UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería Informática

Aplicación de técnicas de alto rendimiento al software de comunicaciones DDS
(Applying high performance techniques for DDS middleware)

Realizado por
Alejandro Fuster López

Tutorizado por
Jesús Martínez Cruz
Juan Adrián Romero Garcés

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre de 2021

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

APLICACIÓN DE TÉCNICAS DE ALTO RENDIMIENTO AL SOFTWARE DE COMUNICACIONES DDS

APPLYING HIGH PERFORMANCE TECHNIQUES FOR DDS MIDDLEWARE

Realizado por
Alejandro Fuster López

Tutorizado por
Jesús Martínez Cruz
Juan Adrián Romero Garcés

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2021

Agradecimientos

A mis tutores, Jesús y Adrián, por toda la confianza depositada en mí, darme la oportunidad de realizar este trabajo y todos los esfuerzos realizados en guiarme.

A mi familia, por todo el apoyo incondicional durante todo este trayecto y todo su cariño que siempre me anima.

A mi pareja, por ayudarme a desconectar durante todos estos años de carrera y entenderme siempre.

A mis amigos, por estar ahí desde que éramos unos críos y ser como hermanos.

Abstract

Nowadays there are a large number of applications that require high data throughput, and the Ethernet standard has been constantly improving to provide higher transfer rates in a data-driven digital world. However, the protocol stack of operating systems has not evolved so much, so these advances are not fully exploited due to the context switches produced by the network interface card (NIC) drivers. In this final thesis, we will study new high-performance software techniques to try to alleviate this bottleneck present in operating systems and we will try to apply some techniques to an implementation of the DDS (Data Distribution Service) middleware to study the possible improvement of the use of these techniques in a data-centric distributed application.

Keywords: High performance, Protocol Stack, Bottleneck, Middleware, DDS

Resumen

En la actualidad, hay muchas aplicaciones que se intercambian grandes cantidades de datos continuamente, y el estándar Ethernet no ha parado de mejorar para proveer mayores tasas de transferencias en un mundo movido por los datos. Sin embargo, en la pila de protocolos de los sistemas operativos no ha habido una gran evolución, por lo que estos avances no son aprovechados en su totalidad debido a los cambios de contexto que producen los controladores de las tarjetas de red. En el presente trabajo fin de grado, se estudiarán nuevas técnicas de software de alto rendimiento para tratar de paliar este cuello de botella presente en los sistemas operativos y se procurará aplicar alguna técnica a una implementación del *software* de comunicaciones DDS (*Data Distribution Service*) para estudiar la posible mejoría de la utilización de estas técnicas en una aplicación distribuida centrada en los datos.

Palabras clave: Alto Rendimiento, Pila de protocolos, Cuello de botella, Software de comunicaciones, DDS

Índice

1 Introducción

1.1 Motivación

Hoy en día vivimos en un mundo interconectado donde la transmisión de datos tiene una gran importancia. La evolución del estándar Ethernet es un gran ejemplo de cómo ha surgido la necesidad constante de tener una mayor velocidad, propiciando un gran avance en las tarjetas de red desde la aparición del primer estándar IEEE 802.3 en 1985, capaz de ofrecer una velocidad de 10 Mbit/s (megabits por segundo). Tal y como se puede apreciar en la figura 1, la velocidad máxima declarada en las distintas enmiendas realizadas al estándar Ethernet por el *Institute of Electrical and Electronics Engineers* (IEEE) en los últimos años no ha parado de crecer, presentando un crecimiento gigantesco desde la aparición del *Fast Ethernet* en 1995.

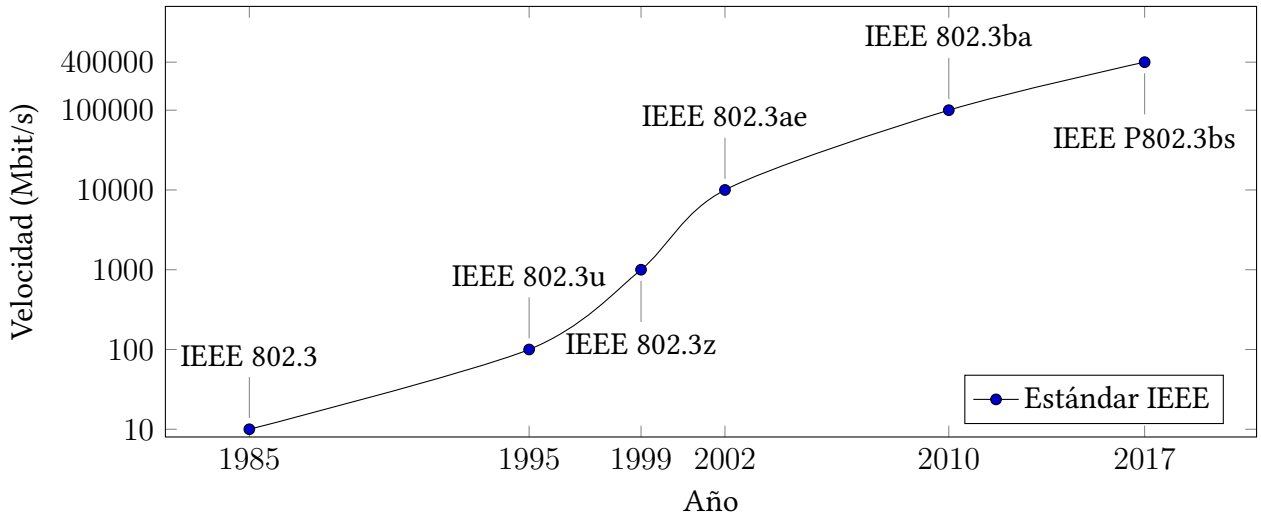


Figura 1: Evolución de la velocidad según el año de la publicación del estándar [1] (Eje y en escala logarítmica)

El ancho de banda obtenido con esta evolución ha sido beneficioso para infinidad de apli-

caciones distribuidas, que realizan múltiples operaciones en las que se intercambian un gran volumen de datos.

No obstante, las aplicaciones críticas y de tiempo real demandan unas características de fiabilidad y rendimiento que, en ocasiones, se ven lastradas también por los sistemas operativos y su pila de protocolos TCP/IP disponible en el *kernel*. Así, es inevitable incurrir en penalizaciones de tiempo por cambios de contexto (las interrupciones que se originan por necesidades de entrada y salida en la tarjeta de red, y por las llamadas al sistema de las funciones de sockets) y copias de datos entre espacios de memoria de usuario y de *kernel*.

Desde hace algunos años, en el ámbito de los equipos de interconexión de redes se está migrando, con notable éxito, desde el modelo tradicional de hardware y software propietario en routers y switches, hacia una arquitectura hardware que usan los ordenadores de propósito general. En este nuevo contexto (de indudable interés económico), también se consiguen beneficios de rendimiento evitando el paso de mensajes a través de la pila de protocolos del kernel, y teniendo todo el control de la tarjeta de red y el procesado de las capas TCP/IP en el espacio de usuario.

La principal motivación para este trabajo ha sido el estudio de las técnicas software empleadas en estos nuevos routers de propósito general, de tal forma que se experimente con la posibilidad de ser aplicadas en otras aplicaciones distribuidas críticas diferentes a las de encaminamiento/clasificación de tráfico. Un ejemplo son los sistemas que siguen un modelo de intercambio de datos basado en la publicación-suscripción. Este modelo, utilizado ampliamente en sistemas en tiempo real y de alta tolerancia a fallos, proporciona una forma precisa para enviar mensajes de manera asíncrona entre proveedores de mensajes (toman el rol de publicador) y los diversos receptores de mensajes (toman el rol de suscriptor) de una forma completamente desacoplada. Otras ventajas de este enfoque es que existen estándares, como DDS (*Data Distribution Service*) [2] que proporcionan a las aplicaciones una capa de *software* de comunicaciones (*middleware*) con calidades de servicio configurables (fiabilidad, disponibilidad, prioridad, consumo de recursos...). DDS es ampliamente utilizado, pues, en diversas áreas, como la robótica, las ciudades inteligentes, dispositivos médicos o la industria aeroespacial y de defensa.

1.2 Objetivos

Como objetivo de este trabajo fin de grado se plantea la posibilidad de migración de (ciertas partes de) un middleware DDS de código abierto, de manera que incluya tanto el manejo de la tarjeta de red como el procesado de la pila TCP/IP directamente en el espacio de memoria de usuario. Para ello, se estudiarán técnicas de puenteo (by-pass) de la pila del kernel, el diseño de la integración de alguna(s) de esas técnicas en un middleware DDS de uso común y la implementación y pruebas que permitan determinar cómo afectan dichas cambios en el rendimiento y latencia de las aplicaciones distribuidas involucradas.

1.3 Estructura del documento

La presente memoria consta de los siguientes apartados:

- **Capítulo 1**: **Introducción.** Trata de ser un breve preámbulo antes de entrar en materia, exponiendo en este capítulo la razón de empezar este proyecto y el objetivo final planteado.

- **Capítulo 2**: **Tecnologías utilizadas.** Se analizarán las diversas tecnologías y herramientas que han sido necesarias para la realización del proyecto, comentando brevemente su origen y el motivo por el cual fueron planteadas para su uso.

- **Capítulo 3**: **Metodología de Trabajo.** Explicación del marco de trabajo utilizado para poder mantener unas buenas prácticas con el fin de conseguir un buen resultado en el trabajo.

- **Capítulo 4**: **Fases del desarrollo.** Descripción de las distintas fases por las que ha pasado el proyecto y los diferentes procedimientos que se han llevado a cabo hasta la obtención del resultado final.

- **Capítulo 5**: **Resultados.** Presentación del resultado obtenido en la ejecución de la solución final lograda en el proyecto, así como una valoración de los datos logrados.

- **Capítulo 6**: **Conclusiones y Líneas futuras.** Con el resultado del trabajo presente, se analizará la posible mejoría en las métricas del *middleware*, sus limitaciones actuales y las posibles extensiones y mejoras a realizar.

2

Tecnologías utilizadas

El presente capítulo analiza las distintas tecnologías que han sido necesarias para la realización de este trabajo fin de grado, explorando sus características técnicas y presentar al lector los distintos aspectos de cada una de ellas que pueden ser desconocidos.

2.1 DDS e implementaciones

Se introduce el software de comunicaciones central sobre el que versa este trabajo fin de grado, así como las distintas implementaciones con las que se ha trabajado en el transcurso del proyecto.

2.1.1 Data Distribution Service (DDS)

Se trata de una especificación de un *middleware* (lógica de intercambio de información entre aplicaciones) siguiendo el patrón de mensajería publicador/suscriptor para su uso en sistemas en tiempo real y sistemas embebidos [3]. Su objetivo es el de simplificar el desarrollo de aplicaciones en sistemas distribuidos ofreciendo una abstracción y descentralización de la comunicación utilizando el patrón publicador/suscriptor, pero también el de proveer una mayor fiabilidad para su implementación en sistemas críticos (que por su ámbito de uso necesitan de tolerancia a fallos y un gran porcentaje de disponibilidad) y proporcionar un alto rendimiento para poder distribuir grandes cantidades de datos con una baja latencia. La versión más reciente de la especificación es la versión 1.4, publicada por el consorcio de estándares tecnológicos OMG (*Object Management Group*), esta versión fue publicada por el consorcio en diciembre de 2004, aunque sus orígenes son muy anteriores tal y como veremos en el siguiente apartado. La versión 1.2 del estándar lanzada en 2006 fue un punto de inflexión al presentar dos niveles

distintos de interfaces: DCPS (*Data-Centric Publish-Subscribe*) y una opcional de mayor nivel, DLRL (*Data Local Reconstruction Layer*). DCPS tuvo una mayor adopción, por lo que la interfaz de menor nivel se convirtió en la más utilizada y en la última versión se decidió sacar DLRL del estándar, teniendo DLRL su propio estándar aparte. En 2010 apareció el estándar DDSI (*DDS Interoperability Wire Protocol*), definiendo un protocolo de interoperabilidad entre las distintas implementaciones de DDS puedan interactuar entre ellas. Las distintas implementaciones adoptaron este protocolo y lo añadieron a su implementación. Es por ello por lo que es común referirse a DCPS y DDSI cuando se habla de la familia de estándares que conforman DDS.

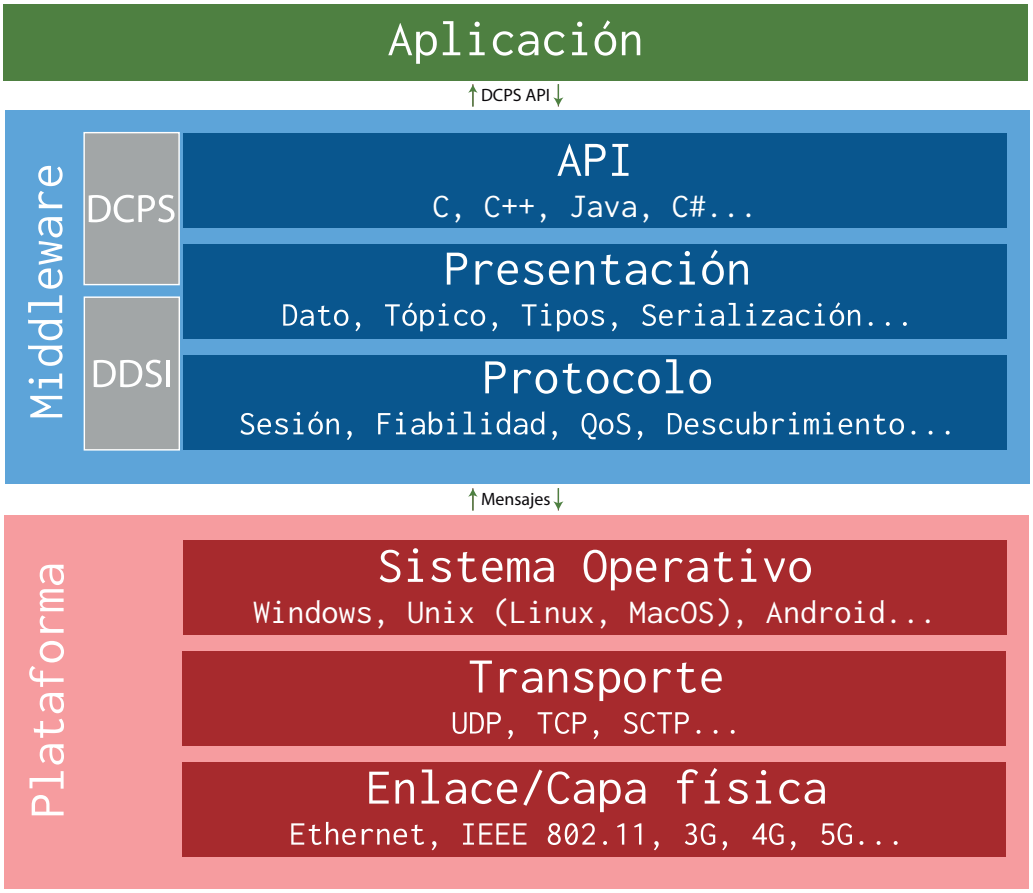


Figura 2: Situación del *middleware* en las distintas capas. Figura traducida extraída de [3] con añadidos.

En la figura 2 podemos situar la capa de abstracción que provee este *middleware* en una arquitectura: abstrae la aplicación de los detalles de bajo nivel como protocolos de transporte o el formato de las tramas, que son manejados por el *middleware*. Es en esta parte de la abstracción donde debemos fijarnos para poder ser capaces de reemplazar las llamadas de red al

sistema operativo y así poder aplicar técnicas de alto rendimiento en la pila de protocolos del sistema operativo: el objetivo principal de este trabajo.

Este *middleware* consta de diversas características diferenciadoras que lo hacen más atractivo frente a otros estándares. Provee una calidad de servicio (más conocido por las siglas QoS, del inglés: *Quality of Service*) altamente configurable que permite controlar el comportamiento del sistema frente a fallos, el consumo de recursos o la fiabilidad en la comunicación. Consta también de un espacio de datos global (el estándar tiene un gran enfoque en los datos) representado en la figura 3, donde se encuentran los tópicos (en inglés: *topics*), que representan la unidad de información que puede ser producida (por el rol de publicador) o consumida (por el rol de suscriptor). Publicador y suscriptor son los encargados de escribir y leer datos respectivamente. Cada tópico, al igual que los publicadores y suscriptores, tienen su propia configuración de la calidad de servicio.

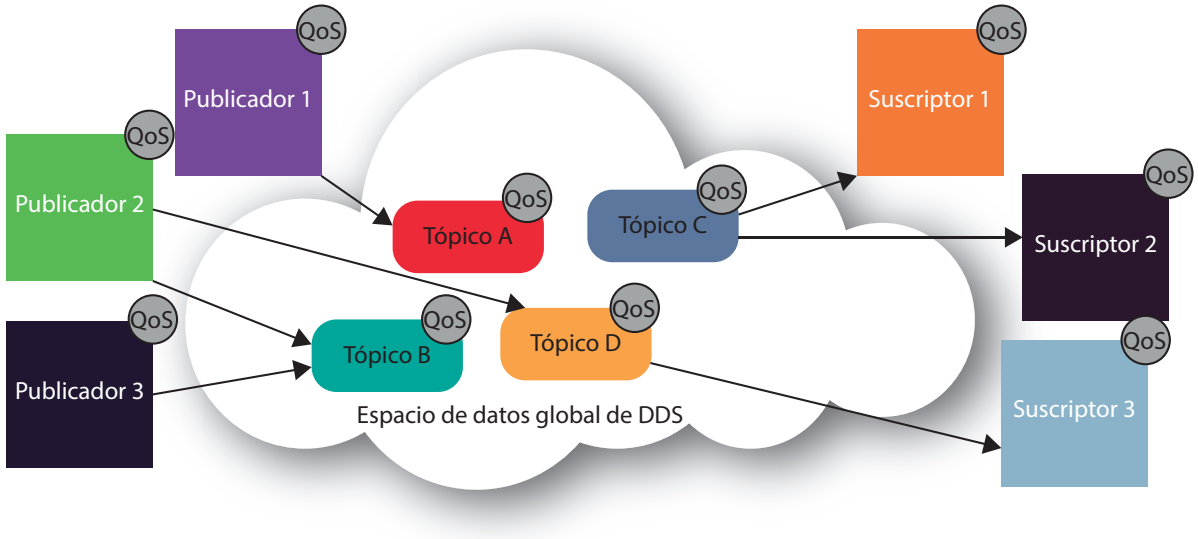


Figura 3: Representación del espacio de datos global y su interacción con otros actores. Figura traducida extraída de [2]

2.1.2 Vortex OpenSplice

Esta implementación del *middleware* DDS tiene sus orígenes en la compañía francesa Thales (más concretamente en su departamento naval para los Países Bajos, *Thales Naval Netherlands*)

bajo el nombre SPLICE-DDS [4] para utilizarlo como piedra angular en la comunicación de su sistema *TACTICOS Combat management system* lanzado en 1991 [5], un sistema que sigue vigente hoy en día. Debido a su origen y por ser una implementación que ha estado en desarrollo durante tantos años, tanto su código como su funcionamiento es muy robusto. Sin embargo, en el transcurso de este trabajo fin de grado, los desarrolladores de la versión de código abierto anunciaron la decisión de discontinuar esta implementación en pos de una implementación más reciente llamada Cyclone DDS. La versión comercial y privativa de OpenSplice sigue siendo mantenida por la empresa ADLINK.

2.1.3 Cyclone DDS

Desde su aparición a finales de la década de 2010, esta implementación de DDS arropada por la Fundación Eclipse ha tenido un gran apoyo por parte de la comunidad, lo que le ha permitido crecer mucho en muy poco tiempo. Ese gran crecimiento ha hecho que se posicione como una de las implementaciones más importantes: ha llegado a ser utilizada en la última generación de ROS (*Robot Operating System*), ROS2. Reutiliza la implementación del protocolo RTPS (*Real-Time Publish-Subscribe*, situado en la capa de DDSI) de OpenSplice y muchos de los programadores que actualmente contribuyen al proyecto contribuían anteriormente a la versión de código abierto de OpenSplice, tal y como se ha comentado en el apartado 2.1.2.

2.2 Lenguajes de programación

Se comentan los lenguajes de programación utilizados para la realización del proyecto, especificando los motivos para su elección y la finalidad de su uso.

2.2.1 C

Diseñado por Dennis Ritchie en los 70 con el fin de programar el sistema operativo UNIX para el ordenador DEC PDP-11 en los laboratorios Bell [6], C es un longevo lenguaje de programación de propósito general que goza de una gran vigencia gracias a su versatilidad, ausencia de restricciones para el programador y la posibilidad de crear programas eficientes y rápidos, unos adjetivos perfectos para la intención del trabajo. Un ejemplo de la ausencia de restricciones en comparación con otros lenguajes es la gran libertad que da el lenguaje C en la gestión de

memoria, de la que se benefician las distintas bibliotecas utilizadas. Además, la gran mayoría de implementaciones de DDS están programadas en C, por lo que este lenguaje es ideal para el desarrollo del proyecto.

2.2.2 Python

Desde su creación en 1991 por el neerlandés Guido van Rossum no ha parado de generar adeptos entre los desarrolladores de *software*, siendo hoy en día uno de los lenguajes de programación más populares. Es un lenguaje de programación multiparadigma que ofrece una gran legibilidad de código simplificando enormemente el desarrollo gracias a su sintaxis débilmente tipada. Su sencillez ha sido clave al permitir la creación de herramientas para facilitar el desarrollo (por ejemplo, la modificación de código) en muy poco tiempo en comparación con otros lenguajes de programación.

2.3 Librerías

Estas librerías fueron vitales para obtener una pila TCP/IP mejorada sobre la que trabajar, pudiendo así reemplazar las funciones de red del *middleware* DDS por las implementadas por estas librerías para tratar de obtener un mejor rendimiento evitando el cuello de botella mencionado en la introducción del trabajo situada en el capítulo 1.

2.3.1 Data Plane Development Kit (DPDK)

Esta librería creada por Intel y administrada por la Fundación Linux ofrece funciones para controlar desde el espacio de usuario el procesamiento de paquetes TCP/IP mediante el uso de controladores de tarjetas de red que hacen uso del sondeo, creando así un marco de trabajo para el procesamiento veloz de paquetes sin ofrecer una pila de protocolos [7]. Utiliza la licencia BSD-3 para las librerías y controladores, y la licencia BSD-2 para los componentes del núcleo.

2.3.2 OpenDataPlane (ODP)

Proyecto de código abierto iniciado en 2013 por la organización Linaro y posteriormente impulsado por empresas como ARM, Enea, Marvell o Nokia que trata de definir una interfaz de programación de aplicaciones (API) para crear un plano de datos (*data plane*) de alto rendi-

miento [8]. El plano de datos es la parte de una red que se encarga de transportar el tráfico del usuario, por lo que la API se encarga de recibir, manipular y transmitir paquetes de red. Tiene dos repositorios con implementaciones de la API de referencia: odp-linux y odp-dpdk, ambos repositorios gozan de buena salud y están mantenidos. La licencia utilizada es BSD-3.

Esta librería es un componente clave para la realización del *bypass*, técnica consistente en la creación de un desvío para evitar algún tipo de obstáculo. En este caso, se busca evitar que el núcleo se encargue de la capa de procesamiento de protocolos para evitar las interrupciones que vienen por la implementación de los *Berkley Sockets* (BSD) por parte del sistema operativo, diseñados a principios de los años 80 haciendo uso de llamadas al sistema para leer y escribir datos a un *socket*, llamadas al sistema que causan cambios de contexto que a su vez merman el rendimiento por necesitar tiempo para realizar todas las operaciones necesarias para el cambio: entre ellas el almacenar los registros del proceso en ejecución en el *process control block* (PCB), para una vez terminada la operación que ha invocado el cambio de contexto se pueda retomar la ejecución. Llevando ese procesamiento al espacio del usuario se evitan los cambios de contexto y el tiempo dedicado a él, ganando así un rendimiento al permitir que la aplicación pueda tener acceso a una cola de paquetes directamente en el espacio de direcciones.

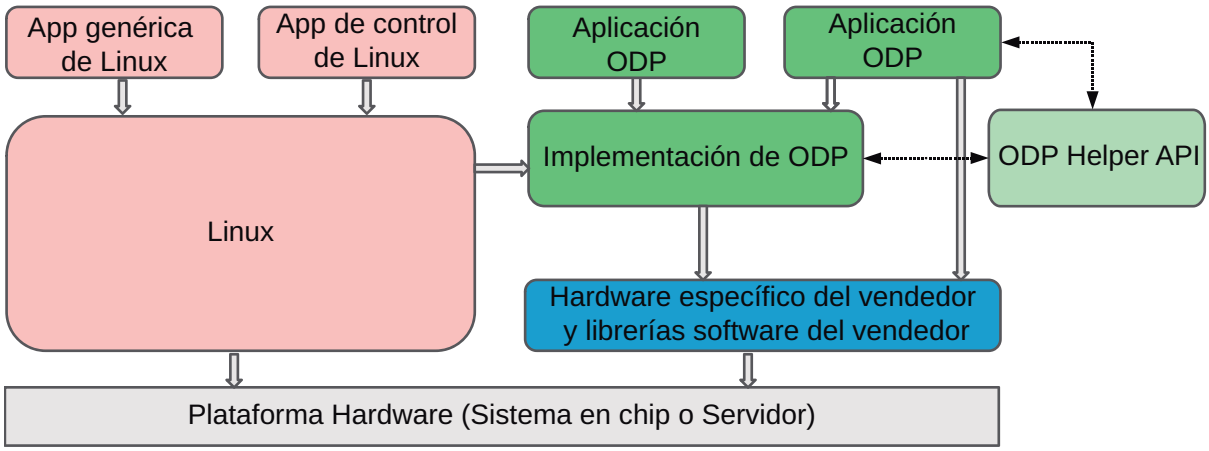


Figura 4: Diagrama de la relación entre los componentes de ODP.

(Figura traducida extraída de [8])

Como vemos en la figura 4, la librería se encarga de ofrecer un plano de datos al núcleo de Linux que pueda interactuar con distintas tecnologías específicas de proveedores, como

podrían serlo las tecnologías privadas de Cisco o Juniper. La implementación se apoya en una API de ayuda, que ofrece apoyo al programador frente a algunas funcionalidades que no están aceleradas por *hardware*.

2.3.3 OpenFastPath (OFP)

Tal y como se puede deducir de su nombre, es un proyecto hermano de OpenDataPlane con los mismos impulsores, pero una misión distinta: OpenFastPath es una biblioteca que define una pila TCP/IP de alto rendimiento mediante técnicas de copia cero utilizando la API proporcionada por OpenDataPlane. Las implementaciones de TCP, UDP e ICMP han sido portadas de libuinet, un *port* de la pila TCP/IP de FreeBSD en el espacio de usuario. La API está muy optimizada al estar basada en eventos no bloqueantes con un diseño multihilo y modular que le permite ser fácilmente ampliable y dotar de sencillez para integrar las novedades desarrolladas en FreeBSD. Actualmente el repositorio original tiene un ritmo bajo de actualizaciones, pero un antiguo colaborador del proyecto llamado Bogdan Pricope mantiene un *fork* [9] muy activo donde añade nuevas características al proyecto.

2.4 Herramientas

Se comentarán algunas herramientas dignas de mención que han posibilitado el desarrollo de este trabajo fin de grado

2.4.1 VMware Workstation

Un hipervisor para la virtualización ha sido necesario para la creación de distintas máquinas virtuales para la instalación de distribuciones basadas en GNU/Linux, así como simular y crear redes virtuales. Para ello ha sido necesaria la versión Pro, que permitía estas funcionalidades. Este software es un software privativo desarrollado por la empresa VMware, por lo que ha sido necesario utilizar la licencia que pone a disposición la Universidad de Málaga para sus estudiantes.

2.4.2 Wireshark

Para la depuración de las aplicaciones Wireshark ha sido un gran apoyo al ofrecer la función de analizar paquetes. Mediante el análisis de los paquetes se han podido sacar conclusiones y encontrar fallos en el código durante el desarrollo. Es un programa de código abierto bajo la licencia GPLv2.

3

Metodología de Trabajo

Para la realización del presente trabajo fin de grado se ha decidido utilizar una metodología ágil de desarrollo de *software*. Debido a la elección de tecnologías punteras para alcanzar la meta de este trabajo, se han encontrado diversos problemas inherentes al uso de tecnologías tan inmaduras. Siguiendo la división que propone el modelo Cynefin [10], este proyecto se podría calificar en su gran mayoría como un proyecto complejo, ya que la escasez de documentación y ejemplos de uso de las bibliotecas ha imposibilitado un buen conocimiento previo de los posibles problemas que podrían surgir al aplicar la biblioteca para llevar la capa de red al espacio de usuario del sistema operativo. Es por ello por lo que afrontar este proyecto desde una perspectiva experimental, es decir, darle un papel principal a la experimentación a la hora de programar tratando de avanzar en el proyecto con pequeños pasos para permitir sacar conclusiones tras probar distintas hipótesis y encontrar cuál es la correcta para poder seguir adelante. Estos "pequeños pasos" son ideales para aplicar una metodología ágil, ya que el concepto de desarrollo iterativo y creciente se adapta perfectamente a la filosofía tomada para afrontar el desarrollo.

Una metodología ágil ampliamente utilizada hoy en día que da una gran importancia al desarrollo incremental e iterativo es Scrum [11]. Presentado en 1995 por Ken Schwaber y Jeff Sutherland en la conferencia OOPSLA (*Object-Oriented Programming, Systems, Languages & Applications*), Scrum es un marco de trabajo para el desarrollo ágil de *software* para tratar de facilitar la gestión de proyectos de gran envergadura. Para ello, se centra en el *sprint*, un periodo de tiempo que no suele exceder las 4 semanas en el que se aborda una parte del proyecto a realizar. Una vez terminado, se revisa el trabajo realizado y se toman decisiones para abordar

el resto del proyecto.

Aunque parta de la idea sencilla de centrar el desarrollo en distintos *sprints*, realmente Scrum ofrece una especificación mucho más desarrollada para mantener buenas prácticas en un proyecto donde distintos roles deben trabajar juntos para conseguir un resultado final que sea del agrado del cliente. Es por ello que no se ha aplicado este marco de trabajo de una manera rigurosa y se han tomado ciertas licencias para simplificar todo el proceso. Se ha realizado la siguiente serie de forma ininterrumpida hasta la obtención del resultado final:

- **Fase 0**: Estudio de la tecnología a utilizar.
- **Fase 1**: Planificación de los objetivos del *sprint*.
- **Fase 2**: Realización del *sprint*.
- **Fase 3**: Análisis del resultado obtenido.

Como se puede apreciar, dista mucho de ser una forma canónica de llevar a cabo Scrum, pero ha sido la mejor opción para encauzar este trabajo y ha dado buenos resultados a la hora de detectar posibles fallos y problemas futuros que se podrían dar.

4

Fases del desarrollo

Una vez detalladas tanto las distintas tecnologías como la metodología utilizada para la realización de este trabajo fin de grado, se explica en este capítulo todas las fases por las que ha pasado el proyecto, las dificultades encontradas y las decisiones tomadas hasta obtener un resultado funcional.

4.1 Diseño

Al comienzo del trabajo se decidió utilizar una combinación de DPDK (*Data Plane Development Kit*), ODP (*Open Data Plane*) y OFP (*Open Fast Path*) para aplicarlas a una implementación de DDS. Para ello, fue necesario la lectura de la documentación de las distintas librerías para proceder a la instalación en una distribución de GNU/Linux. Todo este proceso ha quedado documentado en los apéndices A, B y C correspondientes a los manuales de instalación de DPDK (a través de la capa ODP-DPK), ODP y OFP respectivamente. Una vez que se consiguió tener las librerías instaladas de forma correcta en el sistema (e investigar ciertas configuraciones opcionales que eran vitales para el objetivo del proyecto, como el uso del driver *Virtual Function I/O* (VFIO) con DPDK para dotar la implementación de una mayor seguridad), el siguiente paso fue integrar las librerías en una implementación de DDS. La implementación escogida en esta fase del proyecto fue OpenSplice, ya que su amplio uso y su larga trayectoria denotaban que esta implementación consta de un código con una gran robustez y podría ser un buen candidato para acoplar las librerías. Sin embargo, esa larga trayectoria también conllevaba lidiar con código anticuado o algunas tecnologías que cada vez son menos comunes en detrimento de otras nuevas capaces de proveer al usuario un uso más agradable. Esto se ha visto ejemplificado a la hora de dar primer paso de la integración de las librerías en OpenSplice, puesto que utiliza la herramienta de automatización de compilación *make* y diversos *makefiles* para las distintas plataformas para las que se puede compilar el *middleware*. Se tu-

vieron que modificar los CFLAGS y LDFLAGS para incluir las librerías de DPDK, ODP-DPDK y OFP. Respetando el esquema que proponen los desarrolladores de OpenSplice se realizaron las siguientes modificaciones en un *fork* creado para realizar este trabajo:

- Creación de un nuevo directorio en la carpeta "setup" llamado "x86_64.linux-ofp", siguiendo la convención de la arquitectura que queremos utilizar: x86 de 64 bits, para distribuciones basadas en Linux. En este directorio, fue necesario crear un nuevo archivo "config.mak" donde se añadieron los CFLAGS y LDFLAGS anteriormente comentados.

- En la carpeta "setup", un nuevo archivo "x86_64.linux-ofp" para cambiar las variables necesarias para nombrar ficheros de salida de la compilación.

Con estos cambios realizados se probó la compilación y se procedió al siguiente paso: reemplazar las llamadas a las funciones de red originales por las que provee la librería OFP. Como la implementación del *middleware* aunque contara con una capa de abstracción de todas las llamadas de funciones de red localizadas en el directorio "src/abstraction/os-net/" tenía una cantidad inmensa de llamadas a las funciones por todo el código fuente, se realizó un sencillo programa con el lenguaje de programación Python para reemplazar las llamadas a las funciones de red o los tipos necesarios por los de OFP.

Debido a la disposición del código se pudo realizar la herramienta para reemplazar las llamadas en los ficheros de código utilizando únicamente expresiones regulares, evitando así afrontar programar una aplicación similar mediante utilidades como YACC/BISON y Lex para interpretar el código C y reemplazar las llamadas a las funciones que nos interesan, un enfoque mucho más potente pero que podría eternizar el desarrollo y tomar un camino que atrasaría la obtención del objetivo final del trabajo. Una vez solucionados algunos problemas surgidos en los *makefiles* a la hora de compilar OpenSplice con las llamadas a OFP (esto llevó a hacer los *makefiles* de una manera más robusta con el uso de la herramienta pkg-config), se detectó que OFP no proveía una implementación de la función `getsockname()`. En un principio, se integró una función para que actuara de adaptador para posibilitar que la función `getsockname()` original recibiera unos parámetros compatibles, lo que fue suficiente para posibilitar la compilación. Sin embargo, finalmente se creó un *fork* de OFP donde se añadió una implementación de la función realizada en el marco de este trabajo fin de grado.

Una vez que todas las llamadas originales habían sido reemplazadas por las de OFP, se ejecutó los distintos ejemplos de OpenSplice con un resultado insatisfactorio: se obtuvo un error 22 (EINVAL, argumento inválido) en una llamada a `pthread_create()` en la inicialización de DDSI. Al investigar la traza de ejecución no había rastro de ninguna llamada de OFP, por lo que fue necesario dar un paso atrás y realizar distintos ejemplos haciendo uso de OFP. Se realizó un ejemplo de cliente/servidor eco en el *fork* de OFP que se creó anteriormente. Para ello, se tomó como base los ejemplos de la documentación de z/OS de IBM [12] y se reemplazaron las llamadas por las de OFP tal y como se hizo con OpenSplice. Esto sirvió para buscar posibles limitaciones de OFP y probar si distintas hebras en ejecución eran capaces de recibir y mandar mensajes con las funciones de OFP, además de adquirir una mayor familiaridad con el funcionamiento de la librería y sus funciones para la inicialización, más relacionadas con ODP.

Hechas las comprobaciones, se verificó que OFP era capaz de mandar mensajes en distintas hebras, pero se detectó que las funciones `connect()` y `accept()` con TCP daban problemas independientemente de si el mensaje se mandaba entre hebras en una máquina distinta o el destinatario era una máquina externa. Aunque fue un descubrimiento útil, no imposibilitó seguir con el trabajo, ya que el objetivo principal era utilizar UDP al ser el caso de uso más común del *middleware* DDS. Con estos conocimientos adquiridos de la librería y descartados problemas y limitaciones técnicas de ésta, se plantearon las siguientes opciones para tratar de arreglar el problema.

Opción 1: Eliminar llamadas de OFP para encontrar algún culpable.

Esta opción se probó, pero no se llegó a obtener ninguna conclusión clara al tener que eliminar la gran mayoría de llamadas.

Opción 2: Compilar OpenSplice con librerías estáticas.

Para descartar algún problema a la hora de compartir espacio de memoria con las librerías y posibles conflictos entre ellas, se compilaron todas las librerías de forma estática. A pesar del cambio, el error seguía presente en la ejecución.

Opción 3: Probar la utilización de OFP sin DPDK.

Se reemplazó la capa de ODP-DPDK por la implementación de ODP, tal y como recomiendan los programadores de la librería para descartar fallos inherentes a

DPDK. El error persistía, por lo que el error no residía en esta librería.

Opción 4: Probar otra librería que proporcione una pila TCP/IP para DPDK.

Se eligió F-Stack como candidato, ya que además de tener más funciones implementadas que OFP, gozaba de buenas referencias en internet de su uso. Desarrollada por la empresa china Tencent, esta librería se integró en OpenSplice haciendo uso del programa con Python realizado para modificar los ficheros fuentes comentado anteriormente. Sin embargo, al ver que no se ejecutaba correctamente, se encontró un *issue* en GitHub donde se comentaba que no era posible utilizar esta librería con aplicaciones que no utilicen una implementación con *polling* en vez de interrupciones [13], por lo que no era compatible con la implementación realizada con OpenSplice.

Opción 5: Utilizar LD_Preload.

Una opción que se menciona en la documentación de OFP es el uso de la variable de entorno LD_Preload que permite al enlazador resolver el enlazamiento de librerías compartidas y resolución símbolos en tiempo de ejecución. Esto permite ejecutar aplicaciones que hagan uso de las llamadas POSIX de red que reimplementa OFP sin cambiar ninguna línea de código, aprovechando la posibilidad de cambiar la resolución de símbolos para reemplazar las funciones y los tipos originales por los de OFP. Se siguieron los pasos de la documentación de OFP y se ejecutó OpenSplice compilado con las llamadas a las funciones de red originales. Se obtuvo de nuevo el mismo error, por lo que se descartó algún fallo a la hora de reemplazar las llamadas en OpenSplice.

Opción 6: Cambiar implementación de DDS.

Como última opción se contemplaba el cambio de OpenSplice por dificultar los cambios en una fase temprana y la imposibilidad de encontrar el origen del fallo con `pthread_create()`. Finalmente hubo que tomar este camino para poder seguir avanzando con el desarrollo del proyecto.

4.2 Implementación

Tras el periplo con OpenSplice se inició la búsqueda de una implementación de DDS que pudiera ser más propicia a ser modificada y tuviera tanto una importancia similar a la de OpenSplice como un buen respaldo por parte de la comunidad para asegurarse de la continuidad de la implementación. Después de revisar las distintas implementaciones libres disponibles en internet, la implementación de DDS escogida para volver a empezar el proyecto fue CycloneDDS que compartía similitudes en su implementación tal y como se comentó en el capítulo 2. Como ya se había utilizado la opción de LD_Preload para utilizar OFP, se decidió utilizar este enfoque al permitir un desarrollo más seguro frente a posibles fallos al reemplazar múltiples funciones en el código. También proporciona una gran facilidad de portar los avances a otras implementaciones de DDS al trabajar con una herramienta que permite reemplazar los símbolos necesarios en tiempo de ejecución, pudiendo así aplicarlo a otras implementaciones de DDS. Esto podría facilitar el uso de OFP con otras implementaciones como es el caso de FastDDS, ya que esta implementación hace uso de la librería Asio (una librería para la programación de redes escrita en C++) y se podrían sobrecargar las funciones, evitando así tener que reescribir gran parte del proyecto.

La implementación para el uso de LD_Preload en OFP está realizada de la siguiente manera:

- En el directorio "example/ofp_netwrap_proc" se encuentra la implementación necesaria para la inicialización y configuración de ODP y OFP.
- En "example/ofp_netwrap_proc" se encuentra la sobrecarga de símbolos mediante el uso de la función `dlsym()` y la conversión de tipos nativos y los necesarios para OFP.
- En los ficheros "scripts/ofp_netwrap.sh" y "scripts/ofp_netwrap.cli" se encuentran respectivamente el script para inicializar un ejecutable haciendo uso de LD_Preload y el fichero necesario de configuración de OFP.

No todas las llamadas estaban implementadas para realizar la sobrecarga de símbolos. En "example/ofp_netwrap_proc" aparecen varias funciones realizadas, pero no todas las llamadas indispensables para su uso con CycloneDDS, especialmente las que tenían una relación

estrecha con UDP. Se crearon funciones para `sendto()`, `getsockname()` y `recvfrom()`. La estructura típica para implementar una función para su sobrecarga es la siguiente:

```
int func(int sockfd, params)
{
        int func_value;

        if (IS_OFP_SOCKET(sockfd)) {
                func_value = ofp_func(sockfd);
                errno = NETWRAP_ERRNO(ofp_errno);
        } else if (libc_func)
                func_value = (*libc_func)(sockfd);
        else { /* pre init*/
                LIBC_FUNCTION(func);

                if (libc_func)
                        func_value = (*libc_func)(sockfd);
                else {
                        func_value = -1;
                        errno = EACCES;
                }
        }

        /*printf("Func returns:'%d'\n",
                sockfd, func_value);*/
        return func_value;
}
```

Figura 5: Estructura estándar de la implementación de la sobrecarga

Analizando el código de la figura 5 vemos como comprueba si el descriptor del socket ha sido inicializado para OFP. Para ello, en el valor que devuelve la función `socket()` se activa un bit del entero para su posterior verificación con una máscara: si ese bit está activo, el descriptor habrá sido creado con la función `socket()` de OFP. Si fuera el caso, se debe escribir el código necesario para convertir los tipos originales del resto de argumentos por los necesarios para OFP. Si el descriptor no procediera de una llamada a la función `socket()` de OFP, se intentaría ejecutar la función original. También se contempla el caso en el que todavía no se ha realizado la sobrecarga de funciones y no se han cargado las funciones de OFP, por lo que en ese caso también se ejecutará la función original.

Además de las funciones añadidas, también se necesitaban las funciones `recvmsg()` y `sendmsg()`, funciones que no estaban implementadas en OFP como era el caso de `getsockname()`. Sin embargo, se modificó el código para utilizar `recvfrom()` en vez de `recvmsg()` y tanto `send()` para TCP como `sendto()` para UDP en vez de `sendmsg()`, posibilitando así el uso de estas funciones sin tener que implementar nuevas funciones de OFP a partir de la implementación de libuinet.

Con todos estos progresos, al ejecutar CycloneDDS con el LD_Preload modificado, se encontraba un error en el inicio. Depurando con GDB se pudo encontrar la causa: no se estaba inicializando bien OFP. El problema venía a la hora de crear hilos en la capa DDSRT (*DDS Runtime*), una capa que ofrece implementaciones de funciones necesarias para DDSC y DDSI. Esta capa provee una función para la creación de hilos, `ddsrt_thread_create()`, que a su vez llama a `os_startRoutineWrapper()`, un envoltorio para la creación de hilos mediante `pthread_create()`. Era necesario inicializar OFP para cada hilo que necesitara realizar una llamada a sus funciones, por lo que una vez localizada la función envoltorio solo hubo que implementar una función de inicialización de OFP para que se pudiera llamar a la hora de crear los hilos.

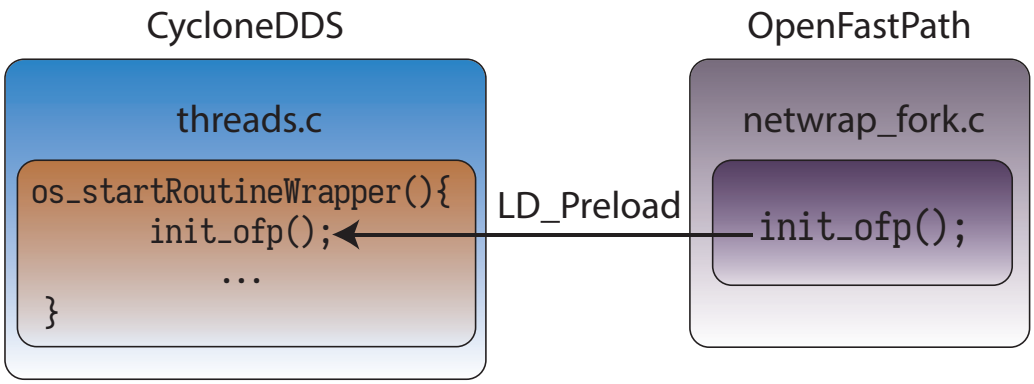


Figura 6: Inyección de la función necesaria para la inicialización local de la librería en las hebras.

Como se puede observar en la figura 6, se hizo uso de LD_Preload para llamar a esa función de inicialización, ya que no se tenía acceso a las funciones de OFP y ODP desde el código de CycloneDDS al estar utilizando LD_Preload para sobrecargar las llamadas necesarias. Para poder inicializar la función, se definió la función `init_ofp()` tanto en CycloneDDS como en OFP, pero siendo la implementación en CycloneDDS una implementación vacía para poder definir el símbolo para su posterior sobrecarga con LD_Preload, cargando así la función definida en OFP y teniendo acceso a las funciones y variables necesarias para la inicialización de la librería en cada hebra lanzada.

Con estos cambios ya se consiguió ejecutar el *middleware* realizando llamadas a las funciones de OFP sobre la capa de ODP sin DPDK.

4.3 Pruebas

Para probar el correcto funcionamiento de CycloneDDS con la integración de OFP mediante LD_Preload, se ejecutaron los ejemplos de CycloneDDS como *HelloWorld*, un ejemplo sencillo de Hola Mundo donde un publicador y un suscriptor comparten un mensaje sencillo consistente en un ID de usuario y un mensaje. Se ejecutó el publicador y suscriptor en la misma máquina virtual y con Wireshark se comprobaron los paquetes de la red para supervisar el correcto funcionamiento. Con esta herramienta se vio que había un problema: publicador y suscriptor no conseguían conectar. Se trató de configurar a mano CycloneDDS a través de su fichero de configuración para descartar un problema en la configuración automática, pero

no sirvió para solucionar el problema. Para tratar de descartar otros problemas, se ejecutó un cliente y un servidor con OFP mediante LD_Preload para descartar problemas en la implementación. Primeramente, se probó el cliente y servidor con TCP para ver las limitaciones de OFP (por ejemplo, si no se podían ejecutar dos aplicaciones al mismo tiempo haciendo uso de OFP en la misma máquina virtual o si era posible conectar distintos dispositivos haciendo uso de llamadas del sistema en un dispositivo y llamadas a OFP en otro). Se realizaron las siguientes comprobaciones:

- Ejecutar en la misma interfaz de red en una única máquina virtual, con la interfaz tomada por OFP. Se obtuvo un error de ODP: "Bad exit status cpu #0".
- Ejecutar con dos interfaces de red en una misma máquina virtual, tomando ambas interfaces con OFP con una configuración distinta. De esta forma, se podía ejecutar en una misma máquina virtual.
- Ejecutar en dos máquinas virtuales distintas utilizando OFP en ambas. También se podía ejecutar sin problema de esta manera.
- Ejecutar una máquina virtual con OFP y otra máquina virtual utilizando las llamadas nativas de la biblioteca C de GNU. Se podían comunicar en ambas direcciones sin problema.

Conociendo finalmente las limitaciones de configuración de OFP con LD_Preload para tomar interfaces de red para enviar y recibir paquetes, así como la forma de desactivar un interfaz de red para su uso con OFP (si no se borra la configuración del interfaz puede causar un error con ICMP), se procedió a probar el cliente y servidor con UDP. Se detectó que había un problema en la implementación de `recvfrom()` para la sobrecarga en LD_Preload, un *flag* del datagrama no estaba implementado en OFP y al utilizarlo descartaba el paquete en su recepción.

Una vez con todo el LD_Preload funcionando correctamente, se volvieron a ejecutar los ejemplos de CycloneDDS. Esta vez se logró una ejecución satisfactoria del Hola Mundo y de otro ejemplo interesante para medir el impacto de nuestro trabajo, *roundtrip*, que ofrece métricas de tiempo de ida y vuelta de los paquetes.

Posteriormente se probó a utilizar la capa de ODP-DPDK para usar DPDK con OFP, pero causaba un conflicto por no tener CycloneDDS el código necesario para reconocer la interfaz creada por DPDK, lo que imposibilitaba el tener acceso a la interfaz y por ende el envío y recepción de paquetes. Además, se encontró un *issue* en el GitHub de ODP donde los temporizadores de las hebras de control no funcionaban correctamente con la capa de ODP-DPDK [14], por lo que podría generar problemas a la hora de utilizar esta capa en vez de utilizar ODP sin DPDK.

5 Resultados

En este capítulo se representan los resultados obtenidos con la ejecución de ciertas pruebas para tratar de comprobar si efectivamente el uso de las librerías en el estado actual han supuesto una mejora en el rendimiento a la hora de ejecutar el *middleware* o si, por el contrario, la aplicación de las tecnologías utilizadas para reemplazar las llamadas de red originales no han supuesto una mejora significativa.

5.1 Latencia

Una vez obtenida una implementación funcional capaz de sobrecargar las funciones necesarias para utilizar OFP con CycloneDDS, se ejecutó una prueba de *roundtrip* para medir la latencia y comprobar si hay una posible mejora después del uso de OFP. La máquina virtual se ejecuta sobre un anfitrión con el sistema operativo Windows 10 Education (una edición con menos telemetría similar a la edición Pro) y se encarga de virtualizar el sistema operativo Lubuntu 20.04.3, una distribución GNU/Linux basada en Ubuntu pero utilizando el entorno de escritorio LXDE en lugar del entorno por defecto de Ubuntu, Gnome, para poder aprovechar mejor los recursos de *hardware* liberando memoria RAM y procesos. El ordenador sobre el que se ha ejecutado consta de un procesador Intel© Core© i7-6700HQ de 2.6 GHz. Aunque es un procesador potente, se debe tener en cuenta que estos datos podrían ser mejores: al tratarse de un ordenador portátil, el procesador no consta de la mejor refrigeración posible, por lo que puede alcanzar fácilmente temperaturas que provoquen una caída en el rendimiento. También hay que recordar que todo el desarrollo y la ejecución del resultado final se ha realizado con máquinas virtuales, por lo que los datos que se obtendrían con equipos físicos distintos conectados por tarjetas de red físicas y no virtuales habrían sido mucho más estables. Sin embargo, como ambas implementaciones se ejecutan en las mismas condiciones, ambas se encuentran en igualdad y es posible compararlas siempre y cuando tengamos en cuenta ciertos factores.

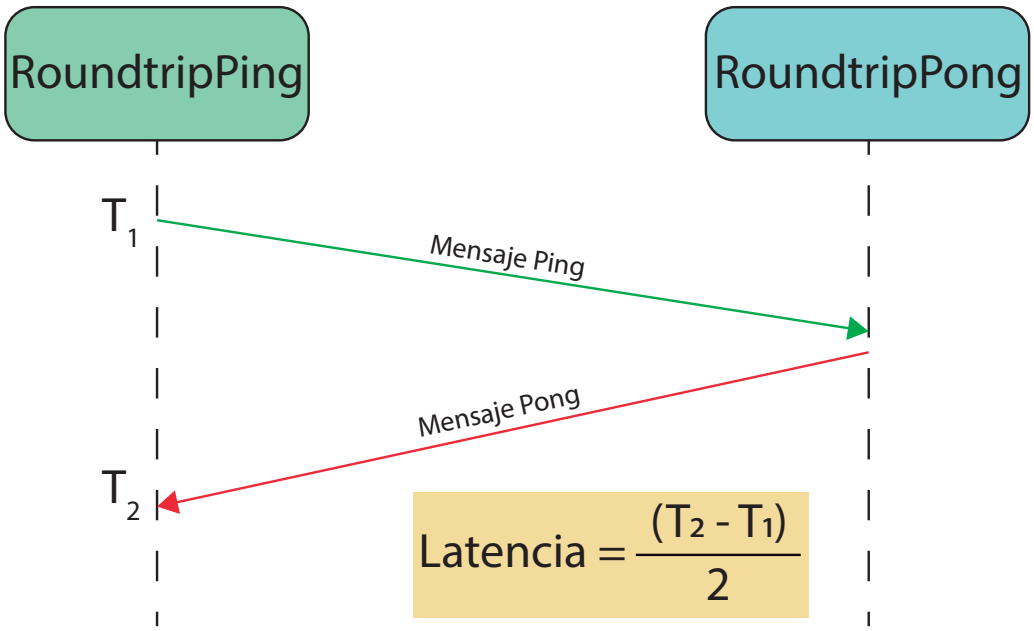


Figura 7: Representación de la medición de la latencia en la prueba.

(Basado en la figura de [15])

La latencia (un concepto muy presente en la ingeniería y la física) en redes informáticas es la cantidad de tiempo en la que un paquete (o un mensaje) tarda en transmitirse entre dos puntos en nuestra red, es decir, mide la cantidad de tiempo que pasa desde el envío hasta la recepción de un paquete entre dos máquinas. Algunas veces, la medición de la latencia no es algo trivial como en este caso y se debe recurrir a pequeños trucos para medirla. Como una vez que se envía el mensaje desde un publicador no se puede conocer inmediatamente el instante de tiempo en el que ha recibido un suscriptor externo el mensaje desde el mismo proceso del publicador, no es posible saber exactamente en tiempo de ejecución el tiempo que se ha tardado en recibir el mensaje. Una solución para conocer una estimación de la latencia en estos casos es el planteamiento representado por la figura 7, donde se realiza lo que se conoce como un *roundtrip* (viaje de ida y vuelta) consistente en mandar un mensaje y esperar a la respuesta del destinatario, anotando así el instante de tiempo en el que se recibe la respuesta. De esta manera, suponiendo que la cantidad de tiempo en la que se realiza la ida y la vuelta es la misma, podemos dividir entre dos la diferencia de tiempo y obtener así una aproximación del tiempo que ha tardado el envío en una única dirección y no en dos, obteniendo así la aproximación del tiempo transcurrido desde el envío del datagrama UDP por parte del ejecutable *RoundtripPing* hasta la recepción por parte del ejecutable *RoundtripPong* (ver figura 7).

5.1.1 Metodología

Para la ejecución del programa se han utilizado dos máquinas virtuales con el objetivo de ejecutar cada programa en una máquina virtual distinta. Para ello, se utilizó una red definida en VMware llamada VMnet 8 de tipo NAT, con servidor DHCP y con la dirección de subred que se puede observar en la figura 8, 192.168.43.0/24. Cada máquina virtual tiene una interfaz fp0, interfaz generada por OFP. En cada máquina virtual se ejecuta el programa correspondiente haciendo uso de LD_Preload para utilizar las llamadas de las funciones de red de OFP.

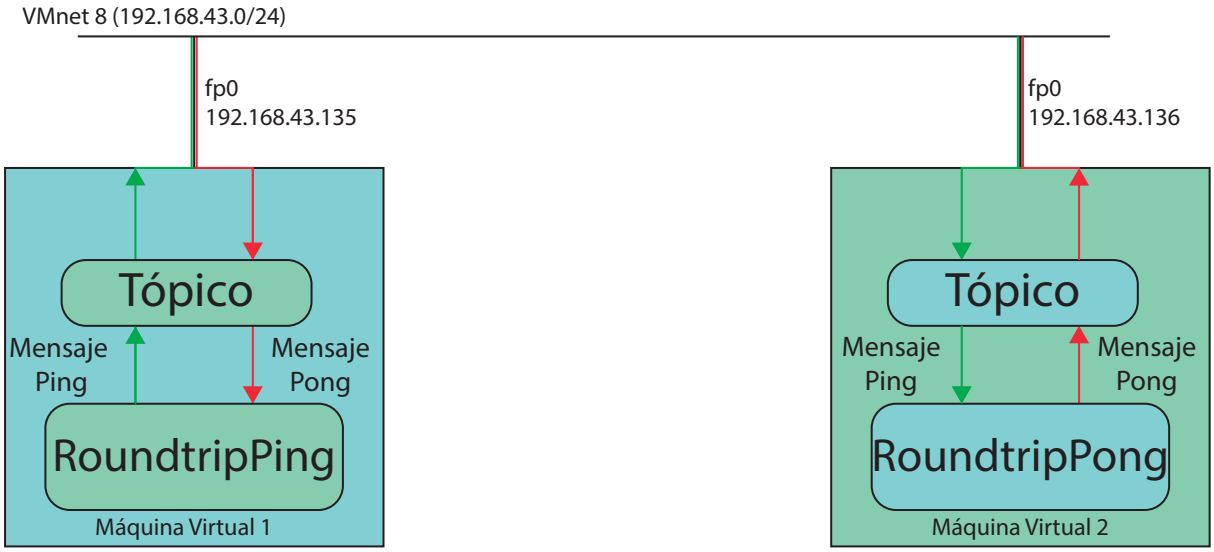


Figura 8: Topología de la red y esquema de la conexión.

La comunicación comienza con el envío del primer mensaje Ping de la figura 7 utilizando el tópico definido de la siguiente manera en el código:

```
topic = dds_create_topic (participant, &RoundTripModule_DataType_desc,
                                       "RoundTrip", NULL, NULL);
```

Donde se le pasa por valor el participante del dominio creado, se le pasa por referencia el descriptor del tópico definido por la figura 9 y se le nombra como "*Roundtrip*". Los parámetros para la configuración del QoS y el *listener* se dejan en Null.

```
module RoundTripModule
{
  struct DataType
  {
    sequence<octet> payload;
  };
  #pragma keylist DataType
};
```

Figura 9: Fichero de la descripción del tópico para el *Roundtrip*.

En la figura 9 se puede observar como se define la estructura del tópico con un campo *payload* (carga útil del mensaje a enviar) cuyo tamaño se puede variar a la hora de invocar el programa que manda el mensaje Ping mediante el uso de los parámetros.

```
static ExampleTimeStats *exampleAddTimingToTimeStats
  (ExampleTimeStats *stats, dds_time_t timing){
        ...
  stats->average = ((double)stats->count * stats->average + (double)timing) / (double)(stats->count + 1);
  stats->min = (stats->count == 0 || timing < stats->min) ? timing : stats->min;
  stats->max = (stats->count == 0 || timing > stats->max) ? timing : stats->max;
  stats->count++;

  return stats;
}
```

Figura 10: Medición del tiempo en el código.

Para realizar la medición de tiempos en el publicador, se hace una llamada a la función de la figura 10 cada vez que hay datos disponibles para la lectura. Se calcula la media y con el operador ternario, se actualizan los valores mínimos y máximos, incrementando también el número de paquetes recibidos. Cada segundo se imprimen por pantalla estas estadísticas que se han guardado en un *struct* y se ponen los valores a cero para poder volver a tomarlas para ese intervalo de tiempo de un segundo de nuevo. La mediana se calcula de manera trivial en otra función distinta.

Tiempo transcurrido (s)		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Latencia mín.	**OFP**	107	111	120	105	103	113	109	110	110	111	106	114	116
(μs)	**Sin OFP**	171	162	163	163	161	162	151	150	151	138	149	152	162

Tiempo transcurrido (s)		**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
Latencia mín.	**OFP**	106	128	120	112	107	119	118	120	122	110	111	111	113
(μs)	**Sin OFP**	162	150	151	147	145	147	150	163	162	151	151	152	150

Tiempo transcurrido (s)		**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
Latencia mín.	**OFP**	109	118	114	99	106	118	108	106	116	114	101	118	117
(μs)	**Sin OFP**	150	152	150	151	150	151	152	150	150	148	150	150	162

Tiempo transcurrido (s)		**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**
Latencia mín.	**OFP**	111	105	101	110	115	99	109	116	111	111	109
(μs)	**Sin OFP**	162	153	148	154	151	163	169	163	158	176	167

Tabla 1: Valores de la latencia mínima obtenidos en la ejecución de *roundtrip*.

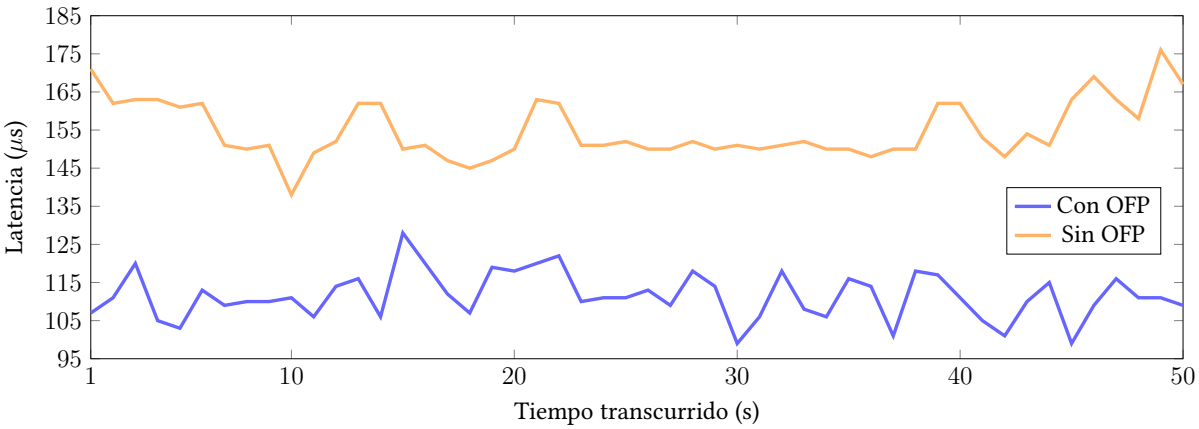


Figura 11: Evolución de la latencia mínima en la ejecución de *roundtrip*. Datos obtenidos de la tabla 1.

Tal y como se puede apreciar en la figura 11, vemos una considerable diferencia en la latencia con una clara ventaja con el uso de OFP. Aunque aparentemente sin OFP obtenemos una mayor estabilidad alrededor del segundo 30, son variaciones que no se encuentran bajo una ejecución en *hardware* real y no virtualizado, por lo que se trata de una inestabilidad causada por la virtualización de la red.

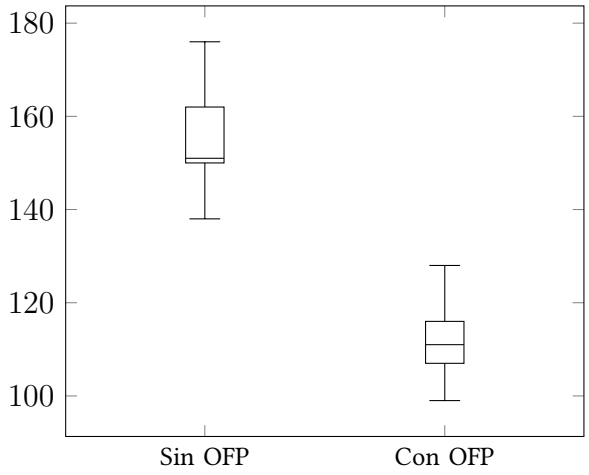


Figura 12: Diagrama de cajas y bigotes de la latencia mínima. Datos obtenidos de la tabla 1.

Con el diagrama de la figura 12, podemos visualizar una gran diferencia entre las medianas de los datos, que muestran una gran separación en la gráfica, lo que nos revela esa ganancia obtenida con OFP ortogando una menor latencia. También se puede apreciar claramente la diferencia entre los valores máximos y mínimos, apreciando como los bigotes están también distanciados. Se puede contemplar también el sesgo que presentan los datos sin OFP, que toman unos valores realmente altos al final de la ejecución.

Tiempo transcurrido (s)		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Mediana	**OFP**	182	182	174	173	178	186	180	170	167	178	180	188	188
(μs)	**Sin OFP**	235	228	220	226	224	218	214	213	210	208	212	218	244

Tiempo transcurrido (s)		**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
Mediana	**OFP**	191	184	180	198	198	182	188	189	197	202	178	182	194
(μs)	**Sin OFP**	223	221	216	214	212	214	221	220	232	214	214	212	210

Tiempo transcurrido (s)		**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
Mediana	**OFP**	197	188	185	178	192	186	196	180	178	190	192	182	188
(μs)	**Sin OFP**	212	219	217	208	214	223	212	210	217	216	215	214	218

Tiempo transcurrido (s)		**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**
Mediana	**OFP**	168	186	190	174	176	171	182	190	182	175	186
(μs)	**Sin OFP**	223	222	209	220	214	218	231	218	219	232	232

Tabla 2: Valores de la mediana de la latencia obtenidos en la ejecución de *roundtrip*.

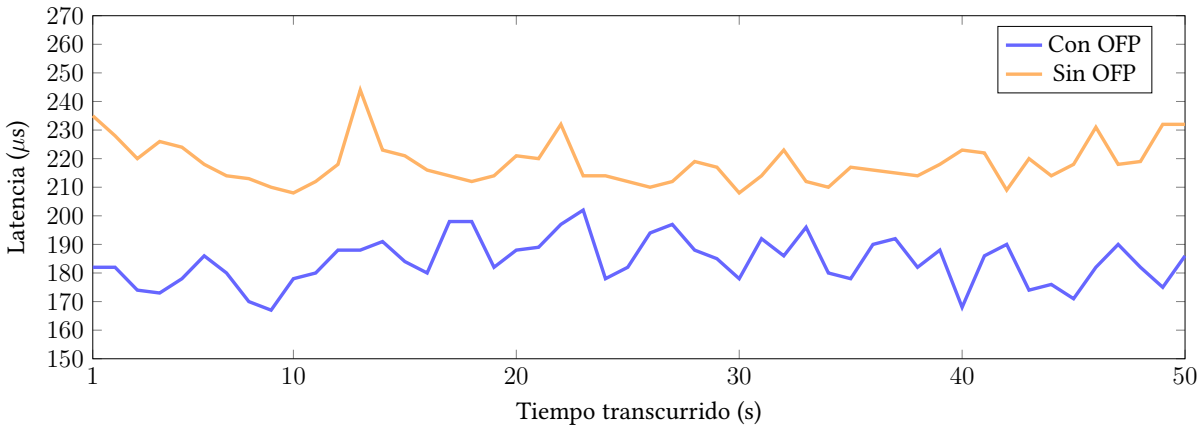


Figura 13: Evolución de la mediana de la latencia en la ejecución de *roundtrip*. Datos obtenidos de la tabla 2.

Como se observa en la figura 13, los datos de la mediana también presentan una mejoría al utilizar OFP. Aunque se percibe claramente el ruido, teniendo en cuenta que estamos midiendo la latencia en microsegundos, los datos obtenidos con el uso de OFP son alentadores y especialmente en los primeros segundos de ejecución la diferencia se aprecia notablemente.

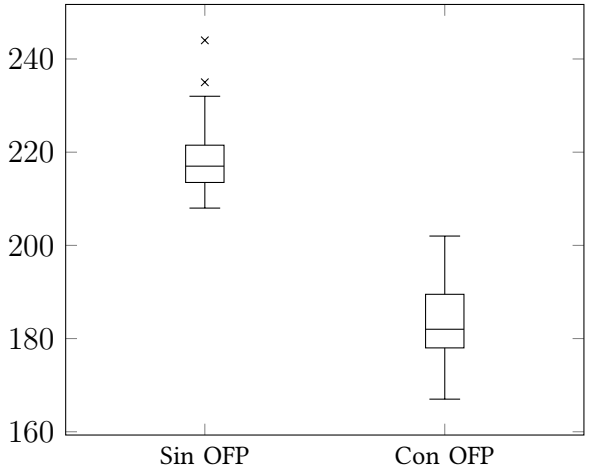


Figura 14: Diagrama de cajas y bigotes de la mediana de la latencia. Datos obtenidos de la tabla 2.

En el diagrama de cajas y bigotes de la figura 14 se aprecia la diferencia entre la mediana de las mediciones de la mediana de cada intervalo, y como con OFP se presenta cierta simetría entre los valores máximos y mínimos representados en los bigotes, mientras que sin OFP encontramos como los valores máximos toman una mayor presencia. La dispersión de los datos

es menor sin OFP y presenta unos valores atípicos, por lo que también se aprecia el ruido.

Tiempo transcurrido (s)		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Latencia máx.	**OFP**	5388	5233	4482	5525	5996	4232	8073	6324	21663	11097	8550	19229	17924
(μs)	**Sin OFP**	13096	4138	16163	2422	13043	3956	15500	4040	12272	4012	4449	2627	10059

Tiempo transcurrido (s)		**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
Latencia máx.	**OFP**	6232	7749	4904	5838	8279	22689	8962	8109	15917	4839	10942	7349	13836
(μs)	**Sin OFP**	2991	13078	2365	7354	2379	6742	3883	10050	2307	19546	4095	16908	2364

Tiempo transcurrido (s)		**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
Latencia máx.	**OFP**	7474	7611	5688	6528	9646	5677	7541	9611	5652	8299	9123	5453	5454
(μs)	**Sin OFP**	18699	2284	9395	2497	13484	2546	10923	3956	5130	2471	13733	2229	17738

Tiempo transcurrido (s)		**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**
Latencia máx.	**OFP**	6903	6665	3927	6756	5854	5453	4463	8316	8363	29414	15880
(μs)	**Sin OFP**	3581	11738	4251	7419	4030	20519	2499	10300	2937	23500	4506

Tabla 3: Valores de la latencia máxima obtenidos en la ejecución de *roundtrip*.

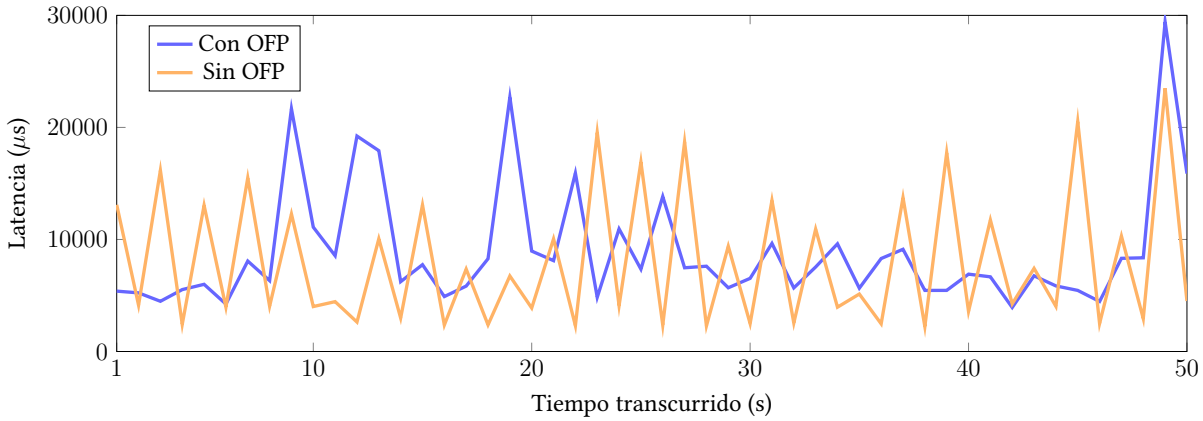


Figura 15: Evolución de la latencia máxima en la ejecución de *roundtrip*. Datos obtenidos de la tabla 3.

Estos datos pueden ofrecernos información realmente interesante para valorar. Primeramente, comparando esta figura 15 con la figura 13, vemos como estos máximos no han afectado en demasía a la mediana por lo que estos valores no han sido muy frecuentes. Una forma de considerar el ruido es comparar la diferencia entre los valores máximos y mínimos obtenidos, es decir, realizar la comparación de esta figura 15 con la figura 11 correspondiente a los valores mínimos. La diferencia es abismal, del orden de 23 segundos, lo que nos demuestra que estos valores son valores atípicos dados por el ruido.

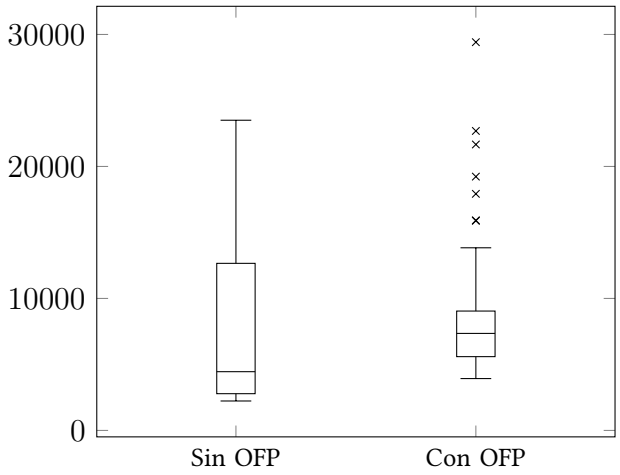


Figura 16: Diagrama de cajas y bigotes de la mediana de la latencia. Datos obtenidos de la tabla 3.

En la figura 16 se aprecia como con OFP existe una menor dispersión de los datos, aunque presenta varios valores atípicos realmente altos. No hay una diferencia extrema en las me-

dianas de los datos, por lo que aparentemente no existe una diferencia significativa de gran magnitud.

Tiempo transcurrido (s)		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Mensajes	**OFP**	1666	1649	1653	1637	1652	1591	1606	1606	1353	1530	1631	1504	1362
recibidos	**Sin OFP**	1518	1531	1504	1496	1446	1535	1428	1544	1477	1562	1515	1546	1401

Tiempo transcurrido (s)		**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
Mensajes	**OFP**	1655	1589	1633	1633	1597	1500	1547	1550	1506	1605	1594	1570	1504
recibidos	**Sin OFP**	1539	1499	1546	1556	1517	1523	1507	1495	1536	1447	1519	1515	1537

Tiempo transcurrido (s)		**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
Mensajes	**OFP**	1646	1571	1612	1552	1587	1566	1649	1604	1641	1535	1576	1636	1636
recibidos	**Sin OFP**	1491	1549	1478	1541	1487	1531	1465	1495	1528	1518	1502	1551	1412

Tiempo transcurrido (s)		**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**
Mensajes	**OFP**	1621	1645	1612	1666	1637	1650	1629	1651	1632	1146	1493
recibidos	**Sin OFP**	1504	1508	1503	1482	1511	1484	1512	1473	1563	1465	1503

Tabla 4: Número de mensajes recibidos en la ejecución de *roundtrip*.

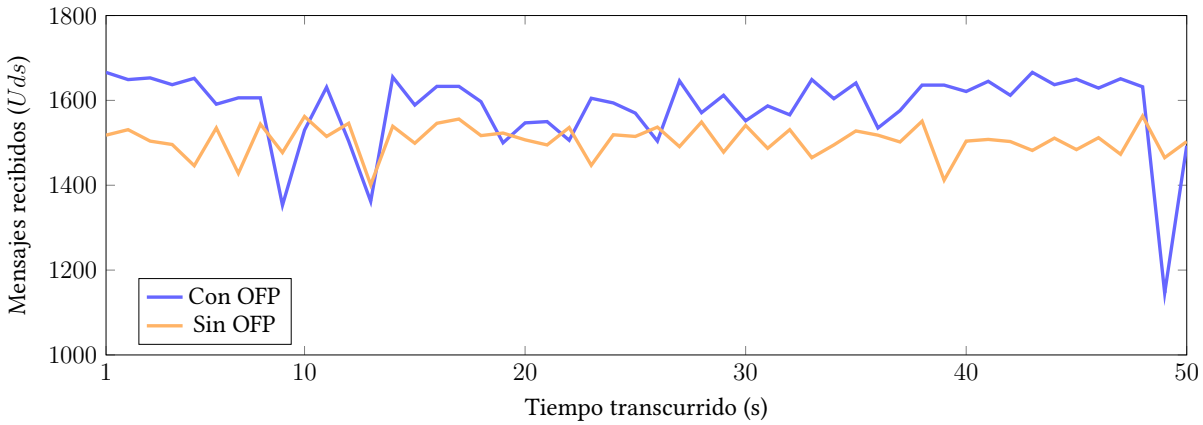


Figura 17: Evolución del número de mensajes recibidos en la ejecución de *roundtrip*. Datos obtenidos de la tabla 4.

Se puede observar que en la figura 17 con OFP se consigue tener un mayor número de mensajes enviados y recibidos, aunque presenta unas mayores bajadas fruto de la inestabilidad ocasional que puede presentar al necesitar de tanta potencia de procesamiento. El procesador, al no tener la suficiente refrigeración, hace caer su rendimiento para tratar de enfriar el chip.

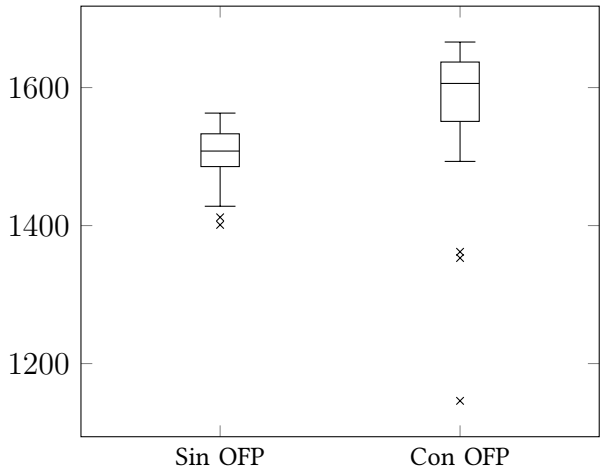


Figura 18: Diagrama de cajas y bigotes del número de mensajes recibidos. Datos obtenidos de la tabla 4.

En la figura 18 podemos observar como la mediana con OFP es mayor que la mediana sin OFP, sin embargo presenta una mayor dispersión con OFP y unos valores atípicos muy bajos que coinciden con lo comentado anteriormente: el uso de OFP da ciertas caídas por el uso intensivo del procesador.

5.2 *Throughput*

Otro parámetro interesante a estudiar es el *throughput*. Se trata de medir la cantidad de información que puede ser mandada o recibida desde un equipo a otro por unidad de tiempo, es decir, se mide la cantidad de bits que puede recorrer un sistema por segundo.

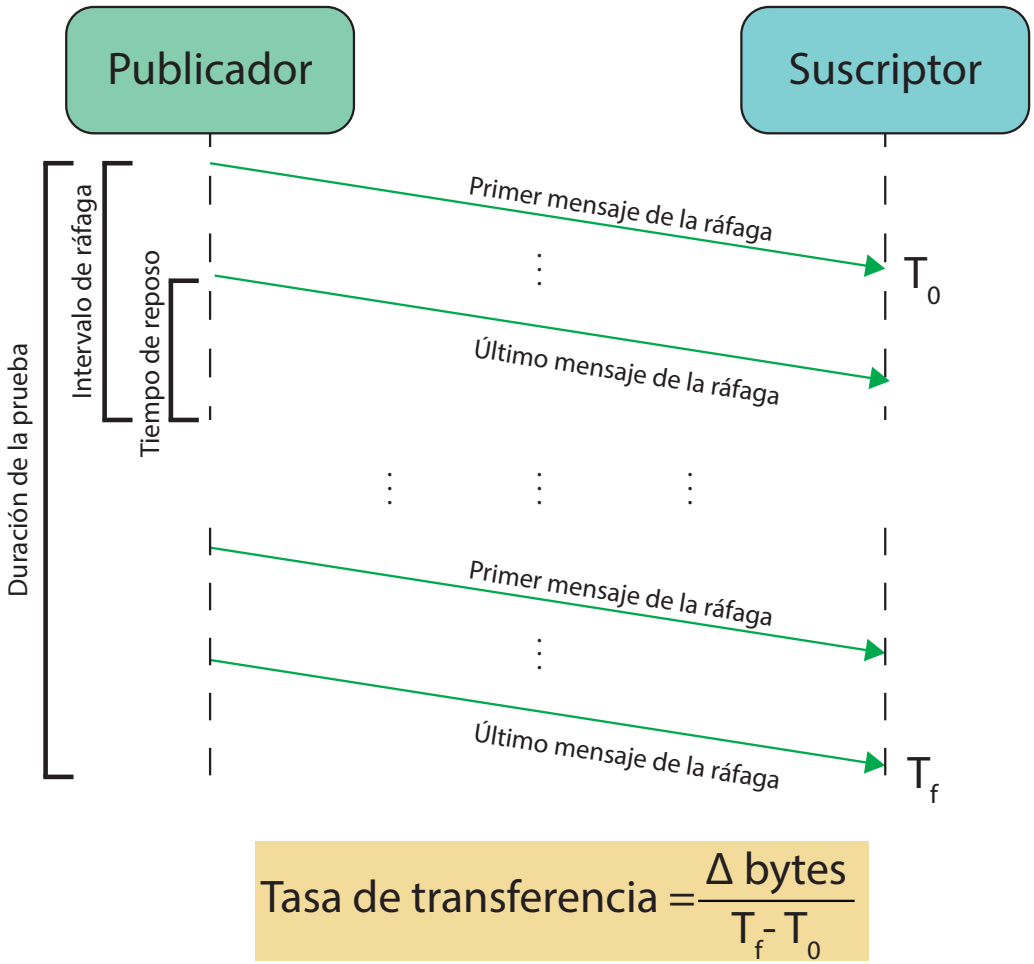


Figura 19: Representación de la medición del *throughput* en la prueba.
(Basado en la figura de [15])

Como se puede apreciar en la figura 19, el publicador manda un grupo de mensajes (una ráfaga) en un intervalo de tiempo. Al finalizar el envío, si la operación ha terminado antes del tiempo definido para el intervalo de la ráfaga, el sistema espera hasta cumplir el tiempo definido para el intervalo. En caso contrario, el publicador continúa con el envío. Esta operación se ejecuta durante un tiempo que será la duración total de la prueba. Conocido este tiempo, es posible calcular el número de bits (o también de mensajes) por segundo de manera trivial.

5.2.1 Metodología

La prueba se ha ejecutado con dos máquinas virtuales con la topología que se puede observar en la figura 8. Al igual que en el programa de la medición de la latencia (y en cualquier aplicación que haga uso de DDS) debemos crear un tópico:

```
topic = dds_create_topic (participant, &ThroughputModule_DataType_desc,
"Throughput", NULL, NULL);
```

Sobre este tópico que hemos llamado "Throughput" en el codigo lanzaremos los mensajes que tendrán el formato definido en la figura 20. La diferencia principal con la descripción proveída en la figura 9 es el añadido de un nuevo campo, *count*, un entero *long long* (para poder almacenar 64 bits en la variable) sin signo que llevará la cuenta de mensajes para detectar posibles mensajes que lleguen desordenados.

```
module ThroughputModule
{
  struct DataType
  {
    unsigned long long count;
    sequence<octet> payload;
  };
  #pragma keylist DataType
};
```

Figura 20: Fichero de la descripción del tópico para la ejecución de *throughput*.

La medición del *throughput* la realiza el suscriptor con el código de la figura 21, donde calcula el número de los valores del tópico recibidos por segundo y el número de Megabits por segundo que se han recibido.

```
Samples = (total_samples - prev_samples) / deltaTime);
Mbits = ((total_bytes - prev_bytes) / BYTES_PER_SEC_TO_MEGABITS_PER_SEC) / deltaTime);
```

Figura 21: Medición de la tasa de transferencia en el código.

Muestras	**OFP**	6	506.98	508.67	509	513.73	513	521.23	501.18	525	510	522.19	511.98	507.06
por segundo	**Sin OFP**	0.96	504	521	517	510.99	512.99	511	512	510	514	509	512	511
Muestras	**OFP**	515.58	512.34	506.73	512.19	511.61	507.72	506	584.96	569.47	628	609.74	619	610.02
por segundo	**Sin OFP**	508	495	503	500	502	502	502	504	503	504	509	500	510
Muestras	**OFP**	616.01	606	615.09	611.43	610.95	614.59	614.28	582	529.48	541.96	564.66	560	560
por segundo	**Sin OFP**	508	511	503	512	512	514	512	511	509	551.86	509	507	498
Muestras	**OFP**	562	566	581.62	568.62	623.68	613	609.46	617.73	624	613.65	647.97	639.15	621.68
por segundo	**Sin OFP**	502	502	501.99	499	502	500	502	502	501	495	522	529	532
Muestras	**OFP**	621	631.02	621.21	620	613.83	644.03	628	623.02	626.64	586.93	534.68	550	551.86
por segundo	**Sin OFP**	531	533	530	532	531	532	529	532	512	502	497	503	501

Tabla 5: Número de muestras recibidas por segundo en la ejecución de *throughput*.

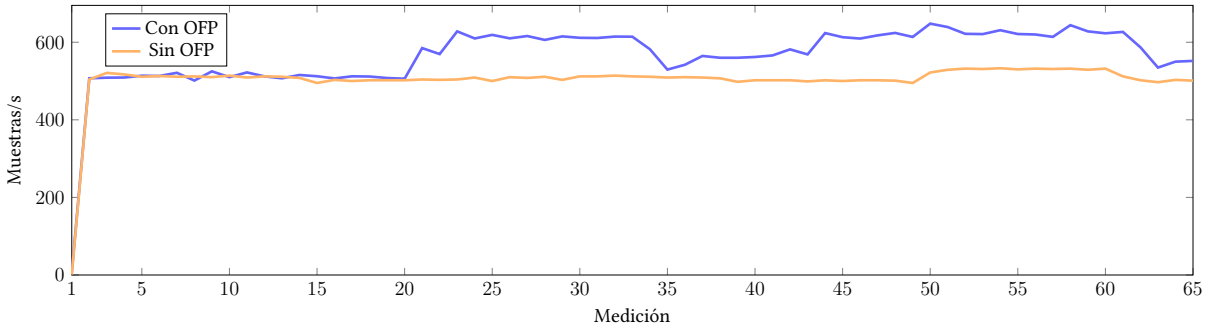


Figura 22: Evolución del muestreo por segundo la ejecución de *throughput*. Datos obtenidos de la tabla 5.

Como se puede apreciar en la figura 22, hay una ligera variación con OFP dándose unos mayores valores en ciertos intervalos de tiempo. Aunque la tendencia en la figura parece alcista, ambas implementaciones siguen la ejecución presentando los mismos valores, con una ligera variación con OFP ofreciendo una pequeña ventaja.

Throughput	**OFP**	0.00	3.98	3.97	4.00	3.98	4.00	3.92	3.93	4.00	4.00	3.97	3.99	3.99
(Mbit/s)	**Sin OFP**	0	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Throughput	**OFP**	3.98	3.99	3.98	3.98	3.98	3.99	4.00	3.95	4.00	5.00	4.96	5.00	4.95
(Mbit/s)	**Sin OFP**	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Throughput	**OFP**	4.94	5.00	4.98	4.95	4.98	4.95	4.96	4.00	3.99	3.96	4.00	4.00	4.00
(Mbit/s)	**Sin OFP**	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Throughput	**OFP**	4.00	4.00	3.98	3.92	4.96	5.00	4.98	4.99	5.00	4.97	4.93	4.93	4.97
(Mbit/s)	**Sin OFP**	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Throughput	**OFP**	5.00	4.98	4.95	5.00	4.98	4.95	5.00	4.97	5.00	4.00	3.95	4.00	3.97
(Mbit/s)	**Sin OFP**	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Tabla 6: Valores de la tasa de transferencia en la ejecución de *throughput*.

Como se puede apreciar en la figura 23, al igual que en la figura 22, en ciertos intervalos de tiempo OFP alcanza una mayor tasa de transferencia, aunque el resto del tiempo están igualados.

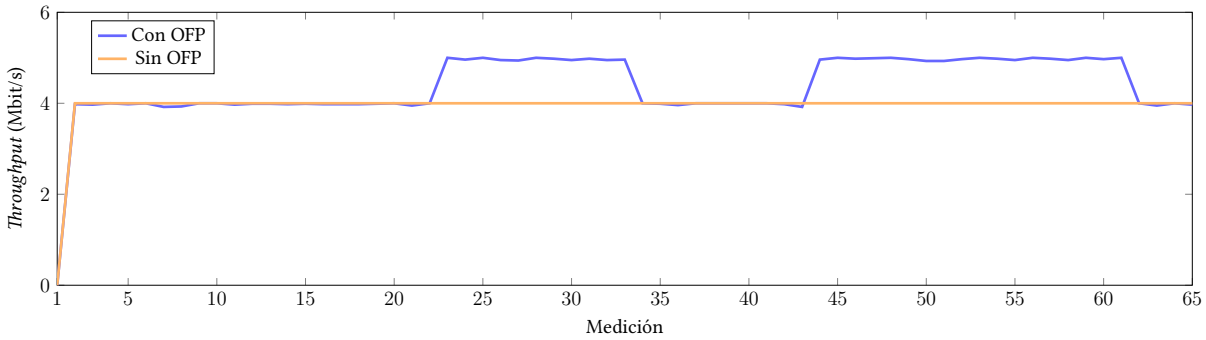


Figura 23: Evolución del *throughput* en la ejecución. Datos obtenidos de la tabla 6.

5.3 Valoración global

Como hemos podido ver en los apartados anteriores, se aprecia una mejoría en ciertas situaciones con OFP a pesar de perder rendimiento por consumir demasiado procesador. Las mejoras de OFP vienen dadas por el uso de nuevas técnicas modernas para manejar *sockets* como la utilización de PACKET_MMAP. Esta interfaz provee un búfer circular desde el espacio de usuario para mandar y recibir paquetes de manera eficiente. De esta forma, la lectura de los paquetes se realiza mediante esperas sin la necesidad de realizar llamadas del sistema, lo que permite alcanzar la mejoría obtenida gracias a minimizar la copia de paquetes.

OFP también se encarga de reemplazar las llamadas a `sendmsg()` por llamadas a `sendmmsg()`, que permite enviar múltiples mensajes por un *socket*, lo que permite un mayor rendimiento en ciertas situaciones.

Estos datos podrían verse mejorados si en vez de utilizar ODP con PACKET_MMAP, se hubiera utilizado la capa que ofrece ODP para la utilización de DPDK. De esta manera, habría un puenteo real del núcleo del sistema operativo para las operaciones de red lo que permitiría que el espacio de usuario pudiera tener un mayor acceso a los paquetes, acelerando aún más las funciones de red.

6

Conclusiones y Líneas Futuras

6.1 Conclusiones

Este trabajo fin de grado ha permitido obtener una herramienta capaz de acelerar tanto implementaciones del *middleware* DDS como otras aplicaciones que hacen uso del protocolo UDP, ya que gran parte de las llamadas necesarias para hacer uso de protocolo de nivel de transporte no estaban implementadas en el envoltorio para utilizar con LD_Preload de la librería OFP. Se ha comprobado cómo la aplicación de la herramienta a una implementación popular como CycloneDDS le ha atribuido grandes beneficios a la hora de reducir su latencia, un componente clave que con su reducción permite muchas posibilidades en casos de uso de la vida real: con una simple reducción de la latencia, un vehículo autónomo que haga uso de DDS podría evitar un accidente o un dron podría ser controlado de una forma más suave.

Desde el plano personal, el trabajo ha supuesto un gran reto al tener que afrontar un proyecto complejo donde entran en juego componente de bajo y alto nivel, conocer el núcleo de Linux y el funcionamiento de sus componentes de red y el uso del lenguaje de programación C en *software* crítico, así como distintas herramientas para poder depurar código con este lenguaje o poder navegar con facilidad por proyectos con una gran cantidad de código. He complementado diversos aspectos de la carrera al dar unos pasos más allá estudiando a fondo cómo funcionan las llamadas de red y la posibilidad de evitar el núcleo del sistema operativo para evitar interrupciones, y también me ha permitido poner en práctica cómo abordar proyectos más complejos gracias a la posibilidad de utilizar y modificar proyectos de código abierto con una gran cantidad de funciones y aplicaciones.

6.2 Líneas Futuras

En este apartado se comentan distintas mejoras futuras que pueden realizarse sobre el trabajo realizado, permitiendo así obtener un resultado más pulido y estudiar algunas otras opciones para comparar con los resultados obtenidos:

- **Reescribir las funciones para tomar interfaces de red de CycloneDDS**. De esta forma, se podría intentar utilizar la librería DPDK con OFP, pudiendo así mejorar incluso los datos obtenidos con la implementación genérica de ODP y OFP.
- **Comparar las distintas opciones de PKTIO de ODP**. La implementación genérica de ODP ofrece distintas opciones para la entrada y salida de paquetes, incluyendo el uso de DPDK para únicamente la entrada y salida de paquetes, por lo que podrían evitarse los fallos en los temporizadores de esta forma. También se ofrece la opción de usar netmap, otro *framework* que permite entrada y salida de paquetes desde el espacio de usuario.
- **Realizar distintas pruebas con distintas configuraciones de CycloneDDS**. CycloneDDS ofrece distintos parámetros para afinar su configuración y así permitir un mayor rendimiento. Sería interesante tratar de encontrar la configuración adecuada para su uso con OFP.
- **Probar otras librerías y comparar su rendimiento con OFP**. Existen algunas otras librerías como es el caso de VPP (*Vector Packet Processing*) que ofrecen un procesamiento de paquetes de alto rendimiento. Estudiar los beneficios que ofrece cada una a un *middleware* como DDS puede ser realmente interesante.

Bibliografía

[1] «IEEE Standard for Ethernet», *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, págs. 22-23, 2018. DOI: 10.1109/IEEESTD.2018.8457469.

[2] A. Corsaro, «The data distribution service tutorial», *Technical Report 4.0*, 2014.

[3] DDS-Foundation, *What is DDS?*, consultado el 22/07/2021. dirección: https://www.dds-foundation.org/what-is-dds-3/.

[4] J. van't Hag, «"Data-centric to the max", the SPLICE architecture experience», en *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, 2003, págs. 207-212. DOI: 10.1109/ICDCSW.2003.1203556.

[5] ——, *OpenSplice Overview White Paper*, consultado el 22/07/2021. dirección: https://web.archive.org/web/20210722154602/https://docs.huihoo.com/opensplice/OpenSplice-White-Paper.pdf.

[6] B. W. Kernighan y D. M. Ritchie, *The C programming language*. Englewood Cliffs, N.J: Prentice Hall, 1988, pág. xi, ISBN: 978-0131103627.

[7] DPDK, *Architecture overview*, consultado el 22/07/2021. dirección: https://doc.dpdk.org/guides/prog_guide/overview.html.

[8] OpenDataPlane, *API Reference Manual Documentation*, consultado el 22/07/2021. dirección: https://opendataplane.github.io/odp/.

[9] B. Pricope, *OpenFastPathGen2*, consultado el 22/07/2021. dirección: https://github.com/bogdanPricope/ofp.

[10] D. Snowden y col., «Liberating knowledge», *Liberating Knowledge CBI Business Guide*, págs. 9-19, 1999.

[11] K. Schwaber y J. Sutherland, «The scrum guide», *Scrum Alliance*, vol. 21, n.º 19, pág. 1, 2011.

[12] IBM-Corporation, *IP Sockets Application Programming Interface Guide and Reference*, consultado el 22/07/2021. dirección: https://www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R5sc273660/$file/hala001_v2r5.pdf.

[13] L. Lai, *How to port complex application to f-stack*, consultado el 22/07/2021. dirección: https://github.com/F-Stack/f-stack/issues/546.

[14] B. Pricope, *ODP Timers starting on control threads don't 'fire'*, consultado el 22/07/2021. dirección: https://github.com/OpenDataPlane/odp-dpdk/issues/119.

[15] Eprosima, *Benchmarking DDS Implementations*, consultado el 08/09/2021. dirección: https://www.eprosima.com/index.php/resources-all/performance/fast-dds-vs-cyclone-dds-performance.

Apéndice A
Manual de Instalación de DPDK

Para instalar y poner en funcionamiento DPDK en una distribución GNU/Linux basada en Debian como Ubuntu se deben realizar ciertos pasos detallados en este mismo apéndice. Los siguientes pasos se han realizado en la última versión LTS (*Long Term Support*) de Ubuntu, que hasta este momento es la versión 20.04.1. Sin embargo, los comandos son fácilmente trasladables a cualquier otro tipo de distribución teniendo en cuenta las posibles diferencias en el gestor de paquetes o la localización de ciertos archivos de configuración. Se tratará de detallar todos los comandos para poder guiar de una forma más sencilla a cualquier persona que requiera instalar DPDK en otra distribución.

Paso 1:

Instalar las distintas dependencias haciendo uso del gestor de paquetes, en el caso de Ubuntu `apt-get`.

```
sudo apt-get install -y \ # Aceptar automaticamente los distintos mensajes
librdmacm-dev librdmacm1 \ # Paquetes para acceso remoto directo a memoria
build-essential linux-headers-`uname -r` \ # Utilidades para la compilación
libnuma-dev \ # Paquete para el acceso a memoria no uniforme
libmnl-dev \ #  Paquete para tratar con la interfaz Netlink
meson #       Utilidad para automatizar la compilación
```

Paso 2:

Habilitar tanto las *hugepages* como IOMMU (*input–output memory management unit*) en el fichero de arranque. Para ello, con un editor de textos (por ejemplo, vim (*Vi IMproved*)) se abrirá el fichero `/etc/default/grub`

```
sudo vim /etc/default/grub
```

Para añadir las siguientes opciones al final del valor de `GRUB_CMDLINE_LINUX_DEFAULT`

```
GRUB_CMDLINE_LINUX_DEFAULT="... default_hugepagesz=1G
↪  hugepagesz=1G hugepages=4 intel_iommu=on iommu=pt"
```

Paso 3:

Una vez guardado el fichero, actualizar GRUB haciendo uso del siguiente comando

```
sudo grub-mkconfig -o /boot/grub/grub.cfg
```

Paso 4:

Reiniciar el sistema, ya sea mediante el uso de la interfaz gráfica o ejecutando el comando `sudo shutdown -r now`

Paso 5:

Para que DPDK tenga acceso a la memoria de *hugepage* reservada, ejecutamos los siguientes comandos

```
sudo cp /etc/fstab /etc/fstab.orig
sudo mkdir -p /mnt/huge_1GB
echo "nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0" | sudo tee -a /etc/fstab
sudo mount -fav
```

Paso 6:

Descargar los ficheros fuentes de DPDK. Accediendo a la página web https://core.dpdk.org/download/ desde un navegador podemos ver la última versión y la versión LTS. Si queremos descargarlo desde la consola de comandos, podemos utilizar `wget` de la siguiente manera (en este caso, descargando la versión 21.02):

```
wget https://fast.dpdk.org/rel/dpdk-21.02.tar.xz
```

Paso 7:

Extraer el fichero descargado y abrir el directorio.

```
tar xvf dpdk-21.02.tar.xz
cd dpdk-21.02
```

Paso 8:

Instalar DPDK a nivel de sistema con los siguientes comandos:

```
meson -Dexamples=all build -Ddefault_library=shared
cd build
meson configure  -Dprefix=$(pwd)/../install
ninja
ninja install
sudo ldconfig # Actualiza la caché de ld.so
```

Paso 9:

Activar el módulo del núcleo de Linux VFIO

```
modprobe vfio-pci
```

Paso 10:

Dar permisos de VFIO para usuarios normales

```
sudo chmod a+x /dev/vfio
sudo chmod 0666 /dev/vfio/*
```

Paso 11:

Hacer que todos los dispositivos que se encuentren en el mismo grupo IOMMU en el que se encuentren las tarjetas de red que queremos utilizar con DPDK utilicen el driver VFIO. Para ello, hay que identificar el grupo IOMMU de la tarjeta de red. Primero, mediante el comando `lspci -v` es necesario buscar en la salida la tarjeta de red, obteniendo algo parecido a lo siguiente.

```
02:00.0 Ethernet controller: Intel Corporation 82545EM Ethernet...
```

En este caso, el identificador del dispositivo es 02:00.0. Con el identificador obtenido, ejecutamos el comando `readlink` de la siguiente manera

```
readlink /sys/bus/pci/devices/0000:02:00.0/iommu_group
```

Obteniendo una salida parecida a

```
../../../../kernel/iommu_groups/5
```

Obteniendo en este caso que el grupo IOMMU del que queremos hacer que todos sus dispositivos usen el driver VFIO es el 5. Para hacer ver todos los dispositivos que se encuentran en el grupo de la tarjeta, es posible utilizar el comando:

```
ls -l /sys/bus/pci/devices/0000:02:00.0/iommu_group/devices
```

Con el que se obtiene una salida similar a la siguiente:

```
../../../../devices/pci0000:00/0000:00:11.0
../../../../devices/pci0000:00/0000:00:11.0/0000:02:00.0
../../../../devices/pci0000:00/0000:00:11.0/0000:02:01.0
../../../../devices/pci0000:00/0000:00:11.0/0000:02:02.0
../../../../devices/pci0000:00/0000:00:11.0/0000:02:04.0
../../../../devices/pci0000:00/0000:00:11.0/0000:02:05.0
```

En este caso, son 5 los dispositivos que necesitan utilizar el driver VFIO (02:00.0, 02:01.0, 02:02.0, 02:04.0 y 02:05.0). Se debe desvincular cada uno de los dispositivos del driver actual para vincularlos con VFIO. Para realizar esta operación, es necesario obtener el identificador de cada uno de los dispositivos.

Tomando como ejemplo el primer dispositivo anterior, 0000:02:00.0, se ejecutará el siguiente comando para cada uno de los dispositivos:

```
lspci -ns 0000:02:00.0
```

Obteniendo una salida similar a

```
02:00.0 0200: 8086:100f (rev 01)
```

En este caso, el identificador es 8086:100f. Con el identificador obtenido, se ejecuta la siguiente secuencia de comandos para desvincular del driver actual y vincular al driver VFIO respectivamente

```
echo 0000:02:00.0 > /sys/bus/pci/devices/0000:02:01.0/driver/unbind
echo 8086 100f > /sys/bus/pci/drivers/vfio-pci/new_id
```

Se debe realizar este procedimiento para cada uno de los dispositivos que se encuentran en el mismo grupo IOMMU. Sin embargo, hay otra opción que puede resultar más sencilla al automatizar todo este procedimiento: utilizar el script `vfio-pci-bind` del usuario *andre-richter* en GitHub, disponible en la siguiente dirección web: https://github.com/andre-richter/vfio-pci-bind. Este script nos permite realizar todo el proceso anterior suministrándole únicamente un identificador (ya sea de la forma *Dominio:Bus:Dispositivo.Función* o *Vendedor:Dispositivo*), por lo que facilita el proceso anterior enormemente. Si git no se encuentra instalado, se puede instalar con cualquier gestor de paquetes. En el caso de apt, es suficiente con ejecutar `sudo apt install git`. Una vez instalado, se clona el repositorio y se ejecuta el script de la siguiente manera:

```
git clone https://github.com/andre-richter/vfio-pci-bind
cd vfio-pci-bind-master
sudo ./vfio-pci-bind.sh 0000:02:00.0
```

Con la ejecución del script, ya tendremos todos los dispositivos vinculados al driver VFIO. La vinculación no es persistente, por lo que es necesario configurar la ejecución del script en el arranque o, de forma alternativa, ejecutar los distintos comandos en cada arranque.

Paso 12:

Probar el correcto funcionamiento de DPDK mediante la herramienta testpmd. Con el siguiente comando es posible ejecutar la prueba de reenvío de paquetes.

```
dpdk-testpmd -l 0-1  -- --port-topology=loop \
--auto-start --tx-first --stats-period=3
```

Como apunte, es relevante comentar que si queremos ejecutar DPDK en una máquina virtual con VMware debemos activar dos opciones del motor de virtualización: Virtualizar Intel VT-X/EPT o AMD-V/RVI y Virtualizar IOMMU, tal y como se aprecia en la figura 24.

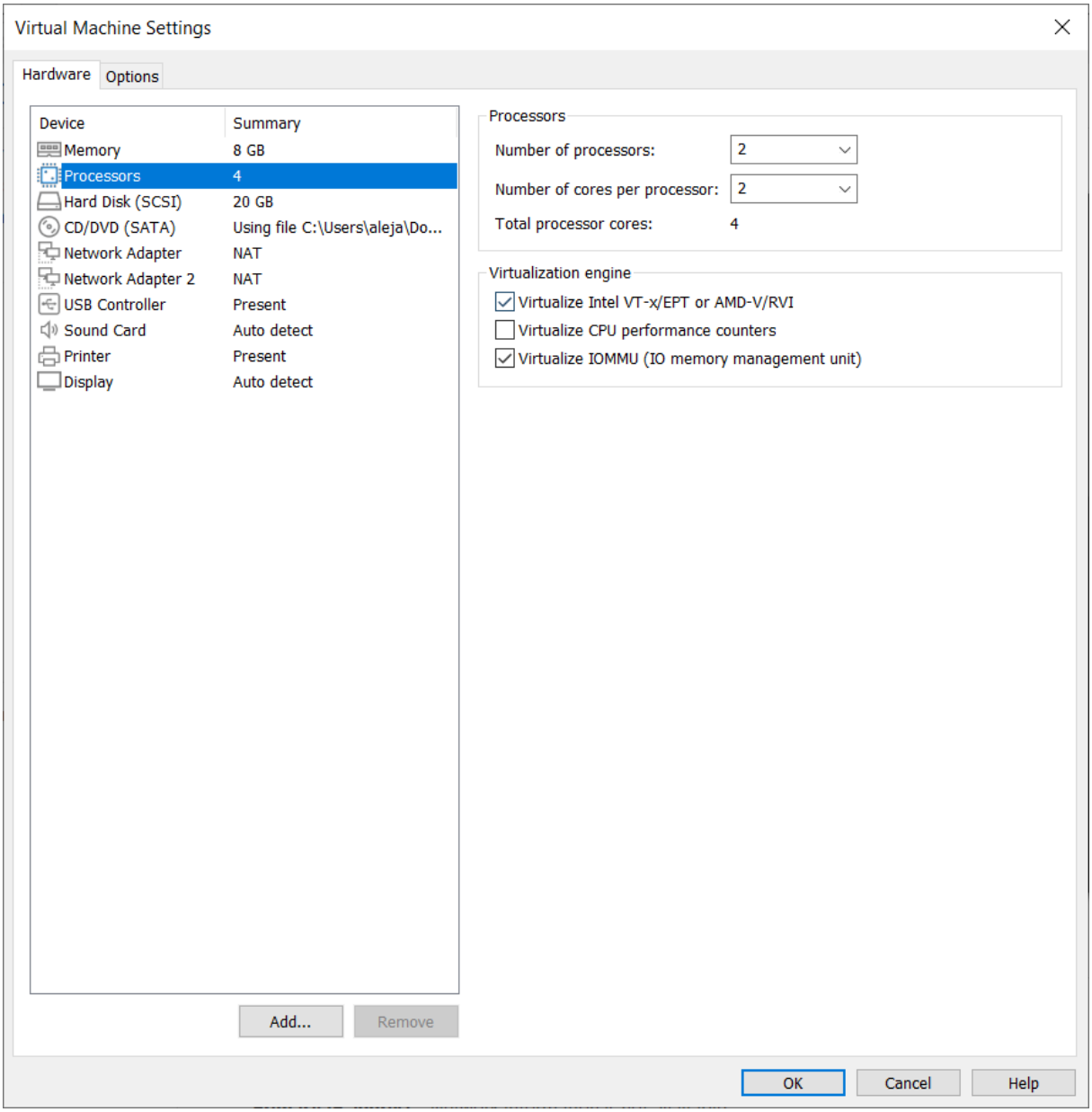


Figura 24: Opciones a configurar necesarias para ejecutar DPDK con VMware.

Apéndice B
Manual de Instalación de ODP

Para la instalación de ODP debemos tener en cuenta qué repositorio queremos usar, ya que tal y como se explicó en el capítulo 2 existen dos repositorios distintos: uno que hace uso de DPDK y otro que no lo necesita. En los *issues* de los repositorios en GitHub se recomienda por parte de los programadores el uso de la versión sin DPDK para comenzar a programar cualquier aplicación que haga uso de ODP para descartar fallos provenientes de DPDK. En este apéndice se verán los pasos a tomar para instalar cada una de las versiones.

· Sin DPDK

La dirección del repositorio de la implementación genérica de ODP es la siguiente: https://github.com/OpenDataPlane/odp. Actualmente por la eliminación de una variable de estadísticas que utiliza OFP no es posible compilar OFP con la última versión de ODP, por lo que si queremos usar ODP con OFP es recomendable el uso de la versión 1.26 para tener una mayor estabilidad. OFP (por ahora) no se beneficia de las nuevas características de ODP, por lo que no hay problema en usar una versión anterior. Es por ello que para la ejecución del código de este trabajo fin de grado se recomienda el uso de esta versión.

Paso 1:

Instalar las distintas dependencias haciendo uso del gestor de paquetes, en el caso de Ubuntu `apt-get`

```
sudo apt-get install -y \
build-essentials \ # Paquetes para la compilación
asciidoctor \ # Generación de documentación
```

```
autoconf \ # Scripts de configuración
automake \ # Automatiza la compilación
ccache \ # Caché para la compilación
clang \ # Front end de compilador
git \ # Gestión de versiones
graphviz \ # Gráficos para la documentación
libconfig-dev \ # Procesamiento de configuración
libcunit1-dev \ # Tests unitarios
libibverbs-dev \ # Habilita el acceso remoto directo a memoria
net-tools \ # Utilidades de red
python3-pip \ # Gestor de módulos de Python
ruby-dev \ # Lenguaje de programación Ruby
xsltproc && \ # Aplicar hojas de estilo a XML
pip3 install coverage # Cobertura de código
```

Paso 2:

Clonar la versión 1.26 de ODP y situarnos en el directorio.

```
git clone --depth=1 -b v1.26.0.0 https://github.com/OpenDataPlane/odp.git odp
cd odp
```

Paso 3:

Realizar la configuración para la instalación.

```
./bootstrap
```

Si queremos una instalación normal:

```
./configure --prefix=`pwd`/install # Se instalará en la carpeta "install"
```

Si queremos una instalación con posibilidad de depuración:

```
./configure CFLAGS="-g -O0" --prefix=`pwd`/install --enable-debug
```

Paso 4:

Instalar ODP

```
make -j install
```

· Con DPDK

La dirección del repositorio de la implementación de ODP para su uso con DPDK es la siguiente: https://github.com/OpenDataPlane/odp-dpdk. La versión recomendada en este caso por los motivos comentados anteriormente es la 1.25.2.0

Paso 1:

Instalación de dependencias según el paso 1 de los pasos para la instalación sin DPDK.

Paso 2:

Clonar el repositorio y situarnos en el directorio:

```
git clone --depth=1 -b v1.25.2.0_DPDK_19.11 \
        https://github.com/OpenDataPlane/odp-dpdk odp-dpdk
cd odp-dpdk
```

Paso 3:

Configuración de la instalación.

```
./bootstrap
```

Si hemos compilado DPDK con el flag -fPIC como librería compartida:

```
./configure --with-dpdk-path=<dpdk-dir>/install
```

Si hemos compilado DPDK como librería estática para un mayor rendimiento:

```
./configure --with-dpdk-path=<dpdk-dir>/install --disable-shared
```

Paso 4:

Instalar ODP-DPDK

```
make -j install
```

Apéndice C
Manual de Instalación de OFP

Aunque el repositorio oficial se encuentra en la dirección https://github.com/OpenFastPath/ofp y es la base que se ha tomado para la realización de este trabajo, se recomienda encarecidamente para futuros proyectos el uso del *fork* de Bogdan Pricope: https://github.com/bogdanPricope/ofp, ya que tiene un mayor ritmo de actualizaciones.

Paso 1:

Instalación de dependencias. Si se han instalado los paquetes necesarios para ODP del paso 1 del apéndice B, solo es necesario instalar los siguientes paquetes:

```
sudo apt-get install -y \
        libtool \ # Creación de librerías portables
        pkg-config \ # Herramienta para obtener configuración de librerías
        doxygen \ # Generar la documentación
        valgrind # Depuración de memoria
```

Paso 2:

Clonar la última versión de OFP (o extraer los ficheros fuentes de la modificación realizada en este trabajo) y situarnos en el directorio.

```
git clone https://github.com/OpenFastPath/ofp.git ofp
cd ofp
```

Paso 3:

Realizar la configuración para la instalación

```
./bootstrap
```

Si queremos usar OFP sin hacer uso de LD_Preload:

```
./configure --prefix=`pwd`/install --with-odp=<DIRECTORIO_INSTALACIÓN_ODP>
```

Si queremos usar OFP con LD_Preload:

```
./configure --enable-sp=no --with-config-flv=netwrap-webserver \
--with-odp=<DIRECTORIO_INSTALACIÓN_ODP> --prefix=`pwd`/install
```

Paso 4:

Instalar OFP

```
make -j install
```

Apéndice D
Manual de Instalación de CycloneDDS

Para instalar CycloneDDS debemos realizar los siguientes pasos:

Paso 1:

Instalar las dependencias necesarias. Si hemos seguido los apéndices B y C, solamente deberemos instalar GNU Bison (un generador de analizadores sintácticos) y CMake (herramienta para compilar) con el siguiente comando:

```
sudo apt-get install -y bison cmake
```

Paso 2:

Clonar el repositorio (o extraer los ficheros fuentes de la modificación realizada en este trabajo) y situarnos en el directorio:

```
git clone https://github.com/eclipse-cyclonedds/cyclonedds.git cyclonedds
cd cyclonedds
```

Paso 3:

Crear el directorio de la compilación y compilar

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=`pwd`/install -DBUILD_EXAMPLES=ON
cmake --build .
```

Paso 4:

Instalar

```
cmake --build . --target install
```

Apéndice F
Ejecución de CycloneDDS con OFP

Para ejecutar CycloneDDS y OFP con las modificaciones realizadas se deben realizar unos pasos extra que se explican en este apéndice.

Paso 1:

Habilitar las *hugepages* ejecutando el siguiente comando:

```
sysctl -w vm.nr_hugepages=2048
```

O ejecutar los comandos relacionados con las *hugepages* paso 3 del apéndice A.

Paso 2:

Crear un fichero de configuración de CycloneDDS y cargarlo. En el fichero de configuración debemos configurar la IP que le daremos a la interfaz de OFP que vamos a utilizar, y las direcciones IP del resto de máquinas con las que nos queremos conectar, ya que por limitaciones de OFP no funciona la multidifusión. El fichero será de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8" ?>
<CycloneDDS
        xmlns="https://cdds.io/config"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://cdds.io/config">
    <Domain id="any">
        <General>
            <NetworkInterfaceAddress>192.168.43.135</NetworkInterfaceAddress>
```

```
        <AllowMulticast>false</AllowMulticast>
        </General>
        <Discovery>
            <ParticipantIndex>auto</ParticipantIndex>
            <Peers>
              <Peer Address="192.168.43.136:7410" />
            </Peers>
        </Discovery>
    </Domain>
</CycloneDDS>
```

Donde la dirección 192.168.43.135 es la dirección que tomaremos para la interfaz de OFP y 192.168.43.136 es la dirección del otro participante, como el caso de la figura 8. Para cargarlo al ejecutar CycloneDDS, debemos ejecutar el siguiente comando (suponiendo que el estamos en el directorio donde hemos guardado el fichero y el fichero se llama "cyclonedds.xml")

```
export CYCLONEDDS_URI=file://$PWD/cyclonedds.xml
```

Paso 3:

Configurar la variable de entorno necesaria para cargar OFP.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<DIRECTORIO_INSTALACIÓN_OFP>/lib/
```

Paso 4:

Ejecutar el *script* de OFP para tomar la interfaz de red para usarla con OFP.

```
<DIRECTORIO_OFP>/scripts/ofp_linux_interface_acquire.sh \
 <NOMBRE_DE_LA_INTERFAZ>
```

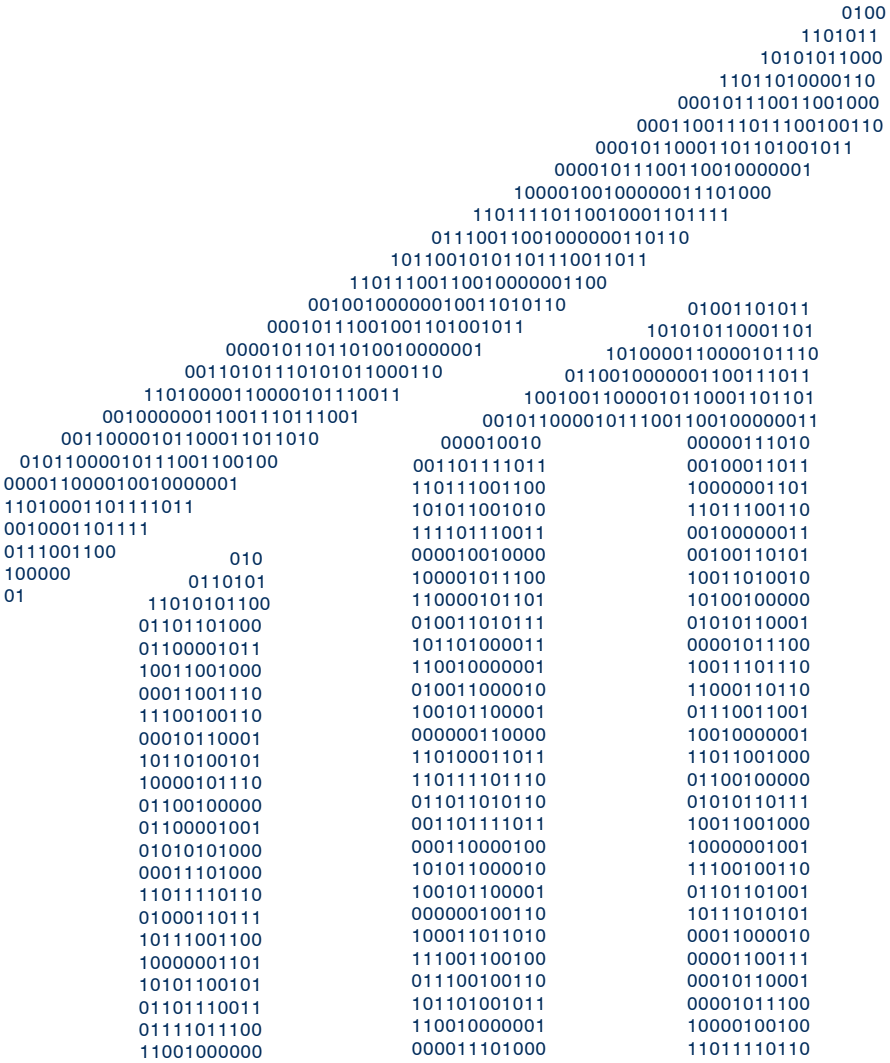
Paso 5:

Configurar los *scripts* de OFP para su uso con nuestra configuración. En el *script* ofp_netwrap.sh debemos sustituir "eth1" por el nombre de nuestra interfaz. En el fichero "ofp_netwrap.cli" debemos sustituir la IP que aparece por la que queramos utilizar para nuestra interfaz. Ambos ficheros se encuentran en la carpeta "scripts".

Paso 6:

Ejecutar el *script* de OFP para usar LD_Preload. Si queremos ejecutar el publicador del ejemplo de Hola mundo" de CycloneDDS, el comando sería el siguiente:

```
. <DIRECTORIO_OFP>/scripts/ofp_netwrap.sh HelloworldPublisher
```



UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga