UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Taller de Sistemas Ciberfísicos

Nicolás Giossa - Santiago Góngora

13 de diciembre de 2021

Índice

1. Introducción

El enrutamiento en los datacenters es un problema abierto, en constante revisión y de gran relevancia actualmente. La cantidad de nodos en un datacenter puede, a día de hoy, alcanzar el orden de los pocos miles de nodos de conmutación (switches o routers) y cientos de miles de servidores, por lo que contar con protocolos de enrutamiento adecuados es primordial. Sin embargo, para probar el funcionamiento y robustez de estos protocolos no es viable realizar *tests* en datacenters reales, por lo que la práctica corriente es realizar emulaciones de sus nodos y su comportamiento.

En este contexto es que se desarrolla nuestro proyecto, que busca explorar diferentes criterios de convergencia para protocolos de enrutamiento y, así, finalizar la emulación de la topología para ahorrar recursos. Es de relevancia aclarar que nos centraremos puramente en resolver la detención de la emulación al finalizar la etapa de *boostrap*, cuando los nodos consolidan sus primeras tablas de *forwarding*, por lo que no realizamos pruebas de caídas de enlaces/nodos y rearmado de las tablas.

Presentaremos entonces el desarrollo de un *framework* que permite construir una topología, emularla y detenerla según diferentes criterios de parada. Todo el código desarrollado se encuentra disponible en `https://gitlab.com/fing-mina/datacenters/tscf-stop-criteria`.

En la sección 2 haremos una breve descripción de conceptos previos que manejamos a lo largo del proyecto. En la sección 3 presentaremos rasgos generales del *framework* desarrollado. En la sección 4 detallaremos los criterios de parada explorados. Finalmente en la sección 5 mostraremos los resultados experimentales de estos criterios desarrollados y en la sección 6 daremos las conclusiones y mencionaremos algunas líneas de trabajo para el futuro.

2. Conceptos previos

Previo a comenzar con los detalles de nuestro proyecto introduciremos algunos conceptos que utilizamos, como el entorno *Kathará* o la topología sobre la que trabajamos, entre otros.

2.1. Fat Tree

La topología *Fat Tree* es un tipo especial de las topologías Clos. Esta topología se organiza típicamente en el agrupamiento de nodos, mediante los llamados *Point of Delivery* (PoD), y una capa de agregación entre ellos, por los llamados *Top of Fabric* (ToF).

Según la ubicación de los nodos, se pueden también identificar direcciones. Hacia el *Norte* tenemos los *ToF*, que están más proximos que el resto de nodos a la “salida hacia internet” (i.e.

el exterior del Datacenter); hacia el *Sur* tenemos los servidores conectados a los *PODs*. Por último podemos hablar de las direcciones *Este* y *Oeste* para referirnos al tráfico entre PODs. Esta noción de posición va a quedar más clara al ver la figura 2.1.

A nivel de individual los nodos se clasifican en cuatro categorías:

- Top of Fabric (ToF) - Estos nodos tienen por objetivo comunicar el tráfico entre PoDs. Se encuentran ubicados al norte. Los nodos ToF **no pertenecen** a ningún *PoD*.
- Spine - Su objetivo es ser el enlace entre los ToFs y los *leaf*, por lo que se encuentran ubicados en el centro de la topología. A aquellos *spine* que se encuentran directamente conectados a algún ToF, se les llama *Top of PoD*. Cada spine **pertenece** a uno y solo un único *PoD*.
- Leaf - Su objetivo es conectar los servidores con el resto del *fat tree*. Se encuentran ubicados al sur y, al igual que los *spine*, pertenecen a uno y solo un *PoD*.
- Servidores - Los servidores son el origen o destino final del tráfico que circula por la topología.

Las diferentes topologías quedan determinadas al variar la cantidad de nodos, de conexiones entre ellos, la cantidad de PoDs, la cantidad de nodos por *PoD*, entre otros factores. Una de esas características es si el *Fat Tree* es ***single plane*** (un plano) o ***multiplane*** (varios planos).

En el caso de *single plane*, cada uno de los *Top of PoD* está conectado a cada nodo *ToF*. Este tipo de organización permite una alta redundancia en la conectividad, lo que lo hace más robusto a fallos. Por otra parte, el *multiplane* reduce esta redundancia en pos de lograr escalar la cantidad de nodos y aprovechar mejor su conectividad.

Más allá de las configuraciones mencionadas, una manera de especificar un *Fat tree* multiplano es mediante un natural k **par**[1], de la siguiente manera:

- k nodos por *PoD*.
- k *PoDs*
- $planos = \frac{k}{2}$

Otra característica que está presente al diseñar la topología es la cantidad de puertos que tiene cada nodo hacia el sur (k_{leaf}) y hacia el norte (k_{top}). Con la notación del k natural tenemos

[1] k debe ser mayor o igual a 2 para que la característica multiplano tenga sentido.

que $k_{leaf} = k_{top} = \frac{k}{2}$.

En la imagen 2.1 vemos un *Fat Tree* multiplano especificado mediante un natural $k = 4$. Observemos que la cantidad de planos en ella es 2, porque cada *ToP* está conectado con la mitad de nodos *ToF*. En una topología de un plano (*singleplane*) cada *ToP* estaría conectado con cada uno de los *ToF*. Por lo tanto, la decisión de la cantidad de planos a usar está fuertemente ligada con la redundancia que se quiera tener.

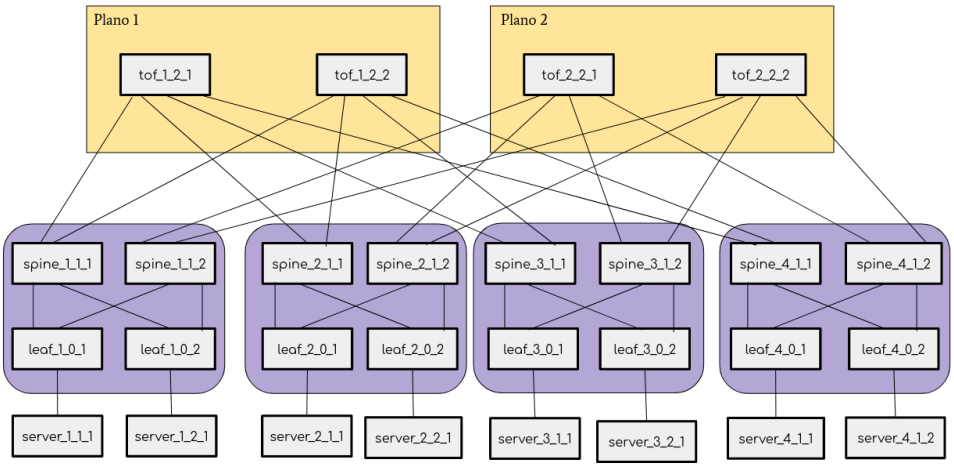

Figura 2.1: *Fat tree* con $k = 4$. Se aprecian los $k/2$ planos y los k PoDs de k nodos.

2.2. Equal Cost Multipathing (ECMP)

Típicamente en una topología se suele calcular teóricamente la ruta de costo mínimo entre un par de nodos. En caso de haber más de una ruta basta quedarse con una sola, puesto que no es de principal interés contar con el catálogo de rutas, sino calcular el costo entre esos nodos. Para resolver este problema se suele utilizar el conocido *Algoritmo de Dijkstra* [1].

Sin embargo, algo fundamental para el ruteo en los datacenters es conseguir todas las rutas de costo mínimo entre cada par de nodos. Es decir, cobra relevancia hallar la redundancia de caminos entre dos nodos, todos de igual prioridad, de modo de poder ser robustos ante fallas o caídas de enlaces pero, sobre todo, para sumar **ancho de banda**. Esta manera de enrutar paquetes es conocida como *Equal-cost multi-path routing* (ECMP).

En este caso el algoritmo original de Dijkstra no funciona, pero puede ser modificado de manera de poder hallar todas las rutas. Más adelante veremos que este algoritmo modificado será imprescindible para resolver uno de los criterios explorados, y que usaremos una

implementación disponible en la biblioteca *NetworkX* [2].

2.3. Kathará

Kathará[3] es un entorno de emulación de redes basado en *docker* desarrollado por la Universidad *Roma Tre*. Este entorno permite emular nodos y su comportamiento, por lo que es ideal para probar protocolos o diferentes escenarios de red.

El entorno es muy grande y permite operar con muchas opciones, como por ejemplo las imágenes que corren los contenedores. Esta herramienta es una parte esencial del *framework* construido por las capacidades de emulación que provee.

En la siguiente sección describiremos el *framework* construido y detallaremos qué funcionalidades de Kathará utilizamos.

3. Framework

El componente en el que trabajamos es un *framework* que permite centralizar y automatizar el uso de varias herramientas con el objetivo de correr experimentos en topologías *Fat Tree*. En esta sección detallaremos el funcionamiento de este *framework*.

3.1. Creación de Fat Tree según K

En primer lugar, para que el *framework* pueda correr experimentos emulados en Kathará, se necesita tener una representación de la topología concreta a emular.

El componente *VFTGen* [4] permite generar una topología *Fat Tree* — para ser emulada en Kathará — a partir de un archivo de configuración, el cual debe ser modificado manualmente. Como vimos en la sección 2.1, la topología tiene muchos componentes que pueden ajustarse, pero hay un subconjunto de *Fat Trees* que pueden ser especificados según un k par.

Como en general la idea es trabajar con *Fat Trees* que pueden ser expresados mediante un k, nuestro *framework* automatiza el proceso de creación con *VFTGen* utilizando dicho número como argumento mediante la terminal. Por lo tanto, una vez iniciado el *framework*, se invoca al componente *VFTGen* con el k pasado como argumento en la terminal, lo que genera la topología lista para ser emulada con Kathará. Por ejemplo:

```
python exp0.py -k=4
```

[2] https://networkx.org
[3] https://www.kathara.org/
[4] https://gitlab.com/uniroma3/compunet/networks/sibyl-framework/fat-tree-generator

VFTGen también permite elegir el protocolo de enrutamiento a usar en la topología. En nuestro caso trabajamos siempre con BGP [2], uno de los protocolos más usados en los datacenters a día de hoy, más allá de que haya sido originalmente diseñado para enrutar paquetes entre sistemas autónomos. Actualmente está en desarrollo otro protocolo llamado RIFT (*Routing in Fat Trees*) [3] para el enrutamiento en los datacenters que usan topologías *Fat tree*. *VFTGen* también tiene soporte para RIFT, pero no alcanzamos a experimentar con él en este proyecto.

3.2. Cambios en VFTGen

Utilizamos un *fork*[5] de la herramienta VFTGen, sobre el cual hicimos algunas modificaciones que detallaremos a continuación.

Modificamos el método `write_startup` de la clase `Laboratory` para que imprima una invocación a *tshark* por cada interfaz en el *.startup* de cada uno de los nodos. Esto es utilizado para el chequeo de la ventana deslizante, que detallaremos en la sección 4.2.

En los scripts de los criterios distribuidos se modifica el *.startup* de los nodos para incluir una invocación al demonio que debe ejecutar en su inicialización. A su vez en se modifica el archivo `lab.conf`, seteando la *flag* `bridged` en `true`, para habilitar la comunicación entre los nodos y el controlador principal, como se explica en la sección 4.1.1.

Por otra parte, modificamos el método `_configure_node` de la clase `BgpConfigurator`, de forma de que cada nodo tenga asignada una imagen de Docker personalizada que construimos, cuyo identificador es `kathara/frr-tscf2021`.

El Dockerfile de la imagen construida se basa en la imagen `katara/frr`[6] y agrega algunas dependencias mediante `apt install`: `tshark`, `python3.7` y `python3-pip`. Se instala también el paquete `iproute2` desde el repositorio `buster-backports` ya que, en la versión que viene en Debian 10, el comando `ip` con la flag `-json` generaba JSON mal formados en algunos casos. A su vez se agregan algunos paquetes de Python mediante pip: `python-daemon`, `pyshark` y `nest-asyncio`.

3.3. Comunicación con Kathará

La comunicación con la herramienta de emulación Kathará se realiza principalmente mediante la función `subprocess` nativa de Python, que permite ejecutar comandos en terminal programáticamente. Por ejemplo, para iniciar el laboratorio creado, se invoca

[5] Fork a partir del commit: bd0382710231a7961bd2efb30a1fad338f9986b0

[6] https://github.com/KatharaFramework/Docker-Images/blob/c9617a91bc8ea821d5fd47170c29af7a88b46110/debian10/frr/Dockerfile

```
subprocess.run(["kathara", "lstart", "--noterminals"])
```

Para solicitar la tabla de *forwarding* de un nodo

```
subprocess.run(["kathara", "exec", node_id,
      "--", "ip", "-json", "route"]
```

Para realizar un `ping` entre dos nodos

```
subprocess.run(["kathara","exec",node_id_a,"--","ping","node_id_b,"-c","1"]
```

Para finalizar la ejecución

```
subprocess.run(["kathara", "lclean"])
```

Evaluamos utilizar una API de Kathará existente para Python[7] pero la cantidad de configuraciones que teníamos que hacer era mucho más grande y dificultaba la automatización que busca hacer el framework.

Para la comunicación entre los nodos emulados y el código principal del *framework* utilizamos paquetes TCP, como detallaremos en la sección 4.1.1.

[7] https://github.com/KatharaFramework/Kathara/wiki/Kathara-Python-API

3.4. Vista general

En la figura 3.1 podemos ver un diagrama sobre el flujo de ejecución y componentes del *framework*.

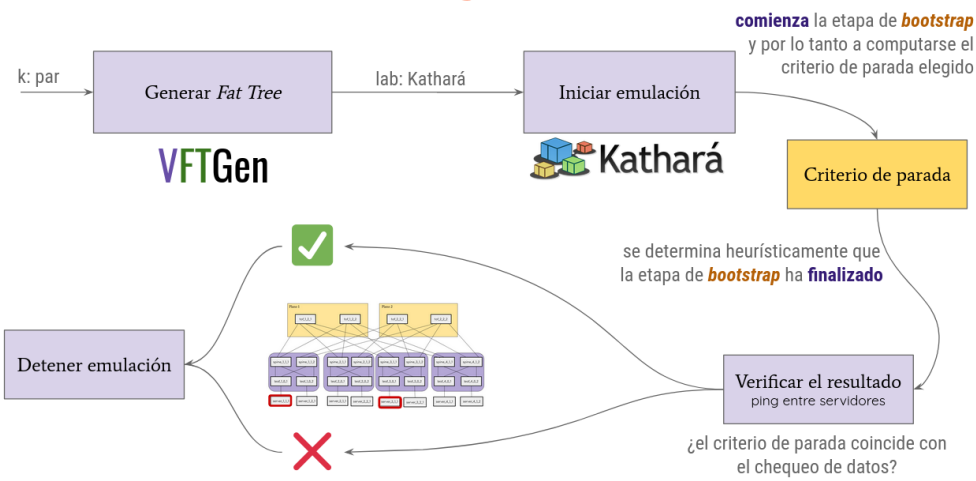

Figura 3.1: Vista general del *framework* construido.

Como vemos, siempre inicia creándose el *Fat Tree* con *VFTGen* según un *k* par, para luego comenzar la emulación de esa topología usando Kathará.

En amarillo podemos ver el componente del criterio de parada, que es lo único que varía entre los diferentes experimentos realizados. Posteriormente se verifica el resultado del criterio de parada mediante `pings` entre **cada par de servers**. En la sección 5.1 veremos un análisis teórico sobre este chequeo. Luego de terminado el chequeo, la topología se detiene.

En la siguiente sección veremos en detalle los criterios de parada desarrollados, que se corresponderían al único elemento del diagrama anterior que cambiaría en cada caso.

4. Criterios de parada

Los protocolos de enrutamiento son diseñados sin un criterio de parada, puesto que es necesario que sigan corriendo para lograr rearmar las tablas de *forwarding* ante cualquier cambio en la topología; es decir, para que sean robustos ante cambios en la topología. Sin embargo al estar en una emulación es importante que pueda detenerse en algún momento para concluir los experimentos y así no ocupar recursos innecesariamente. Esto introduce el problema de establecer un criterio de parada para estos protocolos, donde hay varias soluciones

posibles.

Una de las opciones, que manejaremos en este trabajo, es la de detener la topología cuando ella converge (i.e. termina la etapa de *bootstrap*). Esto puede ser determinado cuando las tablas de *forwarding* quedan estables y no se intercambian más mensajes de actualización, sino solo mensajes del tipo *keepalive*.

Apoyado en esto último es que trabajaremos con dos criterios de parada: la **comparación de tablas de forwarding** y el **chequeo de la ventana deslizante**.

4.1. Comparación de tablas de forwarding

El primer experimento realizado consistió en comparar las tablas de *forwarding* esperadas con aquellas obtenidas por el demonio BGP corriendo en la emulación con Kathará. Este experimento fue realizado tanto de manera **centralizada** como **distribuida**.

Para hallar las tablas esperadas utilizamos la implementación de *NetworkX* del algoritmo de Dijkstra en versión ECMP [8]. El primer paso es parsear el archivo `lab.json` que crea VFTGen, donde se indican los nodos que conforman la topología, junto a sus interfaces y los nodos vecinos por cada una de ellas. Con la información de los nodos y sus vecinos se crea un grafo con la mencionada biblioteca *NetworkX* y se corre ECMP sobre él, lo que da como resultado todos los caminos más cortos de un nodo al resto. Esto da, desde otro punto de vista, la tabla de *forwarding* del nodo, por lo que decimos que esta es la tabla de *forwarding* calculada teóricamente.

Una vez obtenidas estas tablas teóricas, comparamos sus entradas con las que calculó el demonio BGP de cada nodo, mediante el comando

```
kathara exec nombre_nodo -- ip -json route
```

En el enfoque centralizado se cuenta con un *script* que le pide a cada nodo las tablas de *forwarding* y las compara contra las calculadas con ECMP. En caso de coincidir se considera que el nodo ha alcanzado la convergencia. En caso contrario se continúa el chequeo secuencial del resto de nodos. Una vez terminado el chequeo secuencial se vuelve a chequear la lista de nodos, hasta que todos hayan llegado a la convergencia.
Podemos notar que en este caso tenemos un *polling* centralizado: el controlador principal le pide a cada nodo la tabla hasta que converja.

[8] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.generic.all_shortest_paths.html#networkx.algorithms.shortest_paths.generic.all_shortest_paths

Por otra parte, en el enfoque distribuido, se cuenta con un controlador central que es el encargado de chequear que todos los nodos hayan avisado de su convergencia. Cada nodo calcula su tabla teóricamente con ECMP y cuando su tabla coincide, le avisa al controlador. En este caso el *polling* se hace en cada nodo: periódicamente el nodo chequea si alcanzo la convergencia o no. A continuación describiremos con mayor detalle la implementación de este enfoque.

4.1.1. Detalles de implementación del enfoque distribuido

Cada nodo ejecuta un demonio al inicializarse, el cual se encarga de hacer la comparación entre la tabla esperada y la actual, y notifica al controlador principal una vez que se dé la convergencia. Para esto, los nodos de la topología necesitan:

- Tener acceso a su tabla de *forwarding* teórica, idealmente ya calculada
- Tener acceso al código del demonio a ejecutar
- Ejecutar el demonio al inicializar
- Un mecanismo de comunicación entre el nodo y el controlador

Existe una fase previa a iniciar la emulación donde el controlador provee los tres primeros puntos a los contenedores que simulan los nodos de la topología. En esta fase el controlador calcula las tablas teóricas de todos los nodos con ECMP[9] y las escribe en un directorio compartido. A su vez, copia el código necesario para ejecutar el demonio a ese mismo directorio compartido, y altera el archivo `.startup` de cada nodo del laboratorio de Kathara, agregando el comando para la ejecución del demonio.

En cuanto al mecanismo de comunicación, el controlador inicia un servidor TCP, al que los nodos se conectan y envían un mensaje una vez que convergen según el criterio de la comparación de tablas. Para lograr conectividad entre la máquina host (donde ejecuta el controlador) y los contenedores (donde ejecutan los nodos de la topología), se habilita el *flag bridged*[10] en el archivo de configuración del laboratorio de Kathara (`lab.conf`). Este *flag* agrega una interfaz de red virtual a cada uno de los nodos mediante la cual se pueden comunicar con el nodo host.

Como ya mencionamos, el demonio ejecutado en los nodos utiliza una estrategia de *polling* obteniendo su tabla de *forwarding* mediante el comando `ip route` cada *N* segundos (valor configurable). Una vez que la tabla obtenida coincide con la esperada, se conecta al servidor TCP y envía un mensaje con su identificador, para luego finalizar su ejecución.

[9] Usando NetworkX, de la misma manera descrita para el enfoque centralizado
[10] https://www.kathara.org/man-pages/kathara-lab.conf.5.html

4.2. Chequeo de la ventana deslizante

El segundo experimento consistió en realizar un chequeo de ventana deslizante entre los paquetes recibidos por cada interfaz de cada nodo. Este experimento también se hizo en ambas modalidades, **centralizada** y **distribuida**.

La idea principal de este método es verificar que la cantidad de **paquetes de control** que un nodo envía/recibe conforme una proporción muy baja de entre el total de paquetes que envía/recibe. La cantidad de paquetes que se toma en cuenta para este chequeo está dado por el parámetro "ventana" (*window size*) y la mencionada proporción por el parámetro "umbral"(*threshold*).

Cada protocolo cuenta con diferentes mensajes de control; en particular para BGP optamos por utilizar los mensajes UPDATE, que en el protocolo cuentan con el identificador "2"[11].

En el pseudocódigo 1 se muestra el chequeo utilizado. La función $contarUPDATES$ cuenta todos los paquetes con $id = 2$ en la ventana de tamaño w de la interfaz i, formada por $consolidarVentanaEnInterfaz$.

Algoritmo 1 Chequeo de ventana flotante

Parámetros w tamaño de la ventana; t valor del umbral
$UPDATES = 0$
for interfaz i del nodo **do**
  $UPDATES \leftarrow UPDATES + contarUPDATES(consolidarVentanaEnInterfaz(w,i))$
end for
$average \leftarrow \frac{UPDATES}{w * cant\_interfaces\_nodo}$
return $average < t$

Como puede verse en el pseudocódigo, el chequeo se hace sobre la totalidad de mensajes que caben en ventanas de cada interfaz. Por esa razón es que el *average* se calcula sobre $w * cant\_interfaces\_nodo$, ya que el conteo de paquetes de control $UPDATES$ será proporcional a la cantidad de interfaces.

4.2.1. Implementación y resultados posibles

El método teórico describe el cálculo principal sobre la ventana. Sin embargo, en la implementación tomamos algunas decisiones que son relevantes para ser documentadas.

[11] https://www.iana.org/assignments/bgp-parameters/bgp-parameters.xhtml

En primer lugar, los paquetes que conforman las ventanas son leídos de los archivos `.pcap` que guarda cada nodo por cada una de sus interfaces. Estas capturas se guardan por la emulación misma del nodo utilizando *tshark*. Aquí hay un problema de concurrencia a tener en cuenta, pues los mismos archivos que son escritos por los nodos emulados son leídos[12] para determinar la convergencia del nodo. Comentaremos este problema en la sección 5 al analizar los resultados experimentales.

En segundo lugar, la implementación no maneja dos[13] sino cuatro resultados posibles:

- Código 0: El nodo aún no ha alcanzado la convergencia.
- Código 1: El nodo ha alcanzado la convergencia.
- Código 2: El nodo no contiene suficientes paquetes para considerar una ventana.
- Código 3: El nodo no contiene suficientes paquetes para considerar una nueva ventana.

Tanto el código 0 como el código 1 dan su resultado en función de la comparación $average < t$. Sin embargo, los códigos 2 y 3 son propios de la implementación, por lo que los explicaremos más en detalle.

El código 2 cobra relevancia al inicio de la emulación. Un nodo que recién está comenzando a recibir los primeros paquetes puede no haber recibido suficientes como para poder completar una ventana de tamaño w. Digamos, por ejemplo, que la ventana es de tamaño $w = 7$ y el nodo ha recibido por una de sus interfaces solo 6 paquetes. En este caso no es posible realizar el cálculo del *average*, por lo que el nodo aún no llegó a la convergencia. Como se puede notar, este caso de no convergencia es diferente al del código 0, ya que en éste sí se podía calcular el *average*. En la figura 4.1 se ilustra el anterior ejemplo.

Figura 4.1: Caso donde se devuelve código 2, considerando una ventana de tamaño $w = 7$.

[12] En el enfoque distribuido cada nodo lee sus propias capturas. En el enfoque centralizado el controlador principal lee las capturas de todas las interfaces de cada nodo, al mismo tiempo que los nodos emulados las están escribiendo.

[13] Recordemos que el método teórico da como resultado un *boolean*.

El código 3 cubre un caso similar de ventana no formada, pero esta vez para cuando la topología ya está intercambiando mensajes activamente. Para la implementación buscamos que las ventanas consideradas para cada interfaz de cada nodo no consideren **ningún** paquete repetido. En la figura 4.2 se ve un caso para una ventana de tamaño $w = 7$. Supongamos que la primera ventana considerada fue sobre los paquetes 4, 5, 6, 7, 8, 9, y 10, y que el resultado de ese chequeo fue que *aún no alcanzó la convergencia*. Entonces la segunda vez que se invoca al chequeo se hace con los paquetes 9, 10, 11, 12, 13, 14 y 15[14]. En este caso hay dos paquetes que ya fueron considerados, el 9 y el 10, por lo que habrá que esperar para conformar otra nueva ventana.

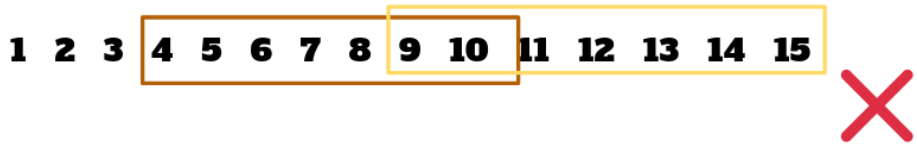

Figura 4.2: Caso donde se devuelve código 3, considerando una ventana de tamaño $w = 7$.

En la figura 4.3 se puede ver que cuando llegue el paquete número 17 por esa interfaz se podrá crear una nueva ventana sin superposición.

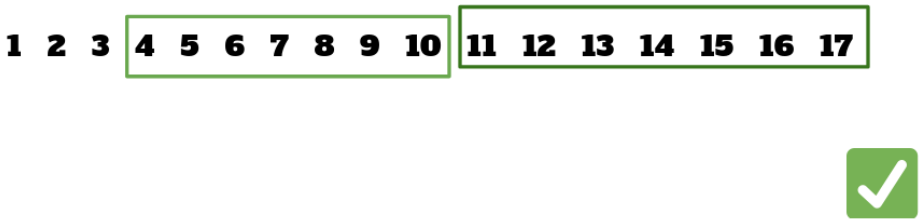

Figura 4.3: Caso de ventanas no superpuestas, considerando una ventana de tamaño $w = 7$.

4.2.2. Enfoque centralizado y distribuido

Como comentamos previamente, este criterio de convergencia se implementó tanto de manera **centralizada** como **distribuida**.

En el enfoque **centralizado** un controlador principal ejecuta el chequeo de ventana deslizante para cada nodo. Es decir, más concretamente, se invoca al chequeo de la ventana

[14] La ventana **siempre** se conforma con los últimos w paquetes. En este caso el último era el 15 así que desde allí, hacia atrás, se consideran los paquetes.

deslizante en cada archivo `.pcap` asociado a cada nodo. Si el código de retorno es 1, el nodo converge. En cualquier otro caso[15] el nodo suma una unidad a su lista de intentos fallidos. Cuando el nodo alcanza cierto valor constante predefinido en una constante (de nombre $MAXIMUM\_FAILED\_ATTEMPTS$), se considera que llegó a la convergencia. Este mecanismo de conteo de intentos fallidos surge de no posponer indefinidamente la terminación de la emulación de la topología, objetivo máximo a alcanzar por los criterios de parada.
Este chequeo se hace primero para los nodos *leaf* y, luego de que todos hayan convergido (ya sea por código 1 o por cantidad de intentos fallidos), se realiza para el resto de nodos. Esto busca optimizar el rendimiento, puesto que una vez que convergen los nodos *leaf* suele converger el resto. Recordemos que si un nodo no ha llegado a la convergencia, el chequeo se vuelve a hacer para él, por lo tanto esta estrategia minimiza los rechequeos y, por tanto, la cantidad de chequeos a realizar.

En el enfoque **distribuido** esta priorización de los nodos *leaf* por sobre el resto no está presente, como es de esperar. En este caso cada nodo realiza su chequeo y cuando converge, por código 1 o cantidad de intentos fallidos, avisa al controlador principal utilizando un paquete TCP, al igual que en el enfoque distribuido del criterio de comparación de tablas (ver sección 4.1.1). Una vez todos los nodos le notificaron de su convergencia, el controlador principal detiene la emulación.

Luego de finalizada la emulación, tanto para el enfoque **centralizado** como el **distribuido**, se verifica el resultado de convergencia con el criterio de comparación de tablas. De esta manera se puede verificar si ambos criterios dan, o no, el mismo resultado para la misma emulación de la misma topología.

5. Resultados experimentales

De manera de realizar una primer medición sobre el desempeño de los cuatro enfoques desarrollados, los probamos sobre dos topologías: $k = 4$ y $k = 6$. Todos estos experimentos se corrieron sobre el servidor *romain* del grupo MINA, que cuenta con $98,3$ GB de memoria RAM, 4 GB de espacio de Swap y 26 núcleos de CPU.

En los enfoques de ventana deslizante, centralizado o distribuido, se utiliza $MAXIMUM\_FAILED\_ATTEMPTS = 30$ de modo de darle tiempo a la topología a que logre estabilizarse antes de converger por intentos fallidos (ver descripción de este caso en la sección 4.2.2).

Los valores reportados en esta sección representan el tiempo de ejecución del criterio

[15] Cualquier otro código de retorno: 0, 2 o 3.

de convergencia correspondiente. Esto es desde inmediatamente después que *Kathará* logra iniciar la emulación[16] hasta que el criterio de parada decide finalizarla.

A continuación vemos las tablas de resultados. Para los chequeos de *comparación de tablas*, el valor de *Ventana* y *Umbral* no corresponde, por lo que aparece un "−" en sus celdas.
Las columnas "*vs. Tablas*" y "*vs. Datos*" muestran si los *chequeos de comparación de tablas* y `pings`, correspondientemente, coinciden con el resultado de convergencia para los experimentos de *ventana deslizante*. Por esa razón es que en los experimentos de *comparación de tablas* no hay un valor ya que no tiene sentido corroborar el resultado del método consigo mismo.

Criterio	k	Ventana	Umbral	Tiempo	vs.Tablas	vs.Datos
Comparación de tablas - Centralizado	4	–	–	00:00:39	–	Sí
Comparación de tablas - Distribuido	4	–	–	00:02:10	–	Sí
Ventana deslizante - Centralizado	4	10	0,2	00:22:47	Sí	Sí
Ventana deslizante - Distribuido	4	10	0,2	00:38:02	Sí	Sí

Cuadro 1: Resultados para los criterios de parada sobre una topología *Fat Tree* de $k = 4$.

Criterio	k	Ventana	Umbral	Tiempo	vs.Tablas	vs.Datos
Comparación de tablas - Centralizado	6	–	–	00:01:33	–	Sí
Comparación de tablas - Distribuido	6	–	–	00:04:43	–	Sí
Ventana deslizante - Centralizado	6	10	0,2	01:00:20	Sí	Sí
Ventana deslizante - Distribuido	6	10	0,2	00:41:42	Sí	Sí

Cuadro 2: Resultados para los criterios de parada sobre una topología *Fat Tree* de $k = 6$.

Como puede notarse, el enfoque de la ventana deslizante es muchísimo más lento que el de comparación de tablas. Si bien no realizamos un *testing* a fondo, podemos ver que hay diversas razones. En primer lugar, la cantidad de operaciones y la complejidad de la solución es mayor. En segundo lugar, las capturas son guardadas en archivos `.pcap` por los nodos emulados, al mismo tiempo que el controlador (o el propio nodo, en el enfoque distribuido) los está abriendo para analizarlos. Esto involucra *locks* utilizados por el propio sistema operativo, de manera de no corromper los archivos y los resultados de los métodos. En tercer lugar, mientras el criterio realiza los chequeos lentamente, en la topología continúan enviándose mensajes, lo que hace que los archivos `.pcap` sean cada vez más grandes y por lo tanto más difíciles de procesar. En último lugar, todo el criterio depende de la biblioteca *PyShark*, ya que es usada para abrir y analizar las capturas de **cada una de las interfaces de cada nodo**. Esto introduce dos problemas:

- No tenemos control sobre cómo *PyShark* abre el archivo y qué otros chequeos realiza.
- Cuanto más lejos se esté del inicio de la emulación, mayor tamaño tendrá la captura y, por lo tanto, más costoso será leerla para analizarla.

[16] En topologías grandes, como $k = 10$, el inicio de la emulación puede tardar mucho tiempo, llegando incluso a la media hora.

Por último, el propio *PyShark* depende de *TShark* que, frecuentemente, presenta inestabilidades. En la figura 5.1 podemos ver uno de los típicos fallos propios de *TShark*, independiente del uso de nuestro *framework*.

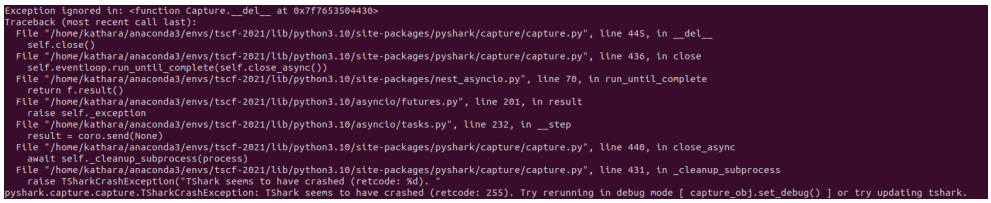

Figura 5.1: Error frecuente de Tshark al intentar abrir una captura.

Todos los problemas descritos en el párrafo anterior se evidencian al probar cualquiera de los enfoques de la *ventana deslizante*, donde la mayoría de las veces los nodos convergen con $average = 0$. Como el método es muy pesado, cuando logra llegar a analizar el tráfico de los nodos, el *bootstrap* ya ha finalizado.

Por último, también intentamos correr los experimentos sobre un $k = 10$. De los cuatro, solo el de comparación de tablas centralizado logró converger en un tiempo aproximado de 4 minutos. El criterio de la ventana deslizante distribuido corrió durante 96 horas y no logró llegar a la convergencia[17]. Por otra parte, ambos enfoques distribuidos tuvieron problemas con la notificación del nodo emulado al controlador principal, por lo que los nodos emulados convergían pero no podían avisar y por lo tanto nunca se determinaba la finalización de la etapa de *bootstrap*.

5.1. Análisis teórico del chequeo de ping

Si bien todos los experimentos expuestos en las tablas presentan el tiempo medido **solo con respecto a la demora en calcular el criterio**, es importante aclarar que el control de datos demora mucho más tiempo.[18] Este alto costo se debe a que la cantidad de servidores crece exponencialmente según k y el ping entre ellos se vuelve cada vez más costoso.

En las topologías *Fat Tree* que hemos trabajado tenemos siempre un servidor por cada *leaf*, por lo que la cantidad de servidores es igual a la de nodos *leaf*. Además, en este tipo de topologías tenemos dos niveles por POD: uno de nodos *spine* y otro de *leaf*.
Ya que tenemos k PODs de k nodos (ver sección 2.1), la cantidad de *servidores* según k es:

[17] En este caso los archivos *.pcap* llegaron a pesar 48 megabytes cada uno, lo que hacía muy larga la carga de todas las capturas del nodo.

[18] Por ejemplo, para $k = 6$ y 5 paquetes usados para el *ping*, llegó a demorar más de una hora y media. Por esta razón, el *ping* lo realizamos enviando solo un paquete, de manera de no encarecer aún más el chequeo, utilizando el método descrito en la sección 3.3.

$$cant\_servers = cant\_leaf\_en\_POD \times cant\_PODs = \frac{k}{2} \times k = \frac{k^2}{2}$$

Entonces la cantidad de *pings* a realizar es:

$$cant\_servers \times cant\_servers - cant\_servers = cant\_servers^2 - cant\_servers = (\frac{k^2}{2})^2 - \frac{k^2}{2} = \frac{k^4}{4} - \frac{k^2}{2} = \frac{k^2}{2}(\frac{k^2}{2} - 1)$$

Así que para $k = 4$, $k = 6$ y $k = 10$, la cantidad de `pings` para el control de datos es:

- $k = 4 \Longrightarrow \frac{4^2}{2}(\frac{4^2}{2} - 1) = \frac{16}{2}(\frac{16}{2} - 1) = 8(8-1) = 8 \times 7 = 56$
- $k = 6 \Longrightarrow \frac{6^2}{2}(\frac{6^2}{2} - 1) = \frac{36}{2}(\frac{36}{2} - 1) = 18(18-1) = 18 \times 17 = 306$
- $k = 10 \Longrightarrow \frac{10^2}{2}(\frac{10^2}{2} - 1) = \frac{100}{2}(\frac{100}{2} - 1) = 150(50-1) = 50 \times 49 = 2450$

En la figura 5.2 se puede ver cómo evoluciona esta cantidad conforme se incrementa el k par que define al *Fat Tree*.

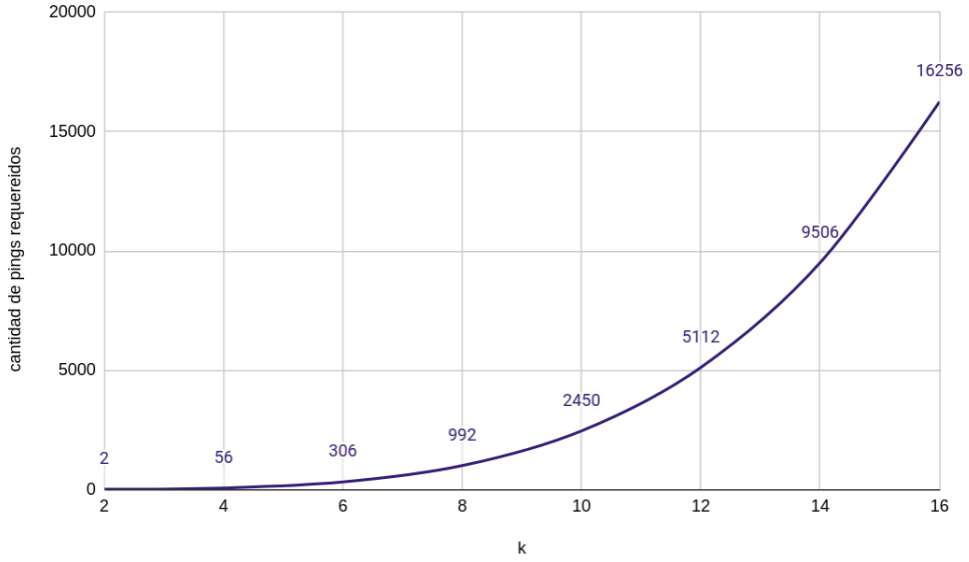

Figura 5.2: Evolución de la cantidad de pings requeridos para chequear una topología de tamano k.

6. Conclusiones y trabajo futuro

En general los objetivos del trabajo fueron cumplidos. Pudimos construir una primera versión del *framework*, resolviendo problemas surgidos como la comunicación entre el controlador y los nodos emulados o la implementación de la ventana deslizante. Incluso reportamos un *bug* en el repositorio oficial de Kathará, el cual fue resuelto por los desarrolladores y será incorporado en la siguiente versión [19].

El enfoque que resultó más rápido fue el de comparación de tablas, debido a la poca cantidad de operaciones y el bajo costo de ellas, en contraste con el *overhead* introducido por las dependencias y procedimientos de la ventana deslizante, como fue descrito en la sección anterior.

Como trabajo futuro vemos la posibilidad de mejorar la implementación de la ventana deslizante, tanto en su versión distribuida como centralizada. Actualmente el nodo escribe sus capturas en archivos `.pcap` (uno por cada interfaz) al mismo tiempo que se realiza el chequeo abriendo esos mismos archivos, lo que termina desembocando en un *overhead* por la concurrencia de lectura/escritura. El *wrapper* PyShark admite la lectura de paquetes en tiempo real desde una interfaz[20]. Utilizar esto en el enfoque distribuido permitiría evitar el *polling* y la escritura/lectura, evitando el *overhead* introducido.

Para el criterio de comparación de tablas centralizado se podría implementar alguna variante con hilos, de modo de paralelizar el *polling* que se le hace a los nodos hasta que presentan una tabla que califique para converger. De esta manera se evitaría recorrer todos los nodos restantes secuencialmente.

Otro punto a trabajar sería una robusta evaluación de eficiencia de cada criterio, más allá del conteo de tiempo que difícilmente conforme una métrica confiable.

[19] https://github.com/KatharaFramework/Kathara/issues/143
[20] https://github.com/KimiNewt/pyshark#reading-from-a-live-interface

Referencias

[1] Edsger W. Dijkstra. «A note on two problems in connexion with graphs». En: *Numerische Mathematik* 1 (1959), págs. 269-271.

[2] K Lougheed e Y Rekhter. *A Border Gateway Protocol (BGP)*. RFC 1105. RFC Editor, jun. de 1989.

[3] M. Aelmans y col. *Day One Routing in Fat Trees*. Juniper Networks Books, 2020. ISBN: 9781736316009.