UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

Taller de Sistemas
Ciber Físicos

Curso 2020

Baja latencia y alta performance en datacenters: el protocolo NDP

Autor:
Belén Brandino

Docentes:
Leonardo Alberro
Alberto Castro
Eduardo Grampin

Contents

1 Introducción

El término sistemas ciber físicos (CPS por sus siglas en inglés) se refiere a una nueva generación de sistemas con capacidades físicas y computacionales integradas que pueden interactuar con los humanos a través de nuevas modalidades. La capacidad de interactuar y expandir las capacidades del mundo físico a través de la computación, la comunicación y el control es un habilitador clave para futuros desarrollos tecnológicos. En el correr de los años, los investigadores de sistemas y control han sido pioneros en el desarrollo de poderosos métodos y herramientas de ciencia e ingeniería de sistemas, como métodos de dominio de tiempo y frecuencia, análisis del espacio de estados, identificación de sistemas, filtrado, predicción, optimización y un largo etcétera. Al mismo tiempo, los investigadores en ciencias de la computación han logrado importantes avances en nuevos lenguajes de programación, técnicas de computación en tiempo real, métodos de visualización, diseños de compiladores, arquitecturas de sistemas integrados y software de sistemas, y enfoques innovadores para garantizar la confiabilidad del sistema informático, la seguridad cibernética y la tolerancia a fallas. Los investigadores en ciencias de la computación también han desarrollado una variedad de poderosos formalismos de modelado y herramientas de verificación. La investigación de sistemas ciber físicos tiene como objetivo integrar el conocimiento y los principios de la ingeniería en las disciplinas computacionales y de la ingeniería (redes, control, software, interacción humana, teoría del aprendizaje, así como también las ciencias eléctricas, mecánicas, químicas, biomédicas, de los materiales y otras disciplinas de la ingeniería) para desarrollar nueva ciencia CPS y tecnología de apoyo [1]. Un Data Center (DC) puede definirse como un sistema "complejo" ciber físico, ya que en él conviven aspectos informáticos y energéticos junto con sus interdependencias [6].

Los servicios en la nube en la última década se han vuelto indispensables. Tradicionalmente, cada organización mantenía servidores web, de correo electrónico, etc. en su propio sitio, lo cual no resulta adecuado para satisfacer las crecientes necesidades de utilizar una variedad de funcionalidades a gran escala. Uno de los servicios que ofrece la nube es el *web hosting*, ya que muchas organizaciones pequeñas no quieren mantener sus propios servidores por razones

de costo y seguridad. También, las empresas han estado trasladando sus servicios a la nube por factores económicos, seguridad, capacidad de gestión y escalabilidad. Otro uso de la nube es que usuarios finales puedan guardar sus datos personales, documentos, fotos, etc. y también para compartirlos con otros usuarios. En términos generales, la nube es extremadamente útil para escalar las necesidades de computación, almacenamiento, uso compartido y alojamiento, además de ser económico, confiable y eficiente. Por todo esto, la nube está viendo una variedad de necesidades para una amplia variedad de clientes, desde usuarios individuales hasta grandes empresas [14].

Como proveedor de servicios en la nube, tener solo un conjunto de *hosts* no es suficiente para satisfacer las crecientes necesidades, sino que se debe tener todos los componentes asociados con la prestación de servicios web. En este lugar es donde entra en juego la necesidad de los *datacenters* a gran escala. Para lograr proporcionar servicios en la nube a gran escala, los *datacenters* se organizan en forma de redes de centros de datos (*data center networks* - DCN). Es importante tener en cuenta que para proporcionar servicios en la nube, un proveedor debe tener *datacenters* que sean lo suficientemente flexibles, de modo que puedan ser ampliados sin una reorganización masiva. En particular, un *datacenter* puede tener una gran cantidad de servidores con soporte para máquinas virtuales, así como amplias instalaciones de almacenamiento. Para mayor confiabilidad, dicha infraestructura debe contar con redundancia suficiente, ya que es común en los *datacenters* que los servidores y dispositivos de almacenamiento fallen de forma regular. Por lo tanto, la confiabilidad de los servicios prestados mediante una redundancia adecuada es importante. [14]

En la figura 1, se puede observar una topología simple de árbol para DCN. En la parte inferior, se encuentran los *racks* de servidores a los que se conectan los *hosts* físicos. Los *hosts* están conectados a los *switches Top-of-Rack* (ToR), que básicamente se encuentran en el medio del *rack* del servidor en una configuración física para menos cableado. Luego, los *switches* ToR que están en una fila se conectan a un *switch* de *End of Row* (EoR) o de borde. Varias filas de servidores con un *switch* de borde en cada fila están conectadas en un *switch* de agregación para la agregación del tráfico. Finalmente, los *switches* de agregación están

conectados a uno o más *switches core*, el cual tiene un enlace saliente para conectarse a Internet. Claramente, puede haber más de un *switch core* y se pueden instalar varios enlaces paralelos para la conectividad a Internet. Hay dos tipos de tráfico que se pueden imaginar en una DCN: tráfico este-oeste, que refiere tráfico entre *racks* de servidores, que es el resultado de aplicaciones internas que requieren transferencias de datos y tráfico norte-sur, que refiere a tráfico como resultado de solicitudes externas de Internet que llegan al *datacenter*, a las que los servidores deben responder. Según el propósito comercial específico de un *datacenter*, la cantidad de tráfico que contribuye al tráfico de este a oeste en comparación con el tráfico de norte a sur podría variar ampliamente. [14]

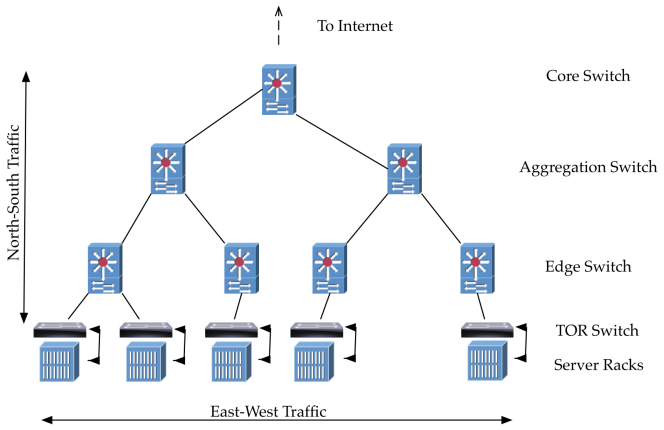


Figure 1: Topología Tree para DCN [14]

Una topología de tipo *Tree* muy utilizada en *datacenters* es la topología *Fat Tree*, que son un caso especial de una *Clos network*. Normalmente se hace referencia a la topolgía *Fat Tree* en términos de cantidad de *pods*. Se numeran de izquierda a derecha como *Pod 0* a *Pod k-1*, donde la topología consiste en k *pods*, con tres capas de *switches*: *switches* de borde, de agregación y *core*. Entonces, en una topología *Fat Tree* de k *pods*, hay k *switches* (cada uno con k puertos) en cada *pod*, dispuestos en dos capas de $k/2$ *switches*, una capa para *switches* de borde y la otra para *switches* de agregación. Cada *switch* de borde está conectado a $k/2$ *switches* de agregación. Hay $(k/2)^2$ *switches core*, cada uno de los cuales se conecta a k *pods*. Es posible ver una topología *Fat Tree* de cuatro *pods* en la figura 2 y la información de la topología en la figura 3 [14].

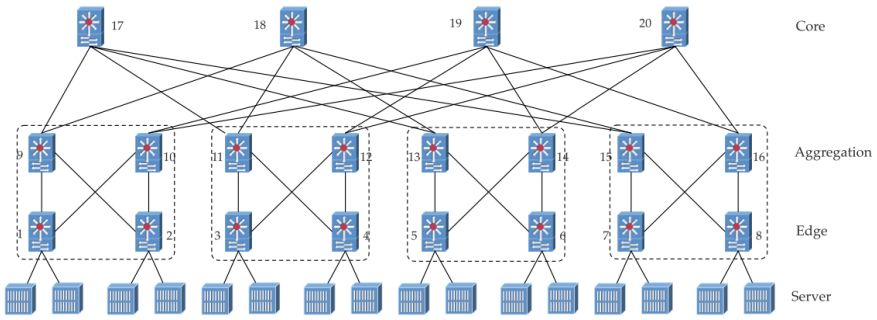


Figure 2: Topología Fat Tree para DCN [14]

Table 12.1 Fat-tree topology summary.

Number of Pods	***k***
Number of Core Switches	$(k/2)^2$
Number of Aggregation Switches	$k^2/2$
Number of Edge Switches	$k^2/2$
Number of Switches, N (all types)	$5k^2/4$
Number of Links, L	$k^3/2$
Number of hosts supported	$k^3/4$

Figure 3: Información Topología Fat Tree para DCN [14]

Las topologías de *Fat Tree* tienen ciertos beneficios. Entre ellos, todos los *switches* son del mismo tipo con el mismo número de puertos, y cada puerto normalmente proporciona la misma velocidad; los hosts finales también admiten la misma velocidad. Además, hay varias rutas entre dos hosts [14].

Las redes de *datacenters* modernos proveen alta capacidad y baja latencia, mediante topologías de Clos redundantes como las mencionadas anteriormente, pero raramente los protocolos de transporte proveen la misma *performance*. NDP es una arquitectura de transporte para *datacenters* que logra tiempos casi óptimos para transferencias cortas y un alto *throughput* para flujos en múltiples escenarios, incluyendo *incast*[1]. Los *buffers* de los *switches* NDP son muy pequeños y cuando se llenan, los *switches* recortan los paquetes quedándose con los encabezados y luego prioriza su envío. Esto brinda a los receptores una visión completa de

[1]Comunicación *many-to-one* (muchos *senders* y un solo *receiver*)

la demanda instantánea de todos los emisores, y es la base del protocolo de transporte NDP, de alto rendimiento y con reconocimiento de múltiples rutas, que puede manejar eventos de *incast* masivos y priorizar el tráfico de diferentes emisores en escalas de tiempo RTT [12].

Con el fin de encontrar una alternativa para poder lograr baja latencia y un alto *throughput*, NDP no tiene un *setup handshake* de conexión (como el *three way handshake* de TCP), sino que permite que se empiecen a enviar flujos de manera instantánea a la velocidad del *link* (*full rate*). Utiliza *per-packet multipath* como balanceo de carga, de manera de evitar la congestión en el núcleo de la red, a expensas de lidiar con la reordenación. Los *switches* NDP utilizan un enfoque similar a *Cut Payload* (CP), que recorta los *payloads* de los paquetes cuando se llena una cola de un *switch* y se queda con los encabezados. Esto proporciona una red sin pérdidas para los metadatos, pero no para *payloads* de tráfico. Esto le da al receptor una imagen completa del tráfico entrante y eso se aprovecha para construir un protocolo de transporte nuevo, que logra una latencia muy baja para flujos cortos, con mínima interferencia entre flujos a diferentes destinos incluso en patrones de tráfico patológicos [12].

NDP fue implementado en *hosts* Linux en el espacio de usuario utilizando la librería DPDK, en un *switch* de software, en un *switch* de hardware basado en NetFPGA, en P4 y en simulación. Se evaluó el desempeño de NDP de estas implementaciones y en simulaciones a gran escala, demostrando simultáneamente soporte para muy baja latencia y alto rendimiento. Las pruebas han demostrado que NDP logra [12]:

- Mejor rendimiento en flujos cortos (*short-flow performance*) que DCTCP o DCQCN
- Más del 95% de la capacidad máxima de la red en una red muy cargada con colas de *switches* de solo ocho paquetes
- *Delay* casi perfecto y equidad en escenarios de *incast*
- Interferencia mínima entre flujos a diferentes *hosts*
- Priorización eficaz del tráfico rezagado durante *incast*

Este documento tiene como fin presentar el trabajo realizado en base al paper presentado en Sigcomm 2017, *"Re-architecting datacenter networks and stacks for low latency and high performance"*[12]. El mismo trata sobre una nueva arquitectura de transporte para *datacenters*, llamada NDP. En particular, se realizó un estudio en profundidad del protocolo, se realizaron distintos *tests* con el simulador de implementación de NDP disponible en [13] y también se probó una aplicación ejemplo utilizando la implementación del *stack* NDP en Linux, también disponible en [13].

El documento se organiza de la siguiente manera: en la sección Protocolo NDP se presentan los conceptos principales del protocolo NDP, en la sección Simulador se presenta el simulador de NDP, junto con las distintas pruebas realizadas. Luego, en la sección Implementación en Linux se presenta la implementación del *stack* NDP en Linux, junto con la aplicación de ejemplo y las pruebas realizadas. Por último, en la sección Conclusiones se presentan las principales conclusiones de los estudios realizados.

2 Protocolo NDP

Como se mencionó anteriormente, el protocolo NDP busca lograr bajo *delay* y alto *throughput*. En particular, se busca mejorar el *stack* de protocolos, de manera que cada *request* pueda usar una nueva conexión y al mismo tiempo esperar acercarse a la latencia bruta y el ancho de banda de la red subyacente, incluso bajo carga pesada. A continuación se presenta el diseño del protocolo para poder cumplir con estos requerimientos, y como se implementa este diseño.

2.1 Diseño del protocolo

2.1.1 Demandas del servicio *end-to-end*

Las aplicaciones desean que un *datacenter* cumpla los requerimientos de:

- Independencia de ubicación: No debe importar en que máquina corren los elementos de una aplicación
- Baja latencia: Debe ser lo primero a priorizar

- Manejar los efectos del incast: Las cargas de trabajo de los *datacenters* por lo general realizan muchos *requests* y reciben sus respuestas de manera simultánea. Un buen *stack* debería ser capaz de protegerse de los efectos del *incast*

- Priorización: Es deseable que un receptor sea capaz de priorizar tráfico rezagado, ya que es el único capaz de priorizar el tráfico entrante, y por ende impacta en el diseño del protocolo

2.1.2 Protocolo de transporte

Varios protocolos de *datacenters* cumplen algunos de los requerimientos mencionados anteriormente, pero lograr cumplirlos todos resulta en algunas demandas inusuales para el protocolo. Estos son:

- Configuración de la conexión de *zero-RTT*: Para minimizar la latencia, muchas aplicaciones desearían una entrega con *zero-RTT* para pequeñas transferencias salientes. Para esto, se requiere de un protocolo que no necesite de un *handshake* (por ejemplo, el *three way handshake* de TCP) completo para empezar a enviar datos. Esto puede traer problemas de correctitud y seguridad

- Comienzo rápido: Otra implicación de entrega con *zero-RTT* es que un protocolo de transporte no puede probar por el ancho de banda disponible. Debe asumir que todo el ancho de banda está disponible, enviando una ventana inicial completa, y reaccionar de manera adecuada cuando no está. A diferencia de Internet, se pueden implementar soluciones más simples en un *datacenter*, ya que la velocidad de los *links* y el *delay* de la red pueden ser mayoritariamente conocidos de antemano

- Per-packet ECMP: Un problema con las topologías de Clos es que el *hash* per-flow ECMP de los flujos a las rutas, puede causar colisiones de flujos no deseadas; una implementación encontró que esto reduce el rendimiento en un 40%. Para transferencias grandes, los protocolos *multipath* como MPTCP pueden establecer suficientes subflujos para encontrar rutas no utilizadas, pero no pueden hacer mucho para ayudar con la latencia de transferencias

muy cortas. La única solución restante es dividir en varias rutas por paquete (*per-packet*). Esto hace más complicado el diseño del protocolo de transporte

- *Handshake* tolerante al reordenamiento: Si se realiza una transferencia con *zero-RTT* con *per-packet multipath forwarding*, incluso la primer ventana de paquetes puede llegar en desorden, por lo que el primer paquete en llegar no tiene por qué ser el primer paquete de la conexión. El protocolo de tranpsorte debe ser capaz de establecer la conexión sin importar cual es el primer paquete en llegar de la ventana inicial.

- Optimizado para *incast*: Estos patrones de tráfico pueden causar altas tasas de pérdida de paquetes, especialmente si el protocolo de transporte es agresivo en el primer RTT. Para manejar esto se requiere de ayuda de los *switches*

2.1.3 Modelo de servicio para *switches*

Cuando se sobrecarga un link de un *switch*, es necesario decidir qué se hace con los paquetes que no entran. Existen varias maneras de abordar esto. Por defecto, los paquetes se descartan, lo cual es malo pero no por la retransmisión, ya que tiene poco costo. Sino, que el reenvío *per-packet multipath* hace difícil detectar si ocurrió una pérdida. Esto resulta en incertidumbre, lo cual causa *delay*. Como opuesto al descarte de paquetes, se tiene *Lossless Ethernet*, donde no ocurren pérdidas, pero se forman largas colas que generan *delay* en flujos no relacionados. Se busca entonces un punto medio entre estas soluciones, tomando como idea el *packet trimming* utilizado en CP (*Cut Payload*). En este caso, el *switch* recorta el *payload* del paquete, guardando el *header* en la cola, para que el receptor pueda tener esta información. De esta manera, evita la sobrecarga, mientras que también evita la incertidumbre sobre los resultados de los paquetes. La ventaja de este método es que no se pierde *metadata*, solo la carga útil, y el receptor sabe exactamente que se envió, incluso cuando existe reordenamiento. Por estas razones, el *switch* NDP planteado realiza tres cambios:

- Un *switch* NDP tiene dos colas: Una de baja prioridad para paquetes de datos, y una de alta prioridad para *headers*, paquetes ACK y NACK. De esta manera se comunica al emisor lo más rápido posible sobre un paquete

que no llegó. Con esta información se puede proceder a su reenvío de manera rápida.

- Round-robin ponderado: Se realiza un round-robin ponderado entre las dos colas, con un radio de 10:1 (10 para la cola de alta prioridad, 1 para la otra cola)

- Si la cola de baja prioridad está llena, cuando llega un paquete nuevo el *switch* decide con un 50% de probabilidad si recortar el paquete nuevo, o el último de la cola

2.1.4 Routing

Se desea que los *switches* NDP realicen *per-packet multihop forwarding* (más de un siguiente salto), para distribuir uniformemente las ráfagas de tráfico en todas las rutas paralelas que están disponibles entre el origen y el destino. Los experimentos realizados muestran que si los *senders* eligen las rutas, pueden hacer un mejor trabajo de balanceo de carga que si los *switches* eligen rutas al azar, que además permite el uso de *buffers* de *switches* ligeramente más pequeños. A diferencia de Internet, en un *datacenter*, los *senders* pueden conocer la topología, por lo que pueden saber cuántas rutas disponibles hay para un destino. Entonces, cada *sender* NDP toma la lista de rutas a un destino, la permuta aleatoriamente y luego envía paquetes en cada ruta en el orden obtenido. Una vez que ha enviado un paquete en cada ruta, permuta aleatoriamente la lista de rutas nuevamente y el proceso se repite. Esto distribuye los paquetes por igual en todas las rutas y evita la sincronización inadvertida entre dos *senders*. Este balanceo de carga es importante para lograr un *delay* muy bajo. Si se usan colas de paquetes de datos muy pequeñas para los *switches* (solo ocho paquetes), experimentos muestran que este esquema simple puede aumentar la capacidad máxima de la red hasta en un 10% sobre una elección de ruta aleatoria *per-packet*.

Los autores afirman que dependiendo de si la red es conmutada L2 o L3, se pueden utilizar rutas de conmutación de etiquetas (del estilo de MPLS), o direcciones de destino para elegir una ruta.

2.2 Protocolo de transporte

NDP utiliza un protocolo de transporte *receiver-driven* diseñado específicamente para aprovechar el *multipath forwarding*, el recorte de paquetes y las colas de *switches* pequeñas. El objetivo en cada paso es, en primer lugar, minimizar la demora para transferencias cortas y luego maximizar el rendimiento para transferencias más grandes.

A diferencia de Internet, en un *datacenter*, las velocidades de los enlaces y los RTT de referencia son mucho más homogéneos y muchas veces se pueden conocer de antemano. Además, la utilización de la red suele ser relativamente baja. Entonces, en lugar de probar la conexión (como TCP), para minimizar la demora el protocolo debe asumir que habrá suficiente capacidad para enviar una ventana completa de datos en el primer RTT. Sin embargo, si resulta que la capacidad es insuficiente, se perderán paquetes, y es en esta situación donde el recorte de paquetes de NDP resulta útil. Los *headers* de los paquetes recortados que llegan al receptor consumen poco ancho de banda del cuello de botella que causó la pérdida, pero informan al receptor con precisión qué paquetes se enviaron. Además, el orden de llegada de los paquetes no es importante cuando se trata de inferir pérdidas, ya que al llegar un *header* al receptor el mismo sabrá exactamente que paquete no llegó, sin importar si llegó antes o después de lo esperado. La cola de prioridad de los *switches* asegura que los *headers* y los paquetes de control lleguen rápidamente; de hecho lo suficientemente rápido como para que una retransmisión llegue antes de que la cola sobrecargada se vacíe. Con los *headers*, el receptor puede conocer la demanda exacta, ya que sabe que *senders* quieren enviarle qué datos. Luego de que los *senders* envían la primer ventana de datos a toda velocidad, paran de enviar, y en adelante el protocolo es *receiver-driven*. Un receptor NDP solicita paquetes de los *senders* utilizando paquetes PULL, estableciendo el ritmo del envío de modo que los paquetes de datos que llegan al receptor lo hagan a una velocidad que coincida con la velocidad del enlace del receptor, siendo estas solicitudes retransmisiones o datos nuevos. Entonces, el protocolo funciona de la siguiente manera:

- El *sender* envía una ventana entera de datos, sin esperar respuesta. Observar que todos estos paquetes llevan número de secuencia.
- Por cada *header* que llega al receptor, el mismo envía inmediatamente

un paquete NACK, indicándole al *sender* que prepare el paquete para retransmisión (sin enviarlo, ya que esto lo hará cuando el receptor se lo indique). Un *header* en este contexto hace referencia a los paquetes a los cuales se les recortó el *payload*, es decir que cuando llega un *header* al receptor es porque no llegó el paquete y debe indicárselo al emisor.

- Por cada paquete de datos que llega al receptor, el mismo envía inmediatamente un paquete ACK, indicando que el paquete llegó correctamente y que se puede liberar el *buffer* del lado del emisor.

- Por cada paquete o *header* que llega al receptor, el mismo agrega un paquete PULL a su *pull queue*, que cuando el receptor lo indique, será enviado al emisor. Un receptor posee solo una *pull queue* y es compartida con todas las conexiones de las cuales es receptor.

- El receptor envía paquetes PULL de la *pull queue* de cada interfaz, con un ritmo tal que los paquetes de datos obtenidos del *sender* lleguen a la velocidad de enlace del receptor, es decir que estos paquetes PULL son los que marcarán la velocidad de envío del emisor, luego del primer RTT. Los paquetes de PULL de diferentes conexiones reciben un servicio justo de forma predeterminada o con una prioridad estricta cuando un flujo tiene mayor prioridad. Además, un paquete PULL contiene el *id* de la conexión, y un contador *pull* por cada *sender*, que se incrementa por cada paquete PULL enviado a ese *sender*

- Por cada paquete PULL que llega al *sender*, el emisor enviará tantos paquetes de datos como incremente el contador *pull* del paquete. Los paquetes en cola para retransmisión son enviados primero, seguidos de los datos nuevos.

- Cuando el emisor se queda sin datos para enviar, marca el último paquete. Cuando este llega al receptor, el mismo remueve todos los paquetes PULL para ese *sender* de su *pull queue* para evitar enviar paquetes PULL innecesarios.

Gracias al recorte de paquetes, es muy raro que un paquete se pierda realmente. Si esto sucede generalmente se debe a la corrupción. Como los ACK y NACK se envían inmediatamente y con prioridad y todas las colas de *switches* son pequeñas,

el emisor puede saber muy rápidamente si un paquete se perdió realmente. Los paquetes PULL desempeñan una función similar al *timer* ACK de TCP, pero generalmente se encuentran separados de los ACK para poder establecer su ritmo sin afectar el mecanismo de tiempo de espera de retransmisión. Por ejemplo, en un escenario de *incast* grande, los paquetes PULL pueden pasar un tiempo largo en la cola de *pull* del receptor antes de ser enviados, pero no se quiere retrasar también los ACK, ya que hacerlo requiere ser mucho más conservador con los *timeouts* de retransmisión. Es posible ver una maquina de estados simplificada para el emisor en la figura 4 y una para el receptor en la figura 5.

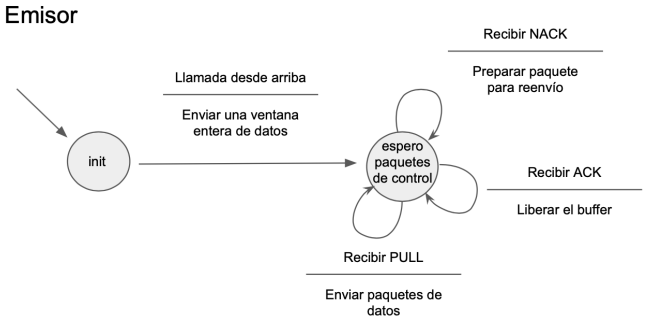


Figure 4: Máquina de estados simplificada para un emisor NDP

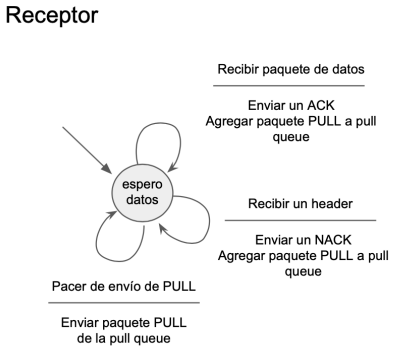


Figure 5: Máquina de estados simplificada para un receptor NDP

2.2.1 Reordenamiento

Debido al *per-packet multipath forwarding*, es normal que se reordenen tanto los paquetes de datos como los ACK, NACK y PULL. El diseño del protocolo básico es robusto en cuanto a reordenamiento, ya que no necesita hacer inferencias sobre la pérdida de los números de secuencia de otros paquetes. Sin embargo, aún es necesario tenerlo en cuenta. Aunque los paquetes PULL tienen prioridad en la cola, no se adelantan a los paquetes de datos, por lo que los paquetes PULL enviados en diferentes rutas muchas veces llegan desordenados, lo que aumenta la frecuencia de las retransmisiones. Para reducir esto, los paquetes PULL llevan un número de secuencia *pull*. El receptor tiene un espacio de secuencia *pull* separado para cada conexión, incrementándolo en uno por cada *pull* enviado. Al recibir un paquete PULL, el emisor transmite tantos paquetes de datos como aumente el número de secuencia de *pull*. Por ejemplo, si un paquete PULL se retrasa, el siguiente PULL enviado puede llegar primero a través de una ruta diferente y el emisor extraerá dos paquetes en lugar de uno. Esto reduce un poco la frecuencia de retransmisiones.

2.2.2 Primer RTT

Debe ser posible enviar datos NDP en el primer RTT. Es decir, que se debe poder enviar una ventana de datos, sin previo acuerdo (*handshake*). Esto agrega tres nuevos requisitos al protocolo:

- Ser robusto ante las solicitudes que falsifican las direcciones IP de origen. La suplantación de identidad se puede prevenir en el hipervisor o NIC, o usando VXLAN para *datacenters* de múltiples inquilinos, por lo que no es un problema mayor.

- Asegurar que ninguna conexión se procese dos veces sin querer, lo cual NDP resuelve manteniendo el estado de tiempo de espera tanto en el cliente como en el servidor. De este modo, cualquiera de los dos puede rechazar conexiones duplicadas. Como la vida útil máxima del segmento es inferior a 1 ms, la cantidad de estado adicional es bastante pequeña.

- Soportar el reordenamiento de los paquetes en el primer RTT. Esto se debe a que en el primer RTT se pueden enviar varios paquetes, los cuales

utilizarán distintas rutas, y por ende el primer paquete en llegar al receptor no es necesariamente el primer paquete de la conexión. Para ser robusto, cada paquete en el primer RTT lleva la *flag* SYN activada, junto con el desplazamiento de su número de secuencia del primer paquete en la conexión. Esto permite que el estado de conexión sea establecido por cualquier paquete que llegue primero al receptor.

2.2.3 Optimizaciones de robustez

Si la red se comporta correctamente, el protocolo planteado funciona muy bien. Sin embargo, a veces fallan los enlaces o *switches*, lo cual es detectado normalmente mediante un protocolo de enrutamiento. Los paquetes NDP se enrutan en el *sender*, por lo que los *hosts* NDP también necesitan recibir estas actualizaciones de enrutamiento para saber qué rutas evitar. Sin embargo, antes de que el protocolo de enrutamiento haya informado a los *hosts*, los paquetes que lleguen a un enlace fallido se perderán. También son posibles otras fallas más sutiles, como un enlace de 10Gb/s que pasa a 1Gb/s, lo que da como resultado una falla que no se detectará de inmediato mediante el enrutamiento. Para solucionar esto, los emisores NDP mantienen un marcador de rutas, realizando un seguimiento de la ruta que atraviesa cada paquete. Cuando llega un ACK o NACK, se incrementa el contador de ACK o NACK para la ruta por la que se envió el paquete de datos. Normalmente, en una topología de Clos que ejecuta NDP, todas las rutas deben tener una proporción muy similar de ACK a NACK. Sin embargo, si una falla ha causado asimetría, algunos enlaces tendrán un número excesivo de NACK. Cuando el emisor permuta la lista de rutas, elimina temporalmente los valores atípicos del conjunto de rutas.

La pérdida de paquetes casi no debería ocurrir. Un emisor NDP que retransmite un paquete perdido siempre lo reenvía por una ruta diferente. También se incrementa un contador de pérdidas de ruta cada vez que se pierde un paquete. Cualquier ruta que sea atípica con respecto a la pérdida de paquetes también se elimina temporalmente del conjunto de rutas. Estos mecanismos permiten que NDP sea robusto para redes donde las rutas no tienen un rendimiento similar, con una pérdida mínima de rendimiento.

2.2.4 Reenviar al emisor

El recorte de paquetes puede soportar grandes *incasts* sin necesidad de descartar ningún *header*. Sin embargo, los *incasts* extremadamente grandes pueden desbordar la cola *headers* y causar pérdidas. Los paquetes faltantes se reenviarían rápidamente, cuando expire el RTO (*retransmission timeouts*) del emisor. Sin embargo, a veces una transferencia completa se puede realizar con un solo paquete, y esa transferencia puede ser de alta prioridad, por ejemplo, puede ser rezagado de una solicitud anterior. Si dicho paquete se pierde, confiar en el RTO agrega demoras innecesarias. Como optimización, cuando la cola de *headers* se desborda, el *switch* puede intercambiar las direcciones del emisor y del receptor del *header* y reenviar el mismo al emisor. A continuación, el emisor podría reenviar el paquete correspondiente. Sin embargo, reenviar siempre podría causar un *eco* del *incast* original. NDP solo reenvía si no espera más paquetes PULL, es decir, no hay paquetes confirmados (ACK) o confirmados negativamente (NACK) para los cuales no llegó un PULL packet, o si todos los demás paquetes de la primera ventana también fueron reenviados al emisor. NDP también reenvía, si la mayoría de los paquetes fueron recientemente confirmados en lugar de confirmados negativamente (*NACKed*). Esto se debe a que muchas confirmaciones de paquetes para una ruta indican que la misma funciona bien, por lo que tiene sentido reenviar los paquetes en esta.

2.2.5 Control de congestión

NDP no realiza control de congestión en una topología de Clos, ya que es innecesario con la combinación correcta de modelo de servicio de red y protocolo de transporte. En términos generales, el control de la congestión de Internet cumple dos funciones: evita el colapso de la congestión y garantiza la equidad. NDP logra ambos sin tener un mecanismo de adaptación de ventana explícito.

Evitar colapso de la congestión: Los *switches* NDP evitan el colapso de la congestión de CP (*cut payload*) al garantizar que los paquetes de datos utilicen la mayor parte de un enlace. El colapso también puede ocurrir si los paquetes se descartan en el receptor. Debido al recorte de paquetes, los emisores NDP rara vez necesitan depender del RTO, por lo que el colapso debido a retransmisiones

innecesarias no es posible. El colapso podría ocurrir si una gran cantidad de paquetes se descarta cerca del receptor que ya ha desplazado otros paquetes antes en su ruta. Sin embargo, en una topología Clos con *per-packet multipath routing* de NDP, los paquetes casi nunca se recortan en los enlaces ascendentes de los *switches core* porque no es posible concentrar el tráfico allí. Cuando se recortan en los enlaces ascendentes, esto se debe a un desequilibrio de carga y es en esta situación donde el equilibrio de carga basado en el origen de NDP proporciona una ventaja sobre el ECMP *per-packet random* realizado en general por los *switches*. Incluso con una carga alta, los paquetes recortados en este lugar comprenden una pequeña parte del tráfico total. El recorte significativo solo ocurre realmente durante los *incasts*, ya que la mayoría de los paquetes se recortan en los enlaces de los *switches* ToR a los hosts, y algunos se recortan entre los *switches* de pod superior e inferior. Por lo tanto, los paquetes que son recortados por los *switches* ToR rara vez han desplazado paquetes antes en la topología.

Equidad: NDP logra una excelente equidad sin necesidad de mecanismos adicionales. Todos los flujos en competencia comienzan con la misma ventana, por lo que no hay necesidad de preocuparse por la convergencia. El punto principal cuando los flujos compiten es por la capacidad del receptor, y el receptor tiene una visión completa de lo que está sucediendo. La equidad del receptor se logra mediante el uso de un esquema de cola equitativa para los paquetes en la cola de PULL que pertenecen a diferentes conexiones. Finalmente, la injusticia deliberada es posible, porque el receptor conoce sus propias prioridades y puede atraer tráfico de alta prioridad con más frecuencia que el tráfico de baja prioridad.
Un caso de injusticia que no puede ser gestionado por el receptor es cuando un flujo a un receptor compite con un incast a otro receptor en el mismo *switch* ToR. NDP mitiga tal injusticia en un RTT porque, después de eso, el ritmo del emisor marcado por los paquetes PULL elimina la sobrecarga [12].

3 Simulador

En esta sección se presentará el simulador provisto junto con las distintas pruebas realizadas, y una guía sobre como configurar las mismas.

3.1 Build del simulador

En primer lugar, es necesario clonar el repositorio disponible en [13]. El simulador se encuentra disponible en el directorio `sim`, donde al ejecutar el comando `make` compilará los archivos fuente TCP, NDP, htsim y de red. Este código no es específico de *datacenter*. El resultado debería ser `libhtsim.a`. Para construir la herramienta de *parser* de salida (`parse_output`), se debe ejecutar el comando `make parse_output` en el directorio `sim`. Esto creará un ejecutable con el mismo nombre. Por último, se debe ejecutar el comando `make` en el directorio `datacenter`. Esto debería compilar todas las topologías de *datacenter* que admite `htsim`, así como crear muchos ejecutables (llamados `htsim_...`) que están destinados a ejecutar diferentes experimentos. Un ejecutable `htsim_X` resulta de un archivo fuente `main_X` correspondiente que configura y ejecuta el experimento [13].

3.2 Estructura general del simulador

El simulador es un programa escrito en lenguaje C++, que consta de un solo hilo de ejecución. Se disponen de varias topologías de *datacenter*, pero la mayoría de los ejemplos brindados utilizan la topología de *Fat Tree*, encontrada en el directorio *datacenter* en los archivos `fat_tree_topology.cpp` y `fat_tree_topology.h`. Todos los nodos, paquetes, *switches*, etc. son una clase (encontrados en el directorio `sim`) y se simula el *delay* esperado por cada *link*. Todos los eventos son manejados con una clase llamada `EventList` y se registran las métricas y datos necesarios con distintos *loggers* encontrados en `loggers.h`.

Resulta importante además, destacar algunas clases que aportan a la compresión general del simulador.

- Clases `NdpSrc` y `NdpSink`: Estas clases, encontradas en el archivo `ndp.cpp`, son el NDP *sender* y el NDP *receiver*, que se encargan de comenzar los flujos de paquetes y de recibirlos. Así como también se encargan de setear las rutas para los paquetes y permutarlas. Estas clases se encargan de todo el procesamiento de paquetes, tanto de recibirlos, como de recibir ACK's, NACK's, retransmisiones, etc.

- Clases `NdpPacket`, `NdpAck`, `NdpNack` y `NdpPull`: Son clases derivadas de la

clase `Packet`. En este simulador, un paquete es una clase con atributos, y existe una base de datos para re usar los objetos de los paquetes. Es decir, que cuando se utiliza el método `newpkt` (para crear un nuevo paquete), en realidad se consulta la base de datos para re utilizar paquetes ya creados que no se encuentran en uso. En particular, estos paquetes no cuentan con carga útil, y se diferencia si es un *header* o no a través de un atributo *is_header*. Por ende, mandar un paquete realmente implica liberar espacio en la base de datos del *sender*, junto con la asignación de métricas para conocer que paquetes fueron enviados.

- `FatTreeTopology`: En esta clase se crea la topología de *Fat Tree* como se mencionó anteriormente. Dentro de la misma se definen los *switches* y *pipes*, donde ambos se representan como vectores de C++, y la clase `switch` contiene atributos tales como un nombre y puertos.

- Escenarios: En el directorio `datacenter` se presentan los archivos *main*, encargados de representar los distintos escenarios, por ejemplo, *incast* en NDP (`main_ndp_incast.cpp`), permutación de tráfico para NDP, etc. y se dispone de un archivo para cada uno de los ejemplos y protocolos presentados.

3.3 Pruebas realizadas

Dentro del simulador, en el directorio `EXAMPLES` se pueden encontrar los distintos ejemplos presentados a lo largo del *paper*. Estos ejemplos utilizan los archivos mencionados en 3.2 para representar los distintos escenarios.

Un primer ejemplo (a parte del directorio `EXAMPLES`), se puede ejecutar con el siguiente comando en el directorio `datacenter`:
`./htsim_ndp_permutation -strat perm -nodes 16 -conns 16 -cwnd 30`
Este comando creará una topología *Fat Tree* que contiene 16 *servers* (`-nodes`), es decir $k = 4$, donde todos los enlaces son de 10 Gbps. Este ejemplo ejecuta una matriz de tráfico de permutación (`main_ndp_permutation.cpp`) que contiene 16 conexiones (`-conns`), donde cada *server* envía y recibe exactamente una conexión NDP de larga duración. La ventana inicial utilizada para los *senders* es de 30 (`-cwnd`). Finalmente, los paquetes se distribuyen a través de las rutas disponibles

utilizando la estrategia de permutación (`-strat`), es decir que cada fuente enviará paquetes en una permutación aleatoria de rutas utilizando *round-robin*. Una vez utilizados todos los caminos, la permutación se genera nuevamente. Esto asegura que no haya congestión de larga duración en ninguno de los puertos en el núcleo de la red [13].

Al correr el ejemplo se imprimen los caminos existentes como se ve en la figura 6.

```
Path:
queue(10000Mb/s,9000000bytes)SRC0->LS0
pipe(1us)
compqueue(10000Mb/s,72000bytes)LS0->US0 id=104 13762pkts 0hdrs 13757acks 0nacks 0stripped
pipe(1us)
compqueue(10000Mb/s,72000bytes)US0->CS0 id=197 20640pkts 0hdrs 20635acks 0nacks 0stripped
pipe(1us)
compqueue(10000Mb/s,72000bytes)CS0->US4 id=248 20638pkts 0hdrs 20633acks 0nacks 0stripped
pipe(1us)
compqueue(10000Mb/s,72000bytes)US4->LS_5 id=161 27517pkts 0hdrs 27509acks 0nacks 0stripped
pipe(1us)
compqueue(10000Mb/s,72000bytes)LS5->DST11 id=71 27515pkts 0hdrs 27509acks 0nacks 0stripped
pipe(1us)
```

Figure 6: Ejemplo de salida al imprimir los caminos

En particular, esto resulta de utilidad, ya que se pueden ver los caminos elegidos, y cuántos paquetes pasaron por cada link. En el ejemplo mostrado, `SRC0` representa el nodo 0, `LS0` el nodo *leaf* 0, `US0` el nodo *spine* 0, `CS0` el nodo *core* 0 y *DST11* el nodo 11. En la figura 7 se puede ver el camino indicado con cada nodo resaltados en amarillo.

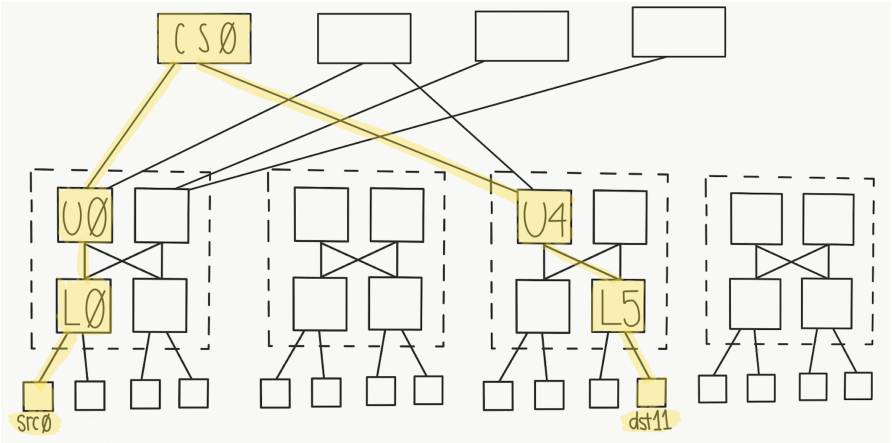


Figure 7: Ejemplo de ruta y nodos

Por defecto, todos los experimentos generarán resultados en un archivo llamado `logout.dat`. Para parsear estos resultados, se puede utilizar el siguiente comando:

```
../parse_output logout.dat -ndp -show
```

Que desplegará una salida similar a la siguiente:

```
9857.16 Mbps val 354 name ndp_sink_15_13(0)
9856.08 Mbps val 350 name ndp_sink_14_1(0)
9855.36 Mbps val 346 name ndp_sink_13_9(0)
9856.80 Mbps val 342 name ndp_sink_12_14(0)
...
Mean of lower 10pc (1 entries) is 1231875000.000000
total mean 1232055000.000000 mean2 0.000000
```

Donde cada línea despliega el *throughput* de cada conexión, seguida de detalles de la misma, incluyendo su identificador y su nombre. El nombre también incluye el *sender* y el *receiver* de la conexión, por ejemplo en la primer línea el *sender* es el nodo 15 y el *receiver* el nodo 13. La última línea de la salida despliega la media del 10% inferior de los flujos en bytes por segundo (el 10% son la cantidad de entradas *entries* indicadas entre paréntesis), así como la media total, también en bytes por segundo. Si se ejecuta el mismo ejemplo varias veces, se puede observar que siempre se eligen las mismas conexiones. Con el fin de que esta elección sea aleatoria, es necesario modificar el archivo `connection_matrix.cpp`. En particular en este archivo se elige el destino en el siguiente código (a partir de la línea 37):

```
for (int src = 0; src < N; src++) {
    do {
      pos = rand()%perm_tmp.size();
    } while(src==perm_tmp[pos]&&perm_tmp.size()>1);
    dest = perm_tmp[pos];
    assert(src!=dest);
    perm_tmp.erase(perm_tmp.begin()+pos);
    to[src] = dest; // src to dest
}
```

Como se puede ver, para que la función `rand` logre que cada vez sea un destino aleatorio, es necesario agregar una semilla. Entonces se agrega la línea:

```
srand (time(NULL));
```

antes de asignar la variable `pos`. Observar que agregando una semilla basada en el tiempo, se generará cada vez que se ejecute un resultado diferente. Si en cambio se desea generar un resultado aleatorio pero que sea posible de replicar, se puede optar por una semilla con valor fijo. También es posible cambiar el tamaño del *rack* en el mismo archivo, en el siguiente código:

```
void ConnectionMatrix::setPermutation(int conn){
  setPermutation(conn,1); // 1 is the rack size
}
```

A continuación se mostrarán algunos de los ejemplos encontrados en el repositorio, junto con una explicación de como modificarlos y ejecutarlos. Para correr estos ejemplos es necesario contar con *Python* y *gnuplot*.

3.3.1 Incast y tráfico cercano

En este ejemplo, encontrado en la ruta `sim/EXAMPLES/collateral`, se analiza como el tráfico *incast*[2] afecta el tráfico cercano. En particular, en el experimento se ejecuta un flujo de larga duración al host 1, luego se inicia un *incast* de 64 a 1 de corta duración al host 2. Este ejemplo se corresponde con la Figura 19 del paper [12].

Para correr el ejemplo por defecto, que tiene 432 nodos (k = 12) y 64 conexiones, basta con ejecutar el *script* `run.sh` encontrado en el directorio. Este ejemplo es probado con DCTCP [2], DCQCN [17] (aproximado con DCTCP con *lossless Ethernet*) y NDP, donde el *script* general ejecuta un *script* para cada una de estas opciones. En particular, resulta interesante mostrar el ejemplo de NDP, que es posible correrlo de la siguiente manera:

```
../../datacenter/htsim_ndp_incast_collateral -strat perm
-nodes 432 -conns 64 -q 8 -cwnd 15
```

[2]Comunicación *many-to-one* (muchos *senders* y un solo *receiver*)

Como se puede ver, se utiliza el archivo generado por el *main* de `ndp_incast_collateral` y con los parámetros indicados. Los resultados obtenidos, son los mismos que en el paper, obteniendo las gráficas presentadas en las figuras 8, 9 y 10.

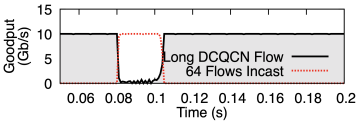


Figure 8: Gráfica 432 nodos *collateral* DCQCN

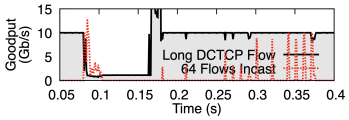


Figure 9: Gráfica 432 nodos *collateral* DCTCP

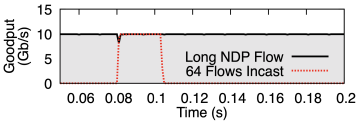


Figure 10: Gráfica 432 nodos *collateral* NDP

Con DCQCN, se evita la pérdida y los flujos de *incast* terminan rápidamente, pero PFC afecta el flujo largo. Con DCTCP, tanto el flujo de larga duración como el *incast* tardan algún tiempo en recuperarse. Con NDP, el *incast* provoca *trimming* (recortado de paquetes) durante el primer RTT. El flujo largo sufre una pequeña caída en el rendimiento de menos de 1 ms debido a esta ráfaga inicial. Después del primer RTT, el receptor marca el ritmo de los paquetes *incast* restantes y el flujo largo se recupera para volver a obtener el rendimiento completo. Se puede ver que NDP es quien tiene el mejor rendimiento.

Para poder realizar un ejemplo personalizado, es necesario cambiar los parámetros en cada *script*. Si se cambian la cantidad de nodos y conexiones, se debe correr el ejemplo deseado para cada protocolo, por ejemplo para 128 nodos y 127 conexiones en NDP se debe ejecutar (en la carpeta `datacenter`):

```
./htsim_ndp_incast_collateral -strat perm -nodes 128 -conns 127 -q 8
                              -cwnd 15
```

luego realizar el parseo, y obtener el identificador del *long sink*, encontrado en la primer línea de la salida del parseo. Luego, es necesario reemplazar en las líneas donde se ejecuta *grep* de un número, en cada *script*, por el nuevo obtenido, para así indicar cuál es el flujo de larga duración en cada protocolo. También es necesario cambiar en cada *main* de *incast_collateral* de NDP, DCQN y DCTCP (`main_ndp_incast_collateral.cpp`, `main_dctcp_incast_collateral.cpp`, etc.) el código

```
if (no_of_nodes <= 128)
    long_src_no = 5;  //a source not on the same LS as dst 1
else if (no_of_nodes == 432)
    long_src_no = 7;  //a source not on the same LS as dst 1
```

agregando un if para que acepte la cantidad de nodos elegidos.

Siguiendo las modificaciones presentadas anteriormente, se corrió un nuevo ejemplo con 128 nodos (k = 8) y 127 conexiones. Se obtuvieron las gráficas presentadas en las figuras 11, 12 y 13.

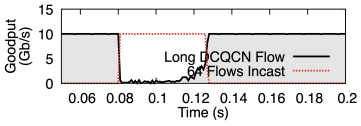


Figure 11: Gráfica 128 nodos *collateral* DCQCN

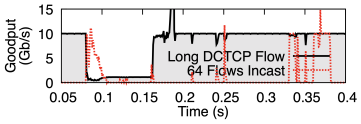


Figure 12: Gráfica 128 nodos *collateral* DCTCP

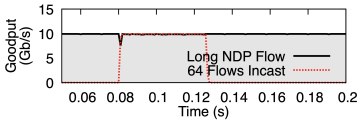


Figure 13: Gráfica 128 nodos *collateral* NDP

Como se puede ver, el comportamiento se mantiene prácticamente igual y NDP sigue obteniendo el mejor rendimiento.

3.3.2 Falla en un link

En este ejemplo, encontrado en la ruta `sim/EXAMPLES/failure`, se genera una matriz de tráfico de permutación completa, al igual que en `sim/EXAMPLES/permutation`, donde cada *host* envía a otro *host* y cada *host* recibe de otro *host*, excepto que en `failure` falla uno de los enlaces entre un *switch* de pod superior y un *switch core*; el mismo pasa de de 10 Gb/s a 1 Gb/s. Este tipo de falla suele ser más problemática que una falla total, ya que las fallas totales pueden detectarse mediante enrutamiento y evitarse. La matriz de tráfico de permutación carga el *core* a su capacidad máxima, por lo que el enlace del core fallido estará muy sobrecargado, lo que resultará en una alta congestión para los flujos que atraviesan este enlace y una gran pérdida para un protocolo como NDP que distribuye paquetes de manera uniforme en todas las rutas. Este ejemplo se corresponde con la Figura 22 del paper [12].

Para correr el ejemplo por defecto, que tiene 128 nodos (k = 8) y 128 conexiones, es suficiente con ejecutar el *script* `run.sh` encontrado en el directorio. A diferencia del *script* del ejemplo presentado anteriormente, en este *script* se encuentran los comandos directamente escritos, en lugar de ejecutar múltiples *scripts*. Este ejemplo es probado con DCTCP [2], MPTCP [15] y NDP. En particular, resulta interesante mostrar el ejemplo de NDP, que es posible correrlo de la siguiente manera:

```
../../datacenter/htsim_ndp_permutation_fail -o ndp_perm -strat perm
-conns 128 -nodes 128 -cwnd 23 -q 8 > debugout_ndp_perm
```

Como se puede ver, se utiliza el archivo generado por el *main* de `ndp_permutation_fail` y con los parámetros indicados. Para poder realizar un ejemplo personalizado, es necesario cambiar los parámetros deseados en el *script* `run.sh`. En particular, se creó un ejemplo con 16 nodos (k = 4) y 16 conexiones, donde más flujos atraviesan el *link* con fallas. Se obtuvieron gráficas para los dos ejemplos, donde la figura 14 muestra el ejemplo por defecto y la figura 15 el ejemplo personalizado.

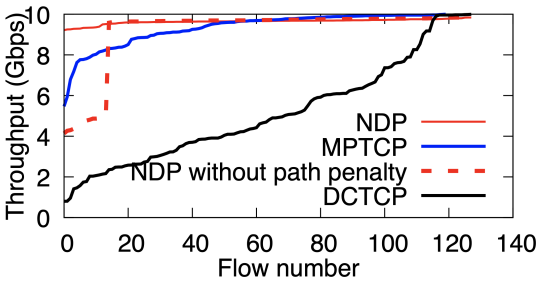


Figure 14: Gráfica 128 nodos *failure*

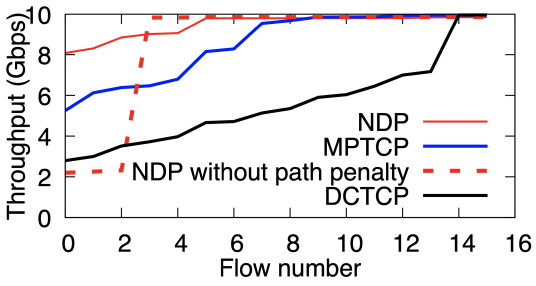


Figure 15: Gráfica 16 nodos *failure*

Donde tanto NDP como MPTCP manejan bien la falla, ya que mantienen información de congestión por ruta y dirigen el tráfico evitando las rutas congestionadas. También se puede ver cómo funciona NDP sin contar con este mecanismo de penalización de rutas, marcado con las líneas punteadas en rojo.

3.3.3 Otros ejemplos

En la carpeta de ejemplos, es posible también encontrar los siguientes ejemplos:

- `in_and_out`, donde se busca estudiar como NDP comparte el ancho de banda, presentando un ejemplo donde dos *senders* envían al mismo *host*, pero uno de ellos no puede llenar la mitad del *link*
- `incast_scalling`, donde se prueba un tráfico de *incast* aumentando (y variando) la cantidad de nodos
- `incast_short_flows`, proporciona un ejemplo donde una aplicación de *frontend* en un servidor realiza solicitudes simultáneas a otros servidores *backend*, que responden inmediatamente generando un tráfico de *incast*. La gráfica obtenida varía el número de servidores. Es posible cambiar el tiempo de comienzo, editando el archivo `main_ndp_incast_shortflows.cpp`, cambiando el valor de `extrastarttime`
- `permutation`, donde cada *host* envía a otro *host* y cada *host* recibe de otro *host*, como se mencionó anteriormente

Todos estos ejemplos pueden ser modificados de manera análoga a como se modificaron los ejemplos presentados anteriormente, y es posible encontrar una breve explicación de los ejemplos en [5]. Se probaron distintas modificaciones para cada ejemplo, obteniendo resultados similares, que confirman los presentados en el *paper*.

4 Implementación en Linux

En esta sección se presentará la implementación en Linux provista, junto con la aplicación de ejemplo brindada, y las configuraciones necesarias para poder ejecutar la misma.

En primer lugar, se tiene la implementación en Linux de NDP, que hace uso de la herramienta DPDK, explicada en profundidad en la sección 4.1. Luego, se brinda una aplicación ejemplo del tipo `ping-pong` donde se tienen uno o más clientes, que envían datos a un servidor y el mismo contesta con otros datos.

Esta aplicación ejemplo constará entonces de dos o más nodos NDP, por lo que se necesita contar mínimamente con dos equipos. En particular, dentro de la carpeta donde se encuentra la implementación (`host_impl`), se pueden encontrar las siguientes tres carpetas:

- `core`: Donde se encuentra la implementación de NDP y por ende donde se hace uso de DPDK. El proceso principal de NDP media el acceso a NIC y mantiene la cola de *pull*, ya que todas las conexiones NDP deben compartir esto. También maneja retransmisiones rápidas causadas por NACK

- `lib`: Donde se encuentra el programa *main* de la aplicación ejemplo (`f_ndp_ping_pong.c`), y a su vez se presenta una API de NDP para aplicaciones. Interactúa con el *core* a través de memoria compartida, comandos como *connect* y *listen* a través de un *ring buffer* de comunicaciones, y datos para cada *socket* activo a través de tres *ring buffer*: RX (*reception*), TX (*transmission*) y RTX (*retransmission*), y un grupo de *buffers* compartido. La biblioteca también maneja retransmisiones de paquetes debido a *timeouts*

- `common`: Donde se encuentran archivos comunes a `core` y `lib`, por ejemplo para *logging*

4.1 DPDK

DPDK (Kit de desarrollo de plano de datos) es una solución de software *Open Source* de plano de datos optimizada, desarrollada por Intel para sus procesadores de múltiples núcleos. El propósito es proveer a los programadores un framework simple y completo para el rápido procesamiento de paquetes, centrado en mejorar el rendimiento del procesamiento de paquetes. De esta manera, los usuarios pueden crear prototipos o agregar su propio protocolo al *stack*, según sus necesidades [3]. En resumen, DPDK es una librería de espacio de usuario, que se encarga de proveer funciones al programador que le permiten interceptar paquetes antes de pasar por el *kernel* y procesarlos según sus necesidades, a su vez acelerando el procesamiento.

DPDK implementa un modelo de *run to completion* para el procesamiento de paquetes, donde todos los recursos deben asignarse antes de llamar a las

aplicaciones del plano de datos, ejecutándose como unidades de ejecución en núcleos de procesamiento lógico, llamados **lcores**. Se accede a todos los dispositivos mediante sondeo (*polling*), para no utilizar interrupciones por la sobrecarga de rendimiento que imponen [10]. En la figura 16, se puede ver a la izquierda de la imagen el procesamiento de paquetes tradicional y a la derecha el procesamiento de paquetes utilizando DPDK. En este último, se puede ver que todas las interacciones con la tarjeta de red se realizan a través de controladores y bibliotecas especiales [16].

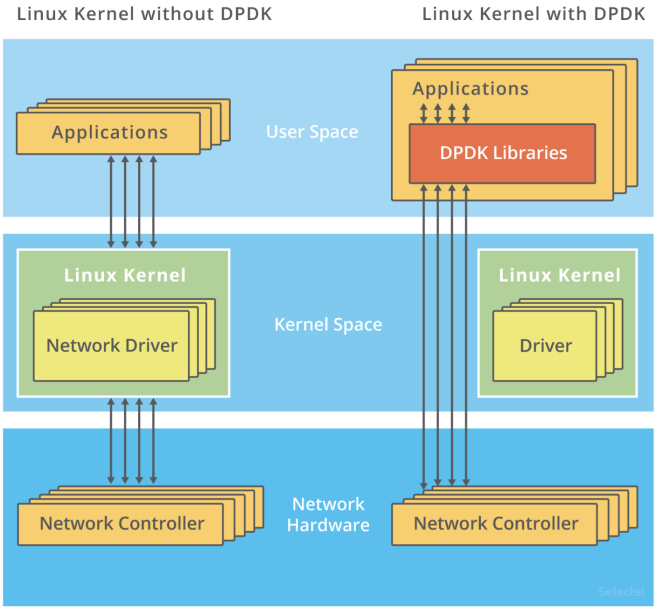


Figure 16: Procesamiento tradicional de paquetes vs Procesamiento con DPDK [16]

Cuando se ejecutan aplicaciones con gran uso de memoria en Linux, el tamaño de página predeterminada de 4KB de tamaño causará muchos fallos de página y fallos de TLB, lo que afecta en gran medida el rendimiento de la aplicación. El uso de páginas grandes puede reducir significativamente la frecuencia de fallos de TLB y fallos de página, reduciendo el consumo innecesario de acceso a la memoria. DPDK utiliza *huge pages*[3] para resolver este problema. Por lo tanto, es de suma importancia configurar *huge pages* a la hora de configurar DPDK.

[3]https://docs.openstack.org/nova/latest/admin/huge-pages.html

4.1.1 Environment Abstraction Layer

Environment Abstraction Layer (EAL) es responsable de obtener acceso a recursos de bajo nivel, como *hardware* y espacio de memoria. Proporciona una interfaz genérica que oculta los detalles del entorno de las aplicaciones y bibliotecas. Es responsabilidad de la rutina de inicialización decidir cómo asignar estos recursos (espacio de memoria, dispositivos, temporizadores, consolas, etc.) [9]. En particular, para el interés de este trabajo, resulta importante comprender el parámetro `-c <core mask>`, el cual setea la máscara de bits hexadecimal de los *cores* para correr la aplicación [8].

4.2 Implementación

4.2.1 Implementación NDP

Para la implementación de NDP, se utiliza DPDK, como se mencionó anteriormente. En el archivo `core/init_dpdk.c` se puede ver la inicialización de DPDK, donde en primer lugar se setean todos los valores necesarios. En particular se setea la máscara que utilizará DPDK para los `lcores`, de manera que la misma indica cuáles serán utilizados, por ejemplo si se desean utilizar los *core* cero y uno, se debe utilizar la máscara 011 (hexadecimal que indica los lugares cero y uno). Luego, se utilizan todas las funciones de DPDK, como inicializar el EAL, configurar los dispositivos, hacer *setup* de las colas, etc.

En el archivo `main.c` de esta carpeta, se encuentra el *loop* principal de un nodo NDP, que busca nuevos registros de aplicaciones, ejecuta comandos desde instancias de biblioteca, envía el primer RTT de paquetes desde el *TX ring* de nuevas conexiones y maneja los paquetes entrantes. Los paquetes de datos, ACK y NACK que llegan se colocan en el *RX ring* del *socket* correspondiente. Los PULL que llegan hacen que los paquetes de datos se envíen desde el *RTX ring o TX ring* del *socket*, dando prioridad a *RTX*. Los paquetes de datos (o *headers*) que llegan hacen que los paquetes PULL se agreguen a la cola de *pull*. Un *thread* de *pull queue* independiente, que se ejecuta en su propio núcleo de CPU, retira estos paquetes PULL uno por uno en el momento apropiado y los envía. Los paquetes de datos también pueden desencadenar la inicialización de un nuevo

socket si el bit SYN está prendido y se llamó a *listen* previamente. Los NACK se pasan a la biblioteca para evitar tiempos de espera espurios, pero el núcleo NDP también agrega el índice de *buffer* correspondiente al *RTX ring* del *socket*, lo que permite una retransmisión muy rápida.

Luego, el *header ndp* de un paquete se encuentra definido en el archivo `common/ndp_header.h`. El formato del cabezal es el siguiente:

```
struct ndp_header{
    ndp_header_flags_t flags;
    ndp_checksum_t checksum;

    ndp_port_number_t src_port;
    ndp_port_number_t dst_port;

    union {
        //data, fin, ack & nack segments
        ndp_sequence_number_t sequence_number;
        //for pull segments
        ndp_pull_number_t pull_number;
    };

    union {
        //for data segments
            ndp_pacer_number_t pacer_number_echo;
            //for pull segments
            ndp_pacer_number_t pacer_number;
            //for ack segments
            ndp_recv_window_t recv_window;
    };
} __attribute__((packed));
```

El mismo consta de 20 bytes, y contiene para todos los tipos de paquete NDP los campos *flags* (una para cada tipo de paquete), *checksum*, puerto origen y puerto destino. Luego, para los paquetes de datos, FIN, ACK y NACK tiene un número

de secuencia, en cambio para los paquetes de tipo PULL tiene un *pull_number*. Luego, para los segmentos de datos contiene un *pacer_number_echo*, para los segmentos PULL un *pacer_number* y para los segmentos ACK la ventana del receptor. El campo correspondiente a las *flags* será de gran utilidad al momento de la inspección del tráfico, ya que permitirá identificar los paquetes.

Resulta importante destacar algunos aspectos de la implementación. En primer lugar, en el archivo `/lib/helpers.c` se encuentran implementadas las funciones de `ndp_send_all` y `ndp_recv_all`, encargadas de enviar y recibir a través de los *buffers* apropiados, donde la función de *receive* también se encarga de enviar los paquetes ACK correspondientes. Luego en `/lib/accept.c`, `/lib/listen.c`, `/lib/connect.c` y `/lib/close.c` se encuentran implementadas las funciones de `ndp_accept`, `ndp_listen`, `ndp_connect` y `ndp_close` respectivamente del *socket* NDP. Dado que NDP es un protocolo *zero RTT*, el comando de *connect* de la biblioteca solo informa al núcleo NDP sobre un nuevo *socket* activo. La conexión se establecerá cuando se envíen los datos. El comando *listen* informa al núcleo NDP sobre un nuevo socket pasivo, pero también reserva varios *sockets* para cualquier conexión entrante, ya que la falta de un protocolo de acuerdo de conexión inicial significa que se debe estar listo para aceptar paquetes sin previo acuerdo.

Para poder utilizar la implementación de NDP, es necesario editar el archivo `core/init_dpdk.c`. En las líneas 59 y 60, donde se asignan la cantidad de *rx queues* y *tx queues*, es necesario cambiar la cantidad 182 por 1 [4]. Esto hace que también deba ser cambiado el parámetro de la línea 63, donde se setea el id de la cola de *tx* de *pull*, el cual debe ser 0, ya que hay solo una cola. Luego, es necesario setear la variable de entorno `NDP_IMPL_DIR` a la ruta donde se encuentra la implementación del stack NDP, por ejemplo con el comando `export RTE_SDK=/host_impl`. Por último, se debe ir a este directorio y ejecutar el comando `make` [11].

[4]Este número fue obtenido al mirar ejemplos de otras aplicaciones DPDK, donde siempre de utilizan una *rx queue* y una *tx queue*. Queda como trabajo a futuro, investigar por qué no es posible usar un número mayor a uno (por ejemplo, 128), y cómo impacta esto.

4.2.2 Implementación aplicación ping-pong

Si se observa el archivo `lib/main.c` se puede ver que en el programa principal se ejecuta la función `f_ndp_ping_pong`, implementada en el archivo `f_ndp_ping_pong.c`. En esta función se puede ver como se tiene un gran *if-else* donde se chequea si *id* (pasado por parámetro) es múltiplo de dos o no, lo cual indica si se está ejecutando el *server* o un cliente.

En el caso del *id* múltiplo de dos, se tiene un cliente, donde se llena un buffer (`buf1`) con información, y luego se realiza un *loop* de 10 iteraciones, donde en cada una se realiza el *connect* con el *server* (`ndp_connect`), el *send* (`ndp_send_all`) de la información, el *receive* (`ndp_recv_all`) de lo que envía el *server* y el *close* (`ndp_close`) del *socket*. Si se desea cambiar el mensaje que se envía al *server*, por ejemplo, enviar la palabra "ping" como se mostrará en las siguientes secciones, alcanza con comentar las líneas 101 a 104 y agregar la siguiente línea

```
sprintf(buf1, "%s", "ping")
```

Si a su vez se desea ver el mensaje recibido en la consola, alcanza con agregar la línea

```
printf("rcvd %s \n", buf2)
```

luego de haber hecho el *receive*. Además, se imprimen en cada iteración el índice de iteración y una métrica de tiempo.

En el caso del *id* no múltiplo de dos, se tiene un server, donde se realiza el *listen* (`ndp_listen`), y luego se tiene un *loop* (mientras cierta variable sea verdadera), y se ejecuta por cada cliente un *accept* (`ndp_accept`), luego un *receive* (`ndp_recv_all`) de lo que envía el *cliente*, un *send*(`ndp_send_all`) de la información deseada y el *close* (`ndp_close`) del *socket*. Si se desea cambiar el mensaje que se envía al *cliente*, por ejemplo, enviar la palabra "pong" como se mostrará en las siguientes secciones, alcanza con comentar las líneas 155 a 158 y agregar la siguiente linea

```
sprintf(buf2, "%s", "pong")
```

Si a su vez se desea ver el mensaje recibido en la consola, alcanza con agregar la línea

```
printf("rcvd %s \n", buf1)
```

luego de haber hecho el *receive*. Además se imprime en cada paquete recibido, la cantidad de bytes, el número de mensaje recibido (con un solo cliente este número será hasta diez) y una suma.

Si se desean enviar más o menos bytes, se deben cambiar los *defines* de las variables `CLIENT_SENDS` y `SERVER_SENDS` encontradas al principio del archivo `f_ndp_ping_pong`. Luego de varias pruebas con diferentes números y el análisis de distintas capturas de tráfico, se llegó a las siguientes conclusiones:

- El tamaño mínimo de segmento serán 6 *bytes*. Es decir que si se envían por ejemplo 3 *bytes*, se rellenará hasta 6 *bytes* y se enviará un paquete con tamaño 60 *bytes* (14 *bytes* por el *header* Ethernet, 20 *bytes* por el *header* IP, 20 *bytes* por el *header* NDP y 6 *bytes* de *payload*)
- La unidad máxima de transferencia (MTU) es de 1514 bytes. Esto permite que el tamaño máximo de segmento (MSS) sea 1460 bytes. Si se intenta enviar un mensaje de mayor tamaño, entonces el mismo se fragmentará en los paquetes necesarios (respetando el tamaño mínimo y máximo de segmento).

Para poder utilizar la aplicación de ejemplo, es necesario realizar algunos cambios, ya que varios datos se encuentran fijados. En los archivos:

- `lib/f_ndp_ping_pong.c`: Es necesario asignar la dirección IP del servidor a la variable `destination_server` (líneas 27, 29 y 30). Se espera que las direcciones IP sean 10.0.0.<nroMaquina>.
- `common/mac_db.c`: Es necesario agregar las direcciones MAC del servidor y el cliente a las estructuras `gaina_mac_info`, `cocos_mac_info` y `cam_mac_info`. El primer número de máquina es 1, y corresponde con la entrada 0 de la tabla de direcciones MAC, la máquina 2 corresponde con la entrada 1, etc. Por ejemplo, si la dirección MAC del servidor es `e0:3f:49:ad:82:48` y la MAC del cliente es `bc:ee:7b:76:8f:1a` se tiene:

```
{0 , 1, { MAC6(e0, 3f, 49, ad, 82, 48) } },
{1 , 1, { MAC6(bc, ee, 7b, 76, 8f, 1a) } },
```

Donde el primer entero representa el número de máquina y el segundo representa la cantidad de direcciones MAC.

4.2.3 Configuración de DPDK

En primer lugar, es necesario descargar la versión 17.05.2 de DPDK[5], ya que es la versión de testeo más reciente de NDP. En las siguientes secciones se referenciará a esta carpeta como `/dpdk`. Luego, DPDK necesita "apoderarse" de la interfaz por la cual recibirá los paquetes, de manera que no sean procesados por el *kernel*. Para ello, se utiliza el comando `ifconfig`, donde se obtienen la dirección IP (será de utilidad en el código de NDP conocer este valor, como se mencionó anteriormente) y nombre de la interfaz que deberá utilizar DPDK, y se debe dar de baja la misma utilizando el comando:

```
$ sudo ifconfig <nombreInterfaz> down
```

Una vez descomprimida la carpeta `/dpdk`, se ejecuta el *script* `dpdk-setup.sh`, encontrado dentro de la carpeta `/dpdk/usertools`. Este *script* debe ser ejecutado con privilegios de *root*. Se ingresan las siguientes opciones:

- `[12] x86_64-native-linuxapp-gcc`: Crea la librería Intel DPDK x86_64-native-linuxapp-gcc
- `[15] Insert IGB UIO module`: Carga el módulo de *kernel* Intel DPDK UIO
- `[18] Setup hugepage mappings for non-NUMA systems`: Configura *hugepages*. En este caso el número a ingresar será 1024, lo cual reservará 1024 *hugepages* de 2MB.
- `[21] Bind Ethernet/Crypto device to IGB UIO module`: Realiza el *bind* del dispositivo *Ethernet* al módulo de *kernel* Intel DPDK UIO. Se debe ingresar el nombre de la interfaz que se dio de baja anteriormente.
- `[32] Exit Script`: Se termina la configuración de DPDK.

Observar que es posible devolver el control de la interfaz al *kernel* utilizando el mismo *script*.

[5]Disponible en: https://fast.dpdk.org/rel/dpdk-17.05.2.tar.xz

Luego, se setea la variable de entorno `RTE_TARGET` con el comando
`export RTE_TARGET=x86_64-native-linuxapp-gcc`
y la variable de entorno `RTE_SDK` a la ruta donde se encuentra DPDK, por ejemplo con el comando `export RTE_SDK=/dpdk`.

Una vez terminada la configuración, se puede utilizar la herramienta `ethtool` de DPDK, para verificar que quedó configurado correctamente. Esta herramienta se encuentra en la ruta `/dpdk/examples/ethtool` y se puede ejecutar con el comando `./ethtool-app/${RTE_TARGET}/ethtool`. Existen múltiples comandos útiles en esta herramienta, disponibles en [7]. Entre ellos, si se ejecuta el comando `link`, se obtendrá el estado de los puertos, donde el puerto recién configurado debería tener estado *Up* y si se ejecuta el comando `macaddr <numeroPuerto>` se puede ver la dirección MAC del puerto configurado, que debería coincidir con la dirección MAC de la interfaz que se dio de baja. Observar que en la carpeta `examples` existen otros ejemplos de aplicaciones DPDK que pueden ser de utilidad o interés.

4.2.4 Correr la aplicación de ejemplo

En las siguientes secciones, se configuró un servidor con dirección IP `10.0.0.1` y dirección MAC `e0:3f:49:ad:82:48`, y un cliente con dirección IP `10.0.0.2` y dirección MAC `bc:ee:7b:76:8f:1a`. Se intentó montar este escenario utilizando máquinas virtuales, pero no fue posible debido a complicaciones con DPDK. Por esta razón se decidió utilizar el escenario con máquinas físicas. Se puede ver la configuración de este escenario en la figura 17. Además, se encuentra disponible en [4] el código con los cambios requeridos mencionados anteriormente, suponiendo las direcciones IP mencionadas.

Figure 17: Configuración del escenario para correr la aplicación de ejemplo

Para correr la aplicación ejemplo, es necesario ejecutar el comando
`build/core <local_ip_address>`
para el servidor y el cliente, en la carpeta `core`, ya que el *core* no tiene manera de saber en que máquina está siendo ejecutado. En el caso del servidor, se debería ver una salida similar a la presentada en la figura 18.

```
root@cookie:/home/cookie/NDP/host_impl# core/build/core 10.0.0.1
removed previous global_comm_ring shm file
estimating tsc_hz...
tsc_hz = 3499995903.790173
cycles_per_pull = 4304
EAL: Detected 8 lcore(s)
EAL: No free hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: PCI device 0000:00:19.0 on NUMA socket -1
EAL:   probe driver: 8086:153b net_e1000_em
rte_lcore_count() = 2
rte_eth_dev_count() = 1
using 1 rx queue(s) and 1 tx queue(s)
original default txconf txq_flags = 00000000
pullq lcore thread launched
hi!
```

Figure 18: Salida del *build* del servidor

En el caso del cliente, se debería ver una salida similar a la presentada en la figura 19.

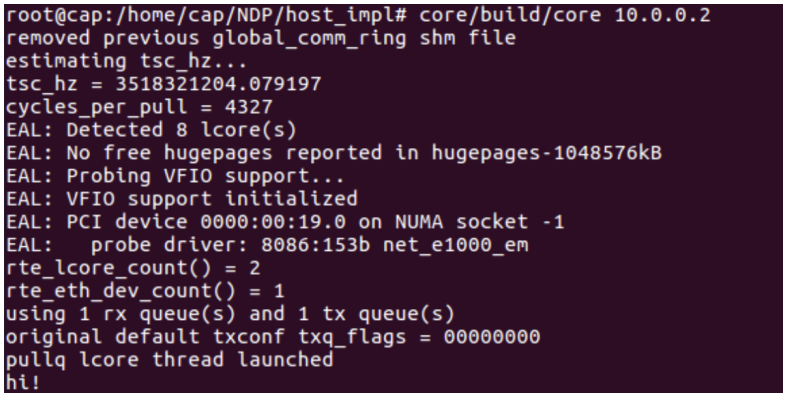


Figure 19: Salida del *build* del cliente

Luego, en otra consola, en la carpeta `lib` (observar que es necesario hacerlo primero en el servidor):

- Para el caso del cliente: ejecutar el comando `./lib 0`, o utilizando cualquier número par
- Para el caso del servidor: ejecutar el comando `./lib 1`, o utilizando cualquier número impar

Si se configura el cliente para que envíe la palabra "ping" y el servidor responda con la palabra "pong", la salida del servidor (luego de ejecutado el cliente) debería ser similar a la de la figura 20 y la del cliente (con el *server* ejecutado previamente) a la de la figura 21.

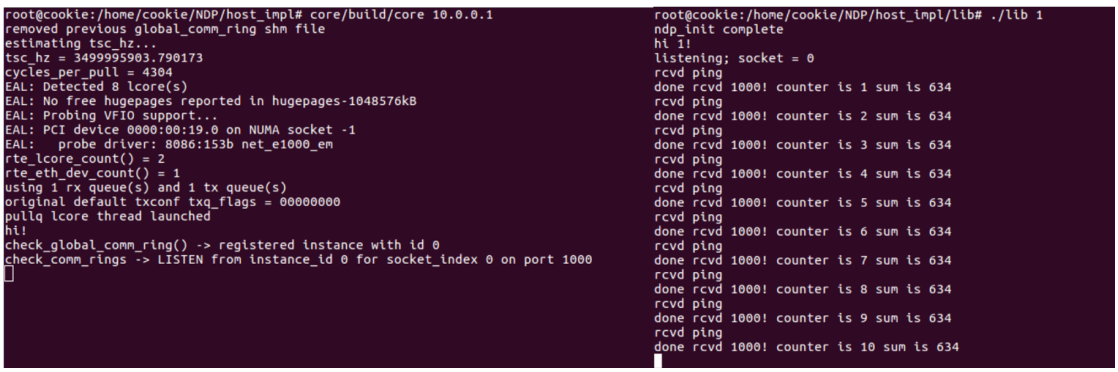


Figure 20: Salida del servidor luego de la ejecución de un cliente

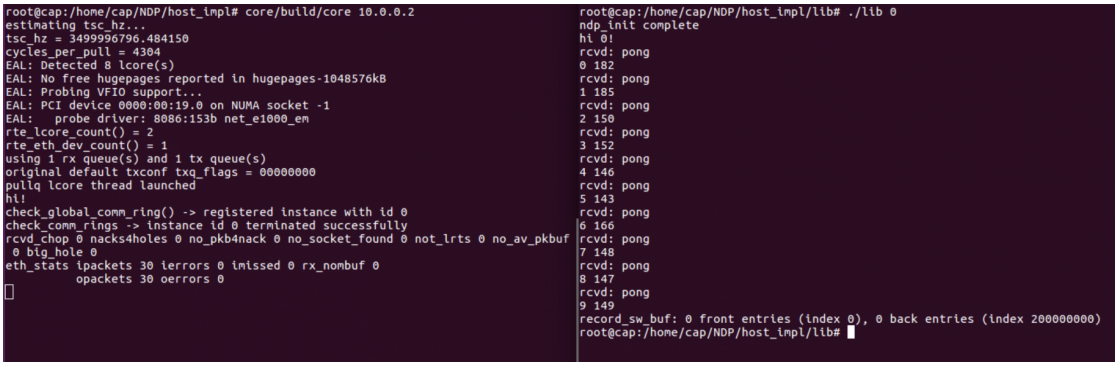


Figure 21: Salida del cliente luego de su ejecución

En el caso del *build* del servidor, se puede observar que se registra un *listen* para una instancia con *id* 0. Cuando se corre el servidor, primero se ve un mensaje que dice *hi 1!* donde el número indica el *id*, y luego se puede ver que se encuentra escuchando conexiones en el *socket* 0. Luego de correr el cliente, se despliega el contenido de cada paquete recibido, la cantidad de bytes del *payload*, el número de mensaje recibido y una suma. En el caso del *build* del cliente, se puede observar que se registra una instancia con *id* 0. Se puede ver cuando se corre el cliente, un mensaje que dice *hi 0!* donde el número indica el *id* y luego se despliega el contenido de cada paquete recibido del *server*, el índice de iteración y una métrica de tiempo. Cuando se termina de ejecutar el cliente, se pueden ver estadísticas en la consola donde se realizó el *build*.

4.3 Pruebas realizadas

En esta sección se presentan las pruebas realizadas sobre la implementación de Linux, utilizando la aplicación de ejemplo. En primer lugar, fue necesario probar que los enviados y recibidos en cada host NDP coincidan con los paquetes especificados en el protocolo. Como DPDK se apodera de la interfaz por la cual intercepta los paquetes y la misma no es visible desde el *kernel*, para *debug* y *sniffing* fue necesario agregar otro *host* en el mismo dominio de *broadcast*, lo cual se hizo con un *switch* configurado para que tenga el comportamiento de un *hub*. Para las pruebas mencionadas a continuación, se utilizó un cliente y un servidor

con las direcciones mencionadas en 4.2.4. Se puede ver la configuración de este escenario en la figura 22.

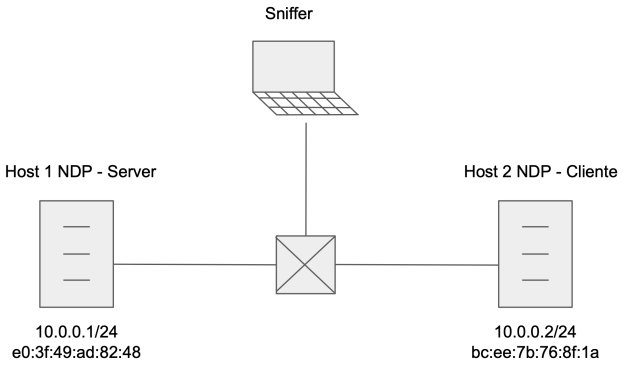
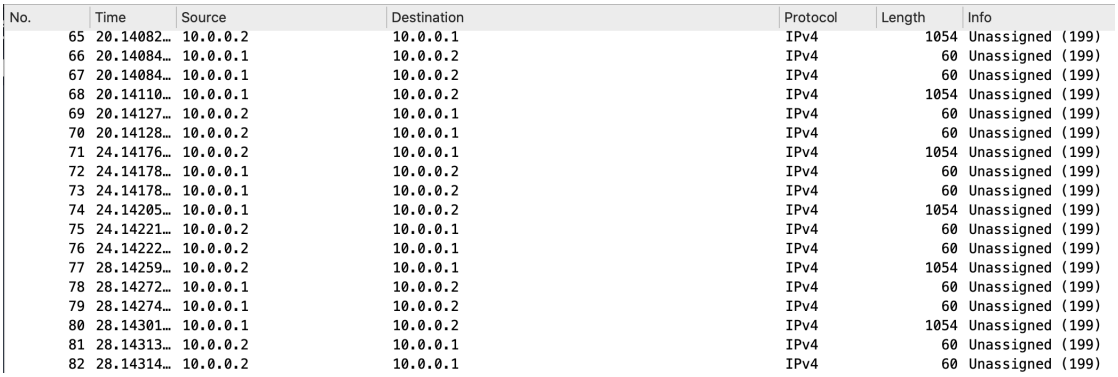


Figure 22: Configuración del escenario para pruebas

4.3.1 Prueba con payload por defecto

Como primer prueba, se utilizó el tamaño de *payload* seteado por defecto, de 1000 *bytes*. Se realizó una captura de tráfico y es posible observar una parte de ésta en la figura 23 utilizando *Wireshark*. Se observa en primer lugar como el cliente (10.0.0.2) envía un paquete al servidor (10.0.0.1) de tamaño 1054 *bytes*, y el mismo responde con dos paquetes de 60 *bytes*. Luego, el servidor envía al cliente un paquete de 1054 *bytes*, y este responde con dos paquetes de 60 *bytes*, y así sucesivamente. Al intentar mapear estos resultados con el protocolo, se podría asumir que el paquete de 1054 *bytes* es el mensaje enviado, y los otros dos paquetes más pequeños son los paquetes PULL y ACK. Notar que se observan paquetes IPv4, lo cual tiene sentido ya que *Wireshark* no puede detectar de que protocolo se trata.



No.	Time	Source	Destination	Protocol	Length	Info
65	20.14082…	10.0.0.2	10.0.0.1	IPv4	1054	Unassigned (199)
66	20.14084…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
67	20.14084…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
68	20.14110…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
69	20.14127…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
70	20.14128…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
71	24.14176…	10.0.0.2	10.0.0.1	IPv4	1054	Unassigned (199)
72	24.14178…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
73	24.14178…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
74	24.14205…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
75	24.14221…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
76	24.14222…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
77	28.14259…	10.0.0.2	10.0.0.1	IPv4	1054	Unassigned (199)
78	28.14272…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
79	28.14274…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
80	28.14301…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
81	28.14313…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
82	28.14314…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)

Figure 23: Captura de tráfico entre cliente y servidor, prueba con payload por defecto

Para poder confirmar esto, es necesario observar el header del paquete NDP. Al inspeccionar el paquete 65, del cual se despliega una parte en la figura 24, se puede observar en primer lugar el cabezal Ethernet y luego el cabezal IPv4. Luego, se observa la primer parte del *payload* del paquete IP, marcado en amarillo, donde los primeros 20 bytes representan el cabezal NDP. Si se compara el cabezal, con la estructura mostrada en 4.2.1, se puede identificar, marcado con rosado las *flags*, marcado con azul el *checksum*, con verde el puerto origen, con celeste el puerto destino, con violeta el número de secuencia o *pull_number* en caso de los paquetes PULL y con rojo el *pacer_number_echo*, *pacer_number* o *recv_window*, según el tipo de paquete. Como se mencionó en 4.2.1, el campo *flags* permite identificar el tipo de paquete y sus diferentes valores pueden ser consultados en el archivo `common/ndp_header.h`. Observando que los equipos utilizados adoptan el sistema *little-endian*, el primer *byte* de las *flags* (`01`) representa la *flag* de datos, y el segundo *byte* (`01`) representa la *flag* de SYN que debe estar activada en todos los paquetes del primer RTT, tal como dice el protocolo. Luego, se puede ver el *payload* NDP, que consta de números de 0 a 1000 (en este caso se envió el *payload* por defecto y no "ping" y "pong" como en la sección 4.2.4).

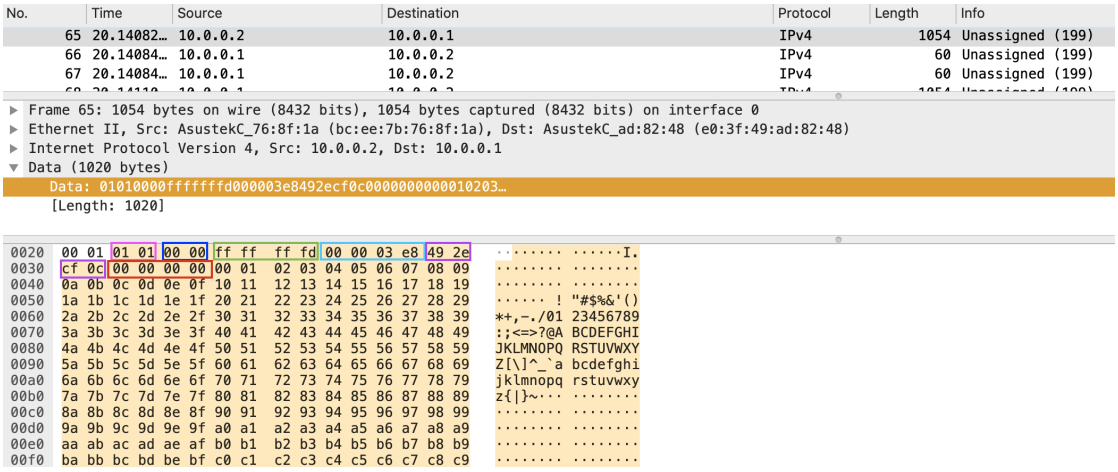


Figure 24: Contenido paquete NDP de datos

Luego, si se analiza el paquete 66, mostrado en la figura 25, se puede ver también el cabezal Ethternet e IPv4. Luego, con los mismos colores que el paquete 65, se puede ver cada sector del cabezal NDP. En particular, el campo *flags* tiene el valor `04 00`, lo cual indica la *flag* de paquete PULL (recordar que se utiliza *little-endian*), lo cual coincide con el protocolo, ya que el receptor envía un paquete PULL por cada paquete procesado. Observar además que se rellenan los últimos 6 *bytes* con ceros, para completar el mínimo tamaño de segmento, y estos *bytes* forman parte del *padding* de Ethernet.

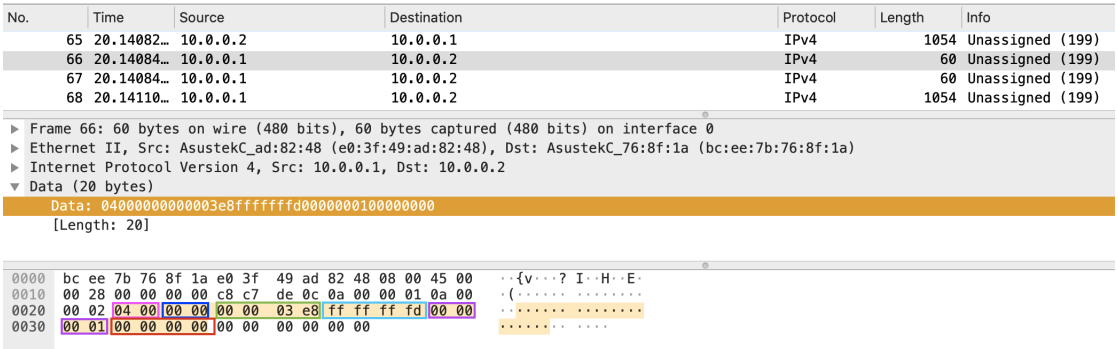


Figure 25: Contenido paquete PULL

Por último, si se analiza el paquete 67, mostrado en la figura 26, se puede ver

también el cabezal Ethternet e IPv4. Luego, con los mismos colores que el paquete 65, se puede ver cada sector del cabezal NDP. En particular, el campo *flags* tiene el valor `02 00`, lo cual indica la *flag* de paquete ACK, lo cual coincide con el protocolo, ya que el receptor envía un paquete ACK por cada paquete entero recibido. Observar que también se rellenan los últimos 6 *bytes* con ceros.

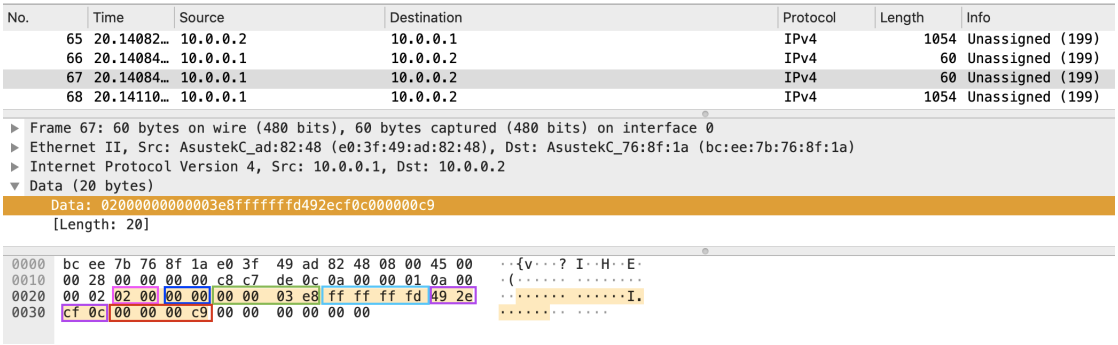
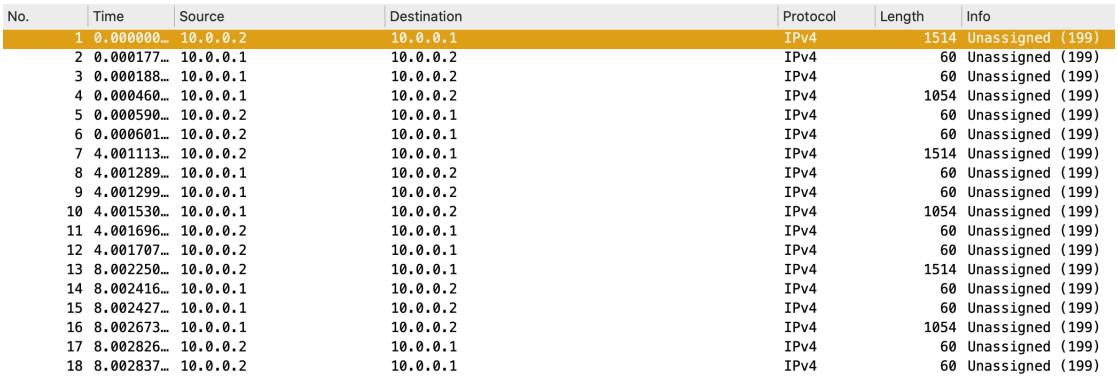


Figure 26: Contenido paquete ACK

4.3.2 Prueba con máximo tamaño de segmento

Como segunda prueba, se utilizó el tamaño de *payload* de los paquetes enviados por el cliente de 1460 *bytes*, que representa el MSS. Se realizó una captura de tráfico y es posible observar una parte de ésta en la figura 27 utilizando *Wireshark*. Se observa que los paquetes enviados por el cliente tienen tamaño 1514 *bytes*, que es el MTU. Si se observan los cabezales de cada paquete se puede confirmar que el comportamiento es el mismo que en la prueba anterior.

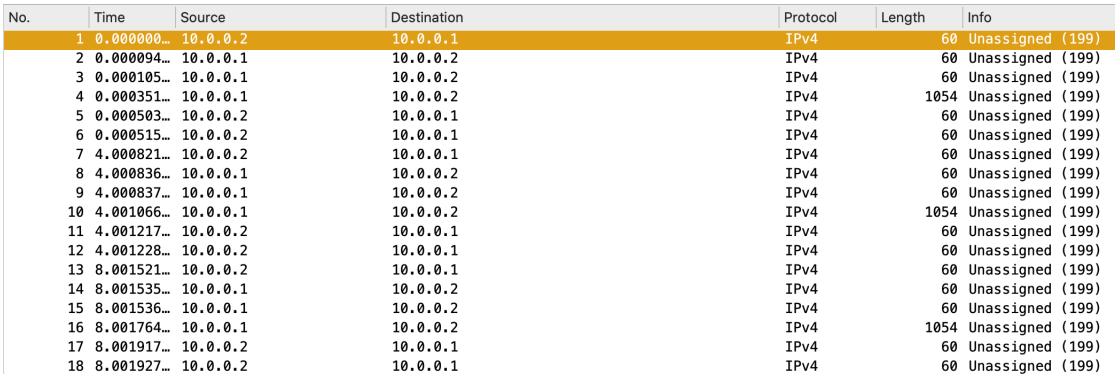


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000…	10.0.0.2	10.0.0.1	IPv4	1514	Unassigned (199)
2	0.000177…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
3	0.000188…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
4	0.000460…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
5	0.000590…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
6	0.000601…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
7	4.001113…	10.0.0.2	10.0.0.1	IPv4	1514	Unassigned (199)
8	4.001289…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
9	4.001299…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
10	4.001530…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
11	4.001696…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
12	4.001707…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
13	8.002250…	10.0.0.2	10.0.0.1	IPv4	1514	Unassigned (199)
14	8.002416…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
15	8.002427…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
16	8.002673…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
17	8.002826…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
18	8.002837…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)

Figure 27: Captura de tráfico entre cliente y servidor, prueba con máximo tamaño de segmento

4.3.3 Prueba con mínimo tamaño de segmento

Como tercer prueba, se utilizó el tamaño de *payload* de los paquetes enviados por el cliente de 4 *bytes*, que es menor al mínimo tamaño de segmento. Se realizó una captura de tráfico y es posible observar una parte de ésta en la figura 28 utilizando *Wireshark*. Se observa que los paquetes enviados por el cliente tienen tamaño 60 *bytes*, ya que se rellenan los dos *bytes* faltantes para el mínimo tamaño de segmento con ceros.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
2	0.000094…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
3	0.000105…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
4	0.000351…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
5	0.000503…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
6	0.000515…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
7	4.000821…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
8	4.000836…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
9	4.000837…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
10	4.001066…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
11	4.001217…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
12	4.001228…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
13	8.001521…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
14	8.001535…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
15	8.001536…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
16	8.001764…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
17	8.001917…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
18	8.001927…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)

Figure 28: Captura de tráfico entre cliente y servidor, prueba con mínimo tamaño de segmento

Si se inspecciona el primer paquete, mostrado en la figura 29, se pueden ver las

flags (marcadas en rosado), que confirman que es un paquete de datos (con la *flag* SYN prendida). También se puede observar, marcado en marrón, que se rellena con dos *bytes* en el *padding* de la trama Ethernet. Esto confirma que el mínimo tamaño de segmento son 6 *bytes*.

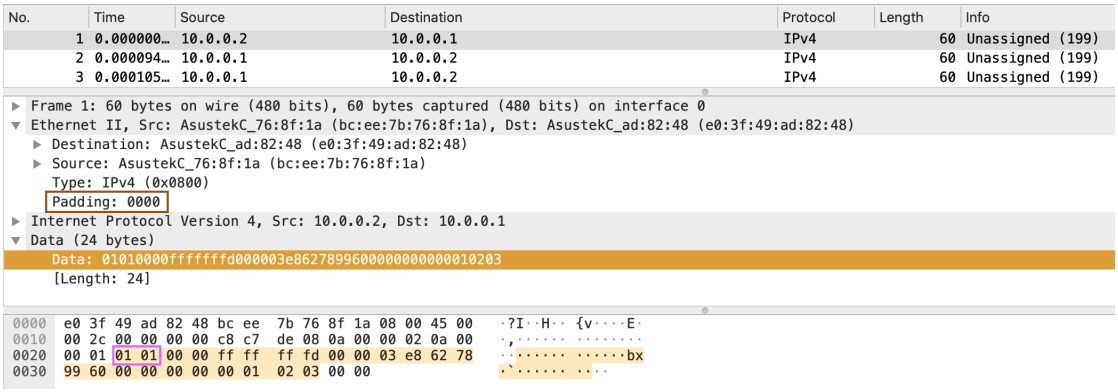


Figure 29: Contenido paquete con *payload* menor al mínimo tamaño de segmento

4.3.4 Prueba de fragmentación

Como cuarta prueba, se utilizó el tamaño de *payload* de los paquetes enviados por el cliente de 1500 *bytes*, que es mayor al MSS y por ende debería causar fragmentación. Se realizó una captura de tráfico y es posible observar una parte de ésta en la figura 30 utilizando *Wireshark*. Se observa que el cliente envía dos paquetes, uno de tamaño 1514 (MTU), de los cuales 1460 *bytes* son de *payload* y otro de tamaño 94, de los cuales 40 *bytes* son de *payload*, los cuales suman los 1500 *bytes* de *payload* enviados. En consecuencia, se observan cuatro paquetes de 60 *bytes*, enviados del servidor al cliente, que representan los dos paquetes ACK y dos paquetes PULL correspondientes a los dos paquetes recibidos.

No.	Time	Source	Destination	Protocol	Length	Info
5	4.002501…	10.0.0.2	10.0.0.1	IPv4	1514	Unassigned (199)
6	4.002518…	10.0.0.2	10.0.0.1	IPv4	94	Unassigned (199)
7	4.002574…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
8	4.002584…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
9	4.002586…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
10	4.002588…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
11	4.002838…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
12	4.002975…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
13	4.002978…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
14	8.003545…	10.0.0.2	10.0.0.1	IPv4	1514	Unassigned (199)
15	8.003562…	10.0.0.2	10.0.0.1	IPv4	94	Unassigned (199)
16	8.003718…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
17	8.003730…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
18	8.003732…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
19	8.003734…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
20	8.003987…	10.0.0.1	10.0.0.2	IPv4	1054	Unassigned (199)
21	8.004140…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
22	8.004152…	10.0.0.2	10.0.0.1	IPv4	60	Unassigned (199)
23	12.00477…	10.0.0.2	10.0.0.1	IPv4	1514	Unassigned (199)
24	12.00479…	10.0.0.2	10.0.0.1	IPv4	94	Unassigned (199)
25	12.00481…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
26	12.00481…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
27	12.00482…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)
28	12.00482…	10.0.0.1	10.0.0.2	IPv4	60	Unassigned (199)

Figure 30: Captura de tráfico entre cliente y servidor, prueba de fragmentación

5 Conclusiones

En este trabajo se estudió una arquitectura de transporte de *datacenter* que busca baja latencia y alta *performance*, logrando tiempos casi óptimos para transferencias cortas y alto rendimiento de flujo en una amplia gama de escenarios, incluido el *incast*.

En primer lugar, se logró entender la importancia de los Sistemas Ciber Físicos, junto con la popularidad, uso y algunas topologías de los *datacenters*. Esto además ayudó a comprender la motivación para crear este nuevo protocolo, como otras soluciones no lo lograban, y que requerimientos eran necesarios para poder cumplir los objetivos. En particular, la mayoría de los *datacenters* modernos proveen muy alta capacidad y baja latencia, pero rara vez los protocolos de transporte proveen la misma performance, por ende surgió la necesidad de idear un nuevo protocolo que cumpla con estas demandas.

Fue posible comprender el diseño y funcionamiento del protocolo en profundidad, comprendiendo los aspectos más importantes del protocolo, como que es un protocolo *zero RTT*, es decir que no cuenta con un *handshake* inicial para comenzar a enviar datos, que es un protocolo de comienzo rápido, es decir que no prueba por el ancho de banda, sino que envía la primer ventana de datos a toda velocidad, y reacciona de manera adecuada cuando no se dispone del ancho de banda necesario, que utiliza *per-packet ECMP* en el origen y es capaz de establecer la conexión sin importar cual es el primer paquete de la ventana inicial en llegar al receptor. A su vez, los *switches* NDP tienen un modelo de servicio especial, utilizando la técnica CP (*cut payload*), evitando la incertidumbre sobre el resultado de un paquete, dándole al receptor un panorama completo de que se le envió y evitando la sobrecarga del *switch*. El enrutamiento se realiza en los emisores NDP, ya que esto tiene mejores resultados que hacerlo en los *switches*, donde cada *sender* permuta aleatoriamente las rutas a un destino y envía un paquete en cada una de estas, logrando un balanceo de carga para lograr un *delay* bajo. También cuenta con un método de penalización de rutas, para poder evitar links fallidos. Por último, es muy importante mencionar, que luego del primer RTT, es un protocolo *receiver-driven*, es decir que luego del primer RTT, con la ayuda

de los paquetes PULL, es el receptor quien marcará el ritmo del envío de paquetes.

Luego, fue posible utilizar el simulador y comprender como se simularon las distintas partes del protocolo. En particular, es un único hilo de ejecución que simula envío de paquetes (simulando los *delays*) y sin carga útil. Se comprendieron las clases involucradas y se logró comparar los resultados obtenidos con los presentados en el *paper*, confirmándolos. Se pudo plantear en profundidad dos de los ejemplos brindados, junto con sus resultados y cómo modificarlos, donde se obtuvieron los resultados esperados. Además, se pudo configurar el resto de los ejemplos brindados, presentando una explicación de como realizarlo y confirmando nuevamente los resultados planteados.

Se pudo comprender la utilidad de la librería DPDK, que tiene alto valor para este tipo de implementaciones, y puede tener muchos usos. También fue posible configurar DPDK, y conocer distintas aplicaciones de ejemplo brindadas. Se logró comprender la estructura general de la implementación de NDP en Linux, comprendiendo más en profundidad las clases involucradas y como realizar los cambios necesarios para su funcionamiento. En particular, se pudo comprender el *header* de un paquete NDP que resulta de gran valor y se concluyó el MTU, MSS y mínimo tamaño de segmento para esta implementación. Luego, se analizó la aplicación de ejemplo brindada, entendiendo su funcionamiento, como cambiar el tamaño y contenido del *payload* a enviar deseado, y como configurar las direcciones de los *hosts* involucrados. Una vez configurados cliente y servidor, se pudo correr la aplicación ejemplo, desplegando los mensajes deseados y métricas. Por último, se realizaron pruebas sobre la implementación en Linux, realizando capturas de tráfico y analizando los paquetes, para confirmar que la implementación se comporta como dice el protocolo.

Como trabajo a futuro, existen varias líneas. En primer lugar, sería de valor probar el funcionamiento de la implementación con varios clientes, y logrando cambiar el formato de las direcciones IP. Además sería de interés poder utilizar la implementación del *stack* con otro sistema operativo, u otra versión de DPDK, como lograr replicar los escenarios planteados con máquinas virtuales. También sería de interés poder crear otra aplicación NDP, utilizando la API brindada. Por

último, restaría investigar las implementaciones de *switches* brindadas y poder probar el protocolo con todas sus funcionalidades en un ambiente adecuado.

Como conclusiones generales, se logró comprender en profundidad un protocolo de gran utilidad, verificando los resultados planteados y dejando distintas líneas de trabajo a futuro.

References

[1] Radhakisan Baheti and H. Gill. Cyber-physical systems. 2011.

[2] S. Bensley, D. Thaler, P. Balasubramanian, L. Eggert, and G. Judd. Data center tcp (dctcp): Tcp congestion control for data centers. RFC 8257, RFC Editor, October 2017.

[3] Bi, Hao and Wang, Zhao-Hun. Dpdk-based improvement of packet forwarding. *ITM Web Conf.*, 7:01009, 2016.

[4] Belén Brandino. Ndp end-host implementation. Available at `https://gitlab.fing.edu.uy/maria.belen.brandino/ndp-tscf/-/tree/master/host_impl` (09/11/2020).

[5] Belén Brandino. Ndp simulator examples. Available at `https://gitlab.fing.edu.uy/maria.belen.brandino/ndp-tscf/-/wikis/NDP-Simulator-Examples` (09/11/2020).

[6] M. Chinnici, D. D. Chiara, and A. Quintiliani. Data center, a cyber-physical system: Improving energy efficiency through the power management. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 269–272, 2017.

[7] doc.dpdk.org. 4. ethtool sample application. Available at `https://doc.dpdk.org/guides/sample_app_ug/ethtool.html`.

[8] doc.dpdk.org. Eal parameters. Available at `https://doc.dpdk.org/guides/linux_gsg/linux_eal_parameters.html`.

[9] doc.dpdk.org. Environment abstraction layer. Available at `http://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html#environment-abstraction-layer`.

[10] doc.dpdk.org. Overview. Available at `https://doc.dpdk.org/guides-16.04/prog_guide/overview.html`.

[11] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, W. Andrew Moore, Gianni Antichi, and Marcin Wójcik. Ndp end-host implementation. Available at `https://github.com/nets-cs-pub-ro/NDP/blob/master/host_impl/README.md` (26/11/2020).

[12] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, W. Andrew Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. *SIGCOMM*, pages 29–42, 2017.

[13] Mark Handley, Costin Raiciu, Vladimir Olteanu, Alexandru Agache, and Marcin Wójcik. Ndp. Available at `https://github.com/nets-cs-pub-ro/NDP` (11/09/2020).

[14] Deep Medhi and Karthik Ramasamy. Chapter 12 - routing and traffic engineering in data center networks. In Deep Medhi and Karthik Ramasamy, editors, *Network Routing (Second Edition)*, The Morgan Kaufmann Series in Networking, pages 396 – 422. Morgan Kaufmann, Boston, second edition edition, 2018.

[15] M. Scharf and A. Ford. Multipath tcp (mptcp) application interface considerations. RFC 6897, RFC Editor, March 2013.

[16] Andrej Yemelianov. Introduction to dpdk: Architecture and principles. Available at `https://blog.selectel.com/introduction-dpdk-architecture-principles/)`.

[17] et. al. Yibo Zhu. *SIGCOMM '15 : proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication : August 17-21, 2015, London, United Kingdom*. Association for Computing Machinery, New York, New York, 2015.