MÁSTER EN INGENIERÍA DE SISTEMAS Y CONTROL

Proyecto fin de máster:

Estudio de plataforma de comunicaciones para sistemas de medición científica basado en lógica reconfigurable

Trabajo para la obtención del máster en sistemas y control en el curso académico 2018–2019, convocatoria de Junio.

Realizado por: José Antonio de la Torre las Heras
Dirigido por:
José Sánchez Moreno
Fernando Rincón Calle

MÁSTER EN INGENIERÍA DE SISTEMAS Y CONTROL

Proyecto fin de máster de tipo B (Proyecto específico propuesto por el alumno):

Estudio de plataforma de comunicaciones para sistemas de medición científica basado en lógica reconfigurable

Trabajo para la obtención del máster en sistemas y control en el curso académico 2018–2019.

Realizado por: José Antonio de la Torre las Heras
Dirigido por:
José Sánchez Moreno
Fernando Rincón Calle

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado: José Antonio de la Torre las Heras (8 de junio de 2018)

Resumen

En los últimos años los sistemas de medición científica han ido evolucionando siguiendo la estela de las tecnologías de la información. Lejos quedan los sistemas de medición aislados donde el operario debía operar manualmente cada uno de los aspectos del instrumento. Fruto de esta evolución han surgido nuevas necesidades que hasta ahora no se habían contemplado. Un ejemplo es la necesidad de sistemas de comunicación eficientes para dar soporte a las grandes cantidades de datos generados en el . Por esta razón, en este trabajo se analizarán las alternativas actuales a la hora de comunicarse con los instrumentos de medición y se tratará de caracterizar los principales puntos de mejora. Seguidamente, se implementa un middleware de comunicaciones basado en objetos distribuidos y se evaluará su eficiencia en un entorno simulado mediante iperf.
Palabras clave— fpga; soc; zynq; xilinx; middleware; tcp; udp; iperf

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Siglas

ACK Acknowledgement. 86, 142

ADC Analog Digital Converter. 9, 11, 12, 18, 23

API Application Programming Interface. 21–23, 32, 36, 43, 44, 47, 53, 94, 101, 109, 112, 116, 124

ASIC Aplication-Specific Integrated Circuit. 15

BSP Board Support Hardware. 5, 6, 72, 73, 85, 94, 101, 102, 111, 139, 140, 152

CAD Computer-Aided Design. 15, 20

CAN Controller Area Network. 124, 125

CPLD Complex Programmable Logic Device. 14

CPU Central Processing Unit. 15

CRC Cyclic Redundancy Check. 71

CSS Cascade Style Selector. 25, 26

DAC Digital Analog Converter. 19, 73

DARPA Defense Advanced Research Projects Agency. 27

DDR Double Data Rate. 20, 69, 82, 86, 95

DHCP Dynamic Host Configuration Protocol. 75

DMA Direct Memory Access. 69, 85, 86, 95

DSP Digital Signal Proccesor. 15

FPGA Field Programmable Gate Array. 12–14, 16–19, 22, 23, 35, 46, 65, 157

FPLA Field Programmable Logic Array. 13

FPU Floating Point Unit. 19

GPL General Public License. 38

GPU Graphical Processing Unit. 15

HDF Hardware Definition Format. 73, 93

HLS High Level Synthesis. 16

HTML HiperText Markup Language. 25, 26

HTTP Hiper Textual Transfer Protocol. 27, 28

IDL Interface Definition Language. 120, 138, 150

IETF Internet Engineering Task Force. 47

INFN Istituto Nazionale di Fisica Nucleare. 3

IOT Internet Of Things. 60

IP Internet Protocol. 30, 75, 77, 103, 134

IPC Inter Process Communication. 53

ISO International Organization For Standardization. 27

ISR Interrupt Software Rutine. 78

JS Javascript. 26

JSON Javascript Serializable Object Notation. 26

LWIP LightWeight IP. 73, 75–77, 94, 101, 102, 109, 110, 112, 113

MAC Medium Access Controller. 5, 68, 70, 71, 76, 85, 100, 124, 158

MIT Massachusetts Institute of Technology. 40

MPGA Mask-Programmable Gate Array. 14

MTU Maximum Transfer Unit. 68, 76

ORB Object Request Broker. 49

OSADL Open Source Automation Development Lab. 40

OSI Open System Interconnect. 4, 27, 28, 30, 75, 77, 103

OTA Over The Air. 60

PAL Programmable Array Logic. 13, 14

PCB Protocol Control Block. 77, 94, 103

PL Programable Logic. 68, 69

PROM Programmable Read Only Memory. 13

RAM Random Access Memory. 119

RAW . 12, 101, 109, 110, 116

RFC Request For Comments. 27, 119

RPC Remove Procedure Call. 60

RTD Resistence Temperature Sensor. 7

SCPI Standard Commands for Programmable Instruments. 27

SD Secure Digital card. 20, 128

SDK Software Development Kit. 72, 82, 139

SIMD Single Instruction Multiple Data. 19

Capítulo 1

Introducción

1.1. Introducción al trabajo

En este trabajo se realiza un estudio del arte de cómo han evolucionado los sistemas de medición desde la lógica analógica hasta los sistemas actuales basadas en tecnologías SoC y con una gran conectividad.

En primer lugar se realizará un estudio del arte donde el lector podrá orientarse sobre la trayectoria que han seguido los diferentes elementos que intervienen en el proyecto. El objetivo de este recorrido por las diferentes tecnologías es mostrar la necesidad y la contribución que busca realizar el trabajo.

Una vez estudiadas las diferentes tecnologías, en la siguiente sección el lector podrá comprobar el desarrollo realizado por el autor en el presente documento.

El desarrollo tiene dos partes bien diferenciadas y buscan remarcar el caracter investigador y metódico del presente trabajo:

1. Experimentos previos y selección de tecnologías: en estos primeros apartados, el lector podrá comprobar el estado actual de las tecnologías de comunicación en el ámbito de los dispositivos empotrados. Estos expe-

rimentos servirán como base y justificación de la propuesta realizada en la siguiente sección.

2. Desarrollo de middleware de comunincaciones: en esta sección se realizará la adaptación de un middleware de comunicación ZeroC-Ice pensando para ser ejecutado sobre sistemas operativos de propósito general a la plataforma Zynq. El desarrollo principal se basa en la adaptación de ZeroC-Ice para ansi c. Este desarrollo consiste en adaptar determinados ficheros de cabecera para dar soporte a la plataforma. Finalmente, tal y como se verá en la sección dedicada al desarrollo, se implementan dos tecnologías de comunicación: UDP y TCP aunque en este trabajo únicamente se muestra la implementación en UDP con el objetivo de simplificar el documento.

Finalmente, una vez realizado el desarrollo, en la Sección 4.1 se realiza una breve reflexion sobre los aspectos conseguidos en el proyecto y los resultados obtenidos. En la Sección 4.2 se proponen numerosas vías de investigación a partir de este trabajo que, en un futuro próxmo, serán exploradas por el laboratorio de investigación ARCO con el objetivo de potenciar el middleware de comunicaciones Icec.

1.2. Motivación

El proyecto surge bajo la colaboración que se ha sostenido en el tiempo entre el grupo de investigación ARCO (Arquitectura de Comunicaciones Universidad de Castilla-La Mancha) y el laboratorio MLAB (Multidisciplinary Laboratory) del centro "The Abdus Salam International Centre for Theoretical Physics" (ICTP).

En ARCO, laboratorio al que pertenece el autor de este trabajo desde hace 4 años, se tiene una gran experiencia en sistemas de comunicación y tecnologías de comunicación basadas en middleware de comunicaciones. Uno de los campos de investigación de este laboratorio es la lógica reconfigurable y FPGAs. Por otro lado, el MLAB centra su investigación en la instrumentación científica mediante dispositivos de lógica reconfigurable, focalizando su atención en los instrumentos para física nuclear. En concreto, en los últimos años el MLAB ha estado trabajando con el Istituto Nazionale di Fisica Nucleare (INFN).

En uno de los cursos impartidos por el MLAB y al cual el autor de este proyecto acudió como ponente, los investigadores del MLAB manifestaron la problemática con la que se encuentran a la hora de desarrollar soluciones basadas en lógica reconfigurable con los últimos avances en dispositivos SoC. El background de los investigadores de este laboratorio está más cerca de la física y la electrónica y se aleja, por tanto, de la programación de alto nivel y los sistemas de comunicación que están involucrados en la transferencia de datos desde el ARM (o cualquier otro microprocesador dentro del SoC) y el ordenador de control. Basados en la experiencia desarrollando aplicaciones de instrumentación, en el MLAB , llegan a la conclusión que existe un problema recurrente a la hora de transmitir los datos desde el dispositivo de medición al ordenador de frontend o control (utilizado de forma indistinta durante este trabajo) y se discuten con el grupo de investigación futuras líneas de trabajo y de colaboración.

Fruto de estas conversaciones, el grupo de investigación ARCO decide aplicar su experiencia con sistemas de comunicación basados en middleware a los sistemas de instrumentación científica. Con el objetivo de valorar la viabilidad de seguir con la filosofía de un middleware de comunicaciones

para sistemas de medición científica y, en general, sistemas que requieran de una alta eficiencia, se propone este trabajo.

Mediante la elaboración de este trabajo se busca, en primer lugar, analizar los diferentes factores que influyen en el proceso de comunicación entre el microcontrolador/microprocesador alojado en el SoC para, más tarde, analizar la viabilidad de desarrollar un middleware de comunicaciones que simplifique el trabajo recurrente comentado anteriormente.

Capítulo 2

Estado del arte

Al ser este un problema tangencial a muchos desarrollos resulta muy complicado cubrir todos los escenarios de aplicación posibles. Por este motivo, se centrará el estudio en la aplicación a los sistemas de adquisición de datos para la instrumentación científica.

2.1. Sistemas de adquisición de datos

Los sistemas de adquisición de datos son aquellos que permiten captar las variables que intervienen en un determinado proceso para, más tarde, ser tratadas o analizadas en un proceso posterior. En el proceso de adquisición intervienen muchos elementos y su estudio detallado requeriría de un documento íntegro dedicado al mismo. No obstante, si el lector desea ampliar los fundamentos aquí recogidos puede apoyarse en [11].

De forma general, un sistema de adquisición realiza las siguientes acciones:

- Adquisición de la señal / variable a medir
- Acondicionamiento
- Digitalización

- Control / envío de datos al ordenador de frontend.

Adquisición de la señal

En primer lugar hay que definir qué es una **señal**; "Una señal, en general, es una onda electromagnética que permite transmitir información a un circuito electrónico". Existen dos tipos, principalmente, de señales:

1. Señales analógicas: señal que varía con el tiempo de manera continua.
2. Señales digitales: señal cuyo valor a medir se puede representar mediante un conjunto finito de valores discretos.

En función de la señal a captar se utilizará un sensor u otro. El **sensor**, es el elemento encargado de hacer variar una propiedad interna de forma proporcional a la variable que se desea medir. Por ejemplo, un sensor de temperatura variará una propiedad (resistencia, diferencia de potencial a la salida, intensidad) en función de la temperatura que esté sensando. La proporción de cambio entre ambas propiedades puede ser diferente en función del sensor y vendrá caracterizada por una función de transferencia. Una vez se ha seleccionado el sensor a utilizar, dicho cambio en la propiedad del sensor deberá ser medido mediante un elemento de adquisición y acondicionado.

El **circuito de acondicionamiento de la señal** es el encargado de manipular la señal de entrada de modo que se reduzcan, en la medida de lo posible, aquellos artefactos naturales como el ruido, errores de transmisión, etc. Este circuito es de vital importancia a la hora de determinar la calidad de un sistema de adquisición de datos debido a que, la calidad de las etapas siguientes, están directamente relacionados con este proceso. Los elementos o etapas más destacables de este proceso y de los dispositivos que intervienen en en el mismo son:

- Aislamiento: el equipo de medición, como por ejemplo un osciloscopio, puede estar en contacto con el operario u otros elementos de la instalación que, puedan suponer un peligro, tanto para el ser humano como

para la propia instalación. Por esta razón, se debe de añadir una capa de aislamiento que se encargue de reducir la exposición de los procesos posteriores a la entrada de la señal. Un ejemplo podría ser la tensión necesaria que se debe aplicar para ionizar un gas determinado.Además de proteger al ser humano y la instalación, este proceso también se encarga de aislar las variables de medición (típicamente cuando es un sistema multicanal). Así, se evitan problemas como el voltaje en modo común o el ruido por acoplamiento.

- Compensación por "Cold-Junction": la compensación por la diferencia de metales en un sensor Resistence Temperature Sensor (RTD). Es solo un ejemplo de la que sería la siguiente etapa en el proceso de acondicionamiento. Esta compensación está asociada a un tipo de sensor concreto. No obstante, otros sensores requerirán de otros tipos de compensación similares que serán aplicados en este paso. Otro ejemplo es la composición por desgaste de excitación térmica en un sensor de temperatura químico.

- Filtrado: el filtrado es un proceso esencial en cualquier sistema de adquisición de datos. Un ejemplo típico de filtrado podría ser el filtrado de 50 Hz que se realiza en los sistemas de audio para evitar el ruido de las fuentes de alimentación. Este proceso determinará la precisión y calidad del sistema.

- Amplificación y atenuación: la amplificación y atenuación juegan un papel determinante a la hora de escoger los dispositivos que formarán parte de las etapas posteriores. Por poner un ejemplo de la atenuación y amplificación, se nombrará la problemática en dos proyectos que se usaron como vehículo para probar uno de los sistemas de adquisición desarrollado en el MLAB. Uno de los proyectos requería de la detección de una corriente del orden de **pico amperios**. Como es lógico, esta corriente debe ser amplificada y convertida a un voltaje que entre dentro de los intervalos de funcionamiento del conversor de señales analógico-digital. En este ejemplo se puede ver claramente la necesidad y la importancia del sistema de acondicionamiento. Al ser la señal tan baja (del orden de los pico amperios) cualquier ruido haría inútil la medida final. Podemos señalar a modo de ejemplo, como en otro proyecto surgió la necesidad de medir un pico de voltaje del orden de los 4000 voltios en unas fuentes de alimentación para unos detectores de

electrones mediante detectores de tipo [15]. En este caso, parece lógico pensar que dicha señal deberá ser atenuada para hacerla usable por el sistema de conversión. Además, en este caso, la descarga dura del orden de los pico segundos por lo que cualquier capacidad parásita en el proceso de acondicionamiento podría hacer que la medida estuviera demasiado desfasada. Estos son solo dos ejemplos de la gran variedad de situaciones donde se debe realizar un proceso de amplificación o atenuación.

- Linealización: la linealización permite mapear la variación de la magnitud a medir con respecto al cambio de la propiedad variable del sensor. De este modo, se busca mantener un cambio lineal a la entrada de la siguiente fase (esto ayudará a definir los rangos de cuantización).
- Multiplexación: esta etapa se encarga de gestionar la entrada por un mismo canal para varias magnitudes. Esto, permite reducir el coste del sistema. Sin embargo, resulta en un sistema más complejo y con más limitaciones debido a dicha multiplexación.

Una vez adquirida la señal, esta se envía al **conversor analógico digital**, en la etapa llamada **conversión**. La labor principal del proceso de conversión consiste en; a partir de una variable analógica representada por un conjunto infinito de valores, mapear dicha variable a un valor dentro de un conjunto finito de valores digitales.

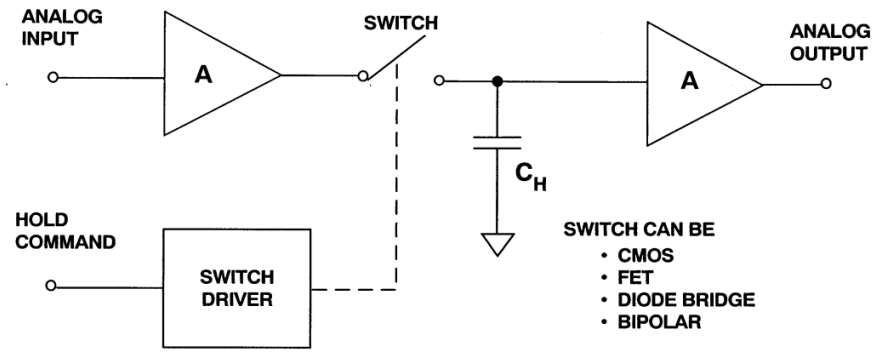

Figura 2.1: Circuito de sampling and hold

Dentro del proceso de conversión se suelen diferenciar 2 procesos esenciales:

- Sampling and holding (Muestreo y retención): la señal analógica varía de forma constante con el tiempo. Por otro lado, los circuitos de conversión requieren de un tiempo mínimo para poder trabajar. El circuito de holding es el encargado de mantener la señal para que esta pueda ser muestreada. En la Figura 2.1 se puede ver el circuito de muestreo y mantenimiento. El circuito de mantenimiento tiene, básicamente, 4 componentes:

  1. Amplificador de entrada
  2. Retenedor de energía (condensador)
  3. Buffer de salida
  4. Circuito de conmutación

  El funcionamiento de este circuito, de forma muy resumida, funciona del siguiente modo: la entrada analógica alimenta un amplificador de entrada. En el modo de seguimiento, el switch está cerrado y por lo tanto el condensador "Ch" se irá cargando hasta llegar a la diferencia de potencial de la entrada. Finalmente, el voltaje derivado del amplificador pasa también al buffer de salida que desacopla el circuito de entrada con respecto a los procesos posteriores. Cuando el circuito está funcionando en modo de holding, el switch se encuentra abierto gracias al "comando" recibido por el driver de conmutación. En ese momento, el condensador hace la función de mantener el nivel de entrada anterior y se conecta a la salida que alimentará a los procesos posteriores.

- Cuantización y codificación: tal y como se ha visto en el punto anterior, a la salida del circuito de sampling and holding se obtiene un voltaje, es decir, una magnitud analógica. En esta fase, la primera parte consiste en la cuantización. La cuantización, descrita de un modo breve, consiste en asignar un valor numérico al valor de entrada. Este valor numérico estará dentro de un conjunto finito de valores y, a su vez, este conjunto finito vendrá determinado por el rango de cuantización del Analog Digital Converter (ADC). Una vez determinado el valor que representará la salida del circuito de sampling and holding, el siguiente

paso consistirá en codificar dicho valor de una determinada manera. Codificar únicamente significa representar un valor de un determinado modo. Típicamente, se utiliza la codificación binaria con "n" bits de resolución. En la Figura 2.2 se puede ver de forma más clara dicho proceso. El valor codificado no representa de forma exacta el valor a la entrada, únicamente es una aproximación que; en función de la precisión, el número de bits y otras muchas características, será más cercano al valor real o no.

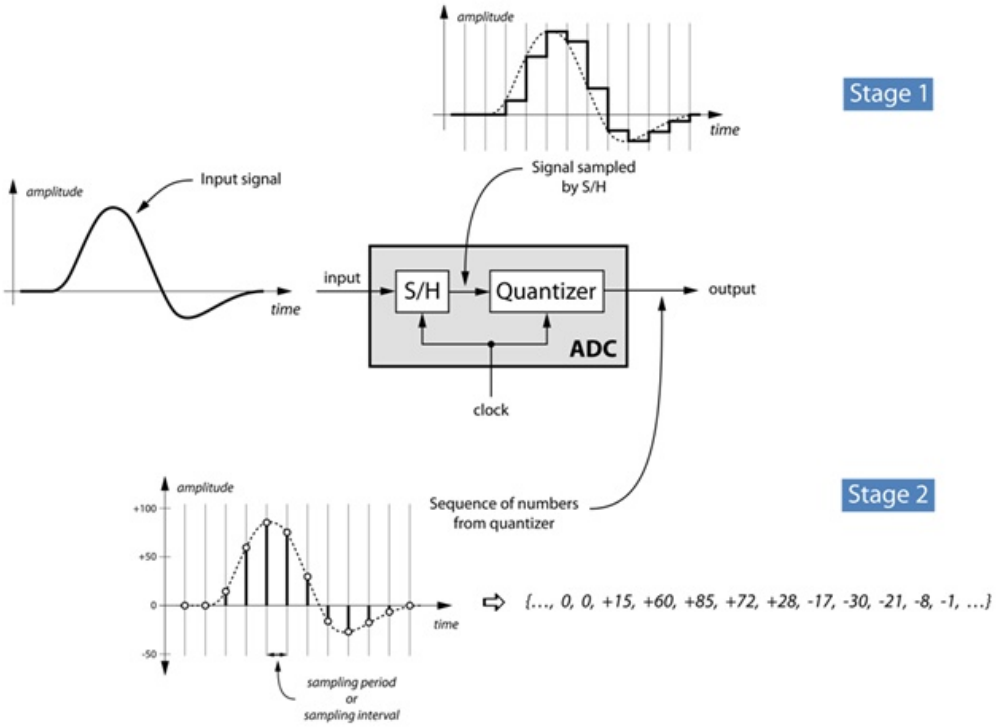

Figura 2.2: Ejemplo de codificación

Transmisión de la información

En el apartado anterior se ha realizado un breve repaso de los conceptos principales que intervienen a la hora de realizar la adquisición de una señal.

Sin embargo, el proceso completo es de gran complejidad y existe toda una disciplina del conocimiento entorno a dichos conceptos. Se puede profundizar en estos conceptos en la literatura especializada como [5].

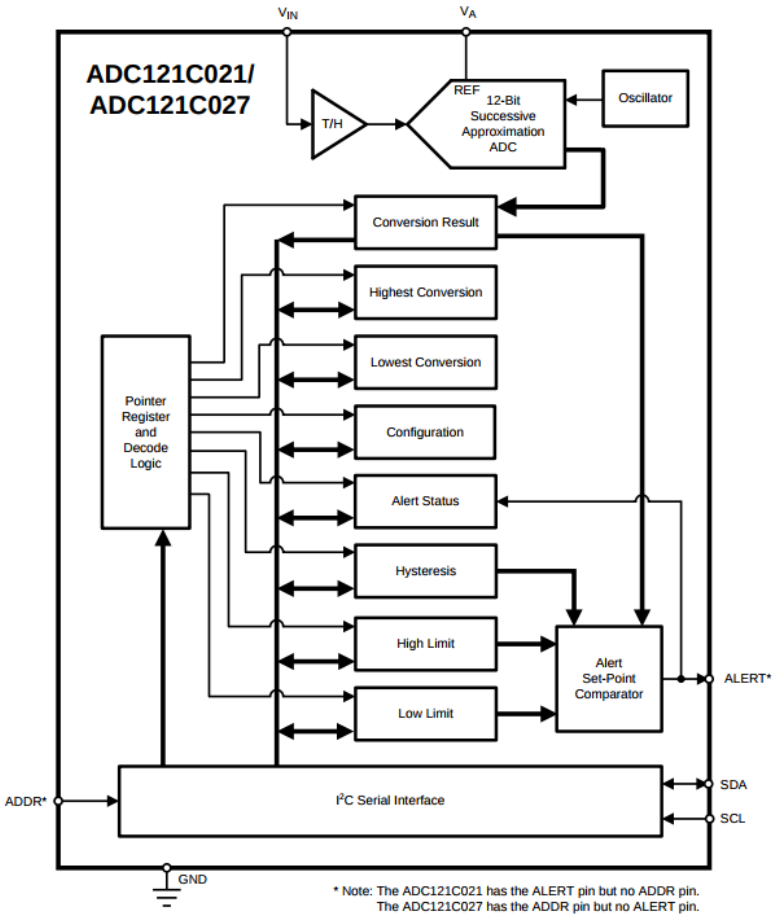

Figura 2.3: Diagrama familia ADC121c021

Una vez adquirida la señal y codificada, el siguiente paso consistirá en la transmisión de dicho valor que representa la magnitud medida para, más tarde, ser consumida por procesos concretos de la lógica de negocio.

En función del tipo de ADC, la interfaz será de un tipo u otra. Típicamente, se distinguen dos tipos de interfaz:

- Interfaz serie: la interfaz serie consiste en la utilización de algún tipo de comunicación de tipo secuencial como: RS-232, I2C, SPI con el objetivo de enviar el valor procesado de forma secuencial hacia el dispositivo de control. Este tipo de comunicación resulta de interés para aquellos ADC que no requieren de gran ancho de banda. Es importante tener en cuenta que la comunicación serie, tal y como su nombre indica, involucra el envío de forma secuencial, por lo que, el ancho de banda se ve reducido. La ventaja principal de este tipo de comunicación es que se simplifica la interfaz con microcontroladores gracias al uso de protocolos estándar. Un ejemplo de ADCs con este tipo de comunicación o interfaz son la familia ADC121C021/ADC121C021Q/ADC121C027 que puede ser consultados desde [1]. En la Figura 2.3 se puede ver el diagrama de bloques de esta familia.

- Interfaz paralela / (RAW): Este tipo de comunicación es la más sencilla de implementar desde el punto de vista del ADC pero el más complejo de manejar a la hora de integrar con el resto del sistema, debido a las dificultades inherentes a la transmisión paralela de información en circuitos de alta frecuencia [8]. A la salida del ADC se hayan tantos pines como bits de precisión se utilizan para la codificación. Estos pines tendrán el valor alto cuando contengan un 1 y el valor bajo cuando contenga un 0. De este modo, desde el controlador únicamente hay que leer dichas entradas digitales de manera sincronizada y calcular el valor. El ADC TLV571 es un ejemplo de ADC de tipo paralelo y la información del mismo se puede encontrar en [12].

2.2. FPGAs y la evolución hacia los SoC

Tal y como se verá a lo largo del desarrollo del documento, las Field Programmable Gate Arrays (FPGAs) juegan un papel clave en el proyecto desarrollado y en la motivación del mismo. Las capacidades de reconfiguración, así como, la flexibilidad inherente a la arquitectura implementada en estos dispositivos hace que, unido a la incorporación de procesadores embebidos, estas plataformas dinámicas sean un entorno ideal en el que implementar algún middleware de comunicaciones que simplifique y homogeneice la comunicación.

Las FPGAs son una evolución de una tecnología que comenzó con las Programmable Read Only Memory (PROM) donde se descubrió que, si se utilizaban las líneas de direccionamiento como entradas del circuito, las salidas en el bus de datos podrían ser la salida de cualquier función lógica. Este era un uso indirecto de la tecnología y no escalaba bien, ni en costes ni en espacio. Por esta razón aparecieron las primeras Field Programmable Logic Array (FPLA) que consistían en dos "paneles" cada uno con un conjunto de puertas lógicas. El primer panel consistía en puertas AND y el segundo en puertas OR. De este modo, las entradas estaban unidas mediante puertas AND y sumadas con puertas OR. Así, se podía aplicar cualquier función lógica mediante suma de productos. En la Figura 2.4 se puede ver un esquema de los primeros FPLA obtenida de [3]

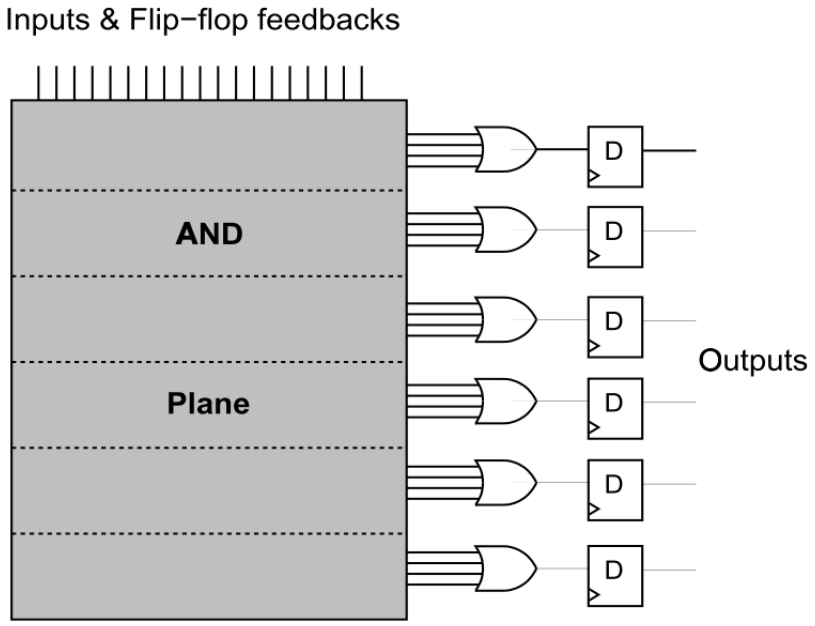

Figura 2.4: Organización de planos lógicos

Siguiendo con esta filosofía introducida por Philips en 1970, se avanzó hacia los Programmable Array Logic (PAL) que, simplificando la jerarquía de planos, reducía los problemas en cuanto a velocidad y costes de producción. Para simplificar esta jerarquía, en vez de tener dos planos, se unifican en un único plano y un número estático de puertas OR. El hecho de tener puertas

fijas reduce significativamente la flexibilidad del dispositivo, no obstante, en el momento en el que aparecieron daban respuesta a las necesidades de mayor velocidad y reducción de costes de los dispositivos.

Finalmente, aparecen los Complex Programmable Logic Devices (CPLDs) de la mano de Altera. Estos dispositivos disponían de una gran capacidad de densidad lógica y por lo tanto se podían implementar funciones lógicas mucho más complejas.

Por último, para devolver a los dispositivos programables la flexibilidad de los primeros PAL, surgen los Mask-Programmable Gate Arrays (MPGAs) y las FPGAs. Las FPGA están formadas por un conjunto de celdas lógicas y recursos de interconexión que permiten implementar cualquier tipo de función lógica.

Para implementar dicha lógica, la FPGA configura cada uno de los "cables" programables mediante la información proporcionada por el diseñador. De este modo, se puede ver como una analogía al cerebro humano. Cada neurona equivaldría a una celda lógica y las dendritas de cada una de ellas serían los recurso de interconexión. Finalmente, los switches que configuran los recursos de interconexión serían los neurotransmisores que intervienen en el proceso de sinapsis.

Existen muchos tipos de switches y tecnologías de enrutamiento, sin embargo, no es objeto de este documento discutir en profundidad estos aspectos. Si se desea se puede consultar [3] para más información.

Ventajas y desventajas de las FPGAs

Tal y como se ha podido ver en la sección anterior, la tecnología de FPGAs lleva muchos años siendo investigada y, en los últimos años, debido a la gran capacidad de integración, así como otros factores como la compra entre diferentes empresas como Intel y Microsemi/Atmel ha hecho que esta tecnología haya evolucionado de forma vertiginosa.

Las ventajas principales de este tipo de dispositivo son, en primer lugar, su **capacidad de reconfiguración y reprogramación**. Se entiende por

reconfiguración “la capacidad para modificar el diseño implementado dentro del dispositivo”. Esto quiere decir que un diseñador puede implementar un diseño de un circuito concreto y más tarde modificar el mismo debido a un fallo en el diseño, una mejora o cualquier otra razón. Esta característica implica que el coste de desarrollo se reduce de forma radical. La alternativa sería el diseño en Aplication-Specific Integrated Circuit (ASIC) y estos carecen de dicha flexibilidad por lo que, cualquier fallo en el diseño supondría grandes pérdidas económicas. Si comparamos las FPGAs con dispositivos como las Central Processing Units (CPUs) estas tienen unas características que las hacen ideales para muchas aplicaciones relacionadas con la computación de altas prestaciones:

- Paralelismo natural: se diseña hardware, no un conjunto de instrucciones sobre un hardware secuencial como las CPUs.
- Baja frecuencia: esto reduce el consumo de potencia debido a que este es directamente proporcional a la frecuencia
- Baja latencia: se puede conseguir implementar funciones lógicas con una latencia muy baja comparada con el equivalente implementado en un dispositivo como una CPU o microcontrolador
- Gran cantidad de recursos Digital Signal Proccesor (DSP): esto lo hace ideal para el tratamiento de señales digitales.

Aunque estos dispositivos tienen grandes ventajas, también se les atribuyen desventajas bastante importantes que han frenado su adopción:

- Atado a herramientas de desarrollo privativas: estas plataformas están completamente atadas a las herramientas de desarrollo proporcionadas por los fabricantes. Estas herramientas Computer-Aided Design (CAD) son, en general, herramientas privativas y con una curva de aprendizaje muy pronunciada.
- Dificultad de diseño: el diseño en estas plataformas es considerablemente más difícil frente a procesadores de propósito general o Graphical Processing Unit (GPU). Esto ha hecho que su adopción en entornos

como la computación científica, su uso haya quedado relegado, únicamente, a expertos en la materia. Sin embargo, en los últimos años se ha hecho un gran avance en las herramientas High Level Synthesis (HLS) que permiten reducir el tiempo de desarrollo.

Con el objetivo de simplificar los diseños y aumentar el uso de las FPGAs. Gracias a la mejora en los procesos de fabricación y el aumento en las capacidades de integración las empresas como Xilinx están creando soluciones SoC que, además de incorporar una lógica reconfigurable o FPGA, también incorporan un procesador de propósito general como un ARM v7 dentro del mismo chip. De este modo, se tienen las ventajas de ambas plataformas y se puede relegar a la lógica reconfigurable únicamente aquellos procesos que resulten de interés debido a sus características.

2.3. Red pitaya, caso de estudio

En este apartado, se realizará un estudio de la plataforma Red pitaya que sirve de ejemplo de cómo han evolucionado los sistemas de medición científica y, cómo se puede implementar una arquitectura para la medición científica mediante SoC.

Qué es Red Pitaya

Red pitaya se define a sí misma como "la navaja suiza para los ingenieros". De forma resumida, Red Pitaya es una tarjeta que permite implementar diferentes instrumentos de medida de forma reconfigurable. De este modo, en vez de tener; un osciloscopio, un analizador de espectro y un generador de señales arbitrarias, se tiene un único dispositivo con las capacidades de ser reconfigurado para adoptar la funcionalidad de cada uno de los instrumentos anteriormente enumerados.

Red Pitaya (en su última versión ha cambiado el nombre a STEMlab) es la implementación de un producto que fue estudiado anteriormente en el ámbito científico por diversos centros de investigación. Curiosamente, este producto

fue planteado años atrás en el paper [4] por el grupo de investigación MLAB, centro dónde surgió la necesidad de realizar este trabajo fin de máster.

Para conseguir esta reconfigurabilidad la Red Pitaya está formada por:

- Sistema de adquisición de datos
- Lógica reconfigurable: incorpora una de las últimas familias de la marca Xilinx. En concreto, implementa el SoC Zynq 7010, que cuenta con dos procesadores A9 y una FPGA Artix.
- Sistema operativo y ecosistema: este es, probablemente, la característica más interesante desde el punto de vista de este trabajo pues es donde se plantea una implementación al problema de la comunicación entre el dispositivo de medida y el dispositivo final que consume los datos medidos. En este caso, la Red Pitaya implementa un sistema operativo GNU/Linux junto a una arquitectura de capas que comunica la lógica reconfigurable o FPGA con el dispositivo final que consume los datos.

En la siguiente sección se tratará, de forma resumida, cómo se ha solucionado la arquitectura de comunicaciones entre la FPGA o, la lógica donde se realiza la captación de los datos, hasta el dispositivo final.

Arquitectura del sistema

Red Pitaya supone un claro ejemplo en la evolución de los sistemas de medición científica de los que hemos hablado anteriormente.

En este apartado se realizará un breve recorrido por la arquitectura del sistema con el objetivo de mostrar la complejidad que supone implementar una arquitectura completa en un sistema SoC como los actuales.

En primer lugar, se partirá del sistema de adquisición de datos y de la comunicación del mismo con la FPGA. Para seguir con una breve descripción del diseño base implementado dentro de la arquitectura reconfigurable o FPGA y, finalmente, analizar el sistema implementado dentro del microprocesador ARM.

En la Figura 2.5 se puede ver el esquema de la arquitectura típica implementada dentro de la FPGA.

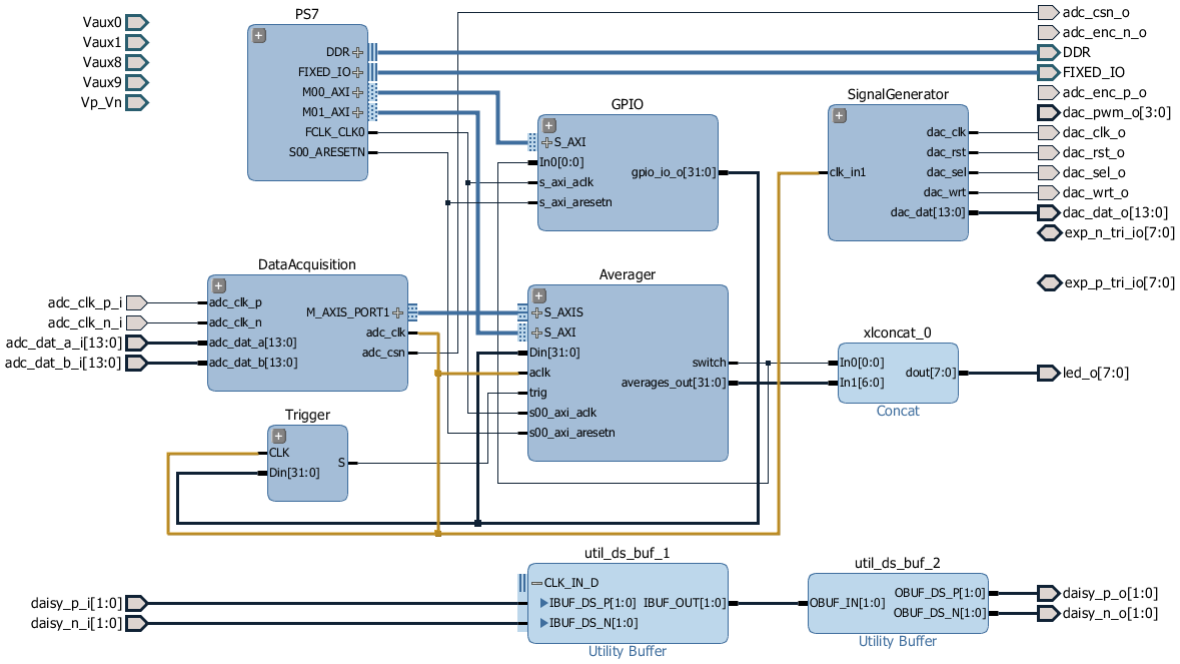

Figura 2.5: Arquitectura de diseño en Red Pitaya

Tal y como se puede observar, las señales provenientes del ADC se conectan directamente al bloque *DataAdcquisition*. Este bloque, normalmente, es el encargado de realizar las tareas de control del ADC. A la salida del bloque *DataAcquisition* se puede ver que la salida es de tipo *AXI_STREAM*. Este tipo de conexión es un estándar dentro del estándar AMBA 4.0 [19]. Básicamente, esta conexión es un stream de datos con alguna señal de control extra. Dichos datos, serán inyectados de forma directa, en este caso, al ponderador. El ponderador es un bloque que se encarga de calcular la media de los valores de entrada. Este es solo un ejemplo de aplicación, en este punto se podría incluir cualquier tipo de lógica. Precisamente es aquí donde se puede apreciar la potencia de este tipo de sistemas al que el usuario final tiene acceso a los datos en crudo en tiempo real y puede realizar cualquier tipo de procesamiento **online**. Resulta interesante resaltar otra de las entradas al bloque *Averager*, la entrada S_AXI. Esta entrada determina que el bloque es un esclavo de un bus de tipo AXI. En este caso, el esclavo está directa-

mente conectado al sistema PS7, que es la instanciación del procesador ARM incluido. Esta conexión se utiliza, típicamente, para configurar los bloques programables desde el procesador.

Además de los bloques explicados existen muchos otros en el diseño que no serán explicados debido a que únicamente son parte de un ejemplo inicial de uso de la Red Pitaya y carecen de sentido para el objetivo de este documento. Finalmente, se puede ver otro bloque llamado SignalGenerator que, está directamente conectado a la entrada de datos del Digital Analog Converter (DAC). Por lo que este diseño en realidad implementa dos dispositivos: un sistema de adquisición y otro de control.

Una vez realizado el diseño en bloques (en el pasado la mayor parte había que describirla mediante lenguajes de descripción de hardware como VHDL), el siguiente paso sería grabar el diseño en la FPGA. De este modo, el conjunto de puertas y de interconexiones, quedarían configuradas para cumplir con las funciones lógicas descritas en el bloque de la Figura 2.5.

Tal y como se ha comentado anteriormente, en el diseño se ha instanciado el procesador ARM. En concreto, el procesador utilizado por la Red Pitaya es el ARM Cortex-A9 con Floating Point Unit (FPU) y coprocesador NEON. Esto implica que en un mismo diseño se integra un procesador de propósito general multinúcleo junto a un procesador Single Instruction Multiple Data (SIMD) así como un sistema de lógica reconfigurable FPGA. Como se puede apreciar, hasta el ejemplo más mínimo supone un sistema de gran complejidad.

El procesador instanciado ARM Cortex A9 puede ser utilizado, de forma general, de dos formas:

- Programación baremetal: se realiza la programación directamente mediante código C o C++ de alto nivel que será compilado offline a código máquina. Esta alternativa se utiliza, tal y como se verá a medida que se avance en el documento, cuando se requiere de una gran granularidad y alta performance.
- Programación sobre sistema operativo: esta es la manera más simple de programar el dispositivo debido a que se instala un sistema operativo que abstrae las particularidades del hardware. El problema principal de

esta aproximación reside en los tiempos de respuestas, por ejemplo, a la hora de atender una interrupción. Dentro de los sistemas operativos se pueden diferenciar dos casos:

1. Sistemas operativos de propósito general: esta es la opción utilizada por Red Pitaya. Normalmente se utiliza un sistema GNU/Linux de propósito general como Debian o Ubuntu y un kernel personalizado que se adapta a las particularidades de la FPGA.
2. Sistema operativo de tiempo real: en esta categoría existen muchos sistemas operativos, la mayoría de pago, como: FreeRTOS, VxWorks, etc. La ventaja de estos sistemas operativos reside en que tienen una baja sobrecarga y además permiten asegurar el tiempo real dentro del sistema.

Tal y como se ha comentado, en Red Pitaya proporcionan toda la suite para ser utilizada bajo un sistema de propósito general como es GNU/Linux.

Desde la web de Red Pitaya se puede descargar la imagen para ser grabada de forma directa sobre una Secure Digital card (SD) que más tarde será insertada y leída por la Red Pitaya.

En el momento de la redacción de este documento, Red Pitaya proporciona soporte únicamente para Ubuntu. Debian se mantiene en una fase experimental sin soporte para las aplicaciones web, que, tal y como se verá más adelante, resultan de vital importancia.

El lector puede preguntarse en qué punto se encuentra el sistema una vez instalado el sistema operativo. Bien, el sistema ya se encuentra listo para recibir los datos que a través del DataAcquisition y el Averager son enviados a la memoria Double Data Rate (DDR) incluida en la placa Red Pitaya.

El sistema operativo tiene acceso a la memoria DDR donde se han mapeado todos los esclavos de tipo AXI implementados en el diagrama de bloques. De este modo, para acceder a cualquier bloque que sea esclavo del bus AXI y cumpla con su estándar, únicamente habrá que escribir en la dirección de memoria asociada a dicho bloque.

Las direcciones de memoria se asignan, en este caso, desde el programa de diseño CAD Vivado.

Una vez instalado el sistema operativo, Red Pitaya proporciona una gran suite para el desarrollo de aplicaciones sobre el diseño implementado en la FPGA. A continuación se dará al lector una breve descripción de la arquitectura de una aplicación dentro de Red Pitaya.

Arquitectura de una aplicación en Red Pitaya

En Red Pitaya una aplicación está dividida en dos grandes bloques, tal y como se puede ver en la Figura 2.6

- Frontend: Es el encargado de dibujar los botones, gráficas, y otros elementos visuales. Tal y como se puede apreciar en la Figura 2.6, el frontend no necesita estar, físicamente, en el mismo lugar que el dispositivo de medición.
- Backend: Está formado por el sistema de adquisición (visto anteriormente), la FPGA (vista anteriormente) y el sistema que forma parte del microprocesador.

En primer lugar, se tratará de explicar de forma resumida el backend para, más tarde, realizar una breve descripción de la arquitectura del frontend.

El backend se puede ver como la placa y todo lo que se está ejecutando sobre ella. Sin embargo, con el objetivo de ser más precisos se pueden identificar las siguientes partes dentro del backend:

- Sistema operativo:
  - Application Programming Interface (API): proporcionado por Red Pitaya, consiste en un conjunto de librerías para simplificar el tratamiento de la información desde la FPGA hasta el controlador.
  - Controlador: hace uso de las APIs para atender a las peticiones de la interfaz de usuario y gestionar el flujo de información.
  - Servidor web: encargado de gestionar las llamadas procedentes de la interfaz gráfica y derivar el procesamiento al controlador específico.

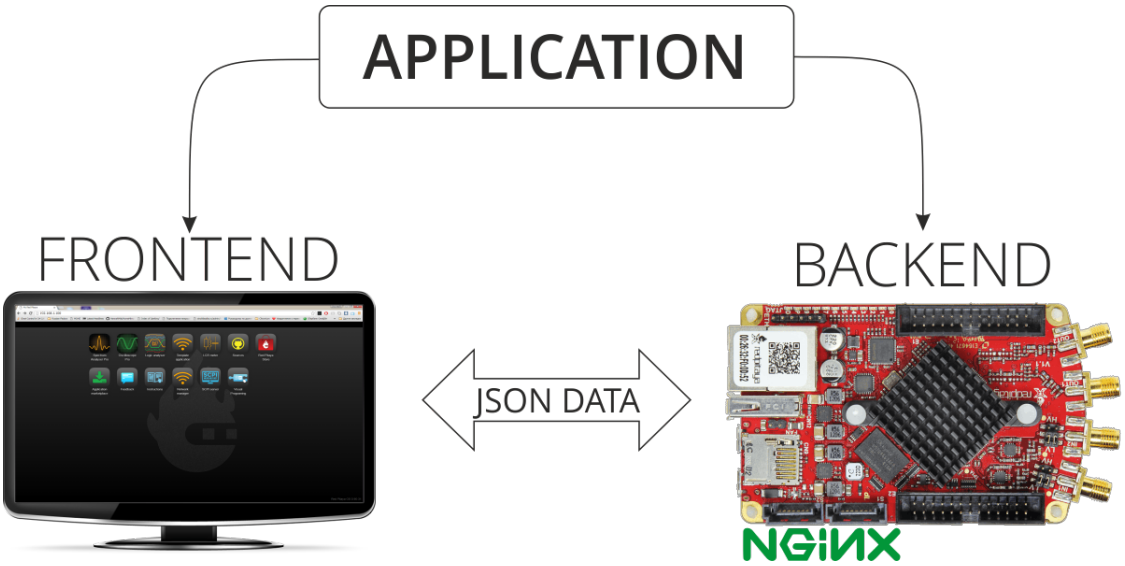

Figura 2.6: Arquitectura de una aplicación Red Pitaya

- Sistema de adquisición: visto anteriormente

A continuación, se tratará de dar una explicación de cada uno de los elementos anteriormente citados. El objetivo de este documento no es el realizar un manual de la Red Pitaya por lo que únicamente se cubrirán los aspectos más generales sin entrar en los detalles más técnicos.

Nginx es el servidor web que se encarga de gestionar las peticiones que se realizan desde el Web UI. De este modo, cuando el usuario pulsa a un botón en la interfaz de usuario, se genera una petición que llega al servidor Nginx. En función del botón o la funcionalidad de la interfaz que se utilice podrá ocurrir lo siguiente:

- Iniciar un nuevo diseño en la FPGA: cuando se pulsa sobre una aplicación nueva, se carga en la FPGA el diseño correspondiente a dicha aplicación. Para ello, el controlador envía, por medio de la API, el bitstream a la FPGA. En ese momento, se inicia una conexión de tipo websocket con la aplicación de usuario. Esta conexión websocket será la encargada de mantener la comunicación entre la interfaz de usuario y la FPGA.

- Envío de datos: si la aplicación ya ha sido desplegada, nginx remitirá la petición al controlador que, mediante la API, se comunicará con la FPGA para:
  - Control: la interfaz envía datos de control o de salida a la FPGA.
  - Adquisición de datos: la interfaz recibe los datos de la FPGA por medio de la conexión.

Tal y como se puede observar, dentro de esta comunicación existen dos elementos bien diferenciados:

- Controlador: el controlador debe tener unas funciones bien definidas por Red Pitaya. Estas funciones tienen, entre otras responsabilidades; iniciar la aplicación (por ejemplo, configurar el ADC), finalizar la aplicación (poner en deep sleep un módulo), fijar parámetros (configurar el offset de un ADC) y envío y recepción de datos. El controlador, se carga de forma dinámica mediante un comando del servidor nginx y está programado completamente en C. El controlador, por tanto, carga la aplicación y sirve de bridge o puente entre las peticiones web y la FPGA.
- API: la API se utiliza dentro del controlador y es la encargada de simplificar el uso de los bloques implementados en la FPGA mediante un conjunto de abstracciones. Por ejemplo, en vez de tener que llamar a una función del sistema operativo para mapear una dirección de memoria, la API permite llamar a una función que, dependiendo del identificador de dispositivo, automáticamente carga las direcciones de memoria asociadas y gestiona los errores. De este modo se simplifica el trabajo con el hardware. En [13] se pueden ver más ejemplos de funciones implementadas en la API.

Como se puede observar, el backend está formado por varios elementos interconectados. Todos estos elementos **suponen una sobrecarga** a la hora de realizar una medición o enviar un determinado parámetro a la FPGA. Sin embargo, en la documentación de Red Pitaya no se ha encontrado ninguna referencia a los costes, en tiempo, que todas estas capas de abstracción suponen para el sistema. las diferentes arquitecturas de comunicaciones para

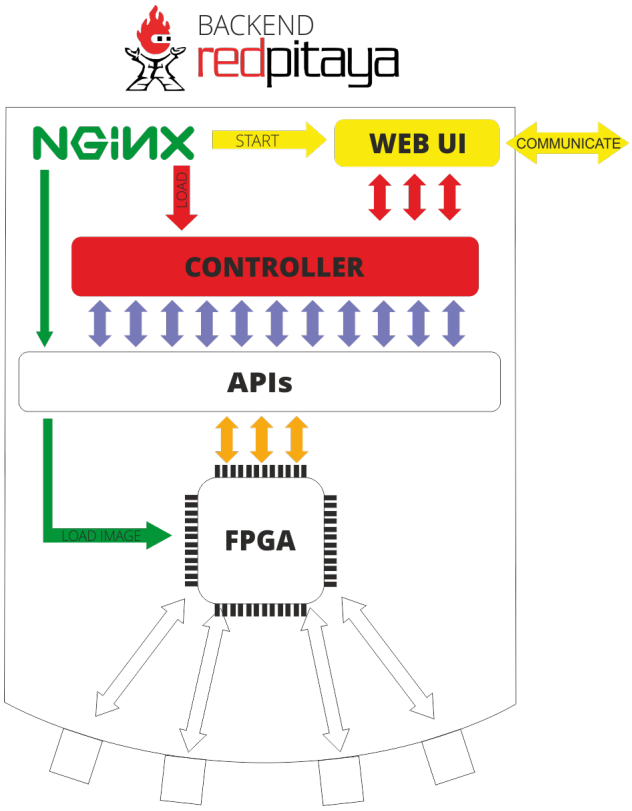

Figura 2.7: Arquitectura del frontend

poder seleccionar la arquitectura que mejor se adapte a las necesidades de cada proyecto.

El frontend, es el encargado de interactuar con el usuario; es lo que el usuario ve. Una analogía con el mundo físico podría ser que el frontend es equivalente a los controles de un osciloscopio y la pantalla donde se visualiza la señal.

En la Figura 2.8 se puede ver un ejemplo de frontend implementado en las aplicaciones por defecto de Red Pitaya.

Tal y como se puede observar, el "osciloscopio" de la figura tiene la apariencia de una página web y eso se debe a que, efectivamente, las tecnologías utilizadas, son tecnologías web. En concreto, una aplicación web implemen-

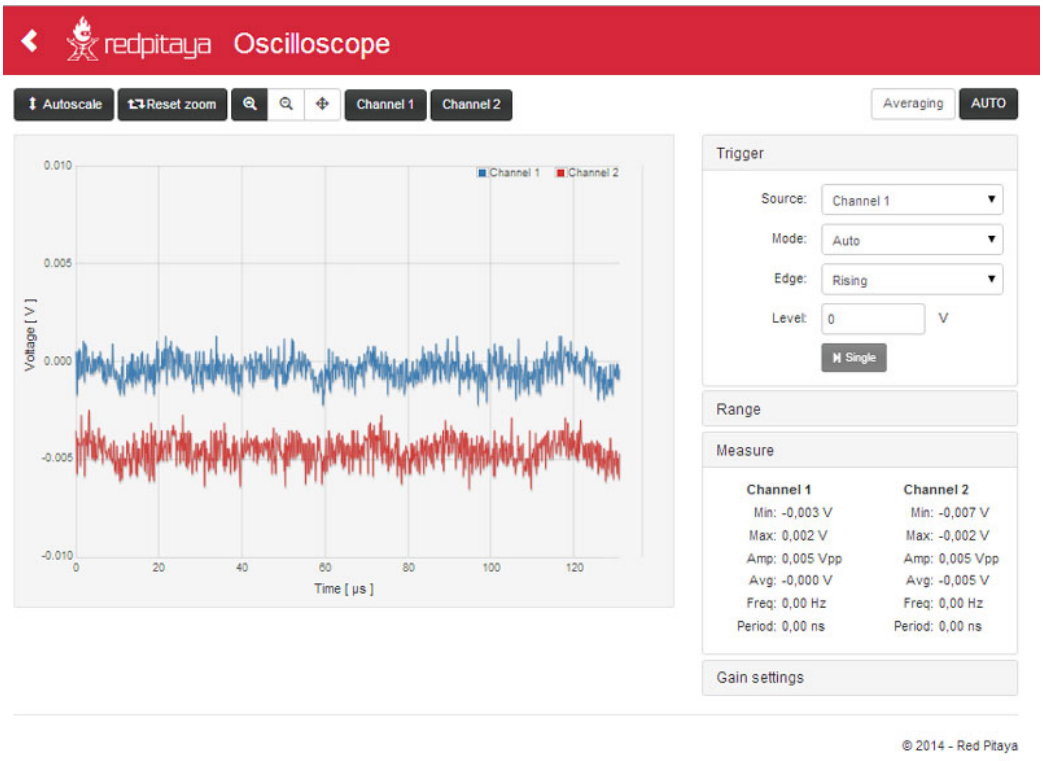

Figura 2.8: Ejemplo de instrumento implementado en Red Pitaya

tada en la Red Pitaya, cuenta con los siguientes elementos:

- Código HiperText Markup Language (HTML): este código es el encargado de indicar qué elementos aparecerán en la aplicación web y qué significado tienen. Red Pitaya se apoya en el último estándar (HTML5). Dentro de la aplicación, se podrá encontrar este código en el fichero index.html. En [14] se puede ver un ejemplo de código para un analizador de espectro. Este código también será el encargado de mostrar los diferentes diálogos y botones que el usuario utilizará para interactuar con el instrumento.
- Código Cascade Style Selector (CSS): este código es el encargado de proporcionar el estilo visual de la aplicación/instrumento. En este código se definen cosas como: posición de botones, color de los textos, dimensiones de las gráficas, etc. Idealmente este código debería estar se-

parado del código HTML por medio de un fichero con extensión ".css". No obstante, en Red Pitaya el código CSS se encuentra embebido en el mismo fichero que el HTML, es decir, en el index.html referenciado anteriormenente.

- Código Javascript (JS): este es, desde el punto de vista técnico, el código más importante del frontend. Este código es el encargado de dotar al sistema de dinamismo y proporciona el nexo de unión entre la interfaz y el instrumento implementado. El código Javascript, idealmente, debería estar situado en un fichero a parte. No obstante, al igual que ocurre con los ficheros CSS, en las aplicaciones por defecto proporcionadas por Red Pitaya, todo este código viene embebido dentro del código HTML referenciado anteriormente.

Para "contactar" con el instrumento implementado dentro de la Red Pitaya, el código Javascript utiliza una tecnología llamada "WebSockets". Los Websockets permiten abrir una conexión entre el navegador del cliente (donde se ejecuta el frontend) y el servidor (donde se encuentra el controlador del que hablamos en la sección dedicada al backend). De este modo, mediante diferentes comandos enviados en formato Javascript Serializable Object Notation (JSON) se gestiona la comunicación entre ambas partes. Para más información se recomienda al lector utilizar la documentación oficial desde `http://redpitaya.readthedocs.io/en/latest/developerGuide/software/webExamples.html`

2.4. Sistemas y tecnologías de la información

Una de las partes fundamentales de este proyecto radica en la comunicación entre el microprocesador ARM y el ordenador donde se ejecuta el Frontend. En este apartado se dará un breve repaso sobre las diferentes tecnologías de comunicación que están relacionadas con esta parte del sistema centrándose el estudio de la capa de transporte que será la más utilizada (de forma directa) durante el desarrollo del proyecto.

Los sistemas de comunicación tienen una larga historia, incluso antes de la aparición del computador. En esta sección se centrará el estudio en los

sistemas de comunicación entre computadores y más concretamente en la pila TCP/IP.

TCP/IP fue el resultado de un proyecto de Defense Advanced Research Projects Agency (DARPA). aproximadamente por el año 1970. En este momento se diseño una red, ARPANET que pretendía ser una red descentralizada y que pudiera soportar diferentes bajas en los nodos de comunicación. Sobre esta red y tras años de estudios, el grupo de investigación de Vint Cert (considerado como el padre de Internet) lanzó el primer Request For Comments (RFC) relativo a TCP. Este RFC 675 fue mejorándose hasta llevar a la cuarta versión que es el protocolo utilizado hoy en día.

Visto el interés de Internet, en 1980, la International Organization For Standardization (ISO) lanzó una especificación, bastante ambiciosa, llamada ISO/IEC 7498–1 o "modelo OSI". Esta especificación buscaba servir como modelo de referencia para la implementación de protocolos de red. Este modelo fue el seguido por todas las redes a la hora de implementar los diferentes protocolos y a día de hoy se mantiene en vigencia.

El modelo OSI, plantea una arquitectura en capas. La arquitectura en capas es una manera de plantear un sistema en el que una capa proporciona servicios a la capa inmediatamente superior y consume los servicios de la capa inmediatamente inferior. Mediante el modelo por capas se aíslan las responsabilidades de cada protocolo (en función de la capa en la que se encuentre) y se aumenta la interoperabilidad entre los mismos.

Con este objetivo en mente, se llegó a la conclusión que 7 capas serían suficientes para catalogar a todos los posibles protocolos dentro de una de ellas.

- Aplicación: ofrece los protocolos que utilizan las aplicaciones para comunicarse entre ellas. Por ejemplo, en el caso de la medición científica, el estándar Standard Commands for Programmable Instruments (SCPI) se podría catalogar dentro de esta capa. Las dos aplicaciones que quieren comunicarse deberán entender este protocolo y será por medio del mismo por el que puedan negociar sus capacidades de comunicación. Otro ejemplo de protocolo implementado dentro de esta capa podría ser el protocolo Hiper Textual Transfer Protocol (HTTP). Este protocolo

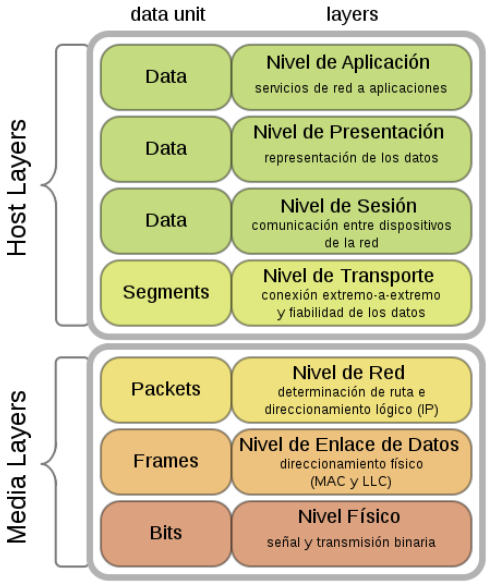

Figura 2.9: Modelo OSI

es el utilizado por la web para la negociación de los diferentes recursos entre el servidor y el cliente. Entre otras muchas funcionalidades HTTP define un conjunto de verbos como: POST, GET, PUT, PATCH, que definen la comunicación entre los dos extremos.

- Presentación: esta capa se encarga de gestionar cómo se deben interpretar los caracteres en cada uno de los nodos involucrados en la comunicación. Esto permite que, aunque cada nodo mantenga una representación interna de los caracteres diferente (uno ASCII y el otro UTF-32) ambos puedan comunicarse.

- Sesión: esta capa se encarga de mantener la sesión abierta durante la transferencia de datos. Los servicios de esta capa, al igual que los de la capa de presentación suelen agruparse en la capa de aplicación debido a que normalmente todos ellos son implementados por un único protocolo.

- Transporte: la capa de transporte es una de las capas más importantes del modelo OSI. Normalmente, la capa de aplicación se modifica con frecuencia, sin embargo, la capa de transporte raramente es modificada

de forma brusca debido a la importancia que supone en la pila. La tarea de esta capa es asegurar el transporte de los datos entre dos máquinas independientemente de las redes físicas en las que se encuentren. De este modo, se puede imaginar que una máquina se encuentra dentro de una red con una latencia extremadamente alta, por ejemplo 600 ms. Esta capa se encargaría de repetir los mensajes y adapta la velocidad de comunicación para que ambos nodos mantuvieran una comunicación sincronizada. Dentro de la capa de transporte se pueden distinguir dos grandes familias de protocolos:

- Orientados a conexión: los protocolos orientados a la conexión tienen una fase de negociación, transmisión de datos y finalización. Una analogía podría ser una llamada telefónica. Una llamada telefónica se inicia con un establecimiento de comunicación (el sonido y la presentación de los interlocutores), seguido de una transferencia de datos (la conversación) y una finalización de llamada (despedirse y colgar el teléfono). Un protocolo orientado a conexión y el más utilizado es el protocolo TCP. La ventaja de estos protocolos es su seguridad y capacidades de control de flujo. Sin embargo, como desventaja tienen su latencia hasta el envío del primer dato y la sobrecarga para implementar los mecanismos de seguridad y control de flujo.
- Sin conexión: estos protocolos no mantienen ningún tipo de sesión o conexión. Una analogía podría ser el servicio postal (ya prácticamente en desuso). En este caso el cartero no llama a la puerta ni se despide, únicamente deposita el paquete en el buzón (en el mejor de los casos) o debajo de la puerta. Si el perro se come el paquete o alguien se lo lleva, nadie sabrá que esto ha ocurrido (a no ser que hayan cámaras). Esta analogía permite identificar las ventajas y desventajas de este tipo de comunicación. La ventaja principal es la velocidad del protocolo y simplicidad en la implementación. La desventaja es que no se proporciona ningún tipo de mecanismo de seguridad ni control del flujo, pero, tal y como se ha hecho referencia en la analogía, se podría implementar dichos servicios en una capa superior (la cámara).

Las primitivas contempladas por la capa de transporte permiten definir la interfaz que proporciona a los servicios superiores, dentro de estas

primitivas se encuentran:

- LISTEN: el servidor espera conexiones.
- CONNECT: el cliente se conecta a un servidor que ha realizado un LISTEN.
- SEND: se envía la información.
- RECEIVE: se espera información
- DISCONNECT: se inicia el proceso de finalización de la conexión

Existen muchos protocolos de comunicación y características implementadas en esta capa del modelo OSI. No obstante, con el objetivo de reducir la longitud del documento y dado que en la sección dedicada al desarrollo se verán con más profundidad los protocolos de la capa de transporte utilizados, se aconseja al lector, en caso de desearlo, consultar la bibliografía recomendada [18].

- Red: este nivel es el encargado de gestionar la problemática que surge a la hora de pasar de una red a otra red. El objetivo de esta capa es la de hacer que un paquete llegue desde un origen a un destino aunque entre medias haya muchos otros intermediarios. Para llevar a cabo esta tarea los protocolos de red son interpretados por un dispositivo llamado router. El protocolo de red más utilizado es Internet Protocol (IP), no obstante existen otros como APPLETALK.

- Enlace: la capa de enlace, junto a la capa de transporte, se considera una de las capas más importantes del modelo OSI. Esta capa es la responsable de asegurar la tranmisión de datos entre dos nodos directamente conectados. Esta capa se encargará de trocear los paquetes provenientes de la capa de red en unidades de datos (tramas) de la longitud óptima para el enlace actual. Si el enlace se realiza a través de un medio compartido como WIFI o una topología de BUS, esta capa se encargará de arbitrar el acceso a dicho medio.

- Físico: esta es la capa de más bajo nivel y es la encargada de proporcionar servicios como: envío bit a bit entre dos nodos, proporcionar estándar de conectores, gestionar la modulación, gestionar el modo de transmisión...

Como se ha podido observar, cada capa tiene unas responsabilidades determinadas y esto hace que, a la hora de implementar protocolos, los diseñadores tengan claro qué función deben implementar y en qué parte se situan dentro de la pila de comunicación.

Normalmente, la gestión de esta pila de comunicación se realiza por medio de un sistema operativo o alguna librería de alto nivel. Del mismo modo, las capas física y de enlace suelen ser implementadas directamente en el hardware de red de los dispositivos.

En el desarrollo del presente trabajo se comprobará como, gracias a esta arquitectura, el desarrollador final únicamente tiene que centrarse, en la mayoría de los casos, en el nivel de aplicación. Como el nivel de aplicación se posiciona (teniendo en cuenta la agregación dentro de la capa de aplicación entre los niveles de sesión, presentación y aplicación) justo encima de la capa de transporte, únicamente deberá implementar el protocolo que desee (o utilizar uno existente) y, en el caso de implementar uno nuevo, decidir qué servicios de la capa de transporte desea utilizar.

2.5. Sistemas operativos y de tiempo real

Tal y como se verá en la sección dedicada al desarrollo del proyecto, en la realización del mismo se han planteado diferentes cuestiones referentes a la utilización de un sistema operativo, dentro del microprocesador ARM implementado en la FPGA utilizada. Con el objetivo de proporcionar al lector de los conocimientos básicos para contextualizar las decisiones y cuestiones planteadas en el desarrollo del proyecto, en este apartado se dará un breve repaso sobre los sistemas operativo. Como los sistemas operativos son un campo de estudio muy amplio, se reducirá el foco a los sistemas operativos más utilizados en este tipo de plataformas y aquellos que han sido tomados en cuenta a la hora de tomar las decisiones de diseño.

Qué es un sistema operativo

Antes de analizar los diferentes sistemas operativos que se pueden encontrar en el mercado, resulta interesante describir qué es un sistema operativo, en qué capa de abstracción se encuentra y qué ventajas/desventajas ofrece a un diseño (teniendo en cuenta en todo momento que únicamente se plantea el uso en los sistemas que en este trabajo se estudian, es decir, los sistemas "empotrados").

Según wikipedia [20], "un sistema operativo es el software principal o conjunto de programas, que gestiona los recursos del hardware y provee servicios a los programas de aplicación de software, ejecutándose en modo privilegiado respecto de los restantes"

El sistema operativo se encuentra, tal y como se puede ver en la Figura 2.10 entre el hardware y la aplicación. De este modo el programador no debe preocuparse por cómo gestionar el acceso al hardware, únicamente tiene que utilizar una API con funciones de alto nivel para su uso.

Existen infinidad de clasificaciones de sistemas operativos en función de muchos factores. En este caso, tal y como se ha comentado, se centra el estudio en aquellas características más relevantes para el trabajo que se desarrollará. En este sentido, podemos clasificar los sistemas operativos en dos grandes categorías:

- Sistemas operativos de propósito general: aquellos sistemas que proporcionan una gran facilidad a la hora de desarrollar aplicaciones sobre los mismos, pero no tienen en cuenta detalles de bajo nivel que pueden ser necesarios a la hora de gestionar de forma más precisa el hardware del sistema. Entre estos sistemas operativos se encuentra Windows 10, Ubuntu, Debian y otros muchos sistemas. Cabe destacar que, en el caso de Ubuntu y Debian, hay que matizar que ambos son distribuciones del sistema operativo GNU/Linux y que pueden existir variantes que se engloben dentro de los sistemas operativos de tiempo real.
- Sistemas operativos de tiempo real: estos sistemas operativos se centran en asegurar que se respeten una serie de condiciones temporales. De este modo, se puede certificar qué determinados eventos ocurrirán

Figura 2.10: Jerarquía de abstracciones en un sistema informático

dentro de un tiempo acotado. Estos sistemas operativos son muy utilizados en los sistemas industriales así como en sistemas de aviónica y en instrumentación científica, entre otros muchos campos de aplicación. Los sistemas operativos de tiempo real se caracterizan por:

- Determinismo
- Menores latencias
- Mayor control sobre el acceso al hardware
- Mayor fiabilidad y tolerancia a fallos
- Menos sobrecarga del sistema

Existen muchos ejemplos de sistemas operativos de tiempo real como QNX, FreeRTOS, VxWorks y algunas variantes de GNU/Linux.

Sistemas operativos en dispositivos empotrados

Los sistemas empotrados han ido evolucionando con el tiempo y a día de hoy gran parte cuenta con un sistema operativo que permite simplificar la labor de desarrollo así como aumentar las capacidades del sistema.

Un ejemplo podría ser la famosa plataforma Raspberry Pi. Su popularidad se debe a su bajo coste y facilidad de desarrollo de aplicaciones. En realidad esta facilidad no tiene nada que ver con la plataforma si no con el soporte de un sistema operativo como GNU/Linux que simplifica, a través de todas sus capas de abstracción, la labor de interactuar con el hardware.

Por lo general, los sistemas operativos utilizados en los sistemas empotrados son versiones modificadas de los sistemas operativos de propósito general o, en el caso de los sistemas operativos de tiempo real, versiones completamente diseñadas para dichos sistemas.

Con el fin de ilustrar dos casos de éxito en la aplicación de sistemas operativos a sistemas empotrados, a continuación se dará una breve descripción de dos sistemas ampliamente utilizados en muchos sistemas empotrados y que además han sido tenidos en cuenta en los estudios llevados a cabo para el desarrollo de este proyecto

GNU/Linux

GNU/Linux es un sistema operativo de operativo libre de propósito general. El sistema operativo a veces aparece referenciado como Linux pero este último únicamente hace referencia al kernel del sistema.

GNU/Linux es un sistema de tipo UNIX [2] y como responsables directos del proyecto tiene a dos desarrolladores: Linus Torvals (creador del kernel Linux) y Richard Stallman (creador del proyecto GNU).

El origen del sistema operativo se puede situar en muchas fases del desarrollo, en este caso se hace referencia a las primeras versiones catalogadas como GNU/Linux sobre los años 90.

Actualmente GNU/Linux está en la mayor parte de los supercomputadores del top 500 [9] y en la mayoría de los teléfonos móviles debido a su uso en los sistemas Android.

GNU/Linux se suele proporcionar junto a un conjunto de aplicaciones que simplifican el aprendizaje y uso del sistema por los usuarios finales, a este conjunto se le denomina distribución. Las distribuciones más conocidas son: Ubuntu, Suse, Debian, etc. Estas distribuciones cuentan con la mayoría de programas que un usuario medio puede esperar como: entorno gráfico, suite ofimática, programas multimedia, navegador... Sin embargo, en el caso de los sistemas embebidos todas estas funcionalidades no son necesarias y únicamente ocupan un espacio valioso de memoria.

Un ejemplo de la utilización de GNU/Linux en el entorno de los sistemas embebidos y que resulta de interés para este proyecto es Petalinux. Petalinux es una distribución GNU/Linux así como un conjunto de herramientas para el desarrollo de Xilinx que, entre otras cosas proporciona:

- Utilidades de línea de comandos
- Generadores de drivers y de librerías de alto nivel para interactuar con el hardware
- Herramientas para el despliegue en tarjetas de desarrollo
- Herramientas para realizar depurado de aplicaciones Simulador

La metodología de desarrollo con un sistema como petalinux es la siguiente:

1. Creación y diseño del hardware (en el caso de una FPGA)
2. Descripción del hardware
3. Configuración del sistema operativo (selección de herramientas)
4. Compilación del sistema operativo
5. Despliegue del sistema operativo
6. Desarrollo de la aplicación

GNU/Linux necesita adaptarse a las necesidades de cada sistema hardware concreto. Antiguamente, la mayoría de este trabajo era resuelto por el kernel que, en forma de drivers introducidos durante la fase de compilación, permitía utilizar el hardware subyacente. No obstante, con el paso del tiempo esto ha ido mejorando creando una estructura modular que dota de una mayor flexibilidad al sistema. Actualmente, GNU/Linux o más concretamente, Linux, el kernel, tiene soporte para módulos del sistema.

Los módulos del kernel son drivers que pueden ser incorporados en tiempo de ejecución, incluso después de ser desplegado y compilado el kernel. Esto proporciona de un gran dinamismo al sistema.

Desde hace unos años, el soporte de Linux para hardware reconfigurable ha mejorado de forma sustancial. El principal problema de estas plataformas reconfigurables radica en que gran parte del hardware no puede ser descubierto de un modo sencillo como ocurriría con un dispositivo Universal Serial Bus (USB). Con el objetivo de poder reconocer todo tipo de dispositivos hardware y proporcionar dotar al kernel de la información necesaria, se incorpora al kernel el Device Tree. El Device Tree es una de las piezas claves a la hora de proporcionar las capas de abstracción sobre el hardware reconfigurable.

De este modo, en el arranque del sistema el kernel lee el fichero Device Tree en formato binario y registra cada uno de los dispositivos descritos.

Una vez desplegado el sistema operativo en el sistema empotrado, el desarrollador se encontrará ante una gran suite de herramientas que le permitirá, entre otras cosas, reducir el tiempo de desarrollo en el sistema final.

Por ejemplo, la capa TCP/IP estudiada en la Sección 3 está completamente implementada en el kernel de Linux y el programador únicamente tiene que utilizar la API proporcionada por el sistema operativo, en este caso una API orientada a sockets.

Las ventajas por tanto de un sistema operativo como GNU/Linux se pueden resumir en:

- Reducción en el tiempo de desarrollo
- Facilidad en el desarrollo

- Mejoras de seguridad
- Escalabilidad

Por otro lado, las desventajas de utilizar un sistema operativo como GNU/Linux son:

- Sobrecarga
- Aumento de la latencia
- Necesidad de aplicar políticas de seguridad y políticas de actualización
- Mayor cantidad de componentes normalmente significa mayor cantidad de puntos de fallo
- Mayor consumo de memoria

Para solventar las desventajas como la latencia y la sobrecarga, dentro de GNU/Linux, existen variantes específicamente pensadas para este propósito. Por otro lado, para mitigar el mayor consumo de memoria, la cantidad de componentes del sistema y reducir las necesidades de políticas de seguridad, existen soluciones como YOCTO que permiten, con una filosofía de arquitectura modular, reducir los componentes que forman parte del sistema operativo. Petalinux utiliza YOCTO para crear su distribución.

Los problemas de latencias están relacionados con el kernel, es decir, con Linux. Con este objetivo se han desarrollado diferentes versiones y parches para dotar al kernel de las capacidades de tiempo real. Todas estas alternativas se pueden dividir en función de la filosofía en:

- Arquitectura multikernel: la mayoría de soluciones multikernel no son modificaciones reales al kernel Linux. Realmente, se establece una jerarquía de kernel, típicamente en dos niveles. El primer nivel es el encargado de gestionar el hardware y, lo más importante, ejecutar el segundo kernel (un kernel Linux normal) que a su vez ejecutará las aplicaciones de usuario cuando no haya tareas de mayor prioridad listas para ser ejecutadas. Como puede apreciarse, este enfoque es demasiado simple

y supone una sobrecarga natural al tener dos kernels. Además, a la hora de desarrollar las aplicaciones finales, se deberá mantener dos kernel diferentes así como implementar los drivers en el kernel de primer nivel que "habla con el hardware". Esto, por el contrario, está resuelto en el kernel Linux ya que posee la mayoría de drivers necesarios. Tal y como se puede ver en la Figura 2.11 el hardware tiene comunicación directa con el RTLinux Plugin (para este ejemplo, pero cualquier tecnología basada en esta jerarquía utilizará los mismos bloques). Entre otras muchas cosas, el RTLinux Plugin, que no es otra cosa que un microkernel, proporciona un Scheduler de tiempo real. Este scheduler se encarga de gestionar las diversas tareas R-T (Real-Time) y el proceso que contiene el kernel Linux. Otro problema derivado de esta arquitectura es la baja productividad y los problemas de licencia dado que la mayoría de tareas de tiempo real se ejecutan en el espacio del microkernel (normalmente licenciado con General Public License (GPL)).

- Arquitectura monokernel: Las arquitecturas monokernel basan su funcionamiento en dotar al kernel de las capacidades de tiempo real de forma directa. El principal proyecto que mantiene esta filosofía es preemptive_rt que a día de hoy sigue siendo incorporado poco a poco al kernel Linux oficial. Desde la url `https://wiki.linuxfoundation.org/realtime/start` se puede visitar la documentación oficial. Tal y como se puede observar el proyecto ha sido "apadrinado" por "The Linux Fundation" y está bajo desarrollo activo. Mediante este parche se pueden alcanzar latencias de 80 microsegundos en una placa de desarrollo Altera Ciclone V con un ARM Cortex A-9. Existe una comunidad que se encarga de realizar baterías de pruebas en las diferentes arquitecturas soportadas por el kernel del Linux para comprobar las capacidades de tiempo real sobre cada una de ellas. Desde la url `https://www.osadl.org/Quality-assurance-at-the-OSADL-QA-Farm.osadl-services-qa.0.html` se puede acceder a los resultados y los test que están siendo ejecutados.

En estas sección se ha cubierto, de forma resumida, la historia de GNU/Linux y las capacidades que proporciona a los diseños. También se ha estudiado la problemática que surge a la hora de utilizar el sistema en operaciones de tiempo real y se ha explorado las principales arquitecturas utilizadas, normalmente, en estos sistemas.

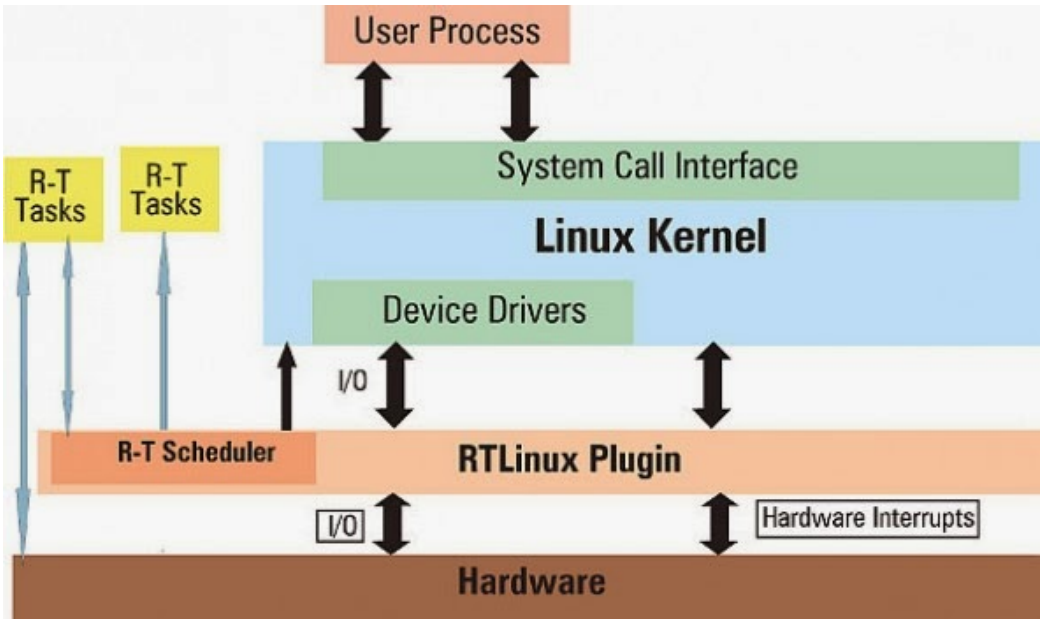

Figura 2.11: Arquitectura de sistema operativo en tiempo real multikernel

Sistemas operativos de tiempo real

Tal y como se vio en la sección dedicada a GNU/Linux, el kernel Linux, por defecto no es un kernel de tiempo real por lo que para dotar al sistema de estas capacidades se tiene que parchear de manera externa.

Aunque se espera que esto cambie y en el futuro próximo Linux pueda ser configurado con las capacidades de tiempo real, a día de hoy no se le puede considerar un sistema operativo de estas características.

Uno de los grandes problemas de los sistemas de tiempo real es que exigen determinismo. El tiempo en que llega una señal de frenado puede ser vital para un sistema de conducción autónoma y por lo tanto no se puede permitir que el tiempo de detección no sea determinista.

A día de hoy, asegurar este tipo de determinismo en procesadores de propósito general resulta extremadamente difícil debido a que no existen modelos matemáticos que puedan acotar los tiempos de ejecución. Esta falta de modelos se debe en parte a las capacidades de paralelización a nivel arqui-

tectura y que no en todos los procesadores se pueden desactivar. Un ejemplo sencillo son las caches, que aunque siguen una distribución estadística, no se puede asegurar de forma determinista la tasa de errores para un programa dado.

Esta es la razón de que plataformas SoC como las proporcionadas por Xilinx utilicen microprocesadores con una arquitectura más sencilla y determinista como ARM y que proporciona al programador más flexibilidad a la hora de configurar la propia arquitectura. Sin embargo, esta es solo una de las partes involucradas en la ejecución de un programa. El código y el sistema operativo (en realidad es otra pieza más del código) debe ser trazable y determinista. En el caso de un sistema operativo de propósito real como GNU/Linux esto es prácticamente imposible de asegurar debido a su gran cantidad de módulos.

Siguiendo con los ejemplos proporcionados por una de las asociaciones que más interés han puesto en dotar a GNU/Linux de las capacidades de un sistema de tiempo real (Open Source Automation Development Lab (OSADL)), en la Figura 2.12 se puede ver como son las gráficas reportadas por los sistemas bajo pruebas. En estas gráficas se da una visión de la distribución de las latencias para un determinado banco de pruebas. No obstante, esto solo es una visión empírica del problema y no asegura que puedan existir ocasiones en que los tiempos sean completamente diferentes.

Debido a esta problemática, desde hace años, se han ido desarrollando los sistemas operativos de tiempo real.

En este apartado se estudiará **FreeRTOS** debido a que será uno de los sistemas utilizados en el desarrollo del proyecto y debido también a su popularidad en los sistemas embebidos.

FreeRTOS es un sistema operativo de tiempo real que a día de hoy, según la Wikipedia, ha sido portado a unas 35 plataformas diferentes. FreeRTOS es un sistema operativo de software libre y licenciado bajo licencia Massachusetts Institute of Technology (MIT).

FreeRTOS tiene como objetivo dotar al sistema de las capacidades necesarias para ejecutar un sistema en tiempo real, para ello, el sistema operativo se mantiene simple y fácilmente trazable.

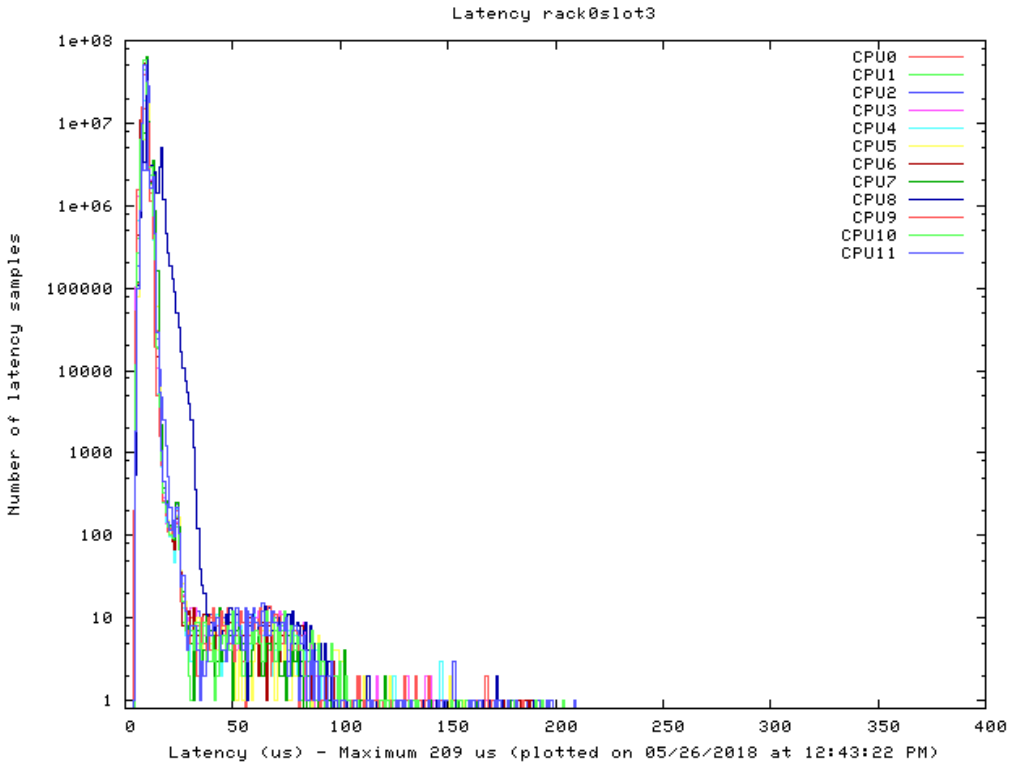

Figura 2.12: Ejemplo de reporte de osadl

Todo el sistema operativo se encuentra distribuido en 3 archivos de código C y algunas funciones en ensamblador para lidiar con las interrupciones y alguna característica concreta de cada port.

Desde FreeRTOS se podrán ejecutar varias tareas de tal manera que, en cada instante de tiempo, la tarea de mayor prioridad se estará ejecutando. Si no hubiera ninguna tarea programada por el usuario, entonces el sistema operativo ejecutará la tarea idle. La tarea idle únicamente se ejecuta cuando no existe ninguna otra tarea con prioridad mayor a la mínima.

Para la comunicación entre tareas en FreeRTOS se proporcionan colas de mensajes. Del mismo modo, para la sincronización, el sistema cuenta con semáforos y barreras entre otras muchas herramientas.

Un problema importante a la hora de asegurar el determinismo de un

sistema radica en la memoria dinámica. Esto se debe principalmente a la fragmentación de memoria. Si una memoria se encuentra muy fragmentada el tiempo para que el orquestador de memoria sea capaz de encontrar espacio para la memoria requerida puede variar.

Para solventar este problema y, a la vez, dotar al sistema de capacidades de memoria dinámica, FreeRTOS proporciona 4 esquemas diferentes de alojamiento de memoria:

- heap_1.c: esta es la implementación más sencilla. En este caso no se permite eliminar la memoria una vez que se a requerido. De este modo se evita cualquier tipo de fragmentación. La implementación de esta estratégia es trivial. En primer lugar se crea en tiempo de compilación un espacio de memoria con tamaño igual al determinado por la variable configAPPLICATION_ALLOCATED_HEAP. Una vez compilado, cada llamada a la funcion pvMalloc() mueve el puntero hasta el siguiente bloque de memoria libre dentro del array. En la mayoría de los sistemas embebidos esta estrategia será suficiente debido a que, normalmente, la mayoría de sistemas "piden" toda la memoria al inicio y no la liberan en toda su ejecución.

- heap_2.c: este esquema utiliza el algoritmo de espacio que mejor se adecua al solicitado. Esto quiere decir que el algoritmo recorrerá toda la pila buscando aquel hueco que menos espacio de fragmentación deja. Esta estrategia permite desalojar bloques de memorias anteriormente utilizados. Además, para reducir la latencia en las operaciones, esta estrategia no desfragmenta la memoria una vez desalojada. Es importante tener en cuenta que esta estrategia no es determinista debido a que si las llamadas para adquirir memoria dinámica se realizan con diferentes tamaños, esto podría dar lugar a fragmentaciones y por lo tanto a tiempos diferentes entre alojamientos.

- heap_3.c: este esquema únicamente proporciona la capacidad de hacer seguras las llamadas malloc y free proporcionadas por el compilador, cuando se usan en un entorno con varias tareas (thread safe).

- heap_4.c: exactamente igual que el método 2 pero con capacidad de unir bloques de memoria anteriormente liberados.

Además de estas implementaciones a la hora de realizar el *port* a una plataforma nueva, el diseñador puede implementar otras estrategias diferentes.

Mediante los 3 ficheros anteriormente citados y estas estrategias para gestionar la memoria, FreeRTOS mediante su API, proporciona un entorno simple donde implementar sistemas de tiempo real deterministas.

Para llevar a cabo la gestión de las diferentes tareas implementadas en el sistema, FreeRTOs implementa un scheduler que, mediante una política apropiativa, es capaz de seleccionar la sucesión de tareas que deben ejecutarse en el tiempo para cumplir con las prioridades configuradas por el usuario.

A continuación se dará una breve explicación de cómo gestionaría una sucesión de tareas típicas en FreeRTOS. Este ejemplo está tomado de la documentación oficial de FreeRTOS la cual puede ser consultada en [6]

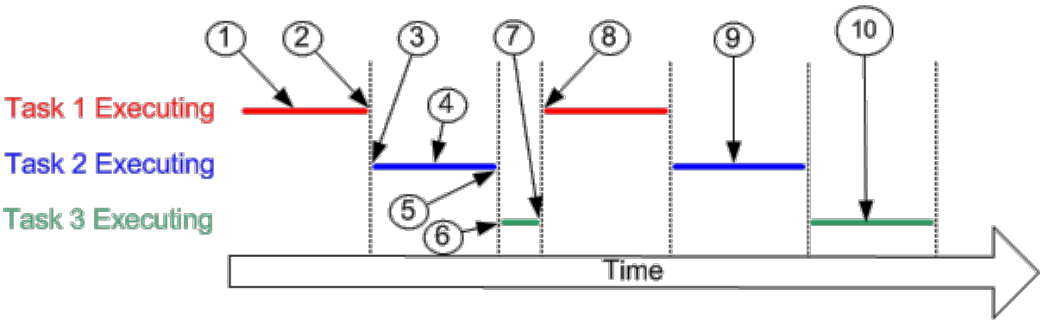

Figura 2.13: Ejemplo de ejecución de tareas en FreeRTOS

En la Figura 2.13 se pueden ver 3 tareas instaladas en el scheduler por medio de la API para la creación de tareas en FreeRTOS. En primer lugar, la tarea 1 se está ejecutando cuando en el T = 2 el kernel desaloja la tarea 1 porque la tarea 2, con mayor prioridad requiere su atención. En T = 3 la tarea 2 se ejecuta e inicia sus cálculos utilizando un recurso compartido por medio de un semáforo. En T = 5 el kernel desaloja a la tarea 2 para ejecutar en T = 6 la tarea 3 que en sus cálculos también requiere del mismo recurso que la tarea 2. Al tener todavía la tarea 2 el recurso, cuando en T = 7 la tarea 3 pide el acceso al recurso se bloquea. En este momento, el kernel desaloja la tarea (no tiene nada que hacer) y ejecuta la siguiente tarea. La siguiente tarea se inicia en T = 8 y es la tarea 1. Llegado T = 9 y tras un desalojo anterior, la tarea 2 termina con el recurso compartido y lo libera. En este

momento, la tarea 3 se ejecuta y continua su ejecución accediendo al recurso compartido.

Como se puede observar, el funcionamiento de las políticas de planificación implementadas por el kernel FreeRTOS son bastante sencillas de entender, no obstante, esta simplicidad es lo que permite asegurar un determinismo en el funcionamiento.

Para conseguir todos estos cambios de contexto el kernel necesita, de algún modo, asegurar los estados en los que se encuentra cada tarea antes de realizar el cambio a otra tarea. En la Figura 2.14 extraída de la documentación oficial se puede ver claramente la problemática a la que se enfrentan los desarrolladores de FreeRTOS a la hora de realizar los cambios de contexto. En la Figura 2.13 se puede ver como la tarea ha cargado dos valores en dos registros para, más tarde, sumar ambos registros. Justo en el momento en el que se procede a realizar la suma el kernel solicita que dicha tarea sea desalojada. Esto puede hacer que si la otra tarea modifica dichos registros y luego se continúa la tarea 1 desalojada dará un error de cálculo.

Esta es una labor básica del scheduler y es una de las funcionalidades más complejas dentro del kernel FreeRTOS.

Al inicio de esta sección se ha comentado lo importante que resulta realizar un trazado de cómo evoluciona el sistema a lo largo de su ejecución. De este modo, el diseñador puede asegurarse que el sistema está funcionando según lo planificado y que no existe ningún error en la planificación. FreeRTOS da soporte a este trazado mediante la definición de un gran conjunto de macros que permiten, entre otras cosas, saber cuando una tarea ha sido iniciada, desalojada, bloqueada esperando un mensaje, bloqueada esperando un recurso, etc.

Finalmente, es importante resaltar que este trabajo se centra en las medidas en la comunicación final entre el microcontrolador y el dispositivo que ejecuta el frontend. Para poder llevar a cabo esta comunicación, hoy en día, resulta indispensable la utilización de la capa TCP/IP. Como ya se ha comentado anteriormente, los sistemas operativos de propósito general ya tienen implementado en su kernel esta funcionalidad por lo que el programador únicamente tiene que utilizar la API para lidiar con la comunicación final.

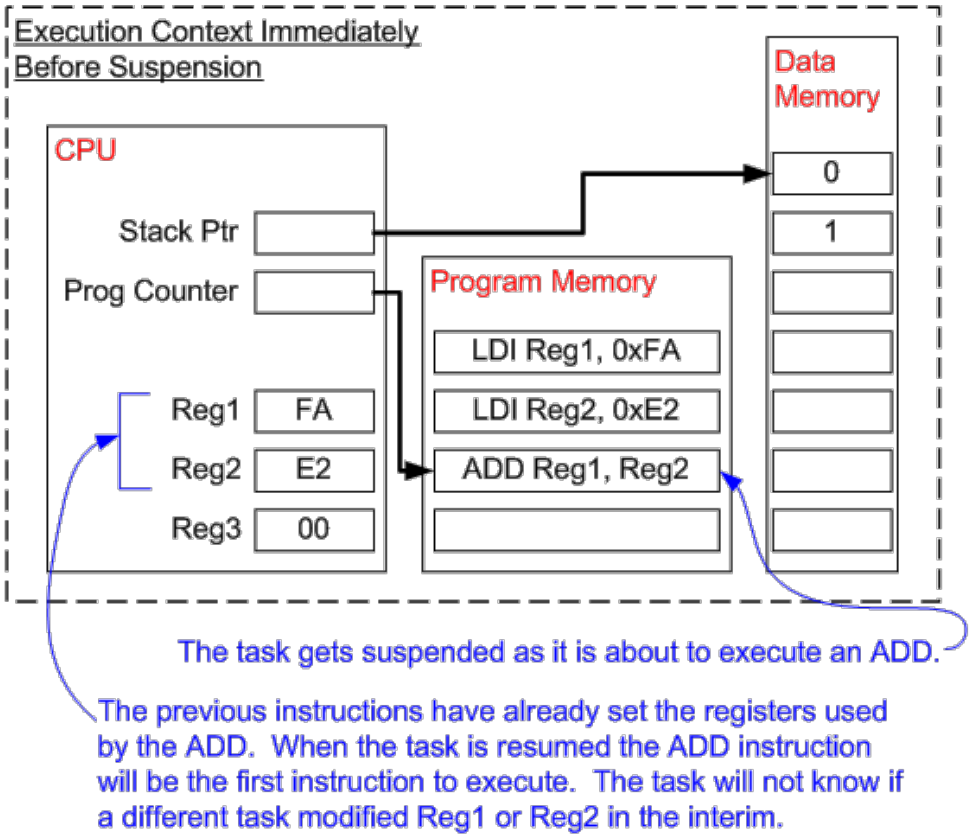

Figura 2.14: Ejemplo de cambio de contexto en FreeRTOs

Es por esto que resulta interesante que el sistema operativo de tiempo real tenga alguna facilidad a la hora de gestionar las comunicaciones con otros dispositivos.

En los últimos años FreeRTOS ha hecho un gran esfuerzo para implementar toda la pila de protocolos TCP/IP sobre FreeRTOS.

Siguiendo la filosofía de FreeRTOS esta funcionalidad se puede agregar por separado al kernel de modo que los diseñadores que no requieran de esta funcionalidad puedan evitarla en la fase de compilación y por tanto ocupen menos memoria en el dispotivo final.

FreeRTOS+TCP, el nombre elegido para la capa TCP/IP sobre FreeR-

TOS, se define como una implementación libre y thread safe que proporciona una API basada en los sockets de Berkeley.

Durante la fase de desarrollo se dará una breve descripción de la implementación de esta pila con el objetivo de que el lector pueda analizar la solución propuesta.

2.6. Middleware de comunicaciones

Durante el transcurso de esta sección dedicada al estado del arte se ha realizado un repaso por los principales elementos que intervienen en el desarrollo del proyecto que será explicado con más detalle en la Sección 1.2. En primer lugar, se ha analizado como han cambiado los sistemas de medición científica para, a continuación, seguir analizando las FPGAs que forman parte de la mayoría de estos sistemas y seguidamente la instanciación de uno de estos sistemas como es la Red Pitaya. Por otro lado, se ha analizado otro de los componentes claves a la hora de brindar una solución dentro de un sistema embebido, como son los sistemas operativos, con el objetivo de que el lector pueda situarse dentro de las diferentes alternativas existentes en el mercado y, más concretamente, las características que muchos de estos sistemas comparten dentro de su misma categoría.

Tal y como se ha podido ver en la Sección 2.3, a la hora de implementar cualquier aplicación, existe una gran cantidad de código que debe generarse a mano (o en el mejor de los casos por el asistente que rellenará código de prueba) y que, en general, realiza la misma funcionalidad una y otra vez, lo único que varía son los datos contenidos y algunos parámetros de comunicación. Todas estas repeticiones de código hacen que, por un lado, el desarrollador tarde más tiempo en realizar su aplicación y, que se generen más errores de codificación y sea más complejo de mantener.

En el caso de la instrumentación científica con lógica reconfigurable, la mayor parte de los diseñadores, son diseñadores hardware con un perfil electrónico. Esto supone una mayor productividad desarrollando el hardware reconfigurable situado en la FPGA, pero, por otro lado, reduce la productividad en el desarrollo software. Por esta razón resulta de gran interés reducir

al máximo posible la programación en estos sistemas.

El mismo fenómeno ocurre del lado del frontend. Los diseñadores de frontend normalmente tienen aptitudes más artísticas y programáticas pero no algorítmicas o técnicas. Esto hace que el diseñador, a la hora de implementar algoritmos o estrategias complejas para la comunicación deje de ser productivo.

Por último, aun siendo productivo en ambas partes, se ha observado que existe una gran repetición de código entre proyectos y que por lo tanto, este código repetido podría encapsularse.

En esta sección se revisarán las tecnologías que permiten encapsular esta funcionalidad. Cabe destacar que se centrará el estudio en aquellas tecnologías que simplifican la labor de la comunicación. Típicamente a estas tecnologías se les llaman middleware.

Qué es un middleware de comunicaciones

Un middleware es una capa de código que contiene un conjunto de servicios para permitir la interacción entre múltiples procesos ejecutándose en múltiples máquinas a través de una red de comunicaciones. Mediante la aplicación de un middleware, el usuario del mismo se abstrae de las complejidades inherentes a la heterogeneidad que puede suponer esta red de procesos y accede a un modelo de programación que imita la programación local.

Según la Internet Engineering Task Force (IETF) en 1997, se define middleware como: "Conjunto de servicios y funciones reutilizables, expandibles, que son comúnmente utilizadas por muchas aplicaciones para funcionar bien dentro de un ambiente interconectado".

En esta definición queda claro el enfoque a la reusabilidad y a la adecuación a estándares de comunicaciones bien probados.

A la hora de buscar una implementación real de un middleware, el usuario se puede encontrar con múltiples definiciones del mismo. Por ejemplo, algunos autores como [16] consideran que la API de comunicaciones Sockets Unix es

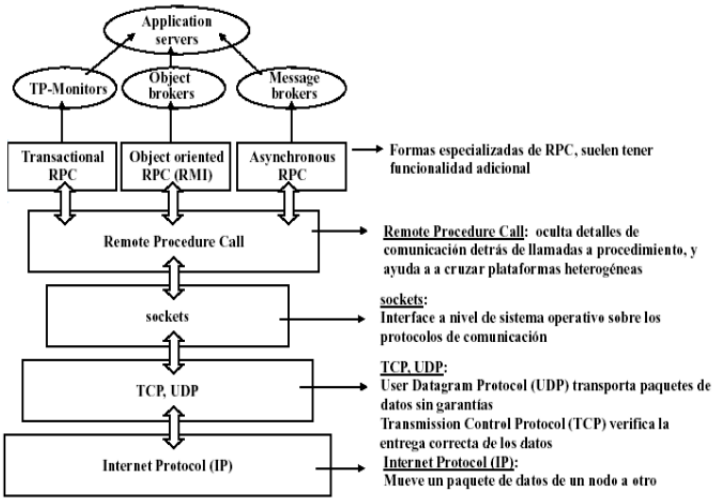

Figura 2.15: Organización en capas de un sistema de comunicaciones

un middleware de comunicaciones dado que abstrae detalles de uso de más bajo nivel. Esto es cierto y, en general, no se tiene un límite claro de qué es un middleware y qué es una librería por lo que en última instancia no siempre queda claro el uso del término.

En la Figura 2.15 se puede ver un ejemplo de organización entre las diferentes capas involucradas en la comunicación.

En primer lugar, se parte de la base de que existen dos aplicaciones, una que actúa como servidor, la mostrada en la Figura 2.15 y otra actuando como cliente. La aplicación de servidor hace uso de alguna tecnología middleware que, en este caso, se clasifican de tres formas:

- TP-monitors: también llamados monitores de procesamiento de transacciones. Esta filosofía mantiene un servidor como un punto de entrada a un proceso o transacción que se procesa en el tiempo. Son una adaptación de los monitores utilizados en el procesamiento multihilo pero, adaptados a la comunicación entre nodos en red. Algunos monitores famosos son: CICS de IBM, Pathway de Tandem, Top End de NCR... Esta arquitectura no será desarrollada en este documento

debido a su baja popularidad a día de hoy en las aplicaciones que se pretenden cubrir en el presente trabajo.

- Object brokers: este modelo es el más famoso debido a la adopción por medio de Corba y su implementación del Object Request Broker (ORB). Según esta arquitectura se proporciona un servicio que abstrae al usuario final de los problemas inherentes a la ejecución de un objeto remoto. Esto quiere decir, el usuario del middleware, del mismo modo que instancia un objeto local, instancia un objeto remoto. Cuando se ejecute la operación, el middleware se encargará de gestionar las labores de enrutamiento y de conexión para ejecutar dichas operaciones en el servidor que contiene el objeto y devolver al usuario los resultados de las operaciones invocadas. Existen varios middleware que mantienen esta filosofía como: Corba (especificación), ICE, orbix, RMI, Thrift...
- Message brokers: los brokers para el paso de mensajes son otro tipo de arquitectura en el que un módulo intermedio hace la función de router y administrador de comunicaciones. De este modo, los servicios se registran en el broker de mensajes, típicamente en un topic concreto, y esperan mensajes para procesarlos. Por otro lado, el cliente envía un mensaje al broker de comunicaciones indicando que servicio quiere consumir y el mensaje que quiere enviar. El broker, revisa en su registro como debe redireccionar este mensaje al servicio deseado y realiza las transformaciones necesarias al mensaje (en caso de que fuera necesario) para hacer llegar el mensaje al destinatario

En este trabajo se centrará el estudio en la arquitectura basada en Object Brokers debido a que, gracias al auge y adopción de los lenguajes de programación orientados a objetos, hoy en día este tipo de middleware resulta en una extensión natural de las capacidades de la orientación a objetos hacia el plano de la computación distribuida.

En concreto, en este trabajo centraremos el estudio en el middleware de comunicaciones Zeroc-Ice que, a día de hoy, resulta la alternativa más viable a la hora de usar un middleware orientado a objetos.

A continuación, y con el objetivo de asentar los conceptos principales alrededor de esta tecnología, en esta sección se realizará un breve repaso

de la tecnología proporcionada por Zeroc-Ice de modo que el lector pueda afrontar la labor de lectura del apartado dedicado al desarrollo del proyecto.

2.6.1. Zeroc-Ice, Caso de estudio

Desde el primer lenguaje de programación orientado a objetos, Simula 67, hasta los lenguajes más actuales como Python, Java, C++. La orientación a objetos ha demostrado ser un paradigma de programación que simplifica la labor de desarrollo debido a la transferencia directa desde el mundo de lo real al mundo virtual. Un objeto en la vida real, como por ejemplo un taladro, tiene una representación directa en un posible programa. De este modo, las acciones típicas de un taladro como: taladrar, percutir... pueden ser directamente trasladadas a código. Del mismo modo, las características del taladro y que afectan a su funcionamiento como son: potencia, tipo de broca... pueden ser, del mismo modo, trasladadas a código. Así, el desarrollador se puede centrar en describir su sistema con un conjunto de objetos que colaboran para llevar a cabo la función deseada.

A la hora de desarrollar sistemas distribuidos, el desarrollador tiene que lidiar con tareas puramente programáticas y que se alejan del nivel de abstracción proporcionado por la orientación a objetos explicada anteriormente. Esto hace que el programador reduzca su eficiencia y invierta mucho tiempo en labores repetitivas como son las labores de comunicación.

Zeroc-Ice nace de la unión de los conceptos de orientación a objetos y middleware de comunicaciones. Esta fusión fue establecida en primer lugar por CORBA, pero, al ver que el desarrollo de CORBA resultaba muy tedioso y lento en Zeroc-Ice decidieron iniciar su propio camino.

A día de hoy CORBA no se encuentra en desarrollo activo aunque los estándares siguen vigentes y únicamente se utiliza como modelo de inspiración para middleware de comunicaciones comerciales más ágiles.

Licencia

Zeroc-Ice se encuentra licenciado bajo dos licencias que conviene repasar debido al entorno sobre el que se está desarrollando este trabajo. Al igual que se remarcó, anteriormente, en la sección dedicada a los sistemas operativos, la importancia de las licencias en las tecnologías utilizadas para los sistemas de instrumentación, se resumirá de modo breve las dos licencias soportadas por Zeroc-Ice y que pueden ser consultadas en [23].

Zeroc-Ice, desde ahora ice, se distribuye bajo dos licencias de código libre y una licencia comercial.

Las licencias de código libre son GPLv2, para el conjunto de herramientas encargadas de proporcionar el Runtime encargado de gestionar ice, y BSD-3 para las herramientas de compilación proporcionadas por ice y que en la mayoría de proyectos únicamente formarán parte del proceso de compilación.

A la hora de elegir la licencia para ice, se deben responder a unas preguntas bastante claras y que resumen en el sitio web oficial. Estas preguntas son:

1. ¿El software del que ice formará parte está licenciado bajo GPLv2?: en caso de respuesta afirmativa no se podrá usar la licencia privativa dado que según as bases de GPLv2 esto no sería compatible.

2. ¿El software del que ice formará parte está licenciado bajo otra licencia open source?: en caso de responder negativamente, según la web oficial no se podría utilizar la licencia libre de ice. No obstante, esto no es del todo cierto dado que existen otras licencias de código libre compatibles con GPLv2 y por tanto sí podría ser usado.

3. ¿El software del que ice formará parte es software privativo?: en caso de responder afirmativamente no se podrá usar la licencia libre debido a que esto es contrario a la licencia GPLv2.

4. ¿El software del que formará parte ice no se distribuye a nadie?: en ese caso de responder afirmativamente ice podrá ser usado en ambas modalidades, privativamente o bajo software libre.

Resulta de vital importancia realizar estas preguntas y, en caso de duda, consultar con un abogado la viabilidad de utilizar esta tecnología en la aplicación que se desee realizar.

Conceptos básicos

En la Figura 2.16 se pueden ver todos los elementos de un sistema complejo utilizando las tecnologías proporcionadas por Ice. En esta sección se tratarán de aclarar los elementos básicos y fundamentales que el lector debe comprender para poder seguir el desarrollo del proyecto.

Tal y como se comentó anteriormente, ice es un middleware orientado a objetos. En función del lenguaje de programación utilizado la definición de una clase/objeto se realizará de un modo u otro. En Listado 1 se puede ver la definición de un objeto en el lenguaje Python.

```
1  class Bicicleta:
2      def __init__(self):
3          self.velocidad = 0
4
5      def pedalear(self, rpm):
6          self.velocidad = rpm*0.4
7
8      def frenar (self, presion, tiempo):
9          if self.velocidad > presion*tiempo/100 > 0:
10             self.velocidad = self.velocidad - presion*tiempo/100
11         else:
12             self.velocidad = 0
```

Listado 1: Ejemplo de clase en python

Ice, simplificando mucho y obviando muchos servicios extras que proporciona, únicamente entra en juego a la hora de hacer este objeto distribuido.

Un objeto distribuido es un objeto que está alojado en la memoria privada de un proceso y es invocado desde otro proceso. El lector podría esperar que

en la definición apareciera la palabra “red”, “Internet” o conceptos similares. Gracias a los modelos de abstracción explicados en la Sección 2.4 dedicada a las tecnologías de comunicación, desde el punto de vista de un computador, invocar un objeto en otro proceso dentro de un computador es prácticamente lo mismo que invocarlo en una computadora diferente. Únicamente varía el modelo de comunicación inter-proceso Inter Process Communication (IPC). Unix usa en muchos de sus servicios sockets de la familia UNIX. Estos sockets utilizan exactamente la misma API que los que se usarían para comunicarse entre dos computadores en zonas diferentes, únicamente varía el protocolo de la capa de transporte utilizado.

Siguiendo con la explicación, Ice ayuda al programador a abstraerse de dónde se encuentra el objeto distribuido que desea invocar y cómo se debe comunicar con él.

Ice tiene las capacidades para “adivinar” si un objeto está dentro de un mismo computador (en otro proceso) o está en otro computador y en función de eso utilizar un protocolo de la capa de transporte u otro.

Para ejemplificar, de forma clara, cuál es la misión de Ice, a continuación se describirá un caso de uso de comunicación sin el middleware Ice y otro caso utilizando el middleware Ice.

En este ejemplo el programador desea invocar un objeto de tipo bicicleta (como el visto anteriormente) en un ordenador que se encuentra en su laboratorio. El programador no utiliza ninguna tecnología como ice y decide utilizar los métodos tradicionales. Para llevar a cabo esta tarea realiza los siguientes pasos:

1. Crea un objeto del tipo “bicicleta”
2. Crea una función llamada pedalear_remoto, frenar_remoto, y convertir_objeto
    - convertir_objeto: esta función recibe como parámetro un objeto de tipo bicicleta y crea una secuencia de bytes con los atributos como la velocidad.
    - pedalear_remoto: recibe un objeto de tipo bicicleta, las revoluciones por minuto y un socket. La función llama a la función con-

vertir_objeto con el objeto bicicleta como parámetro y almacena los bytes del resultado. Junto a los bytes retornados por la función convertir_objeto se concatenan los bytes correspondientes a la rpm y utilizando la API de sockets se envían los datos hacia la dirección donde se encuentra el objeto, donde el programador deberá programa toda la lógica inversa para; gestionar los datos que llegan, construir el objeto en base a los datos recibidos, ejecutar la función correspondiente y enviar los resultados.

3. Gestiona las conexiones con el ordenador donde se encuentra el objeto remoto y todo lo referente a la seguridad y asegurar que la comunicación se realiza de forma fiable.

Por otro lado, en el caso de utilizar la tecnología ice el programador seguiría el siguiente flujo:

1. Inicializa el motor de comunicaciones Ice
2. Indica al motor que quiere obtener un objeto del tipo bicicleta del ordenador con nombre "prueba"
3. Invoca a la función pedalear del objeto bicicleta exactamente del mismo modo que lo haría si la comunicación fuera local.

Como se puede ver, en el segundo caso el programador no tiene porqué preocuparse de ningún tipo de cuestión relacionada con las comunicaciones. Para llevar a cabo todas estas cuestiones de forma transparente el usuario que utiliza ice debe tener en cuenta algunos conceptos claros como (ver Figura 2.16):

- Máquina que host: es la máquina que invoca un determinado objeto remoto, es el cliente del sistema de comunicaciones. Dentro de esta máquina los conceptos más importantes involucrados son:
  - ICE runtime: es el encargado de, entre otras muchas cosas, gestionar los diferentes métodos de comunicación. De este modo, si la comunicación se desea realizar por LoRa, el Ice Runtime generará

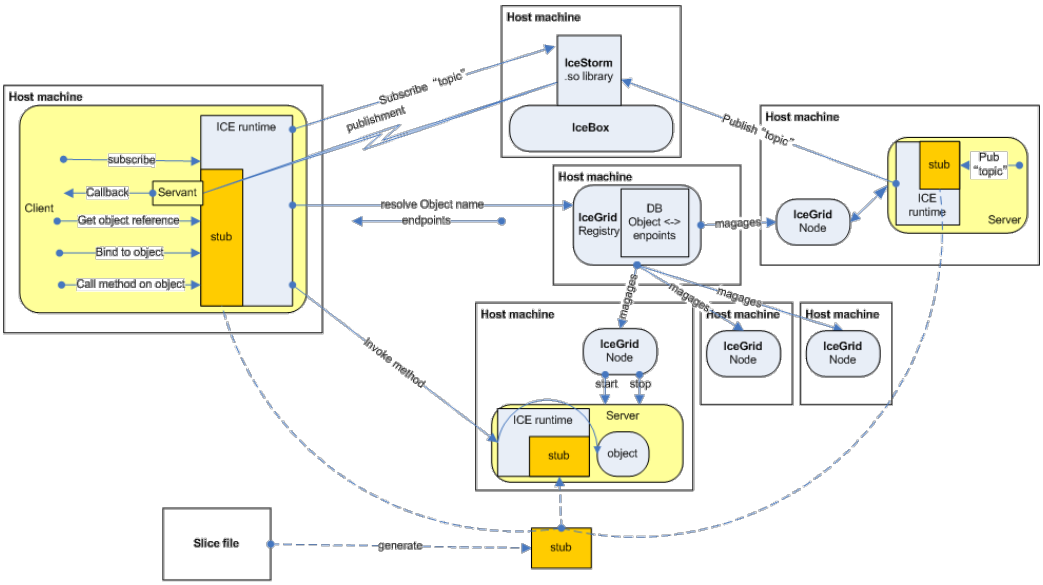

Figura 2.16: Elementos que intervienen en una comunicación con ZeroC Ice

las piezas de código necesarias para gestionar este tipo de comunicación. El runtime será el que realmente lance la petición por el medio de comunicación.

- Proxy: este es uno de los conceptos claves y que, de forma desafortunada, no aparece en la Figura 2.16. El proxy estaría situado dónde se sitúa en este caso la flecha con el nombre "Get object reference". Básicamente un proxy es un objeto con exactamente la misma forma que el objeto local pero que en vez de llamar a las funciones que llamaría en el caso de un objeto local, llama a las funciones proporcionadas por el stub.
- Stub: consiste en la implementación automática de las funciones que define el objeto (en el caso de la bicicleta, pedalear y frenar) y que se encargan de realizar el proceso de marshalling / unmarshalling y envío efectivo de datos:
  - Marshalling: es el proceso mediante el cual el objeto se convierte en una cadena de bytes que puede ser transmitido por el medio de comunicación.
  - Unmarshalling: es el proceso mediante el cual una cadena de

bytes se convierte a un objeto exactamente igual que los objetos locales.

- Máquina destino (servidor): es la máquina que contiene el objeto que se quiere invocar desde el cliente. En la figura se puede ver en la parte inferior. Los elementos importantes en la máquina destino son:
  - ICE runtime: exactamente del mismo modo que en el cliente es el encargado de orquestar toda la capa de comunicación del sistema.
  - Skeleton (no se muestra en la imagen): es el código generado de forma automática. El skeleton es el equivalente al proxy en el cliente, es decir, cuando en el servidor se utiliza el objeto remoto, por medio del skeleton, en realidad se está invocando a los métodos implementados dentro del stub y que se encargan del marshalling y el unmarshalling.
  - Stub: exactamente lo mismo que en el cliente.
  - Adaptador de objetos (no se muestra en la imagen): el adaptador de objetos es una parte fundamental en el código utilizado en el servidor. El adaptador de objetos se encarga de adaptar todas las invocaciones que se reciben a una instanciación concreta del objeto. Esto quiere decir, el servidor puede tener una gran cantidad de objetos distribuidos que desea mostrar al exterior. Mediante el adaptador de objetos el servidor indica a ice que implementa un determinado objeto distribuido y la dirección de memoria donde se encuentra dicho objeto. Así, cuando una petición llega al ICE Runtime este busca en el adaptador de objetos para ver en qué dirección de memoria se encuentra el objeto e invocar dicho objetos con los parámetros recibidos en la invocación.

Estos solo son algunos de los conceptos más importantes que se deben tener en cuenta a la hora de trabajar con ice. Sin embargo ice cubre muchos otros aspectos tales como la serialización y la deserialización que resultan de interés para este proyecto ya que determinan, en parte, la eficiencia de la comunicación.

Serialización y deserialización: Protocolo de comunicaciones Ice

La serialización y deserialización o, su equivalente en inglés, marshalling y unmarshalling respectivamente, son un punto muy importante a la hora de valorar el posible impacto que puede suponer una capa de abstracción como un middleware en el proceso de comunicación.

Existen muchas tecnologías dedicadas únicamente a la serialización y deserialización. Un ejemplo de esto puede ser Protobuffer o el novedoso proyecto de Google FlatBuffer [7].

En esta sección se explicará brevemente cómo implementa Ice la serialización y deserialización y que posibilidades proporciona.

En primer lugar, hay que dejar claro que "ice", como término, es el protocolo de comunicación de la capa de aplicación (del modelo TCP/IP) utilizado por el middleware para serializar y deserializar los objetos y las invocaciones a los objetos. El protocolo ice cuenta de tres partes bien diferenciadas:

- Reglas de codificación: cómo serán codificados los diferentes tipos de datos
- Tipos de mensajes: qué tipos de mensajes están permitidos en una conversación que utilice este protocolo
- Reglas de temporización y secuencia: cómo y en qué momento se deben comunicar cada actor involucrado en la comunicación.

Las reglas de codificación de ice han sido diseñadas desde el primer momento para ser simples y eficientes. Aunque resulte extraño en ice han optado por utilizar little-endian por defecto en sus tipos de datos. Esto se debe a que la mayoría de máquinas utilizan little endian y por lo tanto a la hora de hacer la serialización y deserialización se evita un paso de conversión. Aquí se puede ver claramente como en ZeroC todas las decisiones de diseño están bien documentadas y tienen detrás un porqué claro.

Los tipos de datos definidos por Ice son: bool, byte, short, int, long, float y double. Independientemente de la máquina host final, estos tipos de

datos serán codificados en little endian. Ice define el tamaño de estos tipos de datos como: 1 byte, 1 byte, 2 bytes, 4 bytes, 8 bytes, 4 bytes y 8 bytes respectivamente. En el caso de los valores de coma flotante siguen el estándar IEEE 758.

En el caso de los strings, Ice da soporte a los mismos añadiendo al inicio el tamaño del string seguido del contenido formateado en UTF-8. Con el objetivo de reducir el tamaño las cadenas no se terminan con el carácter nulo.

Ice también proporciona soporte para tipos de datos más complejos como: clases, interfaces, estructuras, enumeradores... Si el lector desea profundizar puede consultar la documentación en [21].

Junto a las reglas de codificación, Ice establece una serie de mensajes en su protocolo. A continuación se dará una breve explicación de cada uno de estos mensajes.

Ice define cinco tipos de mensajes:

1. Request
2. Batch Request
3. Reply
4. Validate connection
5. Close connection

No todos los mensajes se utilizan en todos los casos. Por ejemplo, los mensajes validate connection y close connection únicamente se utilizan en aquellos protocolos de comunicación orientados a conexión (por ejemplo TCP).

Todos los mensajes (salvo validate connection y close connection) están formados por una cabecera y un cuerpo.

La cabecera del mensaje está formada por 14 bytes, estos 14 bytes son:

- magic (4 bytes): constante con el valor de “I” “c” “e” “P”

- protocolMajor (1 byte): el número de revisión mayor
- protocolMinor (1 byte): el número de revisión menor
- encodingMajor (1 byte): el número de revisión mayor de las reglas de codificación
- encodingMinor (1 byte): el número de revisión menor de las reglas de codificación
- messageType (1 byte): el tipo de mensaje que puede ser
  - 0: request
  - 1: batch request
  - 2: reply
  - 3: validate connection
  - 4: close connection
- compressionStatus (1 byte): estado de la compresión
- messageSize (1 byte): el tamaño en bytes incluyendo la cabecera.

Seguido a la cabecera aparece el cuerpo del mensaje que en función del tipo de mensaje variará. Debido a que esto es una cuestión de muy bajo nivel se deja al lector completar la información con la documentación oficial que puede ser consultada desde `https://doc.zeroc.com/ice/3.6/the-ice-protocol/protocol-message`.

Finalmente cabe destacar que ice tiene capacidades de compresión, tal y como se ha visto en la cabecera del mensaje. La compresión no siempre estará disponible y es, por tanto, una capacidad opcional del protocolo.

Según la especificación del algoritmo la compresión utilizada es bzip2. Si la compresión se aplica el campo de longitud del mensaje expresará el tamaño del mensaje comprimido.

Servicios y ecosistema ZeroC Ice

Ice proporciona principalmente la plataforma para la comunicación entre objetos distribuidos de manera sencilla y eficiente. Pero ZeroC no solo ha diseñado Ice si no que ha creado un ecosistema en torno a esta plataforma mediante la implementación de servicios, que precisamente, están implementados utilizando el propio middleware de comunicaciones.

Esto es de vital importancia en muchas aplicaciones y hace que ZeroC Ice se distinga de otras alternativas que únicamente proporcionan capacidades de Remove Procedure Call (RPC) como GRPC. Estos servicios van desde la capacidad de gestionar los servidores, distribución de carga, distribución de eventos, hasta gestión de la configuración de la aplicación distribuida.

Los servicios más usados proporcionados por ice son:

- IceGrid: icegrid consiste en una implementación del servicio de localización descrito por ZeroC-Ice en [22]. Mediante este servicio se proporciona capacidades de indirección mediante información simbólica del objeto. Esto significa que, usando este servicio, cada servidor se registra en un "registry" y proporciona un nombre y una descripción de qué objetos tiene asociados. De esta manera, cuando un cliente realiza una invocación remota, puede realizar dicha invocación con respecto a este servicio y será icegrid el encargado de redireccionar al servidor final mediante diferentes políticas.
- IceStorm: icetorm permite, en cierto modo, cambiar el paradigma de comunicaciones implementando un sistema de eventos. De este modo, un cliente publica eventos en un determinado topic al que otros clientes se suscriben para recibir actualizaciones (patrón pub-sub). Este servicio es la respuesta de ZeroC-Ice al Internet Of Things (IOT).
- IcePatch2: proporciona una implementación de Over The Air (OTA). IcePath permite a los clientes recibir las actualizaciones de diferentes piezas de código.

Ventajas y desventajas de utilizar Ice

Finalmente, en esta sección y a modo de resumen se mostrarán las ventajas y desventajas de utilizar zeroc-ice en los desarrollo.

Entre las ventajas se puede destacar:

- Orientación a objetos: aunque esto pueda ser desventaja para algunos, lo cierto es que la orientación a objetos es un paradigma de programación muy utilizado y que, mediante su uso, permite reducir de forma considerable el tiempo de desarrollo. Por otro lado, los desarrolladores que no estén acostumbrados a usar esta metodología pueden ver en este caso una desventaja. No obstante, la mayor parte de los desarrolladores suelen tener nociones básicas de orientación a objetos.
- Comunicación síncrona y asíncrona: esta característica resulta de vital importancia en muchas aplicaciones desarrolladas en los sistemas embebidos. Soportando estos modelos de comunicación se permite que el servidor se comunique con el cliente de forma proactiva.
- Programación independiente de la arquitectura utilizada: tal y como se ha podido observar los mecanismos de comunicación son completamente agnósticos de la plataforma utilizada. De este modo, la solución desarrollada para un sistema puede ser portada a otra plataforma si esta última estuviera soportada por ZeroC Ice. A día de hoy las plataformas más importantes como x86, ARM e incluso una gran gama de microcontroladores está soportada por ice.
- Independiente del lenguaje de programación: esta característica resulta de gran utilidad debido a que, gracias a la filosofía de diseño planteada por ZeroC-Ice en su middleware, cada pieza del sistema puede estar programada en un lenguaje de programación diferente. Por poner un ejemplo que ilustre de manera rápida las ventajas de esta característica podría ser el caso de la Red Pitaya. El servidor en el ARM podría estar escrito en C o C++. Sin embargo, el cliente puede estar escrito en Python, Java, Javascript... De este modo, de una forma uniforme, se podría dar soporte a nuevos desarrolladores que sean eficientes con una tecnología concreta.

- Soporte multihilo: esta capacidad resulta de mayor interés en aplicaciones de escritorio donde es bastante común tener varios hilos ejecutándose simultáneamente. En este caso ice asegura que su comportamiento se mantendrá coherente independientemente de la interacción con los diferentes hilos del sistema.
- Transparencia de localización: Mediante la descripción de los objetos por medio de una cadena de texto y el soporte del servicio icegrid, se proporciona una suite completa que permite abstraer al programador de los problemas de acceder a una localización concreta. El desarrollador únicamente tiene que preocuparse por indicar al sistema el tipo de objeto que se desea invocar y el runtime se encargará de localizarlo.
- Seguridad: ZeroC-ice ha incorporado el soporte para utilizar encriptación SSL de tal modo que las aplicaciones puedan ser utilizadas incluso en entorno públicos.

Por otro lado, las desventajas de usar un middleware como este son:

- Licencia: la licencia aunque dual puede no ser la más adecuada para algunos proyectos de investigación o proyectos de desarrollo. Este hecho hace que muchos desarrolladores decidan no usar el sistema.
- Falta de ejemplos: aunque la documentación oficial es bastante buena, resulta muy complicado acceder a documentos de formación con ejemplos de uso concreto.
- Performance: aunque, tal y como ser estudiará en la Sección 3 la performance del middleware ZeroC-Ice es de las mejores del mercado con respecto a otras alternativas como Thrift o Cap n' proto.

Capítulo 3

Desarrollo

En este capítulo se realizará un resumen de los experimentos realizados durante el desarrollo del proyecto. Esta parte del documento pretende ser lo más parecido posible a un reporte técnico donde se muestre todo el proceso deductivo y las motivaciones que han determinado las diferentes decisiones de diseño tomadas.

De este modo, en este capítulo el lector encontrará un conjunto de experimentos en el orden exacto en que fueron realizados por el autor del documento. Cada experimento se encuentra estructurado del mismo modo, aunque pueden encontrarse pequeñas variaciones en el caso de que fuera necesario según la naturaleza del documento.

En general el lector encontrará en cada experimento la siguiente estructura:

- Descripción del experimento / Objetivos: en esta sección se detallará el objetivo y motivación del experimento realizado. La motivación permitirá que el lector pueda seguir la historia del proyecto, por otro lado, el objetivo ayudará al lector a poder replicar el experimento con el objeto de mostrar diferentes aproximaciones al mismo problema.
- Hipótesis: en esta sección se detallará la hipótesis planteada **antes de realizar el experimento** por el autor.

- Desarrollo del experimento: se mostrarán todos los detalles relevantes en el desarrollo del proyecto de modo que el lector pueda juzgar la validez de los resultados obtenidos y se puedan derivar nuevos experimentos en base a modificaciones en el desarrollo aquí realizado.
- Resultados: en la medida de lo posible se tratarán de dar resultados tabulados o lo más planos posibles de modo que sirvan de comparación entre los diferentes experimentos realizados en el proyecto.
- Conclusiones: finalmente, al terminar cada experimento se realizará un breve desarrollo de las conclusiones obtenidas por el autor en cada experimento. En algunos casos, fruto de estas conclusiones se propondrá nuevos experimentos que pueden estar cubiertos en este proyecto o archivados como **trabajo futuro**

El objetivo de esta estructura en el desarrollo es el de mostrar la naturaleza de la metodología seguida en el desarrollo del proyecto. Así, en vez de mostrar, una base teórica sobre metodologías formales, de lejos estudiadas en infinidad de documentos existentes, se pretende mostrar la metodología **real** seguida durante la realización del proyecto.

Finalmente, en la sección dedicada a las conclusiones finales se tratará de dar una visión global de las conclusiones obtenidas como resultado del proyecto y se sienten las bases para posibles trabajos futuros.

3.1. Experimento 1: LWIP RAW TCP Servidor

3.1.1. Descripción del experimento / Objetivos

En este proyecto se busca simplificar la labor de la comunicación entre el PC y el equipo de medición. Los objetivos principales del proyecto son: proporcionar un sistema simple y reusable y medir el impacto en la sobrecarga que supone al sistema todas las capas de abstracción añadidas.

En un sistema tan complejo como es una FPGA resulta de vital importancia tener una escenario común sobre el que hacer las diferentes pruebas. No obstante, a la hora de caracterizar esta sobrecarga no influye únicamente el hardware si no que, como es obvio, también influye el software implicado en las labores de comunicación.

Teniendo en cuenta lo anterior, en este experimento se busca caracterizar la velocidad[1] máxima que se puede obtener en este sistema con los métodos tradicionales en cuanto a la comunicación entre el dispositivo de medición y el ordenador de control.

Para simplificar lo máximo posible el desarrollo se usará el servidor iperf realizado por Xilinx para la Zedboard. El utilizar este servidor permitirá realizar medidas con una herramienta ampliamente utilizada en este campo.

Este experimento se ve motivado, en parte, por la necesidad de saber, de forma independiente de la aplicación concreta, las características de la estrategia de comunicación seguida por los diferentes proyectos realizados con el laboratorio de investigación MLAB.

3.1.2. Hipótesis

En este experimento se espera saturar las capacidades del enlace físico entre el computador y el sistema de medición (en este caso únicamente formado por una placa de desarrollo Zedboard). La capacidad del enlace es de 1Gbps.

Se espera alcanzar la velocidad máxima del enlace debido a que se están utilizando las capas más bajas de abstracción posible. Esto quiere decir que cualquier capa adicional supondrá algún tipo de penalización en el sistema.

Se espera que se tenga que realizar alguna labor de *tunning* de parámetros tales como el ancho de ventana, tamaños de memoria... Dado que esta aplicación pedirá los requisitos máximos del sistema.

[1] En este documento, cuando se hable de velocidad de forma genérica se hará referencia al conjunto latencia/ancho de banda obtenido del sistema

3.1.3. Desarrollo del experimento

En primer lugar, se recuerda al lector que todo el código de cada uno de los experimentos se puede encontrar como anexo digital al presente documento. En caso de no disponer del anexo digital contacte con el autor del documento para que se envié una copia del mismo. Es importante que el lector tenga en cuenta que en este documento no se ha incluido todo el código de cada uno de los experimentos debido a que esto haría que el documento fuera complicado de leer. En esta sección únicamente se mostrarán las partes que el autor ha considerado relevantes para la realización del experimento.

Implementación hardware

El primer paso consiste en la definición del hardware que se utilizará para los diferentes experimentos. Por ser este el primer experimento, se detallará el proceso de creación del hardware así como los puntos más importantes a tener en cuenta y que puede que afecten de alguna forma a los resultados.

El *block design* diseñado se puede ver en la Figura 3.1. Tal y como se puede observar, este block design únicamente contiene un elemento, la unidad de procesamiento Zynq. En realidad este bloque no se implementa en la lógica reconfigurable de la placa si no que es un bloque hardware implementado dentro del SoC proporcionado por Xilinx.

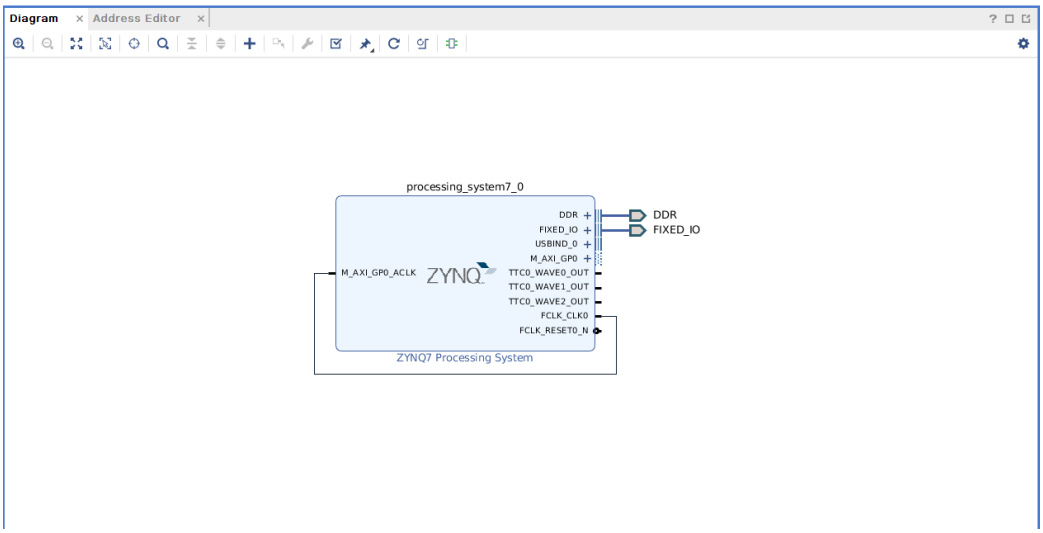

Figura 3.1: Block Design de referencia

Dentro del bloque, en la Figura 3.2 se puede ver la configuración utilizada para el sistema de procesamiento incorporado en la Zynq. Se han restado tanto los elementos incorporados al diseño como aquellos que no se han utilizado.

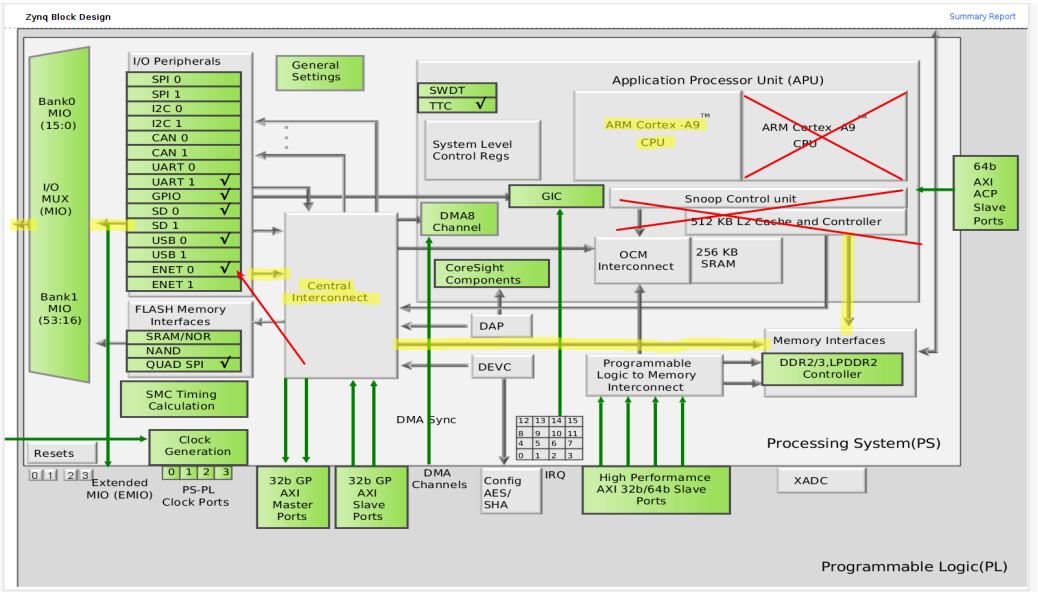

Figura 3.2: Configuración de referencia del bloque de procesamiento

La primera **decisión de diseño** tomada en esta fase fue la de sentar los elementos de comunicación que serán utilizados.

Como se puede observar en la Figura 3.2, se ha representado con una flecha roja la selección de un elemento de entrada/salida llamado “ENET0”. Este elemento, está conectado al controlador MAC marvell incorporado a la placa Zedboard. Este controlador se puede ver señalado en la Figura 3.3.

La razón por la que se ha utilizado este controlador y no otro es porque es la configuración por defecto de la placa y añade menos variables de control en el diseño. Este controlador tiene una serie de ventajas y desventajas con respecto a otros controladores que podrían ser implementados como bloques reconfigurables dentro del Programable Logic (PL). Las principales desventajas detectadas son:

1. Jumbo frame: Este controlador carece de capacidades para jumbo frames. Jumbo frame es una característica del estándar Ethernet que permite ampliar la Maximum Transfer Unit (MTU) del sistema de 1500 byes a 16KiB de datos.

2. Hasta 1 Gbps: El ancho máximo del controlador es de 1 Gbps. Otros controladores implementados en el PL proporcionan mayores velocidades.

Por otro lado las ventajas de utilizar este controlador en el diseño son:

1. Direct Memory Access (DMA) Incorporado: El propio controlador tiene incorporado un DMA que libera la procesador del procesamiento de las tramas y del movimiento a la memoria DDR del sistema.
2. Drivers y ejemplos preparados: La mayoría de ejemplos así como drivers de comunicación y capas de *porting* están basadas en este driver.

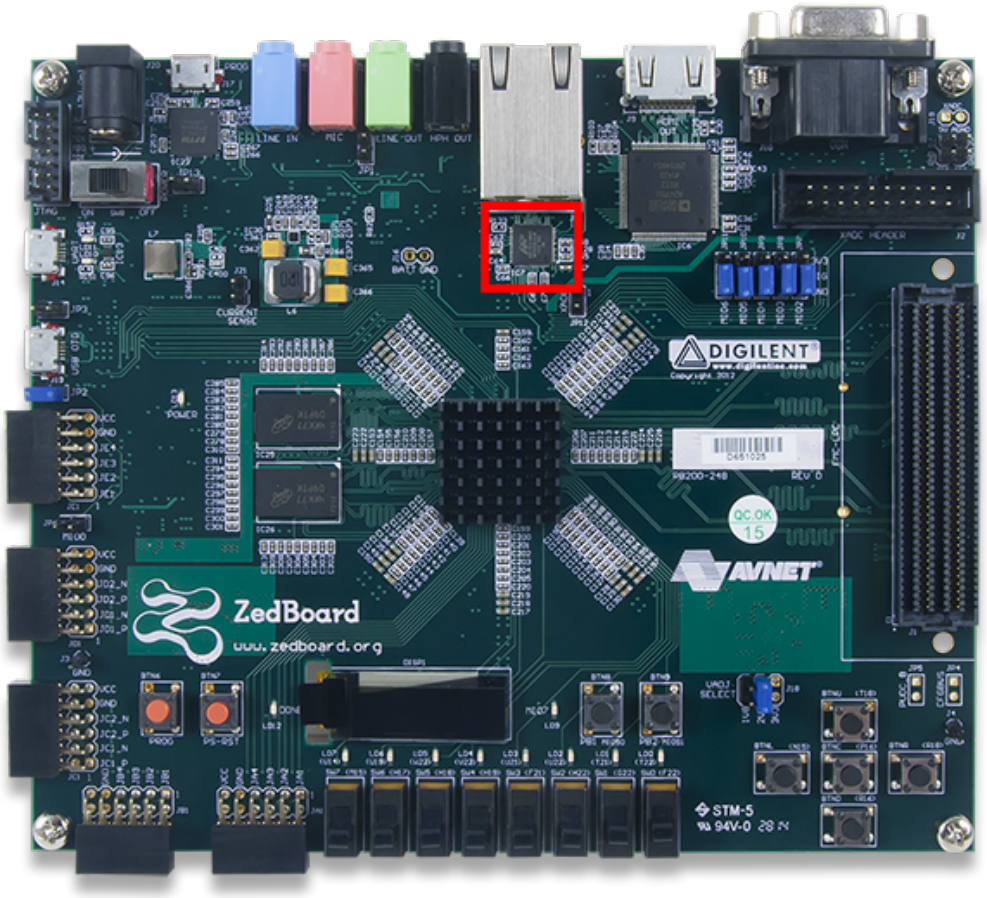

Figura 3.3: Ubicación del dispositivo MAC Marvell

En la Figura 3.3 se puede ver la ubicación del controlador MAC encargado de transmitir la trama Ethernet al dispositivo final.

Finalmente, para terminar de describir el hardware en la Figura 3.4 se puede ver el camino que seguiría una trama desde el conector Ethernet RJ-45 hasta el procesador donde se realizaría el tratamiento de la información.

Todos estos detalles son de vital importancia dado que determinarán cuestiones bastante importantes a la hora de entender los diferentes elementos que intervienen en la comunicación como:

- ¿Quién comprueba si una trama debe ser procesada (el destino es la placa) o no?: En este caso, leyendo el datasheet del MAC se llega a la conclusión de que el encargado de realizar esta labor es el propio MAC. Esto quiere decir que el procesador **no tendrá que realizar una comprobación de las cabeceras Ethernet** debido a que esto pasa a ser competencia del dispositivo MAC.
- ¿Quién procesa la trama?: Esta pregunta se puede responder mirando el datasheet del MAC. En el mismo se puede ver como el propio MAC es el encargado de comprobar el Cyclic Redundancy Check (CRC) de la trama por lo que esto **no supondrá una sobrecarga para el procesador**.
- ¿Cómo se procesan las capas superiores?: A estas alturas del desarrollo no se puede resolver esta pregunta y, en principio, en el datasheet no se hace referencia a ningún procesamiento de las capas superiores, esto queda como **tarea del procesador**.

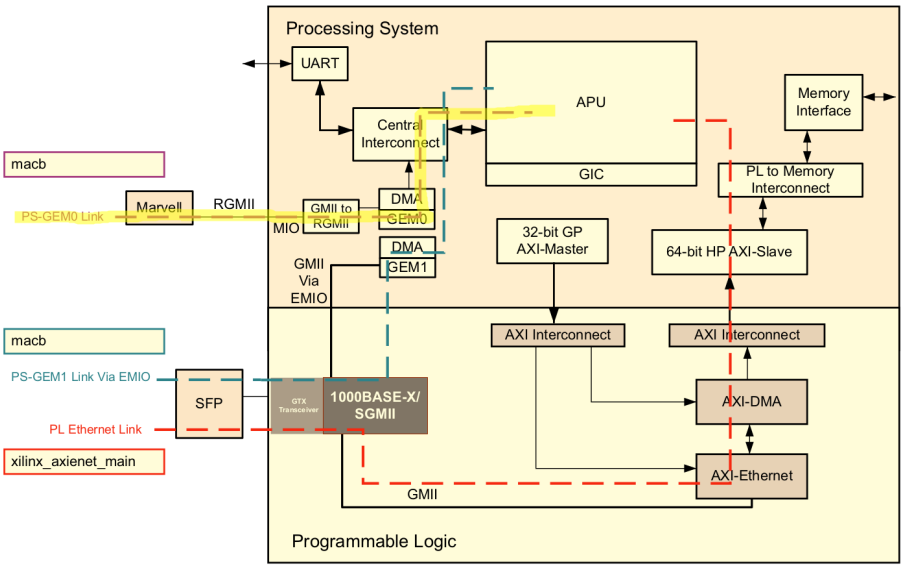

Figura 3.4: "Camino" recorrido por una trama desde el conector al procesador

Con todos estos detalles en mente el siguiente paso antes de pasar al código consiste en exportar el hardware. Exportar el hardware significa úni-

camente indicar al Software Development Kit (SDK) así como a las diferentes herramientas de desarrollo proporcionadas por Xilinx.

El fichero que describe el hardware es un fichero con extensión ".hdf". Este fichero tiene la forma mostrada en la Figura 3.5. Este fichero lo utilizará el SDK para generar los drivers y el BSP utilizado durante el desarrollo. Entender este proceso es esencial dado que será aquí donde se podrán realizará la mayoría de modificaciones para ver los cambios en la eficiencia de la comunicación.

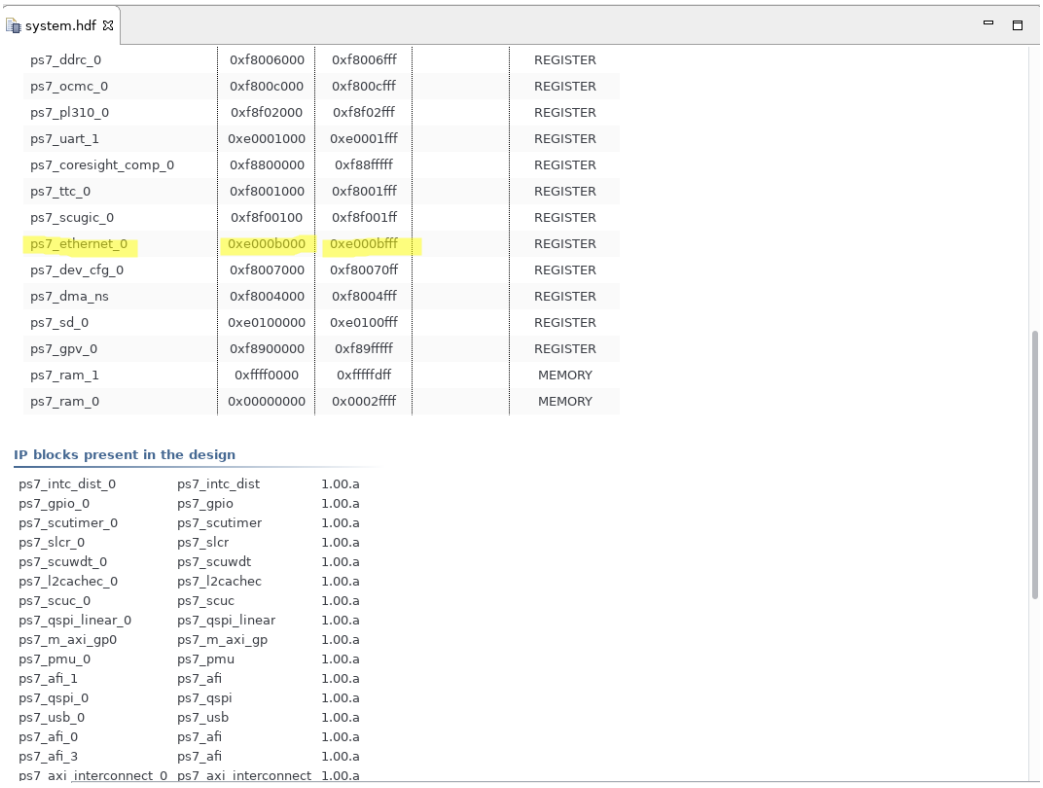

Figura 3.5: Fichero de descripción del hardware

Implementación del software — Servidor en Zedboard

Con el hardware ya implementado, el siguiente paso consistirá en implementar el servidor que se encargará de recibir las invocaciones del cliente con

datos y procesarlas. De este modo se establecen los siguientes roles:

- Zedboard — Servidor: La zedboard o, en general, el instrumento de medición será el servidor de la comunicación. La comunicación no se iniciará hasta que el cliente realice una invocación. Además, en este caso también se define el flujo de los datos. Con la Zedboard como servidor se simula la transferencia de datos que podría darse entre un DAC y un ordenador de gestión. El ordenador mandará la señal que quiere que el DAC convierta.
- Ordenador — Cliente: El ordenador, por tanto, es el cliente en la comunicación y será el encargado de enviar los datos al instrumento de medición.

A continuación se detallará la estructura del proyecto software. Este proyecto, tal y como se ha comentado en los objetivos, utiliza como base los ejemplos proporcionados por Xilinx dado que son la base sobre las que se harán las diferentes medidas en los siguientes proyectos.

Para generar el proyecto el lector puede ir al generador automático de Xilinx para nuevas aplicaciones. En este generador únicamente deberá seleccionar el servidor LightWeight IP (LWIP) TCP, en este caso.

Como resultado el lector debería tener algo parecido a la Figura 3.6

La estructura del proyecto se puede resumir en:

- BSP: El BSP es el proyecto autogenerado por Xilinx a partir del Hardware Definition Format (HDF). Este proyecto contiene todos los drivers necesarios para gestionar los diferentes módulos harware incluidos, como el ethernet.
- Servidor TCP: El servidor iperf es el proyecto principal y está vinculado al BSP de modo que ambos forman un único proyecto.

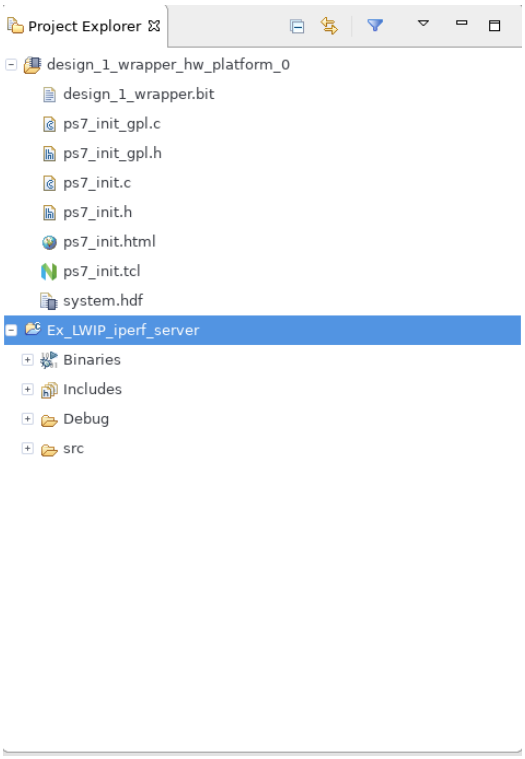

Figura 3.6: Estructura del proyecto

Detalles de la implementación

Debido a que este es el primer experimento realizado tanto con LWIP como con la placa que realiza las labores de instrumentación, en esta sección se expicará brevemente la estructura del programa realizado mostrando, siempre que sea relevante, el código utilizado. En experimentos futuros se partirá de la lectura de este experimento y se mostrarán únicamente las modificaciones realizadas sobre este proyecto base.

En primer lugar, el enlazador o *linker* busca en el código por una función *main* esta función será la encargada de ejecutar el código de configuración de la aplicación. En el Listado 2 se puede un ejemplo de fichero *main.c*. A continuación se explicarán las partes más importantes del mismo.

Lo primero que se definen son las direcciones que serán utilizadas por la capa IP. Esta parte es opcional y normalmente se adquirirán mediante el protocolo Dynamic Host Configuration Protocol (DHCP). No obstante, con el objeto de simplificar al máximo posible las capas de software involucradas en el experimento, se ha omitido el uso de este protocolo.

Lo siguiente que se puede apreciar en el código es la definición de dos variables externas que son: *TcpFastTmrFlag* y *TcpSlowTmrFlag*. Estas variables se han definido con el atributo de *volatile*. Este atributo permite "saber" al compilador que la variable se está actualizando desde algún lugar aunque no aparezca en el código que él interpreta.

Más adelante, se verá dónde se actualiza esta variable y el porqué.

Siguiendo el flujo del código, en la Línea 13 se puede observar la definición de una estructura de LWIP. Esta estructura tiene los siguientes campos (únicamente se mostrarán los campos más importantes para este trabajo, si el lector desea comprobar todo los campos puede encontrar toda la información en la documentación oficial):

- ip_addr, netmask, gw: estos tres parámetros son los encargados de configurar la dirección IP de la interfaz. La dirección IP se utiliza en el nivel 3 del modelo de referencia OSI.

- input: un puntero a función que será utilizado por el *driver* de la tarjeta de red para pasar el procesamiento de las diferentes capas a LWIP.
- output: un puntero a función que será utilizado por la capa 3 del modelo *osi* para continuar con el procesamiento en el driver del dispositivo. En este punto se ve claramente cómo a partir de nivel 3 se delega el procesamiento al MAC.
- linkoutput: este puntero a función se utiliza cuando se quiere mandar un paquete tal cual está, es decir, sin ningún tipo de procesamiento por los niveles inferiores del MAC.
- mtu: configura la MTU de la tajeta de red. En este caso, tal y como se verá más adelante se configurará para 1500 bytes.
- hwaddr: configura la dirección física de la interfaz de red. Esta dirección es la utilizada por el MAC para comprobar si la trama que circula por el cable debe ser procesada por niveles superiores o no.

Como el lector podrá imaginar, configurar todas estas "a mano" puede generar una gran cantidad de errores. Para ello, Xilnx, proporciona una función para configurar todos estos parámetros en función de la interfaz de red utilizada. Esta es la razón por la que, en la Línea 48 se indica la dirección de memoria donde se encuentra el MAC utilizado.

La función proporcionada por Xilinx llamará internamente a la función de LWIP *netif_add* que terminará de configurar los diferentes *callback* o punteros a función encargados de gestionar la comunicación.

Siguiendo con las partes más importantes del código se llega a la Línea 56. Esta función se encarga de iniciar los *timer* encargados de actualizar los *timestamp* internos de la capa TCP con el objetivo de mantener las diferentes políticas de gestión de flujo.

Una vez configurados los diferentes timers involucrados en las políticas de gestión de flujo de TCP se inicia la "aplicación". En realidad esta aplicación no es nada especial tal y como se verá a continuación.

En el Listado 3 se puede ver una parte del código dentro de la aplicación. En este apartado únicamente se explicará los aspectos generales de esta parte

del código y según se vaya requiriendo en el documento se ahondará en los conceptos necesarios.

En primer lugar, se puede observar como se crea un bloque Protocol Control Block (PCB) del tipo LWIP. Este bloque es el encargado de mantener los detalles de una capa de procesamiento concreta. En este caso, el PCB se utilizará para mantener el estado de una conexión de tipo tcp. Una vez definido el bloque de datos, se realiza el proceso de *bind*. Este proceso asocia un bloque de datos a una dirección IP y un puerto. Se recuerda al lector que una conexión, en el nivel de transporte, queda definida por una IP (indica al nivel 3 de la capa OSI dónde se encuentra (lógicamente) el dispositivo) y un perto. Este último identifica dentro de un mismo dispositivo, la aplicación/proceso que atenderá dicha "conversación".

Una vez realizado el proceso de *binding*, el siguiente paso consiste en pasar la conexión al estado de *LISTEN*. En este estado, la lógica implementada en la capa TCP admitirá nuevas conexiones. Con esta llamada la pila de gestión se quedaría bloqueada esperando clientes.

Es importante hacer énfasis en la palabra **bloqueada**. Esto es así en la mayoría de sistemas operativos de propósito general. Este bloqueo simplifica la implementación de aplicaciones dado que, el programador sabe que cuando se ejecute la siguiente línea, habrá un cliente "escuchando" en el otro lado. Sin embargo, esta política no es adecuada para los sistemas empotrados debido a que esto significaría que no se podrían realizar acciones de otro tipo como, por ejemplo, el sensado de alguna variable.

Por esta razón, LWIP, adopta una filosofía basada en interrupciones software o callbacks de modo que no se deba bloquear. Una de estas interrupciones se configura en la Línea 35 que indica a LWIP qué función se debe ejecutar cuando aparezca un nuevo cliente. En este caso, la función ejecutada es tcp_accept_server. Esta función, tal y como se puede observar en el código anterior es la encargada de atender a cada cliente "nuevo". De este modo y, para el ejemplo que aquí se discute, se inicializan diferentes variables que permitirán sacar estadísticas sobre el uso de la comunicación.

En las Líneas 64–66 se realiza la parte más importante de esta fase, se indica a LWIP qué funciones debera ejecutar en cada caso. En este caso,

como el servidor es pasivo y no realiza el envío de ningún dato, únicamente se configuran los callback para la recepción y el error de conexión.

Con todo esto configurado el procesador volverá al *main*. En este punto se iniciará un bucle infinito que únicamente se encargará de actualizar los timer (comprobando un flag actualizado dentro de la Interrupt Software Rutine (ISR)) y comprobar si existen nuevos paquetes en el buffer de entrada.

```
1  #define DEFAULT_IP_ADDRESS        "192.168.0.10"
2  #define DEFAULT_IP_MASK                "255.255.255.0"
3  #define DEFAULT_GW_ADDRESS        "192.168.0.1"
4
5  extern volatile int TcpFastTmrFlag;
6  extern volatile int TcpSlowTmrFlag;
7
8
9  void platform_enable_interrupts(void);
10 void start_application(void);
11
12
13 struct netif server_netif;
14
15
16 static void assign_default_ip(ip_addr_t *ip, ip_addr_t *mask, ip_addr_t
   ↪  *gw)
17 {
18         int err;
19
20         xil_printf("Configuring default IP %s \r\n", DEFAULT_IP_ADDRESS);
21
22         err = inet_aton(DEFAULT_IP_ADDRESS, ip);
23         if (!err)
24                 xil_printf("Invalid default IP address: %d\r\n", err);
25
26         err = inet_aton(DEFAULT_IP_MASK, mask);
27         if (!err)
28                 xil_printf("Invalid default IP MASK: %d\r\n", err);
29
```

```
30          err = inet_aton(DEFAULT_GW_ADDRESS, gw);
31          if (!err)
32                  xil_printf("Invalid default gateway address: %d\r\n",
                    ↪  err);
33  }
34
35  int main(void)
36  {
37          struct netif *netif;
38
39          unsigned char mac_ethernet_address[] = {
40                  0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };
41
42          netif = &server_netif;
43
44          init_platform();
45
46          lwip_init();
47
48       if (!xemac_add(netif, NULL, NULL, NULL, mac_ethernet_address,
49                                  PLATFORM_EMAC_BASEADDR)) {
50                  xil_printf("Error adding N/W interface\r\n");
51                  return -1;
52          }
53
54          netif_set_default(netif);
55
56      platform_enable_interrupts();
57
58          netif_set_up(netif);
59
60          assign_default_ip(&(netif->ip_addr), &(netif->netmask),
            ↪  &(netif->gw));
61
62          start_application();
63
64          while (1) {
65                  if (TcpFastTmrFlag) {
```

```
66                      tcp_fasttmr();
67                      TcpFastTmrFlag = 0;
68              }
69              if (TcpSlowTmrFlag) {
70                      tcp_slowtmr();
71                      TcpSlowTmrFlag = 0;
72              }
73              xemacif_input(netif);
74      }
75
76      /* never reached */
77      cleanup_platform();
78
79      return 0;
80  }
```

Listado 2: Código de inicialización en main.c

```
1   void start_application(void)
2   {
3           err_t err;
4           struct tcp_pcb *pcb, *lpcb;
5
6           /* Create Server PCB */
7           pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
8           if (!pcb) {
9                   xil_printf("TCP server: Error creating PCB. Out of
                    ↪  Memory\r\n");
10                  return;
11          }
12
13          err = tcp_bind(pcb, IP_ADDR_ANY, TCP_CONN_PORT);
14          if (err != ERR_OK) {
15                  xil_printf("TCP server: Unable to bind to port %d: "
16                                  "err = %d\r\n" , TCP_CONN_PORT, err);
17                  tcp_close(pcb);
18                  return;
```

```
19          }
20
21          /* Set connection queue limit to 1 to serve
22           * one client at a time
23           */
24          lpcb = tcp_listen_with_backlog(pcb, 1);
25          if (!lpcb) {
26                  xil_printf("TCP server: Out of memory while
                    ↪  tcp_listen\r\n");
27                  tcp_close(pcb);
28                  return;
29          }
30
31          /* we do not need any arguments to callback functions */
32          tcp_arg(lpcb, NULL);
33
34          /* specify callback to use for incoming connections */
35      tcp_accept(lpcb, tcp_server_accept);
36
37          return;
38  }
39  static err_t tcp_server_accept(void *arg, struct tcp_pcb *newpcb, err_t
    ↪  err)
40  {
41          if ((err != ERR_OK)  (newpcb == NULL)) {
42                  return ERR_VAL;
43          }
44          /* Save connected client PCB */
45          c_pcb = newpcb;
46
47          /* Save start time for final report */
48          server.start_time = get_time_ms();
49          server.end_time = 0; /* ms */
50          /* Update connected client ID */
51          server.client_id++;
52          server.total_bytes = 0;
53
54          /* Initialize Interim report paramters */
```

```
55          server.i_report.report_interval_time =
56                  INTERIM_REPORT_INTERVAL * 1000; /* ms */
57          server.i_report.last_report_time = 0;
58          server.i_report.start_time = 0;
59          server.i_report.total_bytes = 0;
60
61          print_tcp_conn_stats();
62
63          /* setup callbacks for tcp rx connection */
64          tcp_arg(c_pcb, NULL);
65          tcp_recv(c_pcb, tcp_recv_perf_traffic);
66          tcp_err(c_pcb, tcp_server_err);
67
68          return ERR_OK;
69  }
```

Listado 3: Código de iniciación de la aplicación

Con todo esto la placa está lista para recibir paquetes y procesarlos. Como es lógico existen muchas más funciones y variables que entran en juego en la recepción de estos paquetes. No obstante, según se vayan requiriendo se irán explicando.

3.1.4. Resultados

Una vez explicado todo el código referente al experimento, en esta sección se expondrán los resultados obtenidos de esta primera prueba.

Para ejecutar el proyecto únicamente se debe pulsar sobre el botón desde el SDK, esto se encargará de enviar el fichero binario y situarlo en la memoria DDR para que el procesador ARM pueda leer del mismo las instrucciones.

Una vez ejecutado el proyecto habrá que conectarse mediante un puerto serie a la placa. La placa está configurada por defecto para transmitir a 115200 baudios en la configuración típica 8N1.

En la consola aparecerá algo parecido al texto mostrado en la Figura 3.7

```
-----lwIP RAW Mode TCP Server Application-----
Start PHY autonegotiation.
Waiting for PHY to complete autonegotiation.
autonegotiation complete.
link speed for phy address 0: 1000
Configuring default IP 192.168.0.10.
Board IP:       192.168.0.10
Netmask :       255.255.255.0
Gateway :       192.168.0.1

TCP server listening on port 5001
On Host: Run $iperf -c 192.168.0.10 -i 5 -t 300 -w 2M
```

Figura 3.7: Salida obtenida por el primer ejemplo

Esta consola nos muestra claramente el comando que se deberá ejecutar en el cliente para poder realizar la prueba. Como es lógico, el cliente deberá tener instalado el software iperf, que puede ser descargado de manera gratuíta desde la web oficial.

El programa cuenta con una gran cantidad de parámetros algunos de los que serán explicados más adelante según se vayan requiriendo. En este caso, los parámetros involucrados son:

- -i: indica el intervalo de tiempo en segundos entre cada reporte del ancho de banda alcanzado. Por ejemplo, -i 5 hará que aparezcan por pantalla, cada 5 segundos, las estadísticas de la conexión.
- -w 2M: mediante esta opción se indica la longitud de la ventana deslizante de la capa TCP. Tal y como se verá más adelante no siempre se cumple esta configuración debido al sistema operativo, a la memoria (en el caso de sistemas basados en *baremetal*) o la negociación inicial.

Una vez ejecutado el programa el cliente iniciará la transferencia de datos desde memoria a la dirección especificada. En la placa, en este caso, se debería ver el reporte con los datos adquiridos.

El reporte se puede visualizar en la Figura 3.8. Tal y como se puede observar la velocidad máxima adquirida es de 941 Mbits/sec. Además, se observa que el ancho de banda se mantiene en el tiempo. Por lo tanto se puede decir que esta medida está saturando el canal.

```
-----lwIP RAW Mode TCP Server Application-----
Start PHY autonegotiation.
Waiting for PHY to complete autonegotiation.
autonegotiation complete.
link speed for phy address 0: 1000
Configuring default IP 192.168.0.10.
Board IP:       192.168.0.10
Netmask :       255.255.255.0
Gateway :       192.168.0.1

TCP server listening on port 5001
On Host: Run $iperf -c 192.168.0.10 -i 5 -t 300 -w 2M

[  1] local 192.168.0.10 port 5001 connected with 192.168.0.1 port 47598
[ ID] Interval>.>...Transfer   Bandwidth
[  1]  0.0- 5.0 sec   561 MBytes   941 Mbits/sec
[  1]  5.0-10.0 sec   561 MBytes   941 Mbits/sec
[  1] 10.0-15.0 sec   561 MBytes   941 Mbits/sec
[  1] 15.0-20.0 sec   561 MBytes   941 Mbits/sec
[  1] 20.0-25.0 sec   561 MBytes   941 Mbits/sec
[  1] 25.0-30.0 sec   560 MBytes   939 Mbits/sec
[  1]  0.0-34.1 sec  3.74 GBytes   941 Mbits/sec
TCP test passed Successfully
```

Figura 3.8: Reporte del servidor iperf

En las primeras líneas se puede ver como se auto negocia la velocidad de 1000. Esta velocidad está expresada en bytes por segundos. Esto quiere decir que se está utilizando todo el enlace de 1Gbps.

Antes de continuar analizando los datos hay que preguntarse sobre las unidades de medida utilizadas.

Se pueden diferenciar dos grandes estándares en cuanto a unidades de medida en los sistemas de comunicación:

1. Medición de datos relacionados con el almacenamiento: en este caso se

utiliza el estandar IEC donde los múltiplos se expresan en potencia de dos. De este modo, los kibibytes (KiB) equivalen a 2^{10}.

2. Medición de datos relacionados con la velocidad/latencia/ancho de banda: En este caso, se utiliza el estandar decimal. De este modo, los kilobytes (KB) hacen referencia a 10^3.

Iperf muestra las transferencias en MiB aunque no utilice la nomenclartura adecuada, por otro lado, el ancho de banda lo reporta en el estándar decimal.

En este punto y con estos datos se plantean las siguientes preguntas:

- ¿No es necesario realizar ninguna modificación en los parámetros de configuración para alcanzar estos datos?: con la motivación de dar respuesta a esta pregunta se decide buscar el fichero *system.mss*, encargado de modificar diferentes parámetros del BSP. Efectivamente, dentro de este fichero se encuentran varias modificaciones sobre los parámetros por defecto. En las Figuras 3.9a, 3.9b y 3.9c se pueden ver dichas modificaciones. En primer lugar, se puede observar en la Figura 3.9c como el tamaño de la ventana se ha incrementado al máximo posible, respecto a los 8KiB iniciales. Se recuerda al lector que el máximo de esta ventana es de 216 debido a los bits utilizados para codificar la opción. Este tamaño de ventana permite incrementar el ancho de banda y mejorar la tolerancia a fallos del sistema. Si la placa llenara su buffer de entrada se empezarían a perder paquetes. Sin embargo, gracias al ancho de la ventana, el cliente podría seguir mandando mensajes para no reducir la utilización del canal. Por otro lado, se observa como en la Figura 3.9b se ha incrementado el número de *pbuf_pool_size*. Esta variable define cuantas estructuras de tipo pbuf se ubicarán en tiempo de compilación para su uso por la capa TCP/IP. Este parámetro deberá multiplicado por el tamaño de cada buffer, especificado en la opción *pbuf_pool_size*, deberá ser al menos tan alto como el tamaño de ventana, dado que en un momento dado la ventana podría estar llena y se debería asegurar el espacio para todos los paquetes dentro de dicha ventana. Finalmente resulta interesante ver cómo se han modificado el número de descriptores del modo *scater/gather* del DMA incorporado en el MAC. En la

Figura 3.9a, se puede apreciar como, este número ha pasado de 64 a 512. Esto permite al DMA añadir más bloques a memoria DDR sin interrumpir al procesador, aumentando la performance del sistema.

- ¿Cómo se calcula el ancho de banda de la comunicación?: esta pregunta es clave para poder realizar implementaciones compatibles con este servidor estándar iperf y así realizar comparaciones justas. En el Listado 5 se puede ver la lógica detrás del cálculo del ancho de banda del enlace. Básicamente, se calcula con el tiempo transcurrido desde el inicio hasta el final y el tamaño **del paquete**, no del **payload**.

- ¿Cúal es la lógica del servidor iperf?: esta pregunta se puede considerar una extensión de la pregunta anterior. De cara a futuros experimentos, en este punto se planteó la tarea de entender el funcionamiento detrás del servidor iperf. Tal y como se puede ver en el Listado 5, el servidor únicamente recibe datos y los **confirma**. Realizando una captura con el programa de análisis *wireshark* se **se llega a la conclusión** de que el servidor no espera ningún paquete específico, únicamente un *stream* de datos sin ningún formato. La labor del servidor es mandar los Acknowledgement (ACK) correspondientes siguiendo con las políticas impuestas por la lógica de TCP.

Teniendo claro el funcionamiento del servidor iperf, se decide poner en práctica lo estudiado mediante la creación de un cliente, simple, en el lenguaje de programación Python, para iperf. Este cliente permitirá comprobar si las conclusiones a las que se han llegado son ciertas y además, permitirá tener un cliente independiente de iperf para realizar futuras pruebas.

En el Listado 4 se puede ver el código (con el objetivo de no extender el documento más de lo estrictamente necesario únicamente se muestra las partes relevantes del mismo). Como se puede observar este es muy sencillo y únicamente consiste en invocaciones TCP con un *payload* fijo formado por los carácteres ASCII “test”.

Los **resultados** obtenidos son exactamente los mismos a los obtenidos con el cliente oficial de iperf. Sin embargo, si en la Línea 26 se modifica la lógica del programa para realizar varias invocaciones a la función send y dividir el tamaño entre el número de invocaciones la velocidad se reduce

de forma significativa. Esto se debe principalmente al coste de invocar una función en un lenguaje de programación como es Python.

```
#!/usr/bin/python3

import socket
import logging
import os
import sys
import time

logging.basicConfig(level=logging.DEBUG)

IP_ZEDBOARD = os.environ.get("IP_ZEDBOARD", "192.168.0.10")
IP_PORT = int(os.environ.get("IP_PORT", 5001))


def connect_to_zedboard(ip, port):
    logging.info("Connecting to {}:{}".format(ip, port))
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client.connect((ip, port))
    except Exception as ex:
        logging.error("Impossible to connect to {}:{}.
        ↪  Reason:{}".format(ip, port, ex))
        return None

    logging.info("Connected to {}:{}".format(ip, port))
    return client


def disconnect_from_zedboard(socket):
    try:
        socket.close()
    except Exception as ex:
        logging.error("Impossible to disconnect from zedboard:
        ↪  {}".format(ex))

```

```
34
35 def make_test(socket):
36     NUMBER_OF_DATA_CHUNKS = 80000
37     NUMBER_OF_PACKETS = 20000
38     DUMMY_DATA = "test" * NUMBER_OF_DATA_CHUNKS
39     DUMMY_DATA_AS_BYTES = DUMMY_DATA.encode()
40
41     TOTAL_SIZE_IN_BYTES = len(DUMMY_DATA_AS_BYTES) * NUMBER_OF_PACKETS
42     logging.info("It will sent: {} bytes".format(TOTAL_SIZE_IN_BYTES))
43
44     time_of_start = time.perf_counter()
45     socket.send(DUMMY_DATA_AS_BYTES * NUMBER_OF_PACKETS)
46
47     time_of_end = time.perf_counter()
48
49     total_time = time_of_end - time_of_start
50     bandwidth = TOTAL_SIZE_IN_BYTES / total_time
51     logging.info("Test completed, time {} seconds, data sent {} bytes,
       ↪  bandwidth {} bytes/s".format(total_time, TOTAL_SIZE_IN_BYTES,
       ↪  bandwidth))
52
53
54 if __name__ == "__main__":
55     con = connect_to_zedboard(IP_ZEDBOARD, IP_PORT)
56
57     if not con:
58         sys.exit(-1)
59
60     make_test(con)
61
62     disconnect_from_zedboard(con)
```

Listado 4: Cliente de prueba iperf

temac_adapter_options	true	true	boolean	Settings for xps-ll-tema
emac_number	0	0	integer	Zynq Ethernet Interface
n_rx_coalesce	1	1	integer	Setting for RX Interrupt
n_rx_descriptors	512	64	integer	Number of RX Buffer De
n_tx_coalesce	1	1	integer	Setting for TX Interrupt
n_tx_descriptors	512	64	integer	Number of TX Buffer De
phy_link_speed	Autodetect (CONFIG_I	CONFIG_LINKSPEED_A	enum	link speed as negotiate
tcp_ip_rx_checksum_offl	false	false	boolean	Offload TCP and IP Rece
tcp_ip_tx_checksum_offl	false	false	boolean	Offload TCP and IP Trans
tcp_rx_checksum_offloac	false	false	boolean	Offload TCP Receive che
tcp_tx_checksum_offloac	false	false	boolean	Offload TCP Transmit ch
temac_use_jumbo_frame	false	false	boolean	use jumbo frames

(a) Modificaciones en los parámetros de la interfaz

pbuf_options	true	true	boolean	Pbuf Options
pbuf_link_hlen	16	16	integer	Number of bytes that sho
pbuf_pool_bufsize	1700	1700	integer	Size of each pbuf in pbuf
pbuf_pool_size	8192	256	integer	Number of buffers in pbuf

(b) Modificaciones en los parámetros de gestión de memoria

tcp_options	true	true	boolean	Is TCP required ?
lwip_tcp	true	true	boolean	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retransmis
tcp_mss	1460	1460	integer	TCP Maximum segment si
tcp_queue_ooseq	1	1	integer	Should TCP queue segmer
tcp_snd_buf	65535	8192	integer	TCP sender buffer space (I
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN retrans
tcp_ttl	255	255	integer	TCP TTL value
tcp_wnd	65535	2048	integer	TCP Window (bytes)

(c) Modificaciones en los parámetros de la capa TCP

```
1  static err_t tcp_recv_perf_traffic(void *arg, struct tcp_pcb *tpcb,
2                  struct pbuf *p, err_t err)
3  {
4          if (p == NULL) {
5                  u64_t now = get_time_ms();
6                  u64_t diff_ms = now - server.start_time;
7                  tcp_server_close(tpcb);
8                  tcp_conn_report(diff_ms, TCP_DONE_SERVER);
```

```
9                 xil_printf("TCP test passed Successfully\n\r");
10                return ERR_OK;
11        }
12
13        /* Record total bytes for final report */
14        server.total_bytes += p->tot_len;
15
16        if (server.i_report.report_interval_time) {
17                u64_t now = get_time_ms();
18                /* Record total bytes for interim report */
19                server.i_report.total_bytes += p->tot_len;
20                if (server.i_report.start_time) {
21                        u64_t diff_ms = now - server.i_report.start_time;
22
23                        if (diff_ms >=
                          ↪  server.i_report.report_interval_time) {
24                                tcp_conn_report(diff_ms, INTER_REPORT);
25                                /* Reset Interim report counters */
26                                server.i_report.start_time = 0;
27                                server.i_report.total_bytes = 0;
28                        }
29                } else {
30                        /* Save start time for interim report */
31                        server.i_report.start_time = now;
32                }
33        }
34
35        tcp_recved(tpcb, p->tot_len);
36
37        pbuf_free(p);
38        return ERR_OK;
39 }
```

Listado 5: Código encargado de calcular las diferentes estadísticas del servidor iperf

```
1 IP_ZEDBOARD = os.environ.get("IP_ZEDBOARD", "192.168.0.10")
```

```
2  IP_PORT = int(os.environ.get("IP_PORT", 5001))
3
4  def connect_to_zedboard(ip, port):
5      logging.info("Connecting to {}:{}".format(ip, port))
6      client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7      try:
8          client.connect((ip, port))
9      except Exception as ex:
10         logging.error("Impossible to connect to {}:{}.
           ↪  Reason:{}".format(ip, port, ex))
11         return None
12
13     logging.info("Connected to {}:{}".format(ip, port))
14     return client
15
16 def make_test(socket):
17     NUMBER_OF_DATA_CHUNKS = 80000
18     NUMBER_OF_PACKETS = 20000
19     DUMMY_DATA = "test" * NUMBER_OF_DATA_CHUNKS
20     DUMMY_DATA_AS_BYTES = DUMMY_DATA.encode()
21
22     TOTAL_SIZE_IN_BYTES = len(DUMMY_DATA_AS_BYTES) * NUMBER_OF_PACKETS
23     logging.info("It will sent: {} bytes".format(TOTAL_SIZE_IN_BYTES))
24
25     time_of_start = time.perf_counter()
26             socket.send(DUMMY_DATA_AS_BYTES * NUMBER_OF_PACKETS)
27
28     time_of_end = time.perf_counter()
29
30     total_time = time_of_end - time_of_start
31     bandwidth = TOTAL_SIZE_IN_BYTES / total_time
32     logging.info("Test completed, time {} seconds, data sent {} bytes,
       ↪  bandwidth {} bytes/s".format(total_time, TOTAL_SIZE_IN_BYTES,
       ↪  bandwidth))
33
34
35 if __name__ == "__main__":
36     con = connect_to_zedboard(IP_ZEDBOARD, IP_PORT)
```

```
37
38      if not con:
39          sys.exit(-1)
```

Listado 6: Cliente en python iperf

A modo de resumen, como resultados se obtiene un ancho de banda de 1Gbit/s aproximadamente, es decir, con las mínimas capas de abstracción posibles, se satura el canal de comunicaciones.

3.2. Experimento 2: LWIP RAW TCP Cliente

3.2.1. Descripción del experimento / Objetivos

En el experimento anterior se caracterizó la velocidad máxima obtenida con lo que, en este trabajo, se ha considerado las mínimas capas de software posibles. Sin embargo, la medida anterior únicamente hace referencia a uno de los flujos de comunicación posibles, es decir, desde el ordenador de gestión hacia la placa de medición. Este es solo uno de los escenarios posibles en una aplicación real. Normalmente la comunicación fluye en ambos sentidos y es por ello que, este experimento se centrará el estudio en el flujo contrario.

Así, el **objetivo** de este experimento es el de medir el ancho de banda desde la placa de medición que actuará, en este caso, de cliente y el servidor, en este caso el ordenador de gestión.

3.2.2. Hipótesis

En este experimento se espera obtener tasas de transferencia cercanas a las del experimento anterior. Esto se debe a que la sobrecarga que, en principio, está asociada al empaquetado es mínima, tal y como se ha podido observar en el experimento anterior.

3.2.3. Desarrollo del experimento

El desarrollo de este experimento parte en la mayor parte de todo lo explicado en el experimento anterior por lo que en este caso, el desarrollo del esta sección será más reducido.

El **hardware** utilizado es **exactamente el mismo** por lo que no es necesario ni generar un nuevo HDF.

El siguiente paso consistirá en crear, mediante el generador de proyectos de Xilinx el proyecto software. En este apartado sí que existen modificaciones las cuales serán desarrolladas a continuación.

Implementación del software — Cliente en Zedboard

La creación del proyecto sigue los mismos pasos que el experimento anterior. La única diferencia presente se encuentra a la hora de elegir el proyecto base del que partir. Xilinx proporciona un cliente iperf implementado en la placa por lo que se seleccionará dicho ejemplo.

Una vez terminado el proceso se obtiene una estructura exactamente igual a la del experimento anterior, es decir:

- BSP
- Aplicación

El fichero main.c sigue la misma estructura que el mostrado en el Listado 2. La diferencia principal se encuentra en la parte correspondiente a la "aplicación".

En el Listado 7 se puede ver las partes más importantes en cuanto al inicio de la aplicación. En este extracto de código se puede ver que, a diferencia del experimento anterior, en este caso, en vez de pasar el PCB a un estado de *LISTEN* se le pasa a un estado de *CONNECTED*.

Para pasar el PCB al estado connected se utiliza la función proporcionada por la API de LWIP *tcp_connect*. Esta función (ver Línea 21), toma como parámetros una dirección y un puerto. En este caso, la dirección y el puerto serán los del ordenador de gestión.

Tras la conexión, se puede observar la creación de datos de prueba sintéticos en la Línea 31. Estos datos siguen un patrón concreto bastante sencillo. El patrón será enviar los números desde 0 a 9 en formato ASCII. El hecho de enviar los datos en formato ASCII no tiene relevancia. Tras analizar el código original de iperf se llega a la conclusión de que únicamente se envía en ASCII

con el objetivo de poder analizar de forma más sencilla con herramientas como Wireshark.

Probablemente esta parte del código sea la parte **más importante** del experimento. Rapidamente se planteó la hipótesis de si este ejemplo representa un escenario real de aplicación para las aplicaciones de instrumentación científica.

Lo cierto es que normalmente los datos no se tienen de antemano en memoria y puede que se tenga que añadir algún mecanismo de exclusión en el acceso a dichas zonas de memoria.

Típicamente un módulo en el hardware escribirá en regiones de memoria DDR mediante algún tipo de DMA. Será por tanto tarea del procesador comprobar que los datos están listos para enviarse y en el momento de envío "recogerlos de memoria". En este caso se suponen que son fijos.

En el la Línea 36 se puede ver el código que se ejecutará una vez conectado el cliente. Este código, de forma parecida al experimento anterior, únicamente reiniciará las estadísticas y configurará los callback que serán ejecutados en cada uno de los eventos. Tal y como se puede observar, en estas líneas se configura el callback para el caso en que se realiza un envío con la función *tcp_send*.

El lector puede preguntarse en qué momento se realiza el envío pues lo configurado hasta el momento únicamente configura los callback una vez realizado el envío.

El envío de los datos se realiza en la función *main* dentro del bucle principal. En este punto se llama a la función *transfer_data* que realizará el envío de forma efectiva, tal y como puede observarse en la Línea 57. El código se ha simplificado para mantener el documento lo más reducido posible.

```
1  void start_application(void)
2  {
3          err_t err;
4          struct tcp_pcb* pcb;
5          ip_addr_t remote_addr;
```

```
6           u32_t i;
7
8           err = inet_aton(TCP_SERVER_IP_ADDRESS, &remote_addr);
9
10          if (!err) {
11                  xil_printf("Invalid Server IP address: %d\r\n", err);
12                  return;
13          }
14
15          pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
16          if (!pcb) {
17                  xil_printf("Error in PCB creation. out of memory\r\n");
18                  return;
19          }
20
21          err = tcp_connect(pcb, &remote_addr, TCP_CONN_PORT,
22                            tcp_client_connected);
23          if (err) {
24                  xil_printf("Error on tcp_connect: %d\r\n", err);
25                  tcp_client_close(pcb);
26                  return;
27          }
28          client.client_id = 0;
29
30          /* initialize data buffer being sent with same as used in iperf
            ↪  */
31          for (i = 0; i < TCP_SEND_BUFSIZE; i++)
32                  send_buf[i] = (i % 10) + '0';
33
34          return;
35  }
36  static err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t
    ↪  err)
37  {
38          c_pcb = tpcb;
39
40          client.start_time = get_time_ms();
41          client.end_time = TCP_TIME_INTERVAL * 1000; /* ms */
```

```
42          client.client_id++;
43          client.total_bytes = 0;
44
45          client.i_report.report_interval_time = INTERIM_REPORT_INTERVAL *
            ↪  1000;
46          client.i_report.last_report_time = 0;
47          client.i_report.start_time = 0;
48          client.i_report.total_bytes = 0;
49
50          tcp_arg(c_pcb, NULL);
51          tcp_sent(c_pcb, tcp_client_sent);
52          tcp_err(c_pcb, tcp_client_err);
53
54          return ERR_OK;
55  }
56
57  static err_t transfer\_data(void)
58  {
59          err_t err;
60          u8_t apiflags = TCP_WRITE_FLAG_COPY | TCP_WRITE_FLAG_MORE;
61
62          while (tcp_sndbuf(c_pcb) > TCP_SEND_BUFSIZE) {
63                  err = tcp_write(c_pcb, send_buf, TCP_SEND_BUFSIZE,
                    ↪  apiflags);
64                  if (err != ERR_OK) {
65                          xil_printf("TCP client: Error on tcp_write:
                            ↪  %d\r\n",
66                                          err);
67                          return err;
68                  }
69                  err = tcp_output(c_pcb);
70                  if (err != ERR_OK) {
71                          xil_printf("TCP client: Error on tcp_output:
                            ↪  %d\r\n",
72                                          err);
73                          return err;
74                  }
75                  client.total_bytes += TCP_SEND_BUFSIZE;
```

```
76                  client.i_report.total_bytes += TCP_SEND_BUFSIZE;
77          }
78          return ERR_OK;
79  }
```

Listado 7: Código de aplicación, zedboard como cliente

En la función de transferencia *transfer_data* se pueden apreciar varios aspectos interesantes para el desarrollo del experimento. En primer lugar, en la Línea 60 se puede ver el uso de dos flags. El primero de los flags indica a la capa TCP/IP que realice una copia en memoria de los datos que se enviarán. El segundo flag carece de importancia para este experimento. La copia de los datos es lo contrario al mecanismo **Zero-Copy**, en este caso se ha utilizado este flag para partir de una base sin optimización alguna y más tarde poder realizar los experimentos posteriores con diferentes características del sistema de comunicación, como el Zero-Copy.

En la Línea 69 se puede ver cómo se realiza el envío efectivo. Este punto resulta interesante debido a que, 6 líneas más arriba, se realiza una invocación a la función *tcp_write*. El lector puede por tanto preguntarse cuál es la diferencia entre ambas funciones.

La primera función se encarga de reservar un espacio de memoria para alojar un paquete dentro del espacio de memoria "virtual" de la librería de comunicaciones. Una vez reservado el espacio, esta función se encarga de copiar los datos (en caso de que el flag explicado anteriormente esté activo) o apuntar al espacio de memoria donde se encuentra el *payload*. Finalmente, esta función se encarga de encolar el paquete asignándolo a un espacio de memoria perteneciente al *dma* del *mac*.

3.2.4. Resultados

Una vez programada la placa en la pantalla serial aparecerá algo parecido a lo mostrado en la Figura 3.10. En este caso, el comando que habrá que ejecutar en el ordenador de gestión será otro debido a que ahora actúa como servidor y no como cliente.

El comando a ejecutar es el siguiente: *iperf -s -i 5 -w 2M*. El comando es muy parecido al del ejemplo anterior, con la diferencia de que en este caso se activa el flag "-s". Este flag indica al programa iperf de que debe ejecutarse como servidor y no como cliente.

Una vez ejecutado el servidor convendrá reiniciar la placa con el objetivo de detecta lo más rápido posible la conexión. No obstante, la placa debería empezar a mandar datos tal y como se muestra en la Figura 3.10.

Los datos parecen **confirmar** la hipótesis inicial que, sostenía que el ancho de banda sería simétrico con respecto al experimento anterior.

Con este experimento se confirma por tanto que se pueden alcanzar, en ambas direcciones, la saturación del medio físico.

```
Configuring default IP 192.168.0.10
Board IP:       192.168.0.10
Netmask :       255.255.255.0
Gateway :       192.168.0.1

TCP client connecting to 192.168.0.1 on port 5001
On Host: Run $iperf -s -i 5 -w 2M

[  1] local 192.168.0.10 port 49153 connected with 192.168.0.1 port 5001
[ ID] Interval          Transfer   Bandwidth
[  1]  0.0- 5.0 sec   556 MBytes   933 Mbits/sec
[  1]  5.0-10.0 sec   557 MBytes   934 Mbits/sec
[  1] 10.0-15.0 sec   556 MBytes   934 Mbits/sec
```

Figura 3.10: Reporte del experimento, instrumento como cliente

3.3. Experimento 3: LWIP RAW UDP Servidor

3.3.1. Descripción del experimento / Objetivos

En este experimento se pasará de utilizar el protocolo de comunicaciones TCP al protocolo UDP. En este experimento se tratará de reducir aun más la complejidad trasladada al software a la hora de realizar una comunicación.

Otro de los objetivos principales de este experimento es el de comprobar la tasa de pérdida de paquetes en una comunicación punto a punto (comunicación típica en el tipo de aplicaciones que se aborda en este proyecto) y verificar, de este modo, la viabilidad de implementar el middleware de comunicaciones sobre UDP o TCP.

3.3.2. Hipótesis

Con la experiencia de los experimentos anteriores, en este caso se espera obtener los **mismos resultados** que en los ensayos anteriores en cuanto a ancho de banda. El razonamiento detrás de esta hipótesis es bastante simple. En los experimentos anteriores se alcanzó el ancho de banda máximo del medio, es decir, 1Gps. En este caso, el protocolo de comunicaciones UDP, tal y como se explicará más adelante, reduce significativamente la complejidad del software y por tanto la sobrecarga del mismo. Por esta razón, se espera tener como mínimo la misma eficiencia en la comunicación. Se deja como **trabajo futuro** realizar este mismo experimento con MACs de mayor capacidad.

En cuanto a las pérdidas de paquetes no se espera tener ninguna pérdida. Esto se debe a que el cable Ethernet está bien apantallado y en un enlace punto a punto sin intermediarios resulta extremadamente improbable perder paquetes. En [10] se hace referencia a una perdida de 1 paquete por cada 200 millones.

Finalmente, a partir de este experimento únicamente se harán pruebas en una dirección debido a que según los datos obtenidos en el experimento

anterior, se puede comprobar que la sobrecarga parece ser simétrica y por lo tanto los valores obtenidos serán similares. No obstante, se deja como **trabajo futuro** la ampliación de los experimentos de este documento, con el objetivo de contemplar dichos escenarios.

3.3.3. Desarrollo del experimento

En este apartado, al igual que en los experimentos anteriores, se dará una breve explicación de los aspectos más relevantes del código implementado. En este experimento se parte de la base de que el lector ha leído los experimentos anteriores dado que gran parte de la estructura del código será similar y por tanto no se explicará de nuevo.

Al igual que en los experimentos anteriores, el primer paso será generar el proyecto de prueba o plantilla proporcionada por Xilinx. En los experimentos anteriores se seleccionó la variante TCP, en este experimento habrá que seleccionar la variante UDP.

En este experimento se mantiene el software al mínimo e incluso se va un paso más allá mediante la eliminación de la capa TCP y la sustitución por la capa UDP. Al igual que en experimentos anteriores la librería será LWIP y se utilizará la API RAW.

Una vez generado el código se obtendrá una estructura muy similar a la mostrada en experimentos anteriores, es decir, un BSP y un proyecto con una aplicación incorporada.

Si se accede al fichero de configuración *system.mss* se podrá observar como la configuración desactiva la capa TCP de la librería LWIP, tal y como se puede ver en la Figura 3.11. Este simple cambio supone un ahorro en memoria y por tanto en código que no será ejecutado en la cadena de comunicación.

Finalmente, la Tabla 3.1 se pueden ver los resultados de compilar el servidor TCP y UDP. Estos valores se han obtenido, en ambos casos, sin ninguna optimización en compilación. En este punto resulta interesante el hecho de la reducción de un aproximadamente un 13/del código únicamente desactivando la capa TCP. También resulta interesante el hecho de que tanto el

tcp_options	true	true	boolean	Is TCP required ?
lwip_tcp	false	true	boolean	Is TCP required ?
tcp_maxrtx	12	12	integer	TCP Maximum retransmis
tcp_mss	1460	1460	integer	TCP Maximum segment si
tcp_queue_ooseq	0	1	integer	Should TCP queue segmer
tcp_snd_buf	0	8192	integer	TCP sender buffer space (I
tcp_synmaxrtx	4	4	integer	TCP Maximum SYN retran
tcp_ttl	255	255	integer	TCP TTL value
tcp_wnd	0	2048	integer	TCP Window (bytes)

Figura 3.11: Opciones de configuración del BSP UDP

Cuadro 3.1: Comparación de los ficheros objeto entre TCP y UDP

	Text	Data	BSS
TCP	149516	4224	16898336
UDP	130792	4224	16898336
UDP/TCP	87.74 %	100 %	100 %

sector data y el sector bss se mantienen fijos. Esto se debe, principalmente a la estrategia seguida por LWIP a la hora de gestionar la memoria.

LWIP, tal y como se comentó anteriormente, mantiene un pool de segmentos de memoria, que utilizará más adelante para ubicar espacio para cada paquete según las necesidades de cada momento. Esta es la razón por la que en ambos casos se mantiene el uso de memoria, dado que esto no depende del protocolo subyacente si no del número de bytes reservados para la "memoria virtual" de LWIP.

Una vez analizadas las diferencias principales en tiempo de compilación de cada una de las soluciones (TCP y UDP), a continuación se explicarán los detalles de implementación más relevantes para este experimento.

Implementación del software — Servidor UDP en Zedboard

Al igual que en los experimentos anteriores, el código de inicio es exactamente el mismo. Esto se debe a que el código utilizado en el main.c únicamente configura la interfaz de red y el direccionamiento de red. Esto quiere

decir que cubre hasta el nivel 3 dentro del modelo estandarizado OSI. Existe, sin embargo, una pequeña diferencia entre el código main.c de los experimentos anteriores y el código main.c de este experimento. En este caso, en el bucle infinito principal no se actualiza ningún timer debido a que UDP no mantiene ninguna lógica de reenvío y por tanto no requiere de ningún timer para gestionar timeouts entre paquetes.

En la función *start_application* se puede ver el código implementado. En el Listado 8. En este código se puede ver como se siguen utilizando las estructuras de datos PCB. Esto se debe a que las mismas son agnósticas del protocolo de transporte subyacente. Por otro lado, de forma parecida al experimento del servidor TCP, se realiza un *binding* a una IP y puerto concreto.

El *binding* únicamente asocia a nivel interno cómo gestionar una petición desde el exterior.

Finalmente, en la Línea 21 se configura el callback para cuando se reciba una comunicación. Como se puede observar esta parte varía de forma considerable con respecto al experimento del servidor TCP.

En primer lugar, en el experimento TCP se realizaba un flujo a la hora de gestionar una comunicación. Lo primero consistía en realizar el *binding* para, más tarde, poner la librería a "escuchar" y "aceptar clientes". Esta diferencia se debe a la filosofía de cada protocolo de comunicaciones. Tal y como se vio en la Sección 2.4, el protocolo TCP está orientado a conexión frente al protocolo UDP que no mantiene ningún tipo de conexión.

En el Listado 9 se puede ver el código que se ejecutará cada vez que se reciba una invocación en el servidor. El código se ha simplificado con el objetivo de mantener lo más simple posible el documento.

Lo primero que se comprueba es el número de secuencia enviado por el cliente. Este punto puede llevar a engaño. UDP, por defecto, no proporciona ningún número de secuencia en su cabecera (TCP si que gestiona los números de secuencia). Este número de secuencia al que se hace referencia es un número añadido por la capa de aplicación del modelo de referencia OSI, es decir, el número de secuencia lo proporciona el servidor iperf.

Siguiendo el flujo del código, lo siguiente que se realiza es un cálculo de

las estadísticas internas. Las estadísticas mantienen el número de paquetes perdido (nuevo con respecto al experimento TCP) y el ancho de banda del sistema.

El código aunque pueda parecer más sencillo tiene que tener en cuenta más situaciones que en el caso de TCP no hacían falta. Por ejemplo, el cálculo de paquetes perdidos no se realizaba en TCP dado que no tiene sentido, pues es la propia capa TCP la encargada de asegurar que **nunca** se pierde un paquete y que, en el caso de que se "pierda" será la propia capa la encargada de gestionar con el cliente la recuperación de dicho paquete.

```
1  void start_application(void)
2  {
3          err_t err;
4
5          /* Create Server PCB */
6          pcb = udp_new();
7          if (!pcb) {
8                  xil_printf("UDP server: Error creating PCB. Out of
                   ↪  Memory\r\n");
9                  return;
10         }
11
12         err = udp_bind(pcb, IP_ADDR_ANY, UDP_CONN_PORT);
13         if (err != ERR_OK) {
14                 xil_printf("UDP server: Unable to bind to port");
15                 xil_printf(" %d: err = %d\r\n", UDP_CONN_PORT, err);
16                 udp_remove(pcb);
17                 return;
18         }
19
20         /* specify callback to use for incoming connections */
21         udp_recv(pcb, udp_recv_perf_traffic, NULL);
22
23         return;
24 }
```

Listado 8: Código para el inicio de la "aplicación" del experimento UDP

3.3.4. Resultados

```
static void udp_recv_perf_traffic(void *arg, struct udp_pcb *tpcb,
                struct pbuf *p, const ip_addr_t *addr, u16_t port)
{
        static u8_t first = 1;
        u32_t drop_datagrams = 0;
        s32_t recv_id;

        recv_id = ntohl(*((int *)(p->payload)));
        if (first && (recv_id == 0)) {
                pcb->remote_ip = *addr;
                pcb->remote_port = port;
                reset_stats();
                first = 0;
        } else if (first) {
                return;
        }

        if (recv_id < 0) {
                u64_t now = get_time_ms();
                u64_t diff_ms = now - server.start_time;
                udp_sendto(tpcb, p, addr, port);
                udp_conn_report(diff_ms, UDP_DONE_SERVER);
                xil_printf("UDP test passed Successfully\n\r");
                first = 1;
                pbuf_free(p);
                return;
        }

        if (server.expected_datagram_id != recv_id) {
                if (server.expected_datagram_id < recv_id) {
                        drop_datagrams =
                                recv_id - server.expected_datagram_id;
```

```
33                          server.cnt_dropped_datagrams += drop_datagrams;
34                          server.expected_datagram_id = recv_id + 1;
35                  } else if (server.expected_datagram_id > recv_id) {
36                          server.cnt_out_of_order_datagrams++;
37                  }
38          } else {
39                  server.expected_datagram_id++;
40          }
41
42          server.cnt_datagrams++;
43
44          server.total_bytes += p->tot_len;
45
46          pbuf_free(p);
47          return;
48  }
```

Listado 9: Código para la gestión del servidor UDP iperf

Con el código explicado el siguiente paso consistirá, al igual que en los experimentos anteriores en realizar las mediciones y contestar a las hipótesis planteadas.

3.3.5. Resultados

En esta sección se tratará de dar respuesta a la hipótesis planteadas. Se recuerda al lector cuáles eran estas hipótesis, de forma resumida:

- Se espera el mismo ancho de banda que para TCP o algo mayor: al estar el canal saturado con TCP, se espera que con UDP ocurra lo mismo.
- Se espera que el número de paquetes perdidos sea mínimo: en un enlace punto a punto, resulta muy improbable que pierdan paquetes.

En la Figura 3.13. Como se puede observar la velocidad es prácticamente

la misma en todos los intervalos de medición. Por otro lado, en la Figura 3.12 se puede comprobar que la pérdida de paquetes es nula.

```
-----lwIP RAW Mode UDP Server Application-----
Start PHY autonegotiation
Waiting for PHY to complete autonegotiation.
autonegotiation complete
link speed for phy address 0: 1000
ERROR: DHCP request timed out
Configuring default IP 192.168.0.10
Board IP:       192.168.0.10
Netmask :       255.255.255.0
Gateway :       192.168.0.1

UDP server listening on port 5001
On Host: Run $iperf -c 192.168.0.10 -i 5 -t 300 -u -b <bandwidth>

[  1] local 192.168.0.10 port 5001 connected with 192.168.0.1 port 56275
[ ID] Interval       Transfer     Bandwidth         Lost/Total Datagrams
[  1]  0.0- 5.0 sec   565 MBytes   948 Mbits/sec     0/403091 (0%)
[  1]  5.0-10.0 sec   564 MBytes   946 Mbits/sec     0/402295 (0%)
[  1] 10.0-15.0 sec   565 MBytes   947 Mbits/sec     0/402727 (0%)
[  1] 15.0-20.0 sec   564 MBytes   946 Mbits/sec     0/402349 (0%)
[  1] 20.0-25.0 sec   563 MBytes   945 Mbits/sec     0/401908 (0%)
[  1]  0.0-29.4 sec  3.24 GBytes   946 Mbits/sec     0/2367729 (0%)
UDP test passed Successfully
```

Figura 3.12: Resultados de la medición de la perdida de paquetes UDP

En la Figura 3.14 se puede ver el estado de la interfaz de red del ordenador de gestión que se encuentra trabajando al máximo posible.

Con estos datos queda probado que tanto en TCP como en UDP actualmente se satura el canal y por tanto el software actual no supone una sobrecarga apreciable o al menos subsanable para la eficiencia de la comunicación.

Con estos resultados se puede plantear **la implantación del middleware de comunicación propuesto**. El único problema al que nos enfrentamos en este punto es que no existe ningún tipo de sistema operativo o facilidad sobre el que portar el middleware de comunicaciones.

Aunque no es un requisito, tal y como se verá en los experimentos siguientes, resulta de mucha utilidad tener algún sistema operativo que permita, en la medida de lo posible utilizar la interfaz estándar de **sockets** sobre la que portar el middleware.

```
------------------------------------------------------------
Client connecting to 192.168.0.10, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11.76 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.0.1 port 35462 connected with 192.168.0.10 port 5
001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 5.0 sec   565 MBytes   947 Mbits/sec
[  3]  5.0-10.0 sec   564 MBytes   947 Mbits/sec
[  3] 10.0-15.0 sec   565 MBytes   947 Mbits/sec
[  3] 15.0-20.0 sec   564 MBytes   947 Mbits/sec
[  3] 20.0-25.0 sec   565 MBytes   947 Mbits/sec
[  3] 25.0-30.0 sec   565 MBytes   947 Mbits/sec
[  3] 30.0-35.0 sec   565 MBytes   947 Mbits/sec
[  3] 35.0-40.0 sec   565 MBytes   947 Mbits/sec
[  3] 40.0-45.0 sec   564 MBytes   947 Mbits/sec
[  3] 45.0-50.0 sec   565 MBytes   947 Mbits/sec
[  3] ............   ..........   ............
[  3] Sent 24153316 datagrams
```

Figura 3.13: Resultados de la medición para un servidor UDP

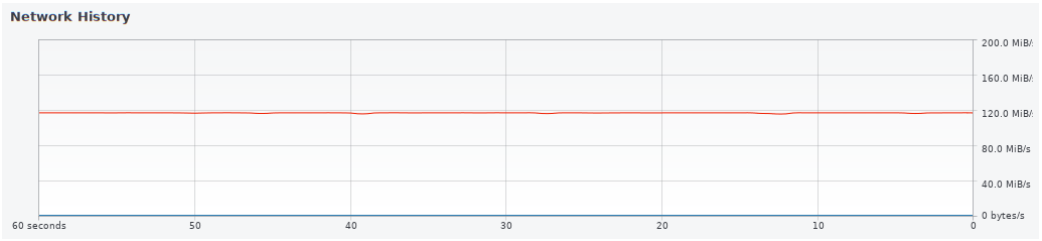

Figura 3.14: Administrador de recursos del ordenador de gestión

Esta interfaz no ha sido probada de forma aislada por lo que en el siguiente experimento se tratará de analizar el impacto (si es que lo hubiera) que supone el uso de esta interfaz.

3.4. Experimento 4: LWIP Sockets UDP Servidor

3.4.1. Descripción del experimento / Objetivos

En los experimentos anteriores, para realizar las mediciones, tanto en TCP como en UDP se ha utilizado la API de LWIP RAW. El uso de esta interfaz no es casual. En este trabajo se busca caracterizar el sistema base sobre el que poder proponer un esquema de abstracción de las comunicaciones con la menor sobrecarga posible. Para saber la sobrecarga en primer lugar se necesita saber el coste exacto del software base sobre el que se añadirán nuevas capas.

Una vez caracterizadas las medidas anteriores para la interfaz RAW, en este experimento se busca ir un paso más allá y dotar al sistema de la interfaz estándar basada en sockets.

Los sockets son una capa de software o interfaz que simplifica el uso de las librerías de red proporcionando una serie de funciones estándares de modo que el programador siempre se encuentre con la misma especificación a la hora de implementar una aplicación que requiera una comunicación de red. Otra de las ventajas de los sockets frente a la interfaz RAW utilizada hasta ahora es que, estos no requieren la instalación de callbacks que trabajarán de forma asíncrona. En su lugar se dispone de una serie de funciones bloqueantes (o no bloqueantes) que se ocuparán de obtener la información de red requerida por el programador.

Existe infinidad de literatura sobre la programación basada en sockets, [17] es solo un ejemplo. En este experimento se tratará de "portar" el experimento anterior basado en UDP con el uso de los sockets.

3.4.2. Hipótesis

En la documentación de LWIP se hace referencia a que existe una penalización en la eficiencia en las comunicaciones por el uso de sockets en lugar

de la interfaz de más bajo nivel o interfaz RAW. Por lo tanto se espera que la eficiencia en las comunicaciones aunque se mantenga la efectividad de la misma, es decir, la pérdida de paquetes se debería mantener al igual que ha ocurrido hasta ahora.

Además, se espera una reducción en la complejidad del software aunque esto venga acompañado de un incremento en el tamaño del programa debido a las capas de abstracción extra.

3.4.3. Desarrollo del experimento

El desarrollo de este experimento parte de la misma base hardware que los otros experimentos. La diferencia principal radica en la incorporación de un pequeño sistema operativo de tiempo real llamado FreeRTOS. La utilización del sistema operativo es un requisito por parte de LWIP para la utilización de una interfaz basada en sockets. Este requisito se debe a que la interfaz debe contemplar el modo de operación bloqueante. En un sistema sin ningún tipo de sistema operativo el modo bloqueante haría que el procesador no pudiera atender ningún tipo de interrupción software y por lo tanto la interfaz basada en callbacks utilizada en los experimentos anteriores no podría utilizarse.

En este experimento se explicarán las principales características de la incorporación de FreeRTOS a los experimentos. Al igual que en los experimentos anteriores, únicamente se detallarán aquellos aspectos del código que se consideran relevantes para el objetivo del experimento.

Esta sección se dividirá por tanto, en dos subsecciones diferenciadas:

1. Implementación del sistema operativo FreeRTOS: En esta sección se detallarán los aspectos más importantes en cuanto a la implementación o incorporación del FreeRTOS a un experimento.

2. Implementación del experimento UDP sobre la interfaz socket: por otro lado, una vez explicadas las características y funcionamiento de FreeRTOS, en este apartado se explicará como portar el experimento a esta nueva interfaz y se proporcionarán los datos necesarios para caracterizar la solución.

Xilinx, como en los experimentos anteriores, ha hecho un gran esfuerzo en simplificar el trabajo del diseñador a la hora de empezar a trabajar en su ecosistema.

Normalmente, a la hora de incorporar FreeRTOS a un proyecto, el diseñador debe acudir a la página web de FreeRTOs que se puede visitar desde: `https://www.freertos.org/a00090.html#XILINX` y buscar el paquete de la última versión de FreeRTOS. Una vez descomprimido, el diseñador se encontrará con una jerarquía de carpetas, donde, entre otras muchas cosas, encontrará los ficheros referentes al *port* para la plataforma concreta.

El proceso, para el caso de los dispositivos Xilinx, aunque puede ser el mismo explicado anteriormente, se simplifica mediante el generador de proyectos de Xilinx que, automáticamente, incorpora FreeRTOS como una librería más dentro del BSP del proyecto. En la Figura 3.15 se puede ver un ejemplo de cómo se mostraría el BSP con el proyecto FreeRTOS incorporado.

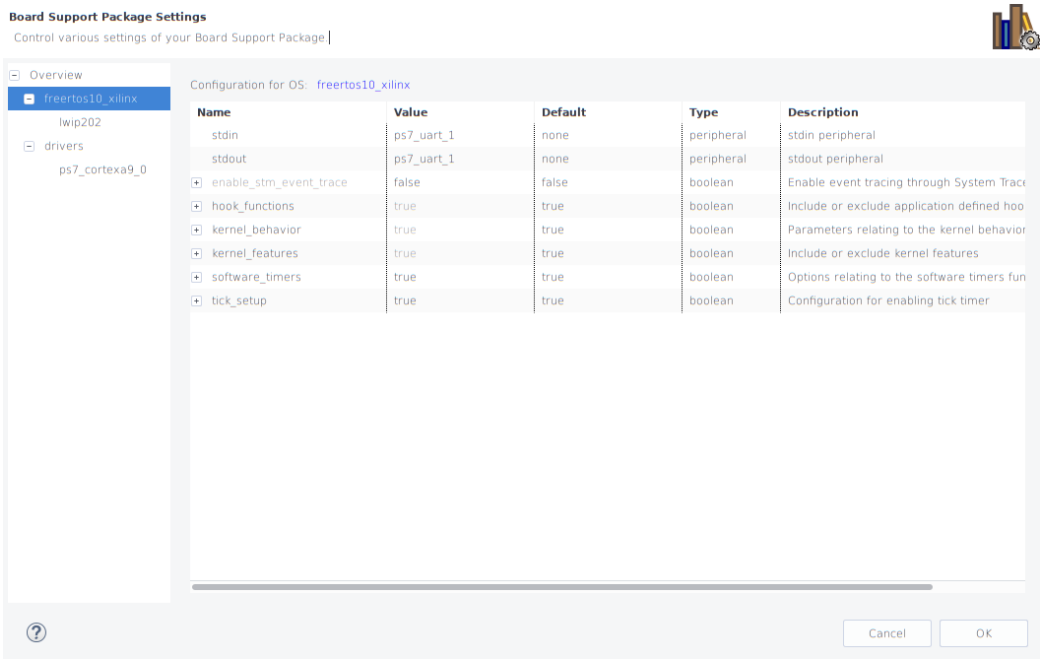

Figura 3.15: Ejemplo de BSP con FreeRTOs incorporado.

Una vez creado el proyecto de referencia, en la siguiente sección se detallarán los elementos más importantes para el desarrollo del experimento.

Análisis de un proyecto FreeRTOS

FreeRTOS supone un cambio sustancial con respecto a los experimentos anteriores. Este cambio supone una nueva capa de abstracción, un sistema operativo de tiempo real que simplificará muchas tareas pero que, puede que incremente la latencia o la sobrecarga del sistema.

Con el objetivo de caracterizar la sobrecarga de este sistema, en primer lugar, resulta interesante estudiar la estructura del proyecto.

En el Listado 10 se puede ver el extracto de código correspondiente el main.c simplificado del proyecto.

Como se puede apreciar este *main* es completamente diferente a los utilizados en experimentos anteriores.

En primer lugar, se crea una nueva tarea mediante la función *sys_thread_new()*. Esta función es parte de la API de LWIP para el uso de sockets. El utilizar esta función permite adaptar lwip a diversas plataformas y no únicamente a FreeRTOS. Esta primera capa de abstracción se encuentra dentro del fichero *sys_arch.c*. Dentro de este fichero se pueden encontrar muchas otras funciones implementadas por Xilinx para portar LWIP a su plataforma. Tal y como se puede observar en la Línea 3 la función recibe un puntero a un callback así como una variable llamada *THREAD_STACKSIZE* y *DEFAULT_THREAD_PRIO*. Estas variables determinan el tamaño de memoria al que tendrá acceso dicha tarea FreeRTOS y la prioridad con la que se ejecutará.

El tamaño del stack resulta de gran importancia a la hora de asegurar que FreeRTOS tiene el espacio suficiente para almacenar todos los paquetes y estructuras de datos necesarias. Por defecto, Xilinx configura esta variable al tamaño de 1024 bytes.

Una vez creada la tarea, siguiendo el flujo del programa, se llega a la

función *vTaskStartScheduler()* que iniciará el planificador FreeRTOS. Esta llamada es "bloqueante" y, si todo va bien (no existe ningún problema de memoria o excepción), el procesador nunca retornará de dicha función.

El scheduler será el encargado de gestionar la ejecución de cada una de las tareas en base a las prioridades, tal y como se explicó en la Sección 2.5.

La tarea principal (*main_thread*) se encarga de iniciar la pila LWIP y mediante la función *network_thread* ejecutar la tarea encargada de la gestión de los paquetes de red. En el Listado 11 se puede ver el código necesario para inicializar la interfaz. El lector puede comprobar cómo el código es prácticamente el mismo que en experimentos anteriores. La única diferencia es la Línea 15. Esta línea invoca a la función mencionada anteriormente para gestionar internamente la recepción de paquetes. De este modo, en vez de gestionar la recepción en un bucle infinito como en los ejemplos anteriores, en este caso se gestionará en una tarea individual.

```
1  int main()
2  {
3          sys_thread_new("main_thread", (void(*)(void*))main_thread, 0,
4                         THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);
5          vTaskStartScheduler();
6          while(1);
7          return 0;
8  }
9
10 int main_thread()
11 {
12
13         lwip_init();
14
15         /* any thread using lwIP should be created using sys_thread_new
           ↪  */
16         sys_thread_new("nw_thread", network_thread, NULL,
17                        THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);
18
19         while(!complete_nw_thread)
20                 usleep(50);
```

```
21
22          assign_default_ip(&(server_netif.ip_addr),
            ↪  &(server_netif.netmask),
23                                  &(server_netif.gw));
24
25          /* start the application*/
26          start_application();
27
28          vTaskDelete(NULL);
29          return 0;
30  }
```

Listado 10: main.c de un proyecto FreeRTOS

```
1   void network_thread(void *p)
2   {
3           u8_t mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02
            ↪  };
4
5           if (!xemac_add(&server_netif, NULL, NULL, NULL,
            ↪  mac_ethernet_address,
6                   PLATFORM_EMAC_BASEADDR)) {
7                   xil_printf("Error adding N/W interface\r\n");
8                   return;
9           }
10
11          netif_set_default(&server_netif);
12
13          netif_set_up(&server_netif);
14
15          sys_thread_new("xemacif_input_thread",
16                         (void(*)(void*))xemacif_input_thread,
                           ↪  &server_netif,
17                         THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);
18
19          complete_nw_thread = 1;
20
```

```
21          vTaskDelete(NULL);
22  }
```

Listado 11: Tarea encargada de gestionar la inicialización de la interfaz

Siguiendo con el flujo principal del programa, se llega a la función encargada de inicializar la "aplicación". Esta función se puede revisar en el Listado 12.

```
1   void start_application(void)
2   {
3           err_t err;
4           int sock;
5           struct sockaddr_in addr;
6
7           if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
8                   xil_printf("UDP server: Error creating Socket\r\n");
9                   return;
10          }
11
12          memset(&addr, 0, sizeof(struct sockaddr_in));
13          addr.sin_family = AF_INET;
14          addr.sin_port = htons(UDP_CONN_PORT);
15          addr.sin_addr.s_addr = htonl(INADDR_ANY);
16
17          err = bind(sock, (struct sockaddr *)&addr, sizeof(addr));
18          if (err != ERR_OK) {
19                  xil_printf("UDP server: Error on bind: %d\r\n", err);
20                  close(sock);
21                  return;
22          }
23
24          udp_recv_perf_traffic(sock);
25  }
```

Listado 12: Inicialización de la aplicación iperf

La función realiza el proceso de *binding* y, seguidamente, inicia un bucle infinito, donde procesa los paquetes del mismo modo al mostrado en experimentos anteriores. La principal diferencia en la recepción es el uso de la API basada en sockets frente a la API RAW.

En la siguiente sección se analizarán los resultados y se tratará de responder a las hipótesis planteadas inicialmente.

3.4.4. Resultados

En la Figura 3.16 se pueden consultar los datos obtenidos.

```
------lwIP Socket Mode UDP Server Application------
Start PHY autonegotiation
autonegotiation complete te autonegotiation.
link speed for phy address 0: 1000
ERROR: DHCP request timed out
Configuring default IP 192.168.0.10
Board IP:       192.168.0.10
Netmask :       255.255.255.0
Gateway :       192.168.0.1

UDP server listening on port 5001
On Host: Run $iperf -c 192.168.0.10 -i 5 -t 300 -u -b <bandwidth>

[  1] local 192.168.0.10 port 5001 connected with 192.168.0.1 port 38043
[ ID] Interval       Transfer     Bandwidth        Lost/Total Datagrams
[  1]  0.0- 5.0 sec   555 MBytes   932 Mbits/sec  5140/401273 (1.3%)
[  1]  5.0-10.0 sec   557 MBytes   935 Mbits/sec  5130/402761 (1.3%)
[  1] 10.0-15.0 sec   555 MBytes   932 Mbits/sec  5121/401200 (1.3%)
[  1] 15.0-20.0 sec   556 MBytes   933 Mbits/sec  5139/401905 (1.3%)
[  1] 20.0-25.0 sec   556 MBytes   933 Mbits/sec  5137/401780 (1.3%)
[  1]  0.0-26.0 sec  2.83 GBytes   933 Mbits/sec  25667/2089940 (1.2%)
UDP test passed Successfully
```

Figura 3.16: Resultados obtenidos en el servidor Iperf sobre FreeRTOS

Tal y como se esperaba en las hipótesis planteadas al inicio del experimento, el ancho de banda del enlace ha bajado ligeramente para situarse en los 932 Mbits frente a los 947 Mbits. Resulta bastante interesante observar la tasa de errores que parece mantenerse fija en un 1.4 %. La **conclusión** a la que se ha llegado a partir de estos datos es que el añadir un sistema operativo de tiempo real y varias tareas ejecutándose en paralelo, puede llevar a que el

Cuadro 3.2: Comparación de los ficheros objeto entre TCP y UDP

	Text	Data	BSS
UDP-SOCK	191284	4220	18995552
UDP-RAW	130792	4224	16898336
UDP-SOCK/UDP-RAW	146 %	100 %	95 %

tiempo para atender una trama sea demasiado alto y se pierda algún paquete. Suponiendo una MTU de 1500 bytes el tiempo máximo de procesamiento de una trama debería ser: $\frac{\frac{932*10^6}{8}}{1500}^{-1}$ o lo que es lo mismo $12.8\mu s$.

Este tiempo de procesamiento puede ser demasiado estricto para un entorno con varias tareas en paralelo, donde ocurren cambios de contexto. Esta es la razón por la que aparecen estas pérdidas.

Por último, con respecto al tamaño del ejecutable, tal y como se puede observar en la Tabla 3.2 el incremento en el uso de la memoria utilizada por el programa (Text en la tabla) se ha incrementado en un 46 %. Esto puede ser un impedimento, no obstante FreeRTOS permite reducir el tamaño del sistema mediante el fichero "FreeRTOSConfig.h".

Con este experimento queda demostrado y caracterizado el ancho de banda alcanzable de base. En base a estos datos, en la Sección 3.5 se mostrará el desarrollo realizado en la adaptación de un middleware de comunicaciones para simplificar el tratramiento de los sistemas de comunicaciones.

3.5. Desarrollo de un middleware de comunicaciones

En las secciones anteriores se han realizado diferentes experimentos con el objetivo de medir y caracterizar cada una de las diferentes capas software para la comunicación entre un ordenador de gestión y un sistema de medición.

A modo de resumen, en las secciones anteriores se han estudiado las capacidades de comunicación en los siguientes escenarios:

- Baremetal — TCP: En esta configuración se ha analizado el impacto de la capa de transporte TCP sobre un sistema básico, es decir, sin niguna capa de abstracción.
- Baremetal — UDP: Se ha comprobado la diferencia entre TCP y UDP en ambos casos sin ninguna capa de abstracción.
- FreeRTOS — UDP: Se ha comprobado la diferencia entre la eficiencia del sistema de comunicaciones Baremetal y con una capa de abstracción de software como es FreeRTOS. De este modo, de forma indirecta se comprueba la sobrecarga de la capa FreeRTOS.

Tras realizar todas estas mediciones, **se llega a la conclusión** de que el uso de FreeRTOS no supone una sobrecarga excesiva y permite simplificar el diseño de forma considerable. Además, **se comprueba** que la pérdida de paquetes es muy baja (1.3 %) en el peor de los casos y es probable que mediante la optimización de las políticas de scheduler estas pérdidas desaparezcan.

En base a las conclusiones anteriores se propone la adaptación del middleware de comunicaciones ZeroC-Ice, comentado en la Sección 2.6.1

El middleware ZeroC-Ice ha sido desarrollado para ser ejecutado en un sistema con un sistema operativo de propósito general o, en el caso de sistemas embebidos, el sistema YOCTO.

Como trabajo futuro, se propone explorar la utilización y medir la diferencia entre, YOCTO y la implementación de ZeroC-Ice (desde ahora ice o icec).

En este proyecto se busca portar ZeroC-Ice para trabajar sobre las mínimas capas posibles y con las mínimas dependencias posibles. Con estas premisas en el laboratorio ARCO en el que el autor trabaja desde hace 4 años, se ha desarrollado una implementación del protocolo de comunicaciones Ice (utilizado por el middleware ZeroC-Ice) en C sin ningún tipo de dependencia.

3.5.1. Descripción de IceC

Ice-c se puede definir como: "middleware de comunicaciones orientado a objetos, diseñado específicamente para dispositivos con baja capacidad de cómputo.". Escrito en C y basado en un protocolo de comunicaciones binario este middleware mantiene una baja huella y permite su uso en todo tipo de dispositivos.

Entre las características del middleware se puede destacar:

- Portable: la adaptación icec está escrita completamente en ANSI C, esto permite que el middleware pueda ser ejecutado en cualquier dispositivo con soporte para un compilador C estándar.
- Extensible: en icec se ha implementado un sistema de *plug-in* que permite añadir soporte para nuevas plataformas y medios de comunicación de manera muy simple.
- Varios protocolos de comunicaciones soportados: actualmente icec cuenta con soporte para TCP/IP, UDP/IP, puerto serie, RS-485, FIFO Buffers, ZigBee, nRF24...
- Preparado para dispositivos con requisitos estrictos de memoria: la adaptación icec ha sido pensada, desde un primer momento, para soportar dispositivos de clase 0 según el RFC 7228. Actualmente ha sido testado en dispositivos basados en la arquitectura AVR como los famosos Arduino, que, en muchos casos, no tienen más de 23KiB de memoria para código y 2KiB de memoria para Random Access Memory (RAM)
- Protocolo de comunicaciones eficiente: el protocolo de comunicaciones icep ha sido diseñado para aprovechar al máximo las capacidades de la

red. Al contrario que muchos otros protocolos modernos, icep gestiona toda la comunicación de manera binaria. De este modo, las tareas de "parseo" se simplifican y aprovechan de un modo más inteligente las capacidades del procesado.

- Completamente compatible con el Interface Definition Language (IDL) de ZeroC-ice: el compilador *slice2c* es completamente compatible con las especificaciones del lenguaje *slice* de Zeroc-ice.

Antes de la realización de este proyecto icec no había sido portado a la plataforma Zynq y por lo tanto no era posible ejecutar el sistema sobre dicha arquitectura. En este trabajo se ha realizado el "port" de icec a la plataforma Zynq y se ha dado soporte para los protocolos de comunicación TCP y UDP.

3.5.2. Descripción de la aportación

La aportación principal de este proyecto, además del estudio pormenorizado de los diferentes sistemas de comunicación entre el sistema de medición y el ordenador de gestión, ha sido la adaptación del middleware de comunicaciones icec a la plataforma Zynq.

Actualmente, el "core" del middleware ha sido implementado y desarrollado como trabajo de investigación en el laboratorio ARCO de la Universidad de Castilla-La Mancha. Sin embargo, la adaptación a la arquitectura Zynq no había sido explorado hasta la realización de este trabajo.

Como resultado del desarrollo se ha generado:

- Adaptación completa **probada** en plataforma Zynq: El core de icec no había sido probado sobre el microprocesador ARM de la plataforma Zynq hasta la realización de este trabajo.
- Implementación de endpoints (más adelante se explicará el concepto de endpoint) TCP y UDP sobre FreeRTOS: en base a los estudios realizados y las necesidades inherentes al "core" de icec, las adaptaciones, tanto para TCP como para UDP se han realizado sobre FreeRTOS con soporte completa a todas las operaciones requeridas por icec.

- Adaptación del compilador slice2c: el compilador desarrollado por el grupo de investigación ARCO para el lenguaje "slice" de ZeroC-Ice carecía de soporte para el retorno de valores. Fruto de este trabajo y debido a las necesidades del proyecto se ha dado soporte al retorno de valores en la invocación a funciones.

En las siguientes secciones se analizará el desarrollo realizado para dar soporte al protocolo UDP sobre FreeRTOS. El soporte para TCP resulta muy similar debido a que ambos se basan en la orientación a sockets proporcionada por FreeROTS y estudiada en los experimentos anteriores.

3.5.3. Introducción a la arquitectura de Icec

En la Figura 3.17 se puede ver la salida del comando "tree" en GNU/Linux sobre el proyecto icec. En esta figura se distinguen las siguientes carpetas y ficheros de un primer nivel:

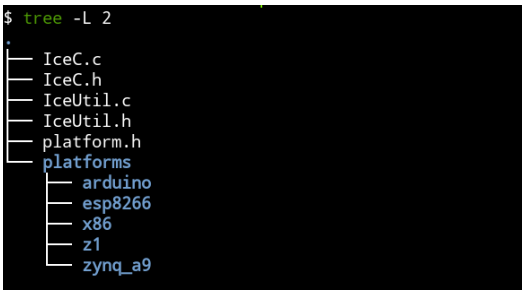

Figura 3.17: Arquitectura del proyecto

- IceC.c: fichero que implementa el "core" de ice en C. Tal y como se verá más adelante este fichero es estático y no ha necesitado ser modificado.
- IceC.h: definición de cabeceras para el "core" de icec.
- IceUtil.c: fichero con funciones de utilidad para usar dentro de las diferentes implementaciones o adaptaciones del middleware. Este fichero no ha tenido que ser modificado.

- IceUtil.h: definición de cabeceras para las utilidades contenidas en el fichero IceUtil.c.
- platform.h: fichero de definición de arquitecturas soportadas. Este fichero si que ha tenido que ser modificado para añadir la nueva plataforma soportada, es decir, la Zynq.
- platforms: carpeta donde se definen cada uno de las adaptaciones a las diferentes plataformas. Tal y como se puede ver, actualmente se encuentra portado para arduino, esp8266, x86 y z1. En esta carpeta se han añadido todos los cambios necesarios para el port de IceC sobre la zynq.

Dentro de la carpeta zynq_9 se han definido todos los ficheros necesarios para portar el middelware a dicha plataforma. En la Figura 3.18 se puede ver el resultado del comando "tree" sobre la carpeta "platforms" haciendo énfasis en la carpeta que interesa para el presente documento, es decir, la carpeta zynq_a9.

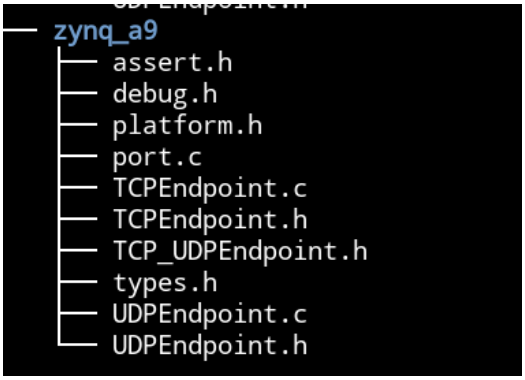

Figura 3.18: Arquitectura del "porting" a la zynq

En las siguientes secciones se explicará la función de cada uno de los ficheros para más tarde, demostrar en un ejemplo implementado en la placa cómo se puede utilizar este "port" para simplificar la labor de comunicaciones entre el sistema de medición y el ordenador de gestión.

3.5.4. Porting de IceC a la plataforma Zynq

En esta sección, tal y como se ha comentado anteriormente, se verán las diferentes fases así como el trabajo de diseño y desarrollo realizado para portar el middleware de comunicaciones IceC a la plataforma Zynq.

Durante el desarrollo de la sección se tratará de dar una visión general del trabajo realizado dado que ahondar en todos los conceptos inherentes a cada decisión incrementaría demasiado la complejidad del documento.

En primer lugar hay que aclarár que icec, en esencia, es únicamente una máquina de estados encargada de gestionar el protocolo de comunicaciones icep. Cualquier protocolo debe especificar 3 aspectos claves:

1. Sintaxis: la sintaxis de icep puede ser consultada en la documentación oficial [21].
2. Semántica: la semántica de cada parámetro puede ser consultada en la documentación indicada anteriormente. La semántica únicamente hace referencia al significado de cada uno de los parámetros y mensajes que forman el protocolo.
3. *timing*: la especificación del timing es la encargada de indicar en qué momento enviar cada mensaje.

Dentro del fichero IceC.h se pueden ver las declaraciones de las funciones encargadas de gestionar todos estos aspectos.

La complejidad de este fichero es bastante alta con más de 1400 líneas de código y decenas de funciones. Sin embargo, en esta sección se tratará de resumir el funcionamiento de la máquina de estados mediante un ejemplo. Para simplificar se supone que la inicialización de la máquina de estados icec ya ha sido realizada (más adelante, en el ejemplo de uso) se verá exactamente las funciones que se han de invocar para inicializar dicha máquina.

Sistema Ice_Plugin

En icec se ha optado por implementar un sistema basado en plugins a la hora de realizar la implementación de los diferenes endpoints. Esto quiere decir, un endpoint se añadirá a icec como un plugin más del middleware. Esto permite simplificar el desarrollo dado que, para añadir un nuevo endpoint, únicamente habrá que cumplir con una estructura de plugin bien definida.

Antes de avanzar, resulta interesante matizar qué se entiende por endpoint. Un endpoint es un punto de conexión del dispositivo con el mundo exterior. De este modo, un endpoint podría ser un trasnceiver Controller Area Network (CAN) o un transceiver . Siendo más estrictos, el endpoint es un nivel más abstracto, lejos de ser un endpoint el trasceiver MAC, el endpoint es el driver preparado para comunicar y recibir tramas icep sobre dicho bus.

En icec se define una API sencilla para registrar y usar plugins. Aunque la api expone gran cantidad de métodos, a continuación se mostrarán los más importantes:

- IcePlugin_registerEndpoint: registra un endpoint dentro de la lista de endpoints disponibles. De este modo, el sistema de plugins de icec puede requerir el objeto en caso de que sea necesario para algún tipo de conexión.
- IcePlugin_EndpointInfo_init: método invocado por icec en el momento de requerir el uso de un endpoint. Este método a su vez invocará la función "EndpointInfo_init" del plugin que se esté utilizando.

Además de las dos funciones mostradas anteriormente IcePlugin cuenta con un conjunto más amplio de funciones que serán utilizadas según se vayan requiriendo en cada caso.

El punto más importante a la hora de estudiar el sistema de plugin de icec es la definición de la interfaz del llamado "IcePlugin_EndpointObject" que se puede comprobar en el Listado 13. Este listado contiene una lista de punteros a función. Los punteros a las diferentes funciones hacen referencia a funciones que se deberán implementar en el endpoint concreto en caso de

que tengan sentido para el mismo. Icec comprobará, antes de hacer cualquier invocación, que dicho método existe. Si no existe fallará o continuará en silencio en función de la fase en la que se encuentre. A continuación se indica la finalidad de las funciones más importantes y que, en la mayoría de los casos, se requieren para el correcto funcionamineto del middleware:

- getProtocolType: esta función es la encargada de "responder" con True o False si implementa la gestión de un tipo de protocolo concreto. Por ejemplo, se puede dar el caso en que un endpoint gestione un protocolo de tipo CAN y por lo tanto, en el proxy que define al objeto que se desea invocar, aparezca la cadena "CAN" en la definicion del protocolo que se quiera usar. En este caso, un endpoint que sea capaz de gestionar el protocolo CAN deberá responder con True a la invocación de esta llamada.

- InputStream_init: delega al plugin la inicialización del sistema de comunicación con el otro extremo. Esta función será la encargada de inicializar dicha conexión. Típicamente, en esta función se inicializa un socket.

- EndpointInfo_readFromInputStream: función encargada de devolver un stream de datos desde el medio de comunicación concreto. Por ejemplo, en el port de Arduino para un endpoint wifi se utilizaría internamente la funcion *recv* de la clase *WifiServer*. Más adelante se verá la implementación concreta para el endpoint desarrollado en este trabajo.

- EndpointInfo_writeToOutputStream: similar a la función anterior con la salvedad de que, en este caso, la función se encargará de escribir, no de recibir.

- Connection_accept: función invocada por el "core" de icec cuando se acepte una conexión de tipo icep. Esta conexión no está ligada a las conexiones de tipos TCP. Son conexiones lógicas entre el middleware y el cliente.

- Connection_send: función que envía los datos de forma real.

- Connection_close: función invocada cuando el "core" desea cerrar una conexión con un cliente. Típicamente, en esta función se limpiarán to-

das las estructuras de datos utilizadas por el endpoint y se cerrarán los sockets (en caso de haberlos).

- Connection_dataReady: función invocada cuando el "core" desea comprobar si hay datos para leer de la conexión.

```
1  struct IcePlugin_EndpointObject {
2      Object _base;
3
4      bool (*getProtocolType)(const char*, Ice_EndpointType*);
5      bool (*InputStream_init)(Ice_InputStreamPtr, Ice_ConnectionPtr,
       ↪  Ice_EndpointType);
6      bool (*ObjectAdapter_activate)(Ice_ObjectAdapterPtr);
7
8      bool (*EndpointInfo_init)(Ice_EndpointInfoPtr, Ice_EndpointType, const
       ↪  char*);
9      bool (*EndpointInfo_writeToOutputStream)(Ice_EndpointInfoPtr,
       ↪  Ice_OutputStreamPtr);
10     bool (*EndpointInfo_readFromInputStream)(Ice_EndpointInfoPtr,
       ↪  Ice_InputStreamPtr);
11     bool (*EndpointInfo_toString)(Ice_EndpointInfoPtr, Ice_String*);
12
13     bool (*ObjectPrx_init)(Ice_ObjectPrxPtr, Ice_EndpointType, const
       ↪  char*,
14                 IcePlugin_EndpointObjectPtr);
15     bool (*ObjectPrx_connect)(Ice_ObjectPrxPtr);
16
17     bool (*Connection_accept)(Ice_ConnectionPtr);
18     bool (*Connection_send)(Ice_ConnectionPtr, byte*, uint16_t*);
19     bool (*Connection_close)(Ice_ConnectionPtr);
20     bool (*Connection_dataReady)(Ice_ConnectionPtr, bool*);
21
22 };
23
24 typedef struct IcePlugin_EndpointObject IcePlugin_EndpointObject;
```

Listado 13: Definición del "objeto" IcePlugin_EndpointObject

Con el sistema de plugins de icec estudiado, el siguiente paso para realizar un "port" a una **nueva plataforma** consiste en proporcionar una serie de ficheros que se encargarán de definir dicha plataforma.

Definición de una nueva arquitectura

En el apartado anterior se ha comentado, brevemente, la arquitectura de plugins utilizada por Icec. Esta arquitectura de plugins únicamente se utiliza para utilizar nuevos endpoints en una arquitectura ya existente. Sin embargo, para la realización de este trabajo también se ha tenido que portar el soporte a la arquitectura completa.

Para definir una arquitectura, tal y como se puede ver en la Figura 3.17 se deben aportar los siguientes ficheros:

- platform.h: este fichero es el "punto de entrada" para el sistema de "ports" del middlware icec. En este fichero se deberán especificar aquellas funciones específicas de plataforma (un ejemplo podría ser malloc, aunque en este middleware no se utiliza). También se deben definir las constantes:

  1. MAX_MESSAGE_SIZE: indica el tamaño máximo del payload que puede gestionar icec.
  2. MAX_IDENTITY_SIZE: indica el tamaño máximo del string que define un "indentity" ice.
  3. MAX_NESTED_ENCAPS: indica el número máximo de encapsulaciones icec dentro de un paquete ice (anidación).
  4. MAX_NUM_OF_ADAPTERS: indica el número máximo de adaptadores de objetos por comunicador. Este parámetro es importante si un sistema puede gestionar comunicaciones desde varios endpoints diferentes.
  5. MAX_NUM_OF_SERVANTS: número máximo de objetos distribuidos que puede servir el dispositivo.

6. MAX_PROTO_SIZE: actualmente no se está utilizando.
7. MAX_OPERATION_NAME_SIZE: indica el tamaño máximo del string que representa una operación.

Dentro de este fichero también se definirán los "include" específicos para la plataforma. Por ejemplo, el middleware icec utiliza una función de debug para realizar la traza del programa. En función del dispositivo en el que se encuentre puede ser más adecuado realizar el debug de una manera u otra. Mediante el "include" del fichero debug.h se puede indicar el modo en que se gestiona el debug.

- debug.h: este fichero permitirá al middleware utilizar los recursos de comunicación necesarios para realizar el log del sistema. En funcion del dispositivo puede resultar interesante mandar los datos por un tipo de conexión u otra. Por ejemplo, en dispositivos con una unidad para tarjetas SD puede resultar interesante utilizar dicha tarjeta para guardar las trazas del sistema. Dentro de este fichero habrá que definir funciones como:
  - hexdump: permite decodificar y mostrar en formato hexadecimal una trama.
  - trace: macro que se mapeará a la función de entrada/salida utilizada. Por ejemplo, en el caso de la plataforma Zynq, se ha mapeado a la función xil_printf.
  - fixme: exactamente lo mismo que la función anterior pero, con la diferencia de que esta macro será utilizada por el middleware únicamente en los casos donde sea necesario mostrar una advertencia de un comportamiento que necesita ser refinado.
  - error_at_line: esta macro permite gestionar los errores de una forma sencilla. En las próximas versiones de icec esta función será eliminada para dar sopote a excepciones anidadas.
- assert.h: esta función permite definir el modo en que se gestionan los "assert" en icec. Es **muy importante** implementar una buena estrategia a la hora de gestionar como se procesan los assert o de lo contrario se puede dejar al procesador en un estado inconsistente.

- port.c: encargada de proporcionar la función "IceInternal_selectOnAdapters". Esta función será la encargada de gestionar el bucle principal para la detección de mensajes entrantes o salientes.
- types.h: definición de los diferentes tipos de datos utilizados por la plataforma que se desea portar.

Con lo explicado anteriormente, el lector tiene los conocimientos suficientes para seguir de modo superficial, el trabajo desarrollado en el "port" de icec para la Zynq realizado en este trabajo. De nuevo, se hace énfasis en que únicamente se explicarán aquellos aspectos más relevantes del port realizado. Algunos aspectos como: configuración de constantes, funciones de debug, etc. no serán explicados con el objetivo de mantener simple el documento.

Definición de la arquitectura Zynq

En primer lugar se describirán los cambios realizados en los ficheros nombrados anteriormente para realizar el "port" a la plataforma Zynq.

En primer lugar, en el fichero platform.h se indica que la plataforma es de tipo "little endian". Por otro lado, se definen los tipos de dato estándares proporcionados por el compilador estándar de arm y se incluyen las funciones de debug utilizando en todo momento el puerto serie de la plataforma.

Una vez definidas las funciones de soporte comentadas anteriormente, se implementa la función "IceInternal_selectOnAdapters".

En el Listado 14 se puede ver el código (simplificado) implementado para dar soporte a esta función.

En primer lugar, en este código se comprueba el número de NetworkBuffers restantes. Los NetworkBuffers, tal y como se verán en el ejemplo de uso del sistema desarrollado, son una estructura de datos básica para la gestión de memoria en FreeRTOS.

Seguidamente, se realiza un bucle por cada uno de los adaptadores existentes en el comunicador. Por cada uno de los adaptadoers se comprueba

el tipo de endpoint. Un adaptador puede tener un endpoint de tipo TCP, o de UDP (endpoints portados en este trabajo a la Zynq). En función del adaptador se obtiene el elemento que determina la conexión.

Una de las **razones** por las que se ha utilizado FreeRTOS para el port de icec es que tiene soporte para la operación *select*. Esta operación, permite recibir notificaciones cuando al menos un socket tenga eventos de: escritura, lectura o error. De este modo, con realizar un select sobre un conjunto de sockets, FreeRTOS se encargará de gestionar la detección de bytes de entrada/salida/error y notificar a la tarea correspondiente para su posterior tratamiento.

En este caso únicamente se selecciona el bit "SELECT_READ" debido a que en esta operación únicamente interesa saber cuando se ha recibido un nuevo byte. Una vez configurada la operación select, se comprueba si existe algún socket con datos para ser leídos (ver Línea 41). En caso afirmativo, se confirma a FreeRTOS que los datos han sido gestionados y se guarda el adaptador de objetos que contiene dicho socket en una variable de tipo Ice_ObjectAdapterPtr que será utilizada en el fichero IceC.c para leer de forma efectiva los datos.

Gracias al uso de FreeRTOS, en la Línea 41 se "avisará" al *scheduler* de FreeRTOS que se va a realizar una operación de entrada/salida y que por lo tanto puede utilizar el tiempo hasta "portMAX_DELAY" para atender a otras rutinas.

```
1  void
2  IceInternal_selectOnAdapters(Ice_ObjectAdapter_ListPtr adapters,
3                  Ice_ObjectAdapterPtr result[],
4                  bool block) {
5      int i, count;
6      Ice_ObjectAdapterPtr adapter;
7      SocketSet_t xFD_Set;
8      Socket_t socket_map[MAX_NUM_OF_ADAPTERS];
9      Ice_ObjectAdapterPtr adapter_map[MAX_NUM_OF_ADAPTERS];
10
11     trace();
12 #ifdef DEBUG
```

```
13      DEBUG( ( "[DBG-PORT] Network buffers: %lu (current free) lowest %lu
        ↪  (history free)\n",
14              uxGetNumberOfFreeNetworkBuffers(),
                ↪  uxGetMinimumFreeNetworkBuffers() ) );
15  #endif
16
17      count = 0;
18      for (i=0; i<adapters->count; i++) {
19          adapter = *(adapters->begin + i);
20          if (! adapter->activated) {
21              continue;
22          }
23          Socket_t socket;
24          TCP_UDPEndpointOptions *options;
25
26          /* Add the created socket to the set. */
27          options =
            ↪  ((TCP_UDPEndpointOptions*)(adapter->connection.epinfo->options));
28          if (adapter->connection.epinfo->type == Ice_EndpointTypeTCP) {
29            socket = (options->connected_socket !=
              ↪  NULL)?options->connected_socket:options->socket;
30          }
31          else {
32            socket = options->socket;
33          }
34          xFD_Set = options->socket_set;
35          FreeRTOS_FD_SET( socket, xFD_Set, eSELECT_READ);
36          socket_map[count] = socket;
37          adapter_map[count] = adapter;
38          count++;
39      }
40
41      if ( (FreeRTOS_select( xFD_Set, portMAX_DELAY ))!= 0) {
42
43          for (i=0; i<MAX_NUM_OF_ADAPTERS; i++)
44              result[i] = NULL;
45
46          for (i=0; i<count; i++) {
```

```
47              BaseType_t event = FreeRTOS_FD_ISSET(socket_map[i], xFD_Set);
48              if (event) {
49                  result[i] = adapter_map[i];
50                  break;
51              }
52          }
53      }
54
55  }
```

Listado 14: Código para gestionar la selección del adaptador

El Listado 14 es uno de los puntos de entrada desde el código genérico e independiente de plataforma IceC.c al código dependiente de la plataforma.

Tal y como se ha visto, en el código se hace referencia a diferentes endpoints. Los endpoints, tal y como se explicó en la Sección 3.5.4 son elementos de tipo "plug-in" y, por lo tanto, dependientes de una plataforma concreta.

El siguiente paso es definir estos endpoints. Tal y como se ha comentado anteriormente, en este trabajo se ha realizado el "port" de los plugin TCP y UDP. No obstante, en este documento únicamente se detallará el código para implementar el endpoint UDP.

3.5.5. Implementación endpoint UDP sobre FreeRTOS

En este punto del documento el lector habrá podido observar y que los endpoints y la plataforma están claramente disociados aunque en última instancia hay un punto de unión. El punto de unión es el código mostrado en el Listado 14 que fue explicado en la sección anterior.

En esta sección se detallará el código implementado para dar soporte al protocolo de comunicaciones UDP sobre FreeRTOS.

En el Listado 18 se puede ver el fichero UDPEndpoint.h. Este código es el encargado de definir las diferentes funciones utilizadas dentro del fichero

UDPEndpoint.c. Como se puede observar, las funciones son muy específicas de cada tipo de comunicación. Por ejemplo, el método connect no se realizará del mismo modo en UDP (no hay conexión como tal) y bluetooth (existe una conexión y emparejamiento). Por esta razón este código se ha separado del "core" de icec.

A continuación, en el Listado 17 se puede ver la función de inicialización del endpoint. Este es uno de los puntos de entrada al endpoint y su función es la de configurar los diferentes callbacks que serán utilizados por el "code" de icec para ralizar las diferentes acciones.

En el caso de UDP se han definido las siguientes funciones:

- getProtocolType: esta función únicamente comprueba si la cadena recibida como parámetro (enviada desde el core de icec) es UDP. En caso afirmativo se devuelve el valor verdadero.
- InputStream_init: el objetivo de esta función es el de "rellenar" el buffer interno con los datos del socket.
- ObjectAdapter_activate: esta función se invoca en el "core" de ice cuando la aplicación de usuario realiza una activación de un adaptador. El objetivo de esta función es iniciar la escucha en el puerto especificado en el proxy.
- ObjetcPrx_init: cuando el dispositivo está actuando como cliente y no como servidor, una invocación a un objeto distribuido pasa por un objeto de tipo proxy. El proxy es el encargado de hacer el paso transparente de los parámetros y gestionar, en el caso de una función con valores de retorno, dichos valores. El objetivo de esta función es iniciarlizar un objeto cuando algún objeto distribuido con el proxy UDP es instanciado. De este modo, se inicializa la conexión y el flujo de salida.
- ObjectPrx_connect: implementado en la función UDP_ObjectPrx_connect, esta función es la encargada de realizar la petición de inicio de conexión al servidor que gestionar el objeto distribuido al que este proxy representa. Tal y como se puede ver en el Listado 16 en primer lugar se obtiene el puerto y la dirección de destino (Líneas 1617). Una vez

adquiridos tanto la IP como el puerto, se realiza la conexión con el servidor.

- Connection_send: función encargada de enviar datos mediante el socket establecido anteriormente (ver Listado 15. Esta función es bastante simple de implementar pues cuando se invoca se suponen todas las estructuras inicializadas y por lo tanto únicamente hay que hacer una petición por el socket.

```
1  bool
2  UDP_Connection_send(Ice_ConnectionPtr self, byte* data, uint16_t* size) {
3      struct freertos_sockaddr xDestinationAddress;
4      int32_t isize_send;
5      trace();
6
7      if (self->epinfo->type != Ice_EndpointTypeUDP)
8          return false;
9
10     UDP_EndpointOptions* opts =
       ↪  (UDP_EndpointOptions*)self->epinfo->options;
11
12     xDestinationAddress.sin_addr = FreeRTOS_inet_addr( opts->host );
13     xDestinationAddress.sin_port = FreeRTOS_htons( opts->port );
14
15     //*size = FreeRTOS_send( opts->socket, data, *size, 0 );
16     isize_send = FreeRTOS_sendto( opts->socket, data, *size, 0,
       ↪  &xDestinationAddress, sizeof(xDestinationAddress) );
17
18     if(isize_send != *size){
19       FreeRTOS_printf( ("Could not send the message") );
20       return false;
21     }
22
23     return true;
24 }
```

Listado 15: Función encargada de enviar datos mediante el socket establecido

```
bool
UDP_ObjectPrx_connect(Ice_ObjectPrxPtr self) {
    Ice_ConnectionPtr conn;
    UDP_EndpointOptionsPtr opts;
    struct freertos_sockaddr serverAddress;

    trace();
    if (self->epinfo.type != Ice_EndpointTypeUDP)
        return false;

    conn = &(self->connection);
    Ptr_check(conn);
    opts = self->epinfo.options;
    Ptr_check(opts);

    serverAddress.sin_port = FreeRTOS_htons( opts->port );
    if (!inet_aton(opts->host, &serverAddress.sin_addr)) {
      FreeRTOS_printf( ("Invalid host name: %s\n", serverAddress.sin_addr
      ↪  ));
      return false;
    }

    return open_freertos_socket(conn);
}
```

Listado 16: Código de la función encargada de gestionar un objeto remoto

```
void
UDPEndpoint_init(Ice_CommunicatorPtr ic) {
    trace();

    Ptr_check(ic);

    ep_options.used = false;
```

```
8       ep_options.socket = NULL;
9       ep_options.connected_socket = NULL;
10      Object_init((ObjectPtr)&ep_options);
11
12      IcePlugin_EndpointObject_init(&endpoint);
13      IcePlugin_PluginListItem_init(&item, (ObjectPtr)&endpoint);
14      IcePlugin_registerEndpoint(&item);
15
16      endpoint.getProtocolType = &UDP_getProtocolType;
17      endpoint.InputStream_init = &UDP_InputStream_init;
18      endpoint.ObjectAdapter_activate = &UDP_ObjectAdapter_activate;
19
20      endpoint.EndpointInfo_init = &UDP_EndpointInfo_init;
21      endpoint.EndpointInfo_writeToOutputStream = NULL;
22      endpoint.EndpointInfo_readFromInputStream = NULL;
23
24      endpoint.ObjectPrx_init = &UDP_ObjectPrx_init;
25      endpoint.ObjectPrx_connect = &UDP_ObjectPrx_connect;
26
27      endpoint.Connection_accept = NULL;
28      endpoint.Connection_send = &UDP_Connection_send;
29      endpoint.Connection_close = NULL;
30      endpoint.Connection_dataReady = NULL;  /* FIXME: not implemented */
31  }
```

Listado 17: Código de inicialización del endpoint

```
1   #ifndef _ICE_UDPENDPOINT_H_
2   #define _ICE_UDPENDPOINT_H_
3
4   #include <IceC/IceC.h>
5   #include "TCP_UDPEndpoint.h"
6
7   #define Ice_EndpointTypeUDP   3
8
9   typedef TCP_UDPEndpointOptions* UDP_EndpointOptionsPtr;
10  typedef TCP_UDPEndpointOptions UDP_EndpointOptions;
```

```
11
12 bool UDP_getProtocolType(const char* proto, Ice_EndpointType* result);
13 bool UDP_InputStream_init(Ice_InputStreamPtr self, Ice_ConnectionPtr
   ↪  connection, Ice_EndpointType type);
14 bool UDP_ObjectAdapter_activate(Ice_ObjectAdapterPtr self);
15 bool UDP_EndpointInfo_init(Ice_EndpointInfoPtr self,
16                            Ice_EndpointType type,
17                            const char* endpoint);
18
19 bool UDP_ObjectPrx_init(Ice_ObjectPrxPtr self, Ice_EndpointType type,
   ↪  const char* strprx,
20                     IcePlugin_EndpointObjectPtr endp);
21 bool UDP_ObjectPrx_connect(Ice_ObjectPrxPtr self);
22
23 bool UDP_Connection_listen(Ice_ConnectionPtr self, const char* host,
   ↪  uint16_t port);
24 bool UDP_Connection_send(Ice_ConnectionPtr self, byte* data, uint16_t*
   ↪  size);
25
26 /* plugin specific functions */
27 void UDPEndpoint_init(Ice_CommunicatorPtr ic);
28
29 #endif /* _ICE_UDPENDPOINT_H_ */
```

Listado 18: Cabecera del endpoint implementado

3.6. Experimento 5: Uso del middleware de comunicaciones

En la sección anterior se ha analizado, de forma resumida, la implementación del middleware de comunicaciones icec a la placa Zynq. Con el objetivo de comprobar la eficiencia del protocolo así como mostrar como simplifica el desarrollo. En esta sección se realizará un ejemplo equivalente a iperf mediante icec.

En esta sección se detallará el proceso para ir desde una IDL a la implementación final. Únicamente se detallarán aquellos aspectos que se consideren de interés para el lector.

3.6.1. Descripción del experimento / Objetivos

En este experimento se implementará un sistema de comunicaciones que sirva de prueba equivalente a iperf imitando su comportamiento.

El objetivo principal es comprobar que el middleware funciona de manera sostenida. Además, se busca caracterizar, de un modo similar a como lo hace el iperf, las capacidades en cuanto a pérdida de paquetes y ancho de banda del middleware.

En este caso, no se podrá utilizar el programa iperf cliente debiado que icec utiliza una encapsulación propia (tal y como se ha explicado en la Sección 2.6.1) y por tanto no "entenderá" los datos enviados por el cliente iperf.

Por esta razón, y con el objetivo de realizar las mismas medidas, se implementará una interfaz que simule el comportamiento de iperf.

3.6.2. Hipótesis

Se espera que el ancho de banda se reduzca debido a la sobrecarga del middleware de comunicaciones. Además, dado que está basado en FreeRTOS,

se espera una perdida de paquetes al rededor del 1 %.

Además, se espera que haya que realizar alguna personalización en cuanto a los tamaños de memoria y el dimensionamiento de los diferentes bufferes. Esto se debe a que, en los experimentos anteriores, se partió de ejemplos de Xilinx donde los tamaños de memoria habían sido modificados para gestionar las diferencias de velocidad

3.6.3. Desarrollo del experimento

En esta sección se tratará de guiar al lector en el código implementado. Los 4 proyectos indicados a continuación se encuentran adjuntos al documento. El lector únicamente tendrá que importar dichos proyectos desde el menú de importación del SDK.

El hardware utilizado es el mismo que en los experimentos anteriores, es decir, únicamente se ha utilizado el SoC Zynq. Si el lector desea replicar el ejemplo, es especialmente importante que el BSP tenga el siguiente nombre: freertos_xilinx_bsp_0, tal y como se puede ver en la Figura 3.19.

Una vez generado el BSP, el lector puede importar los proyectos:

- freeros_ICE: este proyecto incorpora la versión de icec preparada para ser ejecutada en la Zynq, es decir, cuenta con todos los cambios descritos en la sección anterior.
- freertos_TCP: este proyecto ha sido generado a partir del port oficial de la pila TCP de FreeRTOS para la plataforma Zynq. En la Sección 2.5 se habló sobre esta capa de comunicaciones.
- udp_server: en este proyecto se encuentra el código que implementa la "lógica de negocio". Este código será el encargado de crear los objetos distribuidos y usar el hardware determinado.

El resultado del proyecto se puede ver en la Figura 3.20.

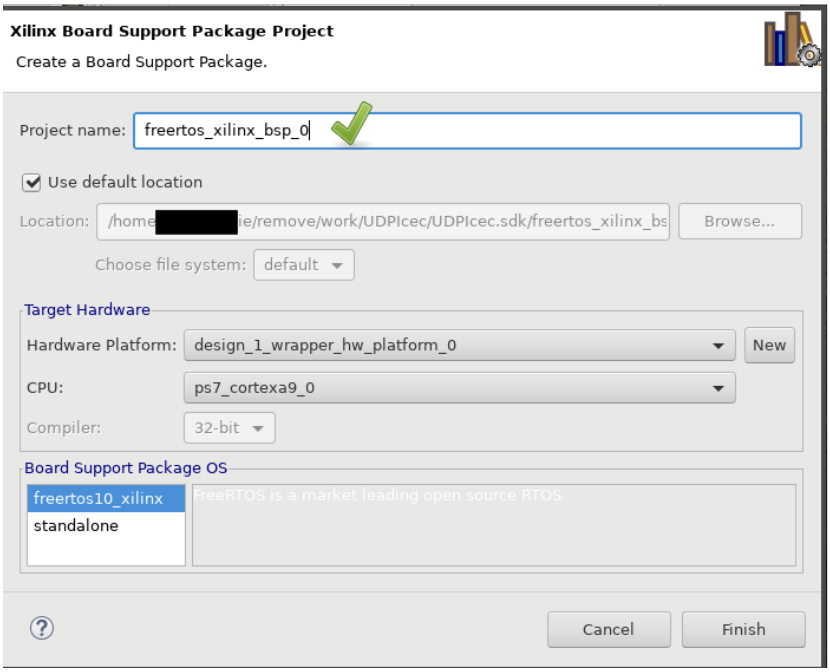

Figura 3.19: Configuración de BSP para Vivado 2018.1 y FreeRTOS compatible con Icec

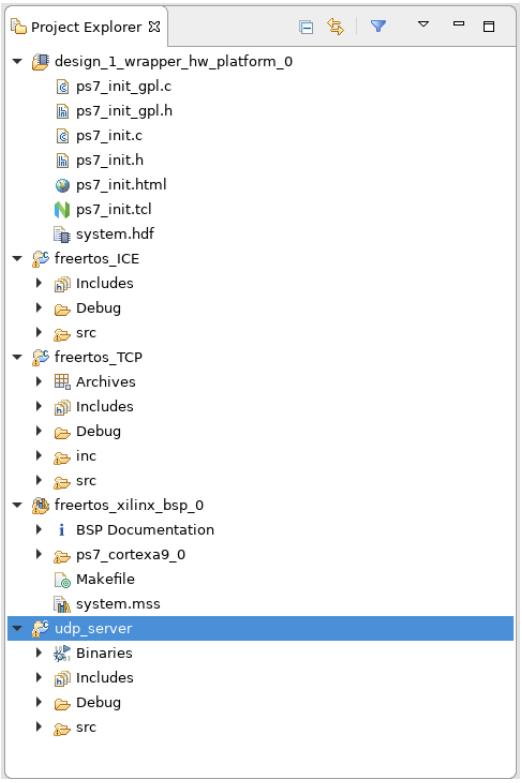

Figura 3.20: Ejemplo de *workspace* una vez importados los proyectos para la gestión de icec

Este experimento pretende servir como ejemplo de cómo se simplifican las comunicaciones mediante el uso de un middleware como icec. Por esta razón en primer lugar se empezará por mostrar las bondades de este tipo de middlewares.

Normalmente, cuando se inicia la labor de crear un sistema distribuido, el primer paso consiste en determinar las interfaces entre cada uno de los subsistemas. Esta interfaz permite que el nodo que proporcione dicha interfaz, modifique su comportamiento y todos los demás subsistemas que colaboren con él seguirán ejecutándose sin ningún problema. Esto se debe a que las interfaces definen la arquitectura, no la estructura (haciendo una analogía con la jerga de los lenguajes de descripción de hardware).

En el Listado 19 se puede ver la interfaz definida para la emulación del servidor iperf. Como se ha visto en secciones anteriores, el comando iperf únicamente consiste en enviar datos (desde el cliente) y procesarlos en el servidor. En función del tipo de prueba que se realice, el servidor hará una de las siguientes acciones:

- UDP: Almacena el número de secuencia (primer entero) y comprueba que haya llegado en orden. En caso de no haber llegado en orden, se aumenta el número de paquetes perdidos.
- TCP: Únicamente se retorna el ACK al cliente.

```
1 module Example {
2     sequence<int> payload;
3     interface Iperf {
4         void sendPayload(payload message);
5     };
6 };
```

Listado 19: Definición de una interfaz compatible con iperf

Una vez escrita la interfaz en el formato slice, el siguiente paso consiste en convertir dicha intefaz al código concreto en función del lenguaje que se desee. Actualmente exisen compiladores para los lenguajes: javascript, c++, python, c, php...

Para este proyecto se utilizarán los lenguajes Python y c, este último ha sido desarrollado por el laboratorio ARCO y forma parte del proyecto icec. Para generar el código a partir de la intefaz se utilizará el comando: slice2xx donde xx será el identificador del lenguaje destino. Por ejemplo, en el caso de python, el comando será slice2py o, en el caso de ansi c, el comando será slice2c.

Una vez generado el código ansi c deberá ser copiado al proyecto udp_server.

En el Listado 20 se puede ver el código principal a la hora inicializar el servidor UDP udp o icec.

A continuación se irá detallando paso a paso el proceso para inicializar el servidor.

En primer lugar, se configura la placa mediante la función prvSeupHardware. Los principales elementos a configurar son el controlador de interrupciones y el timer. El controlador de interrupciones se utilizará para gestionar las diferentes interrupciones pero, principalmente, las interrupciones generadas por el timer.

El siguiente paso, consiste en inicializar la pila TCP/IP de FreeRTOS. FreeRTOS proporciona la función FreeRTOS_IPInit que, como parámetros, recibe la dirección IP del dispositivo.

Finalmente, tal y como se vio en la Sección 2.5, se inicia el scheduler mediante la función vTaskStartScheduler() que se encarga de iterar por cada tarea en base a la política establecida.

```
1  int main()
2  {
3          time_t xTimeNow;
4
5          /* platform setup */
6          prvSetupHardware();
7
8          /* Seed the random number generator. */
9          time( &xTimeNow );
10         prvSRand( ( uint32_t ) xTimeNow );
11
12         xil_printf( ( "FreeRTOS_IPInit\n" ) );
13         FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress,
14                          ucDNSServerAddress, ucMACAddress );
15
16         /* Start the tasks and timer running. */
17         vTaskStartScheduler();
18
19         for( ;; );
20
21         return 0;
```

```
22  }
23
24  void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent )
25  {
26      uint32_t ulIPAddress, ulNetMask, ulGatewayAddress, ulDNSServerAddress;
27      char cBuffer[ 16 ];
28      static BaseType_t xTasksAlreadyCreated = pdFALSE;
29
30      /* If the network has just come up...*/
31      if( eNetworkEvent == eNetworkUp )
32      {
33          if( xTasksAlreadyCreated == pdFALSE )
34          {
35              xTasksAlreadyCreated = pdTRUE;
36              xTaskCreate( IceTask, "IceTask", ICE_TASK_STACK_SIZE, NULL,
37                           ICE_TASK_PRIORITY, NULL );
38          }
39      }
40  }
41
42  void IceTask(void *pvParameters)
43  {
44      uint32_t ulIPAddress, ulNetMask, ulGatewayAddress, ulDNSServerAddress;
45      char cBuffer[ 16 ];
46      Ice_ObjectAdapter adapter;
47      Example_Iperf servant;
48
49      /* create Ice communicator */
50      FreeRTOS_printf( ( "Setting up ICE\r\n") );
51      Ice_initialize(&ic);
52      UDPEndpoint_init(&ic);
53
54      /* create object adapter */
55      Ice_Communicator_createObjectAdapterWithEndpoints(&ic, "Adapter", "udp
        ↪  -h 0.0.0.0 -p 7891", &adapter);
56      Ice_ObjectAdapter_activate(&adapter);
57
58      /* register servant */
```

```
59      Example_Iperf_init(&servant);
60      Ice_ObjectAdapter_add(&adapter, (Ice_ObjectPtr)&servant,
        ↪  "IperfServer");
61
62      Ice_Communicator_waitForShutdown(&ic);
63
64      for(;;);
65  }
```

Listado 20: Código principal para la inicialización del servidor icec

En la Línea 24 se encuentra la función invocada por FreeRTOS cuando se ha configurado la interfaz de red. Es en esta parte del código donde se configura realmente el servidor icec. En la Línea 51 se puede ver la inicialización del stack icec. Esta función únicamente se encarga de inicializar las estructuras internas utilizadas por el middleware. La línea más importante es la siguiente (Línea 52). Mediante la función UDPEndpoint_init se configura el plugin para el endpoint UDP. El código de esta función se mostró en la sección anterior, concretamente en el Listado 17. Esta función únicamente configurará los "callback" que serán ejecutados ne cada fase de la comunicación.

Seguidamente, se inicializa el adaptador de objetos encargado de recibir las invocaciones a la identidad "IperfServer". Cuando el "core" de icec reciba una invocación con la identidad "IperfServer" llegará a este adaptador de objetos y, este último, delegará la tarea de procesar la invocación al "servant".

Por último, en la Línea 62 se inicia el bucle principal de icec.

Tal y como se explicó en la sección anterior, el método selectOnAdapters debe ser redefinido por cada arquitectura y deberá contemplar los diferentes endpoints. De este modo, en el bucle principal de ice se pregunta por los adaptadores disponibles con mensajes y se procesan los mensajes para más tarde delegar el proceso al adaptador.

```
1  void ICEC_FUNC_ATTR
2  Ice_Communicator_waitForShutdown(Ice_CommunicatorPtr self) {
```

```
3       uint8_t i;
4       Ice_ObjectAdapterPtr result[MAX_NUM_OF_ADAPTERS];
5
6       trace();
7       Ptr_check(self);
8
9       while (true) {
10          IceInternal_selectOnAdapters(&(self->adapters), result, true);
11
12          for (i=0; i<self->adapters.count; i++) {
13              if (result[i] == NULL)
14                  continue;
15
16              Ice_ObjectAdapter_incomingMessage(result[i]);
17          }
18      }
19  }
```

Listado 21: Bucle principal de icec

Al generar el código "c" a partir del código slice mediante el compilador slice2c se genera un código con el nombre del fichero que contiene la interfaz y la extensión ".h". Este fichero contiene el código necesario para procesar los mensajes icec una vez se delega el procesamiento al adaptador. En el Listado 22 se puede ver parte del código, generado automáticamente.

```
1   static void
2   Example_Iperf_methodHandler(Example_IperfPtr self,
3                               Ice_InputStreamPtr is,
4                               Ice_OutputStreamPtr os) {
5       Ice_Int operationSize = 0;
6       char operation[MAX_OPERATION_NAME_SIZE];
7       byte mode = 0;
8       byte contextSize = 0;
9       Ice_Int encapSize = 0;
10      byte major = 0;
11      byte minor = 0;
```

```
12
13      trace();
14
15      Ptr_check(self);
16      Ptr_check(is);
17      Ptr_check(os);
18
19      Ice_InputStream_readSize(is, &operationSize, false);
20      assert(operationSize <= MAX_OPERATION_NAME_SIZE);
21
22      Ice_InputStream_readString(is, operation);
23
24      Ice_InputStream_readByte(is, &mode);
25
26      Ice_InputStream_readByte(is, &contextSize);
27      assert(contextSize == 0);
28
29      Ice_InputStream_readInt(is, &encapSize);
30      Ice_InputStream_readByte(is, &major);
31      Ice_InputStream_readByte(is, &minor);
32      assert(major == 1 && minor == 0);
33      assert(encapSize - 6 == (is->size - (is->next - is->data)));
34
35      if (strcmp(operation, "sendPayload") == 0) {
36          Example_Iperf_sendPayload__dispatch(self, is, os);
37          return;
38      }
39
40      c_throw("Ice_OperationNotExistException", __FILE__, __LINE__);
41  }
42
43  void
44  Example_IperfI_sendPayload(Example_IperfPtr self,
45                             Example_payload message) __attribute__((weak));
46
47  static void
48  Example_Iperf_sendPayload__dispatch(Example_IperfPtr self,
49                                      Ice_InputStreamPtr is,
```

```
50                                      Ice_OutputStreamPtr os) {
51      Example_payload message;
52
53      trace();
54
55      if (&Example_IperfI_sendPayload == 0) {
56          return;
57      }
58
59      /* allocate space for sequence */
60      Ice_InputStream_readSize(is, &message.size, false);
61      byte message_items_data[sizeof(int) * message.size];
62      message.items = (int*)message_items_data;
63      Example_payload_readFromInputStream(&message, is);
64
65      Example_IperfI_sendPayload(self, message);
66  }
```

Listado 22: Código iperf.h autogenerado por slice2c

Esto solo es parte del código autogenerado. Explicar todo el código generado haría que el documento excediera a longitud deseada. En primer lugar, en el Listado 22, se puede ver la función Example_Iperf_methodHandler. Esta función se encarga de comprobar si la invocación delegada por el "core" de icec puede ser procesada por este "servant". En caso de que esta operación haya sido contemplada en la creación de la interfaz, se realiza el dispatch mediante la función Example_Iperf_sendPayload_dispatch, para este ejemplo.

El método dispatch únicamente se redireccionará la invocación al método final implementado por el usuario. Justo en la línea superior al método dispatch se puede ver la declaración de una función con el atributo *weak*. Este atributo hace referencia a que la función será sobreescrita en otro fichero por el usuario y esta será la implementación utilizada.

Por último, queda definir qué se desea realizar cuando se invoque el método sendPayload. Para poder permitir a icec redireccionar a este función es importante que la misma tenga exactamente la misma "firma" que la función autogenerada. Para conseguir esto se recomienda copiar y pegar la línea con

el atributo “weak” y, finalmente, borrar dicho atributo.

En el Listado 23 se puede ver el código implementado para atender a dicha invocación.

```
void
Example_IperfI_sendPayload(Example_IperfPtr self,
                           Example_payload message) {


    if(0 == message.items[0]){
        old = xTaskGetTickCount();
    }

    if(expected_id != message.items[0]){
        dropped += 1;
    }
    expected_id += 1;
    if(message.items[0] == -1){

        new = xTaskGetTickCount() - old;
        //xil_printf("%d\n\r", xTaskGetTickCount() - old);
        //xil_printf("Dropped: %d\n\r", dropped);
        reset_stats();
    }
}
```

Listado 23: Implementación de la interfaz iperf.h

El código es muy simple con el objetivo de que la sobrecarga sobre la medición sea mínima. Cuando se atiende una invocación se comprueba el primer entero del payload. Si el entero es 0 entonces se marca el inicio del tiempo. Cuando llega el último mensaje se marca con un -1 y se guarda un timestamp. Con esos valores , se puede tener una medida indirecta del tiempo de procesamiento por cada invovación y el número de paquetes perdidos.

Con esta última pieza de código queda completa la parte del servidor. El siguiente paso sería definir el código del cliente.

Tal y como se ha comentado anteriormente, el lenguaje IDL slice, puede ser "traducido" a otros lenguajes. De este modo, el servidor puede estar implementado en c (tal y como se ha realizado en el código anterior) y el cliente en cualquier otro lenguaje.

En el Listado 24 se puede ver el código el cliente implementado para simular un cliente iperf simple.

```
1  import sys
2  import Ice
3  import time
4
5  Ice.loadSlice("iperf.ice")
6  import Example
7
8
9  class Client(Ice.Application):
10     def run(self, args):
11         if len(args) < 2:
12             return self.usage()
13
14         ic = self.communicator()
15         server = ic.stringToProxy(args[1])
16         server = Example.IperfPrx.uncheckedCast(server)
17
18         payload = []
19         for i in range(0, 350):
20             payload.append(i % 255)
21
22         counter = 0
23
24         while True:
25             for i in range(0, 4900):
26                 payload[0] = counter
27                 server.sendPayload(payload)
28                 counter += 1
29
30             counter = -1
```

```
31              payload[0] = counter
32              server.sendPayload(payload)
33              time.sleep(0.1)
34
35      def usage(self):
36          print("USAGE: {0} <proxy>".format(sys.argv[0]))
37
38
39  if __name__ == '__main__':
40      Client().main(sys.argv)
```

Listado 24: Implementación del cliente en python

El código del cliente es muy sencillo. En primer lugar se crea una clase que hereda de una clase "helper" proporcionada por Ice. Esta clase es "Ice.Application". El uso de esta clase permite simplificar el manejo de las diferentes interrupciones del sistema operativo así como la obtención del comunicador etc. Una vez creada la clase, se crea el payload de prueba. El payload consiste en una secuencia de enteros desde 0 hasta 254.

El payload consiste en 350 enteros que dan un total de 1400 bytes. Seguidamente, en el bucle infinito, se envían 4900 tramas y finalmente se envía una trama con -1 para indicar que se han terminado de enviar todas las tramas. Este proceso se hace de manera infinita.

En la siguiente sección se analizarán los resultados obtenidos.

3.6.4. Resultados

En este apartado se analizarán los resultados obtenidos con la utilización del middleware de comunicaciones icec.

En la Figura 3.21 se pueden ver los resultados en cuanto a ancho de banda obtenidos mediante la utilización del middleware de comunicaciones. Este ancho de banda se ha obtenido con el monitor del sistema operativo del cliente para no sobrecargar el servidor en FreeRTOS.

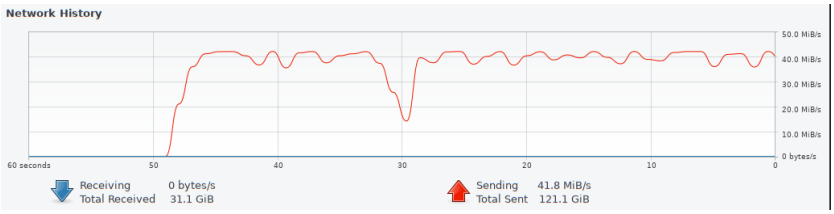

Figura 3.21: 40 MiB sostenidos mediante middleware de comunicaciones

En los primeros experimentos se comprobó que a la hora de añadir información de desarrollo por la conexión serie el ancho de banda disminuía. Se cree que esto se debe a la prioridad de las tareas, no obstante, por falta de tiempo no se ha podido determinar con exactitud el origen de la sobrecarga. Se deja como **trabajo futuro** el estudio de esta sobrecarga.

Por otro lado, los valores situados en el código del cliente (Listado 24) como 4900 iteraciones han sido situados bajo prueba y error.

En una primera aproximación se implementó el bucle infinito sin un número fijo de iteraciones. Eso hacía que el cliente, ejecutado en un ordenador de última generación, enviará a la máxima velocidad posible para un procesador de 3.5 Ghz. Lógicamente, el procesador con capacidad suficiente para saturar el enlace de 1 Gbps realizaba demasiada "presión" sobre el servidor implementado en un ARM con una frecuencia **máxima** de 666 Mhz.

Tras observar que los bufferes de tipo *NetworkBuffer* disminuían de forma constante. Se llegó a la conclusión de que el problema radicaba en que el cliente enviaba demasiado rápido las tramas como para que el cliente pudiera procesarlas.

Para confirmar la teoría se incrementó el número de bufferes reservados para el sistema de comunicaciones. Esta modificación se puede realizar desde el configurador del BSP. Tras realizar la modificación el tiempo hasta recibir el aviso de inexistencia de NetworkBuffer.

Tras calcular mediante los timer del código del servidor se llega a la conclusión de que el tiempo de procesamiento por trama era de unos 18 μs.

Existen muchas variables que influyen a la hora de calcular este tiempo

de procesamiento. Una de las modificaciones que se realizaron para mejorar el rendimiento fue en la recepción de la secuencia.

En el código proporcionado por Xilinx el buffer pasa directamente desde la memoria del stack de comunicaciones al código del usuario mediante una estructura del tipo *pbuf*, de este modo se evitan todo tipo de copias. este modo de actuar es una estratégia *zero-copy*. En el código Listado 25, se puede ver el código comentado y el código con "solución" aportada.

Cuando se recorre el array que determina la secuencia se realiza una copia a un buffer fuera del "stack" de icec. Esto se debe a que no se aporta ningún mecanismo para asegurar que dicha región no pueda ser eliminada por el driver de comunicaciones. Por tanto, como solución intermedia para mediar la **eficiencia real** del middleware se ha limitado la copia a un único entero, el entero que será utilizado para comprobar los números de secuencia. Si no se hace esta modificación no se estarían realizando las mismas medidas que con iperf.

```
1  static void
2  Example_payload_readFromInputStream(Example_payloadPtr self,
3                                      Ice_InputStreamPtr is) {
4      int i = 0;
5
6      trace();
7
8      Ptr_check(is);
9
10     Ice_InputStream_readSize(is, &(self->size), true);
11
12     //for (i=0; i<self->size; i++) {
13     for (i=0; i<1; i++) {
14         Ice_InputStream_readInt(is, self->items + i);
15     }
16 }
```

Listado 25: Mejora en la eficiencia

Es **muy importante** tener en cuenta que esta modificación no se podría realizar en un entorno real ya que todo lo que está a continuación del primer byte se considera basura dado que no se realiza la copia.

Como **resultados** por tanto podemos llegar a la conclusión de que no se ha conseguido perder ningún tipo de paquete (a no ser que el canal se sature) y que por lo tanto **UDP** resulta un protocolo viable para la comunicación en este tipo de aplicación. Se llega a la conclusión, también, de que el uso del middleware resulta de mucha utilidad para resolver el problema de las comunicaciones de manera rápida y eficiente. Aunque el middleware todavía está en una fase temprana los resultados obtenidos son esperanzadores.

Capítulo 4

Conclusiones

4.1. Conclusiones

En este apartado se tratará de agregar las conclusiones / resultados obtenidos en cada uno de los experimentos anteriores.

La **conclusión principal** a la que se ha llegado tras la realización de este trabajo es que existen muy pocas fuentes con la información suficiente para que se pueda replicar las medidas reportadas. Xilinx proporciona una nota de aplicación (XAP1026) donde reporta diferentes medidas. Sin embargo, a la hora de replicar los resultados se puede ver que estos únicamente son una muestra del "poder bruto" del SoC y no representan un caso de uso real. Muestra de ello puede ser la modificación que se ha tenido que realizar para igual las medidas realizadas por el programa utilizado por Xilinx (iperf) (Listado 25).

Otra de las conclusiones a las que se ha llegado y en cierto modo fruto de la conclusión anterior es que las medidas realizadas por iperf **no representan** un caso de uso real aunque representan el caso de uso más exigente, al requerirse un sistema que esté el 100 % del tiempo realizando envíos a la velocidad del medio, cosa que no ocurre normalmente.

Al inicio de este trabajo se había propuesto realizar la adaptación del

middleware de comunicaciones icec mediante el protocolo de comunicaciones TCP. Este protocolo de comunicaciones permite asegurar la recuperación frente a fallos, congestión de la línea, recepción fuera de orden... Sin embargo, tras realizar los experimentos mostrados en este trabajo, se **ha llegado a la conclusión** de que resulta más interesante mejorar el endpoint UDP y analizar estratégias de recuperación de errores sencillas con el objetivo de cubrir los casos donde pueda haber una sobrecarga y se pierdan paquetes.

Quizá, la **conclusión** más importante de este trabajo desde el punto de vista del grupo de investigación al que pertenece el autor es que se ha demostrado la viabilidad del middleware de comunicaciones Zeroc-ice y su adaptación icec para aplicaciones de medición y con requisitos exigentes.

Con este trabajo se sientan las bases para líneas de investigación futuras en el ámbito de aplicación del middleware icec para la plataforma Zynq y, en general, dispositivos basados en lógica reconfigurable. En la Sección 4.2 se verán algunas de estas líneas.

4.2. Trabajo futuro

Por último, tras realizar todos los experimentos y el desarrollo principal del proyecto mediante el "porting" del middleware de comunicaciones a la plataforma Zynq, en esta sección se propone una serie de líneas de trabajo futuras en base al presente trabajo y que, en un futuro próximo serán exploradas por el grupo de investigación ARCO al que el autor pertenece:

- Normalizar las mediciones mediante un entorno más controlado: en este proyecto, desde el primer momento se han mantenido fijas, tanto las versiones software como el hardware y sistemas de medición utilizados. No obstante, algunos de estos sistemas, tal y como se ha podido ver en los resultados del desarrollo propuesto, pueden afectar al desarrollo del propio experimento. Con el objetivo de caracterizar de una manera más exacta y mejorar por tanto, el conocimiento sobre los aspectos que influyen en la comunicación. Se propone especificar un entorno de prueba y elección de instrumentos de medición no invasiva para obtener medidas más fiables.

- Añadir los elementos reconfigurables de la FPGA: en este trabajo se ha supuesto que los datos viajan desde el ordenador hacia la FPGA. Este es uno de los flujos posibles y, aunque no afecta a los resultados, sería interesante realizar las mismas medidas desde la otra dirección. Aprovechando estas medidas, se propone la incorporación de un elemento generador de datos desde la lógica reconfigurable que permita simular el modo de operación de un instrumento de medición. Así, en vez de realizar una medida sintética, se tendrán resultados que cubran la cadena de comunicación completa.

- Portar el middleware a la lógica reconfigurable: para mejorar el rendimiento del sistema, se propone trasladar la lógica del middleware de comunicaciones del microprocesador ARM a la lógica reconfigurable de la FPGA. Esto permitirá dos cosas principalmente:

    1. Mejorar la latencia del sistema: al eliminar el microprocesador de la cadena de comunicaciones la latencia se reducirá y se podrá trabajar prácticamente a la velocidad del cable. Esto se debe a que

se está trabajando directamente en hardware y no en un software secuencial.

2. **Nuevo paradigma de hardware distribuido**: cada elemento hardware podrá ser expuesto a la red mediante la definición de una interfaz. De este modo, el diseñador hardware únicamente definirá una interfaz que compilará mediante un compilador slice (slice2hdl) y le permitirá conectarse a una interfaz AXI para recibir los datos del controlador MAC y enrutar los objetos a cada uno de los modulos hardware. Por otro lado, el diseñador distribuirá la interfaz a los programadores del frontend que, mediante slice2c, slice2js, slice2java o cualquier otro compilador podrán "hablar" con el hardware sin necesidad de tener que considerar ningún aspecto referente a la comunicación subyacente.
3. Soporte zero-copy: estudiar la viabilidad de dar soporte al middleware de comunicaciones para la comunicación mediante Zero-copy.

Bibliografía

[1] *ADC121C021/ADC121C021Q/ADC121C027 I2C-Compatible, 12-Bit Analog-to-Digital Converter with Alert Function*. ADC121C027. Texas Instruments. Mar. de 2013.

[2] Maurice J. Bach. *The design of the UNIX operating system*. Prentice-Hall software series. Englewood Cliffs, N.J: Prentice-Hall, 1986. ISBN: 9780132017992.

[3] Stephen Brown y Jonathan Rose. "Architecture of FPGAs and CPLDs: A tutorial". En: *IEEE Design and Test of Computers* 13 (2002).

[4] A. Cicuttin y col. "A Block-Based Open Source Approach for a Reconfigurable Virtual Instrumentation Platform Using FPGA Technology". En: *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*. Sep. de 2006, págs. 1-8. DOI: 10.1109/RECONF.2006.307751.

[5] Maurizio Di Paolo Emilio. *Data acquisition systems: from fundamentals to applied design*. New York: Springer, 2013. ISBN: 9781461442134.

[6] FreeRTOS. *FreeRTOS Official webpage*. 2018. URL: https://www.freertos.org/implementation/a00005.html (visitado 04-06-2018).

[7] Google Inc. *Gihub, Flatbuffer*. 2018. URL: https://google.github.io/flatbuffers/ (visitado 04-06-2018).

[8] Howard W. Johnson y Martin Graham. *High-speed digital design: a handbook of black magic*. Englewood Cliffs, N.J: Prentice Hall, 1993. ISBN: 9780133957242.

[9] Hans Werner Meuer y col. *The TOP500: History, Trends, and Future Directions in High Performance Computing*. 1st. Chapman & Hall/CRC, 2014.

[10] P Moreira, A Marchioro y K Kloukinas. "The GBT, a Proposed Architecture for Multi-Gb/s Data Transmission in High Energy Physics". En: May 2014 (2007). URL: https://cds.cern.ch/record/1091474/files/p332.pdf.

[11] Ramón Pallás Areny. *Adquisición y distribución de señales*. Barcelona: Marcombo, 2008. ISBN: 9788426709189.

[12] *Parallel Analog-to-digital converter*. TLV571. Texas Instruments. Feb. de 2000.

[13] Red Pitaya. *Github Red Pitaya*. 2018. URL: https://github.com/RedPitaya/RedPitaya/blob/master/api2/src/rp_api.c (visitado 04-06-2018).

[14] Red Pitaya. *Github Red Pitaya HTML code*. 2018. URL: https://github.com/RedPitaya/RedPitaya/blob/master/apps-free/spectrum/index.html (visitado 04-06-2018).

[15] Fabio Sauli. "The gas electron multiplier (GEM): Operating principles and applications". En: *Nucl. Instrum. Meth.* A805 (2016), págs. 2-24. DOI: 10.1016/j.nima.2015.07.060.

[16] Sosa Sosa y Víctor J. "MIDDLEWARE : Arquitectura para Aplicaciones Distribuidas Contenido Middleware : Definición Middleware : Más definiciones ..." En: (2014), págs. 1-21. URL: http://www.tamps.cinvestav.mx/%7B~%7Dvjsosa/clases/sd/Middleware%7B%5C_%7DRecorrido.pdf.

[17] W Richard Stevens, Bill Fenner y Andrew M Rudoff. {*UNIX*} *network programming.* {*Vol*}. *1: ...* 3. ed., 8. Addison-{Wesley} professional computing series. Boston, Mass.: Addison-Wesley, 2008. ISBN: 9780131411555.

[18] Andrew S Tanenbaum y Elisa Nez Ramos. *Redes de computadoras*. Spanish. OCLC: 931948790. Mxico, D.F., Mxico: Prentice-Hall Hispanoamericana., 2012. ISBN: 9786073208178.

[19] *The Standard for On-Chip Communication*. AMBA. ARM.

[20] Wikipedia. *Sistema operativo — Wikipedia, La enciclopedia libre*. [Internet; descargado 4-junio-2018]. 2018. URL: https://es.wikipedia.org/w/index.php?title=Sistema_operativo&oldid=108456368.

[21] ZeroC. *ZeroC, Basic Data Encoding*. 2018. URL: https://doc.zeroc.com/ice/3.6/the-ice-protocol/data-encoding/basic-data-encoding#id-.BasicDataEncodingv3.6-2 (visitado 04-06-2018).

[22] ZeroC. *ZeroC, Icegrid*. 2018. URL: https://doc.zeroc.com/ice/3.7/client-server-features/locators (visitado 04-06-2018).

[23] ZeroC. *ZeroC, Licensing*. 2018. URL: https://zeroc.com/licensing (visitado 04-06-2018).