Universidad de Cantabria

Facultad de Ciencias

ENCAMINAMIENTO NO MÍNIMO EN REDES INDIRECTAS Y SUS APLICACIONES

(Non-minimal routing in indirect networks and its applications)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Cristina Brinza

Director: María del Carmen Martínez Fernández

Co-Director: Cristóbal Camarero Coterillo

Septiembre - 2024

Resumen

Los sistemas High Performance Computing (HPC) están destinados a resolver problemas computacionales complejos de gran tamaño, por lo que su rendimiento tiene un impacto importante en la eficiencia de las aplicaciones paralelas que ejecutan.

Hoy en día contribuyen significativamente en el avance tecnológico y son indispensables en el desarrollo de modelos de lenguaje grandes y del apredizaje automático. Es por ello por lo que cobran importancia las redes de interconexión, que conectan cientos de miles de nodos de cómputo.

Como parte de este Trabajo de Final de Grado se estudian el Fat-Tree, una de las topologías más populares en la supercomputación, y el Orthogonal Fat-Tree (OFT). El OFT se diferencia del primero en que es menos costoso y más escalable, aunque presenta ciertas desventajas con tráfico adverso. Asimismo se estudia la topología Random Folded Clos (RFC) que destaca por su gran escalabilidad sin comprometer el rendimiento.

Bajo patrones de tráfico más estresantes, el rendimiento de las topologías OFT y RFC empeora si se emplea encaminamiento mínimo, contrariamente al Fat-Tree, lo que impulsa el estudio del encaminamiento no mínimo. Además, la expansión de la RFC puede hacer que pierda conectividad, obligando al uso de este encaminamiento.

Con el fin de explorar las ventajas del enrutamiento no mínimo en estas redes, se ha llevado a cabo un estudio experimental con un simulador de redes de interconexión. Por esta razón, fue necesario implementar un algoritmo para dicho enrutamiento en el simulador, cuya utilización ha permitido evaluar el rendimiento de las topologías presentadas.

PALABRAS CLAVE - Encaminamiento no mínimo, Orthogonal Fat-Tree, Random Folded Clos, Redes para Computación de Alto Rendimiento (HPC)

Abstract

High Performance Computing (HPC) systems are designed to solve large-scale, complex computational problems, making their performance crucial for the efficiency of the parallel applications they run.

Today, they contribute significantly to technological advancement and are indispensable in the development of large language models and machine learning. This highlights the importance of interconnection networks, which connect hundreds of thousands of computing nodes.

As part of this undergraduate thesis, we study the Fat-Tree, one of the most popular topologies in supercomputing, and the Orthogonal Fat-Tree (OFT). The OFT differs from the former in being less costly and more scalable, though it presents certain disadvantages under adversarial traffic. Additionally, the Random Folded Clos (RFC) topology, which stands out for its great scalability without compromising performance, is also studied.

Under more stressful traffic patterns, the performance of the OFT and RFC topologies deteriorates when using minimal routing, unlike the Fat-Tree, which drives the study of non-minimal routing. Moreover, the expansion of the RFC can cause it to lose connectivity, necessitating the use of non-minimal routing.

To explore the advantages of non-minimal routing in these networks, an experimental study was conducted using an interconnection network simulator. For this reason, it was necessary to implement a routing algorithm in the simulator, which allowed for the evaluation of the performance of the presented topologies.

KEY WORDS - Non-minimal routing, Orthogonal Fat-Tree, Random Folded Clos, High-Performance Computing (HPC) Networks

Agradecimientos

Desde el comienzo del grado, he lidiado constantemente con dificultades en un campo casi desconocido para mi, pero a la vez tan interesante. Tanto la elección de la carrera como de la mención fueron decisiones difíciles que sabía que cambiarán mi trayectoria. A pesar de la incertidumbre sobre si estaba siguiendo el camino adecuado, pude seguir adelante gracias al apoyo incondicional de mi familia y a la motivación que me han inspirado mis compañeros, muchos de los cuales ya tenían vocación por la informática. También les agradezco a mis amigos más cercanos de la facultad por haberme aliviado estos cuatros años, sobretodo en el aspecto emocional. Agradezco a los profesores, cuyo papel ha sido esencial en mi formación, y en particular a los que más paciencia han tenido conmigo. Por último, quiero agradecer a Carmen y Cristóbal por toda la ayuda proporcionada y por ofrecerme este trabajo tan especial.

Al final, estudiar y practicar en un campo que no dominas tiene su magia y trasmite una gratificación realmente única.

Índice general

1. Introducción

Este capítulo se divide en dos secciones. La primera introduce el contexto y la motivación del trabajo. La segunda establece los objetivos iniciales y su grado de consecución.

1.1. Contexto y motivación

Uno de los pilares fundamentales de la supercomputación es la forma en que se conectan los elementos del supercomputador y se asegura la comunicación efectiva entre sus nodos. A medida que las demandas de carga trabajo en los sistemas HPC aumenta, especialmente con el auge del 5G, la computación en la nube y la inteligencia artificial, la escala de interconexión de la red de los sistemas de computación de alto rendimiento también sigue expandiéndose, afectando a la mejora del rendimiento del sistema.

A lo largo de la historia, un porcentaje importante de redes de interconexión en supercomputación ha adoptado una topología Fat-Tree, Folded Clos o variantes relacionadas [1]. Aunque el Fat-Tree destaca por su elevado coste, es una red que ofrece muy buen rendimiento con encaminamiento sencillo bajo cualquier patrón de tráfico, además de ser fácil de implementar. Entre las alternativas de las topologías indirectas sobresale el Orthogonal Fat-tree (OFT) [2], dada su gran escalabilidad. Sin embargo, cuando se aplican cargas de trabajo con patrón de comunicación no uniforme el rendimiento de los OFT se deteriora, a diferencia de los Fat-Trees, lo que ha promovido el uso de encaminamientos no mínimos [3]. Las redes Random Folded Clos (RFC) [4], por otro lado, se plantean como alternativa que facilita ampliar el tamaño del sistema, manteniendo un compromiso entre el rendimiento y la escalabilidad. En cambio, en ciertas configuraciones de la RFC las restricciones de su conectividad topológica impiden que se realice encaminamiento mínimo, llevando a considerar de nuevo el encaminamiento no mínimo en estas topologías.

Por estos motivos, en este trabajo final de grado se propone considerar la viabilidad del encaminamiento no mínimo y sus posibles ventajas en topologías indirectas como las mencionadas.

Hoy en día las topologías más empleadas en los supercomputadores del Top500 son Fat-Trees, torus y dragonfly. Tras un análisis de los supercomputadores del Top10 de la lista de Junio 2024 [1] se ha creado la Tab. 1.1 en la que se presentan sus topologías.

En el estudio previamente mencionado [3] se han analizado diferentes algoritmos de encaminamiento en topologías alternativas al Fat-Tree cuyo coste es menor y que son la Slim Fly, Multi-Layer Full-Mesh y Orthogonal Fat-Tree. En él se propone enrutamiento no mínimo load oblivious y adaptive deadlock-free en topologías indirectas de la clase Stacked Single-Path Trees. Asimismo se estudia el rendimiento para tráfico uniforme, sintético adverso en el peor de los casos, de todos a todos e intercambios con los vecinos más cercanos, que son los más comunes en aplicaciones HPC.

En este informe se presenta otra alternativa de encaminamiento no mínimo para las topologías Fat-Tree, OFT y RFC.

N°	Supercomputador	Topología
1	Frontier	Dragonfly
2	Aurora	Dragonfly
3	Eagle	Fat-Tree
4	Fugaku	Torus
5	Lumi	Dragonfly
6	Alps	Dragonfly
7	Leonardo	Dragonfly+
8	MareNostrum5	Fat-Tree
9	Summit	Fat-Tree
10	Eos NVIDIA DGX SuperPOD	Fat-Tree

Tabla 1.1: Topologías del Top10

1.2. Objetivos

En este Trabajo de Final de Grado se estudia el encaminamiento no mínimo y sus beneficios en topologías indirectas como las presentadas anteriormente. Los objetivos que se han definido son los siguientes:

1. Estudio del arte de topologías indirectas destacables y los mecanismos de encaminamiento considerados en las mismas.
2. Análisis de los requerimientos para la implementación de mecanismos de encaminamiento no mínimo en las topologías indirectas Fat-Tree, OFT y RFC.
3. Implementación de algún mecanismo de encaminamiento no mínimo en las topologías consideradas en un simulador de redes de interconexión.
4. Evaluación de las propuestas y discusión de su aplicabilidad.

Dichos objetivos se han conseguido de manera satisfactoria en su mayoría. Pues se ha llevado a cabo un análisis de las características de las topologías indirectas más relevantes, lo que constituyó el fundamento de la implementación y evaluación realizadas posteriormente. También se ha logrado identificar los requisitos necesarios para el diseño del routing, así como su propia programación. Por último, se ha realizado una evaluación completa de las propuestas a partir de los resultados obtenidos.

Este trabajo ha permitido profundizar en el conocimiento científico mediante la aportación de datos empíricos, contrastando con el experimento previo realizado por Kathareios et. al [3] explorando esta propuesta. Se organiza de esta forma:

- Capítulo 1: proporciona una visión general del trabajo junto con la motivación.
- Capítulo 2: incluye los conceptos básicos de topologías y redes de interconexión.
- Capítulo 3: trata sobre las estrategias de encaminamiento abordadas.
- Capítulo 4: incluye los experimentos realizados.
- Capítulo 5: presenta las conclusiones finales.

2. Redes de interconexión

Este capítulo incluye el contexto técnico que facilita el entendimiento del trabajo, así como la descripción de las topologías de red y los mecanismos de encaminamiento con los que se ha trabajado.

2.1. Contexto técnico

Una topología de red hace referencia a la disposición física de los dispositivos y el cableado en una red de comunicación. El término "nodo" sirve para describir cualquier dispositivo que puede procesar, recibir o transmitir datos en una red. Los equipos finales o servidores pueden comunicarse a través de routers y enlaces que determinan la mejor ruta para el tráfico de datos y proporcionan canales por los que los datos viajan respectivamente.

Un buen diseño de la topología busca maximizar el rendimiento y minimizar el coste de la red. Por un lado, el rendimiento está asociado al throughput, que es la cantidad de datos real que viaja con éxito por un canal de comunicación en un período determinado de tiempo, y a la latencia o retraso en la comunicación de datos de su origen al destino. El coste, por otro lado, mide el número y la complejidad de los chips, así como la densidad y longitud de sus interconexiones.

El mayor número de saltos que un paquete de datos puede dar entre cualquier par de nodos se denomina diámetro de red e influye directamente en el coste y rendimiento. Cuanto menor es el diámetro más se reduce la latencia, lo que beneficia el rendimiento.

Otros conceptos relevantes son el grado y la bisección. El primero, también llamado radix del router, representa el número de puertos o conexiones que este tiene disponibles. Emplear routers con el mismo radix es conveniente porque se pueden agregar routers a la red más fácilmente, haciendo la red más escalable. La bisección de una red se refiere a la cantidad de enlaces que se debe cortar para dividir el conjunto de sus nodos en dos mitades. El número mínimo de enlaces que se puede cortar para conseguir dicha división representa el ancho de bisección y es deseable que sea cuanto mayor para construir una red más estable. En una red con el ancho de banda de la bisección total o full bisection bandwidth en inglés indica que cualquier mitad de los nodos puede comunicarse simultáneamente con la otra mitad.

En una red de interconexión un nodo puede servir como router y servidor a la vez, y es lo que ocurre en el caso de las topologías directas. En las topologías indirectas un nodo no puede ser ambas cosas, por lo que los paquetes viajan entre nodos finales por medio de routers dedicados. La diferencia se puede apreciar en la Fig. 2.1. Un ejemplo típico de topología directa es el torus (Fig. 2.2), usado en supercomputadores como IBM Blue Gene.

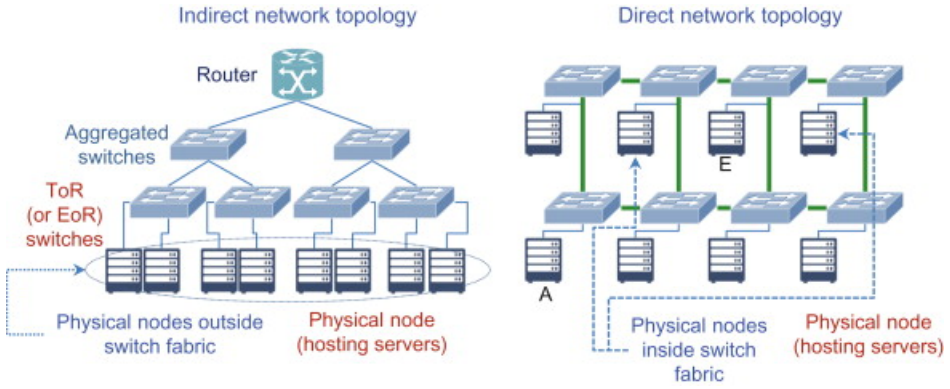

Figura 2.1: Comparativa de red indirecta y directa [5]

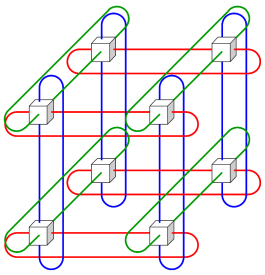

Figura 2.2: Torus 3D de IBM Blue Gene [6] como ejemplo de red directa

2.2. Topologías de red

Las topologías abordadas principalmente en este informe son el Fat-Tree, el Orthogonal Fat-Tree y la Random Folded Clos y pertenecen a la categoría de indirectas.

2.2.1. Fat-Tree

Fat-Tree (FT) se refiere a una topología en forma de árbol con varios niveles, donde las raíces representan los routers del nivel superior y las hojas los conectados a los servidores en una red. Por ejemplo, la red mostrada en Fig 2.3 consta de tres capas de routers: núcleo, agregación y borde, que se conecta directamente a los hosts. Su estructura se apoya en los principios de la red Clos, propuesta por Charles Clos en 1952 [8], que es un tipo de red de conmutación multinivel, no bloqueante y que representa la base de las redes de centros de datos a gran escala actuales, dada su capacidad de optimizar las operaciones de conmutación en las comunicaciones de datos Ethernet. Pues el FT es concretamente una red Folded Clos plegada, que reduce las conexiones mediante el plegado de las capas. Esto permite el escalado eficiente a medida que se agregan más dispositivos.

Se trata de una topologia ampliamente usada en entornos de computación paralela gracias a que es rearrangeable (reorganizable) y non-blocking (no bloqueante).

Una red no bloqueante es aquella en la que puede establecerse cualquier comunicación entre un par de nodos sin interferir con el resto de conexiones existentes y es reorganizable si siempre se puede constituir una conexión reajustando las rutas de otras conexiones, permitiendo que, si una ruta no está disponible, se pueda encontrar otra alternativa. Pues

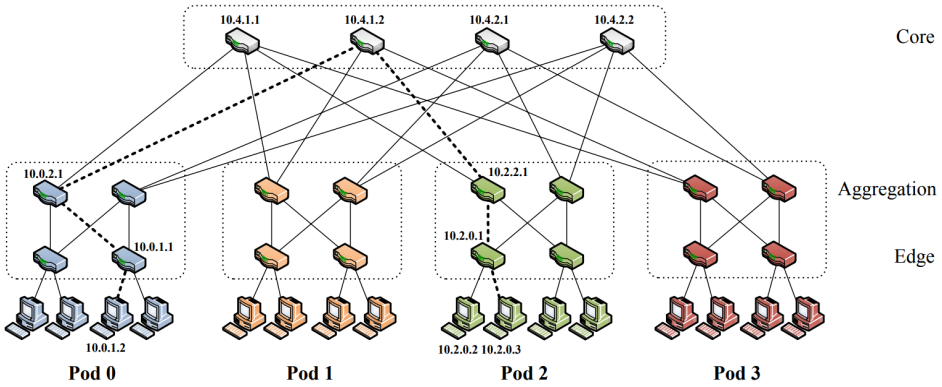

Figura 2.3: Fat-Tree como ejemplo de red indirecta [7]

una red reorganizable puede realizar todas las bisecciones posibles entre sus entradas y salidas. Si es estrictamente no bloqueante implica que es también reorganizable [9].

Estas características se originan en el diseño de Clos, en el que está basado el FT. El propósito de dicho diseño era minimizar la cantidad de puntos de cruce para admitir la conmutación de llamadas telefónicas, ofreciendo conectividad sin bloqueos. Esto se conseguía organizando una estructura en tres etapas [8].

El FT tiene varios niveles donde el número de enlaces que conectan el nivel intermedio con el superior, es decir, los que suben, es el mismo que el de los enlaces que bajan hacia el nivel inferior, lo que significa que por cada enlace que sube hay un enlace que baja. Esta propiedad permite que, en el caso de que todas las parejas de nodos empiecen a comunicarse, cada par pueda usar el enlace al máximo ancho de banda, para cualquier patrón de tráfico. Por otro lado, el proceso de reorganización (rearrangeability) modifica el estado de la red para desbloquear una comunicación bloqueada.

Algunas de las ventajas que ofrece esta topología es la homogeneidad de los routers, puesto que permite que todos tengan el mismo número de puertos, por lo que es fácil reemplazarlos, y también el balanceo de carga, porque existen varios caminos entre diferentes hosts.

En este trabajo se aborda concretamente el FT de dos niveles. La restricción para que este tenga full bisection es que el número de conexiones que tiene un router de primer nivel con el segundo nivel debe ser el mismo que el número de hosts conectados a dicho router. Dada la gran diversidad de caminos y la propiedad no bloqueante, la red es capaz de trasmitir datos a máximo ancho de banda para cualquier permutación. Además su coste es bajo y el diámetro es pequeño (dos saltos).

Sin embargo, expandir una red FT habitualmente requiere añadir muchos nodos o hasta más niveles, incrementado los recursos necesarios de forma “agresiva” e influyendo en el coste. Como esta topología ofrece baja escalabilidad bajo un determinado grado del router, es de interés trabajar con una versión que proporcione menor diversidad de caminos mientras preserva el diámetro pequeño y el coste bajo. Emplear una topología de diámetro dos como el Orthogonal Fat-Tree resulta más rentable, ya que tiene menor coste que el FT si utiliza los mismos recursos, y mejor escalabilidad.

2.2.2. Orthogonal Fat-Tree

Una de las topologías indirectas que se abordan en el estudio de Kathareios et al. [3] es el Orthogonal Fat-Tree (OFT) [2] de dos niveles, donde se define como parte de la clase Stacked Single Path Trees (SSPT). La estructura de los SSPT implica dos niveles de routers interconectados de manera que existe exactamente una ruta mínima entre cualquier par de routers de nivel uno, y se utiliza un número mínimo de routers de nivel dos. La escalabilidad de los SSPT se expresa en términos de grado de router y el número de nodos conectados a cada router. Apilando dos Single Path Trees (SPTs) se consigue un Two-Level k-OFT donde el grado de los routers de nivel uno y dos coincide con k. En la Fig. 2.4a se muestra un 3-OFT, pues separando los SPTs se puede ver que los routers del nivel superior tienen tres conexiones hacia abajo, al igual que los del nivel inferior hacia arriba (Fig .2.4b). Los niveles inferiores de los SPTs se denominan $L0$ y $L2$, mientras que $L1$ es el nivel superior común.

Aunque el patrón de construcción de un OFT, llamado Maximal Leaves Basic Building Block of degree k (k-ML3B), reduce la diversidad de rutas en comparación con un FT, está diseñado para proporcionar más escalabilidad. La Tab. 2.1 incluye las características de unos ejempos de estas topologías para destacar la diferencia en escalabilidad entre ellas. Pues se puede obtener un OFT de diámetro dos con un número de servidores cercano a un FT de tres niveles y con el mismo coste que el de un FT de diámetro dos.

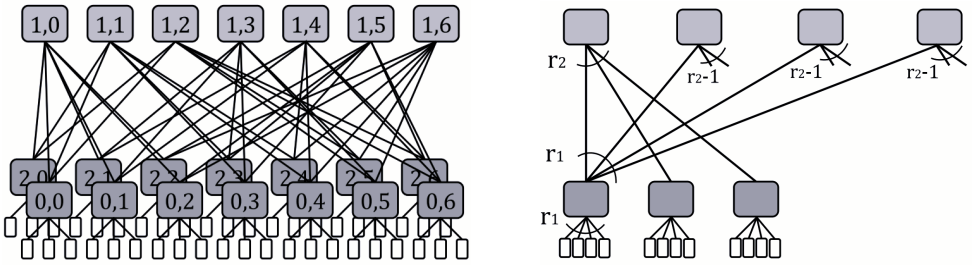

Figura 2.4: Topologías 3-OFT y SPT [3]

Topología	Radix	Niveles	Routers	Servidores
FT	12	3	180	432
OFT	12	2	93	372
FT	36	3	1620	11664
OFT	36	2	921	11052

Tabla 2.1: Ejemplos de FT y OFT con diferente número de nodos

2.2.3. Random Folded Clos

Finalmente, la tercera topología destacable en este trabajo es la Random Folded Clos (RFC), que fue propuesta por Cristóbal Camarero, Carmen Martínez, presentes directores,

y Ramón Beivide en [4]. En dicho artículo una RFC con parámetros P se define como una topología elegida aleatoriamente con probabilidad uniforme de todas las redes Folded Clos posibles con parámetros P, que se definen en la Tab. 2.2. Pues cuenta con una cierta estructura y un nivel de aleatorialidad a la vez (Fig. 2.5), que le permite expandirse de forma elegante (gracefull expasion), ya que se pueden añadir routers sin incrementar los niveles de la red. Se trata de una topología que constituye un balance entre coste, escalabilidad, tolerancia a fallos y rendimiento.

Parámetros	Definición
T	Número de nodos de cómputo o terminales.
R	Radix de los switches (número de puertos).
l	Número de niveles.
N_i	Número de switches en el nivel i.
k_i	Aridad del nivel i (como un árbol).

Tabla 2.2: Parámetros Folded Clos [4]

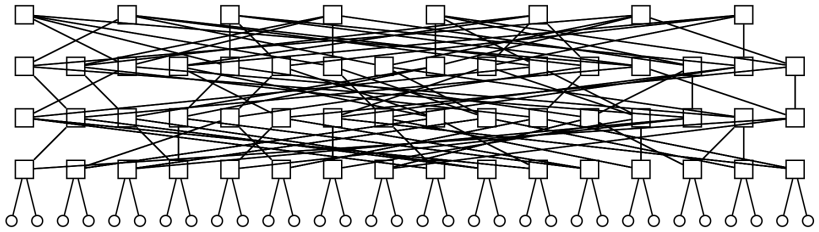

Figura 2.5: Topología RFC de 4 niveles [4]

2.3. Mecanismos de encaminamiento en redes indirectas

En el FT, al ser una topología organizada en varios niveles de routers (mutistage), se suele emplear el encaminamiento denominado Up/Down. Este routing resulta eficiente al evitar la creación de bucles cíclicos, porque asigna las direcciones up y down a los enlaces de la red. Esto es posible gracias a que los routers no permiten que un enlace se utilice en la dirección ascendente tras usar uno en la dirección descendente, preveniendo de esta forma los deadlocks. El término deadlock o bloqueo mutuo, como se explicará en detalle en el capítulo siguiente, hace referencia a la situación en la que elementos del sistema se quedan bloqueados cada uno a la espera de que un recurso sea liberado por otro antes de poder continuar, formando un ciclo de dependencias circulares en el que no pueden avanzar.

Según el Teorema 1 de Duato [10], una función de enrutamiento conectada y adaptativa R para una red de interconexión I no tiene deadlocks si no hay ciclos en su grafo de dependencia de canales D, cuyos vértices son los canales de la red de interconexión I y las aristan son los pares de canales (c_i, c_j) tales que existe una dependencia directa de c_i a c_j.

Por lo tanto, en topologías como el FT, empleando Up/Down se garantiza que los paquetes siempre sigan un camino consistente por una subida del nivel inferior al superior, seguida por una bajada al nivel inferior hacia el nodo destino.

El OFT de dos niveles, a su vez, es libre de deadlock si se emplea enrutamiento mínimo, dado que los enlaces unidireccionales se clasifican o como "hacia" el nivel superior o como "desde" dicho nivel, equivalente a up y down respectivamente, y también porque se impone un orden ascendente en la clase de enlace a ocupar. Esto se puede lograr numerando los enlaces de manera que cada ruta válida progrese en un orden ascendente (o descendente) sin retroceder, lo que garantiza que ningún patrón de tráfico pueda llevar a un bloqueo mutuo.

En este contexto se ha de explicar brevemente la arquitectura del router para facilitar la compresión de conceptos que se mencionan a continuación, como los canales virtuales. En un router se pueden identificar puertos de entrada y salida asociados a sus unidades correspondientes. Estas unidades de entrada y salida guardan en buffers la información manejada en phits y están conectadas por el switch, como se puede ver en la Fig. 2.6. Llegados los paquetes, la lógica de enrutamiento determina por qué puerto tienen que salir para reenviarse y el virtual channel allocator asigna los canales a los paquetes. Los canales virtuales [11] permiten separar el flujo de paquetes de un canal físico en varios flujos creando varias colas. Después el switch allocator asigna a cada phit del paquete un intervalo de tiempo específico en el switch durante el cual se reenvía a la unidad de salida correspondiente.

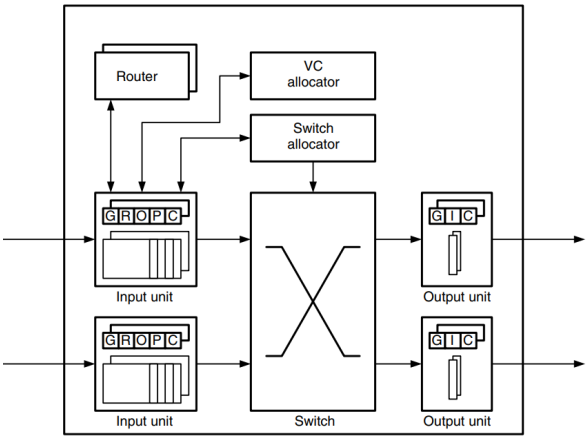

Figura 2.6: Diagrama de bloques de un router con canales virtuales [12]

Emplear routing no mínimo, en cambio, sí que supone el riesgo de formar bucles cíclicos, puesto que, al dar más saltos (más de un up/down) o utilizar más enlaces en las rutas no mínimas, se crean dependencias entre los paquetes. En este caso conviene utilizar canales virtuales, que separan las parejas de subida y bajada evitando la formación de ciclos. Si se tuviese dos canales virtuales el primero se asignaría a la primera ruta up/-down hasta el nodo intermedio y el segundo a la segunda ruta desde el intermedio hacia el destino.

Aunque en el pasado se han estudiado varias soluciones al problema del bloqueo mutuo

en las redes de interconexión [13], el uso de los canales virtuales es una de las más comunes en las topologías indirectas.

3. Encaminamiento no mínimo en redes indirectas

Este capítulo tratará de los modelos de tráfico y de las estrategias y principales retos relativos al routing en las redes indirectas.

3.1. Modelos de tráfico

La forma en que los datos o los paquetes se transmiten entre los nodos de la red es determinada por el patrón de tráfico. De los más comunes que se utilizan son el uniforme y la permutación aleatoria. En el tráfico uniforme cada nodo transmite paquetes a otros nodos con igual probabilidad, puesto que trata de elegir uniformemente un nodo destino aleatorio. De esta manera la carga se distribuye equitativamente simulando una situación sin cuellos de botella. La permutación aleatoria, por otro lado, consiste en seleccionar un único destino, que debe ser destino una vez, por lo que este recibe datos de un único origen. Al no seguir un patrón predecible es más conveniente para identificar puntos de congestión en la red.

3.2. Encaminamiento no mínimo en redes directas para mejorar el rendimiento

En determinadas situaciones puede resultar rentable emplear encaminamiento no mínimo en redes directas, aunque no lo requieran. Esto sirve para evitar cuellos de botella que puedan surgir cuando el encaminamiento mínimo es ineficiente y mejorar el rendimiento. Para ello se puede utilizar el routing de Valiant [14], cuyo algoritmo se basa en seleccionar un nodo intermedio al azar al que dirigir el tráfico primero, antes de que este lo reenvíe hacia el destino. Así se evita que el tráfico se concentre en determinados puntos de la red, aunque puede implicar mayor latencia y complejidad de la implementación. Además, en los experimentos en el capitulo siguiente, se verá que con tráfico uniforme Valiant rinde peor que el encaminamiento mínimo (UpDown).

Este enrutamiento no tiene en cuenta el tráfico de la red o su estado global, pues la ruta de cada paquete se determina en función de su propia información, independientemente del estado del resto de los paquetes de la red, de ahí que se denomine “oblivious”, tal como se menciona en [14].

En un estudio sobre la topología Slim Fly (SF) [15] se han evaluado varios routings entre los cuales se encuentra Valiant. SF es una topología directa diseñada para reducir el diámetro de la red, los costes y la latencia manteniendo un alto rendimiento. En dicho estudio se ha visto que Valiant es capaz de dispersar el tráfico por diferentes rutas y de manejar hasta el 40% de la carga total.

3.3. Deadlock en routing no mínimo

Como ya se ha explicado, el deadlock ocurre cuando un conjunto de paquetes no puede avanzar en la red debido a que están esperándose mutuamente para acceder a los recursos que han tomado, creando un bucle. La ventaja de las topologías indirectas es que generalmente no sufren deadlock debido a que, como los routers intermedios no son fuente de tráfico, los flujos de paquetes no se interrumpen en mitad del camino. En cambio, en una red directa todos los nodos generan paquetes, permitiendo que se de el deadlock en rutas de más de un salto, aunque sean mínimas.

En el caso de las redes multinivel, como el OFT de dos niveles, al realizar encaminamiento no mínimo, se pasa como poco dos veces por routers hoja, incluido el origen, que son fuentes de paquetes, permitiendo así que se creen ciclos.

Al realizar encaminamiento no mínimo surge mayor variedad de rutas entre el par origen y destino, lo cual sirve para balancear la carga de la red entre los routers y mantenerla constante lo máximo posible. El problema que tiene este routing en topologías multinivel es que puede producir ciclos si los canales o enlaces no se utilizan adecuadamente. Se ha elaborado la Fig. 3.1 para representar una simplicación de una situación clásica de bloqueo mutuo en una red Clos o similar. En ella los paquetes tienen las siguientes rutas:

$$P_0\colon A \to B \to C$$
$$P_1\colon B \to C \to A$$
$$P_2\colon C \to A \to B$$

Estos paquetes comparten enlaces en su camino por lo que se deben esperar entre sí hasta liberarse. Luego P_0 no podrá entrar en el router B hasta que P_1 no entre en C y P_2 en A, quedándose bloqueados esperando un enlace que está ocupado por otro paquete.

Sin embargo, el deadlock se puede evitar empleando canales virtuales, a los que se asigna cada flujo de paquetes de forma ordenada. Pues en el routing de Valiant se ocuparía el canal virtual 0 para el misrouting y el canal 1 para la ruta mínima entre el router intermedio y el destino. Al igual que en el ejemplo de la figura, en el primer up/down del paquete P_0 se ocuparía el primer canal virtual y en el segundo el siguiente en orden.

3.4. Routing no mínimo para la mejora del rendimiento

La primera aplicación del routing no mínimo se presenta con el objetivo de mejorar el rendimiento en tráficos adversos, como la permutación aleatoria, para el OFT. En el estudio de Kathareios [3] se ha comprobado que el rendimiento se degrada cuando se aplica routing mínimo para tráficos de peor caso. Este patrón de tráfico se da cuando todos los servidores de un router origen R_s del nivel $L0$ o $L2$ se comunican exclusivamente con todos los servidores de un router destino R_d del mismo nivel, el cual no es el equivalente simétrico de R_s. Como el OFT no tiene una gran variedad de caminos mínimos, las rutas se sobrecargan excesivamente, resultanto en un solo 8.3% de capacidad de carga aceptada antes de que se saturen. También se observa que utilizando routing no mínimo, llamado Indirect Random, el rendimiento mejora hasta conseguir un 48% de throughput.

De ahí surge la propuesta de implementar un encaminamiento no mínimo, que se ha nombrado UpDownDeroutingAlways, para mejorar el desempeño del OFT para un patrón

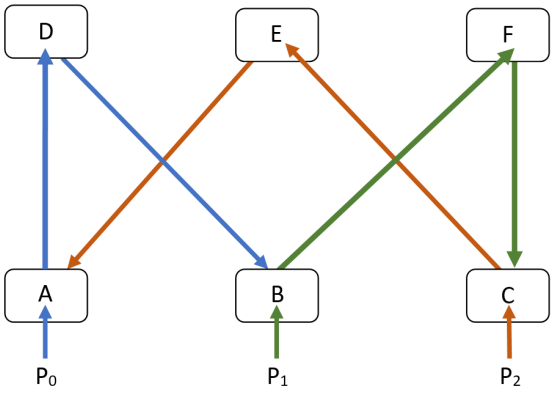

Figura 3.1: Ejemplo de deadlock, donde P_0, que debe pasar por la ruta azul para llegar a su destino C, se queda bloqueado en la entrada de B. Asimismo P_1 no puede seguir por el camino verde que tiene como destino A, debido a que se queda bloqueado en C, ya ocupado por P_2. Este último trata de seguir la ruta naranja para llegar a B pasando por A, que a su vez está ocupado, resultando en un bloqueo mutuo.

de tráfico de permutación aleatoria. Concretamente se proponen dos versiones: una que consiste en realizar misrouting, independientemente de si existen rutas up/down entre el origen y el destino, que es la mencionada anteriormente, y otra destinada a la segunda aplicación, descrita a continuación.

3.5. Routing no mínimo para garantizar la existencia de rutas

Otra aplicación de interés es en las redes up/down desconectadas, concretamente en la RFC. Que una red sea up/down conectada en este contexto significa que todos los pares de servidores están conectados por rutas mínimas de dos saltos. En la expasión de las RFCs pueden aparecer parejas de servidores sin conexión up/down, si el grado de los routers exceden un determinado umbral que se presenta en el Teorema 4.2 de [4]. Es por ello por lo que es necesario emplear un enrutamiento que permita dar más saltos haciendo las rutas más largas. En este caso se opta por una versión del routing conservadora, donde siempre que se detecta un camino mínimo lo hace y que en caso contrario da más saltos. Ambas versiones se explican con más detalle en el capítulo siguiente.

4. Experimentación

En este capítulo se describe el entorno de trabajo, concretamente las herramientas utilizadas para llevar a cabo los experimentos, junto con la metodología de trabajo. También se explican los modelos de tráfico, las topologías y métricas evaluadas, y el misrouting propuesto en las dos aplicaciones.

4.1. Entorno de simulación y metodología

La herramienta empleada para los experimentos y análisis de rendimiento se denomina CAMINOS (Cantabrian Adaptable and Modular Interconnection Open Simulator) [16], que es un simulador de interconexión de red escrito en Rust [17]. Fue desarrollado por el miembro del grupo de investigación Arquitectura y Tecnología de Computadores (ATC) de la Universidad de Cantabria (UC) Cristóbal Camarero, quien ha actuado como co-director del presente trabajo.

De las características más destacables de este programa es su arquitectura modular que permite añadir funcionalidades fácilmente. Cuenta también con herramientas para gestionar experimentos en hosts remotos, encolar simulaciones en sistemas Slurm y generar salidas en formato tex/pdf. Además, el lenguaje en que está programado, Rust, garantiza la seguridad de la memoria y concurrencia, previene los errores y se compila a código de bajo nivel optimizado. También la programación en Rust resulta más ágil que en C++, cuya sintaxis es similar.

Debido a la carga excesiva de trabajo, las simulaciones se han realizado en el entorno Triton, que es un clúster de cómputo del grupo ATC de la Universidad de Cantabria. Para agilizar su ejecución los experimentos se han encolado en el sistema Slurm, de forma que se puedan llevar a cabo varias tareas paralelamente.

En cuanto a la metodología, para efectuar el estudio primero se han configurado los experimentos a ser ejecutados del fichero principal `main.cfg`, incluyendo los parámetros de la topología, el tráfico, el router, el enrutamiento y los enlaces. Por ejemplo, el código 4.1 muestra la configuración del tráfico empleado en una de las pruebas.

Las pruebas pequeñas iniciales que ayudaron a familiarizarse con el simulador se han realizado en el ordenador propio, para que posteriormente los experimentos pesados sean

```
traffic: HomogeneousTraffic {
    pattern: ![
        Uniform { legend_name:"uniform" },
        RandomPermutation { legend_name:"random server permutation" }],
    servers: 432,
    load: ![0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    message_size: 16 }
```

Código 4.1: Ejemplo de configuración de tráfico en CAMINOS

lanzados a Tritón en el sistema Slurm. Fue necesario por lo tanto la instalación en el sistema local del paquete `caminos` desde el repositorio oficial de paquetes para Rust (crates.io).

Para realizar un experimento primero se debe crear una carpeta con la configuración de CAMINOS por defecto. El comando que lo posibilita es `caminos <experiment-index> --action=shell`, donde el argumento `<experiment-index>` indica el nombre de la carpeta e `index` es el número del experimento. Dentro de dicha carpeta se puede ejecutar el experimento en local utilizando el flag `--action=local`. Una vez se ha entendido mejor el funcionamiento del simulador se ha pasado a ejecutar las simulaciones en el clúster. Para ello se debe utilizar la acción `--action=slurm` que permite lanzarlas a Slurm.

Como se ha mencionado, el simulador genera las salidas de forma interna, lo que simplifica bastante el flujo de trabajo. Pues para obtener las salidas de los resultados procesados se ha usado la acción `--action=output`.

Al comienzo de estas pruebas se encontraron dificultades con ciertos paquetes de LaTeX que son requeridos para mostrar los resultados con el flag `--action=output`. Pues fue necesario instalar el paquete `texlive-pictures` que contiene recursos que facilitan la creación de documentos LaTeX.

4.2. Tráfico, métricas y topologías evaluadas

Se ha trabajado con dos patrones de tráfico diferentes en las simulaciones. Estos son los mismos que los detallados en el capítulo anterior: uniform y random permutation (en CAMINOS llamados `Uniform` y `RandomPermutation`). El uniforme es un patrón en el que cada nodo tiene la misma probabilidad de enviar paquetes a cualquier otro nodo, por lo que ofrece una visión más general y simple del rendimiento en la red. A diferencia de la permutación aleatoria, el tráfico uniforme refleja un escenario donde las condiciones son ideales. Al distribuir la carga de forma homogénea se puede valorar la capacidad máxima de la red. La permutación, por otro lado, puede generar situaciones donde la carga se centra en determinadas zonas o conjuntos de nodos, estresando más la red y permitiendo que se formen cuellos de botella.

Estas topologías se han analizado en base a tres métricas relevantes: el throughtput, la latencia y la longitud de ruta. El throughput o carga aceptada es la cantidad de datos que puede ser transmitida desde un punto de la red a otro en un determinado tiempo y se mide en phits (bits transferidos en paralelo en un ciclo [12]) por segundo. Esta cantidad, representada en las simulaciones en porcentaje, se compara con la carga inyectada, por lo que se ha valorado la carga que la red puede soportar en relación con la que se le ofrece. La latencia indica el tiempo que tardan los datos en atravesar la red y su medida son los ciclos. Finalmente, se ha considerado el número de saltos que los paquetes dan en la red, que representa la path length del camino en función de cada enrutamiento que se emplea en las simulaciones. Las dos últimas métricas están relacionadas en el sentido de que una longitud de ruta mayor sacrifica la latencia de los mensajes y es por ello por lo que un algoritmo de enrutamiento ideal obtendría el menor número de pasos posible. Asimismo si interesara balancear la carga por las conexiones de la red, por ejemplo, en el caso de trabajar con una permutación, la longitud de ruta aumentaría para todos los mensajes.

Las topologías a evaluar son las explicadas en el segundo capítulo, FT, OFT y RFC,

```
topology: MultiStage{
    stages:[
        Fat { bottom_factor:6, top_factor:6 },
        Fat { bottom_factor:12, top_factor:6 }],
    servers_per_leaf: 6,
    legend_name: "a fat-tree defined using stages" }
```

Código 4.2: Ejemplo de configuración del FT

```
topology: OFT {
    height: 1, // number of levels = 2
    prime: 5,
    servers_per_leaf: 6,
    legend_name: "OFT over the projective plane of 5 points" }
```

Código 4.3: Ejemplo de configuración del OFT

con las configuraciones indicadas en las tablas 4.1 y 4.2. Las simulaciones iniciales se han realizado con redes pequeñas (de pocos servidores) de las primeras dos topologías y después se ha introducido la RFC, que destaca por su escalabilidad y aleatoriedad de su diseño.

A continuación se describen las características y configuración de cada topología junto con el enrutamiento usado.

Fat-Tree

El FT, al ser una topología estructurada en múltiples niveles, se encuentra implementada en el módulo `caminos_lib::topology::multistage`, al igual que OFT y RFC. Para configurarla se han de especificar los parámetros de sus niveles y el número de servidores por router hoja. El código 4.2 incluye un ejemplo en el que el grado de los routers es 12 y se indican el número de conexiones que tiene un router de un nivel con uno del otro asignándolo a `bottom_factor` y `top_factor`.

OFT

En el caso del OFT un termino significativo es la altura, que es el número de etapas o conexiones de un nivel al siguiente. Equivale al número de niveles - 1, por lo que los niveles están en el rango [0, height]. En la construcción de esta topología, que se realiza con planos proyectivos y se explica con más detalle en [18], es necesario un número primo `prime` para definir el conexionado de la red. Su valor se emplea en una función $q^2 + q + 1$, donde q es la potencia del número primo, que obtiene el número de routers de cada nivel de los SPTs apilados. En el código 4.3 se puede ver cómo están configurados estos parámetros.

Random Folded Clos

La configuración de la RFC que se ha empleado inicialmente se muestra en el código 4.4. En este caso también se debe especificar el número de enlaces hacia el nivel inferior (`down`) y superior (`up`), indicando en el ejemplo que los routers tienen grado 12. Como en

```
topology: RFC{
    height: 1,
    down: [12],
    up: [6],
    sizes: [22,11],
    servers_per_leaf: 6,
    legend_name: "RFC of radix 12 with 22 leaf routers" }
```

Código 4.4: Ejemplo de configuración de RFC

```
Valiant {
    first: ChannelMap{routing:UpDown,map:[ [0] ]},
    second: ChannelMap{routing:UpDown,map:[ [1] ]},
    legend_name: "Valiant over UpDown",
    selection_exclude_indirect_routers: true }
```

Código 4.5: Configuración de Valiant

el inferior no hay enlaces hacia abajo y en el superior hacia arriba, los arrays son de un elemento. Asimismo se ha de añadir el número del routers de cada nivel comenzando por el inferior (`sizes`) y el de los servidores por router hoja (`servers_per_leaf`).

Las primeras pruebas se han realizado con dos routings, `UpDown` y `Valiant`, dado que la RFC pequeña está conectada, pues todas sus parejas de routers tienen conexiones up/down. En el FT y OFT esto siempre ocurre.

En el simulador el enrutamiento `UpDown` está implementado para determinar la distancia ascendente y descendente entre el nodo origen y el destino. En el caso de que esta sea cero el servidor destino es el que está conectado directamente al nodo actual. Sino, determina qué puertos del nodo actual pueden llevar al nodo destino de manera más eficiente. De esta forma se generan las rutas de salida correspondientes.

Valiant, por otro lado, combina `UpDown` en dos etapas, como se puede ver en el código 4.5. En la primera emplea dicho routing para alcanzar un nodo intermedio aleatorio y en la segunda etapa lo utiliza desde ese nodo hasta el destino final. En la primera fase mapea todos los canales virtuales a 0, conservando el routing original, y en la segunda los mapea a 1.

Además, ha sido necesario establecer `selection_exclude_indirect_routers` a `true`, cuyo valor por defecto es `false`. Este parámetro permite que se elijan como routers intermedios a los que son hoja, es decir, están en el nivel inferior. En caso contrario, este enrutamiento resulta incompatible con la topología, ya que puede seleccionar un intermedio con el que no exista una ruta de dos saltos (up/down) desde el origen, como un router raíz que puede estar a una distancia de tres saltos, dando el mensaje de panic:
`thread 'main' panicked at 'The topology does not provide an up/down path from 3 to 67'...`
Pues poniendo este parámetro a `true` se restringe la elección a routers con los que sí que están conectados por caminos up/down.

```
UpDownDerouting{
    allowed_updowns: 2,
    stages: 1 }
```

Código 4.6: Configuración de `UpDownDerouting`

4.3. Misrouting

Al aumentar el tamaño de la RFC aparecen parejas de nodos que no están conectados por rutas up/down. Estas parejas se pueden conectar incrementando el número de saltos del algoritmo y es donde se ha implementado y comprobado la funcionalidad del misrouting denominado `UpDownDerouting`. Su algoritmo podría tener dos enfoques: uno "ciego", en el que se haría up/down-up/down siempre, y otro "conservador", donde se detectaría si la conexión entre el origen y el destino es de un up/down, en cuyo caso se aplicaría encaminamiento mínimo. Más adelante se hace referencia a estos algoritmos mediante `UpDownDeroutingAlways` y `UpDownDeroutingLazy`, respectivamente.

El nuevo routing tiene la configuración representada en el código 4.6. El parámetro `allowed_updowns` define el número de saltos no mínimos o de desvíos permitidos durante la ruta. En este ejemplo, si no existe una ruta up/down entre dos nodos tiene que ser posible que se realicen cuatro saltos, es decir, dos up/down. `stages`, similar a `height` indica el número de etapas en el enrutamiento multietapa, que por defecto es 1 (número de etapas - 1).

Para la incorporación de esta implementación se ha utilizado el CAMINOS del crates que está en un repositorio git bare en Tritón y se ha partido de la última versión oficial `caminos 0.6.3`. Como el simulador está programado en Rust, se debe utilizar Cargo como herramienta de gestión de paquetes y compilación para dicho lenguaje. Como parte de un proyecto Cargo existe el archivo `Cargo.toml` que define las configuraciones del proyecto y en el que se ha cambiado la dependencia para usar el simulador local y no el que está en crates.io (el público).

El diseño de `UpDownDeroutingAlways` y `UpDownDeroutingLazy` en este trabajo es específico para rutas de máximo cuatro saltos, dado que las redes consideradas tienen dos niveles y son suficientemente escalables.

La implementación del misrouting se ha complementado con el uso de la estructura `RoutingTable`. Esta struct auxiliar se ha implementado a su vez para almacenar información de las rutas de la topología y, por lo tanto, seleccionar los routers candidatos en cada salto. Cuenta con un método `build_distance_2_paths` 4.1 que construye una tabla de rutas de distancia de dos saltos (router-router) desde cada router de la red. Esta función se invoca, por lo tanto, en la inicialización del routing, cuando ya se ha obtenido la topología. Pues se contruye una tabla al comienzo de cada experimento, por lo que con el crecimiento de la red, se volvería a reconocer la red y se actualizaría la tabla.

En la primera aplicación del routing no mínimo se ha planteado utilizar `UpDownDeroutingAlways`, en el que siempre se dan dos up/down, ya que se ha enfocado en evaluar el rendimiento del OFT, que es siempre up/down conectado. Se han considerado cuatro situaciones dependiendo de la distancia entre el router actual y el destino:

```
1: function build_distance_2_paths(topology)
2:     for all source_router in Routers do
3:         for all intermediate_port in Ports(source_router) do
4:             intermediate_router ← GetNeighborRouter(source_router, intermediate_port)
5:             for all destination_port in Ports(intermediate_router) do
6:                 destination_router ← GetNeighborRouter(intermediate_router, destination_port)
7:                 if destination_router is not source_router then                ▷ Evitar bucles
8:                     if PathDoesNotExist(source_router, destination_router) then
9:                         InitializePath(source_router, destination_router) with empty list
10:                    AppendPortToPath(source_router, destination_router, intermediate_port)
```

Algoritmo 4.1: Pseudo-código de `build_distance_2_paths`

1. Si la distancia es 0: el router actual es el router destino.
2. Si la distancia es 1: el actual se encuentra a un salto del destino, entonces se añaden como candidatos los puertos a traves de los cuales se llegaría al router final.
3. Si la distancia es 2 y ya se ha realizado un up/down: el router actual es un router hoja y le queda otro up/down por hacer.
4. Si la distancia es 2 y todavía no se ha dado ningún salto: para llegar al router final se debe realizar dos up/down o cuatro saltos, que es el máximo permitido.
5. Si la distancia es 3: el router actual se encuentra en el nivel superior.

Excepto para el primer caso, se han creado métodos auxiliares, a fin de modularizar el código y facilitar su comprensión, que permiten elegir de forma aleatoria el siguiente router de las rutas existentes entre un par origen-destino dado, si es que hay más de una.

El algoritmo 4.2 contiene el pseudo-código de la función `next` de este routing, donde se gestionan los cuatro casos planteados. Por simplicidad, los métodos auxiliares mencionados y que son parte de `RoutingTable`, se han denominado `next_router_1`, `next_router_2` y `next_router_3`, para los casos 2, 3 y 4, respectivamente.

Para la segunda aplicación se ha implementado un algortimo adaptativo `UpDownDeroutingLazy`, que a diferencia del anterior no tiene en cuenta los saltos dados. Como se puede ver en el algoritmo 4.3 el método `next` sigue gestionando los cuatro casos, teniendo en cuenta que pueden existir rutas no mínimas, donde el origen y el destino no están conectados por un up/down, requiriendo realizar dos. En caso de que lo haya, coge la ruta mínima.

En cuanto a las funciones auxiliares, estas trabajan con la tabla de rutas llamada `paths`, que es un campo de `RoutingTable`. Esta tabla es un `HashMap` cuya llave es la tupla de dos valores (`source_router`, `destination_router`) que representan los índices de los routers correspondientes a una o varias ruta. Su valor es un vector que contiene una lista de puertos a través de los cuales se debe enrutar el tráfico para llegar al router intermedio, si hay más de uno.

```
1: function Next(current_router, target_router, target_server, topology, virtual_channels, rou-
   ting_info)
2:     num_ports ← GetPorts(topology, current_router)
3:     r ← CreateList(capacity = num_ports × num_virtual_channels)
4:     distance ← GetDistance(current_router, target_router)
5:     if distance = 0 then
6:         target_server ← GetTargetServer(target_server)
7:         for all port in 0 to num_ports do
8:             if IsTargetServer(topology, current_router, port, target_server) then
9:                 return CreateCandidates(port, virtual_channels[0])
10:        RaiseError("Target server not reachable")
11:    else if distance = 1 then
12:        for all port in 0 to num_ports do
13:            if IsNeighborRouter(topology, current_router, port, target_router) then
14:                AddCandidates(r, port, virtual_channels[0])
15:    else if distance = 2 and routing_info.hops = 2 then
16:        next_port ← Next_Router_1(routing_table, current_router, target_router)
17:        AddCandidates(r, next_port, virtual_channels[0])
18:    else if distance = 3 then
19:        next_port ← Next_Router_2(routing_table, current_router, target_router, topology)
20:        AddCandidates(r, next_port, virtual_channels[1])
21:    else if (distance = 2 and routing_info.hops = 0) or distance = 4 then
22:        next_port ← Next_Router_3(routing_table, current_router, target_router, topology)
23:        AddCandidates(r, next_port, virtual_channels[1])
24:    else
25:        RaiseError("Long routes not supported")
26:    return CreateRoutingCandidates(r)
```

Algoritmo 4.2: Pseudo-código de `next` para `UpDownDeroutingAlways`

Por ejemplo, lo que hace el algoritmo de `next_router_1` 4.4 es obtener la lista de puertos para un par de routers dados como argumentos a partir de `paths`. Después selecciona aleatoriamente un único puerto y lo devuelve. En la implementación siempre se asume que dicha lista tendrá por lo menos un elementos, ya que las topologías abordadas en este trabajo no tienen rutas de más de cuatro saltos, por lo que el caso de la lista vacía no se gestiona.

El pseudo-código de `next_router_2` 4.5 es similar en que busca el puerto y elige la ruta de forma random, pero la condición de búsqueda cambia. Ahora debe encontrar sólo las rutas en que el router origen es el mismo que el argumento `source_router` y el destino está a un salto de `destination_router`. Concretamente itera sobre los vecinos de `destination_router` para guardar los que están a un up/down de `source_router`. Después elige uno aleatorio, al igual que el puerto que lleva hacia él si hay varios.

El algoritmo de `next_router_3` 4.6, por otro lado, busca encontrar los routers hoja que están a dos saltos tanto del origen como del destino. Añade los valores de las rutas encontradas a `intermediates` y selecciona uno de ellos de manera random.

```
1:  function Next(current_router, target_router, target_server, num_virtual_channels)
2:      num_ports ← GetPorts(topology, current_router)
3:      r ← CreateList(capacity = num_ports × num_virtual_channels)
4:      distance ← GetDistance(current_router, target_router)
5:      if distance = 0 then
6:          for all port in 0 to num_ports do
7:              if IsTargetServer(topology, current_router, port, target_server) then
8:                  return CreateCandidates(port, virtual_channels[0])
9:      else if distance = 1 then
10:         for all port in 0 to num_ports do
11:             if IsNeighborRouter(topology, current_router, port, target_router) then
12:                 AddCandidates(r, port, virtual_channels[0])
13:     else if distance = 2 then
14:         next_port ← Next_Router_1(routing_table, current_router, target_router)
15:         AddCandidates(r, next_port, virtual_channels[0])
16:     else if distance = 3 then
17:         next_port ← Next_Router_2(routing_table, current_router, target_router, topology)
18:         AddCandidates(r, next_port, virtual_channels[1])
19:     else if distance = 4 then
20:         next_port ← Next_Router_3(routing_table, current_router, target_router, topology)
21:         AddCandidates(r, next_port, virtual_channels[1])
22:     else
23:         RaiseError("Long routes not supported")
24:     return CreateRoutingCandidates(r)
```

Algoritmo 4.3: Pseudo-código de `next` para `UpDownDeroutingLazy`

```
1:  function Next_Router_2(source_router, destination_router, topology)
2:      neighbours ← CreateEmptyList
3:      for all port in Ports(destination_router) do
4:          neighbour_router ← GetNeighbourRouter(destination_router, port)
5:          if PathExists(source_router, neighbour_router) then
6:              Append(neighbours, neighbour_router)
7:      random_neighbour ← SelectRandom(neighbours)
8:      intermediates ← GetPaths(source_router, random_neighbour)
9:      random_intermediate ← SelectRandom(intermediates)
10:     return random_intermediate
```

Algoritmo 4.5: Pseudo-código de `next_router_2`

```
1: function Next_Router_1(source_router, destination_router)
2:     intermediates ← GetPaths(source_router, destination_router)
3:     len_intermediates ← Length(intermediates)
4:     random_index ← GenerateRandomIndex(0, len_intermediates - 1)
5:     random_intermediate ← intermediates[random_index]
6:     return random_intermediate
```

Algoritmo 4.4: Pseudo-código de `next_router_1`

```
1: function Next_Router_3(source_router, destination_router, topology)
2:     intermediates ← CreateEmptyList
3:     for all leaf_router in leaf_routers do
4:         if PathExists(source_router, leaf_router) and PathExists(leaf_router, destination_router) then
5:             intermediates ← GetPaths(source_router, leaf_router)
6:     random_intermediates ← SelectRandom(intermediates)
7:     random_intermediate ← SelectRandom(random_intermediates)
8:     return random_intermediate
```

Algoritmo 4.6: Pseudo-código de `next_router_3`

4.4. Experimentos con routing no mínimo para la mejora del rendimiento

En esta sección se muestran los resultados de unos experimentos con las topologías FT y OFT. Estos se separan en dos grupos, topologías cuyos routers son de radix 12 y 36, con un número de servidores equiparables, y sus parámetros se indican en la tabla 4.1. Únicamente en el FT se usó encaminamiento mínimo UpDown para tenerlo como referencia de la situación ideal.

Topología	Radix	Niveles	Routers	Servidores
FT	12	3	$72 + 72 + 36 = \mathbf{180}$	$2 \times 6^3 = \mathbf{432}$
OFT	12	$2(q = 5)$	$62 + 31 = \mathbf{93}$	$62 \times 6 = \mathbf{372}$
FT	36	3	$36 \times 18 + 36 \times 18 + 18 \times 18 = \mathbf{1620}$	$38 \times 18 \times 18 = \mathbf{11664}$
OFT	36	$2(q = 17)$	$2 \times 307 + 307 = \mathbf{921}$	$2 \times 307 \times 18 = \mathbf{11052}$

Tabla 4.1: Parámetros para los FT y OFT propuestos

En el primer FT cada router tiene 12 conexiones, luego cada router raíz está conectado a 12 pods. En el nivel superior (spine) hay 36 routers y en los siguientes 72 en cada uno. Pues la configuración necesaria es la misma que la mostrada en el código 4.2. Las Fig.

```
CartesianTransform {
    sides: [6,62],
    patterns: [Identity, RandomPermutation],
    legend_name: "random switch permutation" }
```

Código 4.7: Configuración de `CartesianTransform`

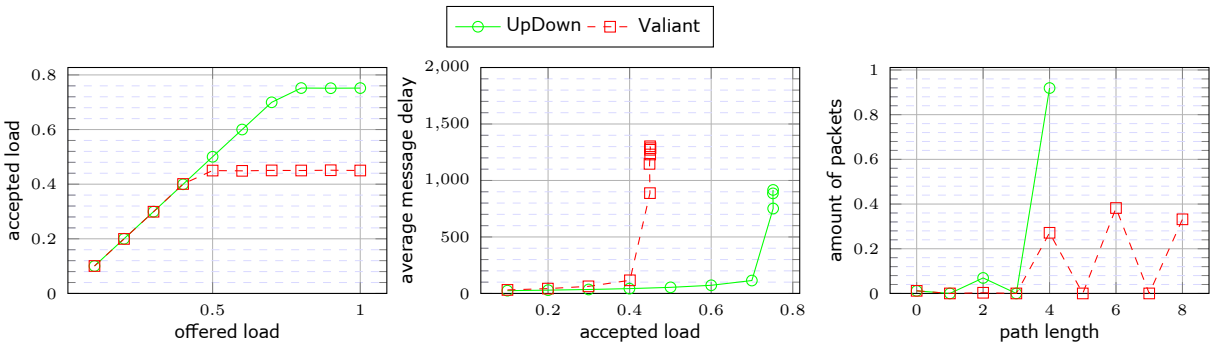

Figura 4.1: FT (12, 432) con tráfico uniforme, donde 12 indica el radix y 432 el número de servidores

4.1 y 4.2 incluyen las gráficas obtenidas con tráfico uniforme y permutación random, respectivamente, y se puede observar que su rendimiento con enrutamiento mínimo es similar para ambos tráficos, luego se toma como referencia. El FT con Valiant, por otro lado, sólo consigue la mitad de throughput y llega a dar hasta ocho saltos, porque al tener tres niveles su diámetro es 4.

En cuanto al OFT, como se ha visto que el rendimiento no mejora en la permutación random de servidores con Valiant tras realizar unas sencillas pruebas (Fig. 4.3, 4.4), para esta sección se ha utilizado la permutación aleatoria de routers hoja (random switch permutation), más agresiva que la otra permutación (random server permutation). Este tráfico se llama `CartesianTransform` en el simulador y es un patrón que transforma las coordenadas de origen con operaciones matemáticas. Los parámetros usados en este caso son los del código 4.7, donde `sides` define las dimensiones del OFT de radix 12, con 62 routers hoja y 6 servidores por router hoja. El valor de `patterns` indica que en la segunda dimensión se aplica permutación random, mientras que la primera se mantiene. Luego en Fig. 4.5 se muestra cómo el throughput para enrutamiento mínimo se reduce considerablemente en comparación con Valiant.

Comparando con las Fig. 4.6 y 4.7 se observa que con `UpDownDeroutingAlways` y tráfico de permutación de switches se obtiene un rendimiento muy parecido al obtenido con Valiant, alrededor de 0.4. Además, se puede apreciar que la amplia mayoría de los paquetes dan hasta cuatro saltos.

En la escala mayor de los FT (Fig. 4.8, 4.9) los resultados, con los mismos tráficos, se ven similares a los de la escala menor, aunque en el caso de los OFT (4.10, 4.11) se ve un pequeño deterioro en el rendimiento. Para este grupo de topologías se ha cambiando el valor de `sides` a [18, 614] en la configuración de `CartesianTransform`.

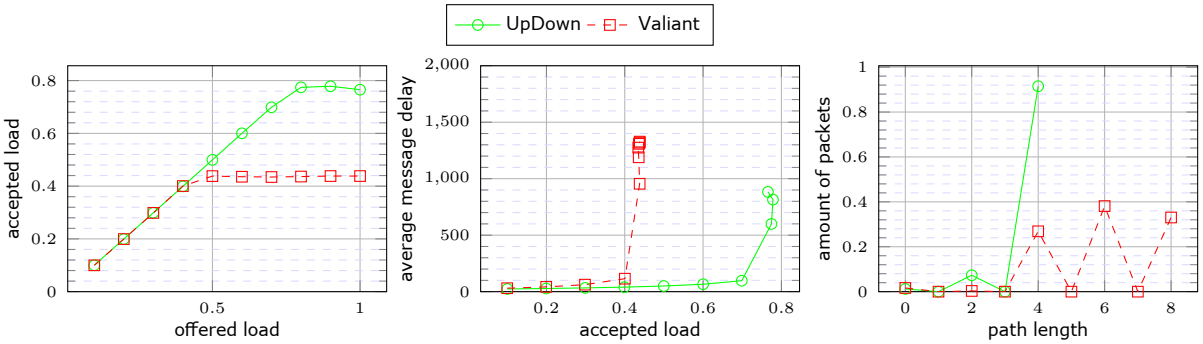

Figura 4.2: FT(12, 432) con tráfico permutación random

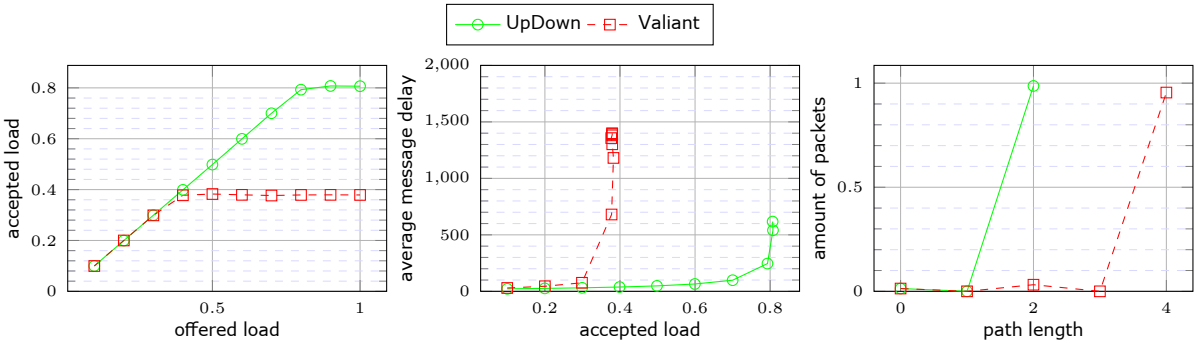

Figura 4.3: OFT(12, 372) con tráfico uniforme

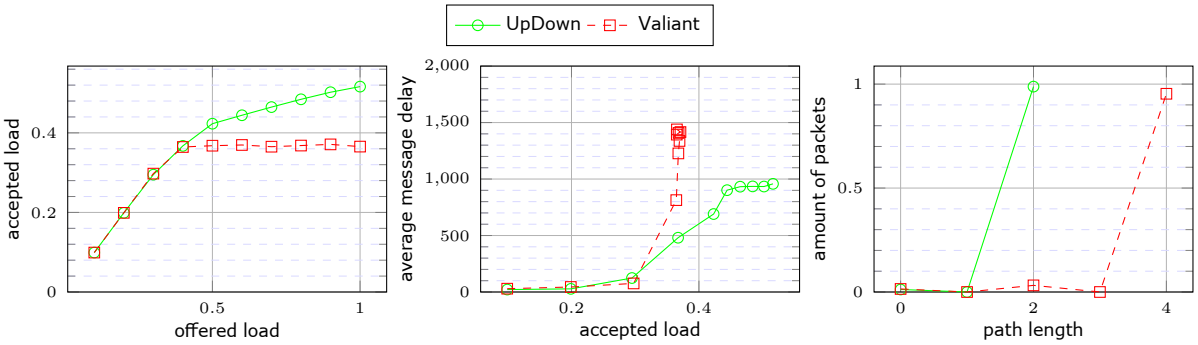

Figura 4.4: OFT(12, 372) con tráfico permutación random de servers

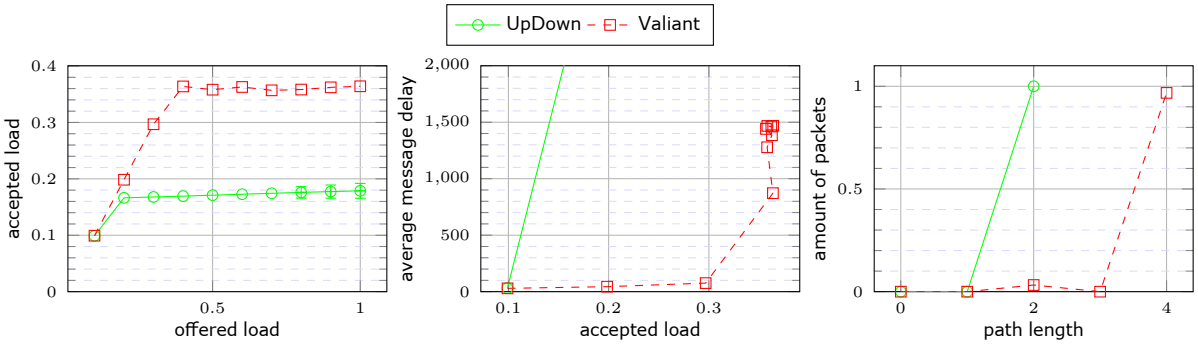

Figura 4.5: OFT(12, 372) con tráfico permutación random de switches

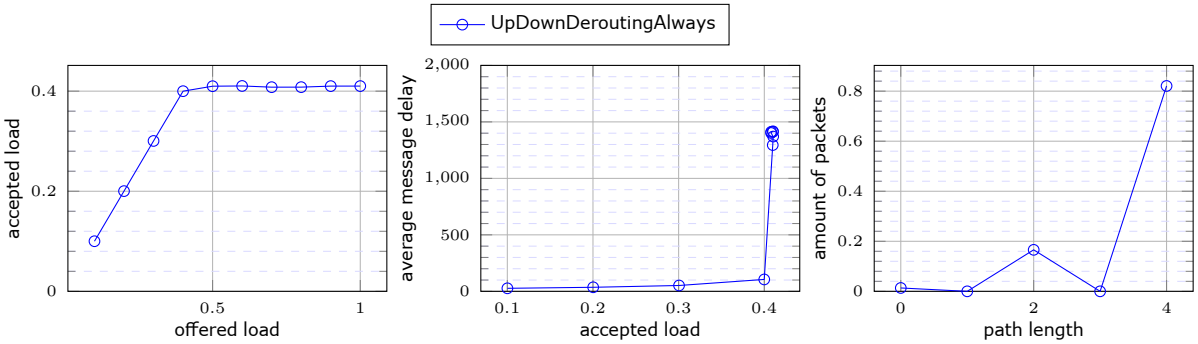

Figura 4.6: OFT(12, 372) con tráfico uniforme

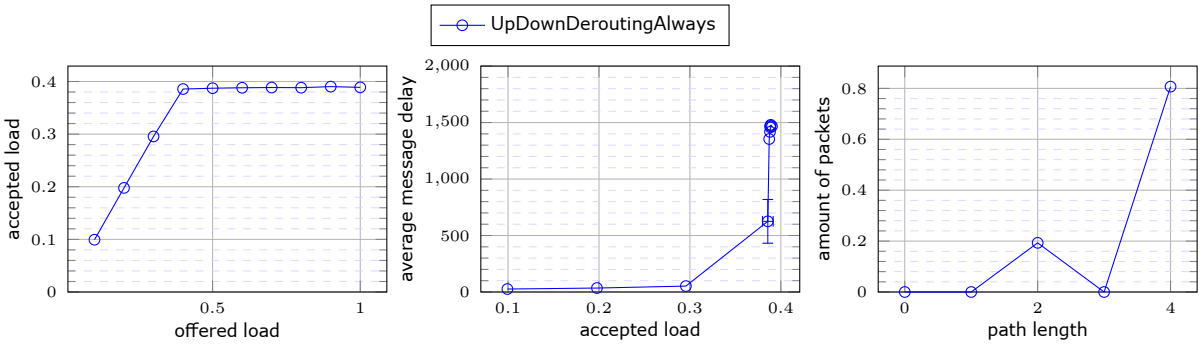

Figura 4.7: OFT(12, 372) con tráfico permutación random de switches

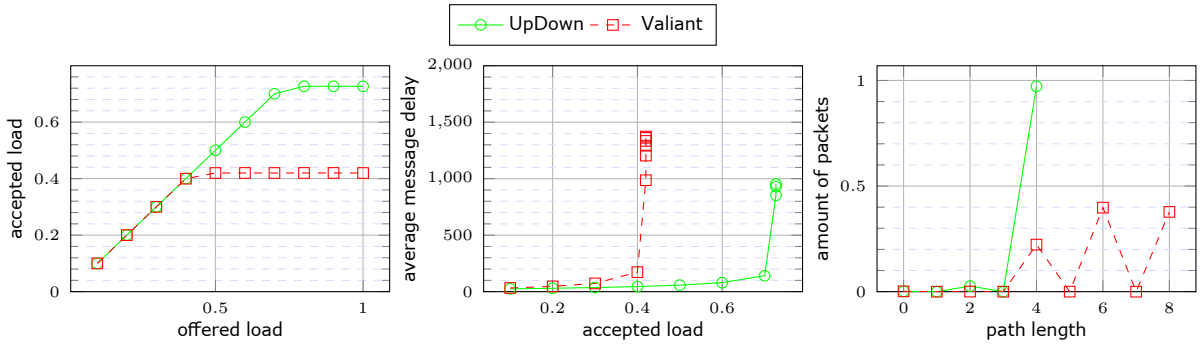

Figura 4.8: FT(36, 11664) con tráfico uniforme

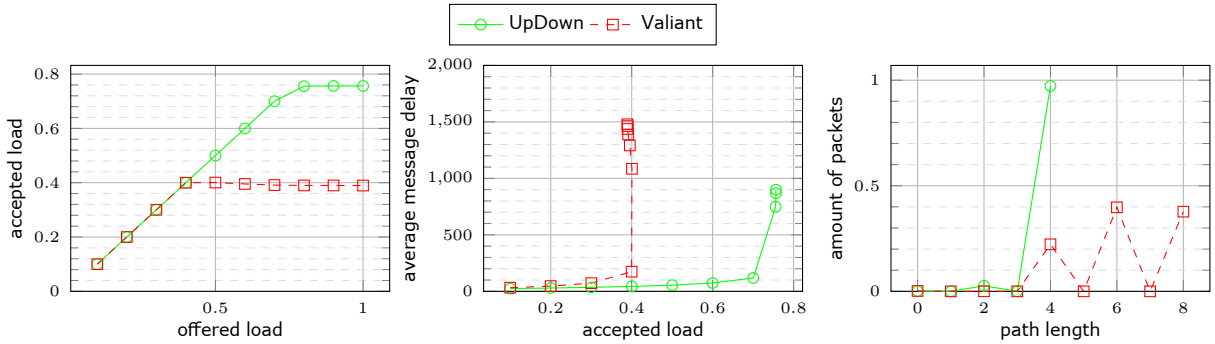

Figura 4.9: FT(36, 11664) con tráfico permutación random

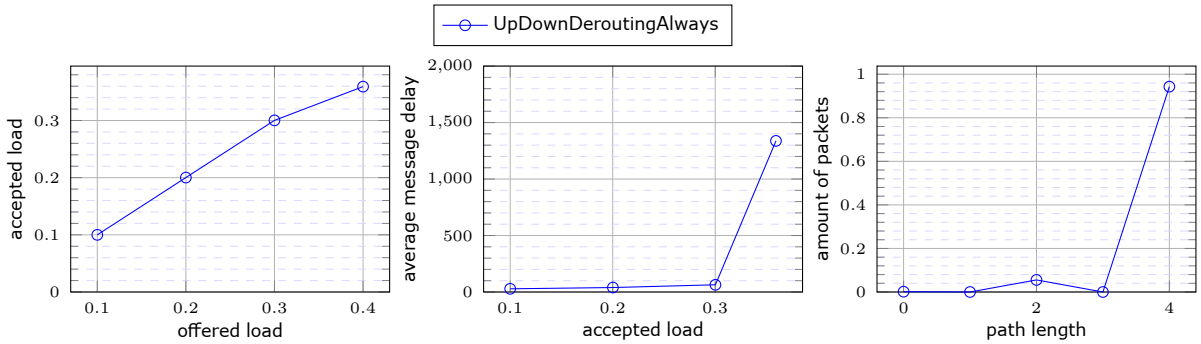

Figura 4.10: OFT(36, 11052) con tráfico uniforme

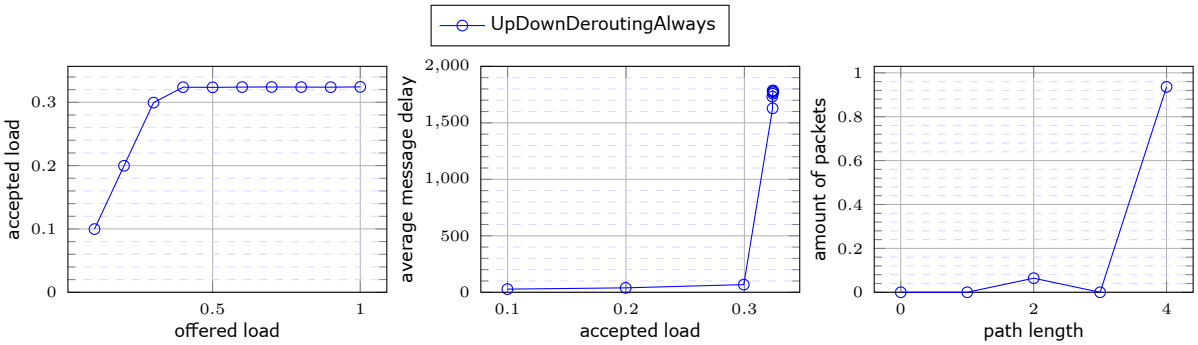

Figura 4.11: OFT(36, 11052) con tráfico permutación random de switches

4.5. Experimentos con routing no mínimo para garantizar la existencia de rutas

Los experimentos detallados en esta sección incluyen redes indirectas, en las que no siempre existen rutas up/down entre cada par de servidores, excepto la primera, que es un caso pequeño, y son los de la tabla 4.2. Al igual que en la sección anterior, estos experimentos se agrupan en topologías de radix 12 y 36. Dentro de cada grupo hay tres configuraciones de RFC: una up/down conectada, una con unos pocos pares de servidores que no tienen rutas mínimas, y otra donde la mayoría de parejas no está conectadas por rutas up/down.

Topología	Radix	Niveles	Routers	Servidores
RFC	12	2	$22 + 11 = \mathbf{33}$	$22 \times 6 = \mathbf{132}$
RFC	12	2	$26 + 13 = \mathbf{39}$	$26 \times 6 = \mathbf{156}$
RFC	12	2	$62 + 31 = \mathbf{93}$	$62 \times 6 = \mathbf{372}$
RFC	36	2	$90 + 45 = \mathbf{135}$	$90 \times 18 = \mathbf{1620}$
RFC	36	2	$130 + 65 = \mathbf{195}$	$130 \times 18 = \mathbf{2340}$
RFC	36	2	$614 + 307 = \mathbf{921}$	$614 \times 18 = \mathbf{11052}$

Tabla 4.2: Parámetros para las RFC propuestas

La primera topología es una RFC pequeña que está up/down conectada, por lo que se ha podido realizar una prueba con el enrutamiento `UpDown` original, también con la opción `selection_exclude_indirect_routers` a `true`. Como el misrouting `UpDownDeroutingLazy` se ha diseñado para utilizar un canal virtual diferente para cada up/down de la ruta, concretamente el 1 para el primer up/down y el 0 para el segundo (resultando en un enrutamiento más determinista), se han limitado los recursos del router y se puede apreciar la diferencia de rendimiento entre el routing `UpDown` y

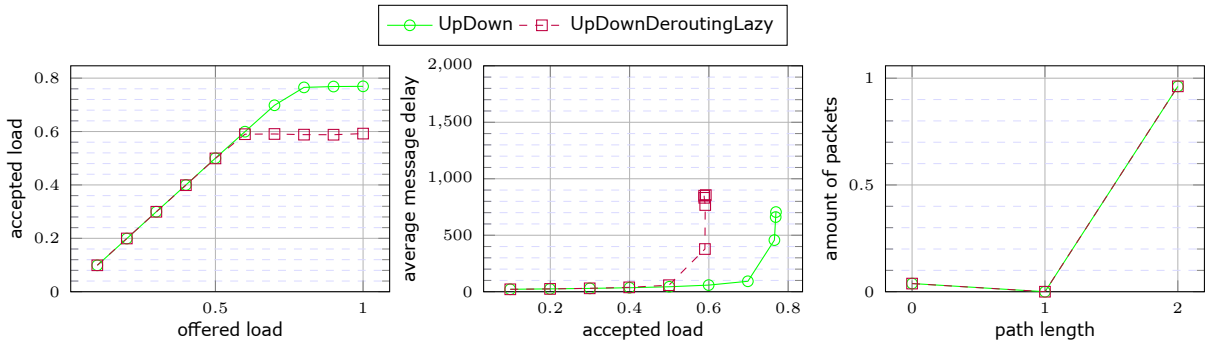

Figura 4.12: RFC(12, 132) con tráfico uniforme

`UpDownDeroutingLazy` en las Fig. 4.12 y 4.13. El primero proporciona mayor throughput y menor latencia debido a que, al asignar varios canales virtuales a los flujos, se reduce la contención. Además, cuando se elige el candidato al siguiente salto en la implementación de `UpDownDeroutingLazy`, se obtiene el router siguiente conectado a un puerto únicamente del actual, lo que supone otra limitación que podría influir en el rendimiento.

La RFC de 156 servidores sigue siendo pequeña, pero ya tiene parejas de origen-destino desconectadas, de ahí que aparezcan rutas de cuatro saltos (Fig. 4.14, 4.15). En comparación con la red anterior el rendimiento es muy parecido, ya que no hay una diferencia significativa en escala, pues se mantiene en una carga aceptada de 0.6.

La última RFC de radix 12 representa un caso más extremo donde, como ya se ha mencionado, gran parte de los pares origen-destino no están conectados por rutas mínimas. Dado su tamaño de 372 servidores, es comparable con el OFT del primer grupo de experimentos de la sección anterior. En Fig. 4.16 y 4.17 se puede ver que el rendimiento de la RFC mejora en relación con el OFT, pues se consigue hasta un 0.5 de carga aceptada, mientras que el OFT obtiene aproximadamente 0.4 con tráfico uniforme. Pues al hacer `UpDownDeroutingLazy` se aprovecha las rutas mínimas, a diferencia del OFT, donde la mayoría de las distancias son más largas.

En la escala mayor de 1620 servidores, donde la amplia mayoría de RFCs con los parámetros de la tabla anterior están up/down conectadas, se ve que todas las rutas realizadas son de dos saltos (Fig. 4.18, 4.19).

La RFC de 2340 ya empieza a carecer de conexiones up/down y, según los resultados de Fig. 4.20 y 4.21, el throughput es similar a la red anterior.

Finalmente, la RFC de 11052 servidores se compara con su equivalente del OFT (Fig. 4.22, 4.23) y también presenta un throughput mejor.

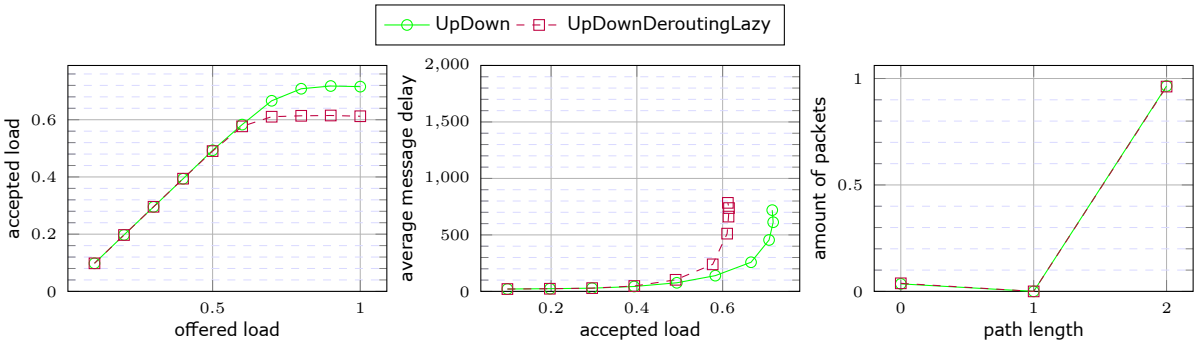

Figura 4.13: RFC(12,132) con tráfico de permutación random

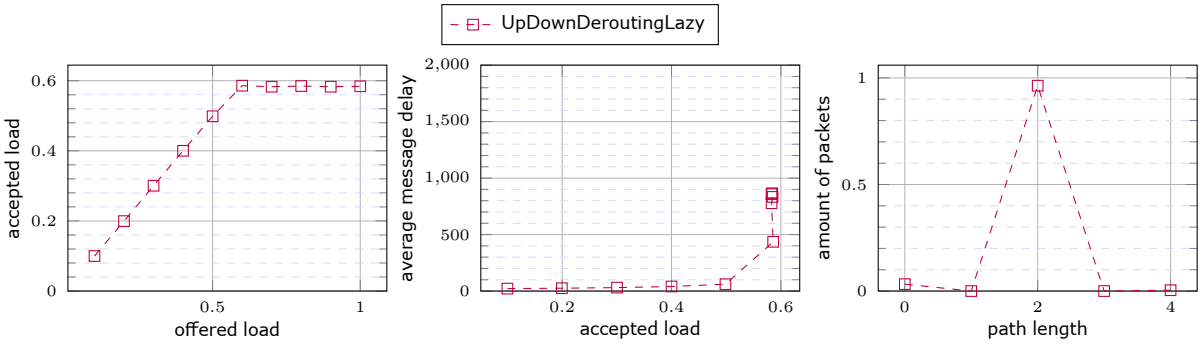

Figura 4.14: RFC(12, 156) con tráfico uniforme

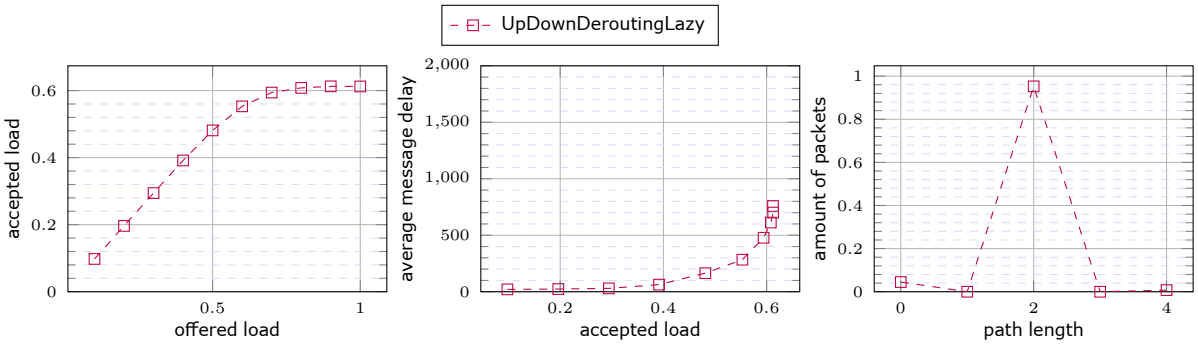

Figura 4.15: RFC(12, 156) con tráfico de permutación random

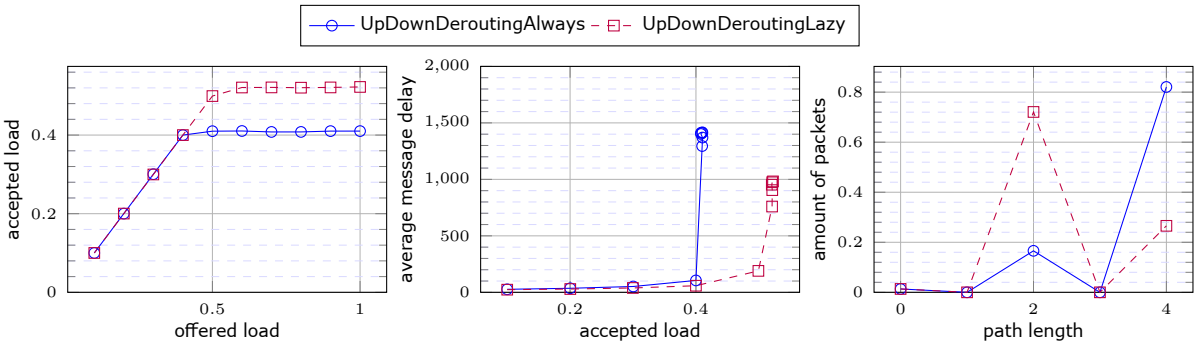

Figura 4.16: OFT(12, 372) y RFC(12, 372) con tráfico uniforme

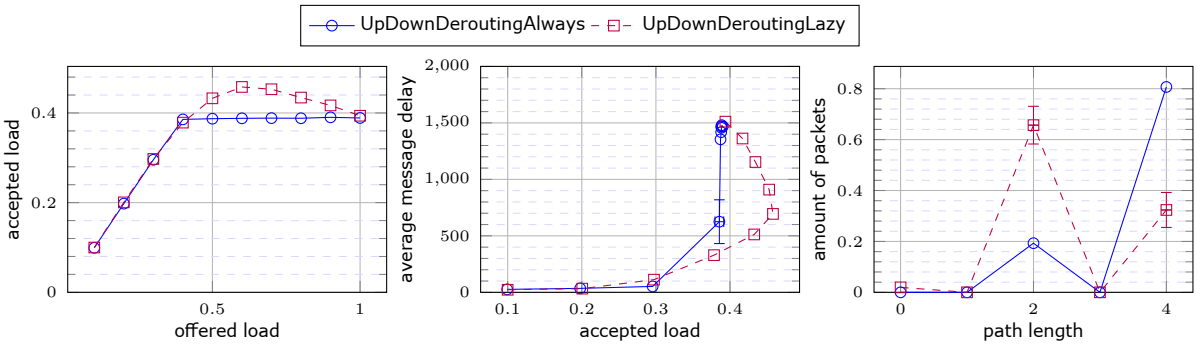

Figura 4.17: OFT(12, 372) y RFC(12, 372) con tráfico de permutación random

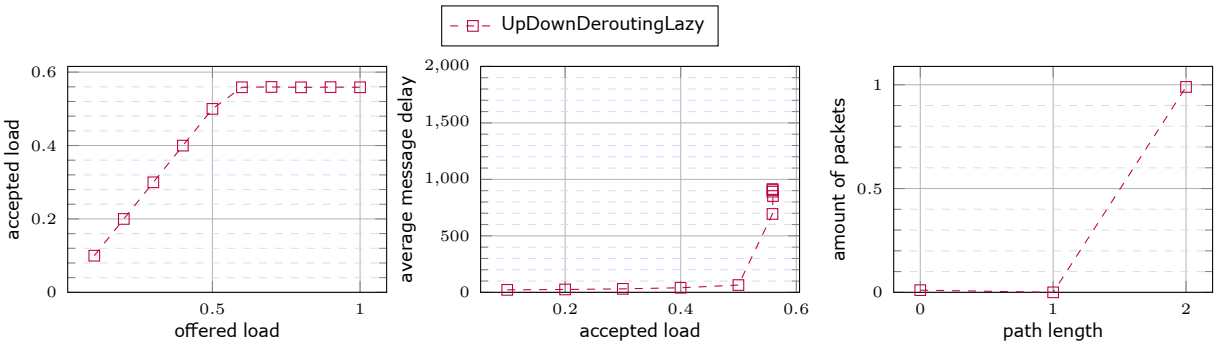

Figura 4.18: RFC(36, 1620) con tráfico uniforme

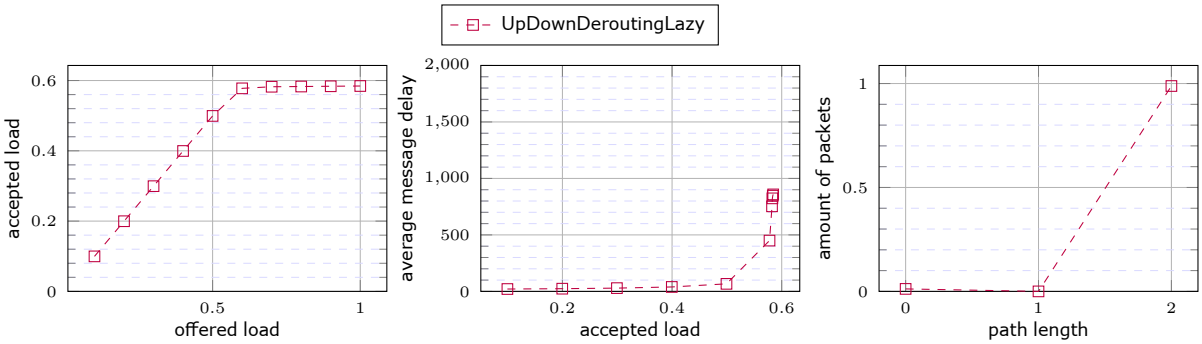

Figura 4.19: RFC(36, 1620) con tráfico de permutación random

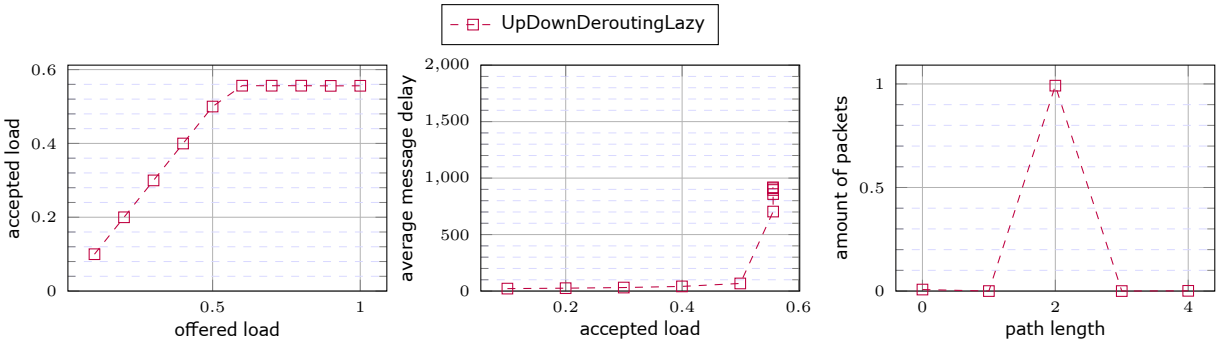

Figura 4.20: RFC(36, 2340) con tráfico uniforme

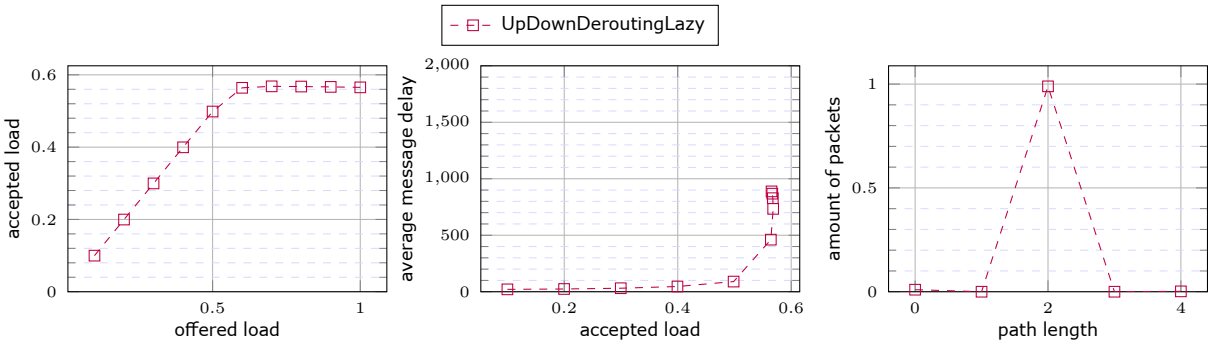

Figura 4.21: RFC(36, 2340) con tráfico de permutación random

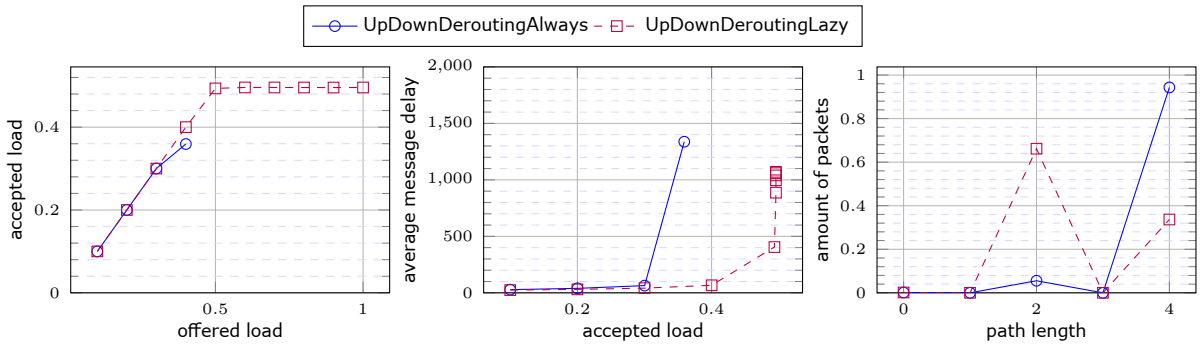

Figura 4.22: RFC(36, 11052) con tráfico uniforme

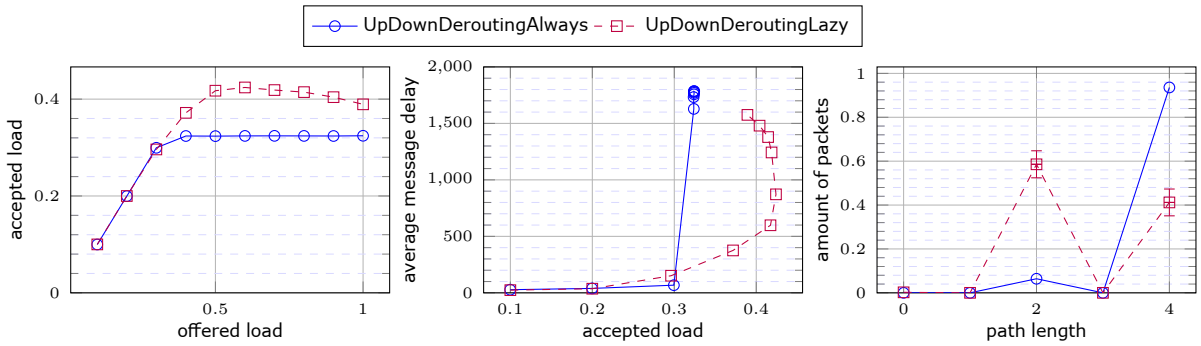

Figura 4.23: RFC(36, 11052) con tráfico de permutación random

5. Conclusiones y discusión final

Este trabajo propone el empleo del encaminamiento no mínimo en el Orthogonal Fat-Tree y Random Folded Clos, dos topologías con potencial para ser utilizadas en sistemas HPC, donde generalmente se encuentran Fat-Trees. Su objetivo es mejorar el rendimiento bajo condiciones de tráfico estresante y maximizar la escalabilidad, manteniendo un coste menor que el del Fat-Tree. Para ello se han diseñado y comprobado el funcionamiento de dos algoritmos específicos para las aplicaciones presentadas con enfoques ligeramente diferentes. En primer lugar, el enrutamiento `UpDownDeroutingAlways` destinado a mejorar el rendimiento del OFT para tráficos adversos, como random switch permutation, en cuyo caso se ha visto que no muestra una mejoría significativa frente a Valiant, pues los resultados quedan muy similares. Y en segundo lugar, el enrutamiento `UpDownDeroutingLazy` ante la ausencia de rutas up/down, que ha permitido realizar el misrouting en topologías de mayor extensión, como la RFC de radix 36 y 11052 servidores, donde las distancias de dos saltos entre parejas origen-destino ya son escasas, y es equiparable al OFT del mismo radix. Además, el rendimiento obtenido es razonable y se podría mejorar modificando la implementación.

En trabajos futuros se consideraría lo siguiente:

1. Extender las topologías a más de dos niveles, aunque estas ya proporcionan bastante escalabilidad.
2. Modificar el routing para que pueda realizar más deroutes (desvíos de la ruta mínima).
3. Implementar un enrutamiento que pueda tomar decisiones o adaptarse en función del nivel de congestión en los OFT, así mitigando posibles cuellos de botella. Por ejemplo, un algoritmo que pueda elegir cuándo hacer un tipo de routing u otro, como una combinación entre `UpDownDeroutingAlways` y `UpDownDeroutingLazy`.
4. Añadir optimización de los algoritmos.
5. Integrar mecanismos de tolerancia a fallos en el enrutamiento.

Referencias

[1] TOP500 Team. Top500: The list. https://www.top500.org/, 2024. Accedido: 08-ago-2024.

[2] Marcos Valerio, LE Moser, and PM Melliar-Smith. Fault-tolerant orthogonal fat-trees as interconnection networks. In Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing, volume 2, pages 749–754. IEEE, 1995.

[3] Georgios Kathareios, Cyriel Minkenberg, Bogdan Prisacari, German Rodriguez, and Torsten Hoefler. Cost-effective diameter-two topologies: analysis and evaluation. In SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2015.

[4] Cristóbal Camarero, Carmen Martínez, and Ramón Beivide. Random folded clos topologies for datacenter networks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 193–204, 2017.

[5] Caesar Wu and Rajkumar Buyya. Chapter 13 - data center networks. In Caesar Wu and Rajkumar Buyya, editors, Cloud Data Centers and Cost Modeling, pages 497–576. Morgan Kaufmann, 2015.

[6] Alan Dr and José Moreira. IBM Blue Gene Supercomputer, pages 891–900. 01 2011.

[7] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[8] Charles Clos. A study of non-blocking switching networks. The Bell System Technical Journal, 32(2):406–424, 1953.

[9] A. Jajszczyk. Nonblocking, repackable, and rearrangeable clos networks: fifty years of the theory evolution. IEEE Communications Magazine, 41(10):28–33, 2003.

[10] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Transactions on Parallel and Distributed Systems, 4(12):1320–1331, 1993.

[11] W.J. Dally. Virtual-channel flow control. IEEE Transactions on Parallel and Distributed Systems, 3(2):194–205, 1992.

[12] William Dally and Brian Towles. Principles and practices of interconnection network. 01 2004.

[13] Aniruddh Ramrakhyani, Paul V. Gratz, and Tushar Krishna. Synchronized progress in interconnection networks (spin): A new theory for deadlock freedom. In 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pages 699–711, 2018.

[14] L. G. Valiant. A scheme for fast parallel communication. SIAM Journal on Computing, 11(2):350–361, 1982.

[15] Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 348–359, 2014.

[16] Cristóbal Camarero Coterillo. caminos-lib 0.6.3: Cantabrian adaptable and modular interconnection open simulator. `https://crates.io/crates/caminos`, 2023. Versión 0.6.3, disponible en: `https://docs.rs/crate/caminos-lib/0.6.3`.

[17] Rust programming language. `https://www.rust-lang.org/`, 2024.

[18] Cristobal Camarero, Carmen Martínez, Enrique Vallejo, and Ramon Beivide. Projective networks: Topologies for large parallel computer systems. IEEE Transactions on Parallel and Distributed Systems, PP, 02 2017.