UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Evaluación e implementación de técnicas de reenvío

Informe de Proyecto de Grado presentado por

Sebastián Rodríguez

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

Eduardo Grampin
Leonardo Alberro

Montevideo, 29 de diciembre de 2023

Agradecimientos

Quiero agradecer a mis supervisores Eduardo Grampin y Leonardo Alberro por su apoyo y orientación durante el transcurso de este proyecto. Su compromiso y conocimiento, reflejado en las rápidas y acertadas respuestas a mis dudas tanto en las reuniones bisemanales como por correo, fueron esenciales para que este trabajo llegara a buen puerto.

También quiero agradecer a mis padres por su incondicional apoyo durante este año y durante todo mi tiempo en la facultad.

Resumen

Los datacenters son el factor clave que hace posible la computación en la nube. Están compuestos típicamente por gran cantidad de servidores conectados en red, y para que funcione apropiadamente se debe tener en cuenta no solo los servidores que proveen los recursos de cómputo y almacenamiento, sino también la red que los conecta. Por esta razón existen topologías específicas para datacenters. Una topología muy popular para datacenters son los *fat-tree*, los cuales permiten conectar decenas de miles de servidores, y pueden ser construidos con (pocos) miles de switches iguales, lo cual resulta muy económico y conveniente desde el punto de vista del mantenimiento y gestión de la infraestructura.

El objetivo de este proyecto es simple, explorar formas para aprovechar de mejor manera los beneficios de la topología para datacenters *fat-tree*. En particular se buscan técnicas de reenvío aptas para *fat-trees* que permitan utilizar todos los caminos entre servidores en un *fat-tree*.

En este sentido, se realizó un estudio sobre el estado del arte de las técnicas de reenvío de paquetes en topologías *fat-tree* y se decidió trabajar con una implementación en particular, e independiente a la topología, de dichas técnicas: *Equal Cost Multi Path* (ECMP) por flujo y por paquete. Estas técnicas permiten mejorar el uso de los recursos de la red.

Las principales contribuciones de este proyecto son la implementación de una modificación al Kernel Linux que implementa ECMP por paquete, la expansión de un entorno emulado para *fat-trees* basado en Mininet, la creación de un nuevo entorno basado en Máquinas Virtuales para experimentar de forma más controlada sobre ECMP y cubrir algunas carencias del primer entorno, y la creación de diversas pruebas en estos ambientes.

Palabras clave: Técnicas de reenvío, ECMP, Emulación, Linux

Índice general

Capítulo 1

Introducción

Los datacenters proveen acceso flexible y escalable a recursos de cómputo y almacenamiento de forma rentable, necesarios para las necesidades modernas de computación en la nube. Un datacenter típico esta compuesto por decenas de miles de servidores conectados a través de una red grande. Para proveer acceso de calidad a una variedad de aplicaciones y servicios es necesario utilizar los recursos de dicha red de forma eficiente. Aquí entran en juego las topologías de red que utilizan dichos datacenters y cómo utilizarlas de manera eficaz.

Un *fat-tree* (Noormohammadpour y Raghavendra, 2018) es una topología de árbol con múltiples raíces para datacenters. Estas raíces componen la capa core y están conectados con una capa de agregación, la cual a su vez esta conectada a una capa ToR (Top of Rack) la cual conecta a todos los servidores dentro de un rack como se ve en la Figura 1.1. Los *fat-trees* se construyen a partir de switches con k puertos. Un *fat-tree* construido con switches de k puertos soporta hasta $k^3/4$ servidores y $k^2/4$ caminos entre cualquier par de servidores de distintos PoDs (Point of Deliverys).

La motivación para este proyecto es simple, se busca como explotar de mejor manera los beneficios de la topología para datacenters *fat-tree*. En particular se buscan técnicas de reenvío aptas para *fat-trees* que permitan, por ejemplo, utilizar todos los $k^2/4$ caminos entre servidores. Es evidente que un enrutamiento estándar no lo haría, ya que construiría un árbol de cubrimiento mínimo y solo se utilizaría un camino entre servidores. Utilizar $k^2/4$ caminos en lugar de 1 tiene como beneficio que el *throughput* total se puede multiplicar por hasta $k^2/4$.

En este proyecto se realizó un estudio sobre el estado del arte de las técnicas de reenvío de paquetes en topologías *fat-tree* de datacenters. Luego, se trabajó con una implementación particular y agnóstica a la topología de dichas técnicas: *Equal Cost Multi Path* (ECMP) por flujo y por paquete en Linux. ECMP permite mejorar el uso de los recursos de la red mediante el balanceo de carga. Finalmente se compararon ambos tipos de estrategias ECMP en base a diversas pruebas y en diversos entornos.

Se observó que la implementación ECMP de Linux solo proveía ECMP por

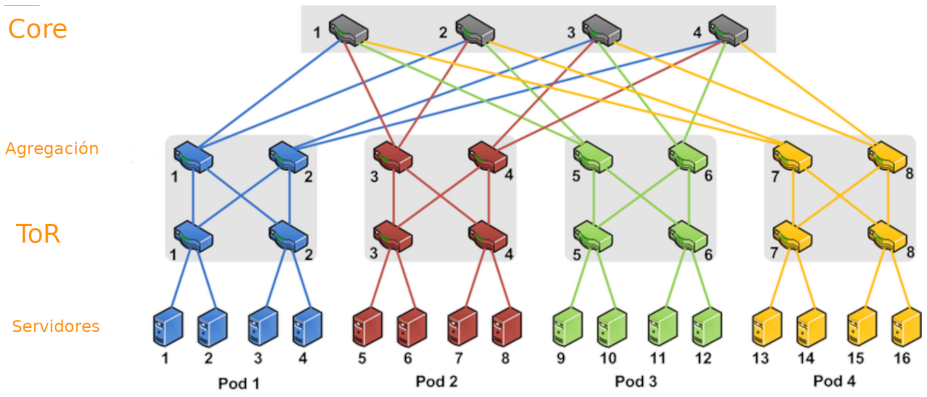

Figura 1.1: Topología fat-tree con k=4, imagen base tomada de (Vardhan, Thomas, Ryu, Banerjee, y Prakash, 2011)

flujo, y no por paquete, por lo tanto, se implementó una modificación al kernel para otorgarle esta funcionalidad, con el objetivo de comparar dichas estrategias.

Inicialmente se trabajó en el ambiente emulado Mininet(Mininet Team, 2023). Dicho ambiente emula la topología de red *fat-tree* para datacenters y fue expandido para este proyecto. Se agregó soporte para ECMP por flujo y por paquete. Se agregaron pruebas automatizados que permiten cambiar parámetros del sistema o de las pruebas y analizar cuál fue el comportamiento del entorno durante las pruebas en forma gráfica o textual. Estas pruebas pueden ser en base a carga uniforme o cargas basadas en datacenters y universidades reales utilizando TrafPy(Christopher W. F. Parsonson, 2023). Además, permite crear y ejecutar conjuntos de pruebas.

Luego de dichas pruebas resultó que el comportamiento del entorno emulado no era el correcto para las estrategias por flujo, pero si para la estrategia por paquete. Debido a este problema, se desarrolló otro ambiente emulado, basado en contenedores, Kathará, pero resulta que el comportamiento inesperado continua de igual forma. Finalmente se desarrolló otro ambiente, basado en máquinas virtuales VirtualBox(Virtualbox Team, 2023), en el cual el comportamiento si fue el esperado. Se sospecha que el problema radica en la herramienta fundamental que estos ambientes utilizan para emular redes, los *network namespaces* (Linux Man Page, 2023a) de Linux.

Finalmente se relevaron las ventajas y desventajas que trae utilizar ECMP por paquete y ECMP por flujo utilizando estos entornos emulados. En pocas palabras ECMP por paquete utiliza de forma casi perfecta los recursos de la red, pero tiene el gran inconveniente que causa el reordenamiento de segmentos TCP el cual causa un gran impacto en el uso de CPU de los endpoints TCP. En comparación, ECMP por flujo funciona idealmente cuando hay muchos flujos pequeños, pero tiene problemas cuando los flujos son heterogéneos y al menos uno de estos flujos es grande.

Se decidió hacer énfasis en ECMP sobre las otras técnicas de reenvío ya que muchos protocolos de enrutamiento como OSPF, ISIS, y BGP proveen soporte para ECMP, es decir, al momento de determinar las tablas de enrutamiento de los routers, computan todos los caminos de igual costo en lugar de solo uno. Esto se traduce en tablas de reenvío con posibles múltiples *next-hop* con mismo costo para un mismo prefijo de red. Adicionalmente, Linux provee una implementación de ECMP una vez dados estos requisitos. Entonces, como este recurso está disponible para explotar y permite mejorar el uso de los recursos de la red resultando en mejor performance, se comprobarán y validarán en un entorno experimental dichos beneficios y su correcto funcionamiento.

Es importante aclarar que ECMP es un primer paso para mejorar la utilización de los recursos de red y que existen muchas otras alternativas que hacen un mejor trabajo en situaciones especiales. En particular, existen muchas alternativas para datacenters con topología *fat-tree*. Si bien se relevan en este informe, tienen el problema que muchas de ellas requieren hardware especializado no disponible e incluso no comúnmente disponible (routers con funcionalidades no comunes, tarjetas FPGA específicas y sin acceso al código), o software específico no disponible, o una combinación de ambas que hace la replicación en un entorno emulado una tarea muy laboriosa. Lo que si existe son implementaciones de estas soluciones en simuladores, pero dichas implementaciones están fuera del alcance de este informe.

Este documento se estructura de la siguiente manera: en el capítulo Revisión de antecedentes, se introducen las distintas tecnologías utilizadas en la creación de los ambientes y se aborda de forma teórica varias técnicas de reenvío de paquetes además de ECMP. A continuación de este, en el capítulo Ambientes de experimentación desarrollados, se describen los ambientes desarrollados y la modificación al kernel de Linux realizada. Luego, en el capítulo Experimentación se muestran algunos de los diversos experimentos realizados en estos ambientes. Por último, en el capítulo Conclusiones y Trabajo Futuro, se concluye sobre todo el trabajo realizado durante el proyecto en general y en qué áreas se puede seguir trabajando en este tema.

1.1. Principales contribuciones

A continuación se enumeran los principales aportes de este proyecto al área de estudio.

1. Creación de una modificación para el Kernel Linux que implementa la estrategia ECMP por paquete y permite elegirla como alternativa a los otros tipos de ECMP implementadas en Linux en tiempo de ejecución.

2. Expansión de un ambiente Mininet existente, permitiendo crear y automatizar pruebas sobre la implementación de Linux de ECMP u otros tipos de pruebas.

3. Creación de un ambiente basado en máquinas virtuales, permitiendo crear

pruebas sobre la implementación de Linux de ECMP en un entorno más realista. También permite realizar otros tipos de pruebas.

4. Creación de diversas pruebas en diversos ambientes que validan el funcionamiento y los beneficios de los varios tipos de técnicas de reenvío basadas en ECMP.

Capítulo 2

Revisión de antecedentes

En este capítulo se introducen las tecnologías utilizadas durante el transcurso de este proyecto para el desarrollo de los ambientes y los experimentos. También se realiza un trabajo sobre el estado del arte de las tecnologías de reenvío, prestándole especial atención a *Equal Cost Multi Path* ya que esta es la técnica con la cual se trabajó experimentalmente.

2.1. Tecnologías utilizadas

A continuación se presentan las tecnologías utilizadas más importantes para este trabajo: Network Namespaces, Mininet, Kathará, Máquinas Virtuales y TrafPy. Adicionalmente también se presenta la topología utilizada en el entorno Mininet: el *fat-tree*.

2.1.1. Fat Trees

Un *fat-tree* es una topología de árbol con múltiples raíces para datacenters. Se construyen a partir de switches con k puertos organizados en tres capas, la capa ToR, la capa de agregación y la capa core. Los switches de la capa core tienen los k puertos conectados a switches de la capa de agregación. Los switches de la capa de agregación tienen k/2 puertos conectados a los switches de la capa core y k/2 conectados a los switches ToR. Los switches ToR tienen k/2 conectados a los switches de agregación y k/2 conectado a los servidores. Como se ve en la Figura 1.1 para k=4, un *fat-tree* construido con switches de k puertos soporta hasta $k^3/4$ servidores y $k^2/4$ caminos entre cualquier par de servidores de distintos PoDs. Una gran ventaja de los *fat-trees* es que pueden soportar una gran cantidad de servidores sin necesidad de utilizar switches con muchos puertos (los cuales son costosos) a diferencia de, por ejemplo, topologías *leaf-spine* (Noormohammadpour y Raghavendra, 2018).

2.1.2. Network Namespaces

Los *network namespaces* de Linux son uno de los Linux *namespaces* que permiten la virtualización de todo el stack de red. Es utilizado tanto por Mininet, como por Kathará(Kathara Team, 2023) a través de Docker(Docker, Inc, 2023). Docker y por extensión Kathará, además utiliza otros *namespaces* para implementar contenedores.

2.1.3. Mininet

Mininet es un emulador de red de código abierto, crea una red de hosts, switches, controladores y enlaces virtuales. Los hosts en Mininet corren software de red estándar Linux, y los switches soportan OpenFlow. También permite que los switches utilicen software estándar de red de Linux. Soporta tareas de investigación, desarrollo, aprendizaje, prototipado, testeo, debugeo y cualquier otra tarea que pueda beneficiarse de tener una red completa experimental en una sola PC o laptop. Entre otras cosas provee una API de Python para crear y configurar hosts, switches, enlaces, pruebas, etc. En Mininet los hosts y los switches son procesos, los cuales tienen distintas interfaces de red, tablas de enrutamiento y tablas ARP gracias a la tecnología de virtualización de red de Linux, *network namespaces*. Estos hosts y switches se conectan utilizando veth(Linux Man Page, 2023b), dispositivos de Ethernet virtuales, otra funcionalidad de Linux.

2.1.4. Kathará

Kathará es una herramienta de emulación de red de código abierto basada en contenedores Docker. En Kathará cada dispositivo de red esta implementado por un contenedor y cada enlace esta implementado como una red virtual. Permite una forma sencilla en un solo archivo de definir una red (los dominios de colisión y los dispositivos). También permite definir scripts bash para que ejecuten al iniciarse los contenedores, y ofrece una consola XTerm por cada contenedor/proceso que representa un dispositivo virtual. La diferencia principal entre las tecnologías que usa Kathará y Mininet es que cada dispositivo Kathará tiene su propio *root file system* (chroot) y están más aislados que en Mininet, donde todos comparten el mismo *file system*.

2.1.5. VirtualBox

VirtualBox es un hipervisor de tipo 2 para virtualización usando procesadores x86 desarrollado por Oracle. Es decir, permite la creación de máquinas virtuales invitadas en un sistema operativo anfitrión. Provee muchas funcionalidades pero en particular fue útil la funcionalidad de crear redes virtuales, asignándole interfaces de red virtuales en estas redes virtuales a las máquinas virtuales.

2.1.6. TrafPy

TrafPy es un paquete de Python para la generación, manejo y estandarización de tráfico de red. En particular interesa el módulo trafpy.benchmarker, que sirve para generar, reproducir y establecer *benchmarks* estándar de tráfico de red. Esto permite generar pruebas de tráfico basado en tráfico real tomado de datacenters proveedores de herramientas en la nube, universidades, entre otros. La generación de *benchmarks* para *fat-trees* resulta particularmente relevante para este proyecto. Estos *benchmarks* se pueden exportar en formato .csv donde cada fila define un flujo entre dos hosts incluyendo el momento de tiempo en el que inicia y su tamaño.

2.2. Otras Tecnologías utilizadas

A continuación se presentan las tecnologías utilizadas por técnicas de reenvío que no se utilizaron experimentalmente: *Software Defined Networking* (SDN) (Wikipedia, 2023a), Shadow MACs (Agarwal, Dixon, Rozner, y Carter, 2014), y Planck (Rasley y cols., 2014).

2.2.1. Software Defined Networking (SDN)

Las redes definidas por software (SDN) son un acercamiento al manejo de redes que permite la configuración de la red de forma dinámica y programática para mejorar la performance y el monitoreo de la red. SDN intenta responder a la arquitectura estática de redes tradicionales, y se puede utilizar para centralizar toda la inteligencia de la red en un componente mediante la separación del reenvío de paquetes (plano de datos) del enrutamiento (plano de control). El plano de control en SDN consiste de uno o más controladores, los cuales son el cerebro de una red SDN y controlan al plano de datos de todos los dispositivos de red conectados. Un problema es que esta centralización tiene desventajas relacionadas con la seguridad y escalabilidad de la red.

2.2.2. Shadow MACs

Shadow MACs es una forma de utilizar direcciones MACs como etiquetas para realizar enrutamiento basado en etiquetas, es utilizado por Planck y Presto (He y cols., 2015).

Shadow MACs describe una forma de utilizar direcciones MAC de destino como labels para realizar *label switching*. Lo que permite a un controlador SDN utilizar las grandes tablas de reenvío de capa 2 ($MAC_d \rightarrow outputport$) que existen en los switches para balancear tráfico, evitar links fallidos o dirigir flujos a través de middleboxes (firewalls, por ejemplo). Argumenta que los switches con soporte MPLS (*Multiprotocol Label Switching*) con suficientes reglas son muy caros y por ende ShadowMACs es/era una forma preferible de realizar *label switching*.

En el core de la red los switches L2 reenvian los paquetes de forma normal (es decir utilizando MAC_d). Excepto que estas MAC_d no son los de ninguna NIC real conectada a la red, sino que por cada NIC real puede haber varios shadowMACs que representan los caminos hacia esta. Estas reglas de reenvío son instaladas en los switches por el controlador de SDN. En el Edge de la red, antes de forwardear los paquetes al core, se reescribe con la ShadowMAC deseada para que los paquetes se muevan por la red por el camino deseado. Luego, antes de ser recibido por el host destino se reescribe la MAC_d a la correcta para evitar que este dropee el paquete.

Como dato interesante, el controlador SDN básicamente utiliza ARP poisoning para modificar la tabla ARP del host el cual tiene un flujo a otro host por una ruta que debe cambiarse. Esto se hace para que envié los paquetes a otra MAC_d y de esta forma los switches reenvían los paquetes a través de otro de los múltiples caminos, ya que están configurados por el controlador para funcionar de esta forma. Finalmente, antes de ser recibido por el host destino esta MAC es reescrita por el ultimo switch. Esto es en parte de lo que consiste "ShadowMACs". El cambio en la tabla ARP y el re enrutamiento del flujo demora entre 2.5 y 3.5ms.

2.2.3. Planck

Planck no es una técnica de reenvío, pero es utilizado para mejorar la efectividad de la técnica de reenvío PAST (Per Address Spanning Tree) (Stephens, Cox, Felter, Dixon, y Carter, 2012), la cual no es relevada directamente en este informe. Aun así, resulta interesante ver hasta qué extremo se puede llegar para tratar de mejorar una técnica de reenvío. Adicionalmente, no solo funciona para la técnica PAST, sino que para cualquier técnica que necesite medir el estado de la red.

Planck es una arquitectura para la medición del estado de la red de forma muy rápida que permite reaccionar rápidamente a la congestión en la red. En particular, es 11x – 18x veces más rápido que otras propuestas en determinar la utilización de todos los links de los switches de la red y *throughput* de todos los flujos TCP que lo atraviesan. 4.2ms (Planck) vs 77.4ms (helios) y 100ms (sFlow) vs 5s (Hedera). Con un porcentaje de error del 3 % en el valor del *throughput* de cada flujo. Y en teoría los autores afirman que si los switches soportaran *port mirroring* sin buffering de salida sería 291x veces más rápido (275–850 μs), ver tabla 1 del paper por más detalles). De esta forma se puede reaccionar muy rápidamente a la congestión en la red.

Utiliza *port mirroring* sobre todos los puertos de cada switch a un mismo puerto para enviar una muestra de los paquetes que atraviesan el switch a un "collector". Hacen esto ya que los métodos disponibles en su momento para realizar un muestreo de paquetes todos necesitaban pasar por la capa de control del switch (sFlow), y con *port mirroring* puede ir directo por la capa de datos la cual esta implementada en ASICS (directo en el hardware) y es mucho más rápida.

En los switches que usaron una vez que se satura el puerto de salida aun así se

envía una muestra representativa de los puertos del switch. Este collector estima varias métricas del switch y luego un controlador de SDN de openflow (floodlight extendido) puede utilizar esta información para modificar las rutas que toman los flujos para utilizar rutas con menor congestión si es necesario. Como routing multipath utilizan PAST y ShadowMACs (Los cuales tienen papers asociados de similares autores).

Cada 14 procesos "collector" se necesita un servidor nuevo (con el hardware que utilizan, procesador de 16 cores y conexión al switch de 10Gbps) para correrlo. Esto es del 2014 entonces hoy en día serian distintos ambos valores. Esto resulta en 0.58 % más dispositivos en un fat tree k = 62.

Finalmente prueban todo esto en 5 switches de 64 puertos conectados de tal forma que simulan un *fat-tree* k=4 con 16 hosts. Y muestran que bajo distintos workloads (shuffle, random, stride, random bijection) el *throughput* promedio usando Planck está a 1 % a 4 % del optimo en todos los casos menos shuffle (12.3 %). Y frente a sus simulaciones similares a Hedera y Devoflow obtienen resultados de *throughput* mejores que varían entre 11 % y 53 % más.

Como conclusión, creo que es muy interesante Planck, pero me parece mucho más un proyecto de investigación que puede influir en cambios en el diseño de switches más que algo que alguien realmente quiera utilizar en un datacenter real. Es un "hack" que muestra lo útil que sería si se pudiera obtener información del estado de los switches (o de los flujos que lo atraviesan) de forma más rápida, o por lo menos un muestreo de forma más rápida. Incluso los mismos autores proponen que se tome en cuenta su trabajo en el diseño de switches y firmware de switches futuro.

2.3. Técnicas de Reenvío

Las técnicas de reenvío refieren a la decisión que toma un nodo en una red de conmutación de paquetes a la hora de decidir por qué puerto debe reenviar un paquete entrante. La técnica indica que debe tener en cuenta para realizar este reenvío, por ejemplo, aspectos del paquete y condiciones globales o locales de la red. También da una pista sobre cómo deben ser los protocolos de enrutamiento para que esta técnica de reenvío sea posible, por ejemplo determinar las tablas de enrutamiento multi-camino es necesario para utilizar una estrategia de reenvío ECMP.

En esta sección se relevan distintas propuestas de técnicas de reenvío para datacenters que intentan algunos o todos los siguientes objetivos: minimizar los *Flow Completion Time* (FCT, tiempo que demora un flujo en terminar) y la *lateness* (tiempo que pasa entre una *deadline* y cuando realmente se termina el flujo), maximizar la utilización de la red, y mejorar la *fairness* (división justa de los recursos de la red entre los distintos usuarios/tenants). También se presentan distintos protocolos o estrategias que son usados por dichas técnicas o que las utilizan para mejorar el uso de la red.

Entre los protocolos y técnicas relevados se encuentran: ECMP, MPTCP(IETF, 2020), Expeditus(Wang, Xu, Niu, Han, y Xiong, 2017), DRB(Cao y cols., 2013),

CONGA(Alizadeh y cols., 2014), NDP(Handley y cols., 2017), Random Packet Spray(Dixit, Prakash, Hu, y Kompella, 2013), y Presto. ECMP para determinar el siguiente salto en su versión por flujo utiliza los cabezales del paquete y en su versión por paquete utiliza *Round Robin* o elección aleatoria. MPTCP es un protocolo de capa de transporte que se beneficia de tener una red que utiliza ECMP por flujo por debajo. Expeditus calcula para cada conexión TCP una estimación del camino óptimo entre los switches ToR de distintos PoDs por el cual transitara esa conexión TCP. DRB sigue una estrategia por paquete y determina por qué camino enviar un paquete en base a determinar hacía que switch core reenviar el paquete. CONGA es un mecanismo de balanceo de carga consciente de congestión diseñado para topologías *leaf-spine* (para ver que existen soluciones para las diversas topologías no solo *fat-trees*). NDP tiene muy buenos resultados, pero requiere de software o hardware especifico tanto en la red como en los hosts para funcionar. RPS utiliza ECMP por paquete e intenta balancear las colas de los switches para que no se produzca reordenamiento. Presto divide flujos grandes en pequeñas cells en el software edge de la red.

Es importante aclarar que en este proyecto se hace énfasis en ECMP para la experimentación.

2.3.1. Equal Cost Multi Path (ECMP)

En un contexto donde ya se realizó la ejecución de los algoritmos de enrutamiento con soporte multicamino (como puede ser OSPF, ISIS, y BGP) y se determinaron los múltiples siguientes saltos (o *next-hops*) válidos para cada prefijo IP asociados a los múltiples caminos que puede haber entre los nodos. *Equal Cost Multi Path* (ECMP) es una técnica de reenvío de paquetes en la cual dados varios *next-hop* válidos para el mismo prefijo en un router determina por cual efectivamente reenviar el paquete usando estrategias por paquete o por flujo. A las estrategias por paquetes se le llaman ECMP por paquete y a las estrategias por flujo se les llama ECMP por flujo. En el RFC 2992(C. Hopps, 2000) se presentan varias formas de implementar tanto, ECMP por paquete como, ECMP por flujo. Dados varios *next-hop* válidos para el mismo prefijo en un router:

- ECMP por paquete utiliza *Round Robin* o *random choice* para determinar el siguiente hop. Por ejemplo, si un switch tiene 4 *next-hop* posibles en 4 puertos distintos para un prefijo y le llegan 10 paquetes de ese prefijo. Entonces en el caso *Round Robin* el primero sale por el puerto 1, el segundo por el puerto 2, el tercero por el puerto 3, el cuarto por el puerto 4, el quinto por el puerto 1, y así sucesivamente. En el caso *random choice* los paquetes salen aleatoriamente por cualquiera de estos puertos con igual probabilidad.
- ECMP por flujo utiliza los cabezales del paquete y aplica alguno de los siguientes métodos *HashTreshold*, o Modulo-n, o HRW para determinar el siguiente hop. Los cabezales usados pueden ser cualquier combinación de mac_s, mac_d, vid, ip_s, ip_d, ip_p, $port_s$, $port_d$ y estos determinan el flujo.

ECMP por paquete tiene la gran ventaja que utiliza los recursos de la red de forma completa, pero tiene la gran desventaja de que puede traer el problema de reordenamiento de segmentos en la capa de transporte lo cual puede confundir al mecanismo de control de congestión de TCP y (más importante si se utiliza una versión de TCP resistente a dicho reordenamiento) causa un gran aumento en el uso de la CPU del host destino en reordenar los paquetes.

En el contexto de ECMP por flujo, en *HashTreshold* se le asignan regiones de igual tamaño al espacio de claves del rango de la función de hash utilizada y la cantidad de regiones esta dado por la cantidad de *next-hop* posibles. Entonces se computa el hash a los cabezales deseados y dependiendo de en qué región caiga el resultado cual salto se utiliza. Esto claramente define un flujo ya que la misma combinación de cabezales retornara el mismo hash. En modulo-n el procedimiento es muy similar, excepto que luego de computar el hash se le aplica modulo-n, donde n es la cantidad de *hops*, y cada región tiene asignado un número entero de 0 a n-1.

En *highest random weight* (HRW), dada una semilla para un generador de números pseudo aleatorios, se calcula un número aleatorio (weight) para cada *next-hop* con la información de los cabezales del paquete y del *next-hop*. Entonces se elige el *next-hop* con el número (weight) más grande correspondiente. Claramente dado una semilla, un conjunto de cabezales con el mismo valor y el mismo *next-hop* el mecanismo siempre retorna el mismo número y por ende define un flujo.

ECMP por flujo es óptimo cuando en el tráfico de la red hay muchos flujos pequeños y estos tienen el mismo tamaño. Pero su desventaja radica cuando, por ejemplo, los cabezales de 2 flujos grandes producen una colisión de hash. Se produce congestión en la red innecesaria que podría ser evitada utilizando otro camino.

En Linux desde el kernel 4.4 se utiliza la estrategia *HashTreshold*, es decir ECMP por flujo. Pero esto no siempre fue así, se agregó en el siguiente commit del 2015(David S. Miller, 2015). El kernel de Linux en un momento tuvo ECMP por paquete, pero esto cambio. Antes del kernel 3.6 el comportamiento multicamino era *"more or less destination-based"*. Luego del kernel 3.6 y antes del kernel 4.4 el multicamino era *"quasi random per paquet scheduling"*. Y desde Linux 4.4 el comportamiento multicamino ahora es basado en flujos, utilizando hashes en los cabezales de los paquetes (IP destino e IP fuente en esta versión), y dejo de utilizar el acercamiento por paquete por problemas de reordenamiento de paquetes(Kernel Newbies, 2016).

Posteriormente se agregó soporte para la selección de distintos cabezales (Nikolay Aleksandrov, 2017). Todas las opciones actuales se pueden encontrar en (Kernel Developers, 2023) bajo las opciones $'fib\_multipath\_hash\_policy'$ y $'fib\_multipath\_hash\_fields'$.

En resumen, siempre se utiliza ECMP por flujo basado en *HashTreshold*, pero $fib\_multipath\_hash\_policy$ puede valer 0, 1, 2 o 3 y se puede configurar por separado para IPv4 e IPv6 utilizando un comando *sysctl*. La diferencia entre estos valores indica a cuáles cabezales del paquete se les aplica el hash. 0 son los cabezales de capa 3, es decir las IPs. 1 son los cabezales de capa 4 y

capa 3, es decir las IPs, los puertos y el protocolo de transporte. 2 es lo mismo que 1 excepto si se detectan mensajes IP encapsulados se utilizan los cabezales internos para determinar el hash en lugar de los externos. 3 implica utilizar los cabezales dados por $'fib\_multipath\_hash\_fields'$

2.3.2. Multipath TCP (MPTCP)

MPTCP es un protocolo de capa de transporte que le permite a un host que lo implementa a comunicarse con otro host que lo implementa a través de más de un subflujo. El ejemplo más directo es con un dispositivo móvil conectado a wifi y a la red telefónica, entonces si se quisiera podría tener un subflujo para cada red. Pero en el caso que nos compete (balanceo de carga dentro de un datacenter) MPTCP permite balancear la carga entre distintos caminos, aunque los hosts no sean *multi-homed* (como un dispositivo móvil), ya que permite crear múltiples subflujos (distintas conexiones TCP) con la misma ip_s e ip_d pero distinto puerto.

Por lo tanto, si la red soporta ECMP por flujo, cada uno de estos subflujos tomaría un camino distinto (dado por un hash de los headers, que incluye los puertos TCP). De esta forma MPTCP intenta maximizar la utilización de la red. Queda claro que si la red no soporta ECMP todos los subflujos tomaran el mismo camino y MPTCP solo agregaría *overhead* innecesario. También sería problemático si la red implementa ECMP por paquete. Adicionalmente, MPTCP debe reordenar los *payloads* de los distintos subflujos para luego pasarlos a la capa de aplicación.

Existe una implementación en el kernel de Linux. MPTCP v1 (RFC 8684) para kernel v5.6 o más nuevo. Es un buen candidato para probarlo en los distintos entornos, pero no se hizo porque el proyecto estaba centrado en técnicas de reenvío (y no protocolos de transporte) y además por falta de tiempo.

2.3.3. Expeditus

Expeditus es un protocolo distribuido, consciente de congestión, para balancear carga en topologías clos(Scott Hogg, 2014) de 3 niveles (*fat-tree* es un caso particular, aunque en los experimentos utilizan *fat-trees*). En Expeditus los switches controlan la utilización de sus *uplinks* (ToR → Agregación, y Agregación → Core). Si a esto se le suma un algoritmo de decisión de camino en dos pasos se obtiene el protocolo Expeditus. Se determina un camino de ida entre 2 switches ToR T1 y T2 en distintos PoDs de la siguiente forma:

1. T1 le envía a T2 la información de congestión de salida de sus *uplinks* y T2 con la información obtenida de T1 más la propia de entrada determina A1 y A2 switches de agregación en sus respectivos PoDs.

2. A1 y A2 realizan un procedimiento similar y determinan el switch core que van a utilizar.

Entonces el camino es T1 → A1 → C → A2 → T2. Y en este proceso se minimizo primero la congestión T1 → A1 + A2 → T2. Y luego se minimizo

A1 $\rightarrow$ C + C $\rightarrow$ A2. Este óptimo no es global, pero, según los autores, en muchos casos se acerca a un esquema que compute este óptimo global y tiene mucho menos complejidad. Adicionalmente este protocolo corre cuando sucede el 3-way handshake de TCP. No causa reordenamiento de paquetes ni *overhead* en la latencia.

Es un protocolo de capa 2 y agrega un cabezal de Expeditus dentro del cabezal Ethernet 802.3 de capa 2. En particular entre la Source Mac y EtherType. Similar a lo que podrían ser las etiquetas 802.1Q para VLANs. Dentro del cabezal de Expeditus el primer campo es TPID y tiene valor 0x9900 que lo identifica, a diferencia de 802.1Q el cual es 0x8100. Se agregan estos tags a los paquetes del handshake TCP para intercambiar información entre los switches y así realizar el proceso en 2 pasos descrito antes.

Durante este proceso se determina flowid como hash de la 5-tupla(ips,puertos, protocolo) y se recuerda en los switches una tabla con la asociación flowid $\rightarrow$ puerto salida. Solo para el *northbound traffic*, ya que el *southbound traffic* es siempre igual dado el switch core. De esta forma para un mismo flujo se utiliza el mismo camino (lo que evita el reordenamiento de paquetes). Es un protocolo muy interesante que a diferencia de ECMP utiliza información del estado de la red para determinar el camino que utiliza un determinado flujo. Lastimosamente no se encontró ninguna implementación en software.

2.3.4. Digit-Reversal Bouncing (DRB)

El paper que presenta Digit-Reversal Bouncing (DRB) utiliza una propiedad interesante de los *fat-trees*. En un *fat-tree*, dado un switch core existe uno y solo un camino entre el host i hacia el host j. Por lo tanto, un switch ToR para determinar el camino a elegir por un paquete en DRB determinan que switch core se quiere usar.

Sea r= *Random Core Switch* cualquiera; M = *Core Switch Count* = $k^2/4$; k= *Amount of Switch ports*; DR la función de Digit Reversal. En DRB el camino de los próximos i paquetes a enviar está dado por el switch core numerado:

$DRB(0) = DR(r)$

$DRB(i) = DR(r + i\ MOD\ M)$

Básicamente le aplican la función DR a lo que sería un *Round Robin* común. Donde Digit Reversal está definido así: dado x en $\{0, 1, 2, \ldots, M-1\}$ con $x = a_1a_2$ siendo a_1 y a_2 los dos dígitos de x en base k/2 entonces $DR(x) = a_2a_1$.

Por ejemplo, para k=4 tenemos 4 switches core y se utiliza base k/2=2 para la función DR. En este caso *Round Robin* da 0, 1, 2, 3 =(en base k/2=2) 00, 01, 10, 11 y DR da 0, 2, 1, 3 =(en base k/2=2) 00, 10, 01, 11.

Esto se hace para evitar el agrupamiento de paquetes que sucede con *Round Robin*. Por ejemplo, en el caso anterior con k=4 utilizando RR, dos paquetes consecutivos utilizan el mismo switch de agregación y luego utilizan dos veces seguidas el otro.

Para enviar un paquete desde el host 'i' al host 'j' a través del switch core 's' determinado por DRB en 'i' se utiliza encapsulamiento IP-in-IP. 'i' encapsula el paquete IP que quiere enviarle a 'j' en otro paquete IP y se lo envía a 's' el cual

luego de desencapsularlo lo envía a 'j'. Además, instala rutas estáticas para que esto funcione correctamente.

También utilizan un protocolo de actualización de topología que permite a los switches distribuir información sobre el estado de sus enlaces y de sí mismos. De esta forma los hosts determinan que caminos son accesibles y no envían paquetes a un switch core inaccesible. De esta forma argumentan que la performance se degrada linealmente a medida que se pierden switches core.

Para lidiar con el hecho de que los algoritmos de balanceo por paquete (como DRB) esparcen los paquetes de un mismo flujo en múltiples caminos, implementaron un algoritmo de re secuenciación de paquetes en los hosts para lidiar con ello. No mencionan uso de CPU de los hosts en los test que hicieron. La mayoría de los tests fue en un simulador NS3 y unos pocos test en hardware real. Pero en ningún caso se menciona el uso de CPU directamente, sino que uno capaz podría inferirlo por el *throughput*, pero esto último dependería de que tan rápido es la CPU y la velocidad de los enlaces (1Gbps no es tanto).

En Presto lo citan a este paper y dicen que su efectividad (en términos de uso de CPU en los end hosts) no está comprobada para redes con enlaces de 10+Gbps (o más). En particular el balanceo de carga a nivel de end host. Pero la idea es muy interesante. Adicionalmente cabe aclarar que no existen implementaciones en software para poder emularlo.

2.3.5. CONGA

CONGA es un mecanismo de balanceo de carga *network-based*, distribuido, y consciente de congestión global para datacenters. Especialmente diseñado para topologías *leaf-spine* y fuertemente resistente a las asimetrías causadas por fallos de enlaces (mucho mejor que ECMP). Necesita de switches específicos con ASICS específicos que lo implementan para funcionar.

La ventaja principal es que CONGA tiene FCTs (*Flow Completion Times*) 5 veces mejores que ECMP cuando hay un fallo en los enlaces y performa entre 2 y 8 veces mejor que MPTCP en casos incast. Esta ventaja sobre ECMP se puede observar (si uno mira la parte de experimentos que hicieron) cuando la carga es máxima y además hay un fallo en un enlace. Entonces, el FCT en ECMP incluye el tiempo que necesita en recuperarse de la falla, el cual es mucho más lento que CONGA. Aunque en casos de pequeños flujos y sin falla de enlaces ECMP es mejor que CONGA y MPTCP para cualquier porcentaje de carga. Aun así, es mejor que CONGA por muy poco en estos casos.

Un flujo TCP se divide en flowlets y un switch leaf determina antes de reenviar el primer paquete (el resto de los paquetes del flowlet siguen el mismo camino y no se toman nuevas decisiones hasta que se genere otro flowlet) de un flowlet:

1. Cuales hosts están conectados a cuáles switches leaf. Rompiendo la convención de solamente conocer el next hop.

2. Un estimado de la congestión (DRE) entre sí mismo y el switch leaf que está conectado al host destino, para cada uno de sus uplinks, pero sin

tomar en cuenta la congestión del *uplink*, solo la del camino. Esto se determina con un protocolo donde se intercambian información entre los switches leaf y en el camino se mide la congestión con asistencia de los dispositivos intermedios. Esta información viaja junto con los paquetes normales (*piggybacked* utilizando encapsulamiento VXLAN)

3. Un estimado de la congestión (DRE) en sus puertos de salida a cada *uplink* local.

Entonces, si un paquete de un host H1 destinado a un host H2 llega a un switch leaf A. Este ya había determinado que H2 está conectado a leaf B. Además, conoce para todos sus uplinks cuanto es la congestión en el camino hacia el leaf B y la congestión del *uplink*. Entonces decide usar el *uplink* con menos congestión en la suma.

CONGA utiliza ASICS para implementar este comportamiento en los switches, y no se encontró una implementación en software.

2.3.6. Novel Datacenter Protocol (NDP)

NDP requiere de ayuda de tanto la red como de los hosts para funcionar. Es decir, necesita de switches específicos y hosts que implementen un protocolo de transporte especifico (NDP). Pero a su favor tiene que provee tiempos de transferencia cortos con poca latencia para flujos pequeños, alto *throughput* para flujos grandes y es resistente al problema de incast.

Las colas en un switch NDP son pequeñas y cuando se llenan, los mismos recortan los paquetes (dejando los cabezales) y los agregan a una cola de alta prioridad en lugar de dropearlos. Existen dos redes lógicas, una de baja prioridad para los datos y otra de alta prioridad para los paquetes de control (ACK, NACKs, PULL requests, paquetes recortados). Esto le provee al host receptor de NDP una visión completa de la demanda de los emisores, y es la base del protocolo de transporte NDP el cual es consciente de los múltiples caminos y puede lidiar con incast asignándole prioridad a los distintos emisores en tiempo RTT. Los switches implementan weighted *Round Robin* entre la cola de alta prioridad y la de baja prioridad con un ratio de 10:1, lo cual define las colas.

En los hosts el protocolo de transporte NDP no tiene handshake inicial, y los flujos son iniciados instantáneamente a velocidad de línea por el emisor, pero luego de esto NDP limita la velocidad de transmisión de todos los emisores incast manteniendo una cola PULL en el receptor que es cargada con nuevos PULL requests a medida que llegan nuevos paquetes (estos PULL request también indican cantidad de paquetes). Adicionalmente, esto también sirve para priorizar que emisor envía paquetes primero, ya que no enviaran paquetes hasta que el receptor se los diga explícitamente con una PULL request.

Los hosts conocen todos los caminos, determinan un orden aleatorio entre ellos y luego envían un paquete por cada camino en ese orden. Luego determinan otro ordenamiento aleatorio y envían un paquete por cada camino en ese orden. Y así sucesivamente. Esto, según los autores, aumenta la capacidad de la red

por hasta 10 % en lugar de realizar *random choice* en los switches y además permite tener colas más pequeñas lo cual baja el delay de transmisión.

Cuando un receptor recibe muchos NACKs asociados a un paquete enviado por cierto camino, temporalmente deja de enviar paquetes por ese camino. En una topología clos la cantidad de NACKs debería ser uniforme entre los caminos, si no lo es podría significar alta congestión en un switch en el camino o una falla en este. Lo cual lo hace resistente a asimetrías causadas por enlaces caídos en la red.

Los autores comentan que su implementación es moderadamente cara en términos de uso de CPU en los hosts debido a la necesidad de realizar *pacing* en los PULLs (tiempo entre que se envía un pull y otro para no sobresaturar los enlaces) y las retransmisiones con baja latencia. Argumentan que estas funciones se podrían implementar en smart NICs y reducir el *overhead* que NDP genera en el uso de CPU (utiliza un core de CPU dedicado para garantizar estas funciones). Por otro lado, no se menciona como maneja el reordenamiento de paquetes, en particular no mencionan cual es el impacto de este en el uso de CPU.

También es interesante notar que la mayoría de las pruebas se hicieron en un simulador y una prueba real con 6 switches FPGA de 4 puertos y 8 hosts organizados en una topología fat tree de 2 niveles. Los hosts están implementados con DPDK (librería para controlar el plano de datos en Linux), los switches tienen 3 implementaciones: con DPDK, en P4 y en tarjetas NetFPGA(NetFPGA Project, 2023).

Los autores tienen un GitHub público con las varias implementaciones, pero, en particular, la implementación de los switches NDP en software DPDK no está funcionando. También la implementación en P4 probaron su correctitud en un switch P4 en software, pero no en un switch P4 físico y no garantizan su funcionamiento. Adicionalmente mencionan que las pruebas en el switch P4 en software de referencia no las incluyen porque los resultados son malos debido a la poca performance de este.

En general es muy interesante este protocolo, pero acceder a una implementación por software para emularlo no fue posible.

2.3.7. Random Packet Spray (RPS)

El método de balanceo de carga de Random Packet Spray hace lo que uno esperaría en base a su nombre. Es básicamente ECMP por paquete con *random choice*. Es decir, dado un paquete a reenviar se reenvía aleatoriamente por uno de los *next-hop* que componen un camino de igual costo, sin importar si este paquete pertenece a un flujo.

Los autores argumentan que el problema evidente que surge cuando uno hace esto (reordenamiento de paquetes y sus consecuencias en TCP) es mucho menor en topologías con árboles multi raíz simétricas (por ejemplo, *fat-trees*) y por ende RPS da buenos resultados (mejor que ECMP por flujo). Fundamentan esto describiendo unas observaciones de forma teórica y (más importantemente) en un experimento con hardware real (links de 1Gbps, 20 tarjetas NetFPGA como switches y 16 hosts consolidando un *fat-tree* k=4).

La primera prueba que hicieron fue dividiendo los 16 hosts en 8 emisores y 8 receptores y ejecutando netperf una vez en cada emisor simultáneamente con un receptor distinto. En esta, se observó que RPS consigue 90 % del *throughput* óptimo con una pequeña varianza entre distintas pruebas, ECMP entre 40-80 % aproximadamente y MPTCP similar a RPS. También probaron reduciendo la ratio de sobresuscripción, limitando la velocidad de los enlaces entre switches core y de agregación. Pasando de 1:1 en el fat tree hacia 4:1 (250Mbps) y 8:1 (125 Mbps). En estos casos RPS funciona mejor (90 %) que ECMP (75 % aprox) y MPTCP (75 % aprox). No saben si esta degradación en MPTCP es debido a la implementación que usaron (Kernel Linux con MPTCP de su momento, en el cual era mucho más experimental) o algo inherente a MPTCP. También dicen que repitieron esta prueba con k=6 y k=8 y aunque haya más reordenamiento de paquetes (ya que al aumentar k aumentan los caminos de igual costo entre hosts, dado por $k^2/4$). Dicen que la perdida en *throughput* no es sustancial (aunque no mencionan exactamente cuanto sería). Esto me da la preocupación de que pasaría con un fat tree k=64 con 1024 caminos, aunque en teoría debería funcionar bien. También este es el único experimento que realizan con distintos valores de k. Tampoco se tiene en cuenta que al aumentar la velocidad de transmisión (por ejemplo, utilizando switches modernos que soporten 100 Gigabit Ethernet) el reordenamiento puede aumentar y el costo de reordenarlo también aumenta.

Observan que RPS funciona bien ya que las colas de los switches en los múltiples caminos de igual costo experimentan casi los mismos niveles de ocupación y por ende casi la misma latencia en un *fat-tree*. Además, los pocos paquetes que si se reordenen TCP (y en particular su implementación en Linux) es robusto a este reordenamiento. Entonces, aunque se produjera más reordenamiento con RPS que ECMP, la perdida extra en *throughput* debido al emisor cortando su ventana de congestión de TCP a la mitad cada vez que pasa un fast retransmit es un pequeño problema comparado con la ventaja de tener una mejor utilización del total agregado de banda ancha entre todos los caminos.

Adicionalmente, observan que la razón por la cual RPS funciona bien (similares niveles de ocupación en las colas en los múltiples caminos de igual costo) se rompe cuando la topología no es simétrica. Esto puede suceder en un datacenter con *fat-trees* si por ejemplo un enlace falla, lo cual es habitual. En estos casos si no se hace nada RPS experimenta una gran caída en performance. Como solución a esto, proponen utilizar estrategias de Active Queue Management (AQM) para mantener los niveles de ocupación de los buffers cola de los switches iguales. Probaron en un experimento similar al anterior, pero fallando un link utilizar la técnica estándar Random Early Detection (RED) y una técnica propuesta por ellos Selective-RED. Esta última activa RED solo para los paquetes de los flujos que causan un diferencial en el largo de la cola de los switches, es decir los flujos cuyos caminos se vieron afectados por la falla del enlace. Esto lo hacen conectando todos los switches ToR o los hosts a un controlador, marcando los paquetes de dichos flujos al ser enviados. Esto en base al conocimiento del controlador de los fallos en la topología. Luego en los switches solo se aplica RED a los paquetes marcados y el resto Droptail normalmente. Luego realizan un

experimento basado en 2 flujos, uno de los cuales es afectado por el fallo de un enlace y comparan la performance de RPS utilizando Droptail, RED y SRED. En general SRED funciona mejor que RED y RED funciona mejor que Droptail. También compara estos 3 casos con MPTCP en un experimento como la primera prueba con sobresuscripción de 4:1 (recordemos que MPTCP funciona peor en 4:1 o 8:1 que 1:1). SRED termina siendo el mejor y con menor varianza en los resultados de *throughput*.

2.3.8. Presto

Presto se basa en la idea que ECMP tiende a ser una solución óptima cuando la cantidad de flujos aumenta y la varianza entre el tamaño de los flujos disminuye (en general sucede cuando hay muchos flujos pequeños similares). Por lo tanto, a nivel de hipervisor de un servidor, es decir en el "software edge" de la red, presto decide partir los flujos grandes de diverso tamaño en múltiples cells, transformándolos en múltiples flujos (cells) pequeños del mismo tamaño. Esto permite (a diferencia de MPTCP) que se instalen VMs customizadas sin un stack MPTCP/IP. Ya que esta división ocurre fuera del sistema operativo host.

Esto claramente es vulnerable al problema de desordenamiento de paquetes en los flujos grandes y debe lidiar con esto igual que MPTCP. Si hay mucho reordenamiento, no solo el mecanismo de control de congestión de TCP se confunde, sino que se utiliza mucho CPU en el host destino reordenando los paquetes.

Para lidiar con esto modifican la implementación de Generic Receive Offload (GRO) del Kernel Linux v3.11.0 para que maneje el reordenamiento de paquetes en una capa inferior a TCP. Se enfocan en hacer esta implementación lo más eficiente posible y utilizan información de control extra (flowcell ID) para lograrlo. Afirman que tiene un *overhead* del 6 % de uso de CPU versus GRO sin reordenamiento al mismo *throughput*. Aunque creo esto dependerá de que tan desordenados lleguen los paquetes.

Utiliza shadow macs para realizar MPLS (lo utiliza como método de packet labeling y luego hace label routing en los switches). Es decir, conecta todos los switches a un controlador que determina los múltiples caminos e instala las tablas de forwading en ellos. Luego, los vSwitches en los hosts balancean la carga enviando paquetes a las distintas shadowMACs de un host destino utilizando *Round Robin*. Adicionalmente, implementa un mecanismo de asignar peso a estos caminos. Por ejemplo, si para un host destino existen 3 shadow macs con pesos: $mac1_d : 0{,}25, mac2_d : 0{,}5, ymac3_d : 0{,}25$. Entonces, para implementar estos pesos hace *Round Robin* entre $mac1_d, mac2_d, mac2_d ymac3_d$.

Presto es un mecanismo de balanceo de carga dinámico proactivo ya que asume que si porciones de tráfico casi uniformes (cells) son balanceadas sobre una topología simétrica, entonces el balanceo de carga puede ser agnóstico a la congestión. Por ende, no toma medidas dinámicas reactivas para controlar el tráfico (como puede ser medir la congestión en los enlaces y cambiar las tablas de reenvío como reacción a la congestión). Este trabajo se relega a capas superiores (como puede ser TCP). Pero, si tiene aspectos dinámicos reactivos

ya que reacciona reactivamente a la caída de enlaces, actualizando las tablas de reenvío tanto en los switches como en los vSwitches.

2.4. Conclusiones

En este capítulo se relevaron las tecnologías más importantes que se utilizaran en los entornos experimentales: Network Namespaces, Mininet, Kathará, VirtualBox y TrafPy. También se relevaron varias técnicas de reenvío como ECMP, Expeditus DRB, CONGA, NDP, RPS y Presto. Además, se relevó MPTCP como protocolo de transporte, y Shadow MACs una estrategia para realizar *label switching* de forma masiva.

Si bien todas estas técnicas son relevantes, se decidió hacer énfasis en ECMP sobre las otras técnicas de reenvío en los experimentos ya que muchos protocolos de enrutamiento como OSPF, ISIS, y BGP proveen soporte para ECMP y adicionalmente Linux provee una implementación de ECMP a diferencia de las demás técnicas.

Es importante aclarar que ECMP es un primer paso muy importante para mejorar la utilización de los recursos de red, pero existen otras alternativas que hacen un mejor trabajo en situaciones especiales, pero todas tienen sus respectivos problemas y razones por la cual no se utilizaron experimentalmente en este proyecto.

En particular existen muchas alternativas para datacenters con topología *fat-tree*. Algunas de las cuales se relevaron en este informe, el problema es que muchas de estas soluciones solo existen en simuladores, o requieren hardware especializado no disponible e incluso no comúnmente disponible (routers con funcionalidades no comunes, tarjetas FPGA específicas y sin acceso al código), o software específico no disponible, o una combinación de ambas que hace la replicación en un entorno emulado una tarea muy laboriosa. Por lo tanto, en los siguientes capítulos se utiliza ECMP en Linux.

Capítulo 3

Ambientes de experimentación desarrollados

En esta sección se describe el desarrollo de una pequeña modificación para el Kernel Linux que incluye la funcionalidad de ECMP por paquete que será utilizada en los siguientes entornos. También se describe la expansión del entorno Mininet para *fat-trees*, el desarrollo desde cero de un entorno Kathará y un entorno de máquinas virtuales con una topología reducida para experimentar con ECMP debido a las limitaciones de los otros entornos.

3.1. Modificación del Kernel Linux

El kernel desarrollado agrega ECMP por paquete al kernel a través de la interfaz *sysctl* (utilidad de software de Linux que lee y modifica atributos del kernel). Dicha interfaz es expuesta a través de *procfs*(sistema de archivos especial de Linux que presenta información sobre el sistema en una estructura de archivos jerárquica, montado en */proc*).

Entonces el archivo $/proc/sys/net/ipv4/fib\_multipath\_hash\_policy$ permite el nuevo valor: 4, lo cual hace que la resolución de hash para el reenvío ECMP siempre retorne un valor aleatorio en lugar de utilizar los cabezales del paquete como en 0, 1, 2 o 3, efectivamente implementando ECMP por paquete aleatorio lo cual va a ser muy útil para los posteriores experimentos. En resumen, si la variable $fib\_multipath\_hash\_policy$ vale:

- 0, el kernel utiliza ECMP por flujo basado en *HashThreshold* y la entrada al hash son los cabezales de capa 3, es decir las IPs.
- 1, el kernel utiliza ECMP por flujo basado en *HashThreshold* y la entrada al hash son los cabezales de capa 4 y capa 3, es decir las IPs, los puertos

y el protocolo de transporte.

- 2, el kernel utiliza ECMP por flujo basado en *HashThreshold* y es equivalente a usar 0 excepto si se detectan mensajes IP encapsulados se utilizan los cabezales internos para determinar el hash en lugar de los externos
- 3, el kernel utiliza ECMP por flujo basado en *HashThreshold* y la entrada al hash son los cabezales dados por $'fib\_multipath\_hash\_fields'$ que se pueden configurar por separado.
- 4, el kernel (modificado) utiliza ECMP por paquete *random choice*. Es decir, cada paquete que llegue a un router determina por qué siguiente salto reenviar el paquete, entre todos los válidos para el prefijo IP correspondiente a la IP destino del paquete, de manera aleatoria.

En el anexo A.1 están disponibles las instrucciones para realizar las modificaciones pertinentes, construir el kernel con dichas modificaciones, instalarlo y ejecutarlo. Dejando de lado el anexo, hablaremos aquí solamente de las modificaciones, las cuales son:

En *net/ipv4/route.c* agregar una nueva opción al switch para que el hash del paquete 'mhash' sea aleatorio:

```
switch (READ_ONCE(net->ipv4.sysctl_fib_multipath_hash_policy)):
(...)
    case 4:
        get_random_bytes(&mhash, sizeof(mhash));
        break;
```

En *net/ipv4/sysctl_net_ipv4.c* modificar el rango de valores de fib_multipath_hash_policy para que 4 sea un valor valido y se pueda cambiar en tiempo de ejecución:

```
{
    .procname        = "fib_multipath_hash_policy",
    (...)
    .extra2                 = &four,
},
```

Las instrucciones completas para obtener el código fuente, construir luego de hacer estos cambios e instalar el nuevo kernel están en el anexo mencionado. Los siguientes entornos fueron configurados para utilizar este kernel desarrollado.

3.2. Descripción de los Entornos

Incluye las herramientas y tecnologías en las que se basan, su capacidad para realizar distintos tipos de pruebas y las respectivas limitaciones. Adicionalmente se detalla que se agregó en este proyecto de grado para el caso del entorno Mininet. Por otro lado, siempre que se menciona una cantidad específica de paquetes en cualquiera de los entornos esto significa que se marcaron paquetes con el campo TOS del cabezal IP(Wikipedia, 2023b) y luego se contaron dichos paquetes.

3.2.1. Mininet

El entorno Mininet que se tomó como base(Leonardo Alberro, 2022) permitía crear *fat-trees* virtuales para cualquier k y además tenía resuelto el problema de enrutamiento utilizando BGP, incluso determinaba las tablas de enrutamiento con múltiples *next-hop* aunque no las utilizaba.

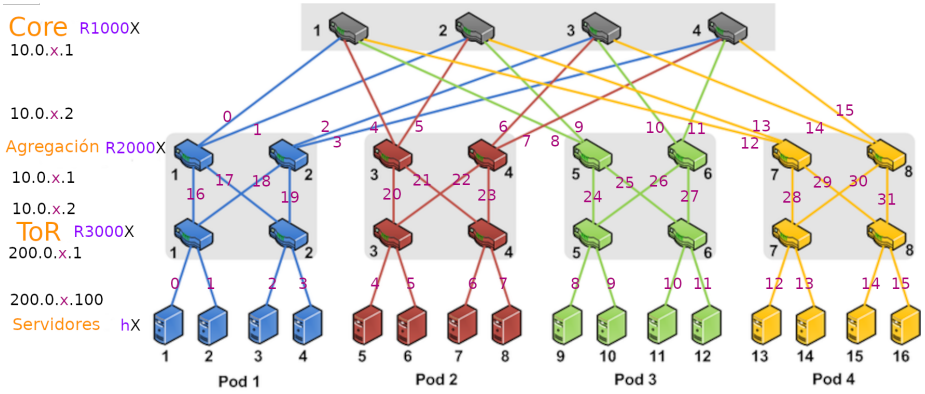

Figura 3.1: Topología del entorno Mininet, con k=4, imagen base tomada de (Vardhan y cols., 2011)

La Figura 3.1 muestra el entorno Mininet en cuestión para k=4, se observan las 4 capas de la topología: Core, ToR, Agregación, y Servidores. Adicionalmente, se pueden ver las IPs de cada nodo en cada subred y los nombres de los nodos. Por ejemplo, el router de más arriba a la izquierda es un router en la capa core, con nombre R10001, y tiene 4 IPs (10.0.0.1, 10.0.4.1, 10.0.8.1, 10.0.12.1) cada una asignada a una interfaz distinta y corresponden a PoDs distintos.

Ahora el entorno permite más variables de configuración, principalmente seleccionar el tipo de ECMP que se desea utilizar, establecer un límite para la banda ancha de los enlaces, y decidir si graficar el tráfico de la red. Además, se solucionó un problema importante que tenía, al obtener con la API de Python las IPs de los nodos estos retornaban una IP incorrecta, el problema también ocurría al hacer un dump de la red utilizando la CLI de Mininet, ahora muestra la IP correcta.

La otra gran adición realizada es la posibilidad de utilizar este entorno para correr pruebas parametrizadas. Por ejemplo, una prueba permite realizar una conexión entre cada host con cualquier otro host utilizando la herramienta iperf3(ESnet, 2023). Se permite variar si el test utiliza UDP o TCP, si transmite por cierto tiempo o transmite una cantidad de bytes y cuál es el *bitrate* objetivo.

También permite realizar una prueba similar a la anterior pero en lugar de tener una conexión entre cada par de hosts tenemos que todos los hosts se conecten al mismo, simulando una situación de incast. Esta permite las mismas variables.

Como resultado se obtiene de cada host el promedio de bytes que transmitió, el *bitrate* promedio al cual transmitió, el tiempo promedio de transmisión y el uso de CPU. Aunque este último resultó no ser muy confiable en el entorno emulado. Opcionalmente, si así se lo desea, se puede marcar (con el cabezal tos de IP) y capturar los paquetes que se transmitieron y graficar el uso de la red. Aunque en esta última opción hay que tener cuidado que la suma de las velocidades de transmisión no supere la velocidad de escritura a disco.

Para probar los distintos tipos de ECMP uno realizaría varias instancias de estas pruebas cambiando los parámetros de las pruebas, cambiando el tipo de ECMP y observando los resultados. Esto se puede hacer creando un archivo con estos parámetros en formato json para cada prueba, luego se puede correr la prueba basada en dichos parámetros y adicionalmente se provee la funcionalidad de correr varias pruebas con dicho formato en forma secuencial. Permitiendo, por ejemplo, realizar tests de regresión.

Por otra parte, se creó un script de Python para generar *benchmarks* de TrafPy en formato csv para *fat-trees*, con k y la velocidad de transmisión de los hosts como parámetros. Luego, se desarrolló un *scheduler* que lee los csv y genera una conexión entre los hosts indicados, con la cantidad de tráfico indicada y en el momento de tiempo indicado. Esta conexión se implementa con iperf3 al igual que las demás conexiones de las otras pruebas. También existe la posibilidad de realizar una conexión extra durante la ejecución de este *benchmark* y marcándola utilizando el cabezal tos de IP, esto permite estudiar el comportamiento de una conexión en particular cuando existe tráfico realista de fondo.

Finalmente, como Mininet utiliza el kernel subyacente para emular la red y proveerle a los nodos emulados capacidades de red, es suficiente cambiar el kernel que se está utilizando en el sistema operativo e instalar Mininet en dicho sistema.

3.2.2. Kathará

El entorno Kathará surgió debido a que en el entorno Mininet se observó un comportamiento extraño durante la experimentación. Para determinar si esto es debido a la implementación de ECMP de Linux o al entorno Mininet, se decidió utilizar este entorno. En particular se utilizó Kathara ya que es una herramienta de emulación de red conocida. Finalmente, resultó tener el mismo comportamiento, pero esto nos acerca un poco más a explicar el comportamiento observado.

Se definió un laboratorio Kathará con la topología de la Figura 3.2. En rojo se pueden observar los dominios de colisión, en verde las interfaces de los nodos, y en azul los rangos de IPs de las subredes y las IPs de los nodos. Se configuró manualmente en scripts de inicio (*startup*) para cada nodo las IPs de sus interfaces y las rutas multicamino. Luego, se realizan conexiones entre h1 y h2, y mediante capturas de tráfico en r21 y r22 se determina que camino tomó cada conexión.

Este es un entorno más sencillo ya que no es un *fat-tree* a diferencia del entorno Mininet, pero aun así es muy útil para analizar el funcionamiento de

ECMP y determinar si se comporta extrañamente o no. En particular, con los experimentos en este entorno se confirmó la hipótesis de que estaba causando el comportamiento extraño.

Para esto, fue necesario modificar una imagen de Linux de Docker agregándole utilidades como iperf3 y tcpdump para utilizarla en Kathará como imágenes para los nodos. Por otro lado, similar al caso de Mininet, Docker utiliza el kernel subyacente para virtualizar la red, y las imágenes no tienen su propio kernel. Por lo tanto, es suficiente instalar Kathará en un sistema operativo que el kernel modificado si se quiere probar ECMP por paquete en este entorno.

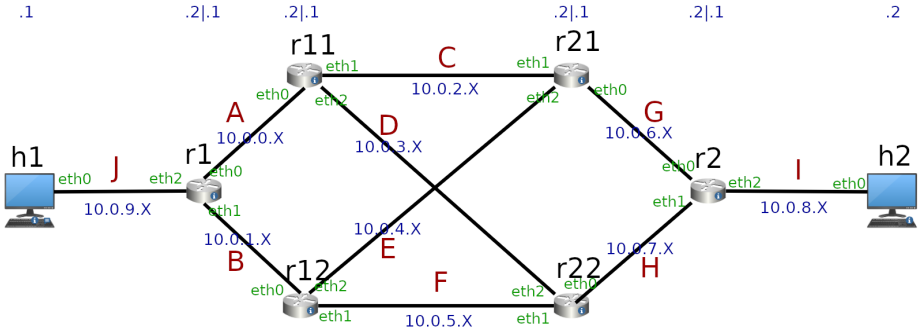

Figura 3.2: Topología de los entornos Kathará y Máquinas Virtuales

3.2.3. Máquinas virtuales

La necesidad de tener un entorno de Máquinas Virtuales surgió porque el entorno Kathará seguía teniendo problemas y se necesitó recurrir a un entorno aún más virtualizado para que ECMP funcione como debería. Se recreó la topología de la Figura 3.2 con máquinas virtuales en VirtualBox, donde cada nodo de la topología es una máquina virtual y cada dominio de colisión es una red interna dada por VirtualBox. Además, cada máquina virtual utiliza una imagen de Ubuntu 22.04.2 con el kernel modificado. La única diferencia con la Figura 3.2 es que las interfaces se llaman distinto, eth0 es enp0s8, eth1 es enp0s9 y eth2 es enp0s10. Cabe aclarar que además se utiliza otra interfaz para poder conectarse por ssh y ejecutar los scripts correspondientes, una limitación importante de este entorno es que VirtualBox limita la cantidad de interfaces que puede tener cada VM a cuatro, por lo tanto, no se podría expandir más la topología presentada ya que la mayoría de los nodos utilizan el límite de interfaces.

Se configuró de tal forma que cada máquina al iniciar corra un script que configura las interfaces con las IPs correctas, y asigna las tablas de enrutamiento multicamino correctas a cada host. Estos scripts se pueden copiar todos juntos ejecutando un solo script, permitiendo cambiar rápidamente los scripts de inicio de todos los hosts.

Por otro lado, se crea una carpeta adicional por cada test que se desee hacer, en esta carpeta se coloca un script con el nombre del nodo por cada nodo. Posteriormente, un script principal ejecuta dichos scripts en el siguiente orden: primero los nodos r1, r11, r12, r21, r22 y r2 para determinar el tipo de ECMP que se quiere probar y para iniciar las capturas correspondientes. Después se inicia el o los servidores iperf3 en h2 y finalmente se realizan las conexiones deseadas desde h1 a h2. Luego, este script recolecta las capturas y analiza los resultados utilizando tshark(The Wireshark Team, 2023) para contar paquetes, y de esta forma determinar que caminos tomaron los paquetes.

3.3. Conclusiones

En este capitulo se detalló el desarrollo de la modificación al Kernel Linux que re-agrega ECMP por paquete a Linux, permitiendo configurar esta opción en tiempo de ejecución (sin reiniciar el kernel). Posteriormente, se utilizó este kernel modificado en los distintos entornos.

También se describió la expansión del entorno Mininet para *fat-trees* agregándole soporte para ECMP, pruebas en base a carga uniforme, cargas basadas en datacenters y universidades reales utilizando TrafPy, conjuntos de pruebas automatizados que permiten cambiar parámetros del sistema o de las pruebas, graficar el tráfico durante las pruebas y computar estadísticas de las conexiones usadas en las pruebas. Aunque se detectó un comportamiento extraño de ECMP en dicho ambiente.

A continuación, se detalló un entorno más sencillo basado en una herramienta conocida: Kathará, para intentar diagnosticar el problema con el comportamiento de ECMP. Este entorno presenta el mismo problema que Mininet, pero sirvió para determinar la razón del problema.

Finalmente, en el entorno de máquinas virtuales se determinó que el problema eran los Linux Namespaces y no la implementación de ECMP de Linux ya que, en máquinas virtuales, donde la simulación de red no es hecha por el mismo kernel y cada host tiene su propio kernel, funciona como se espera. Este es el único ambiente emulado que muestra el correcto funcionamiento de ECMP. Por lo tanto, las pruebas más justas, y con menos sesgo, van a ser en este entorno, ya que el comportamiento extraño de ECMP en los otros entornos causa un problema grande en la performance y sería injusto comparar ECMP por flujo y ECMP por paquete en dichos entornos hasta que este problema no sea solucionado. Tiene la gran desventaja que es el entorno que escala menos.

Capítulo 4

Experimentación

En este capítulo se detallan los distintos experimentos que se realizaron en los distintos entornos desarrollados, los resultados obtenidos y algunas conclusiones sobre ECMP.

4.1. Mininet

Experimentalmente el entorno Mininet se utilizó dentro de una VM con Ubuntu 22.04.2 corriendo el kernel modificado.

4.1.1. Pruebas con tráfico todos a todos

Dentro del conjunto de pruebas donde todo par de host intercambian paquetes, se realizaron tres pruebas. En la primera se utiliza ECMP por flujo utilizando para los hashes los cabezales de capa 3($fib\_multipath\_hash\_policy$=0), como se ve en la figura 4.1. En la segunda se utiliza ECMP por flujo utilizando para los hashes los cabezales de capa 4($fib\_multipath\_hash\_policy$=1), como se ve en la figura 4.2. Finalmente, en la tercera se utiliza ECMP por paquete($fib\_multipath\_hash\_policy$=4), como se ve en la figura 4.3. En las imágenes, en rojo están los nodos, en gris los enlaces, y en azul la cantidad de paquetes que se transmitieron por ese enlace. Se observa que en todos los casos la cantidad de paquetes capturados cuadra con el total de paquetes enviados, cada switch ToR recibe o envía $616 = 11 * 14 * 2 * 2$ paquetes a través de sus *uplinks*(11 paquetes por 14 conexiones en ambos sentidos y por dos hosts conectados a él). En la imagen no se registra pero por definición del problema cada host transmite $330 = 11 * 15 * 2$ paquetes (11 paquetes por 15 conexiones en ambos sentidos). Estos números también cuadran con la cantidad de paquetes recibidos en los switches ToR. Por lo tanto, se esta seguro de la correctitud de la implementación de las pruebas todos a todos.

Para los casos de ECMP por flujo capa 3 y capa 4 no se observan diferencias relevantes. Esto sucede ya que al haber solo una conexión entre cada par de

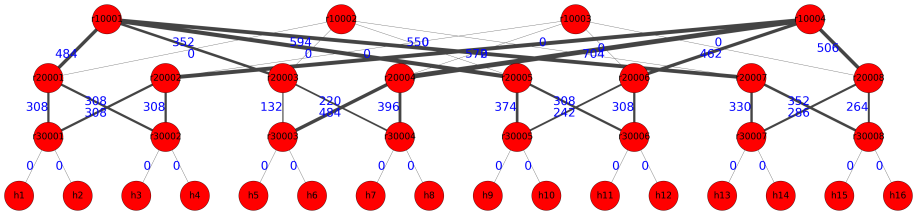

Figura 4.1: Una conexión con once paquetes entre todo par de hosts utilizando ECMP por flujo capa 3, sobre un *fat-tree* k=4.

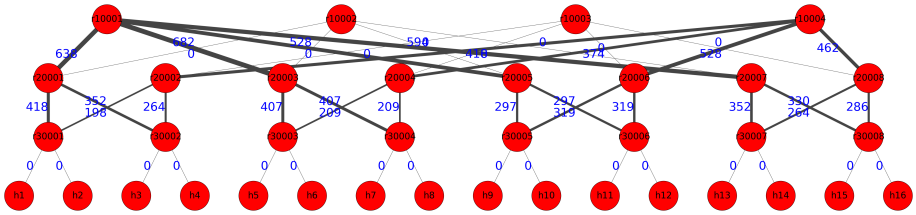

Figura 4.2: Una conexión con once paquetes entre todo par de hosts utilizando ECMP por flujo capa 4, sobre un *fat-tree* k=4.

nodos esto significa que el hash de la quíntupla IPs, puertos, protocolo, es igual de aleatorio que el par de IPs. Para que se observen las ventajas de ECMP por flujo capa 4 sobre capa 3 debe haber muchas conexiones entre el mismo par de hosts. Entonces todo lo que se diga sobre ECMP por flujo capa 4 en esta parte también aplica para capa 3. La única diferencia es si se rehace la prueba en ECMP por flujo capa 4 los resultados varían un poco ya que los puertos varían, mientras que en capa 3 las IPs no varían y por ende los hashes siempre dan igual (hasta que se reinicie el kernel lo cual cambia la semilla del hash).

Se observa que los nodos en la capa ToR balancean los paquetes entre los dos posibles puertos de salida, pero los nodos en la capa de agregación no. Es más, pareciera que el primer salto determina al segundo, es decir si un switch en la capa ToR decidió reenviar un paquete por su segundo puerto, entonces el switch en la capa de agregación también lo reenvía por su segundo puerto. Este es el comportamiento extraño mencionado en el capítulo anterior y como consecuencia divide la cantidad de caminos posibles en 2 (k/2) y por lo tanto los resultados numéricos de estos experimentos emulados no representarían correctamente a los de un escenario real para ECMP por flujo en un *fat-tree*. En los siguientes ambientes se confirma esta hipótesis, se determina porque sucede y se obtiene un ambiente donde el fenómeno no ocurre.

Dejando de lado este problema, se observa que con ECMP por paquete si se balancean los paquetes en ambas capas del *fat-tree*, obteniendo un resultado

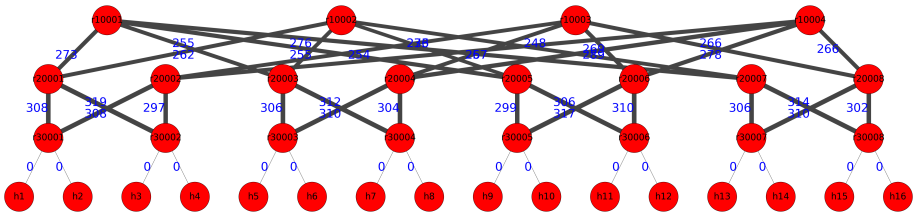

Figura 4.3: Una conexión con once paquetes entre todo par de hosts utilizando ECMP por paquete, sobre un *fat-tree* k=4.

que utiliza todos los caminos del *fat-tree* de forma homogénea. Adicionalmente se puede ver que el experimento funciona correctamente para todos los tipos de ECMP, en el sentido que todo par de hosts intercambian paquetes, tal como se diseñó el experimento, aunque solo ECMP por paquete funciona como debería.

4.1.2. Pruebas con tráfico todos a uno

Por otra parte, dentro del conjunto de pruebas donde todo host le envía paquetes a h1 (Figura 4.4, Figura 4.5 y Figura 4.6), emulando una situación de incast, se observan resultados similares al escenario todos a todos. Es decir, no hay mucha diferencia entre ECMP por flujo capa 3 y capa 4 y ambos continúan con el fenómeno descrito anteriormente lo cual impide el correcto balanceo de los paquetes. Además, ECMP por paquete si balancea correctamente los paquetes.

Por otro lado, el experimento hace lo que está diseñado para hacer, tener una conexión entre cada host y h1, con el problema de que ECMP por flujo no funciona correctamente. Se observa que en el switch ToR r30001 la cantidad de paquetes capturados cuadra con el total de paquetes enviados, r30001 recibe $154 = 11 * 14$ paquetes a través de sus *uplinks*(11 paquetes por 14 conexiones). En la imagen no se registra pero por definición del experimento cada host transmite 11 paquetes. Estos números también cuadran con la cantidad de paquetes recibidos en los switches ToR. Por lo tanto, se esta seguro de la correctitud de la implementación de la prueba todos a uno. Finalmente, cabe aclarar que ECMP no se comporta de forma distinta en una situación de incast como esta.

4.1.3. Pruebas con TrafPy

Finalmente, ECMP por paquete suena como la solución ideal que utiliza todos los caminos del *fat-tree*, pero esto no es así ya que tiene un gran problema: el reordenamiento de segmentos TCP. Por lo tanto, para ver esto se utilizó tráfico de fondo dado por un *benchmark* de datacenter cloud provisto por TrafPy, y en el medio de este tráfico se realizó una conexión TCP de 10MB entre h1 y h16. Esto se hizo con las tres opciones de ECMP y el resultado en Mininet fue que no hubo reorden alguno en ninguno de los casos. Este resultado es un poco

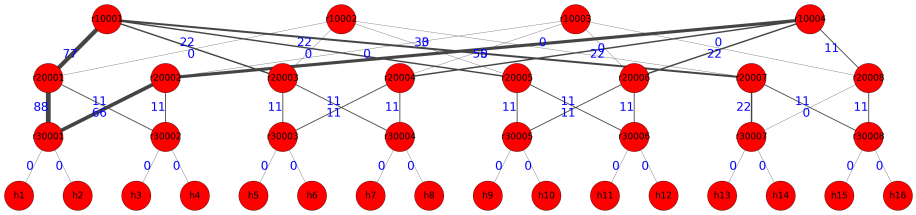

Figura 4.4: Una conexión con once paquetes entre todos los hosts y h1 utilizando ECMP por flujo capa 3, sobre un *fat-tree* k=4.

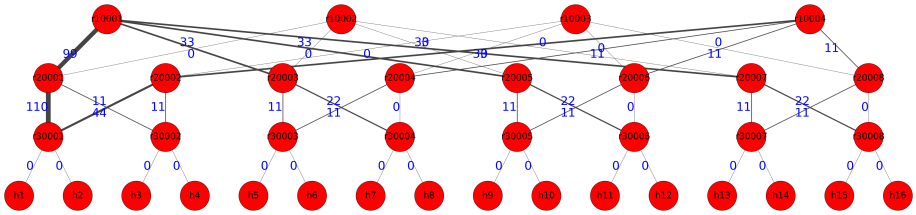

Figura 4.5: Una conexión con once paquetes entre todos los hosts y h1 utilizando ECMP por flujo capa 4, sobre un *fat-tree* k=4.

sorprendente pero no tanto, ya que el entorno emulado Mininet parece emular una situación ideal en donde el reordenamiento de paquetes no ocurre, es decir los delay de todos los caminos son iguales y por ende no hay reordenamiento alguno. En una situación real esto no funcionaría así, basta que una cola de un switch presente en un camino este más llena que otra en otro camino para que esto no suceda. Más adelante se verá como a en el entorno de máquinas virtuales que ECMP por paquete si hay reordenamiento. Esto no quita que ECMP por paquete podría no causar reordenamiento en situaciones ideales.

4.2. Kathará

El experimento más importante que se hizo en Kathará confirma que el comportamiento extraño está ocurriendo fuera de Mininet. Es decir, el primer salto multicamino determina al segundo salto multicamino. En este entorno en particular y si hablamos de un paquete viajando de h1 a h2 esto significa que el primer salto de r1 a r11 o de r1 a r22 determina el segundo salto de r11 a r22, o r11 a r21, o r12 a r21, o r12 a r22. Entonces, la configuración en r1 determina el comportamiento de r11 o r12, lo cual rompe la virtualización que está utilizando este entorno.

En pocas palabras si tenemos un paquete viajando de h1 a h2, y r1 decide

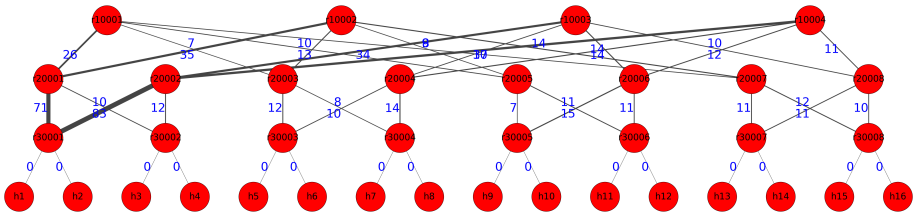

Figura 4.6: Una conexión con once paquetes entre todos los hosts y h1 utilizando ECMP por paquete, sobre un *fat-tree* k=4.

reenviarlo a su primera opción r11 (luego de computar un hash sobre los cabezales del paquete), entonces r11 va a reenviarlo a su primera opción r21. Lo que está sucediendo acá es que tanto r11 como r21 están computando el mismo hash, ya que utilizan la misma semilla (o sal) de hash y esto resulta en que ambos hashes caigan en la primera región.

Para ver esto experimentalmente, es necesario tener en cuenta las configuraciones de rutas multicamino de r1, r11 y r12 con destino la subred de h2. Con la siguiente configuración:

```
-------------------------------------r1---------------------------------------------
ip route add 10.0.8.0/24 nexthop via 10.0.0.2 weight 1 nexthop via 10.0.1.2 weight 1
 ------------------------------------r11--------------------------------------------
ip route add 10.0.8.0/24 nexthop via 10.0.2.2 weight 1 nexthop via 10.0.3.2 weight 1
 ------------------------------------r12--------------------------------------------
ip route add 10.0.8.0/24 nexthop via 10.0.4.2 weight 1 nexthop via 10.0.5.2 weight 1
```

Si enviamos 50 paquetes de h1 a h2 con un puerto fuente distinto, y usando ECMP por flujo con hashes en capa 4, el resultado que obtenemos es que los paquetes solo utilizan dos caminos r1-r11-r21-r2(23 paquetes), o r1-r12-r22-r2 (27 paquetes) como se ve en la Figura 4.7. Esto sucede para r1-r11-r21-r2 ya que si vemos con cuidado la configuración 10.0.0.2 es la primera opción de r1 y corresponde al bridge con r11, y además 10.0.2.2 es la primera opción para r11 que corresponde con r21. Y sucede para r1-r12-r22-r2 ya que 10.0.1.2 es la segunda opción de r1 y corresponde con r12, y además 10.0.5.2 es la segunda opción de r12 y corresponde con r22. Ahora, si cambiamos la configuración de r1 tal que intercambiamos sus opciones para el siguiente salto, de la siguiente forma:

```
-------------------------------------r1---------------------------------------------
ip route add 10.0.8.0/24 nexthop via 10.0.1.2 weight 1 nexthop via 10.0.0.2 weight 1
```

Sucede que el resultado que obtenemos es que los paquetes solo utilizan los otros dos caminos, r1-r11-r22-r2(29), o r1-r12-r21-r2(21) como se ve en la Figura 4.8. Para r1-r12-r21-r2 esto sucede ya que ahora la primera opción de r1 corresponde con r12, y la primera opción de r12 corresponde con r21. Y para r1-r11-r22-r2 esto sucede ya que la segunda opción de r1 corresponde con r11 y la segunda opción de r11 corresponde con r22.

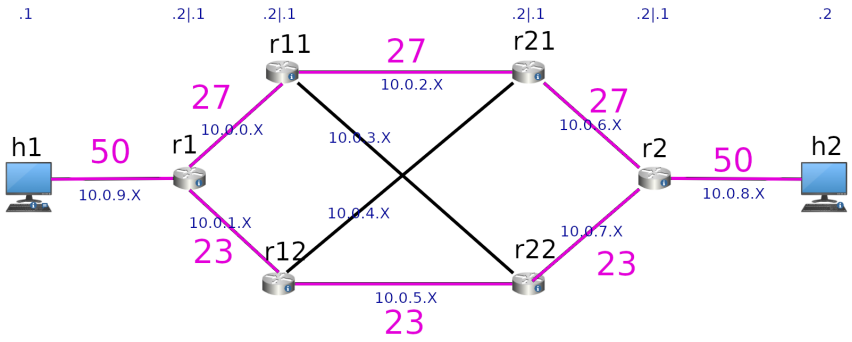

Figura 4.7: Prueba con la primera configuración del entorno Kathara, donde la primera opción de r1 es r11 y la primera opción de r11 es r21. Adicionalmente, la segunda opción de r1 es r12 y la segunda opción de r12 es r22. Se envían 50 paquetes con puertos origen distintos desde h1 a h2, ECMP por flujo capa 4.

Este comportamiento se explica por un error en *network namespaces*, es decir en la virtualización de red que utilizan tanto Mininet como Kathará (Docker). Pero otra forma de tratar de explicar este fenómeno sería asumir que la implementación de ECMP de Linux no utiliza una semilla de hash, o que esta está hardcodeada y siempre es igual (en la realidad esta semilla se randomiza cada vez que se reinicia el kernel). Entonces para desmentir esta hipótesis experimentalmente se requiere utilizar el siguiente entorno, de máquinas virtuales, donde se observa que ECMP funciona correctamente. Por lo tanto, lo que está sucediendo es que tanto Mininet como Kathará no virtualizan esta semilla, entonces todos los nodos equivocadamente comparten la misma semilla. Con esto se concluye que existe una limitación de la virtualización de los *network namespaces*.

4.3. Máquinas virtuales

En el entorno de máquinas virtuales se realizaron varias pruebas sobre ECMP. La primera prueba es igual a la prueba en Kathará pero en este entorno. Las próximas cuatro pruebas se realizan 3 veces variando los tipos de ECMP entre por flujo capa 3, por flujo capa 4 y por paquete *random choice*. En la segunda prueba se envían 100 paquetes UDP en una sola "conexión" entre h1 y h2 (es decir todos los paquetes tienen la misma quíntupla IP, puerto, protocolo). En la tercera prueba se envían 100 paquetes UDP en 10 conexiones distintas entre h1 y h2. En la cuarta prueba se envían 100 paquetes UDP en 100 conexiones distintas entre h1 y h2. En la quinta prueba hay una "conexión" grande de 100 paquetes y 10 'conexiones' pequeñas de 1 paquete entre h1 y h2. Finalmente,

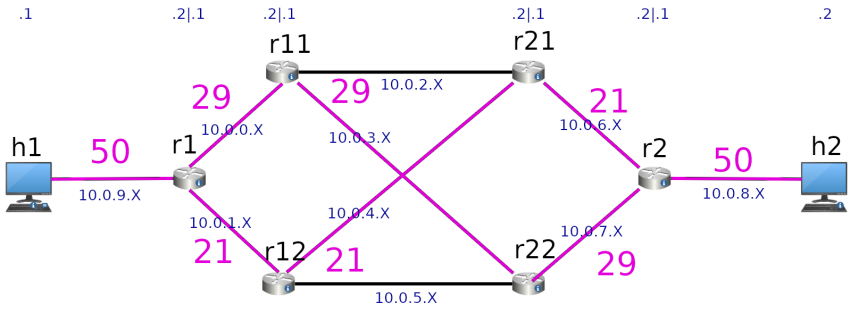

Figura 4.8: Prueba con la segunda configuración del entorno Kathara, donde la primera opción de r1 es r12 y la primera opción de r12 es r21. Adicionalmente, la segunda opción de r1 es r11 y la segunda opción de r11 es r22. Se envían 50 paquetes con puertos origen distintos desde h1 a h2, ECMP por flujo capa 4.

en la última prueba se envían 10MB en una conexión TCP entre h1 y h2.

4.3.1. Validación de ECMP por flujo

La primera prueba se realizó para ver que ECMP por flujo funciona correctamente en este entorno y no presenta el fenómeno extraño que se observa en los otros entornos. Entonces se enviaron 50 paquetes de h1 a h2 con distintos puertos de origen y se midió en r21 y r22 por que enlaces fueron estos paquetes. Todo usando ECMP por flujo capa 4. Esto resulto en que los paquetes ahora si utilizaron los 4 caminos disponibles y en particular r1-r11-r21-r2 (11), r1-r11-r22-r2 (15), r1-r12-r21-r2 (10), r1-r12-r22-r2 (14) como se puede ver en la Figura 4.9. Esto demuestra que el problema no es la implementación de ECMP de Linux, sino que su integración con *network namespaces*. Es desafortunado tener que recurrir a máquinas virtuales para probar ECMP de forma justa ya que esto limita bastante la escala de las pruebas que se pueden realizar, pero es altamente probable que alguna actualización del kernel arregle este problema algún día y de esta forma poder utilizar los otros ambientes, en especial el ambiente Mininet para probar ECMP en *fat-trees*.

4.3.2. Una "conexion" entre h1 y h2 de cien paquetes

En la segunda prueba se envían 100 paquetes UDP en una sola "conexión" entre h1 y h2. Se observa que tanto ECMP por flujo capa 4 y capa 3 solo utilizan un solo camino para los 100 paquetes como se ve en la Figura 4.10. La diferencia es que con ECMP por flujo capa 3 en distintas ejecuciones del test siempre utiliza el mismo camino (ya que el hash solo utiliza el par de IPs que es constante),

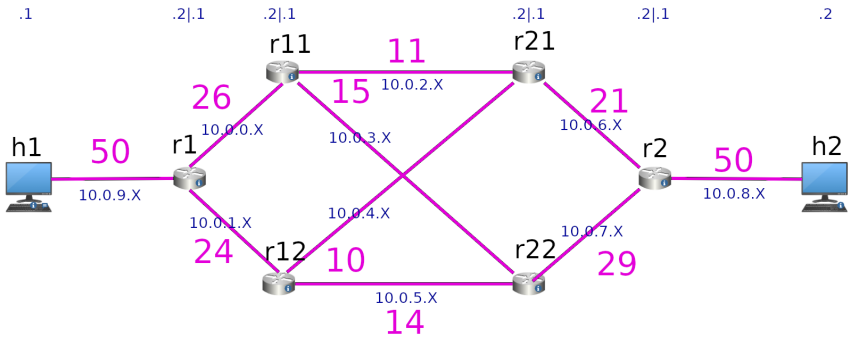

Figura 4.9: Primera prueba entorno Máquinas Virtuales, se envían 50 paquetes con puertos origen distintos desde h1 a h2 utilizando ECMP por flujo capa 4.

mientras que con ECMP por flujo capa 4 en distintas ejecuciones utiliza distintos caminos (ya que el puerto de origen es variable). Por otro lado, ECMP por paquete siempre utiliza los 4 caminos como se ve en la Figura 4.11. Esto es una constante en todas las otras pruebas, ECMP por paquete siempre utiliza todos los caminos y ECMP capa 3 siempre utiliza el mismo camino. La única forma de que ECMP capa 3 cambie de camino es reiniciando las máquinas virtuales y de esta forma regenerando la semilla del hash. Por lo tanto, lo interesante es ver el comportamiento de ECMP por flujo capa 4 en las siguientes pruebas. Adicionalmente, es claro que en este caso ECMP por paquete es el que mejor utiliza los recursos y este escenario presenta una de las grandes debilidades de ECMP por flujo capa 4.

4.3.3. Diez “conexiones” entre h1 y h2 de diez paquetes

En la tercera prueba se envían 100 paquetes UDP en 10 conexiones distintas entre h1 y h2, y se observa que ECMP capa 4 ahora si utiliza todos los caminos, pero el uso de los caminos es múltiplo de 10, es decir todos los paquetes de una misma “conexión” caen dentro del mismo camino, pero los paquetes de distintas conexiones caen en distintos caminos (aunque puede haber colisiones obviamente). Por ejemplo, en una ejecución se observó: r1-r11-r21-r2 (20), r1-r11-r22-r2 (40), r1-r12-r21-r2 (20), r1-r12-r22-r2 (20) como se ve en la Figura 4.12. Este caso se acerca muchísimo, pero no completamente, al caso ideal de ECMP por flujo capa 4, es decir, muchos flujos pequeños distintos.

4.3.4. Cien “conexiones” entre h1 y h2 de un paquete

En la cuarta prueba se envían 100 paquetes UDP en 100 conexiones distintas entre h1 y h2, y se observa que ECMP por flujo capa 4 obtiene un resultado

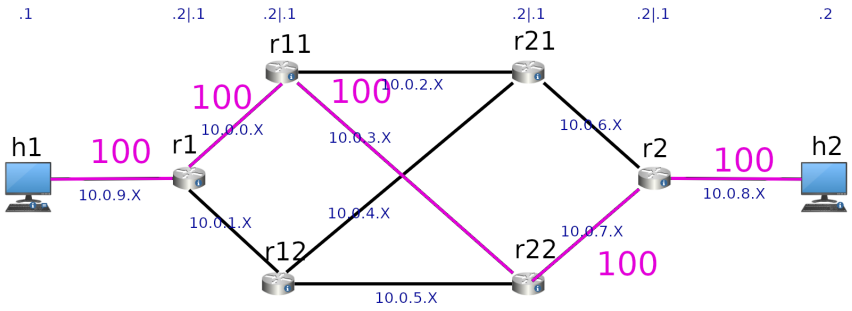

Figura 4.10: Segunda prueba entorno Máquinas Virtuales, se envían 100 paquetes con puertos origen y destino iguales desde h1 a h2. Uno de los resultados en los que ECMP por flujo capa 4 y capa 3 coinciden. Este resultado es representativo de ECMP por flujo capa 3 en todos los demás experimentos.

similar a ECMP por paquete, es decir similar a la Figura 4.11. Esto sucede ya que todas las conexiones resultan en un hash distinto ya que todas utilizan un puerto origen distinto y por lo tanto es equivalente a por paquete ya que siempre retorna un hash aleatorio. Este es el mejor caso para ECMP por flujo capa 4 aunque no es para nada realista.

4.3.5. Una "conexion" de diez paquetes y diez conexiones de un paquete entre h1 y h2

En la quinta prueba hay una "conexión" grande de 100 paquetes y 10 'conexiones' pequeñas de 1 paquete entre h1 y h2 y se observa que la "conexión" grande de 100 paquetes en el caso de ECMP capa 4 siempre cae toda por el mismo camino y además las conexiones pequeñas pueden caer con igual probabilidad en este camino en uso como en cualquiera de los otros 3 sin ningún o poco uso. El resultado de una ejecución fue r1-r11-r21-r2 (2), r1-r11-r22-r2 (3), r1-r12-r21-r2 (2), r1-r12-r22-r2 (103), como se puede ver en la Figura 4.13. Por lo tanto, aquí se observa la gran debilidad de ECMP por flujo capa 4. No solo 3 paquetes cayeron en un camino altamente utilizado cuando había caminos poco utilizados, sino que los 100 paquetes cayeron en un mismo camino cuando hay múltiples caminos en los que se podrían repartir.

4.3.6. Una conexión de diez megabytes entre h1 y h2

En la sexta prueba se observa la gran debilidad de ECMP por paquete, el reordenamiento de paquetes/segmentos en TCP. Si, ECMP por paquete utiliza de forma óptima los recursos de la red, pero ¿a qué costo? Veamos el porcentaje

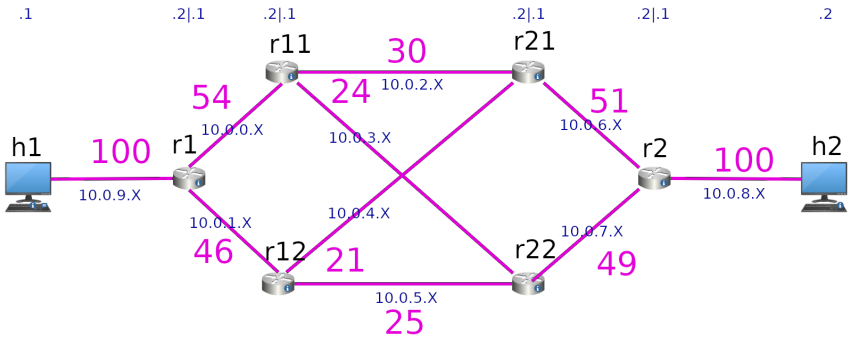

Figura 4.11: Segunda prueba entorno Máquinas Virtuales, se envían 100 paquetes con puertos origen y destino iguales desde h1 a h2. ECMP por paquete *random choice*. Este resultado es representativo de ECMP por paquete *random choice* en todos los demás experimentos.

de paquetes reordenados usando ECMP por paquete y ECMP por flujo capa 4 en una conexión de 10MB TCP. Utilizando ECMP por flujo el porcentaje de segmentos fuera de orden es 1 %, mientras que con ECMP por paquete varia en múltiples ejecuciones entre 15 % y 40 %. Poner de nuevo estos paquetes en orden puede llegar a causar un gran uso de CPU, especialmente si estamos hablando de flujos con tamaños en los GB y velocidades de transferencia en los MB/s. Esto muestra la gran debilidad de ECMP por paquete.

4.4. Conclusiones

En este capítulo se detallaron los distintos experimentos que se realizaron en los entornos Mininet, Kathará y máquinas virtuales descritos en el capítulo anterior.

En el entorno Mininet, se realizaron pruebas con tráfico todos a todos, todos a uno y una prueba con tráfico de un benchmark de datacenters. Se observó que la implementación de las pruebas todos a todos y todos a uno son correctas pero que aún así presentan un fenómeno extraño al utilizar ECMP por flujo, el cual limita su aplicabilidad. También se observo que en la prueba con tráfico de benchmark usando ECMP por paquete no hubo reordenamiento de paquetes alguno y se sospecha el ambiente Mininet provee un entorno ideal en el cual no hay reordenamiento.

En el entorno Kathará, se describió completamente el fenómeno extraño y se determino su causa. Este fenómeno implica que si un paquete es reenviado por al menos dos switches y el primer switch lo reenvía por su primera opción multicamino el segundo switch también lo reenviara por su primera opción mul-

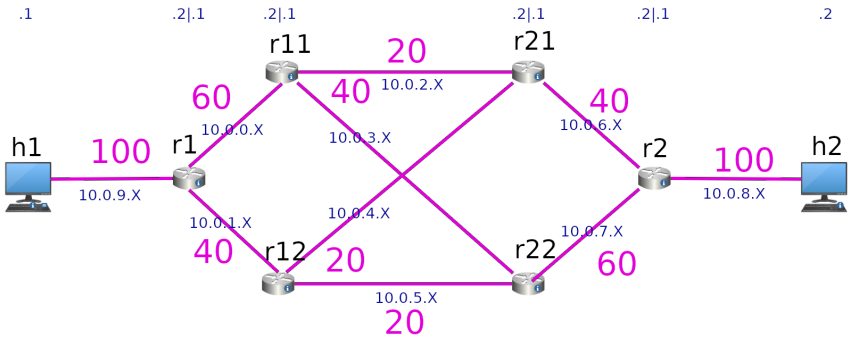

Figura 4.12: Tercera prueba entorno Máquinas Virtuales, se envían 100 paquetes UDP en 10 conexiones distintas entre h1 y h2, ECMP por flujo capa 4.

ticamino. Esto sucede ya que todos los switches están calculando el mismo hash y es causado por una carencia de la virtualización dada por los *network namespaces* presentes en Mininet y Kathará. Como consecuencia solo se utilizan $k/2$ caminos en lugar de todos los $k^2/4$ caminos de un *fat-tree*.

Finalmente, en el entorno de máquinas virtuales se confirmo la hipótesis anterior ya que en este entorno si funciona correctamente ECMP por flujo. También se realizaron varias pruebas que muestran las ventajas y desventajas de ECMP por paquete y por flujo.

4.4.1. Resumen de los experimentos

Pruebas en el entorno Kathará:

Experimento	ECMP	Resultado
50 "conexiones" distintas entre h1 y h2 de un paquete, r1 tiene como primera opción a r11	ECMP por flujo	Solo se balancean paquetes por dos caminos: r1-r11-r21 y r1-12-r22
50 "conexiones" distintas entre h1 y h2 de un paquete, r1 tiene como primera opción a r12	ECMP por flujo	Solo se balancean paquetes por dos caminos: r1-r12-r21 y r1-11-r22

Pruebas en el entorno de máquinas virtuales:

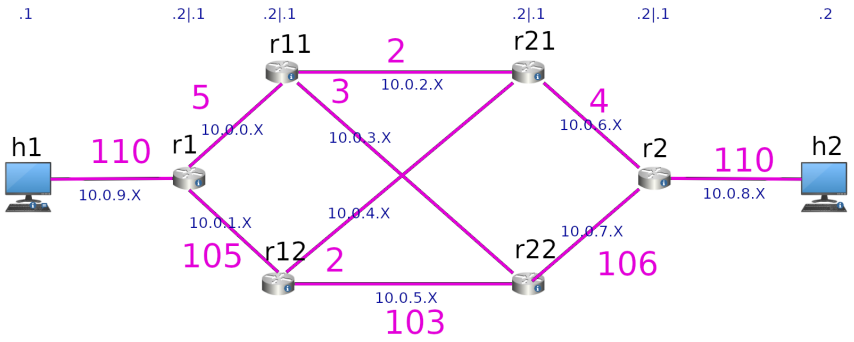

Figura 4.13: Quinta prueba entorno Máquinas Virtuales, se envían 110 paquetes UDP en 11 conexiones distintas entre h1 y h2, ECMP por flujo capa 4. Hay una "conexión" grande de 100 paquetes y 10 conexiones pequeñas de 1 paquete

Experimento	ECMP	Resultado
Primera prueba. 50 "conexiones" distintas entre h1 y h2 de un paquete	ECMP por flujo	Se balancean paquetes en todos los caminos a diferencia del entorno Kathará
Segunda prueba. Una "conexion" entre h1 y h2 de 100 paquetes	ECMP por flujo	MALO. Solo se utiliza un camino. En el caso de ECMP por flujo capa 3 los demás tests tienen este mismo resultado
Segunda prueba. Una "conexion" entre h1 y h2 de 100 paquetes	ECMP por paquete	BUENO. Se utilizan todos los caminos en igual cantidad. En ECMP por paquete los demás tests tienen este mismo resultado
Tercera prueba. Diez "conexiones" entre h1 y h2 de diez paquetes	ECMP por flujo capa 4	BUENO. Se utilizan todos los caminos pero solo se balancea por conexión
Cuarta prueba. 100 "conexiones" entre h1 y h2 de un paquete	ECMP por flujo capa 4	BUENO. Se utilizan todos los caminos en igual cantidad
Quinta prueba. Una "conexión" de 100 paquetes y 10 de un paquete	ECMP por flujo capa 4	MALO. Un camino es sobreutilizado y los demás son infrautilizados.
Sexta prueba. Una conexión de 10MB TCP entre h1 y h2	ECMP por flujo	BUENO. Reordenamiento de aproximadamente 1 % de los paquetes.
Sexta prueba. Una conexión de 10MB TCP entre h1 y h2	ECMP por paquete	MALO. Reordenamiento de entre 15 % y 40 % de los paquetes

Pruebas en el entorno Mininet con *fat-tree* k=4:

Experimento	ECMP	Resultado
Una conexión con once paquetes entre todo par de hosts	ECMP por flujo capa 3	Solo los switches de la capa ToR balancean paquetes y se realizan conexiones de once paquetes entre todo par de hosts. Distintas ejecuciones retornan el mismo resultado.
Una conexión con once paquetes entre todo par de hosts	ECMP por flujo capa 4	Solo los switches de la capa ToR balancean paquetes y se realizan conexiones de once paquetes entre todo par de host. Distintas ejecuciones retornan distintos resultados
Una conexión con once paquetes entre todo par de hosts	ECMP por paquete	Todos los switches balancean paquetes y se realizan conexiones de once paquetes entre todo par de hosts. Distintas ejecuciones retornan distintos resultados
Una conexión con once paquetes entre todos los hosts y h1	ECMP por flujo capa 3	Solo los switches de la capa ToR balancean paquetes y se realizan conexiones de once paquetes entre todo host y h1. Distintas ejecuciones retornan el mismo resultado
Una conexión con once paquetes entre todos los hosts y h1	ECMP por flujo capa 4	Solo los switches de la capa ToR balancean paquetes y se realizan conexiones de once paquetes entre todo host y h1. Distintas ejecuciones retornan distintos resultados
Una conexión con once paquetes entre todos los hosts y h1	ECMP por paquete	Todos los switches balancean paquetes y se realizan conexiones de once paquetes entre todo host y h1. Distintas ejecuciones retornan distintos resultados
Una conexión de 10MB entre h1 y h16 en el medio del tráfico TrafPy	ECMP por paquete y ECMP por flujo	No se observa reordenamiento de paquetes alguno. Adicionalmente, el sistema replica correctamente lo indicado por el archivo en formato .csv

4.4.2. Limitaciones y facilidades de los entornos

Este es un pequeño resumen de las limitaciones y facilidades de los entornos respecto a la investigación sobre ECMP.

Entorno	Limitaciones	Facilidades
Mininet	Solo se virtualiza el stack de red de cada proceso (nodo) utilizando network namespaces. La implementación de Linux de ECMP por flujo no funciona correctamente.	Es el entorno que mas escala en terminos de cantidad de hosts.
Kathará	Cada nodo NO tiene su propio kernel y la implementación de Linux de ECMP por flujo no funciona correctamente.	Virtualiza el stack de red y además provee aislamiento entre procesos a través de contenedores Docker, los cuales utilizan capacidades del kernel subyacente para funcionar. Escala bastante bien y utiliza Docker que es una tecnología muy robusta.
Máquinas Virtuales	Es el entorno que menos escala, muy limitado en la cantidad de nodos y enlaces por nodo. En VirtualBox solo permite cuatro adaptadores de red por host y el uso de CPU por host es enorme en comparación con los otros entornos.	Virtualización a nivel de hardware, cada VM tiene su propio kernel y ECMP por flujo funciona correctamente

En lineas generales podemos observar que al aumentar el nivel de virtualización los entornos se hacen mas realistas pero son más difíciles de escalar hacia una mayor cantidad de nodos. El caso extremo es utilizar un dispositivo real por cada nodo que se desee emular, en este caso se tiene el mayor realismo posible pero agregar nodos tiene un costo real muy evidente.

Por otro lado, es muy claro que la implementación de Linux de ECMP por flujo requiere de al menos el nivel de realismo otorgado por las máquinas virtuales, es decir que cada nodo tenga su propio kernel. Lo cual es desafortunado ya que es difícil escalar este tipo de entornos aunque por lo menos no es necesario tener que utilizar muchos dispositivos distintos para probar dicha implementación.

Finalmente, en esencia nada impide que las capacidades del kernel Linux aumenten y contemplen la virtualización de la semilla de hash utilizada por ECMP por flujo, lo cual transformaría a Mininet y Kathará en buenos entornos para emular ECMP. El problema es que la única razón que se me ocurre por la cual agregarían esto al kernel es para realizar estos experimentos ya que en una red virtual dentro de una misma computadora no tiene mucho sentido utilizar ECMP. Pero no todo son malas noticias, la implementación de ECMP de Linux se actualiza con frecuencia y es posible que en un futuro se arregle este problema.

Capítulo 5

Conclusiones y Trabajo Futuro

En este proyecto se comprobó y verificó el funcionamiento de ECMP por flujo presente en las últimas versiones de Linux. También se desarrolló una modificación al kernel que agrega ECMP por paquete, y luego se comprobó y verificó el funcionamiento de este. Se observaron los puntos fuertes y los puntos débiles de ambas estrategias y como se comparan entre sí. Esto se vio en un ambiente de máquinas virtuales debido a un problema encontrado en los ambientes de Mininet y Kathará.

Por otra parte, se confirmó que el caso ideal para ECMP por flujo es muchas conexiones de pequeño tamaño, mientras que los tres peores casos son cuando hay una sola conexión grande, o si hay una conexión grande y muchas pequeñas, o si dos conexiones grandes tienen una colisión. Es decir, ECMP por flujo es ideal cuando hay muchos flujos homogéneos pequeños y el peor caso es cuando hay una combinación heterogénea de flujos y por lo menos uno de estos es un flujo grande.

Además, se constató que ECMP por paquete utiliza todos los caminos de forma uniforme, pero tiene una gran debilidad que lo hizo ser eliminado del kernel Linux: el reordena miento de paquetes y su impacto en el uso de CPU de los endpoints cuando se utiliza TCP.

Por otro lado, también se expandió un ambiente Mininet para *fat-trees*, agregándole soporte para ECMP, corrigiendo algún problema interesante, y permitiendo realizar diversas pruebas parametrizadas. Se desarrollaron pruebas con tráfico todos a todos, todos a uno y cargas basadas en mediciones sobre redes reales. Permitiendo observar gráficamente el uso de la red luego de estas y obtener información sobre el promedio de las conexiones realizadas en dichas pruebas. Este trabajo puede ganar más valor en un futuro cuando se arreglen los problemas subyacentes al entorno Mininet.

El trabajo futuro más evidente es probar cada vez que se actualice el kernel Linux si el problema de la no virtualización de la semilla para ECMP es

solucionado. Lo que mejoraría enormemente el uso del entorno Mininet para realizar pruebas con ECMP. Se podría presentar este problema a la *mailing list* de Linux por si hay algún interesado en resolver esto o también se podría tratar de implementar una modificación que lo solucione.

Adicionalmente, es claro que tanto ECMP por paquete y ECMP por flujo no son soluciones perfectas, es más, estamos frente a un problema abierto con muchas soluciones heterogéneas a las cuales también se podrían emular y verificar su funcionamiento aunque, como ya se mencionó antes, esto es bastante más difícil de hacer que ECMP. Una alternativa para abordar el problema puede ser tratar de simular estas técnicas de reenvío en lugar de emularlas.

5.1. Cronograma de trabajo del proyecto

Como este proyecto de grado tuvo una índole exploratoria, resulta interesante observar como fue el desarrollo de este en el plano temporal. El esquema de trabajo fue tener una reunión bisemanal para planificar las actividades de las próximas dos semanas, luego realizar dichas actividades y en la próxima reunión reportar el trabajo realizado y ajustar las actividades para las próximas semanas.

Primero, se realizó un estudio del estado del arte en técnicas de reenvío. Segundo, se trabajó extensamente con el entorno Mininet realizándose múltiples pruebas y expansiones. Tercero, se creó un kernel Linux modificado para poder realizar pruebas sobre ECMP por paquete. Cuarto, se detectó el error en este entorno y fue necesario cambiar el rumbo del proyecto. Quinto, se acudió al entorno Kathará y luego al entorno de máquinas virtuales para poder probar los tipos de ECMP correctamente y sin sesgos. Sexto, se realizaron las pruebas correspondientes en este entorno para verificar que el comportamiento de los diversos tipos de ECMP es el correcto y cuáles son sus ventajas y desventajas. Finalmente, se elaboró este informe.

Es fútil preguntarse qué hubiera pasado si el entorno Mininet hubiera funcionado correctamente y la trayectoria hubiera sido más previsible. En los planes estaba realizar más pruebas con ECMP en Mininet y comprarlos en este entorno que escala muy bien, y en parte se hicieron algunas pruebas de performance comparativas pero los resultados no son correctos ya que no es justo comparar ECMP por paquete y ECMP por flujo en este entorno. Por otro lado, si se hubiera tenido más tiempo capaz se podría haber explorado otras técnicas de reenvío como MPTCP, la cual se estaba investigando para integrarla al entorno, pero debido al cambio de rumbo se dejó de lado.

Con avances y con retrasos el cronograma es el de la Figura 5.1. En azul actividades pertinentes a escritura de documentación o informes, en naranja actividades pertinentes al ambiente Mininet, en gris actividades que inicialmente fueron para el ambiente Mininet pero luego se usaron en los demás ambientes, y finalmente en amarillo las actividades relacionadas a el ambiente Kathará y de máquinas virtuales.

Además de lo que se ve en el cronograma hay dos horas cada dos semanas

CRONOGRAMA	SEMANAS																	
ACTIVIDADES	1 2	3 4	5 6	7 8	9 10	11 12	13 14	15 16	17 18	19 20	21 22	23 24	25 26	27 28	29 30	31 32	33 34	Total Horas
Estudio del estado del arte	30	10	10	10	0	0	0	0	0	0	0	0	0	0	0	0	0	60
Instalación y exploración del entorno Mininet existente	0	20	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	25
Experimentos con ECMP por flujo en la CLI de Mininet	0	0	15	5	0	0	0	0	0	0	0	0	0	0	0	0	0	20
Creación de un Kernel Linux que agrega ECMP por paquete	0	0	0	20	15	0	0	0	0	0	0	0	0	0	0	0	0	35
Creación de una Maquina Virtual con el kernel modificado y con todos los requerimientos para el entorno Mininet, junto con una guía para crear dichos artefactos	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	15
Agregar ECMP por flujo y por paquete al entorno Mininet	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	5
Corrección de algunos errores en el entorno Mininet y adición de configuración para limitar el uso de los enlaces	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	10
Implementación y creación de pruebas todos contra todos en Mininet	0	0	0	0	0	20	20	0	0	0	0	0	0	0	0	0	0	40
Adición de un graficador para las pruebas todos contra todos que indique que caminos se utilizaron	0	0	0	0	0	0	0	30	0	0	0	0	0	0	0	0	0	30
Mejora en la forma que se determinan los caminos utilizados pasando de traceroute a capturas de tráfico que indican cuantos paquetes se enviaron por cada enlace	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	10
Pasar de usar pyshark a tshark para contar paquetes	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	5
Corrección de IPs inconsistentes en Mininet y correcciones relativas a la portabilidad del script	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	5
Marcar paquetes para solo contar los enviados por las pruebas, ignorando el tráfico de fondo de la red y obteniendo resultados mas consistentes	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	10
Creacion de pruebas todos contra uno simulando incast	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	15
Estudio de la herramienta TrafPy resolviendo el problema de resolución de dependencias a la hora de usarla	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	15
Creación de un script que utiliza TrafPy que genera benchmarks para fat-trees y exportarlo como archivo en formato .csv	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	15
Creación de un scheduler en el entorno Mininet que lee este archivo .csv e implementa el benchmark iniciando los flujos correspondientes en el momento correspondiente	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	15
Determinación de una gran limitante del entorno Mininet y creación del entorno Kathará	0	0	0	0	0	0	0	0	0	0	0	35	0	0	0	0	0	35
Creación de dos pruebas en el entorno Kathará	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	5
Determinación de la misma limitante en el entorno Kathará y creación de un entorno de máquinas virtuales	0	0	0	0	0	0	0	0	0	0	0	0	40	0	0	0	0	40
Mejora del entorno de máquinas virtuales cambiando la imagen base y agregándole el kernel modficado	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	5
Simplificación de la manera en la que se ejecutan pruebas en el entorno de máquinas virtuales	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	10
Creación de las pruebas en el entorno de máquinas virtuales	0	0	0	0	0	0	0	0	0	0	0	0	0	0	30	0	0	30
Realización de este informe	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	40	40	80
Total Horas	30	30	30	35	35	30	20	30	15	30	45	40	40	15	30	40	40	535

Figura 5.1: Cronograma

que no están comprendidas, una hora de preparación para la reunión bisemanal y una hora de reunión.

Una reflexión interesante, luego de trabajar numerosas horas en este tema es que los entornos posteriores a Mininet fueron más rápidos de desarrollar en comparación debido a las lecciones aprendidas en el desarrollo de este entorno. También influyo que el entorno Mininet es más complejo y que se tomó un proyecto ya empezado, pero creo que lo que marcó la diferencia fueron los aprendizajes. Por esta misma razón cerca del final la cantidad de actividades fue menor, ya que no tuvieron que realizarse descubrimientos intermedios, sino que se pudo usar lo aprendido. Un ejemplo muy claro es la forma en la cual se determina el uso de la red, lo cual al principio consumió bastante tiempo para perfeccionarlo y luego fue simplemente aplicarlo.

Referencias

Agarwal, K., Dixon, C., Rozner, E., y Carter, J. (2014). Shadow macs: Scalable label-switching for commodity ethernet. En *Proceedings of the third workshop on hot topics in software defined networking* (p. 157–162). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2620728.2620758 doi: 10.1145/2620728.2620758

Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., ... Varghese, G. (2014). Conga: Distributed congestion-aware load balancing for datacenters. En *Proceedings of the 2014 acm conference on sigcomm* (p. 503–514). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2619239.2626316 doi: 10.1145/2619239.2626316

Bram Stolk. (2022). *Buildyourownkernel.* https://wiki.ubuntu.com/Kernel/BuildYourOwnKernel. (Accessed: 2023-12-6)

C. Hopps. (2000). *Rfc 2992.* https://www.rfc-editor.org/rfc/rfc2992. (Accessed: 2023-16-11)

Cao, J., Xia, R., Yang, P., Guo, C., Lu, G., Yuan, L., ... Maltz, D. (2013). Per-packet load-balanced, low-latency routing for clos-based data center networks. En *Proceedings of the ninth acm conference on emerging networking experiments and technologies* (p. 49–60). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2535372.2535375 doi: 10.1145/2535372.2535375

Christopher W. F. Parsonson. (2023). *Trafpy.* https://trafpy.readthedocs.io/en/latest/. (Accessed: 2023-11-16)

David S. Miller. (2015). *Kernel commit.* https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=07355737a8badd951e6b72aa8609a2d6eed0a7e7. (Accessed: 2023-16-11)

Dixit, A., Prakash, P., Hu, Y. C., y Kompella, R. R. (2013). On the impact of packet spraying in data center networks. En *2013 proceedings ieee infocom* (p. 2130-2138). doi: 10.1109/INFCOM.2013.6567015

Docker, Inc. (2023). *Docker.* https://www.docker.com/. (Accessed: 2023-12-5)

ESnet. (2023). *iperf3.* https://github.com/esnet/iperf. (Accessed: 2023-12-5)

Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., y Wójcik, M. (2017). Re-architecting datacenter networks and stacks for low latency and high performance. En *Proceedings of the conference of the acm special interest group on data communication* (p. 29–42). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/3098822.3098825 doi: 10.1145/3098822.3098825

He, K., Rozner, E., Agarwal, K., Felter, W., Carter, J., y Akella, A. (2015). Presto: Edge-based load balancing for fast datacenter networks. En *Proceedings of the 2015 acm conference on special interest group on data communication* (p. 465–478). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2785956.2787507 doi: 10.1145/2785956.2787507

IETF. (2020). *Rfc 8684*. https://www.rfc-editor.org/rfc/rfc8684. (Accessed: 2023-16-11)

Kathara Team. (2023). *Kathara*. https://www.kathara.org. (Accessed: 2023-11-16)

Kernel Developers. (2023). *Kernel documentation*. https://www.kernel.org/doc/Documentation/networking/ip-sysctl.rst. (Accessed: 2023-16-11)

Kernel Newbies. (2016). *Kernel newbies article for linux 4.4*. https://kernelnewbies.org/Linux_4.4. (Accessed: 2023-16-11)

Leonardo Alberro. (2022). *Fat tree emulators*. https://gitlab.com/fing-mina/datacenters/fat-tree-emulators/-/tree/main/mininet. (Accessed: 2023-25-11)

Linux Man Page. (2023a). *Network namespaces*. https://www.man7.org/linux/man-pages/man7/network_namespaces.7.html. (Accessed: 2023-11-16)

Linux Man Page. (2023b). *Virtual ethernet*. https://man7.org/linux/man-pages/man4/veth.4.html. (Accessed: 2023-11-16)

Mininet Team. (2023). *Mininet overview*. https://mininet.org/overview/. (Accessed: 2023-11-16)

NetFPGA Project. (2023). *Netfpga*. https://netfpga.org/About.html. (Accessed: 2023-12-5)

Nikolay Aleksandrov. (2017). *Kernel commit*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bf4e0a3db97eb882368fd82980b3b1fa0b5b9778. (Accessed: 2023-16-11)

Noormohammadpour, M., y Raghavendra, C. S. (2018). Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, *20*(2), 1492-1525. doi: 10.1109/COMST.2017.2782753

Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., ... Fonseca, R. (2014). Planck: Millisecond-scale monitoring and control for commodity networks. En *Proceedings of the 2014 acm conference on sigcomm* (p. 407–418). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2619239.2626310 doi: 10.1145/2619239.2626310

Scott Hogg. (2014). *Clos networks: What's old is new again.* https://www.networkworld.com/article/747658/clos-networks-what-s-old-is-new-again.html. (Accessed: 2023-12-8)

Sebastián Rodríguez. (2023). *Repositorio emuladores ecmp.* https://gitlab.com/r.g.sebastian/ecmp-emulators. (Accessed: 2023-12-8)

Stephens, B., Cox, A., Felter, W., Dixon, C., y Carter, J. (2012). Past: Scalable ethernet for data centers. En *Proceedings of the 8th international conference on emerging networking experiments and technologies* (p. 49–60). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/2413176.2413183 doi: 10.1145/2413176.2413183

The Wireshark Team. (2023). *tshark.* https://tshark.dev/setup/about/. (Accessed: 2023-12-5)

Vardhan, H., Thomas, N., Ryu, S.-R., Banerjee, B., y Prakash, R. (2011, 01). Wireless data center with millimeter wave network. En (p. 1 - 6). doi: 10.1109/GLOCOM.2010.5684121

Virtualbox Team. (2023). *Virtualbox.* https://www.virtualbox.org/. (Accessed: 2023-11-16)

Wang, P., Xu, H., Niu, Z., Han, D., y Xiong, Y. (2017). Expeditus: Congestion-aware load balancing in clos data center networks. *IEEE/ACM Transactions on Networking*, *25*(5), 3175-3188. doi: 10.1109/TNET.2017.2731986

Wikipedia. (2023a). *Software defined networking.* https://en.wikipedia.org/wiki/Software-defined_networking. (Accessed: 2023-12-5)

Wikipedia. (2023b). *Type of service.* https://en.wikipedia.org/wiki/Type_of_service. (Accessed: 2023-12-5)

Anexo A

Replicar los entornos de experimentación

En este anexo se aborda más detalladamente como compilar el kernel utilizado en este proyecto. También se muestra cómo crear una VM Ubuntu 22.04.2 con los requerimientos de los entornos. Finalmente, se detalla cómo utilizar esta VM para replicar los entornos utilizados.

Por otro lado, cabe aclarar que se puede acceder tanto a los paquetes binarios (.deb) del kernel ya compilado como a una VM con todo instalado en el repositorio (Sebastián Rodríguez, 2023).

A.1. Compilar el kernel modificado

Esta guía se basa en la documentación de Ubuntu relevante que se puede encontrar en (Bram Stolk, 2022) con algunas adaptaciones ya que la guía es para Ubuntu 19.04 y se utilizó Ubuntu 22.04.2 LTS dentro de una VM. Para compilar e instalar el kernel modificado que agrega la técnica de reenvío ECMP por paquete se debe:

1. Habilitar repositorios de código fuente deb-src en */etc/apt/sources.list* (descomentando las lineas correspondientes)
2. Actualizar los repositorios y el sistema:

   ```
   sudo apt update && sudo apt upgrade
   ```

3. Instalar dependencias para compilar el kernel actual con apt build-dep:

   ```
   sudo apt build-dep linux linux-image-unsigned-$(uname -r)
   ```

4. Bajar el código fuente del kernel con los parches de Ubuntu para el kernel actual y darle permiso de ejecución a los build scripts:

   a) sudo apt source linux-image-unsigned-$(uname -r)

b) cd linux-5.15.0 (para uname -r = 5.15.0-72-generic)

c) sudo chmod a+x debian/scripts/*

d) sudo chmod a+x -R ./scripts

e) sudo chmod a+x debian/rules

5. Aplicar los cambios propuestos al kernel:

   a) Implementar hash random como multipath hash policy. Modificar la función "`int fib_multipath_hash(...){`" en *net/ipv4/route.c* agregándole un caso extra al switch:
   `switch (READ_ONCE(net->ipv4.sysctl_fib_multipath_hash_policy))`
   de la siguiente forma:

```
case 4:
    get_random_bytes(&mhash, sizeof(mhash));
    break;
```

   b) Habilitar el valor 4 para *fib_multihash_policy* en sysctl. En el archivo *net/ipv4/sysctl_net_ipv4.c*, cambiar:

```
{
.procname       = "fib_multipath_hash_policy",
.data             = &init_net.ipv4.sysctl_fib_multipath_hash_policy,
.maxlen             = sizeof(u8),
.mode             = 0644,
.proc_handler       = proc_fib_multipath_hash_policy,
.extra1             = SYSCTL_ZERO,
.extra2             = &three,
},
```

   por

```
{
.procname       = "fib_multipath_hash_policy",
.data             = &init_net.ipv4.sysctl_fib_multipath_hash_policy,
.maxlen             = sizeof(u8),
.mode             = 0644,
.proc_handler       = proc_fib_multipath_hash_policy,
.extra1             = SYSCTL_ZERO,
.extra2             = &four,
},
```

   (se cambio &three por &four)

6. Copiar la configuración actual del kernel y aplicársela al kernel que vamos a compilar.

   a) cp /boot/config-$(uname -r) ./.config

   b) make olddefconfig (considera si hay nuevos simbolos en el kernel respecto al .config anterior)

   c) make localmodconfig (indica que solo se compilen los módulos que se precisen en esta máquina) (no usarlo si se quiere hacer un build para distribuir)

7. Desactivar los símbolos de debug: `scripts/config --disable DEBUG_INFO`
8. Compilar el kernel con 12 threads(jobs) y version 5.15.0-72-pgreenvio: make -j12 bindeb-pkg LOCALVERSION=-pgreenvio
9. Instalar los debs que acabamos de compilar, actualizar grub y reiniciar:
   a) cd ..
   b) sudo dpkg -i *.deb
   c) sudo update-grub
   d) sudo reboot
10. Es posible que sea necesario seleccionar el kernel modificado en grub al momento de bootear. Esto sucedería especialmente luego de una actualización del sistema traiga un nuevo kernel, ya que por defecto se utiliza el último en el proceso de boot.

Luego de este proceso además de tener el kernel instalado se obtienen los paquetes .deb que sirven para instalar el kernel en cualquier sistema operativo Linux basado en Debian (como puede ser Ubuntu o Debian).

Respecto a la guía original se agregó el paso de copiar la configuración actual en lugar de realizar toda la configuración desde cero. Otra adición fue agregarle la versión pgreenvio (al hacer uname -r cuando se corre este kernel aparece 5.15.0-72-pgreenvio en lugar de 5.15.0-72 y es importante para distinguir las versiones y que no haya problemas). Adicionalmente, en lugar de usar fakeroot se utilizó el usuario root real de la máquina virtual ya que no funcionaba correctamente de esa forma. Finalmente, el comando de compilación también se cambió.

Inicialmente, la mayor dificultad en este proceso fueron los grandes tiempos de espera a la hora de compilar el kernel (2-3hs) y la cantidad de diferencias con la documentación encontrada para compilar. Adicionalmente, era necesario recompilar todo el kernel cada vez que se realizaba un cambio. Estos problemas se pudieron solventar indicando en el proceso de compilación que se compilen solo los módulos que actualmente está utilizando el sistema, e indicando que se realicen compilaciones parciales en lugar de compilar desde cero cada vez. Por lo tanto, si se sigue la guía de este anexo, es esperable una compilación bastante rápida (10min aproximadamente dependiendo del hardware utilizado).

A.2. Crear una VM con el kernel modificado e instalar los requerimientos de los entornos

En este anexo se muestra una forma de configurar una VM para que pueda correr los entornos Mininet y Kathara. Adicionalmente, esta VM puede usarse

como imagen para el ambiente de máquinas virtuales.

Empezando con una instalación limpia de Ubuntu 22.04.2:

1. Actualizar los repositorios y el sistema:

   `sudo apt update && sudo apt upgrade`

2. Instalar Mininet

   a) sudo apt install mininet

   b) git clone https://github.com/mininet/mininet;

   c) mininet/util/install.sh -f; Si no se instala esto se obtendrá el error *Could not find a default OpenFlow controller* aunque no se use OpenFlow

3. Instalar dependencias de Python para el script del entorno Mininet

   a) sudo apt install frr (para los demonios de routing)

   b) sudo apt install pip; sudo pip install networkx;

   c) sudo pip install matplotlib; sudo pip install igraph (para graficar los *fat-tree*)

   d) sudo apt install iperf3 (para realizar las conexiones)

   e) sudo apt install tshark (para contar paquetes en capturas de tráfico)

   f) sudo apt install net-tools (ifconfig, route...)

4. Darle permisos de ejecucion a other y group sobre la carpeta home del usuario. Si no se hace esto los demonios de frr (zebra y bgpd) no pueden leer sus archivos de configuración y levantan una excepción:

   `'vty_read_config: failed to open configuration file /home/pgreenvio/mytest/conf/bgpd_r10001.conf'`

   chmod og+rx /home/pgreenvio

5. Instalar kathara: sudo apt install kathara

6. Instalar el kernel modificado. Para esto es necesario bajar o compilar (siguiendo la guía del anexo anterior) los archivos .deb. Se instalan todos con

   a) dpkg -i .deb (instalar)

   b) update-grub

   c) reboot

7. Luego es buena idea verificar que el kernel deseado esta ejecutando, utilizando uname -r

8. Para exportar la imagen es interesante minimizar el tamaño, y esto se puede hacer de la siguiente forma:

   a) Adentro de la VM, correr zerofree. Esto deja todo el espacio inutilizado en el sistema de archivos en 0.

   b) Afuera de la VM, utilizar vboxmanage –compact (para virtualbox) y exportar la imagen desde la GUI

Luego de realizar estos pasos ya es posible seguir las guías particulares de cada entorno. También es posible saltearse estos pasos bajando la imagen ya configurada del link en el repositorio.

A.3. Crear el benchmark de tráfico para fat-trees con TrafPy

Como las instrucciones de instalación de TrafPy no indican la versión de Python necesaria para poder resolver las dependencias que necesita TrafPy se utilizó el IDE PyCharm ya que permite resolver dependencias de forma más sencilla. Aun así, fue necesario probar distintas versiones de Python hasta que la resolución de dependencias funcionó correctamente con Python 3.7.9, probablemente también funcione para Python 3.7.x con $x > 9$ ya que luego de esta versión son solo parches de seguridad. Una vez tenemos la versión correcta de Python instalado las instrucciones fueron fáciles de seguir. En resumen:

- Instalar Python 3.7.9
- Instalar el modulo TrafPy (se puede hacer con los siguientes comandos o a través de la interfaz de PyCharm)
  - git clone https://github.com/cwfparsonson/trafpy (clonar el repo)
  - python setup.py install (instalar el modulo trafpy)
  - pip install -r requirements/default.txt (instalar las dependencias)
- Puede ser que la instalación no agregue los benchmarks, es necesario copiar los archivos .py desde

  `trafpy-master/trafpy/benchmarker/versions/benchmark\_v001/` hacia donde se instalo el modulo (con PyCharm es un venv dentro del proyecto y si se instalo con pip en Linux es en /usr/local/lib/python3.7/)

Una vez esta pronto el entorno es suficiente ejecutar/interpretar el archivo main.py desarrollado dentro de la carpeta trafpy del repositorio para generar los .csv con los benchmarks. Al inicio de este script se puede configurar el k del *fat-tree*, la velocidad de los endpoints (para que se alargue en más flujos pequeños el test) y el nombre del benchmark (commercial_cloud y university son los que se usaron). Se calculan automáticamente los parámetros necesarios para que TrafPy funcione correctamente a partir de estos valores (como puede ser la cantidad de hosts/endpoints de un *fat-tree* a partir de k).

```
K = 4
endpoint_amount = K*K*K/4
endpoint_speed = 250  # (KB/s)
#benchmark_name = "university"
benchmark_name = "commercial_cloud"
(...)
```

Anexo B

Replicar los experimentos realizados

En este anexo se precisa como replicar los experimentos una vez que se tiene un sistema operativo configurado como la VM creada en el anexo A.2. Se utiliza el código disponible en el repositorio (Sebastián Rodríguez, 2023).

B.1. Experimentos en Mininet

Primero que nada, iniciar la VM y conectarse por ssh para correr comandos dentro de esta. Luego, copiar la carpeta *mininet* desde el repositorio hacia la VM (se puede usar scp en otra consola o carpetas compratidas).

El siguiente paso es verificar que el comportamiento multicamino esta activado, esto es necesario setearlo en tiempo de compilación. Aun así, esta activado por defecto en el kernel usado por Ubuntu 22.04.2 (sino habría que agregarlo manualmente en la configuración a la hora de compilar). Se verifica con:

`cat /boot/config-$(uname -r) | grep CONFIG_IP_ROUTE_MULTIPATH`

Si esta activado imprime `'CONFIG_IP_ROUTE_MULTIPATH=y'`

Para ejecutar el entorno mininet con la configuración dada en parameters.json y para un *fat-tree* k=4 con k/2 hosts por switch ToR se corre el comando: sudo python3 fattree_bgp.py 4.

B.1.1. Experimento traceroutes

Para seguir este experimento y antes de correr el entorno, en el archivo parameters.json se setea *doCLI* en true y todas las demás opciones que empiezan con 'do' en false para desactivar los tests y tener acceso a la consola de mininet. Para la primera prueba seteamos fibMultipathHashPolicy a 0 en este mismo archivo y disableRPFilter en false. Ahora, se corre el script con sudo python3 fattree_bgp.py 4

Correr en la consola de mininet algunos traceroutes:

```
mininet> h1 traceroute h4
traceroute to 200.0.3.100 (200.0.3.100), 30 hops max, 60 byte packets
 1  200.0.0.1 (200.0.0.1)  0.033 ms  0.005 ms  0.003 ms
 2  10.0.18.1 (10.0.18.1)  0.013 ms  0.006 ms  0.005 ms
 3  * * *
 4  200.0.3.100 (200.0.3.100)  0.032 ms  0.010 ms  0.014 ms

-fibMultipathHashPolicy:0
-disableRPFilter:false
```

Obsérvese como el tercer salto no responde al traceroute, lo que está pasando acá es que r30001 (10.0.18.2) está dropeando los paquetes de r30002 (10.0.18.1) ya que el reverse path filtering esta activado (rp_filter). Mas específicamente, está dropeando los paquetes que tienen IP fuente que no saben cómo alcanzar. Hace esto para evitar el spoofing de IPs fuentes. Se puede desactivar para todos los switches en parameters.json setando disableRPFilter en true.

Por ahora podemos desactivarlo solo para r30001 usando

```
r30001 find /proc/sys/net/ipv4/conf -name rp_filter -exec sh -c "echo 0 > '{}'" ;
```

Si corremos el mismo traceroute de nuevo ahora si se imprime el tercer salto correctamente.

```
  mininet> h1 traceroute h4
traceroute to 200.0.3.100 (200.0.3.100), 30 hops max, 60 byte packets
 1  200.0.0.1 (200.0.0.1)  0.016 ms  0.004 ms  0.003 ms
 2  10.0.18.1 (10.0.18.1)  0.012 ms  0.006 ms  0.005 ms
 3  10.0.19.2 (10.0.19.2)  0.013 ms  0.008 ms  0.007 ms
 4  200.0.3.100 (200.0.3.100)  0.014 ms  0.011 ms  0.009 ms

-fibMultipathHashPolicy:0
-disableRPFilter:false (set to false manually for r30001)
```

Este es el resultado para fib_multipath_hash_policy=0 (ECMP por flujo capa 3). Si lo seteamos a 1 (ECMP por flujo capa 4) obtenemos distintos resultados en el segundo salto ya que traceroute está utilizando distintos puertos para las distintas probes. Si corremos el script de nuevo, pero con este cambio obtenemos al correr un traceroute:

```
mininet> h1 traceroute h4
traceroute to 200.0.3.100 (200.0.3.100), 30 hops max, 60 byte packets
 1  200.0.0.1 (200.0.0.1)  0.016 ms  0.004 ms  0.003 ms
 2  10.0.16.1 (10.0.16.1)  0.010 ms 10.0.18.1 (10.0.18.1)  0.010 ms  0.005 ms
 3  10.0.19.2 (10.0.19.2)  0.013 ms  0.008 ms  0.008 ms
 4  200.0.3.100 (200.0.3.100)  0.013 ms  0.012 ms  0.009 ms

-fibMultipathHashPolicy:1
-disableRPFilter:true
```

Se puede observar que ahora si hay balanceo de carga en el switch ToR r30001 ya que el segundo salto varía la IP del switch que responde. Si obligamos que traceroute utilice el mismo puerto fuente y destino en las distintas probes:

```
h1 traceroute -T --sport=8999 --port=13699 h4
traceroute to 200.0.3.100 (200.0.3.100), 30 hops max, 60 byte packets
 1  200.0.0.1 (200.0.0.1)  0.043 ms  0.007 ms  0.006 ms
 2  10.0.18.1 (10.0.18.1)  0.013 ms  0.008 ms  0.007 ms
 3  10.0.19.2 (10.0.19.2)  0.014 ms  0.009 ms  0.008 ms
 4  200.0.3.100 (200.0.3.100)  0.034 ms  0.012 ms  0.011 ms


-fibMultipathHashPolicy:1
-disableRPFilter:true
```

Se puede observar que no hay balanceo de carga alguno ya que al mantener los puertos fijos no se puede sacar provecho de ECMP por flujo capa 4. Por otro lado, si corremos el script de nuevo, pero con fibMultipathHashPolicy en 4 conseguimos:

```
h1 traceroute -T --sport=8999 --port=13699 h4
traceroute to 200.0.3.100 (200.0.3.100), 30 hops max, 60 byte packets
 1  200.0.0.1 (200.0.0.1)  0.017 ms  0.008 ms  0.005 ms
 2  10.0.18.1 (10.0.18.1)  0.010 ms 10.0.16.1 (10.0.16.1)  0.011 ms 10.0.18.1 (10.0.18.1)  0
 3  10.0.19.2 (10.0.19.2)  0.014 ms  0.010 ms  0.023 ms
 4  200.0.3.100 (200.0.3.100)  0.017 ms  0.012 ms  0.012 ms

-fibMultipathHashPolicy:4
-disableRPFilter:true
```

Ahora no importa si utilizamos los mismos puertos ya que la política de hash es ECMP por paquete y siempre balancea la carga.

B.1.2. Todos a todos y Todos a uno

Se pueden correr todos los experimentos realizados en el informe como un batch utilizando ./batch_tests.sh que corre el entorno secuencialmente bajo la configuración de cada parameters.json dentro de la carpeta batch_tests. Dentro de esta esta carpeta se encuentran todos los tests realizados. También se puede correr cada test manualmente copiando el .json correspondiente a la carpeta raíz y ejecutando sudo python3 fattree_bgp.py 4.

El archivo parameters_mp0.json corresponde con los tests para ECMP por flujo capa 3, parameters_mp1.json corresponde con los tests para ECMP por flujo capa 4, y parameters_mp4.json corresponde con los tests para ECMP por paquete. Los archivos comprenden tanto el caso todos a todos como el caso todos a uno.

Veamos una configuración de ejemplo (parameters_mp1.json).

```
{
"testName":"test2bmp1",
"testDescription":"Simple Byte based test with both incast and all to all scenarios",

"doCLI":false,
"withCaptures":false,

"fibMultipathHashPolicy":1,
"linkBW":20,

"doIperf3Test":true,
"useUDP":true,
"doTimeTest":false,
"iperfTargetBitrate":"20K",
"iperfTransmissionTime":"10",
"iperfBytes":"15K",
"iperfTimeout": 15,

"incastDoIperf3Test":true,
"incastUseUDP":true,
"incastDoTimeTest":false,
"incastIperfTargetBitrate":"20K",
"incastIperfTransmissionTime":"10",
"incastIperfBytes":"15K",
"incastIperfTimeout": 15,

"doTracerouteTest":false,
"disableRPFilter":true,

"doGraphCaptures":true,

"doTrafpyTest":false,
(...)
}
```

Se observa que están activados los test de todos a todos (doIperf3Test) y todos a uno (incastDoIperf3Test), mientras que el test con trafPy esta desactivado (doTrafpyTest) y la consola de mininet también (doCLI). Por otro lado, se utiliza un límite de bandwidth por enlace de 20Mb/s y que se utiliza fibMultipathHashPolicy=1, lo cual indica que la prueba utiliza ECMP por flujo capa 4. La prueba utiliza UDP, se transmiten 15KB a un bitrate de 20Kb/s. Esto se traduce a 11 paquetes con payload de 1448 bytes ya que iperf redondea 15KB de esta forma en su modo UDP. En el modo TCP es muy inpreciso ya que en cantidades pequeñas de tráfico por lo menos transmite algunos MB aunque se le pida que transmite algunos KB, es solo preciso a nivel de MB y es por esto que se utilizó UDP en estas pruebas. También para realizar el scheduler dado

por los flujos de trafpy se utilizó UDP por esta misma razón.

Como resultado de esta ejecución se obtienen las imágenes presentadas en el informe, los logs de ejecución de iperf3 para cada conexión, un resumen de estos en formato .txt, y las capturas de tráfico realizadas. Esto se obtiene para el test todos a todos y todos a uno de forma separada. Por ejemplo, el resumen para ECMP por paquete en el caso todos a todos es:

```
Test Info:
iperf transport (TCP/UDP): UDP
iperf type (Bytes/Bitrate): Bytes
iperf Target Bitrate: 20Kbps
iperf Transmission Time: N/A
iperf Bytes: 15KB
iperf Timeout: 15s
iperf command prefix: iperf3 --tos 0xb1 --bytes 15K --udp --bandwidth 20K

Test results:
Avg bytes per host: 0.015 MiB
Avg bitrate per host: 0.022 Mbps
Avg CPU utilization host_total: 0.55%
Avg Transmission time seconds: 5.83s
```

Se observa que la cantidad de bytes transmitidos en promedio por host es 0.015 MiB = 15.36KiB, esto cuadra con la cantidad realmente transmitida (1448B*11paquetes=15.55KiB) con un pequeño margen de error debido al redondeo de los números en el resumen. El bitrate también cuadra, 0.022 Mbps = 22.5Kbps lo cual es casi el bitrate pedido 20Kbps. La diferencia en este caso radica que no es tan exacta a nivel de bitrate la herramienta iperf3, en este caso transmitió un poco más rápido de lo pedido. Esto sucede porque transmitimos a velocidades en los Kbps, si fuera en los Mbps no se observaría la diferencia con los dígitos de precisión mostrados. Finalmente, no se puede confiar mucho en el uso de CPU en el host.

B.1.3. Una conexión con tráfico TrafPy de fondo

La configuración de las pruebas con trafpy realizadas son: parameters_trafpy_cloud_mp1.json y parameters_trafpy_cloud_mp4.json. Se puede correr de igual forma que las pruebas anteriores.

Estas pruebas leen el archivo con formato .csv dado por el parámetro trafpyTestData y emula los flujos ahí presentes iniciando una conexión iperf3 UDP con el tamaño de ese flujo en el momento indicado. Adicionalmente se ejecuta, en el medio de esta emulación, una conexión TCP de 10MB entre el primer y último host del *fat-tree*. Esto lo hacemos para analizar luego si hubo reorden de paquetes.

```
{
"testName":"testTrafpyCloud",
```

```
"testDescription":"Simple Trafpy Test",
(...)
"fibMultipathHashPolicy":4,
"linkBW":50,
(...)
"doTrafpyTest":true,
"trafpyTestData":"flow_centric_demand_data_commercial_cloud.csv",
"maxFlows":2000,
"trafpyIperfTimeout":30,
"speedFactor":50,
"trafpyIperfTargetBitrate":"30K"
}
```

Luego de ejecutarse, estas pruebas retornan un archivo con la captura del host receptor de la conexión TCP de 10MB ejecutada en el medio de este test. En la misma se observó que no hubo reorden de paquetes utilizando tshark o wireshark con el filtro tcp.analysis.out_of_order. Esto sucede tanto en la prueba para ECMP por paquete como ECMP por flujo capa 4 y tambien ECMP por flujo capa 3. Por esto es que decimos que mininet es un entorno que no reordena paquetes.

Por otro lado también retorna un archivo .json con el log de cada conexión iperf3 que dicta el archivo .csv que se haga (el cual está dentro de la carpeta trafpydata). Por ejemplo la linea 251 del csv `249,flow_2290,server_8,server_10,55425.0,1288.684` Indica que el flujo 249 tiene id flow_2290, y que el server_8(h9) debe enviarle al server_10(h11) 55425B en el milisegundo 1288.68. Queda registro de esta conexión en flow_2290.json donde se aprecia que se le pidió transmitir 55425B pero termino transmitiendo 56472 (ya que iperf3 reondea en múltiplos de 1448). También se observa en este log que el host con IP 200.0.8.100 (h9) se conecta al host con IP 200.0.10.100 (h11) tal como lo especifica el .csv.

Otras de las opciones de configuración como maxFlows limitan la cantidad de flujos emulados, en lugar de realizar todos los flujos del .csv se realizan algunos (ya que el benchmark puede llevar una cantidad considerable de tiempo si se hacen todos los flujos y es interesante realizar pruebas más rápidas). Para el propósito de probar esto demoró 320s aproximadamente. El speed factor es para acelerar o relentizar la prueba para que el ambiente lo tolere, un speed factor de 1000 significa tiempo real en milisegundos y 50 significa 20 veces más lento. Se eligió esta velocidad para que el tiempo de ejecución real de la prueba no sea mucho mayor al teórico dado por el .csv (el ambiente lo tolere). Ambos de estos parámetros modifican en tiempo de ejecución algo que se puede modificar a la hora de crear los .csv, y el resultado es básicamente el mismo (cuando se baja la capacidad de los endpoints se alarga en el tiempo la prueba de forma uniforme, pero usando los mismos flujos).

B.2. Experimentos en Kathará

Para realizar los experimentos en Kathará, primero se debe bajar la carpeta *kathara* del repositorio. Luego es necesario tomar de base la imagen de Docker "kathara/quagga" que utiliza por defecto Kathará para expandirla agregándole iperf3 y tcpdump, herramientas necesarias para que funcione el experimento. Para hacer esto se pueden seguir las instrucciones:

1. sudo docker pull kathara/quagga (bajar la imagen del registro dockerhub si no la bajo todavía Kathará)
2. `sudo docker run -tid --name <container_name> kathara/quagga` (correr un contenedor con el nombre `<container_name>` que use la imagen kathara/quagga)
3. `docker exec -ti <container_name> bash` (levantar una terminal bash dentro de este contenedor)
4. apt install iperf3 tcpdump net-tools
5. exit (cerrar la terminal bash)
6. `sudo docker commit <container_name>` kathara/quaggaiperf (se guarda la nueva imagen como kathara/quaggaiperf)
7. `sudo docker rm -f <container_name>` (se elimina el contenedor ya que no lo necesitamos más)

Luego de tener una imagen de kathara con iperf3 y tcpdump se puede correr la prueba que se hizo en Kathará con ./runtest.sh. Es necesario modificar las configuraciones de las rutas multicamino en los archivos .startup relacionados con r1, r11 y r12 de la forma que se describe en el informe para observar el fenómeno extraño. El laboratorio Kathará lab.conf contiene tanto los dominios de colisión como la configuración pertinente al tipo de ECMP. Este script simplemente levanta el laboratorio Kathará, espera 30 segundos y luego lo termina. Al terminar analiza las capturas de tráfico con tshark (que se realizaron en r21 y r22 con tcpdump en sus .startup) indicando la cantidad de paquetes que transitaron por cada camino, imprimiendo esto en la terminal.

Analizando los resultados al cambiar las configuraciones es que se determinó cual era el problema.

B.3. Experimentos con Máquinas Virtuales

Este entorno lleva un poco más de esfuerzo para recrear y es necesario partir de una VM con el kernel modificado. Por ejemplo, la VM que se indica como crear en el anexo A.2. Esta guía es para recrear el entorno usando VirtualBox. Es necesario agregar la imagen a VirtualBox y realizar unas modificaciones.

- Configurar una IP estática en una red de solo anfitrión a cada VM para poder conectarse por ssh. Con una sola VM funcionaba bien el dhcp pero con las VM clonadas fue necesario hacer que esta IP sea estatica. Para esto en Ubuntu 22.04.2 la mejor forma fue utilizar systemd netplan, entonces se modificó el archivo /etc/netplan/00-installer-config.yaml de esta forma:

```
network:
  ethernets:
    enp0s3:
      addresses:
        - 192.168.57.4/24
  version: 2
```

- Utilizar cron para que cada vez que se reinicia una VM se ejecute el script de inicio startup.sh directamente de la carpeta home del usuario.

```
sudo crontab -e ; @reboot sh /home/pgreenvio/startup.sh
>> /var/log/myjob.log 2>&1
```

- Posteriormente a cada VM se le pasa el startup.sh con scp (el script copystartup.sh copia todos los startup.sh de todos los nodos) y este se ejecuta al iniciar la VM.

Una vez lista la VM hay que clonarla 8 veces (una para cada nodo de la topología). Se recomienda realizar una clonación enlazada para que ocupe menos espacio en el disco, botón derecho en la VM base y clonar. Luego, como indica la Figura B.1, se selecciona clonación enlazada y se generan nuevas MACs para todas las interfaces.

Luego para crear los dominios de colisión es necesario entrar a la configuración de red de la VM y conectar cada VM a las redes internas correspondientes. Por ejemplo, para el dominio de colisión A, para r1 en la Figura B.2.

Luego de agregar los tres dominios de colisión para r1 se ve como en la Figura B.3. Es necesario que el orden de los dominios sea el correcto, indicado por la topología (Figura 3.2), es decir eth0 es el segundo adaptador, eth1 el tercero y eth2 el cuarto. Mientras que el primer adaptador se utiliza para conectarse por ssh a la VM, es necesario asignar una IP distinta a cada VM con la configuración netplan mencionada anteriormente. Este paso debe repetirse para todas las VMs clonadas.

Ahora es necesario modificar copystartups.sh y runtest.sh agregando las IPs estáticas de estas máquinas virtuales, o en su defecto utilizar estas IPs a la hora de configurar las IPs estáticas. Esto se hace para que los scripts puedan conectarse a la VM correctamente y orquestar la ejecución de las pruebas o copiar los startup.sh correspondientes a cada VM (scripts dentro de la carpeta startups con los nombres de los nodos).

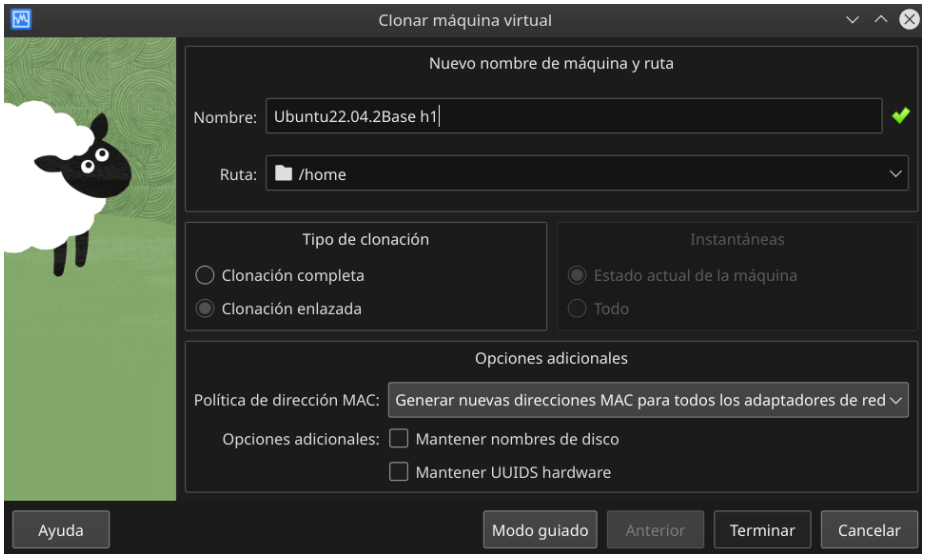

Figura B.1: Instrucciones para clonar la VM

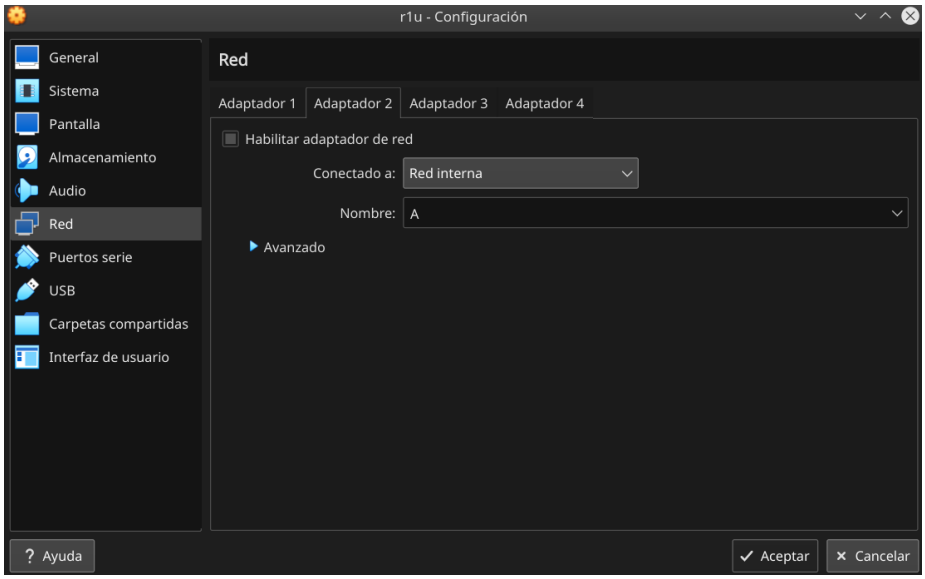

Figura B.2: Agregar dominio de colisión A a la VM r1

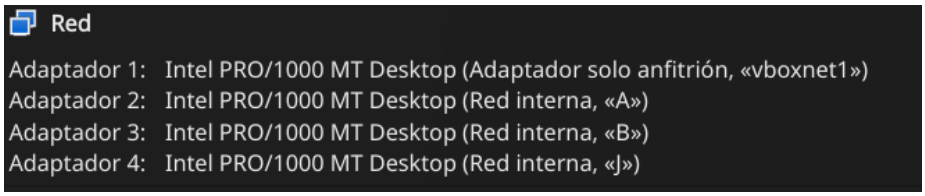

Figura B.3: Todas las interfaces de r1

Test Informe	Carpeta Test
Primera prueba	test0
Segunda prueba	test1.m0, test1.m1, y test1.m4
Tercera prueba	test2
Cuarta prueba	test3
Quinta prueba	test4
Sexta prueba	test5.m0, test5.m4

Tabla B.1: Asociación entre secciones del informe y pruebas presentes en el repositorio para el entorno de VMs

Posteriormente a todo este trabajo de setup, es posible ejecutar los tests realizados en este informe de la siguiente forma:

1. Ejecutar copystartups.sh para copiar todos los scripts de inicio (esto solo es necesario hacerlo una vez)
2. Reiniciar las VMs
3. Ejecutar `./runtest.sh <test\_name>` (test_name es el nombre de la carpeta donde se encuentran los scripts que deben correr cada VM, este script es el encargado de que se ejecutan en el orden que se menciona en el informe, adicionalmente una vez iniciadas las VMs se pueden correr todos los test que se deseen y no es necesario reiniciar para realizar cada test)

La correspondencia entre pruebas en el informe y los tests en la carpeta vms es la de la tabla B.1.