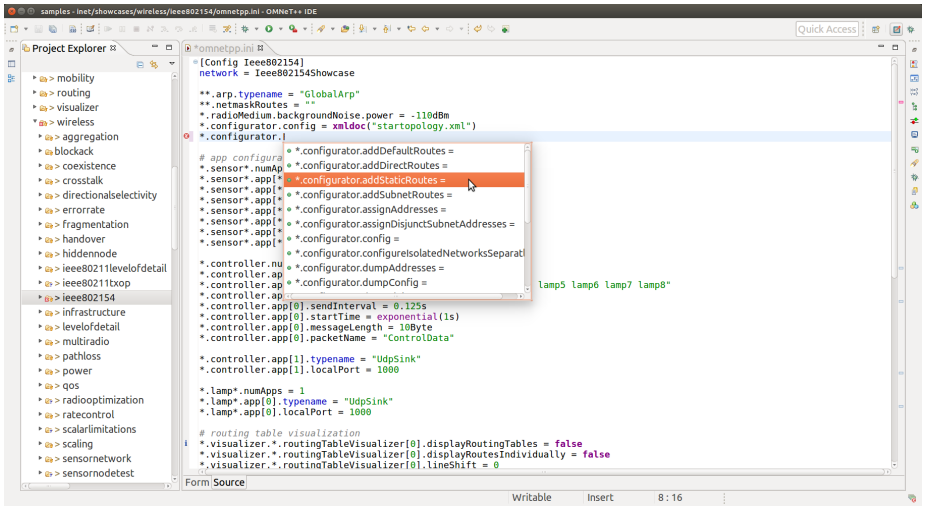

Figura 2.6: Edición de archivos *.ini* desde IDE (OMNeT++, 2022).

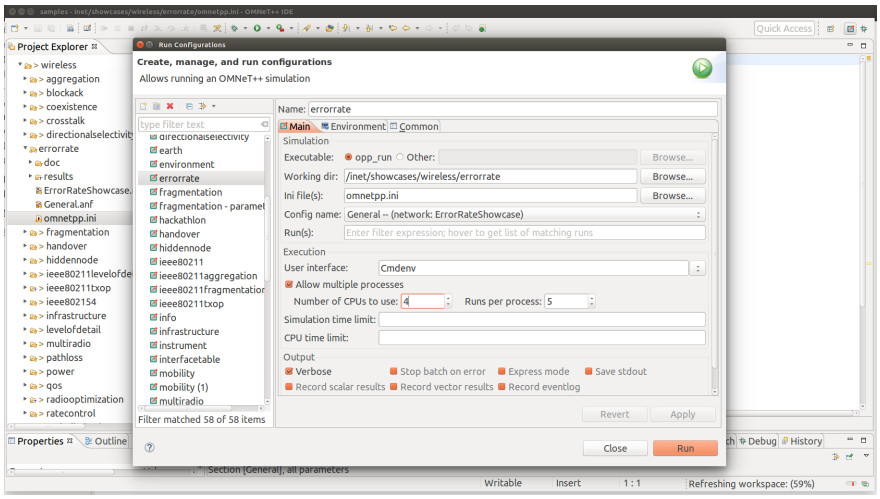

Figura 2.7: Configuración de parámetros de ejecución desde IDE (OMNeT++, 2022).

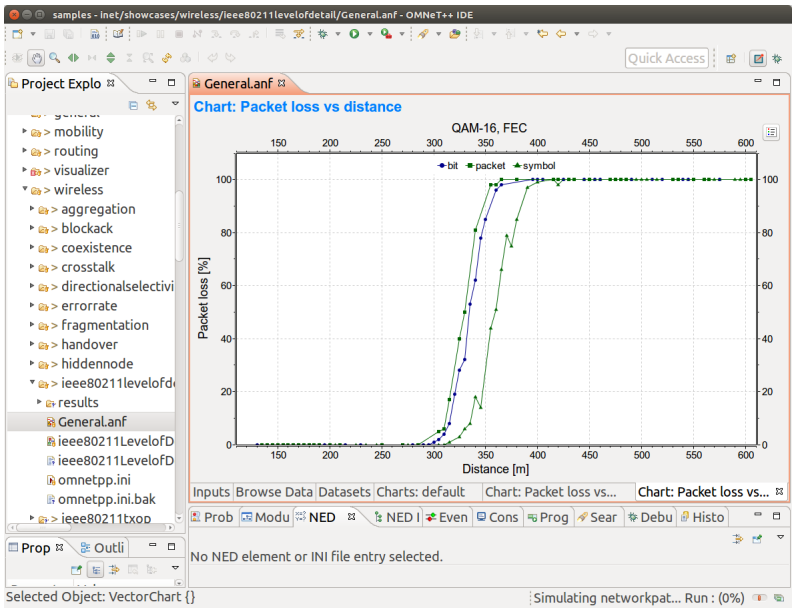

Figura 2.8: Visualización de resultados de simulación en IDE (OMNeT++, 2022).

Dentro de las opciones de ejecución de simulaciones, el ambiente Qtenv es muy útil para la etapa de desarrollo, ya que permite visualizar fácilmente el comportamiento de cada uno de los módulos durante la ejecución en distintos instantes del tiempo simulado. Esto contribuye a facilitar el proceso de debugging en la etapa de desarrollo de la simulación. La figura 2.9 muestra la visualización de una transmisión de datos por parte de varios host y un nodo receptor.

A su vez, este ambiente también cuenta con animaciones que permiten comprender a alto nivel como está operando una red entera. Al mismo tiempo, permite navegar entre los distintos submódulos de cada componente para obtener información de la ejecución a la interna de los mismos. La figura 2.10 muestra un ejemplo de esto.

Dado que OMNeT++ soporta integración con APIs (Application Programming Interface) de terceros, es posible introducir capas de datos realistas dentro de las simulaciones que se deseen ejecutar. Por ejemplo, la integración con osgEarth [7] para añadir una capa de referenciación geográfica. La figura 2.11 muestra un ejemplo de esta capacidad.

La descripción de la topología de red que se necesita diseñar se realiza mediante los archivos *.ned* donde se indican, por ejemplo, los módulos a utilizar, como se combinan para formar otros más complejos y las conexiones entre los mismos.

[7] `http://docs.osgearth.org/en/latest/#`

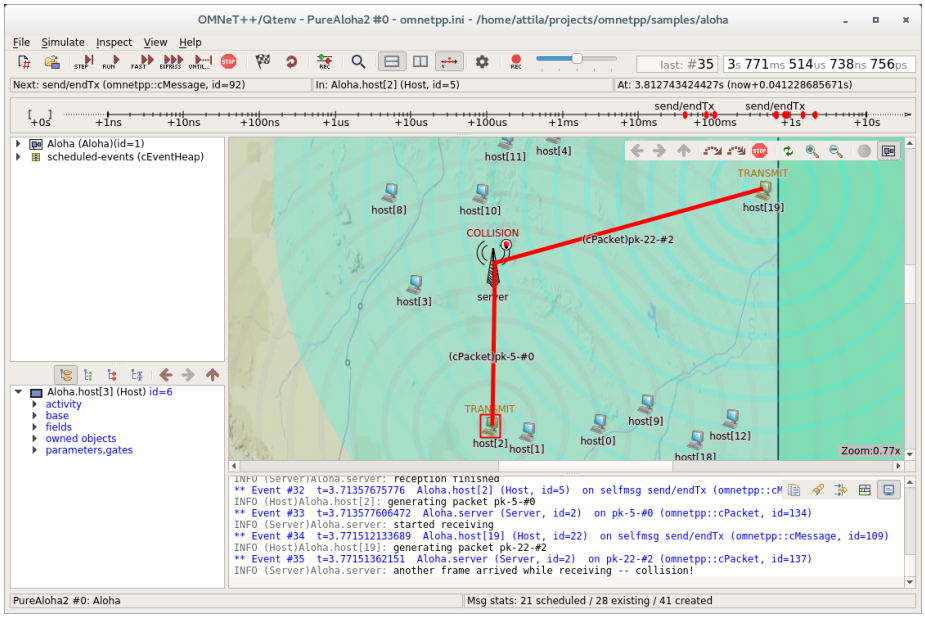

Figura 2.9: Ejecución de simulación en IDE utilizando Qtenv (OMNeT++, 2022).

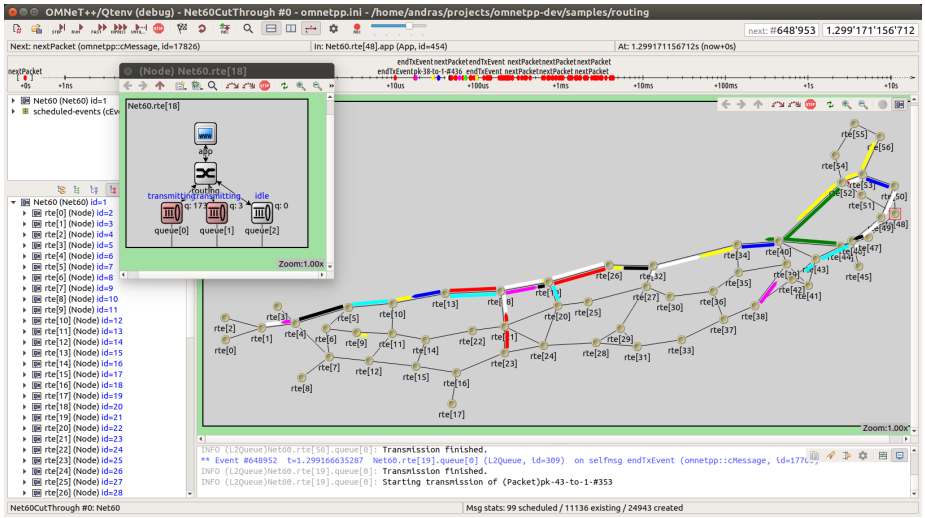

Figura 2.10: Animación de simulación en IDE utilizando Qtenv (OMNeT++, 2022).

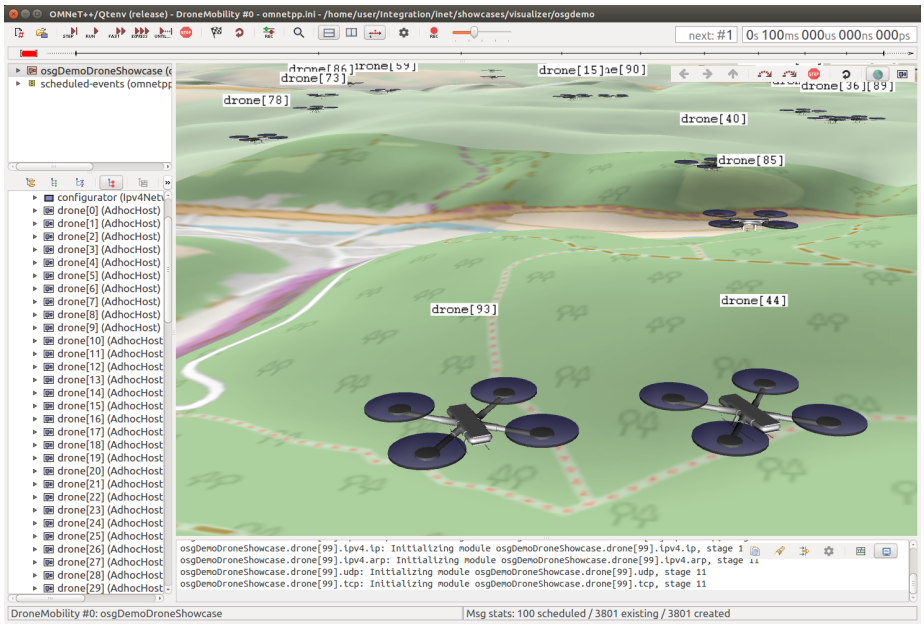

Figura 2.11: Integración con osgEarth para simulación de redes ad hoc (OMNeT++, 2022).

Estos archivos utilizan el lenguaje NED para su especificación. Dicho lenguaje, según la documentación oficial (*OMNeT++ - Simulation Manual*, s.f.), cuenta determinadas características que son importantes para el manejo de simulaciones de gran porte:

- Jerarquía: permite subdividir módulos de forma sencilla así como también combinarlos para generar estructuras de mayor complejidad.
- Reusabilidad: facilita el reciclaje de módulos desarrollados permitiendo integrarse fácilmente con otros proyectos (INET).
- Manejo de interfaces: brinda la opción de generar un módulo personalizado que implemente las funcionalidades de una interfaz, permitiendo intercambiar rápidamente entre ellos.
- Herencia: dentro de la jerarquía establecida, los módulos que heredan de otros pueden ser extendidos. Agregando parámetros, interfaces de red, etc. o cambiando los valores de los mismos.
- Paquetes: el lenguaje NED tiene un manejo de paquetes similar al de Java para evitar repeticiones de nombres en diferentes modelos.
- Anotaciones de metadata: es posible generar anotaciones para determinados módulos, parámetros, etc. Las cuales no son tenidas en cuenta por el motor de simulación, pero si pueden ser procesadas por otras herramientas que se integren con OMNeT++.

El IDE de OMNeT++ brinda soporte para la edición de estos archivos en el lenguaje NED . Allí es posible diseñar la topología de red a través de un entorno gráfico con *drag & drop* como lo muestra la figura 2.12. También se cuenta con soporte para la edición directamente en el código fuente del archivo *.ned* como se muestra en 2.13.

El motor de simulación de OMNeT++ cuenta con capacidades de *logging* para tener trazabilidad de lo sucedido durante la ejecución. Esto es de gran importancia para el usuario como un apoyo extra al momento de analizar comportamientos y solucionar posibles errores que existan en la simulación.

Dentro del IDE es posible analizar de una forma sencilla los archivos de logs generados para cada simulación. En la figura 2.14 se ve un ejemplo de la visualización.

En resumen, el conjunto de herramientas brindadas por OMNeT++ permite construir simulaciones para múltiples tipos de redes. Para el contexto de las DCN será útil integrarlo con el framework INET.

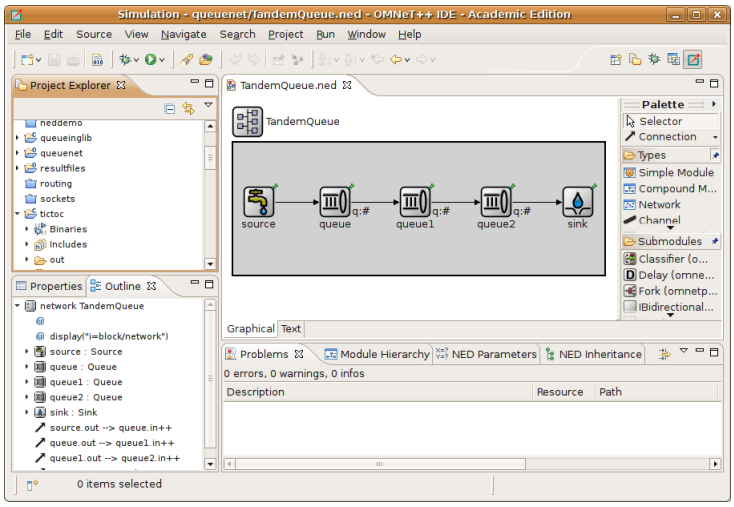

Figura 2.12: Edición de archivo *.ned* en modo gráfico desde el IDE (OMNeT++, 2022).

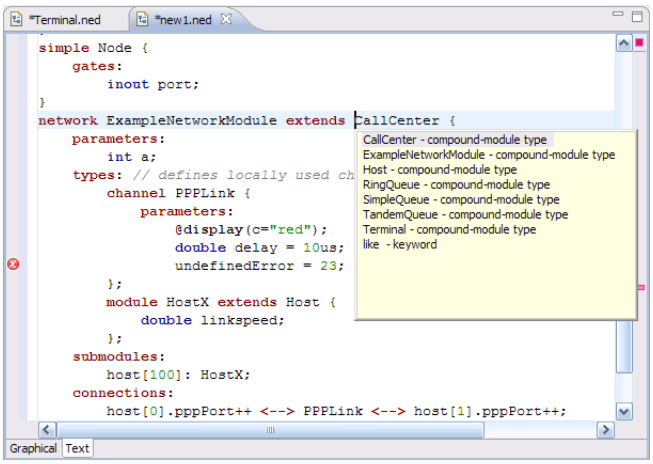

Figura 2.13: Edición de archivo *.ned* directamente en código fuente (OMNeT++, 2022).

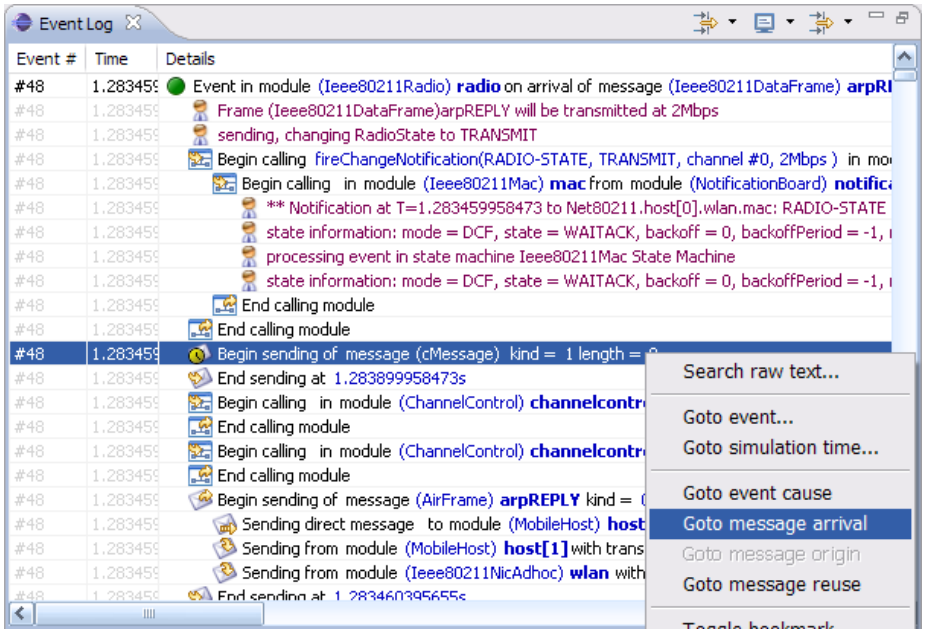

Figura 2.14: Visualización del log de eventos de una simulación (OMNeT++, 2022).

2.6.2. INET

En el caso particular de las DCN las capacidades de simulación de OMNeT++ se ven enriquecidas con el framework INET (*INET Framework*, s.f.) el cual brinda soporte para redes móviles, cableadas e inalámbricas. Siendo una herramienta open-source, brinda soporte (implementación de módulos) para los principales protocolos.

Este framework está construido basado en estos componentes que intercambian información a través del pasaje de mensajes, es posible combinar múltiples módulos para generar elementos más complejos como: servidores, routers, switches, etc. Además, es posible realizar modificaciones fácilmente sobre cualquiera de estos componentes así como generar nuevos módulos (Alasmar y Parisis, 2019).

INET extiende a OMNeT++, en el sentido de que más allá de la reutilización del motor de simulación en sí mismo, los módulos desarrollados para conformar INET pueden ser ejecutados dentro del ambiente OMNeT++ con total naturalidad, dentro del IDE o con el ambiente de línea de comandos. La figura 2.15 muestra un ejemplo de esto desde el IDE de OMNeT++.

Algunas de las características principales entregadas por INET:

- Implementación de todas las capas del modelo OSI (física, enlace, red, transporte y aplicación).
- Implementación de protocolos *plug & play*.
- Soporte IP versión 6.
- Implementación de los principales protocolos en capa de transporte (TCP, UDP, Stream Control Transmission Protocol (sctp)).
- Soporte para redes wired y wireless (Ethernet, PPP, 802.11, etc.).

Desarrollar simulaciones para DCN en el entorno OMNeT++ / INET permite asegurar resultados reproducibles, permitiendo el debugging en tiempo de ejecución y la anexión de tráfico realista mediante herramientas de terceros.

Existen múltiples casos de simulación de DCN con la combinación de OMNet++ / INET algunos de ellos están presentados en: (Montazeri y cols., 2018); (Alizadeh y cols., 2008); (Das y Sivalingam, 2013).

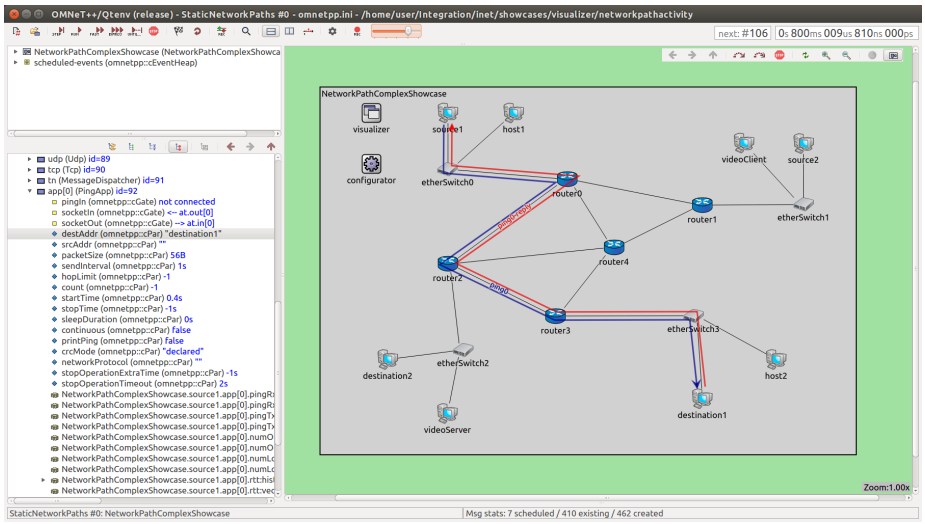

Figura 2.15: Visualización de una topología de red de datos utilizando INET (OMNeT++, 2022).

2.7. TrafPy

TrafPy es un paquete Python para la generación, manejo y estandarización de tráfico de red (*Overview of TrafPy*, s.f.).

Está compuesto por 3 módulos principales:

- *Generator*: es un paquete que permite generar trazas de tráfico personalizado (o caracterizado), las cuales pueden ser exportadas a formatos de texto como: Comma Separated Values (CSV), JavaScript Object Notation (JSON), etc, permitiendo ser importadas luego en cualquier ambiente de simulación, emulación o experimentación.

- *Benchmarker*: con este módulo se permite la generación y reproducción de indicadores de tráfico estándar. Cuenta con las mismas funcionalidades de exportación que el módulo generador.

- *Manager*: para permitir la simulación de DCN, con diferentes protocolos de ruteo.

Con TrafPy es posible replicar distribuciones de trafico teórico, incluso en la ausencia de fuentes de datos de acceso abierto. Esto sucede a menudo con las trazas de las DCN de propiedad comercial, que no se encuentran disponibles.

Dentro del motor de generación de TrafPy cada petición de información es considerada como un flujo, definido como una tarea solicitando el envío de información desde un nodo origen hacia uno destino dentro de la red. Cada

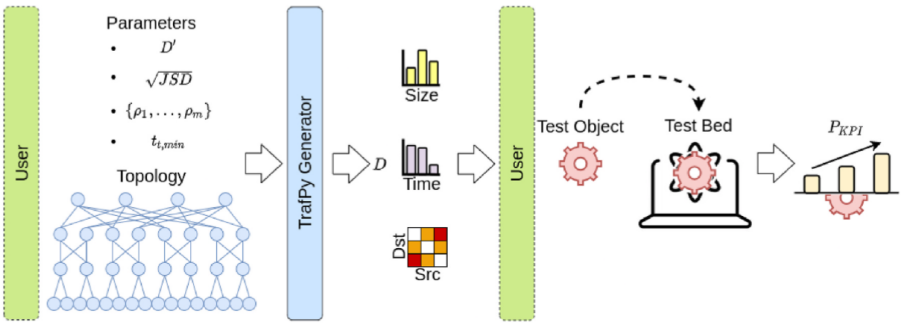

Figura 2.16: Interacción del usuario con la API de TrafPy (Parsonson y cols., 2022).

flujo deberá estar definido (y así se verá en la traza de salida) como una tupla que considera: el tamaño del mismo (en cantidad de información), el tiempo de inicio de transmisión del flujo, y un par de nodos origen/destino.

En la filosofía detrás del desarrollo de TrafPy existen dos nociones clave: no debe ser necesario el uso de datos crudos para generar trazas de tráfico y cada aspecto de la API debería ser parametrizado para asegurar la reproducibilidad (Parsonson, Benjamin, y Zervas, 2022). Para cumplir con lo primero, TrafPy brinda soporte para modelar visualmente distribuciones, de forma que un usuario solo necesita una referencia escrita o una descripción visual de la distribución de tráfico para producir el mismo. Lo segundo se cumple a través del conjunto de parámetros D' que se debe indicar al momento de seleccionar la distribución.

Como se plantea en (Parsonson y cols., 2022) todas las distribuciones de TrafPy pueden ser resumidas en el conjunto de parámetros D'. Una vez que se logra establecer dicho conjunto (por trabajo de la comunidad de desarrolladores o como fruto de la investigación) es fundamental poder reproducir la distribución original parametrizada por D'.

Entonces, es necesaria una garantía de que la distribución generada es "cercana" a la original. Para ello TrafPy utiliza la divergencia de Jensen-Shannon (JSD) (Rao, 1982). La raíz cuadrada de la divergencia de Jensen-Shannon ($\sqrt{JSD}$) se conoce como distancia de Jensen-Shannon, la cual es una métrica entre 0 y 1 que describe la similitud entre dos distribuciones (siendo 0 el indicador de igualdad y 1 la diferencia absoluta). La API de TrafPy permite a los usuarios definir que valor de $\sqrt{JSD}$ utilizar. En base a que similitud se desee se obtendrán diferentes cantidades de muestras para la aproximación de la distribución. Con valores cercanos a 0 se tomarán más muestras hasta que, por ley de los grandes números, se converja al umbral definido.

En la figura 2.16, es posible tener una vista de alto nivel del funcionamiento de la API expuesta para la interacción del usuario con los diferentes módulos de TrafPy.

El usuario debe ingresar un conjunto de parámetros de entrada para obtener la traza de tráfico D que servirá como entrada del simulador. A partir de D es posible obtener todos los indicadores clave P_{KPI} que se deseen en el simulador externo.

Entre los parámetros que se deben ingresar se encuentran: la topología de red (TrafPy brinda soporte para fat-tree); D' como el conjunto de parámetros que definen la distribución; $\sqrt{JSD}$ como la distancia de Jensen-Shannon seleccionada; t_{min} como el tiempo mínimo de duración de los flujos de tráfico para la cantidad a generar de los mismos.

Capítulo 3

Desarrollo del ambiente

En el presente capítulo se presentará el ambiente de simulación construido, mostrando el diseño del mismo, los casos de uso más relevantes, las características principales y la metodología de las métricas que se pueden obtener con dicho ambiente.

La primer sección presenta la arquitectura de la solución, detallando sus componentes principales y mostrando la interacción entre los mismos, así como alguna de las funcionalidades principales.

Luego, se presenta la metodología de las métricas que se tendrán en cuenta para el análisis de los resultados obtenidos en las simulaciones.

Más tarde, la siguiente parte del capítulo presenta los casos de uso principales del ambiente de simulación, mostrando la ejecución de los mismos mediante los diferentes ambientes de OMNeT++ y presentando el modelado de las redes y las métricas conseguidas.

Por último, se incluye una sección a modo de breve guía de usuario.

3.1. Diseño de la solución

3.1.1. Arquitectura

Existe una fuerte integración (al punto de ser difícil distinguirlos) entre el motor de simulación de OMNeT++ y el framework INET. Es posible ver al framework como una extensión del motor de simulación que incorpora nuevos módulos que permiten hacer desarrollos enfocados al ambiente de las DCN.

La figura 3.1 presenta la interacción de los principales componentes de la arquitectura, a saber:

- Kernel OMNeT++, encargado de brindar soporte para la ejecución de las simulaciones, implementando la comunicación entre los distintos módulos a través del pasaje de mensajes.
- Framework INET, para extender las capacidades de simulación de OMNeT++, principalmente enfocado al modelado de redes de datos.

- TrafPy, para generar las trazas de tráfico realista.
- *dcn_trafficGen*, el script Python [1] encargado de cargar los benchmarks necesarios para generar el tráfico.
- *traffic_processor*, el script Python para procesar los archivos *.csv* generados por TrafPy.

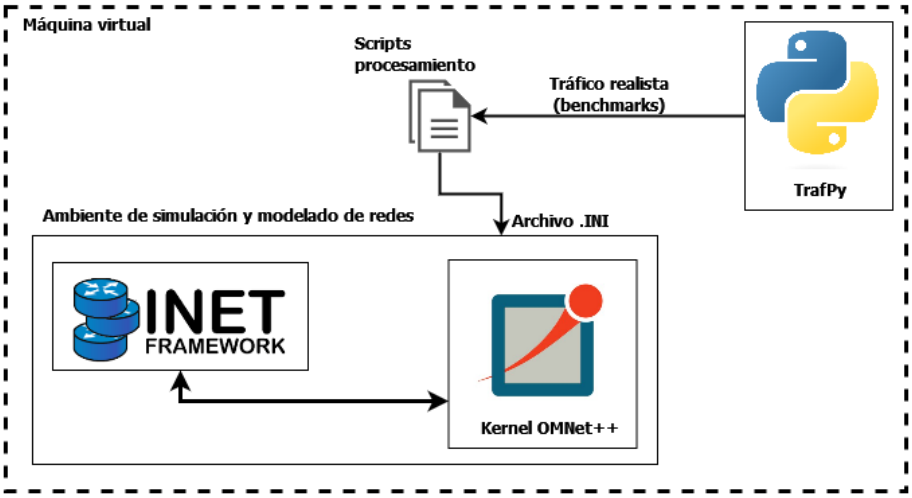

Figura 3.1: Arquitectura del ambiente de ejecución de simulaciones construido.

A nivel del framework INET se introdujeron múltiples módulos para extender las funcionalidades de simulación del kernel OMNeT++. En particular, enfocados al diseño de redes de datos. Uno de los puntos más importantes a resolver a nivel de arquitectura es la interacción de estos módulos para el enrutamiento de los paquetes / datagramas a través de la red. Este proceso comprende principalmente cinco etapas:

1. Construcción de la topología y la asignación de las direcciones IP para cada uno de los nodos en la misma. Resuelto en *Network-ConfiguratorBase::Topology*.
2. Establecimiento de los enlaces y sus costos en *Network-ConfiguratorBase::Topology*.
3. Utilizar el algoritmo de Dijkstra para encontrar el o los caminos más cortos entre dos pares de nodos. Con el método *Topology::calculateWeightedSingleShortestPathsTo*. Esto tiene como objetivo el enrutamiento "offline" de la simulación, evitando ejecutar un algoritmo en cada nodo para lograrlo.

[1] `https://www.python.org/`

4. Propagar la ruta hacia todos los nodos de la red con *Ipv4NetworkConfigurator::addStaticRoutes* como puede verse en la sección de código 2.

5. Generar las tablas de ruteo en los nodos correspondientes mediante *Ipv4RoutingTable::printRoutingTable*.

Diseño de la red

Dentro del abanico de módulos introducidos por el framework INET se destaca *Network-ConfiguratorBase::Topology* como el encargado del diseño de la topología de red para cada instancia de simulación que el usuario necesite generar.

Este módulo define a una red como un conjunto de nodos y enlaces (equivalente a un grafo), a su vez, considera a los enlaces como un par de interfaces (pertenecientes a un nodo) de origen y destino. En el anexo de este documento pueden consultarse porciones de código fuente relevantes para el entendimiento, en particular en la sección de código 1 es posible ver estas definiciones de entidades.

Para el manejo de la topología como un objeto en memoria y utilizarla como parámetro de entrada en las operaciones de otros módulos del framework es clave el uso del método *Network-ConfiguratorBase::extractTopology*. Este método se encarga de construir la representación de la topología a través de una recorrida por la jerarquía de los módulos, considerando los nodos como aquellos módulos que cuentan con la propiedad *@networkNode* y los enlaces como las conexiones (wired y wireless) entre las interfaces de red.

3.1.2. Múltiples caminos de ruteo - Adaptación ECMP

Como se detalló durante el relevamiento del estado del arte, el algoritmo de ruteo ECMP permite transportar información a lo largo de diferentes caminos de igual costo en una topología de red. Existen dos implementaciones diferentes de este algoritmo: *per-packet* ECMP y *per-flow* ECMP.

En la versión *per-flow* los paquetes en la red se agrupan en flujos, mediante la aplicación de una función de hash sobre los mismos a partir de una 5-tupla compuesta por: IP de origen, IP de destino, número de protocolo, puerto de origen y puerto de destino. Los paquetes de un mismo flujo viajan sobre los mismos enlaces.

Por otro lado en la opción *per-packet* el reenvío de los paquetes se maneja aleatoriamente sobre todos los caminos de igual coste permitiendo el balance óptimo de carga.

La implementación de ambas versiones para INET / OMNeT++ se encuentra presentada en (Alasmar, 2019). Dentro del alcance de este trabajo se tomó dicha implementación, realizada sobre OMNeT++ 5.2.1 e INET 3.6.3, adaptando el desarrollo para las versiones de producto utilizadas.

Considerando las cinco etapas del proceso de enrutamiento de paquetes / datagramas que maneja el framework INET presentadas en la sección 3.1.1, para

la implementación de ECMP, se modificó el tercer paso para que se registren todos los caminos más cortos entre todos los pares de nodos de la red. Puede consultarse la implementación en la sección de código 2.

Además, se modificó el cuarto paso para que los cambios de la etapa anterior se incluyeran al momento de cargar las rutas hacia todos los nodos. En la implementación 4 puede verse el cambio en el método. Esto último implica que las rutas de la quinta etapa se actualizan automáticamente.

La función de hash necesaria para ECMP se implementa en *IPv4RoutingTable::findBestMatchingRouteEcmp* invocada por el método *IPv4::routeUnicastPacket*. Para la obtención del camino a utilizar existen dos opciones posibles en base al tipo de ECMP usado:

- per-packet ECMP: $selectPath = rand() \,\%numPossibleEcmpRouts$
- per-flow ECMP: $selectPath = hashValue \,\%numPossibleEcmpRouts$, donde $hashValue$ es el resultado de la función de hash sobre la 5-tupla detallada anteriormente.

Puede verse esa lógica en implementada en 3.

3.1.3. Procesamiento de archivos

El uso de TrafPy como herramienta de generación de trazas realistas para las DCN simuladas allana el camino para introducir dicho tráfico dentro de las simulaciones. De cualquier forma este no es un proceso directo ya que hay diferencias importantes entre el formato de salida de TrafPy y lo que necesita OMNeT++ para poder modelar el "guión" de la simulación en el archivo *.ini*.

Por lo tanto, fue necesario desarrollar dos scripts Python para el procesamiento de los resultados generados por TrafPy, adaptando los flujos generados al formato de entrada de OMNeT++. En la figura 3.1 puede verse su interacción con el resto de los componentes de la arquitectura. El objetivo es que actúen como adaptadores entre los formatos esperados por ambas herramientas.

El primero de ellos, *dcn_trafficGen*, es el encargado de generar los benchmarks de TrafPy para el armado de las trazas de tráfico realista. Dicho script produce como salida un archivo *.csv* con la información de los flujos de tráfico generados, cada uno de ellos se identifica como una tupla de la forma: (id, nodo origen, nodo destino, cantidad de información enviada, tiempo de inicio, indicador booleano de estatus e índice). Se presenta una sección del archivo en 3.1 con dicha estructura.

```
,flow_id,sn,dn,flow_size,event_time,establish,index
0,flow_3825,servers_0,servers_7,450.0,0.0,1,0
1,flow_1590,servers_14,servers_8,5225.0,30.440066109730694,1,1
2,flow_430,servers_14,servers_8,40950.0,40.586754812974256,1,2
3,flow_2758,servers_6,servers_5,1375.0,50.73344351621782,1,3
4,flow_1125,servers_10,servers_3,10050.0,60.88013221946138,1,4
5,flow_5139,servers_14,servers_4,75.0,61.28922784872963,1,5
```

```
6,flow_3602,servers_13,servers_10,550.0,71.4359165519732,1,6
7,flow_1234,servers_13,servers_11,8500.0,81.58260525521676,1,7
8,flow_5092,servers_10,servers_5,75.0,81.99170088448501,1,8
9,flow_4977,servers_4,servers_8,100.0,92.13838958772858,1,9
10,flow_3346,servers_1,servers_12,750.0,102.28507829097215,1,10
11,flow_2258,servers_8,servers_1,2350.0,132.72514440070285,1,11
12,flow_4665,servers_13,servers_10,150.0,142.87183310394641,1,12
13,flow_3213,servers_3,servers_12,850.0,183.45858791692066,1,13
14,flow_85,servers_3,servers_2,231600.0,193.60527662016423,1,14
15,flow_2810,servers_6,servers_0,1275.0,213.89865402665137,1,15
16,flow_551,servers_3,servers_0,30425.0,234.1920314331385,1,16
17,flow_1997,servers_8,servers_9,3125.0,274.77878624611276,1,17
18,flow_883,servers_13,servers_2,14975.0,275.18788187538104,1,18
19,flow_1012,servers_14,servers_15,11900.0,295.4812592818682,1,19
20,flow_3518,servers_14,servers_13,600.0,305.62794798511175,1,20
21,flow_4386,servers_14,servers_6,225.0,315.7746366883553,1,21
22,flow_354,servers_12,servers_6,53475.0,325.9213253915989,1,22
23,flow_3042,servers_1,servers_15,1025.0,366.50808020457316,1,23
24,flow_4952,servers_1,servers_8,100.0,366.91717583384144,1,24
25,flow_3077,servers_1,servers_14,975.0,407.5039306468157,1,25
```

Listing 3.1: Estructura de archivo de salida generado por script *dcn_trafficGen*.

Luego, el segundo script, *traffic_processor*, tiene como objetivo transformar el archivo de salida generado por TrafPy a un formato que sea operable por OMNeT++. Para ello la solución encontrada fue modelar todo el escenario dentro de el archivo "*.ini*" que hace las veces de "guión" en la simulación.

Para llevar adelante este procesamiento se definió que cada uno de los flujos de tráfico generados por TrafPy se mapearían a una aplicación del tipo "*TcpSessionApp*[2]" que es la encargada de generar los datos en el nodo emisor, mientras que una aplicación del tipo "*TcpSinkApp*[3]" es la encargada de recibir dichos datos en el nodo destino del flujo.

El script *traffic_processor* recorre el archivo *.csv* como fuente de datos y carga dinámicamente los valores de configuración de cada aplicación (que representa el flujo) con los que corresponden a cada fila del archivo de entrada.

```
,flow_id,sn,dn,flow_size,event_time,establish,index
0,flow_3825,servers_0,servers_7,450.0,0.0,1,0
1,flow_1590,servers_14,servers_8,5225.0,30.440066109730694,1,1
2,flow_430,servers_14,servers_8,40950.0,40.586754812974256,1,2
3,flow_2758,servers_6,servers_5,1375.0,50.73344351621782,1,3
4,flow_1125,servers_10,servers_3,10050.0,60.88013221946138,1,4
```

Listing 3.2: Detalle de los flujos generados por TrafPy (observar valores

[2] https://doc.omnetpp.org/inet/api-current/neddoc/inet.applications.tcpapp.TcpSessionApp.html

[3] https://doc.omnetpp.org/inet/api-current/neddoc/inet.applications.tcpapp.TcpSinkApp.html

"flow_1590").

```
**.servers[14].app[0].typename = "TcpSessionApp"
**.servers[14].app[0].tOpen = 30.440066109730694us
**.servers[14].app[0].tSend = 30.440066109730694us
**.servers[14].app[0].sendBytes = 5225.0B
**.servers[14].app[0].dataTransferMode = "bytecount"
**.servers[14].app[0].connectPort = 1590
**.servers[8].app[407].typename = "TcpSinkApp"
**.servers[8].app[407].localPort = 1590
**.servers[14].app[0].connectAddress = "FatTree.servers[8]"
```

Listing 3.3: Detalle de un flujo modelado en archivo ".ini".

En 3.2 y 3.3 se presentan secciones de los archivos de configuración que permiten ilustrar este punto. Observar que para el "flow_1590" los valores de su tupla son los mismos que el de los parámetros: "*tOpen*", "*tSend*", "*sendBytes*". Luego el puerto de conexión ("*connectPort*") y la dirección de conexión ("*connectAddress*") referencian a la aplicación correspondiente a este flujo en el nodo destino.

3.1.4. Generación de redes con topología fat-tree

El ambiente de simulación construido presenta la posibilidad de generar de forma dinámica múltiples topologías de red del tipo fat-tree a partir de la cantidad de PoDs como parámetro de entrada.

Para esto se cuenta con un archivo NED encargado de la descripción de la topología. La sección de código 3.4 muestra la definición de los principales parámetros que se calculan a partir del valor k (cantidad de PoDs ingresado en la entrada), estos parámetros refieren a la cantidad de: core switches; aggregation switches; edge switches y servidores, que son necesarios para el ensamblaje de la red.

```
parameters:
    // Only even values are allowed.
    int k = default(4);
    int cant_server = default(-1);
    int dest_server = default(-1);

    int khalf = int(k/2);
    int kCore = int(k/2)^2;
    int kAgg = int((k^2)/2);
    int kEdge = int((k^2)/2);
    int kServers = int((k^3)/4);
```

Listing 3.4: Parámetros principales generación de topologías fat-tree.

Luego, es necesario instanciar submódulos del tipo *"EthernetSwitch"* para todos los switches del fat-tree y *"StandardHost"* para los servidores, esto se hace mediante la definición de una colección con el tamaño asignado por el parámetro correspondiente. La sección de código 3.5 muestra el armado de estos submódulos.

```
submodules:

  // (k/2)^2 coreSwitch routers.
    coreSwitch[kCore]: EthernetSwitch {
        @display("p=900,100,r,200");
    }

    //(k/2) aggSwitch in each pod
    aggSwitch[kAgg]: EthernetSwitch {
     @display("p=300,300,r,200");
    }

  //(k/2) edgeSwitch in each pod
    edgeSwitch[kEdge]: EthernetSwitch {
     @display("p=300,500,r,200");
    }

    servers[kServers]: StandardHost {
        @display("p=40,700,r,100");
    }
```

Listing 3.5: Definición de submódulos generación topologías fat-tree.

Por último, se deben generar las instancias de todos los enlaces que son necesarios para el armado del fat-tree, esto es lo que formará el conjunto de switches y servidores en cada PoD así como permitirá la conexión entre diferentes PoDs de acuerdo a la topología.

La sección de código 3.6 muestra el armado de estos submódulos, mediante el uso de iteraciones para generar los enlaces dinámicamente de acuerdo a los requerimientos de la topología respecto a las conexiones de los diferentes tipos de switches entre sí y con los servidores.

```
connections allowunconnected:

    for z=0..kCore-1, for p=0..k-1, for i=0..khalf-1, if
        int(z/khalf) == i {

     coreSwitch[z].ethg++ <--> EthernetLink <--> aggSwitch[khalf * p
         + i].ethg++;

    }

    for z=0..kAgg-1, for p=0..k-1, for i=0..khalf-1, if int(z/khalf)
```

```
    == p {

 aggSwitch[z].ethg++ <--> EthernetLink <--> edgeSwitch[khalf * p
     + i].ethg++;

}

for z=0..kEdge-1, for i=0..khalf-1 {
    edgeSwitch[z].ethg++ <--> EthernetLink <--> servers[khalf * z
        + i].ethg++;
}
```

Listing 3.6: Definición de enlaces para generación topologías fat-tree.

Combinando este generador de topologías con el ambiente gráfico de OMNeT++ (QtEnv) es posible validar visualmente la correctitud de este generador dinámico. En la figura 3.3 puede verse un ejemplo para un tamaño de $k = 4$, se validó con diferentes cantidades de PoDs que se omiten por ser difícil su visualización. En el capítulo de limitantes se hace mención de las problemáticas relacionadas con valores de k mayores.

3.1.5. Aplicaciones TCP utilizadas

En la sección 3.1.3 se detalló que cada uno de los flujos de tráfico generados por TrafPy se implementan como un par de aplicaciones TCP emisor/receptor, estás son *TcpSessionApp* y *TcpSinkApp* respectivamente.

TcpSessionApp

Cada una de las instancias de esta aplicación es del tipo single-connection TCP, es decir, abre una única conexión contra el nodo receptor, envía la cantidad de bytes definida como parámetro (el tamaño del flujo) y cierra la conexión. Además soporta el envío de datos a través del uso de scripts para configurar tuplas de tiempo y cantidad de bytes.

A su vez, dependiendo del tipo de conexión, esta aplicación puede actuar como cliente o servidor, lo cual se configura mediante el uso del parámetro *active*. Para el caso del servidor simplemente espera conexiones entrantes para enviar datos de respuesta, en el rol del cliente abre la conexión contra la dirección configurada y envía los datos necesarios.

El parámetro *dataTransferMode* es el que define el tipo de generación y envío de datos que se utilizará, y como esto impactará en los tiempos de ejecución de las simulaciones.

Existen 3 posibilidades para asignarle valor a *dataTransferMode*:

- *bytecount* aquí se manejan "bytes virtuales", lo que significa que únicamente se consideran las cantidades de bytes como representación, pero no los datos reales.

- *object* con esta opción se envían objetos de la clase "cMessage" sobre la conexión TCP y es ese mismo objeto el que se recibe en el nodo receptor.

- *bytestream* genera la transimisión de datos reales por la cantidad deseada, esto tiene fuerte impacto en los tiempos de simulación.

TcpSinkApp

El comportamiento de este tipo de aplicación es extremadamente sencillo, simplemente consiste en aceptar cualquier cantidad de conexiones TCP, obtener los datos que por ella arriben y descartarlos. Simplemente tiene como objetivo modelar la conexión TCP propiamente dicha, por ejemplo, implementando el envío de paquetes ACK para indicar la correcta recepción de datos.

3.2. Métricas

El simulador OMNet++ entre sus características principales cuenta con la posibilidad de generar métricas sobre las diferentes ejecuciones que se realicen. La visualización de las mismas se da en el IDE.

Estas métricas se obtienen a partir de los archivos de salida que genera el simulador para cada ejecución. En dichos archivos se graban las métricas principales que tienden a ser importantes para que los usuarios conozcan el comportamiento de la red durante la simulación.

En particular, en el contexto de las DCN y del presente trabajo, es crucial poder monitorear el desempeño del protocolo de transporte TCP para los diferentes algoritmos de control de congestión implementados por el simulador, así como la revisión del desempeño en el caso de las ejecuciones con el protocolo UDP. En general, es de mucho valor para el usuario conocer la utilización de los enlaces, monitorear si existen "puntos calientes" dentro de la red u otros análisis que sean pertinentes para él.

Durante el transcurso de este trabajo se tornó de suma importancia contar con información asociada al: throughput; la latencia y el Flow Completition Time (FCT). En el contexto de las DCN estas tres métricas son las claves para determinar la correcta calidad de servicio brindado por el data center para las aplicaciones allí desplegadas.

Un alto throughput implica que se transportan mayores volúmenes de información por unidad de tiempo a través de los enlaces de la red, lo cual repercute en mayor velocidad en la transmisión de datos. En el caso de la latencia, a nivel de aplicación, la misma tendrá fuertes impactos en los tiempos de carga experimentados por los usuarios del data center. Por último, el FCT es un buen indicador para el tiempo que toma la transmisión a través de la red de un flujo de datos de tamaño definido, lo cual impacta fuertemente en la experiencia del usuario al momento de interactuar con las aplicaciones desplegadas en las DCN.

3.3. Casos de uso

Uno de los conceptos clave, y principal ventaja, de TrafPy es que dado un conjunto de parámetros, mencionados en el capítulo de estado del arte, es posible generar trazas de tráfico de múltiples tipos de redes. Según se muestra en (Parsonson y cols., 2022) es posible modelar cualquier distribución de tráfico mediante la asignación de valores a dichos parámetros.

En la nomenclatura interna de TrafPy cada instancia del conjunto de parámetros se conoce como *benchmark*. La herramienta brinda benchmarks para los tipos de DCN más usuales. En esta línea, se consideraron los siguientes cuatro como casos de uso en el ambiente construido:

- *Commercial Cloud*
- *University*
- *Private Enterprise*
- *Social Media Cloud*

Mediante los scripts de procesamiento de archivos detallados anteriormente se generaron escenarios de prueba para cada uno de estos tipos de DCN. Modelando los casos de uso mediante "guiones" como archivos del tipo *.ini*.

Las características especificas de cada tipo de tráfico pueden verse en el cuadro 3.1.

Categoría	Tipo de tráfico	Tamaño (Bytes)	Tamaño entre arribos (μs)	Tráfico Inter-Intra-Rack (%)	Puntos calientes Carga (%)
University	Backups de bases de datos, hosting de file systems distribuidos (email, servers, web services para portales de usuario), video stream multi-cast.	80 % <10000	10 % <400, 80 % <10000	70 — 30	20 — 55
Private enterprise	Tráfico de universidad + aplicaciones 'custom' y ambientes de desarrollo.	80 % <10000	80 % <1000	50 — 50	20 — 55
Commercial cloud	Aplicaciones desplegadas en internet (búsqueda, indexado, email, vídeo, etc.), data mining, procesamiento de bigdata.	80 % <10000	Mediana 10	20 — 80	20 — 55
Social media cloud	Generadores de request/ responses (email, mensajería, etc.), bases de datos MySQL.	10 % <300, 90 % <10000	10 % <20, 90 % <10000	12.9 — 87.1	20 — 55

Cuadro 3.1: Especificaciones de los benchmarks disponibilizados por TrafPy (Parsonson y cols., 2022).

Dado que los casos de uso mencionados están construidos en base al generador de TrafPy, es posible asumir que en todos ellos se presenta el fenómeno de Outcast por las características propias de los flujos de tráfico dentro de las DCN (convivencia entre flujos pequeños y aquellos de mayor tamaño) y al uso de TCP como protocolo de transporte según lo detallado en la subsección 2.3.1.

Además de lo anterior, es importante resaltar que el escenario de validación del fenómeno de incast presentado está diseñado manualmente sin el motor de TrafPy. Modelado bajo la premisa de que en determinado momento de la simulación todos los nodos de la DCN empiezan a generar tráfico con un mismo nodo de destino, lo cual implica la sobrecarga del enlace que conecta a dicho nodo con el switch de nivel más bajo en la topología.

3.4. Guía de uso

El ambiente de simulación construido durante el presente proyecto se entrega desplegado dentro de una máquina virtual que puede ser iniciada en diferentes servidores para su uso.

A su vez, el código fuente de los componentes construidos para la interacción de las herramientas, así como los archivos de configuración se encuentran disponibles en el repositorio GitLab del proyecto (*Ambiente simulación redes fat tree*, s.f.).

Antes de ejecutar cualquiera de las simulaciones es necesario cargar las variables de entorno necesarias tanto para OMNeT++ como para INET. Para ello basta con ejecutar los siguientes comandos en una consola UNIX:

```
cd relative_path/omnetpp-6.0-linux-x86_64/omnetpp-6.0
source setenv
```

```
cd relative_path/omnetpp-6.0-linux-x86_64/omnetpp-6.0/samples/inet4.4
source setenv
```

La ejecución de las simulaciones es posible realizarla en dos ambientes diferentes: *Cmdenv* y *Qtenv*. El primero de ellos refiere a la ejecución en linea de comandos de la simulación, mostrando información pertinente sobre el desarrollo de la misma. Por otro lado, *Qtenv* es un ambiente de ejecución gráfico para las simulaciones, soporta debbuging y animaciones para facilitar la experiencia de usuario.

Para la ejecución en el ambiente *Qtenv* basta con iniciar el IDE mediante el siguiente comando (una vez cargadas las variables de entorno):

```
omnetpp
```

Dentro del IDE basta con seleccionar el archivo .ini correspondiente a la simulación y editar la configuración de lanzamiento para que se seleccione el ambiente Qtenv como muestra la figura 3.4.

El ambiente *Qtenv* es particularmente útil para la validación del correcto modelado de la red y de la interacción de los nodos durante la simulación, por ejemplo, para validar que haya múltiples caminos en uso, como puede verse en las figuras 3.3 y 3.4 .

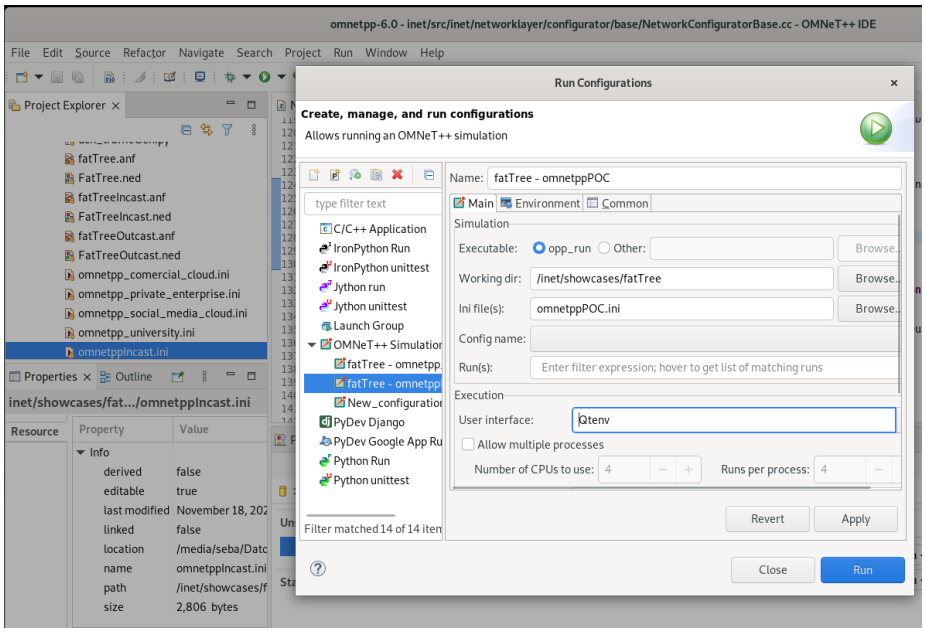

Figura 3.2: Selección ambiente de ejecución.

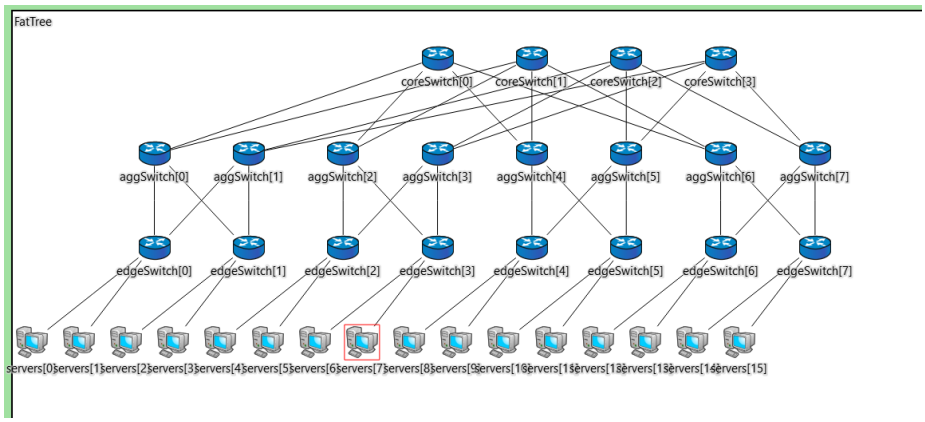

Figura 3.3: Visualización de una red con topología fat tree $k = 4$.

Dada la sobrecarga del motor gráfico y el procesamiento de las animaciones etc. la ejecución de las simulaciones en el ambiente *Qtenv* es considerablemente más lenta que su correspondiente en el ambiente *Cmdenv*. Por lo tanto, se recomienda el uso de *Qtenv* únicamente con propósitos de testing y validación.

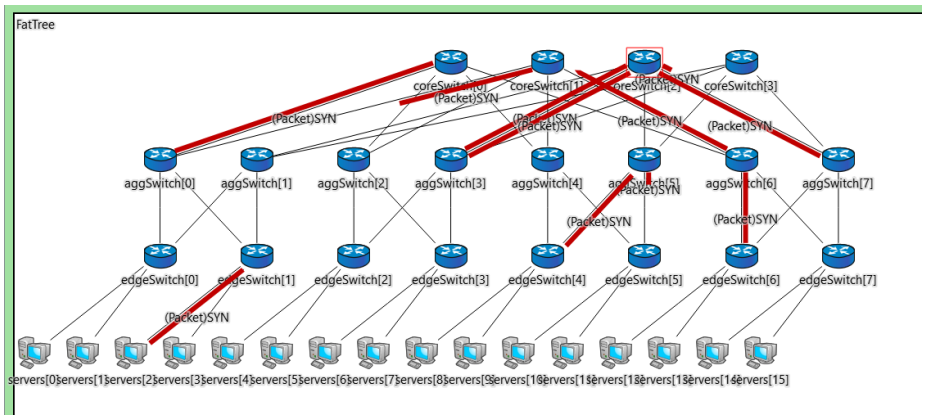

Figura 3.4: Ejecución de una simulación de tráfico en topología fat tree $k = 4$.

Para la ejecución de las simulaciones en el ambiente *Cmdenv* es necesario generar un archivo ejecutable con el empaquetado de todo el framework inet y las simulaciones construidas. Esto tiene la ventaja de que en un siguiente paso es posible utilizar la herramienta *opp_runall*[4] para paralelizar varias instancias de una simulación.

Dicho ejecutable se genera a partir de la herramienta auxiliar *opp_makemake*[5] de la siguiente forma:

```
opp_makemake -f -o inet
make MODE=release
```

Una vez generado el ejecutable, es conveniente copiar el mismo a la carpeta donde se encuentran los archivos .ini para facilitar el manejo de las rutas en el comando de ejecución. Para ello basta con ejecutar:

```
cp inet showcases/fatTree/inet
cd showcases/fatTree
```

Por último, se invoca al archivo ejecutable generado de la siguiente manera:

```
./inet
    -u Cmdenv
    -n ../../examples:..:../../src:../../test/validation:../../tests
        /networks:../../tutorials
```

[4] https://github.com/shigeya/omnetpp/blob/master/src/utils/opp_runall
[5] https://github.com/shigeya/omnetpp/blob/master/src/utils/opp_makemake

```
    -l ../../src/INET
    -f simulation_file.ini
```

Esto puede mejorar con el uso de la herramienta *opp_runall* (detalles del motor de paralelización de OMNeT++ serán presentadas en la sección 5.1), para ello basta con ejecutar el siguiente comando donde se indica el archivo ejecutable generado como parámetro:

```
opp_runall ./inet
    -u Cmdenv
    -n ../../examples:..:../../src:../../tests/validation:../../tests
       /networks:../../tutorials
    -l ../../src/INET -f simulation_file.ini
```

Capítulo 4

Validación de entorno

A continuación se presentan algunas ejecuciones del ambiente de simulación para los escenarios disponibles, junto a las salidas y resultados obtenidos para validar el funcionamiento del entorno así como también la correctitud del tráfico realista generado por TrafPy en cada escenario. Para ello, se tendrán en cuenta las métricas desarrolladas en la sección 3.2.

Todos los escenarios se ejecutan para el protocolo de transporte TCP. Cada una de las simulaciones se ejecuta dos veces, variando únicamente el algoritmo de control de congestión para TCP en cada DCN. La primera simulación utiliza TCP Reno y el segundo caso se realiza con DCTCP.

Dadas las características de la simulación (tamaño de la red, carga de tráfico, duración de la simulación, etc) no se observaron diferencias en las gráficas de DCTCP y TCP Reno, por lo que únicamente se presentan las primeras. En la sección 5.1 se detallan los inconvenientes vistos en ejecuciones de mayor tamaño.

Los parámetros utilizados son los siguientes:

- Cantidad de PoDs del fat tree: $k = 4$ (ver figura 3.3).
- Ancho de banda de los enlaces: $1Gbps$.
- Implementación de ECMP: *per-flow*.
- Cantidad de flujos generados por TrafPy: 6000.
- Distancia de Jensen-Shannon (utilizada en el motor TrafPy para la generación de tráfico): 0,9.

Por otra parte, se utilizó la herramienta de análisis de resultados provista por el IDE de OMNeT++ permite realizar diferentes tipos de gráficos, integrando múltiples nodos en una misma métrica para comparar su performance o incluso comparar diferentes ejecuciones de una simulación.

4.1. Tráfico en nube comercial

El tráfico utilizado para esta simulación fue obtenido con TrafPy a partir de los benchmarks correspondientes a DCN de nubes comerciales. Se validan dichas trazas en la ejecución de la simulación con los parámetros mencionados.

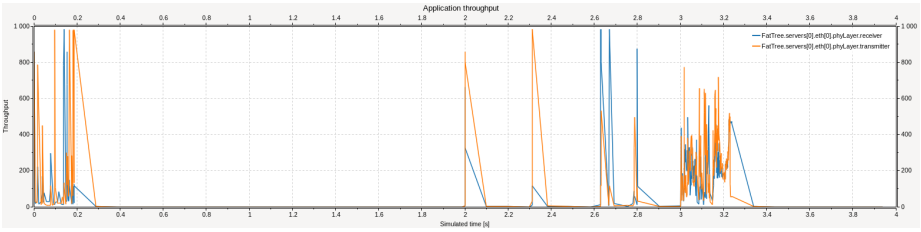

Figura 4.1: Throughput sobre tráfico emitido y recibido en el servidor 0.

En la validación del tráfico de tipo *Commercial Cloud* se obtienen los valores presentados en las figuras 4.1 y 4.2.

Para el caso del throughput en el nodo 0 se observan "ráfagas" de throughput durante distintos momentos de la simulación. Esto es razonable dados los intervalos de peticiones que arriban a dicho nodo en función de los flujos de tráfico definidos, así como también por la carga propia en la red correspondiente a los otros flujos que en cada momento se encuentran siendo transportados a través de ella.

Además, se observan picos cercanos a 1 Gbps que es la capacidad teórica máxima de los enlaces y la media de throughput alcanzado está en el entorno de los 400 - 500 Mega bit por segundo (Mbps).

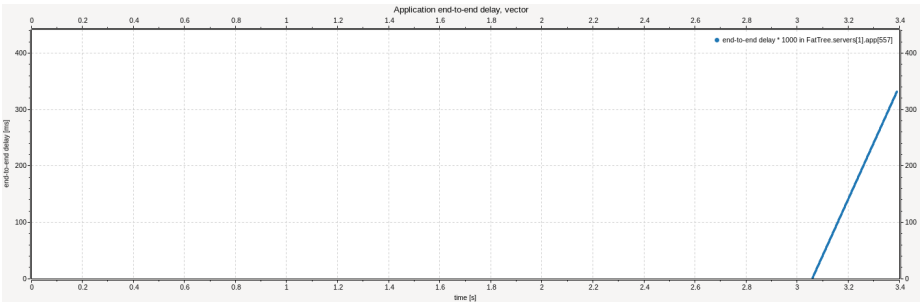

Figura 4.2: Latencia para el flujo 557 en el servidor 1.

En la gráfica 4.2 se observa la latencia presentada por una de las aplicaciones (557) que ejecutan en el servidor 1 durante la simulación. Como se observa, esta aplicación representa un flujo de tráfico que inicia de los $3,05s$ para finalizar aproximadamente a los $3,4s$. La latencia experimentada por la aplicación va

creciendo durante toda la transmisión de datos en la misma, alcanzando un pico de 340ms. Esto puede explicarse por el aumento de la carga del enlace que une al nodo 1 con el switch edge durante el transcurso de la simulación.

4.2. Tráfico de universidad

Para este escenario se usó tráfico modelado con TrafPy a partir de los benchmarks correspondientes a DCN de universidades.

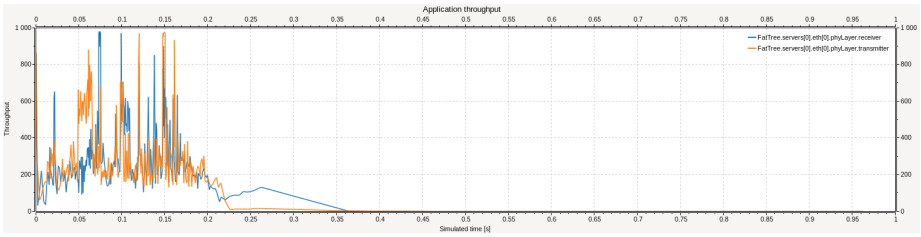

Figura 4.3: Throughput sobre tráfico emitido y recibido en el servidor 0.

En la figura 4.3 para el nodo 0 se observa que todo el tráfico de datos se da al comienzo de la simulación. Allí se observa que el throughput medio está en el entorno de los 250 Mbps con picos cercanos al Gbps. Este comportamiento puede explicarse por el modelado de los flujos en ese nodo en particular, es posible suponer una alta demanda de peticiones contra ese servidor para el inicio de la simulación, una vez que se generó la cantidad de datos definida para esos flujos el throughput desciende a 0 para el enlace que une el servidor al switch de menor nivel, lo cual es correcto.

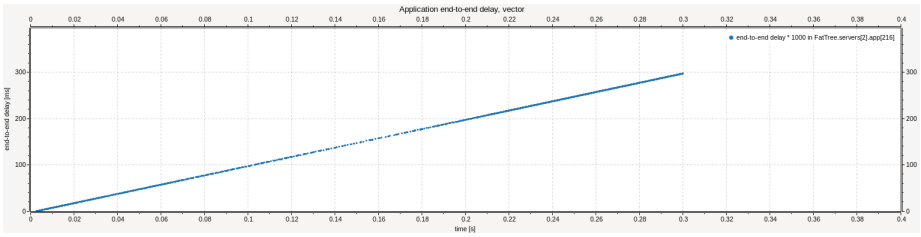

Figura 4.4: Latencia para el flujo 216 en el servidor 2.

En la gráfica 4.4 se observa la latencia presentada por una de las aplicaciones (216) que ejecutan en el servidor 2 durante la simulación. Este flujo comienza al inicio de la simulación y termina aproximadamente a los $0,3s$. La latencia experimentada por la aplicación va creciendo durante toda la transmisión de datos en la misma, alcanzando un pico de $300ms$. Esta latencia se encuentra

particularmente determinada por la carga que exista en el enlace que une el servidor 2 con el switch edge, en este caso a medida que el enlace va aumentando su carga por el flujo 216 (y otros que podrían estar actuando concurrentemente) la latencia experimentada a nivel de la aplicación aumenta de forma lineal hasta el final de la transmisión. Esto puede ser consistente con el comportamiento de la red en los casos donde la saturación de ese enlace va aumentando a medida que avanza el tiempo.

4.3. Tráfico de red empresarial

En este escenario el tráfico se obtuvo con TrafPy a partir de los benchmarks correspondientes a DCN empresariales.

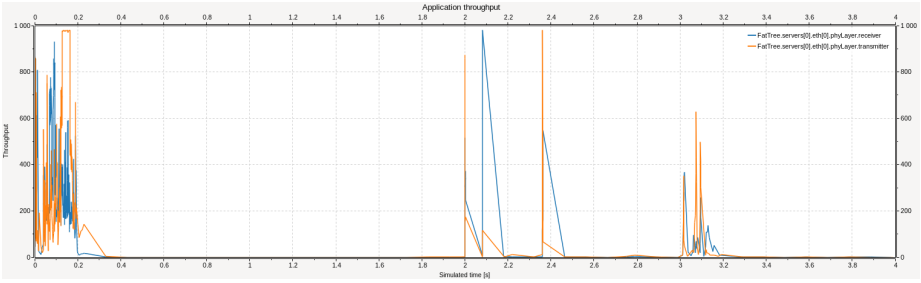

Figura 4.5: Throughput sobre tráfico emitido y recibido en el servidor 0.

Analizando la figura 4.5 aquí el patrón de tráfico sobre el nodo 0 consta de tres momentos. Las "rafagas" de throughput se observan al inicio, cercano al medio de la ejecución y al final de la simulación. Se observan picos cercanos a 1 Gbps que es la capacidad teórica máxima de los enlaces y la media de throughput alcanzado está en el entorno de los 400 - 500 Mbps.

Este comportamiento puede explicarse por el modelado de los flujos en ese nodo en particular, es posible suponer una alta demanda inicial de solicitudes contra ese servidor para el inicio de la simulación, luego flujos de menor intensidad en carga de datos para el resto de la ejecución. Por último, el throughput desciende a 0 para el enlace que une el servidor al switch de menor nivel una vez que no se generan nuevas solicitudes con origen/destino en ese nodo, lo cual es correcto.

En la gráfica 4.6 puede verse la latencia presentada por una de las aplicaciones (408) que ejecutan en el servidor 15 durante la simulación. Como se observa, esta aplicación representa un flujo de tráfico que inicia de los $0,15s$ para finalizar aproximadamente a los $0,39s$. La latencia experimentada por la aplicación va creciendo durante toda la transmisión de datos en la misma, alcanzando un pico de $220ms$. Nuevamente, para el caso de esta latencia, la misma queda determinada por la carga que exista en el enlace que une el servidor 15 con el switch edge.

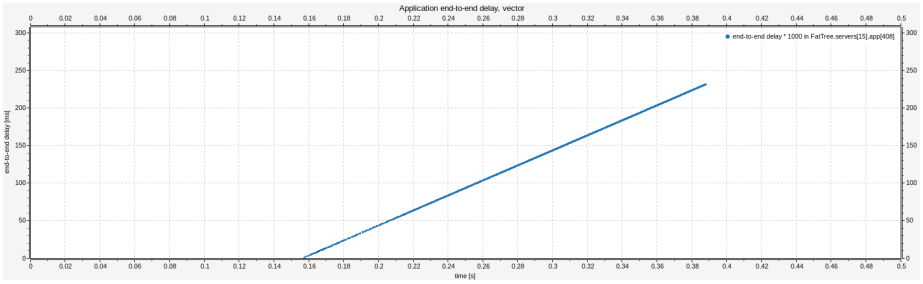

Figura 4.6: Latencia para el flujo 408 en el servidor 15.

A medida que el enlace se va saturando por el propio flujo 408 (y otros que podrían estar actuando concurrentemente) la latencia aumenta, en este caso de forma lineal, hasta el final de la transmisión. La saturación gradual de ese enlace es lo que explica este comportamiento en particular.

4.4. Tráfico en nube de redes sociales

El tráfico utilizado para esta simulación fue obtenido con TrafPy a partir de los benchmarks correspondientes a DCN de nubes que alojan aplicaciones de redes sociales.

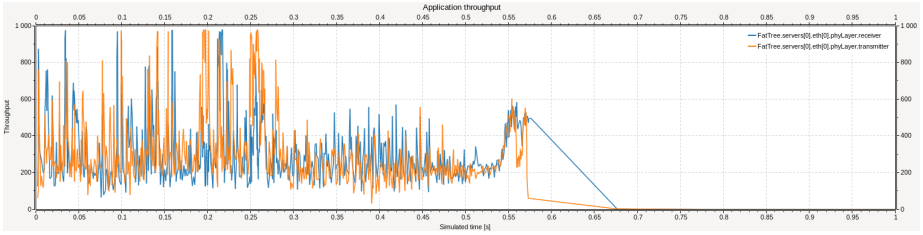

Figura 4.7: Throughput sobre tráfico emitido y recibido en el servidor 0.

Para el caso del tráfico del tipo *Social Media Cloud* presentado en la gráfica 4.7 del servidor 0, se observa un throughput medio de unos 300 Mbps con picos, en la primera mitad de la simulación, cercanos al Gbps (capacidad máxima del enlace). Una vez que los flujos involucrados con este servidor finalizan, el throughput desciende a 0 hasta el final de la simulación.

En este escenario los flujos que operan sobre el nodo 0 (tienen su origen o destino en él mismo) se dan sobre el comienzo de la ejecución, hasta aproximadamente la mitad de la misma, lo cual puede explicarse por las características brindadas por TrafPy a este tipo de tráfico. Una vez finalizadas estas solicitudes

el throughput desciende a 0 para el enlace que une el servidor al switch de menor nivel.

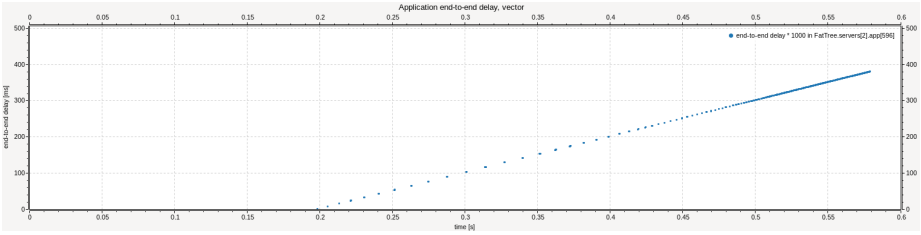

Figura 4.8: Latencia para el flujo 596 en el servidor 2.

Para este tipo de tráfico en la figura 4.8 se observa la latencia presentada por una de las aplicaciones (596) que ejecutan en el servidor 2 durante la simulación. Como puede verse, esta aplicación representa un flujo de tráfico que inicia de los $0,2s$ para finalizar aproximadamente a los $0,58s$. La latencia experimentada por la aplicación va creciendo durante toda la transmisión de datos en la misma, alcanzando un pico de $380ms$. La línea punteada del inicio de la gráfica se explica por la ausencia de mediciones para esa aplicación por parte de OMNeT++.

Una vez más, esta latencia queda determinada por la carga que exista en el enlace que une el servidor 2 con el switch edge. La cual está relacionada fuertemente con el estado del enlace a raíz de la interacción del flujo 596 con él mismo (y otros que podrían estar actuando concurrentemente).

Por lo tanto, se observa un aumento de la latencia de forma lineal, hasta el final de la transmisión. La saturación gradual de ese enlace es lo que explica este comportamiento en particular.

4.5. Incast

Para el modelado de este escenario no se utilizan trazas de tráfico generadas por TrafPy, por el contrario, se define un comportamiento especifico para asegurar la existencia del fenómeno de Incast durante la simulación. Se manejó de esta manera ya que a partir de los *benchmarks* de TrafPy no es posible asegurar que el fenómeno suceda (a diferencia del outcast como se comentó en 4) ya que sería necesario que los flujos de tráfico tuvieran como destino un mismo servidor en el mismo instante y no sería posible identificar el momento en el que eso sucede a partir del CSV generado por la herramienta. Ante esta situación se optó por el diseño manual del caso de uso.

Para ello, en el archivo *.ini* se define que todas las aplicaciones *TcpSessionApp* generen tráfico con el servidor 0 como destino al mismo tiempo, esto a fin de sobrecargar el enlace que une dicho servidor con el switch de nivel más bajo (edge).

En particular, se identifica el tráfico del servidor 4 al 0 con una etiqueta *4->0* para simplificar la obtención de métricas. Esto puede verse en la sección de código presentada en 4.1.

```
**.servers[*].numApps = 2

**.servers[*].app[0].typename = "TcpSessionApp"
**.servers[*].app[1].typename = "TcpSinkApp"

**.servers[*].app[0].tOpen = 0s
**.servers[*].app[0].tSend = exponential(3us)
**.servers[*].app[0].sendBytes = 1GB
**.servers[*].app[0].dataTransferMode = "bytecount"

**.servers[*].app[0].localPort  = 8000
**.servers[*].app[0].connectPort = 8001
**.servers[*].app[1].localPort  = 8001

*.servers[4].eth[0].measurementLayer.typename = "MeasurementLayer"
*.servers[0].eth[0].measurementLayer.typename = "MeasurementLayer"
*.servers[4].eth[0].measurementLayer.measurementStarter.flowName = "4->0"
*.servers[0].eth[0].measurementLayer.measurementRecorder.flowName =
    "4->0"

# addresing

**.servers[*].app[0].connectAddress = "FatTreeIncast.servers[0]"
```

Listing 4.1: Modelado de flujo de tráfico entre servidores 4 y 0 para escenario de Incast.

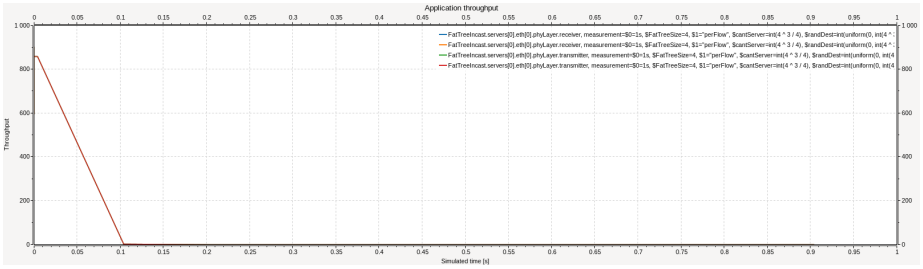

Figura 4.9: Throughput sobre tráfico generado por aplicación del servidor 0.

En el escenario de incast presentado en la gráfica 4.9 se observa el throughput alcanzado en el enlace que une al servidor 0 con el switch edge de nivel más bajo en la topología. El throughput comienza a descender a medida que se congestiona el enlace, hasta que finaliza la transmisión de datos por parte de los nodos.

Este comportamiento se corresponde precisamente con el planteado a nivel teórico por el fenómeno de incast. En este sentido, se observa una fuerte degradación de la performance en la red para este servidor a medida que el tiempo avanza durante la simulación, hasta el final de la transferencia de datos. El pico observado es de unos 850 Mbps, una vez que los flujos involucrados con este servidor finalizan, el throughput desciende a 0 hasta el final de la simulación.

En la gráfica 4.9 se evalúa el impacto del conjunto de todos los flujos de tráfico que actúan sobre el servidor 0. A continuación, en la gráfica 4.10 se detalla el comportamiento de un flujo en particular. Para ello se utiliza la etiqueta *4->0* establecida en el archivo que hace de guión para la prueba.

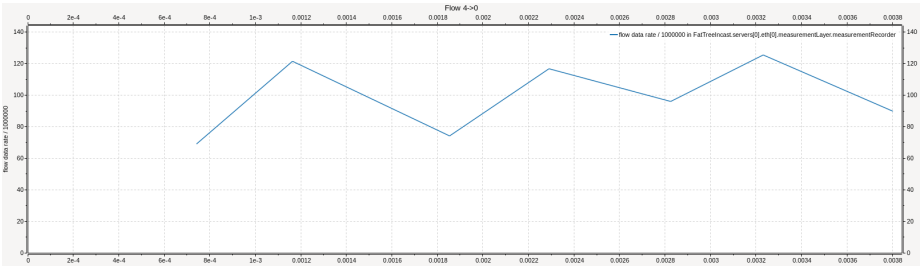

Figura 4.10: Throughput asociado al flujo monitoreado entre el servidor 4 y el 0.

En la gráfica 4.10 se observa como el throughput asociado al tráfico entre este par de nodos oscila en función de la carga del enlace respecto al resto del tráfico, manteniéndose siempre entre los 70 y los 130 Mbps. Esto implica que la degradación de la performance en la red no es igual para todos los flujos en cada momento, dependiendo de sus características particulares (tipo de tráfico, tamaño de los mismos, etc.) y de como se desocupen las colas de los switches que se encuentran sobrecargados en la red, permitiendo diferencias en el throughput de los diferentes flujos.

Capítulo 5

Limitantes

En este capítulo se presentan las principales limitantes y condicionantes del entorno desarrollado. Dichos inconvenientes surgen respecto al uso del simulador OMNeT++ y los diferentes frameworks integrados con él los cuales se detallan a continuación.

5.1. Performance

En primer lugar, para topologías relativamente pequeñas (fat-tree con $k = 4$) los tiempos de simulación para los experimentos eran considerablemente altos, en el entorno de los 60 - 80 minutos por corrida.

Con el fin de mitigar esta situación se realizó una investigación dentro del entorno de ejecución con el comando unix *htop* [1]. Allí se determinó que el cuello de botella en la ejecución es el propio CPU y no la memoria RAM del ambiente como puede verse en la figura 5.1.

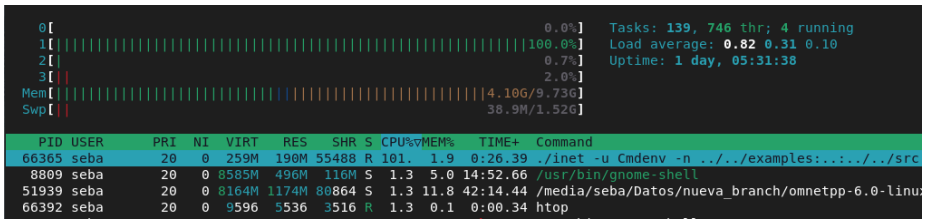

Figura 5.1: Monitoreo de ejecución de simulación.

En la instalación OMNeT++ viene incluida la herramienta *opprunall* (OMNeT++, 2022) que es la encargada de realizar la paralelización de las ejecuciones. Este

[1] htop - an interactive process viewer `https://htop.dev/`

programa se encarga de asignarle CPUs independientes a cada simulación que se desee ejecutar.

Si bien es útil para disminuir los tiempos de ejecución total de forma lineal respecto a la cantidad de repeticiones o simulaciones independientes que se deseen correr, no permite asignar diferentes CPUs a una misma instancia de simulación.

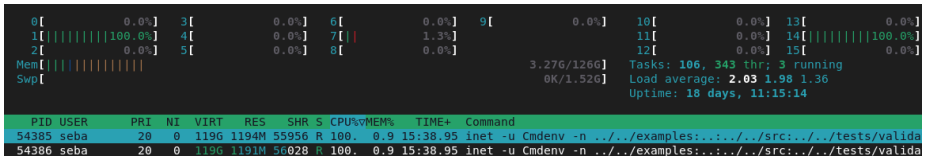

Figura 5.2: Ejecución de simulación con *opprunall*.

La figura 5.2 muestra la salida de *htop* para una corrida de dos simulaciones con la herramienta de paralelización *opprunall*. Pudiendo observar que se repite el fenómeno de núcleos de CPU con carga del 100 % mientras otros permanecen sin trabajo asignado, pero en este caso cada experimento tiene asignado un CPU para su ejecución simultanea.

Ahondando sobre este tema en los foros de discusión sobre OMNeT++ / INET [2] [3] [4] se pudo ver que para que una simulación sea paralelizable a nivel de CPU en una misma instancia se requiere que todos los módulos de OMNeT++ involucrados se comuniquen vía pasaje de mensajes. En el caso de INET esto no se cumple por lo que utilizar los módulos de dicho framework implica perder la capacidad de paralelización de una misma instancia de simulación.

5.2. Integración de herramientas

Durante el proceso de diseño y desarrollo del presente trabajo se revisaron las opciones a nivel de OMNeT++ para poder cargar información de fuentes externas al motor de simulación. En dicho proceso se ahondó sobre el *Scenario Scripting* [5], el cual permite, mediante el módulo *Scenario Manager* definir acciones sobre todos los módulos que componen la red a un tiempo determinado por el usuario, guionando la simulación.

Como se comentó en la sección 3.1, fue necesario el desarrollo de un script de procesamiento de archivos de texto. Esto con el fin de poder convertir entre el formato de salida generado por TrafPy y la entrada de datos necesaria por OMNeT++ para poder guionar la simulación. Dadas las limitaciones del *Scenario*

[2] https://groups.google.com/g/omnetpp/c/VHUyvTy8-q0/m/s9qQgZqYBwAJ

[3] https://stackoverflow.com/questions/33399184/omnet-parallelize-single-run-simulation

[4] https://arxiv.org/abs/1409.0994

[5] https://inet.omnetpp.org/docs/users-guide/ch-scenario-scripting.html

Manager para las necesidades particulares en este caso, las cuales se detallarán como línea de investigación futura.

5.3. Métricas

A su vez, por una limitante propia de INET para el manejo de las estadísticas asociadas a la representación de los flujos no fue posible medir el FCT para los escenarios TCP y UDP. Para la obtención de métricas relacionadas a los flujos de datos durante la simulación se deben utilizar los módulos *"FlowMeasurementStarter"* y *"FlowMeasurementRecorder"* de acuerdo a lo presentado en la documentación oficial del framework [6]. Dichos módulos generan diversas estadísticas que luego pueden ser procesadas por el usuario, esto puede consultarse en [7][8].

Aquí se presentan dos situaciones diferentes:

- Para el caso de TCP, las estadísticas generadas por el motor de simulación de OMNeT++ están completamente enfocadas al concepto de "paquete" no así para los segmentos de este protocolo. Por lo tanto, en aquellos escenarios donde se opere con aplicaciones TCP dentro de la simulación no se registran datos para esas aplicaciones.

- Utilizando el protocolo UDP la situación es la siguiente: Dentro de las aplicaciones disponibilizadas por parte del framework INET, únicamente *"UdpSourceApp"* maneja los módulos *"FlowMeasurementStarter"* y *"FlowMeasurementRecorder"* para el registro de datos asociados a un flujo (la estructura de dicha aplicación puede verse en la figura 5.3). Pero el comportamiento de la misma es diferente del que se necesita para modelar los flujos de tráfico realista generados por TrafPy, básicamente, esta aplicación comienza la generación de paquetes UDP en un momento especifico pero no es posible indicarle que se detenga luego de un tiempo determinado a partir de un parámetro o similar. El resto de las aplicaciones UDP disponibles si cuentan con la opción de detener la generación de tráfico pero no se integran con los módulos de gestión de flujos.

[6] `https://inet.omnetpp.org/docs/showcases/measurement/flow/doc/index.html`

[7] `https://doc.omnetpp.org/inet/api-current/neddoc/inet.queueing.flow.FlowMeasurementStarter.html`

[8] `https://doc.omnetpp.org/inet/api-current/neddoc/inet.queueing.flow.FlowMeasurementRecorder.html`

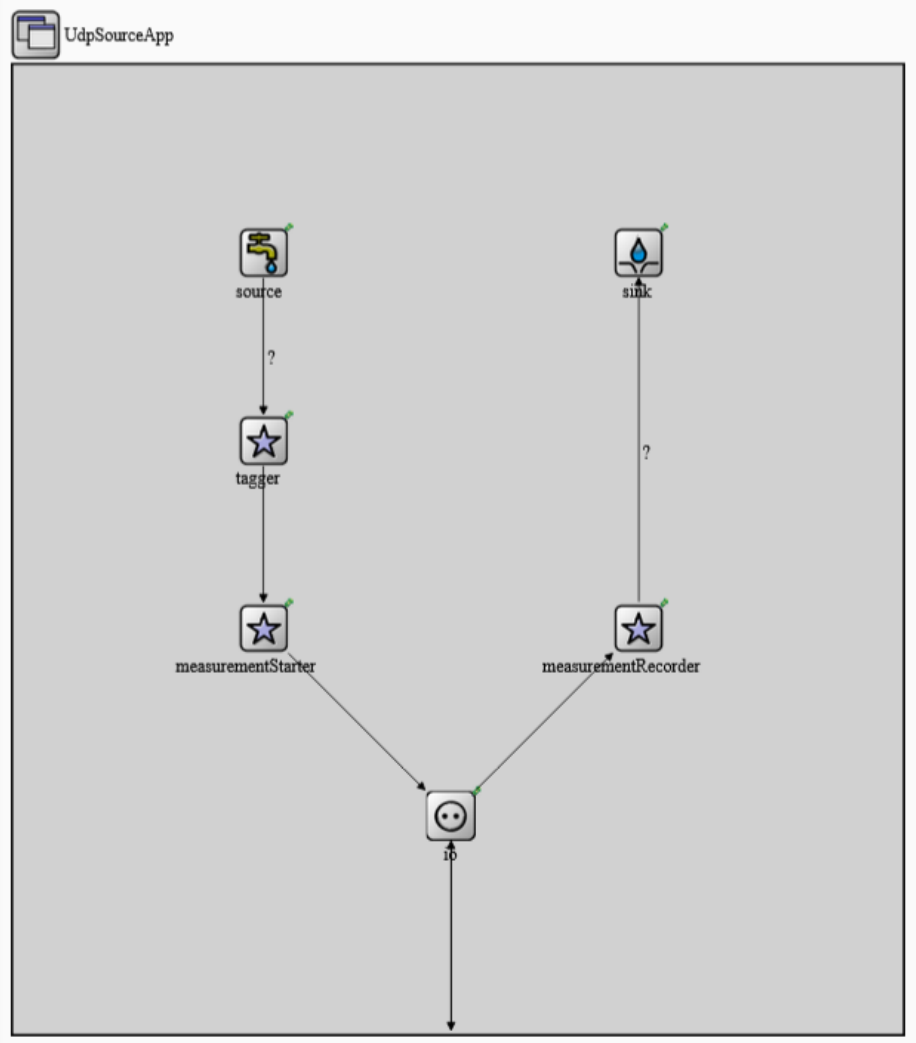

Figura 5.3: Estructura de aplicación *"UdpSourceApp"*.

Capítulo 6

Consideraciones finales

En primer lugar, se cumplió con el objetivo principal del proyecto, el desarrollo de un entorno que permita simular el comportamiento de diferentes data centers con topologías fat-tree utilizando trazas de tráfico realistas. Además este entorno permite obtener métricas para el análisis por parte del usuario.

A su vez, también se logró contar con la posibilidad de parametrizar los aspectos principales de la red, por nombrar alguno de ellos: cantidad de PoDs del fat-tree, ancho de banda de los enlaces, tráfico a generar por parte de las aplicaciones de los servidores, tiempos de inicio de los flujos, etc.

Se realizó una investigación pormenorizada del estado del arte en lo referente a: las DCN, las topologías fat-tree y los protocolos de transporte especialmente diseñados para los entornos de DCN.

Además, se logró integrar el motor de simulación OMNeT++ con el framework de modelado de redes INET y el motor de generación de tráfico realista TrafPy dentro del ambiente construido.

También se logró adaptar la implementación de ECMP para el código estándar de OMNeT++ en base a lo construido en (Alasmar, 2019) con las versiones de OMNeT++ e INET utilizadas en el presente trabajo, se enriqueció la ejecución de las simulaciones con el agregado de el algoritmo de Dijkstra para habilitar múltiples caminos de ruteo entre pares de nodos de la topología.

En la etapa de implementación de este trabajo se tuvo un inconveniente general respecto de las diferencias entre las versiones utilizadas en (Alasmar, 2019) y las actuales disponibles para los componentes de OMNeT++ e INET.

El proceso de adaptación de las implementaciones de ECMP para las versiones 6.0.0 de OMNeT++ y 4.4 de INET, tomando como base lo resuelto para las versiones 5.2.1 y 3.6.3 de OMNeT++ e INET respectivamente, fue arduo. No se cuenta con un registro exacto de la cantidad de horas invertidas pero se estima que un porcentaje alto del tiempo empleado en la implementación recayó en esta tarea.

Esto se explica en los cambios sustanciales a nivel de parámetros (nombres, tipos, estructuras) en cada uno de los módulos de los componentes para las diferentes versiones. Lo cual implicó el análisis de historiales de modificaciones

así como el rastreo manual de los cambios para poder realizar modificaciones equivalentes en las nuevas versiones para todo lo que era desarrollo no estándar del stack INET - OMNeT++.

6.1. Despliegue

La solución construida se encuentra operativa dentro de los servidores del grupo MINA[1], desplegada en una maquina virtual, la cual contiene la integración de todas las herramientas empleadas.

Para este trabajo se emplearon las versiones 6.0.0 de OMNeT++ y 4.4 de INET, así como también la versión del generador 1.0 TrafPy. La figura 3.1 muestra un diagrama general de la arquitectura del ambiente de simulación construido.

6.2. Trabajo Futuro

Dado que ahora se cuenta con el entorno de simulación, esto permite realizar experimentos sobre el comportamiento de los diferentes protocolos de transporte y/o los algoritmos de control de congestión para TCP. Validando hipótesis previas sobre el comportamiento de la red en diferentes escenarios, contrastando los resultados obtenidos en las simulaciones efectuadas contra los teóricamente esperados.

Por ejemplo, se tienen hipótesis de comportamiento a partir de la literatura consultada para este trabajo. Se supone que en escenarios de tráfico realista DCTCP logra mejores valores en las métricas consideradas que sus pares TCP Reno, Tahoe, etc. Se sugiere como trabajo futuro la constatación de estos resultados teóricos mediante simulaciones en el ambiente construido.

Por otra parte, se identifica como un escenario de posible trabajo futuro la adaptación del framework INET para que únicamente se comunique mediante el pasaje de mensajes. Esto implica que se podría contar con soporte para asignar múltiples CPUs a una misma instancia de simulación que se encuentra ejecutando, de esta forma se lograría reducir el cuello de botella visualizado durante el uso del ambiente construido, bajando considerablemente los tiempos de ejecución de las simulaciones.

Otra línea de investigación y trabajo futuro que es posible considerar, se trata del análisis del comportamiento del ambiente de simulación para fat-tree con mayores cantidades de PoDs (valores de k mayores). Durante el presente trabajo, por los tiempos de simulación que se han detallado anteriormente, no fue posible hacer pruebas con tráfico realista y $k > 6$. Sería interesante ver en esos escenarios como se comportan los protocolos de ruteo así como ver las métricas de la DCN para esas simulaciones.

Aportaría gran valor abordar la carga de archivos de tráfico realista en OMNeT++ mediante el uso de *Scenario Scripting*. Esto implicaría resolver dos

[1] `https://www.fing.edu.uy/inco/grupos/mina/`

inconvenientes principales:

- Este módulo trabaja a partir de un archivo eXtensible Markup Language (XML) con la información de lo que se desea que ocurra durante la simulación. Por lo tanto es necesario generar un procesamiento para poder transformar el CSV de TrafPy en el guión XML que acepta OMNeT++.
- Dentro de cada módulo definido en el stack INET/OMNeT++ existen parámetros. Luego de consultarse la documentación oficial del producto es posible ver que mediante *Scenario Scripting* solo es posible modificar aquellos que tengan la etiqueta *"mutable"* al lado de su definición.

  En el caso particular de las aplicaciones TCP / UDP que modelan el flujo, los parámetros *connectAddress* y *connectPort* no están marcados con esa etiqueta, por lo que no es posible ir cambiando su valor en tiempo de ejecución de forma estándar.

Referencias

Alasmar, M. (2019, 12). Understanding the characteristics of internet traffic and designing an efficient raptorq-based data transport protocol for modern data centres. Descargado de `http://sro.sussex.ac.uk/id/eprint/89371/`

Alasmar, M., y Parisis, G. (2019, 12). Evaluating modern data centre transport protocols in omnet++/inet. *EPiC Series in Computing*, *66*, 1-10. Descargado de `https://github.com/mohammedalasmar/ndpTcpDatacentreOmnetppModel` doi: 10.29007/H27X

Alizadeh, M., Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., ... Sridharan, M. (s.f.). Data center tcp (dctcp). Descargado de `http://130.203.136.95/viewdoc/summary?doi=10.1.1.221.2616`

Alizadeh, M., Atikoglu, B., Kabbani, A., Lakshmikantha, A., Pan, R., Prabhakar, B., y Seaman, M. (2008). Data center transport mechanisms: Congestion control theory and ieee standardization. Descargado de `https://web.stanford.edu/~balaji/papers/QCN.pdf`

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., ... Sridharan, M. (2010). Data center tcp (dctcp). *Computer Communication Review*, *40*, 63-74. doi: 10.1145/1851275.1851192

Ambiente simulación redes fat tree. (s.f.). Descargado de `https://gitlab.fing.edu.uy/sebastian.cabrera/proygradocabrera`

A. Przygienda, P. T. B. R. D. A., A. Sharma. (2021). Rift: Routing in fat trees. Descargado de `https://datatracker.ietf.org/drafts/current/`

Brakmo, L. S., y Peterson, L. L. (1995). Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, *13*, 1465-1480. doi: 10.1109/49.464716

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., y Jacobson, V. (2017, jan). Bbr: Congestion-based congestion control. *Commun. ACM*, *60*(2), 58–66. Descargado de `https://doi.org/10.1145/3009824` doi: 10.1145/3009824

Chen, Y., Griffith, R., Liu, J., Katz, R. H., y Joseph, A. D. (2009). Understanding tcp incast throughput collapse in datacenter networks. *Proceedings of the 1st ACM workshop on Research on enterprise networking - WREN '09*. doi: 10.1145/1592681

Clos, C. (1953). A study of non-blocking switching networks. *Bell System*

Technical Journal, *32*, 406-424. Descargado de https://archive.org/details/bstj32-2-406 doi: 10.1002/j.1538-7305.1953.tb01433.x

Das, T., y Sivalingam, K. M. (2013). Tcp improvements for data center networks. *undefined*. doi: 10.1109/COMSNETS.2013.6465539

Dijkstra, E. W. (1959, 12). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269-271. Descargado de https://link.springer.com/article/10.1007/BF01386390 doi: 10.1007/BF01386390/METRICS

Floyd, S., y Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, *1*, 397-413. doi: 10.1109/90.251892

Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., y Crowcroft, J. (2015). *Attaining the promise and avoiding the pitfalls of {TCP} in the datacenter*. Descargado de https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor

Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., y Wojcik, M. (2017, 8). Re-architecting datacenter networks and stacks for low latency and high performance. *SIGCOMM 2017 - Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, 29-42. doi: 10.1145/3098822.3098825

Hopps, C. (2000, 11). Analysis of an equal-cost multi-path algorithm. Descargado de https://www.rfc-editor.org/info/rfc2992 doi: 10.17487/RFC2992

Inet framework. (s.f.). Descargado de https://inet.omnetpp.org/

Kheirkhah, M., Wakeman, I., y Parisis, G. (s.f.). Mmptcp: A multipath transport protocol for data centers.

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

Linux kernel source tree. (s.f.). Descargado de https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0f8782ea14974ce992618b55f0c041ef43ed0b78

Lsvr working group. (s.f.). Descargado de https://datatracker.ietf.org/wg/lsvr/about/

McKenney, P. E. (1990). Stochastic fairness queueing. *Proceedings - IEEE INFOCOM*, 733-740. doi: 10.1109/INFCOM.1990.91316

Medhi, D., y Ramasamy, K. (2018). Routing and traffic engineering in data center networks. *Network Routing*, 396-422. Descargado de https://www.researchgate.net/publication/322173137_Routing_and_Traffic_Engineering_in_Data_Center_Networks doi: 10.1016/B978-0-12-800737-2.00014-4

Montazeri, B., Li, Y., Alizadeh, M., y Ousterhout, J. (2018, 3). Homa: A receiver-driven low-latency transport protocol using network priorities (complete version). *SIGCOMM 2018 - Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 221-

235. Descargado de http://arxiv.org/abs/1803.09615http://dx.doi.org/10.1145/3230543.3230564 doi: 10.1145/3230543.3230564

Munir, A., Qazi, I. A., Uzmi, Z. A., Mushtaq, A., Ismail, S. N., Iqbal, M. S., y Khan, B. (s.f.). *Minimizing flow completion times in data centers.*

OMNeT++. (2022). *Omnet++ discrete event simulator.* https://omnetpp.org/. (Accedido: 2022-09-24)

Omnet++ - simulation manual. (s.f.). Descargado de https://doc.omnetpp.org/omnetpp/manual/#cha:ned-lang

Overview of trafpy. (s.f.). Descargado de https://trafpy.readthedocs.io/en/latest/

Parsonson, C. W., Benjamin, J. L., y Zervas, G. (2022, 11). Traffic generation for benchmarking data centre networks. *Optical Switching and Networking*, *46*, 100695. doi: 10.1016/J.OSN.2022.100695

Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., y Handley, M. (2011). Improving datacenter performance and robustness with multipath tcp. *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM - SIGCOMM '11*. doi: 10.1145/2018436

Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., y Handley, M. (2010). Data center networking with multipath tcp. *Proceedings of the 9th ACM Workshop on Hot Topics in Networks, Hotnets-9*. Descargado de https://www.researchgate.net/publication/220862296_Data_center_networking_with_multipath_TCP doi: 10.1145/1868447.1868457

Ramakrishnan, K., Floyd, S., y Black, D. (2001, 9). The addition of explicit congestion notification (ecn) to ip. Descargado de https://www.rfc-editor.org/info/rfc3168 doi: 10.17487/RFC3168

Rao, C. R. (1982, 2). Diversity and dissimilarity coefficients: A unified approach. *Theoretical Population Biology*, *21*, 24-43. doi: 10.1016/0040-5809(82)90004-1

Rfc 5681 - tcp congestion control. (s.f.). Descargado de https://datatracker.ietf.org/doc/rfc5681/

Rfc 6582 - the newreno modification to tcp's fast recovery algorithm. (s.f.). Descargado de https://datatracker.ietf.org/doc/rfc6582/

Tcp bbr congestion control comes to gcp – your internet just got faster — google cloud blog. (s.f.). Descargado de https://cloud.google.com/blog/products/networking/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster

Wieder, A., Bhatotia, P., Post, A., y Rodrigues, R. (2012). *The tcp outcast problem: Exposing unfairness in data center networks.* Descargado de http://ec.europa.eu/eurostat/statistics-explained/index.php/International_trade_in_goods

Anexo 1

En este capitulo se presenta la información adjunta al proyecto, por motivos de facilitar la visualización del presente informe.

Todas las secciones de código fuente referenciadas en las distintas secciones del presente trabajo se encuentran aquí para facilitar la lectura del mismo.

```
/**
 * Represents a node in the network.
 */
class Node : public Topology::Node {
  public:
    cModule *module;
    IInterfaceTable *interfaceTable;
    IRoutingTable *routingTable = nullptr;
    std::vector<Interface *> interfaces;

  public:
    Node(cModule *module) : Topology::Node(module->getId()) {
        this->module = module; interfaceTable = nullptr; }
    ~Node() { for (size_t i = 0; i < interfaces.size(); i++) delete
        interfaces[i]; }
};

/**
 * Represents an interface in the network.
 */
class Interface : public cObject {
  public:
    Node *node;
    NetworkInterface *networkInterface;

  public:
    Interface(Node *node, NetworkInterface *networkInterface) :
        node(node), networkInterface(networkInterface) {}
    virtual std::string getFullPath() const override { return
        networkInterface->getInterfaceFullPath(); }
};
```

```
    class Link : public Topology::Link {
      public:
        Interface *sourceInterface;
        Interface *destinationInterface;

      public:
        Link() { sourceInterface = nullptr; destinationInterface =
            nullptr; }
    };

    class Topology : public inet::Topology {
      protected:
        virtual Node *createNode(cModule *module) override { return new
            NetworkConfiguratorBase::Node(module); }
        virtual Link *createLink() override { return new
            NetworkConfiguratorBase::Link(); }
    };
```

Listing 1: Definición de topología de red framework INET.

```
void Topology::calculateWeightedSingleShortestPathsTo(Node *_target)
     const
{
    if (!_target)
        throw cRuntimeError(this, "..ShortestPathTo(): target node is
            nullptr");
    auto target = _target;

    // clean path infos
    for (auto& elem : nodes) {
        elem->dist = INFINITY;
        elem->outPaths.clear();
    }

    target->dist = 0;

    std::list<Node *> q;

    q.push_back(target);

    while (!q.empty()) {
        Node *dest = q.front();
        q.pop_front();

        ASSERT(dest->getWeight() >= 0.0);

        // for each w adjacent to v...
```

```
        for (int i = 0; i < dest->getNumInLinks(); i++) {
            if (!(dest->getLinkIn(i)->isEnabled()))
                continue;

            Node *src = dest->getLinkIn(i)->getLinkInRemoteNode();
            if (!src->isEnabled())
                continue;

            double linkWeight = dest->getLinkIn(i)->getWeight();

            // links with linkWeight == 0 might induce circles
            ASSERT(linkWeight > 0.0);

            double newdist = dest->dist + linkWeight;
            if (dest != target)
                newdist += dest->getWeight(); // dest is not the target,
                    uses weight of dest node as price of routing
                    (infinity means dest node doesn't route between
                    interfaces)
            if (newdist != INFINITY && src->dist > newdist) { // it's a
                 valid shorter path from src to target node
                if (src->dist != INFINITY)
                    q.remove(src); // src is in the queue
                src->dist = newdist;
                // the first one will be the shortest
                src->outPaths.erase(std::remove(src->outPaths.begin(),
                    src->outPaths.end(), dest->inLinks[i]),
                    src->outPaths.end());
                src->outPaths.insert(src->outPaths.begin(),
                    dest->inLinks[i]);

                // insert src node to ordered list
                auto it = q.begin();
                for (; it != q.end(); ++it)
                    if ((*it)->dist > newdist)
                        break;

                q.insert(it, src);
            }
            else if (!contains(src->outPaths, dest->inLinks[i]))
                src->outPaths.push_back(dest->inLinks[i]);
        }
    }
}
```

Listing 2: Implementación personalizada algoritmo Dijkstra en el módulo *Topology* de INET.

```
Ipv4Route *Ipv4RoutingTable::findBestMatchingRouteEcmp(Packet *datagram,
    const Ipv4Address& dest)
{

    Enter_Method("findBestMatchingRoute(%u.%u.%u.%u)",
        dest.getDByte(0), dest.getDByte(1), dest.getDByte(2),
        dest.getDByte(3));

    //Get parameters 5 tuple hashing

    const auto& ipv4Header = datagram->peekAtFront<Ipv4Header>();
    Ipv4Address destAddr = ipv4Header->getDestAddress();
    Ipv4Address srcAddr = ipv4Header->getSrcAddress();

    std::string routerName =
        getSimulation()->getContextModule()->getFullPath();

    int protocol = ipv4Header->getProtocolId();
    const Protocol* prot = ipv4Header->getProtocol();

    std::string protName = prot->getDescriptiveName();

    unsigned short srcPort = 0;
    unsigned short destPort = 0;

    try{

        if (protName == "TCP"){
            const auto& tcpHeader =
                datagram->peekDataAt<tcp::TcpHeader>(ipv4Header->getChunkLength());
            srcPort = tcpHeader->getSourcePort();
            destPort = tcpHeader->getDestinationPort();
        } else if (protName == "UDP"){
            const auto& udpHeader =
                datagram->peekDataAt<UdpHeader>(ipv4Header->getChunkLength());
            srcPort = udpHeader->getSourcePort();
            destPort = udpHeader->getDestinationPort();
        } else if (protName == "ICMPv4"){
            const auto& icmpHeader =
                datagram->peekDataAt<IcmpHeader>(ipv4Header->getChunkLength());
            srcPort = icmpHeader-> getType();
            destPort = icmpHeader->getCode();
        } else{
            srcPort = 9999;
            destPort = 9999;
        }

    } catch (cRuntimeError& e) {
        //Falla el chunk. Posiblemente porque llego un paquete de un
            protocolo no manejado layer 3
```

```
    std::cout << "Problema en manejo de paquete layer 3. Protocolo
        no soportado" << std::endl;
}

//Armado del hash

std::hash<std::string> hash;
std::string key = srcAddr.str() + destAddr.str() +
    std::to_string(srcPort)+ std::to_string(destPort) +
    std::to_string(prot->getId()) + routerName;
size_t hashValue = hash(key);

Ipv4Route *bestRoute = nullptr;

range = routingCache.equal_range(dest);
int numPossibleEcmpRoutesCaches = std::distance(range.first,
    range.second);

if (numPossibleEcmpRoutesCaches == 0) {
    for (auto e : routes) {

        if (e->isValid()) {
            if (Ipv4Address::maskedAddrAreEqual(dest,
                e->getDestination(), e->getNetmask())) { // match
                bestRoute = const_cast<Ipv4Route *>(e);

                routingCache.insert (std::pair<Ipv4Address, Ipv4Route
                    *>(dest,bestRoute));

                /* At the edge router the routing table has two
                    options either to route to the client ip
                    address(10.0.0.1) or to the client's subnet
                    address (10.0.0.0) both routes are through the
                    same interface and they have unspecified gateway
                    so if the gateway is unspecified just consider
                    one of the above routes as we need don't want
                    this case to be considered as a multiple paths*/
                if (e->getGateway().isUnspecified()) break;
                /*just take the already found route (e.g. dest addrs=
                    10.0.0.1, gateway=unspecified , Iface = ppp1 )*/
            }
        }
    }

    range = routingCache.equal_range(dest);
    numPossibleEcmpRoutesCaches= routingCache.count(dest);
}

if(numPossibleEcmpRoutesCaches==1) {
    auto it = range.first;
```

```
        return it->second;
    }

    if (numPossibleEcmpRoutesCaches > 1) {
        unsigned int hashValueIII = static_cast<int>(hashValue);

        /*Aca se define si es ECMP per-packet o per-flow. En el caso del
            flow se calcula el modulo entre el valor del hash y la
            cantidad de rutas*/

        std::string ecmp_type = getAncestorPar("ecmp_type");

        int selected = 0;

        if ( ecmp_type.compare("perFlow") == 0 ){
            selected= hashValueIII % numPossibleEcmpRoutesCaches;
        } else {
            selected = ((rand() % numPossibleEcmpRoutesCaches) );
        }

        int i=0;
        for (auto it = range.first; it != range.second; ++it) {
            if ( i == selected) {
                return it->second;
            }
            i++;
        }
    }

    return bestRoute; // shouldn't be reached

}
```

Listing 3: Búsqueda de mejor ruta ECMP en método *findBestMatchingRouteEcmp*.

```
    Node *node = destinationNode;
    Node *nodeTemp = destinationNode;
    Node *nodeCore = destinationNode;
    bool midWay = false;
    int multiplePathsPoint = 0;
    int numShortestPaths = node->outPathEcmp.size();
    int index = 0;

    bool startBranch=true;
    label:
        nodeTemp=node;
```

```
numShortestPaths = nodeTemp->outPathEcmp.size();
for (int moh = 0; moh < numShortestPaths; moh++) {

    Link *link = nullptr;
    InterfaceInfo *nextHopInterfaceInfo = nullptr;

    if (numShortestPaths>1)
        node = (Node *)
            nodeTemp->getPathECMP(moh)->getLinkInRemoteNode();

        while (node != sourceNode) {
            link = (Link *) node->getPathECMP(0);
            if (node != sourceNode && !isBridgeNode(node) &&
                link->sourceInterfaceInfo)
                nextHopInterfaceInfo = static_cast<InterfaceInfo
                                *>(link->sourceInterfaceInfo);

            node = (Node *)
                node->getPathECMP(0)->getLinkInRemoteNode();

            if (node->outPathEcmp.size() > 1 && startBranch==true) {
                startBranch=false;
                goto label;
            }
        }
}
```

Listing 4: Propagación de rutas en método *addStaticRoutes*.

Siglas

anf ANalytic File. 14, 64

API Application Programming Interface. 16, 24, 64

BBR Bottleneck Bandwidth and Round-trip propagation time. 7, 64

BGP Border Gateway Protocol. 2, 64

CE Congestion Experienced. 13, 64

CSV Comma Separated Values. 23, 45, 54, 64

DCN Data Center Networks. 1, 2, 4, 5, 7–10, 12, 13, 19, 22, 23, 26, 29, 34–36, 40–44, 52, 53, 64

DCTCP Data Center TCP. 12, 13, 40, 53, 64

ECMP Equal-cost multi-path. 10–12, 28, 29, 40, 52, 63, 64

ECN Explicit Congestion Notification. 13, 64

ECT ECN Capable Transport. 13, 64

EIGRP Enhanced Interior Gateway Routing. 64

FCT Flow Completition Time. 34, 50, 64

Gbps Giga bit por segundo. 7, 41–44, 64

IDE Integrated Development Environment. 1, 14–17, 19, 20, 22, 34, 36, 40, 64

IP Internet Protocol. 2, 7, 11, 22, 27, 28, 64

JSON JavaScript Object Notation. 23, 64

LSVR Link State Vector Routing. 1, 64

Mbps Mega bit por segundo. 41–44, 47, 64

NED NEtwork Description language. 14, 19, 31, 64

OSPF Open Shortest Path First. 2, 64

PoD Point of Delivery. 1, 6, 7, 31–33, 40, 52, 53, 64

PPP Point to Point Protocol. 2, 22, 64

RED Random Early Detection. 10, 64

RIFT Routing in Fat Trees. 1, 64

RTO Retransmit TimeOut. 8, 64

RTT Round Trip Time. 8, 10, 64

SACK Selective Acknowledgments. 9, 64

sctp Stream Control Transmission Protocol. 22, 64

SFQ Stochastic Fair Queuing. 10, 64

TCP Transmission Control Protocol. 2, 7–13, 22, 33–35, 40, 50, 53, 54, 64

UDP User Datagram Protocol. 2, 22, 34, 50, 54, 64

XML eXtensible Markup Language. 54, 64