



GRADO EN INGENIERÍA TELEMÁTICA

Curso Académico 2017/2018

Trabajo Fin de Grado

OPTIMIZACIÓN DE APLICACIONES EN SISTEMAS ROBÓTICOS DE TIEMPO REAL

Autor : Lorena Bajo Rebollo

Tutor : Francisco Martín Rico

Trabajo Fin de Grado

OPTIMIZACIÓN DE APLICACIONES EN SISTEMAS ROBÓTICOS DE TIEMPO REAL

Autor : Lorena Bajo Rebollo

Tutor : Francisco Martín Rico

La defensa del presente Proyecto Fin de Carrera se realizó el día de
de 2018, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2018

Agradecimientos

En primer lugar, quiero dar las gracias a mis padres, por darme siempre el cariño y el apoyo necesarios, por enseñarme que la vida no siempre es un camino de rosas pero que con todo se puede, por darme la oportunidad de formarme y simplemente por ser mis padres.

A mi hermana Nuria, por estar siempre a mi lado y por tantas horas de diversión.

A mis amigos de toda la vida, por mostrarme que los problemas son menos graves con un kebab en la mano. Y a todos los nuevos amigos que he hecho estos años en el grado, porque sin vosotros la vida no compilaría igual.

A todo el equipo de Erle Robotics por acogerme como una más de su familia durante estos seis meses de exilio en Vitoria y proporcionarme todos los recursos necesarios para la realización de este proyecto.

A mi tutor Paco, por animarme a emprender nuevas aventuras.

Y en general gracias a todas las personas que me han acompañado durante este largo camino y que me han ayudado a ser la persona que soy hoy en día.

AGRADECIMIENTOS

Resumen

En la actualidad, las nuevas tecnologías y la robótica están ganando terreno en el mundo de la industria y poco a poco se están abriendo camino en nuestros hogares para resolver de manera eficiente problemas que pueden aparecer en la vida cotidiana.

El objetivo de este proyecto es proporcionar un entorno más determinista a las herramientas más utilizadas dentro del mundo de la robótica. Herramientas y bibliotecas como de las que hace uso *ROS2*, que para poder crear aplicaciones en varios equipos tiene que apoyarse en un *middleware* de tipo publicación/suscripción que se encargue de conectar sus componentes entre sí.

Para proporcionar determinismo se utilizarán primitivas de tiempo real para la priorización de tareas y un mecanismo eficiente para la reserva de memoria. Además también se instalará un kernel de tiempo real del que se aprovechará su estructura modular para mover código del espacio de usuario al espacio del kernel e intentar conseguir una mejora en los tiempos de respuesta de las aplicaciones.

Summary

At present, the new technologies and robotics are gaining territory in the industry world and bit by bit they are making their own way into our homes to solve problems that may appear in daily life.

The objective of this project is to provide a more deterministic environment to the most used tools in the robotics world. Tools and libraries like the used by ROS2, that to create applications for different systems, ROS2 must be supported by a middleware of publisher/subscriber type that connects its components with each other.

To provide determinism, real-time primitives will be used to prioritize tasks and an efficient mechanism for memory reservation. In addition, a real-time kernel will also be installed, from which its modular structure will be used to move code from the user space to the kernel space and try to achieve an improvement in the response times of the applications.

SUMMARY

Índice general

1. Introducción	1
1.1. UNIX	2
1.2. Kernel de Linux	2
1.3. Kernel de tiempo real	4
1.3.1. Sistemas operativos basados en Linux en tiempo real	5
1.4. ROS y ROS2	6
1.5. Estructura de la memoria	7
2. Objetivos, requisitos y metodología	9
2.1. Requisitos	9
2.2. Objetivos	10
2.3. Metodología	10
3. Entorno y herramientas	13
3.1. Middleware	13
3.1.1. DDS	13
3.1.2. Fast-RTPS	14
3.2. Asio	15
3.3. Profundizando en ROS y ROS2	16
4. Desarrollo del trabajo	19
4.1. Diferenciación entre user space y kernel space	19
4.2. Cambiando de kernel	20
4.3. Desarrollo de módulos del kernel	21
4.4. Convirtiendo aplicaciones a aplicaciones Real Time	22

4.5. Cálculo de latencias en Fast-RTPS	25
4.6. Creación de un módulo del kernel para Fast-RTPS	26
4.6.1. Investigación previa	26
4.6.2. Complicaciones en la creación del módulo	32
5. Experimentación	37
5.1. Moviéndonos entre el user space y el kernel space	37
5.1.1. Escenario 1	37
5.1.2. Escenario 2	38
5.1.3. Escenario 3	39
5.1.4. Gráfica comparativa entre escenarios	39
5.2. Experimentación con el LatencyTest de Fast-RTPS	41
5.2.1. Escenario 1	41
5.2.2. Escenario 2	42
5.2.3. Comparativa gráfica entre escenarios	42
6. Conclusiones	45
6.1. Consecución de objetivos	45
6.2. Competencias utilizadas y adquiridas	46
6.3. Trabajos futuros	46
Bibliografía	49

Índice de figuras

1.1. Robot en el mundo real.	1
1.2. Estructura del kernel del Linux.	3
1.3. Priorización de tareas. Preempting.	4
1.4. Logotipo de <i>ROS</i>	6
3.1. <i>DDS</i> y <i>ROS2</i> . [2]	14
3.2. Relación entre <i>ROS2</i> y <i>Fast-RTPS</i> . [7]	15
3.3. Módulos de <i>Asio</i> . [13]	16
3.4. Interacción entre nodos <i>publisher</i> y <i>subscriber</i> de <i>ROS</i>	17
3.5. Estructura de <i>ROS2</i> . [22]	18
4.1. <i>User space</i> y <i>kernel space</i> en Linux. [17]	20
4.2. Salida de la ejecución del comando <code>uname -a</code> después del cambio de kernel.	21
4.3. Clases de mayor relevancia en <i>Fast-RTPS</i>	27
4.4. Intercambio de datos en <i>Fast-RTPS</i> . [10]	27
4.5. Creación de un fichero que sirve como dispositivo.	31
4.6. Representación gráfica de la idea.	32
4.7. Captura de <i>Wireshark</i> de la fase de descubrimiento de <i>Fast-RTPS</i>	33
4.8. Información necesaria para la fase de descubrimiento.	34
5.1. Escenario 1.	38
5.2. Escenario 2.	38
5.3. Escenario 3.	39
5.4. Gráfica comparativa entre escenarios.	40
5.5. Comparativa de la media en modo <i>best effort</i> entre un kernel RT y no-RT.	43

5.6. Comparativa de máximos en modo <i>best effort</i> entre un kernel RT y no-RT. . . .	43
--	----

Índice de cuadros

4.1. Funciones asociadas entre el <i>user space</i> y el <i>kernel space</i> . [4]	22
5.1. Latencias de los escenarios 1, 2 y 3 (s).	40
5.2. Media, mínimo y máximo de los escenarios 1, 2 y 3 (s).	40
5.3. Latencias del escenario 1 (μ s).	41
5.4. Latencias del escenario 2 (μ s).	42

ÍNDICE DE CUADROS

Capítulo 1

Introducción

La Robótica nació hace muchos años con las promesas de alcanzar metas que hasta el momento solo habían sido posibles en la gran pantalla. El mundo de la informática en constante crecimiento unido a las nuevas disciplinas de la Inteligencia Artificial crean un entorno propicio para la creación de nuevos robots con una gran capacidad de aprendizaje y adaptación al entorno.

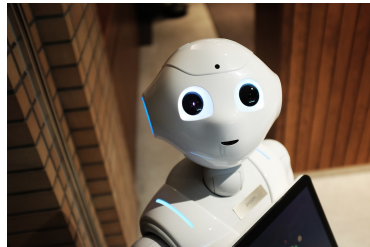


Figura 1.1: Robot en el mundo real.

Los sistemas en tiempo real proporcionarán una respuesta a los estímulos del entorno más determinista y harán que la adaptación sea más fácil, porque ¿quién quiere un robot con un tiempo de respuesta aleatorio? la respuesta es sencilla: nadie.

De esta pregunta nace la idea para el desarrollo de este proyecto de investigación, en el que el objetivo será hacer más determinista el *middleware* del que hacen uso la mayoría de robots del mercado actual. Para llevar a cabo este objetivo se precisará trabajar sobre un sistema de tiempo real, y para ello será necesario que el sistema esté dotado con las mayores herramientas y recursos de tiempo real de las que se disponga, comenzando por el kernel y terminando por *ROS2*, que es la plataforma de robot estándar que permite al usuario configurar las opciones de

comunicación según sus necesidades.

El sistema operativo elegido para el desarrollo de este proyecto será Linux, ya que ofrece una mayor interoperabilidad entre kernel y aplicación.

A continuación se ofrecerá una visión más detallada de los sistemas en tiempo real para el entorno de trabajo elegido.

1.1. UNIX

UNIX es un sistema operativo desarrollado en los años 60 por Ken Thompson y Dennis Ritchie caracterizado principalmente por: [1]

- Alta portabilidad.
- Multitarea.
- Sistema multiusuario.
- Interconexión y comunicación de procesos que además pueden trabajar en serie.
- Los ficheros en UNIX están organizados de una manera jerárquica y pueden ser usados con diferente fin, como es el caso de los ficheros para almacenamiento de datos o los utilizados como dispositivos. [8]

Dos ejemplos de sistemas UNIX son Linux y MacOS X (FreeBSD).

1.2. Kernel de Linux

El kernel o núcleo de Linux se podría definir como el corazón del sistema operativo basado en UNIX. El kernel de Linux es monolítico, es decir, que todos los procesos y servicios principales se encuentran alojados en él. Un problema de estos kernels es que necesitan tener compilados todos los controladores para conseguir que el software y el hardware puedan comunicarse entre sí, lo que hará que el volumen del kernel aumente a medida que trate de dar soporte a un mayor número de dispositivos. Para solucionar este problema, Linux utiliza los módulos, que carga y descarga en memoria según su necesidad. Se puede hacer una separación de estos módulos o subsistemas en los siguientes: [15]

- **Gestión y planificador de procesos:** Qué tareas, en qué orden y con qué prioridad se van a ejecutar.
- **Gestión de memoria:** Uso, asignación y compartición de la memoria de la manera más optimizada posible utilizando mecanismos de gestión de la memoria como el sistema de paginación¹ y memoria virtual². [12]
- **Intercomunicación de procesos y sincronización:** Mecanismos de comunicación entre procesos.
- **Gestión de entrada/salida:** Para el control y gestión de periféricos.
- **Interfaz de red:** Conjunto de operaciones de envío y recepción de paquetes que son iguales para los diferentes tipos de hardware.

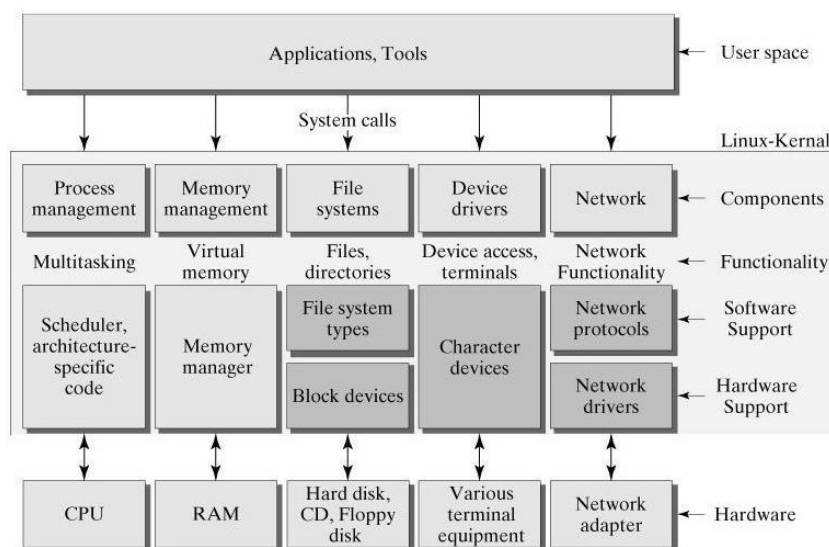


Figura 1.2: Estructura del kernel del Linux.

El kernel de Linux está programado en *C* casi en su totalidad, que es un lenguaje de programación desarrollado por Dennis Ritchie a principios de los 70, está orientado a la implementación de sistemas operativos y permite la programación a bajo nivel, aunque también tiene estructuras características de lenguajes de alto nivel. [30]

¹Paginación: mover las páginas de los espacios de direcciones virtuales al disco duro, libera memoria RAM.

²Se denomina memoria virtual a archivar las aplicaciones y datos menos usados en una parte del disco duro, simulando una ampliación de la memoria RAM.

1.3. Kernel de tiempo real

Como se ha descrito anteriormente, el kernel de Linux es multitarea y por lo tanto puede ejecutar varios procesos a la vez.

Más concretamente, lo que sucede es que estos procesos se encuentran en una cola y el procesador los va ejecutando durante un tiempo³, si este tiempo se agota el procesador interrumpe esta tarea y ejecuta la siguiente. A esto se le llama cambio de contexto.

Cada tarea tiene su propia prioridad, lo que hace que las tareas o procesos con prioridades más altas obtengan un quantum mayor.

Obviamente esto tiene un coste, ya que detener y cargar otra tarea implica sobrecargar el procesador. Para eliminar esta sobrecarga y obtener una ejecución más rápida la solución sería sencilla: ejecutar las tareas por completo. Sin embargo, esto haría que se perdiese el poder atender a varias peticiones de forma simultánea.

Una de las características de los kernels de tiempo real⁴ [29] es que permiten la interrupción de tareas en más sitios que los kernels no-RT. Además, los procesos que tienen una mayor prioridad no tienen ningún miramiento a la hora de hacerse con el control de la CPU⁵, de manera que los demás procesos serán ralentizadas y esperarán el tiempo que sea necesario hasta que se complete este proceso. Esto se traduce en que aunque con el kernel RT no haya una mejora considerable de los tiempos de respuesta, es más determinista que un kernel no-RT. [14]

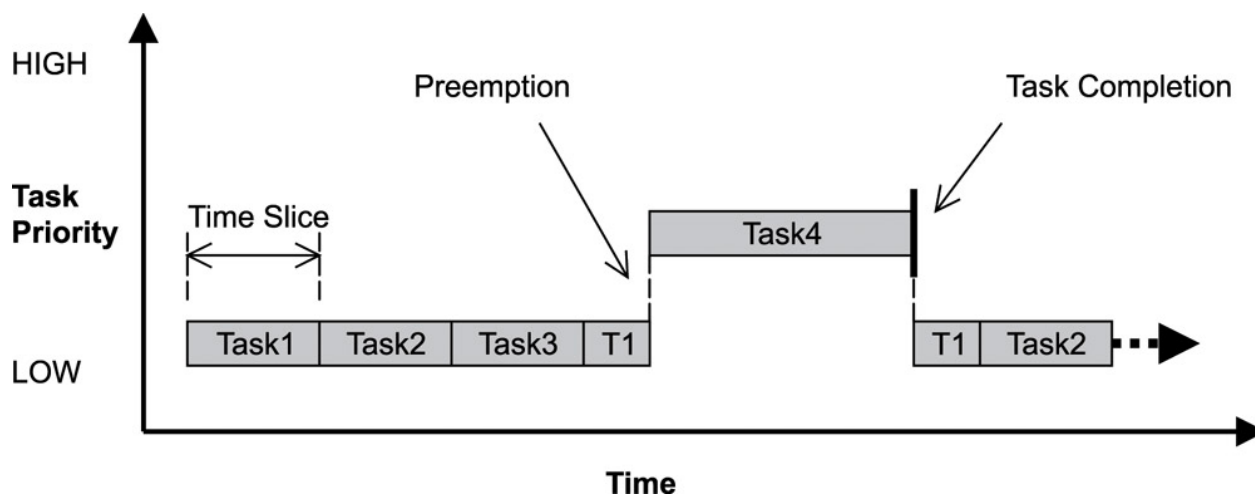


Figura 1.3: Priorización de tareas. Preempting.

³Este periodo de tiempo se denomina quantum o time slice.

⁴De ahora en adelante kernel RT.

⁵A esto se le denomina preempting. [14]

¿Cuándo es interesante tener un kernel RT? Cuando necesitamos que una tarea de alta prioridad no se interrumpa en ningún momento. Un ejemplo, si tuviésemos un dron en el aire, nos interesaría que la tarea de "volar" tuviese prioridad sobre la tarea de "fotografiar el paisaje", para así evitar accidentes.

1.3.1. Sistemas operativos basados en Linux en tiempo real

- **Xenomai.** Es un pequeño kernel adicional que se introduce en el sistema para que trabaje con el kernel de Linux y así aumentar la predicibilidad del sistema. Su principal característica es que permite que las tareas en tiempo real trabajen en el espacio de usuario. Más concretamente, Xenomai introduce dos dominios para la ejecución de tareas, un dominio primario, en el que se encuentran los hilos o *threads* en tiempo real y un dominio secundario que está controlado por el kernel de Linux. La ventaja de esta separación es que los procesos de tiempo real no usan el mismo espacio de memoria que las otras tareas, lo que evita que se produzca interbloqueo entre tareas. Sin embargo, esta separación en dominios hace que se produzca un incremento en las latencias del sistema. [3]
- **PREEMPT_RT.** Es un parche para el kernel de Linux que hace el sistema más determinista. Una de sus principales características es sustituir los *spinlocks*⁶ del código del kernel por *mutexes*⁷ expulsables con herencia de prioridad⁸. Esto hace que la latencia máxima que puede tener una tarea se minimice. Otro punto importante es que este parche también proporciona interrupciones en hilos, de manera que convierte los manejadores de interrupción en hilos de kernel expulsable, cuando habitualmente el planificador de Linux los gestionaba en contexto de proceso. [3]

Para la realización de este trabajo se hará uso del parche de *PREEMPT_RT*, ya que a día

⁶*Spinlock*: primitiva de sincronización de exclusión mutua que espera en un bucle comprobando si se ha cumplido una condición hasta que pueda adquirir un bloqueo. [11]

⁷*Mutex*: primitiva de sincronización que se utiliza para proteger datos compartidos por más de un subproceso.[11]

⁸La herencia de prioridad se utiliza para resolver el problema de la inversión de prioridad (una tarea retrasa la ejecución de otra de prioridad mayor porque está bloqueado esperando un recurso que tiene una tarea con una prioridad más baja), se encarga de que la tarea que tenga un semáforo herede su prioridad para no ser interrumpida y reducir el tiempo en el que se encuentra en la sección crítica. [18]

de hoy Xenomai es un sistema que se está empezando a encontrar obsoleto y *PREEMPT_RT* proporciona mayor determinismo y es bastante estable.

1.4. ROS y ROS2

El *Robot Operating System (ROS)* es un sistema meta-operativo de código abierto para robots mantenido por la *Open Source Robotics Foundation (OSRF)*. Comenzó a desarrollarse en 2007 por el *STAIR (Stanford Artificial Intelligence)* y a partir del 2008 continuó con su crecimiento en el instituto de investigación de Willow Garage. Y finalmente, su desarrollo continúa en la *Open Source Robotics Foundation (OSRF)*, siendo ahora código libre que puede ser usado por cualquiera y además aceptando contribuciones externas.

Proporciona un conjunto de herramientas y bibliotecas de software para ayudar a crear aplicaciones en diferentes equipos. *ROS* está compuesto por varios nodos independientes entre sí, para comunicarse entre ellos utilizan un modelo de mensajería de tipo publicación / suscripción. [21]



Figura 1.4: Logotipo de *ROS*.

Aunque en sus inicios *ROS* fue creado para proyectos de investigación y desarrollo, estos son algunos de los casos para los que más se demanda la utilización de *ROS* hoy en día:

- Creación de equipos de varios robots.
- Plataformas embebidas y profundamente integradas, basadas tanto en microcontrolador o microprocesador.
- Sistemas en tiempo real.
- Redes no ideales.

- Entornos de producción y trabajo en cadena.
- Patrones para la construcción y estructuración de sistemas. [24]

Todas estas nuevas demandas han contribuido al desarrollo de *ROS2*, que proporcionará un nivel de abstracción mayor, la mejora de las ASPIs de usuario, y además será compatible con *ROS*.

1.5. Estructura de la memoria

La estructura que se va a seguir en este proyecto va a ser la siguiente:

- Capítulo 2. Se presentan los requisitos y objetivos perseguidos con este proyecto así como la metodología empleada para ello.
- Capítulo 3. En este capítulo se hará una descripción detallada de las herramientas que existen y se han utilizado para la realización del trabajo.
- Capítulo 4. Explicación detallada de los pasos que se han seguido en el desarrollo del proyecto.
- Capítulo 5. Exposición de los diferentes experimentos realizados y resultados obtenidos así como una breve explicación de los mismos.
- Capítulo 6. Finalización de la memoria con una breve reflexión sobre los resultados obtenidos y los posibles pasos futuros para una mejora del proyecto.

Capítulo 2

Objetivos, requisitos y metodología

En el presente capítulo se expondrá el objetivo principal del proyecto y a su vez se hará una división del mismo en objetivos específicos o subobjetivos más simples. Junto a ello, también se detallan los requisitos y herramientas necesarias, así como la metodología empleada para el desarrollo del trabajo.

2.1. Requisitos

Como se ha descrito en el capítulo introductorio, las nuevas demandas en el mundo de la robótica harán que aumente también la necesidad de desarrollar no sólo robots más rápidos y potentes si no también más deterministas, por este motivo y para la realización de este proyecto se tienen los siguientes requisitos:

- Sistema operativo Linux. Este sistema operativo será el elegido para el desarrollo de este proyecto, ya que ofrece una mayor interoperabilidad entre kernel y aplicación.
- Kernel *Real Time*. Para lograr un sistema más determinista habrá que convertir el kernel de nuestro sistema en un kernel RT, para esto será necesaria la aplicación de una versión compatible de los parches *PREEMPT_RT* con nuestro kernel.
- Raspberry Pi. El uso de estas placas será crucial, ya que al ser unos novatos en la fase de creación de módulos del kernel, es posible que cometamos errores y al insertar el módulo en el kernel el sistema puede quedar bloqueado, de manera que será más cómodo trabajar

con una Raspberry Pi ya que en caso de bloqueo sólo habría que reiniciar la placa y realizar los cambios oportunos en el código.

- Conocimientos de C y C++. El desarrollo de las aplicaciones y programas de este trabajo serán realizados en los lenguajes de programación C y C++.
- *ROS2*. El experimento final de este proyecto consistirá en un intento por hacer más determinista el *middleware* por defecto de *ROS2*, y para ello debemos tenerlo instalado en nuestro sistema.

2.2. Objetivos

El objetivo principal de este proyecto es demostrar que con el uso de herramientas y primitivas de tiempo real se puede obtener un sistema más determinista y además conseguir una mejora en el rendimiento del sistema trabajando a nivel de kernel.

Para abordar el objetivo principal, este proyecto ha sido dividido en unos objetivos más concretos:

- Demostrar que un kernel de tiempo real es más determinista que un kernel normal.
- Usar herramientas de tiempo real en aplicaciones del espacio de usuario.
- Usar herramientas de tiempo real en el espacio de kernel.
- Demostrar que hay una mejora del rendimiento al portar código de una aplicación del espacio de usuario al espacio del kernel.
- Reproducir el punto anterior con *Fast-RTPS*.

2.3. Metodología

Para el desarrollo de este proyecto se ha llevado a cabo un plan de trabajo basado en los siguientes puntos:

- Familiarización y manejo de las herramientas software. Será necesario un tiempo de adaptación a las herramientas y programas que se van a utilizar en el desarrollo del proyecto,

antes de proceder a la realización del mismo. Herramientas tales como un kernel de tiempo real, *ROS2* y otras herramientas que se describirán más adelante en esta memoria.

- Descomposición del problema principal en subproblemas más simples. La descomposición del trabajo en pequeñas subtareas harán que la comprensión y el desarrollo del mismo sea más sencillo.
- Desarrollo secuencial. El desarrollo de forma secuencial de este proyecto será clave ya que hasta que no se cumpla una tarea, no se llevará a cabo la siguiente. Esto es importante porque el entendimiento y consecución de una tarea será necesaria para el desarrollo de la siguiente.
- Fase de experimentación. A medida que se vayan completando los objetivos descritos en el apartado anterior, se irán realizando varios experimentos para comprobar el correcto funcionamiento de los mismos.
- Representación de resultados. Una buena medida será también utilizar Matlab u otra herramienta con la que poder representar gráficamente los resultados obtenidos en la fase de experimentación.

Capítulo 3

Entorno y herramientas

A día de hoy, existen varias plataformas en las que es posible desplegar todas las herramientas y recursos necesarios para la programación y el desarrollo de aplicaciones robóticas, en este capítulo se hará una descripción más detallada de las herramientas y recursos utilizados para este proyecto.

3.1. Middleware

El *Middleware* es un software que se sitúa entre las capas inferiores y las capas de aplicaciones para así conectar el sistema operativo y las aplicaciones que se ejecutan en él. [27] Se encarga de abstraer las funcionalidades más complejas de las capas inferiores, como por ejemplo, de la red o del sistema operativo.

Más concretamente, en *ROS2* se busca la compatibilidad con varias implementaciones *DDS*. Para conseguirlo, se introduce una interfaz *middleware* que comunica la biblioteca de *ROS2* con las muchas implementaciones de *DDS*. [23]

3.1.1. DDS

El *Data Distribution Service (DDS)* es una capa de *middleware* de tipo publicación / suscripción, que ha sido seleccionado como capa del sistema de comunicación de *ROS2* entre otros. Utiliza el *Interface Description Language (IDL)*¹ tal como lo define el *Object Management*

¹*IDL* es un lenguaje de programación que permite la comunicación con otro programa escrito en un lenguaje de programación desconocido o no.

Group (OMG) para la definición y serialización de mensajes. [22]

Como características principales, *DDS* provee al usuario de descubrimiento, definición y serialización de mensaje y transporte de tipo publicación / suscripción.

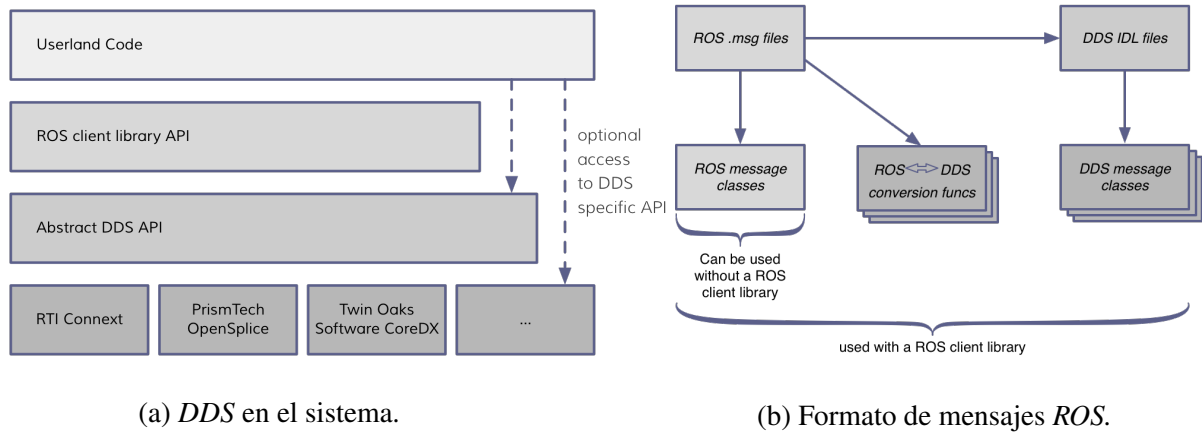


Figura 3.1: *DDS* y *ROS2*. [2]

Algunas de las aplicaciones para las que más se demanda el uso de *DDS* hoy en día son barcos de guerra, sistemas financieros, espaciales y de vuelo. [16]

Las implementaciones de *DDS* más elegidas para utilizar junto con *ROS2* son *OpenSplice*, *Connex-RTI*, *Free-RTPS* y *Fast-RTPS*. [20]

3.1.2. Fast-RTPS

Fast-RTPS es el middleware por defecto que ha adoptado *ROS2*. [10] Como aspectos a resaltar:

- *Fast-RTPS* es una implementación del protocolo *RTPS*², diseñado para poder ejecutar transmisiones *multicast*, *reliable* y *best effort* no orientadas a conexión como *UDP*³. [7]
- Permite que los dispositivos más simples capaces de implementar un subconjunto del protocolo puedan participar en la red e intercambiar información. Esto se traduce en una gran modularidad.

²Real Time Publish Subscribe Protocol.

³User Datagram Protocol (*UDP*) es un protocolo de transporte no orientado a conexión, en otras palabras, no hará uso de ninguna de sus herramientas y recursos para garantizar que los mensajes lleguen a su destino.

- Capacidad de ampliación y mejora del protocolo con nuevos servicios sin perder interoperabilidad ni compatibilidad. Además esta ampliación también permite que los sistemas puedan escalar a redes más grandes.
- Los *headers* o encabezados de *Fast-RTPS* tienen un tamaño bastante grande ya que es un protocolo con una gran versatilidad.
- Un módulo de descubrimiento muy sencillo de configurar para el usuario, ya que únicamente es necesario que los nodos sean compatibles en el *topic* y *QoS*⁴.
- *Fast-RTPS* está programado en **C++**, un lenguaje de programación creado por Bjarne Stroustrup a principios de los 80. Destaca principalmente por contar con la mayoría de las ventajas que proporciona *C*, pero orientado a objetos entre otras cosas. [30]

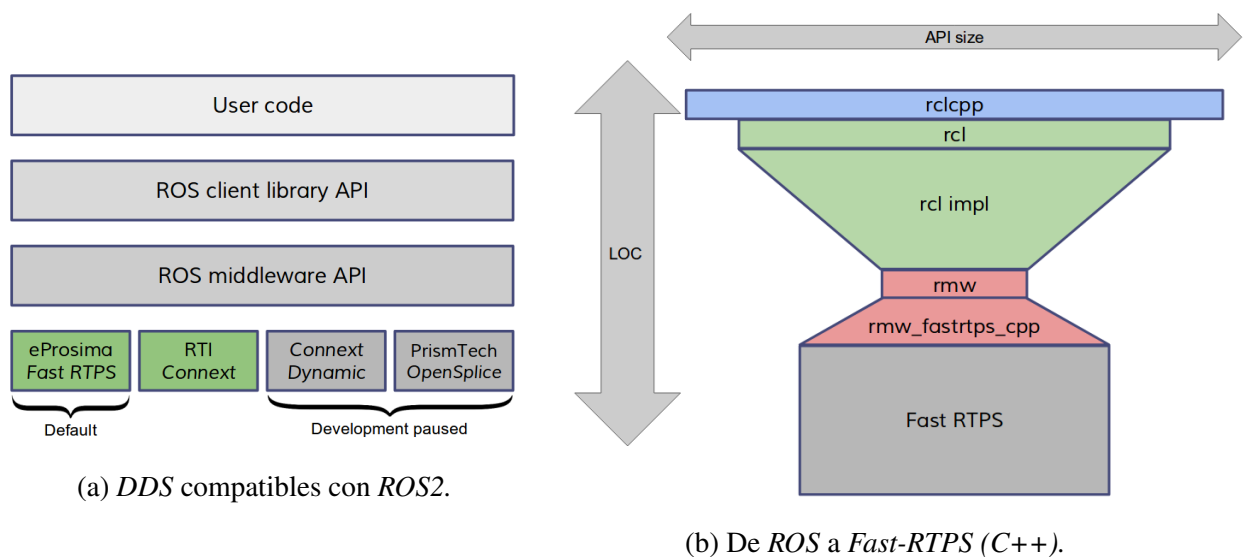


Figura 3.2: Relación entre ROS2 y Fast-RTPS. [7]

3.2. Asio

Asio es una biblioteca multiplataforma de **C++** que proporciona herramientas para la creación de redes **C++** y para la administración de operaciones de entrada / salida que pueden tardar un tiempo considerable en ejecutarse sin necesidad de hacer uso de sincronización con hilos y bloqueo explícito. [13]

⁴Quality of Service o calidad de servicio.

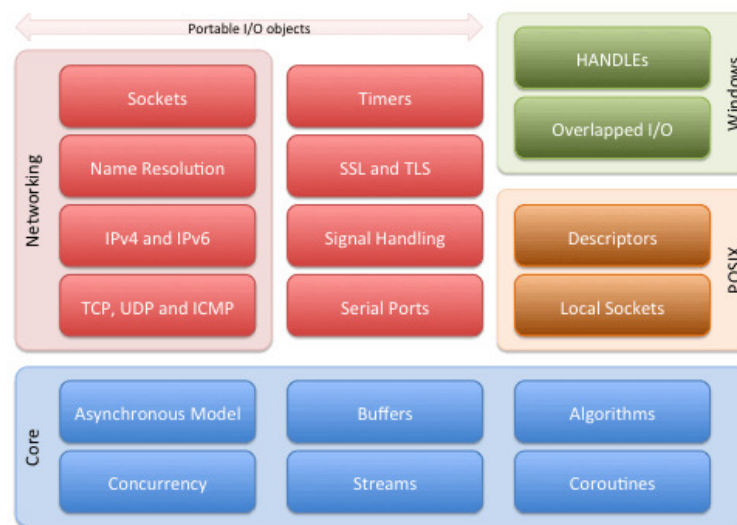


Figura 3.3: Módulos de *Asio*. [13]

3.3. Profundizando en ROS y ROS2

Como se describió en el primer capítulo, *ROS* está compuesto por nodos que se comunican entre sí, pero además de estos nodos, es necesario hablar de otros elementos importantes que también participan en *ROS*:

- **Nodos.** Son ficheros ejecutables de *ROS* que se pueden comunicar entre sí. Generalmente cada nodo se encarga de realizar una función específica. Además las múltiples librerías cliente que provee *ROS* permite la comunicación entre nodos escritos en diferentes lenguajes de programación. Estos nodos pueden publicar (*publisher*) o suscribirse (*subscriber*) a un topic.
- **Topics.** Canales que usan los nodos para intercambiar mensajes. Esta transmisión está basada en el protocolo TCP/IP por defecto, pero también es posible realizar esta comunicación en UDP. Cada *topic* tiene una estructura determinada y puede haber más de un *publisher* y *subscriber* por *topic*, creando así una comunicación asíncrona y unidireccional entre los diferentes nodos.
- **Mensajes.** Estructura de datos determinada que los nodos utilizan para comunicarse a través de *topics*. La especificación de estos mensajes se encuentra en un fichero de un

directorio dentro del paquete que se esté utilizando, y en este directorio se pueden añadir ficheros con nuevos tipos de mensajes con las estructuras que se desee.

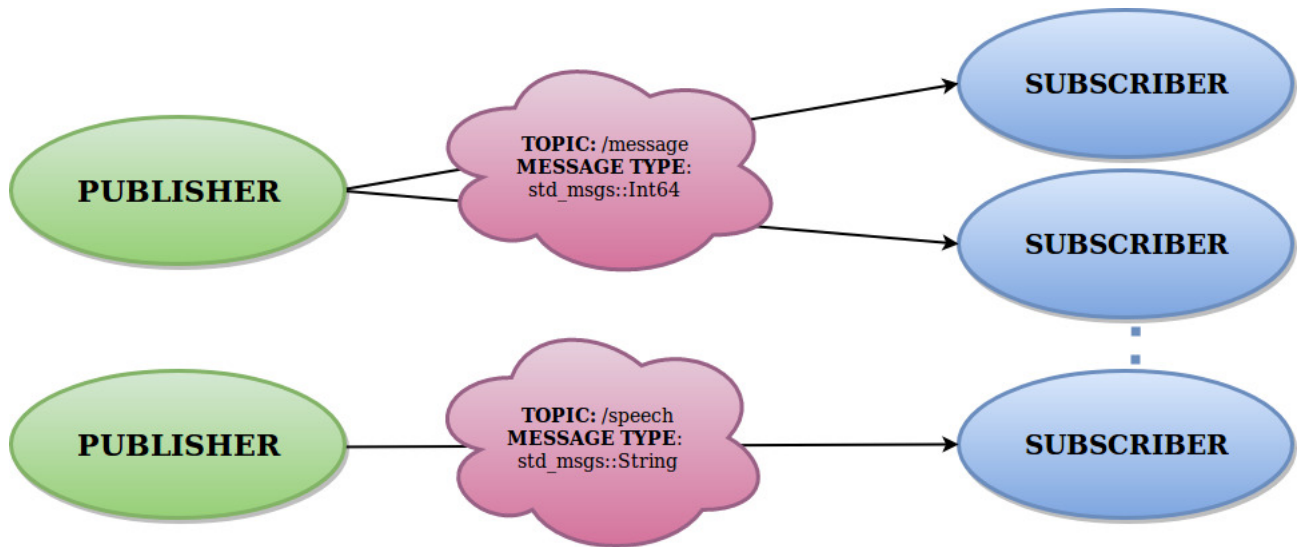
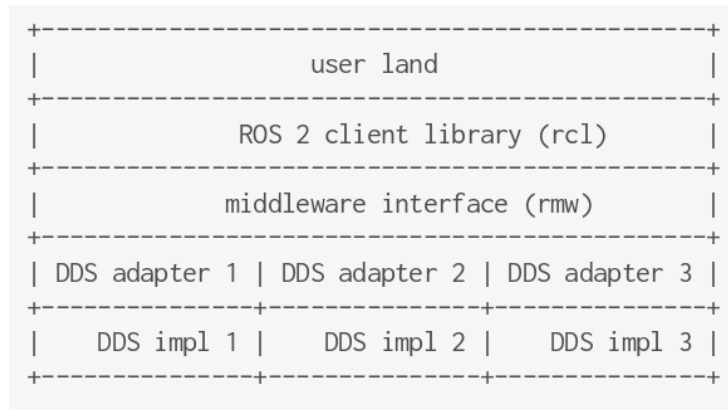


Figura 3.4: Interacción entre nodos *publisher* y *subscriber* de ROS.

Una característica diferenciadora de *ROS* respecto a *ROS2* es la existencia de un *master node*, que se podría describir como el nodo principal de *ROS* que da servicio de registro para los demás nodos del sistema. Ejecutando `roscore`⁵ en una terminal Linux, sería la forma de lanzar este *master node*.

Con *ROS2* este nodo master desaparece, reemplazando toda su fase de descubrimiento por la de *DDS*. Ahora *ROS* tendrá que acceder a la API de *DDS* para obtener toda la información de los *topics*, nodos y sus respectivas conexiones. Una gran ventaja de esto es que al incorporar *DDS* a *ROS2* se crea un mayor nivel de abstracción y todas las definiciones de mensajes y otros campos quedan ocultos, evitando que el usuario final tenga que llamar directamente al *DDS*, aunque sí que admite que el usuario defina nuevos metadatos para esta fase de descubrimiento. Además, al no depender únicamente de un nodo master para la fase de descubrimiento, la tolerancia a fallos será mayor, ya que el sistema de descubrimiento se vuelve distribuido.

⁵`roscore` es un conjunto de nodos y programas que se encarga de lanzar todo lo necesario para cumplir los requisitos previos para que el sistema *ROS* pueda funcionar.

Figura 3.5: Estructura de *ROS2*. [22]

Sin duda el mayor avance que trae *ROS2* consigo es la introducción de *DDS* en su implementación, que será el encargado de añadir a *ROS2* nuevas funcionalidades y ventajas. Entre estas ventajas, puede que la más importante de todas ellas sea la configuración de *QoS* para configurar el comportamiento de las comunicaciones entre nodos, haciendo posible que la utilización del sistema *ROS* sea lo más personalizada posible para las necesidades de cada usuario (todas estas funcionalidades y ventajas irán variando dependiendo del *DDS* utilizado).

La configuración de *QoS*, unido a la nueva fase descubrimiento que hace uso de transmisiones *multicast* harán que *ROS2* cumpla con las nuevas demandas de uso para sistemas distribuidos.

Capítulo 4

Desarrollo del trabajo

En este capítulo se dará una explicación más detallada y cronológica de los pasos seguidos en el desarrollo de este proyecto.

4.1. Diferenciación entre user space y kernel space

Para el desarrollo de este proyecto es muy importante saber en todo momento en qué espacio de trabajo nos estamos moviendo, en Linux existe la siguiente división de espacios:

- **User space o espacio de usuario:** entorno de ejecución de programas que utiliza el usuario final y que utiliza las llamadas al sistema como forma de comunicación con el kernel. Un ejemplo, la *shell*.
- **Kernel space o espacio de kernel:** entorno donde reside el código del kernel con acceso ilimitado a la memoria del sistema y que tiene como función principal la administración de procesos y crear un puente de comunicación entre el *user space* y el hardware de la máquina.

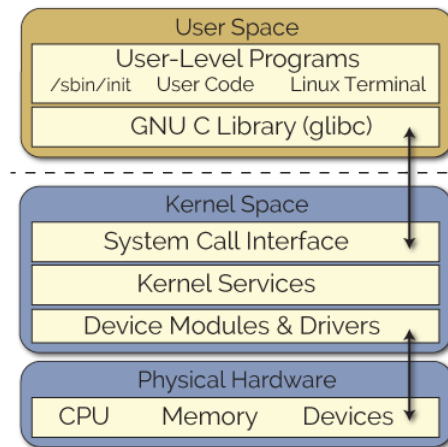


Figura 4.1: *User space y kernel space* en Linux. [17]

4.2. Cambiando de kernel

Antes de comenzar con el desarrollo de programas, lo primero que tenemos que hacer es convertir nuestro kernel a un kernel RT. Estos son los pasos a seguir:

1. Descargar el nuevo kernel.

```
git clone -b linux-4.9.y git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

2. Descargar los parches de *PREEMPT_RT*.

```
wget https://www.kernel.org/pub/linux/kernel/projects/rt/4.9/patch..
```

3. Aplicar los parches y el nuevo kernel.

```
xz -d patch...
cd linux
cat ../patch-... | patch -p1
cp /boot/config-$(uname -r)-generic .config
```

4. Deshabilitar la información de depuración.

```
scripts/config --disable DEBUG_INFO
```

5. Al ejecutar las siguientes líneas aparecerá una pantalla azul en la que hay que seleccionar *PREEMPT_RT* entre las diferentes opciones de kernel.

```
make menuconfig  
make -j `getconf _NPROCESSORS_ONLN` deb-pkg LOCALVERSION=-custom  
make scripts
```

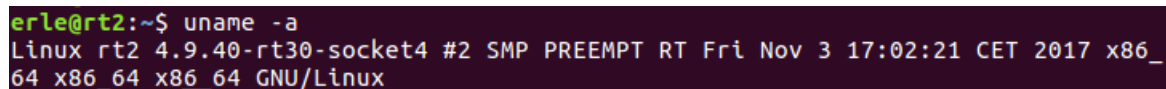
6. Instalar paquetes debian y reiniciar el equipo para que se apliquen todos los cambios realizados.

```
sudo dpkg -i *.deb  
reboot
```

A la hora de descargar nuestro nuevo kernel y los parches es importante tener en cuenta que estamos descargando versiones compatibles, en caso contrario, obtendremos errores de compilación que no nos permitirán hacer uso del nuevo kernel.

Por este motivo para este proyecto se descargará e instalará un nuevo kernel junto con los parches, ya que el kernel actual de nuestro equipo no es compatible con el parche del que se quiere hacer uso.

Para comprobar que todo ha ido bien ejecutamos el siguiente comando¹:



```
erle@rt2:~$ uname -a  
Linux rt2 4.9.40-rt30-socket4 #2 SMP PREEMPT RT Fri Nov 3 17:02:21 CET 2017 x86_64 x86_64 x86_64 GNU/Linux
```

Figura 4.2: Salida de la ejecución del comando `uname -a` después del cambio de kernel.

De la *figura 4.2* nos tiene que llamar la atención la nueva columna con valor *PREEMPT RT* que indica que el parche se ha aplicado correctamente.

4.3. Desarrollo de módulos del kernel

Como ya se comentó anteriormente, en sistemas UNIX los ficheros se pueden usar como dispositivos, lo que permite al programador una interacción más sencilla con el hardware ya que esta comunicación hardware-aplicación se produce a través de funciones de lectura y escritura de ficheros. [4]

En el *cuadro 4.1* se muestran algunas de las funciones más utilizadas para la programación en el *kernel space* y sus funciones asociadas en el espacio de usuario.

¹`uname -a` muestra por pantalla información del sistema.

Eventos	Funciones de usuario	Funciones del kernel
Cargar módulo	insmod	init_module
Abrir dispositivo	fopen	file_operations: open
Leer dispositivo	fread	file_operations: read
Escribir dispositivo	fwrite	file_operations: write
Cerrar dispositivo	fclose	file_operations: release
Quitar módulo	rmmod	cleanup_module

Cuadro 4.1: Funciones asociadas entre el *user space* y el *kernel space*. [4]

La programación de módulos del kernel se realiza en C, pero hay que tener en cuenta que al programar en el espacio del kernel parte del código y funciones a las que estamos acostumbrados se reemplazan por otras funciones equivalentes del kernel. [26]

Además, para compilar nuestro módulo como un módulo externo hay que crear un archivo *Makefile* que se parecerá a la siguiente caja de código:

```
obj-m += module.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Después de ver que la compilación ha ido bien, podemos utilizar el comando `insmod nombre-del-módulo` y comprobar que ha sido cargado correctamente con el comando `lsmod`. [5]

4.4. Convirtiendo aplicaciones a aplicaciones Real Time

Para que un sistema se comporte como un sistema de tiempo real no basta con tener un kernel RT, también es necesario el manejo de varias áreas, como la prioridad de una aplicación

o el bloqueo de memoria. La API de POSIX² constituye la base de todas estas herramientas.

- **Planificador y prioridad de procesos:** como ya se explicó en apartados anteriores, la prioridad de un proceso es muy importante ya que determinará su velocidad de procesamiento y si tiene preferencia sobre otros procesos. [31]

```
int set_thread_priority(pthread_t thread, int prio, int sched)
{
    int policy;
    struct sched_param sch;

    /* Devuelve la política de planificación y los parámetros del
       thread que se pasa como argumento */
    if(pthread_getschedparam(thread, &policy, &sch) != 0){
        printf("pthread_getschedparam_error\n");
        return -1;
    }
    /* Establece la nueva prioridad y las políticas de planificación
       para el thread que se pasa como argumento */
    sch.sched_priority = prio;
    if(pthread_setschedparam(thread, sched, &sch) != 0){
        printf("pthread_setschedparam_error\n");
        return -1;
    }
    return 0;
}
```

- **Bloqueo de memoria:** consiste en asignar memoria física antes de que el programa comience a trabajar³, de este modo no habrá costes de tiempo adicionales por fallos de página⁴. [32, 18]

²Portable Operating System Interface Unix (POSIX) [11] es un conjunto de estándares de llamadas al sistema operativo definido por el Institute of Electrical and Electronics Engineers (IEEE) y su principal objetivo es que una misma aplicación funcione en diferentes plataformas.

³Para saber la cantidad de memoria que va a necesitar un programa ejecutar en una terminal *ps -leyf*, buscar el nombre del programa y ver la octava columna.

⁴Un fallo de página se define como el error que se produce cuando un programa intenta acceder a un bloque de memoria pero este bloque no se encuentra almacenado ni en la memoria física ni en la memoria RAM. Dicho

```

int configure_malloc_behavior(void)
{
    /* Bloquea todas las páginas actuales y futuras */
    if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0){
        printf("mlockall(MCL_CURRENT_|_MCL_FUTURE)_error\n");
        return -1;
    }
    /* Ajustar la configuración para las reservas de memoria. */
    if (mallopt(M_TRIM_THRESHOLD, -1) != 1){
        printf("mallopt(M_TRIM_THRESHOLD, -1)_error\n");
        return -1;
    }
    /* Especifica el número máximo de asignaciones que puede hacer
        mmap a la vez. */
    if (mallopt(MMAP_MAX, 0) != 1){
        printf("mallopt(MMAP_MAX, 0)_error\n");
        return -1;
    }
    return 0;
}

```

```

void reserve_process_memory(int size)
{
    int i;
    char *buffer = (char*)malloc(size);

    /* Mapea cada página de este trozo de memoria en la RAM. */
    for (i = 0; i < size; i += sysconf(_SC_PAGESIZE)) {
        /* Cada escritura en este buffer genera un fallo de página.
            Una vez que se tiene la página, se bloquea en memoria
            y no se devuelve al sistema. */
        buffer[i] = 0;
    }

    /* La memoria que se asigna más arriba se bloquea para

```

de otro modo, es una notificación al sistema operativo para que aloje los datos en la memoria virtual para luego transmitirlos desde el dispositivo de almacenamiento a la memoria RAM.


```
este proceso. Todas las llamadas malloc() y new() provienen
de la memoria reservada arriba.
Con free() y delete() no se elimina el bloqueo. De esta
manera nunca se van a producir fallos de página mientras
se ejecuta el programa. */
free(buffer);
}
```

4.5. Cálculo de latencias en Fast-RTPS

De ahora en adelante, para obtener resultados de latencias de *Fast-RTPS* utilizaremos el *LatencyTest* que provee el propio desarrollador de *Fast-RTPS*⁵. [9]

Entendiendo un poco el código del *LatencyTest*:

1. **Fase de descubrimiento.** El programa espera a que los *publishers* y *subscribers* se descubran entre ellos. En los resultados que se muestran en el *apartado 5.2* del capítulo de experimentación el descubrimiento se hace entre un *publisher* y un *subscriber* aunque esta aplicación permite que haya más de uno.
2. **Ejecución y número de iteraciones.** El *publisher* envía una muestra de cada tamaño de paquete, y entre cada envío de paquete el programa espera 100 μ s. El *LatencyTest* permite también que el usuario determine el número de muestras que se van a enviar. Cuando se produce un envío sucede lo siguiente:
 - a) El *publisher* le envía el comando *READY* al *subscriber* para indicarle que está preparado y espera hasta que el *subscriber* responda.
 - b) Una vez está todo preparado, el *publisher* toma el tiempo antes de que se produzca el envío del mensaje, envía el mensaje y espera la respuesta del *subscriber* con un *timeout*⁶ de 1 segundo. Si se supera este tiempo de espera el paquete se da por perdido y se pasa al siguiente envío. Cuando el paquete llega se vuelve a tomar el tiempo y se calcula el tiempo final.

⁵<https://github.com/eProsima/Fast-RTPS/tree/master/test/performance>

⁶Tiempo de espera.

- c) Cuando se finaliza el envío de un tamaño de paquete el *publisher* envía el comando *STOP* y si quedan más tamaños por enviar comienza este proceso de nuevo.

3. **Reliability u opciones de confiabilidad.** Como ya se comentó anteriormente, *Fast-RTPS* tiene varias opciones de configuración de *QoS*.

- a) *Reliable*. Para asegurar que todos los paquetes enviados llegarán a su destino.
- b) *Best effort*. No se usará ningún recurso para monitorizar y garantizar que los paquetes enviados sean recibidos.

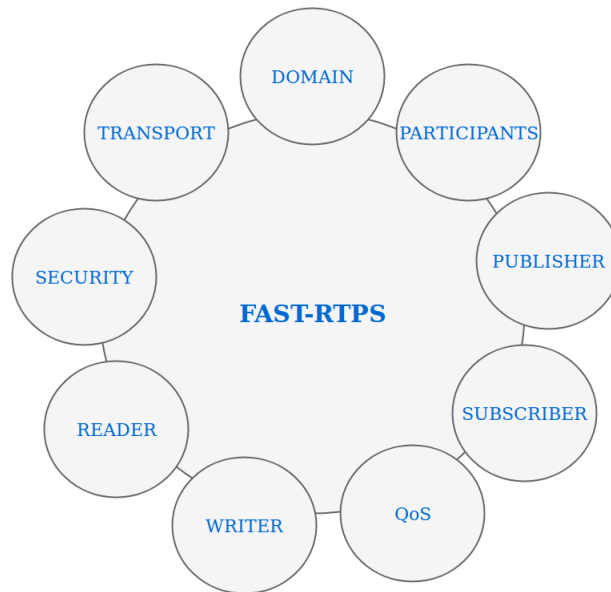
4.6. Creación de un módulo del kernel para Fast-RTPS

Como parte central de este proyecto se aplicarán todas las tecnologías anteriormente descritas en una tecnología ya desarrollada como *Fast-RTPS*.

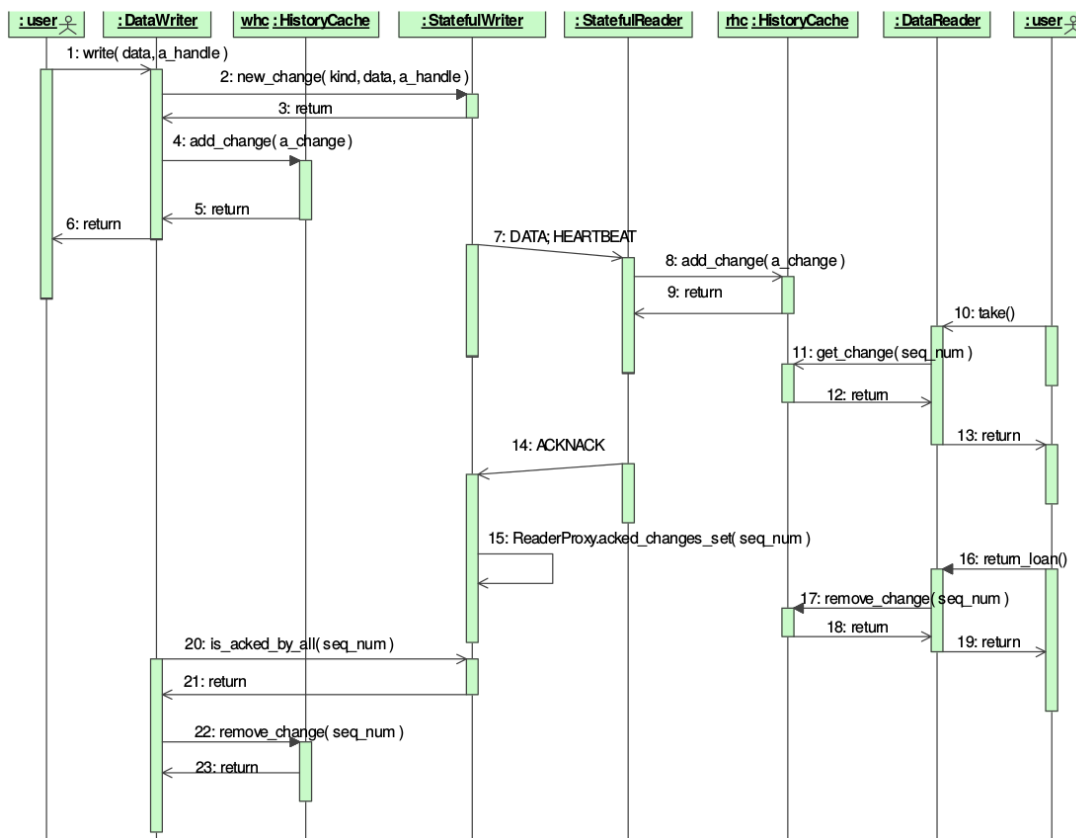
4.6.1. Investigación previa

Portar código de *Fast-RTPS* al kernel trae consecuencias. La primera, *Fast-RTPS* está programado en C++, lo que quiere decir que hay que hacer un cambio de lenguaje de programación a C, esto implica la eliminación del uso de clases y más herramientas de alto nivel que C++ provee. Por este motivo, hay que decidir muy bien que porción del código de *Fast-RTPS* se va a trasladar al kernel, ya que estaremos limitados en cuanto a herramientas de programación como funciones, librerías... etc.

También hay que recalcar que el motivo por el que se va a realizar este módulo del kernel es para una posible mejora de las latencias (el determinismo en el sistema lo introducirá el kernel y las primitivas de tiempo real de las que se haga uso), por lo tanto, de entre todas las diferentes secciones de las que se compone *Fast-RTPS* hay que elegir una en la que se pueda hacer una mejora notable en el rendimiento. Esto implica que debe ser un código que no haga uso de herramientas de alto nivel, ni de muchas otras clases, de cuantas más clases y recursos haga uso, más grande y complicado será el módulo a crear.

Figura 4.3: Clases de mayor relevancia en *Fast-RTPS*.

El **módulo de transporte** será el elegido, ya que aunque *a priori* parece el más complejo, es el que trabaja a más bajo nivel (dentro del espacio de usuario).

Figura 4.4: Intercambio de datos en *Fast-RTPS*. [10]

Hay que detenerse un momento en el paso 7 y 8 de la *figura 4.4*, donde entra en juego el módulo de transporte y su clase principal *UDPv4Transport*⁷, que tiene como objetivos principales:

- Realizar una lista con todas las interfaces permitidas, si se deja esta lista en blanco, quiere decir que todas las interfaces son válidas. (Esta lista se encarga de especificarla el usuario).
- Abrir un canal para la salida de datos, al pasar un localizador se abre un socket por cada interfaz de red en un puerto dado. Un canal corresponde a un puerto y una dirección de destino.
- Abrir un canal para la entrada de datos, al pasar un localizador se abre un socket que escucha por un puerto determinado en cada interfaz de la lista de interfaces disponibles, y se unirá al canal *multicast* que especifica la dirección del localizador.
- Enviar y recibir la información por el canal adecuado.

```
bool UDPv4Transport::SendThroughSocket(const octet* sendBuffer,
                                       uint32_t sendBufferSize, const Locator_t& remoteLocator,
#ifdef ASIO_HAS_MOVE
                                       asio::ip::udp::socket& socket)
#else
                                       std::shared_ptr<asio::ip::udp::socket> socket)
#endif
{
    asio::ip::address_v4::bytes_type remoteAddress;
    memcpy(&remoteAddress, &remoteLocator.address[12], sizeof(
        remoteAddress));
    auto destinationEndpoint = ip::udp::endpoint(asio::ip::address_v4(
        remoteAddress), static_cast<uint16_t>(remoteLocator.port));
    size_t bytesSent = 0;
#ifdef ASIO_HAS_MOVE
    logInfo(RTPS_MSG_OUT, "UDpv4:_" << sendBufferSize << "_bytes_TO_"
            << destinationEndpoint
```

⁷<https://github.com/eProxima/Fast-RTPS/blob/master/src/cpp/transport/UDpv4Transport.cpp>

```

        << "_FROM_" << socket.local_endpoint());
#else
    logInfo(RTPS_MSG_OUT, "UDPv4:_" << sendBufferSize << "_bytes_TO_"
        endpoint:_" << destinationEndpoint
        << "_FROM_" << socket->local_endpoint());
#endif

    try
    {
#if defined(ASIO_HAS_MOVE)
        bytesSent = socket.send_to(asio::buffer(sendBuffer,
            sendBufferSize), destinationEndpoint);
#else
        bytesSent = socket->send_to(asio::buffer(sendBuffer,
            sendBufferSize), destinationEndpoint);
#endif
    }
    catch (const std::exception& error)
    {
        logWarning(RTPS_MSG_OUT, "Error:_" << error.what());
        return false;
    }
    (void) bytesSent;
    logInfo (RTPS_MSG_OUT, "SENT_" << bytesSent);
    return true;
}

```

Esta última función en concreto es la última en la cadena de envío del mensaje. Aunque finalmente, después de seguir todo el flujo del programa, observamos que el envío final de los datos los realiza la librería *Asio* con la llamada al sistema `sendmsg`.

Como ya se comentó anteriormente, *Asio* proporciona herramientas para la creación de redes C++, pero además también incluye una interfaz de sockets de bajo nivel basada en la *BSD Sockets API*⁸ de la que hará uso *Fast-RTPS*.

⁸Parte de la *POSIX API* dedicada a la comunicaciones entre procesos, más concretamente en comunicaciones en Internet, aunque también sirve para procesos del mismo sistema. [28]

La siguiente función de *Asio*⁹ es la que se encarga de enviar un datagrama a un *endpoint* específico, y también es la que nos interesa mover al kernel:

```
signed_size_type sendto(socket_type s, const buf* bufs, size_t count,
    int flags, const socket_addr_type* addr, std::size_t addrlen,
    asio::error_code& ec)
{
    clear_last_error();
#if defined(ASIO_WINDOWS) || defined(__CYGWIN__)
    // Send the data.
    DWORD send_buf_count = static_cast<DWORD>(count);
    DWORD bytes_transferred = 0;
    int result = error_wrapper(::WSASendTo(s, const_cast<buf*>(bufs),
        send_buf_count, &bytes_transferred, flags, addr,
        static_cast<int>(addrlen), 0, 0), ec);
    if (ec.value() == ERROR_NETNAME_DELETED)
        ec = asio::error::connection_reset;
    else if (ec.value() == ERROR_PORT_UNREACHABLE)
        ec = asio::error::connection_refused;
    if (result != 0)
        return socket_error_retval;
    ec = asio::error_code();
    return bytes_transferred;
#else // defined(ASIO_WINDOWS) || defined(__CYGWIN__)
    msghdr msg = msghdr();
    init_msghdr_msg_name(msg.msg_name, addr);
    msg.msg_namelen = static_cast<int>(addrlen);
    msg.msg_iov = const_cast<buf*>(bufs);
    msg.msg_iovlen = static_cast<int>(count);
#if defined(__linux__)
    flags |= MSG_NOSIGNAL;
#endif // defined(__linux__)
    signed_size_type result = error_wrapper(::sendmsg(s, &msg, flags), ec);
    if (result >= 0)
```

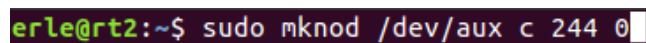
⁹https://github.com/chriskohlhoff/asio/blob/230c0d2ae035c5ce1292233fcab03cea0d341264/asio/include/asio/detail/impl/socket_ops.ipp

```
    ec = asio::error_code();  
    return result;  
#endif // defined(ASIO_WINDOWS) || defined(__CYGWIN__)  
}
```

El objetivo es reemplazar la siguiente línea por un módulo del kernel. Y que en lugar de utilizar la llamada al sistema `sendmsg`, automáticamente salte un módulo que realice la misma función desde el espacio del kernel.

```
signed_size_type result = error_wrapper(::sendmsg(s, &msg, flags), ec);
```

Para que el nuevo módulo del kernel funcione, va a necesitar un intermediario entre el espacio de usuario y el kernel que le pase toda la información necesaria para que el mensaje pueda llegar a su destino. Para ello hay que ligar un fichero al módulo¹⁰. [25, 6, 4]



```
erle@rt2:~$ sudo mknod /dev/aux c 244 0
```

Figura 4.5: Creación de un fichero que sirve como dispositivo.

En la *figura 4.6* se puede ver una representación más detallada de la idea:

¹⁰El 244 es el *major number* o número mayor que utiliza el kernel para relacionar el fichero con su driver.

El 0 es el *minor number* o número menor que se utiliza para uso interno del dispositivo.

Y la c significa que se trata de un dispositivo de tipo *char*.

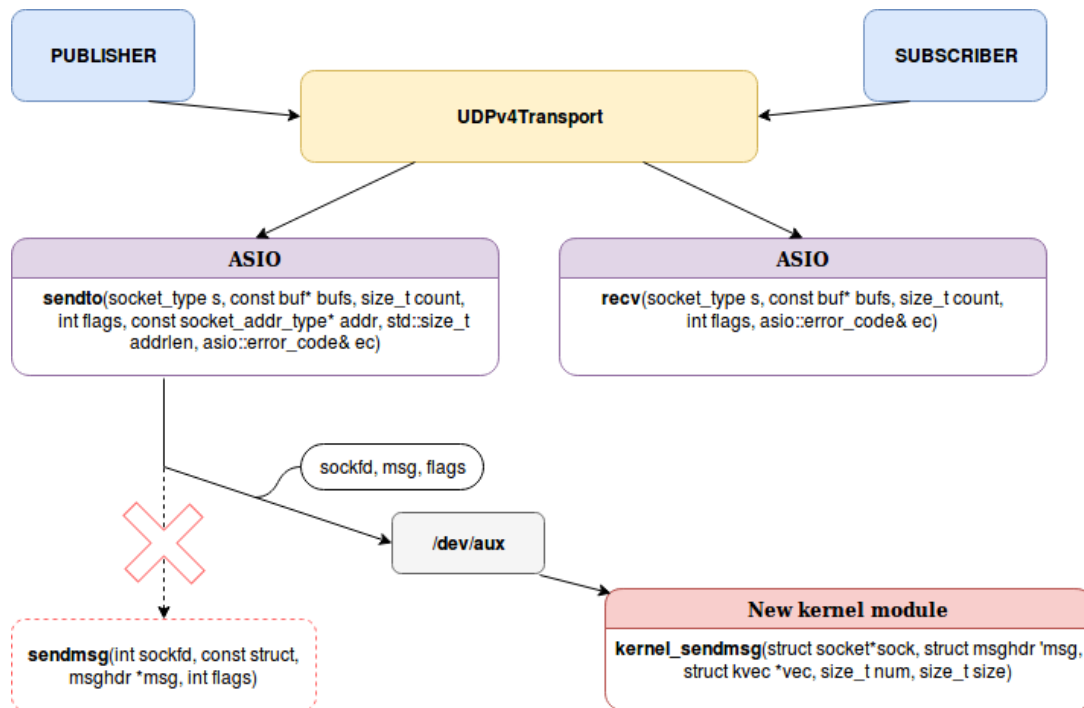


Figura 4.6: Representación gráfica de la idea.

También hay que cambiar un poco el código de *Asio*. De ahora en adelante cada vez que *Asio* llame a la función `sendto` en lugar de hacer el envío del mensaje, escribirá toda la información necesaria para el envío en `/dev/aux` y automáticamente cada vez que se realice una escritura en el fichero, el módulo lo leerá, lo guardará en un buffer y separará la información en diferentes campos.

4.6.2. Complicaciones en la creación del módulo

En esta sección se describen las complicaciones y aspectos a tener en cuenta a la hora de crear el nuevo módulo del kernel para *Fast-RTPS*.

1. Este nuevo módulo también va a utilizar las herramientas para aplicaciones de tiempo real, pero hay un problema, no se puede reservar una cantidad de memoria fija, ya que dependiendo del programa en *ROS2* que ejecuta varias capas más arriba, se hará uso de una mayor o menor cantidad de memoria y recursos.

Por lo tanto, no se utilizará el bloqueo de memoria, pero sí la asignación de prioridades, pero de una manera diferente, ya que ahora se estará priorizando el trabajo de los

threads del kernel (kthreads) y no de los *threads* de *POSIX (pthreads)* puesto que estamos moviéndonos en el espacio de kernel. Hay que ser cuidadoso a la hora de asignar una prioridad, ya que no hay que ceder una prioridad más alta que la que tienen otros procesos críticos del sistema¹¹.

2. Quizá sería más sencillo trabajar directamente sobre la clase *UDIPv4Transport* para así eliminar otras complicaciones, como es el caso de algunos tipos de datos propios de *Asio*, por ejemplo los *buffers*. Pero *Asio* no se puede reemplazar fácilmente, ya que además de ser el encargado de crear los *sockets* responsables del envío y recepción de datos, también es el responsable de la sincronización en *Fast-RTPS*.

3. Fase de descubrimiento de *Fast-RTPS*. [19]

- *Fast-RTPS* envía mensajes por todas las interfaces que se encuentren disponibles. Para ello utiliza direcciones *multicast*.
- 224.0.0.22. Esta dirección es usada por el protocolo *IGMP*¹², permitiendo que los *endpoints* se unan a un canal de multidifusión.

5	0.489651	192.168.1.93	224.0.0.22	IGMPv3	60 Membership Report / Join group 239.255.0.1 for any sources
6	4.743759	192.168.1.129	239.255.0.1	RTPS	262 INFO_TS, DATA(p)
7	4.744726	192.168.1.93	192.168.1.129	RTPS	266 INFO_TS, DATA(p)

Figura 4.7: Captura de *Wireshark* de la fase de descubrimiento de *Fast-RTPS*.

- 239.255.0.1. La configuración por defecto de *Fast-RTPS* incluyen el siguiente localizador *multicast*:

$$\text{DefaultMulticastLocator} = \{\text{LOCATOR_KIND_UDIPv4}, "239.255.0.1", \text{PB} + \text{DG} * \text{domainId} + d0.\}^{13}$$

¹¹Una manera sencilla de ver las prioridades de otros procesos es ejecutando *top* o *htop* en una terminal

¹²El *Internet Group Management Protocol* o protocolo *IGMP* está fuertemente ligado al protocolo IP, su función principal es la creación de direcciones *multicast*, es decir, que al producirse el envío de datos a una dirección IP se alcancen varias máquinas de una red o subred. También se utiliza para añadir miembros a una red y establecer la comunicación entre dichos miembros.

¹³*Port Base Number(PB)=7400, Domain Id Gain(DG)=250, additional offsets(d0)=0.*

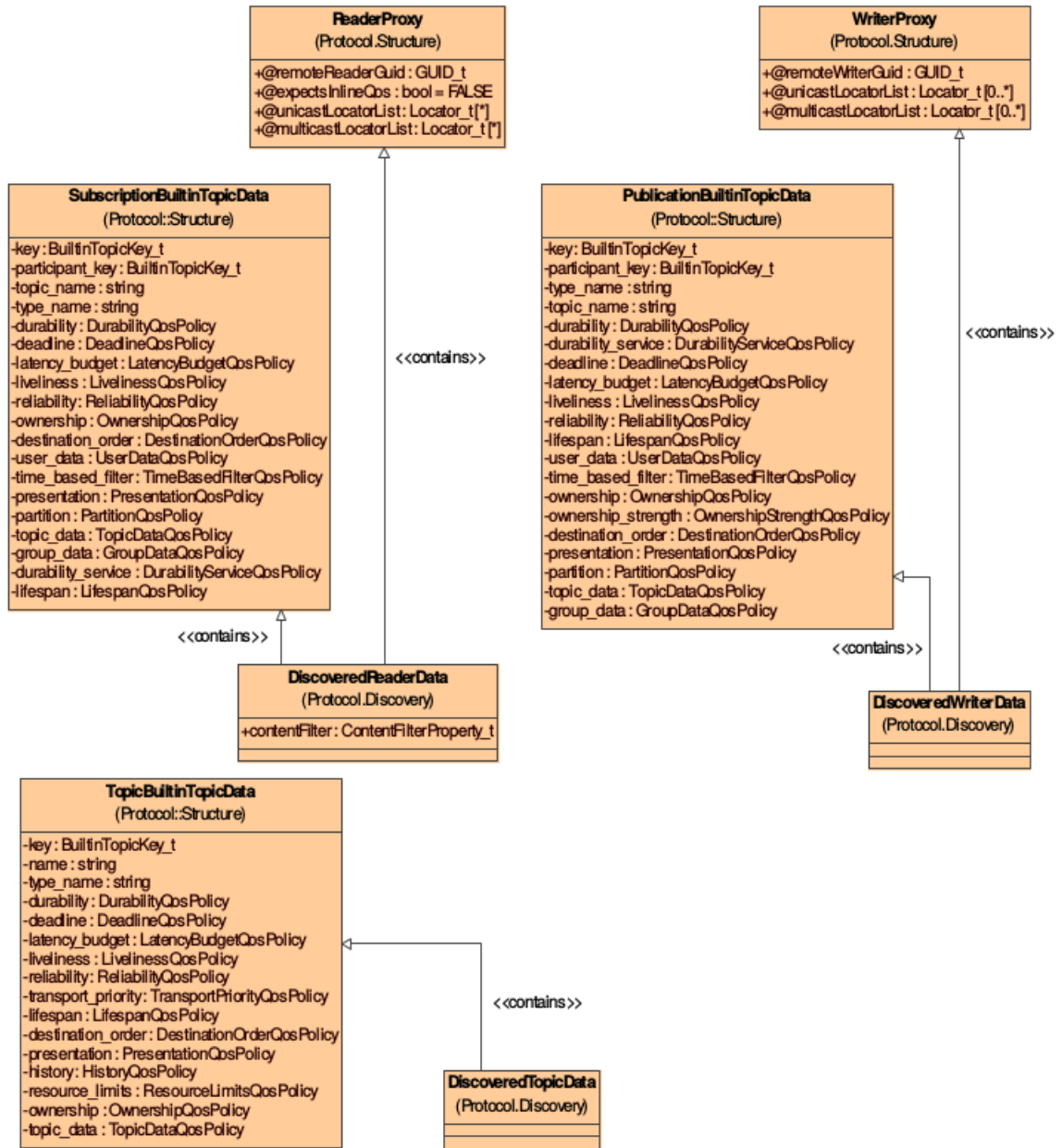


Figura 4.8: Información necesaria para la fase de descubrimiento.

En consecuencia, en la fase de descubrimiento intervienen varios protocolos. Por lo que tanto en *Asio* como en el nuevo módulo habría que hacer una distinción entre la fase de descubrimiento y un envío normal.

4. Diferencias en la estructura `msghdr`. Mientras que en el *user space* la estructura `msghdr` es así:

```

struct msghdr {
    void          *msg_name;      /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec  *msg_iov;       /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void          *msg_control;    /* ancillary data, see below */
    size_t        msg_controllen; /* ancillary data buffer len */
    int           msg_flags;       /* flags (unused) */
};

```

En el *kernel space* los campos de la estructura son diferentes.

```

struct msghdr {
    void          *msg_name; /* ptr to socket address structure */
    int           msg_namelen; /* size of socket address structure */
    struct iov_iter msg_iter; /* data */
    void          msg_control; /* ancillary data */
    __kernel_size_t msg_controllen; /* ancillary data buffer length */
    unsigned int  msg_flags; /* flags on received message */
    struct kiocb   *msg_iocb; /* ptr to iocb for async requests */
};

```

Lo que significa que la información recogida del fichero ha de ser modificada para que se adecue a la estructura `msg_hdr` del *kernel space*.

5. Seguramente la creación del fichero intermedio `/dev/aux` es la mejor opción para conectar *Fast-RTPS* con el kernel, pero en cuanto a latencias se refiere, también es la peor opción, ya que cada vez que se haga una transmisión, hay que abrir el fichero, escribir en él y cerrarlo desde *Asio* y por parte del módulo hay que realizar también las operaciones de abrir, leer y cerrar el fichero. Lo que al final se traduce en un incremento del tiempo de envío del mensaje.

Por todos los motivos descritos en esta sección, sobre todo, debido a la complejidad del módulo de transporte, la creación del módulo del kernel para *Fast-RTPS* no ha sido posible.

Más concretamente, la primera fase del trabajo funciona sin problemas, incluso el paquete llega a enviarse, pero nunca es recibido por la aplicación.

Esto puede ser debido a una falta de datos transmitidos al módulo y por consiguiente, una mala encapsulación, ya que si visualizamos una captura de *Wireshark* se observa claramente la transmisión del paquete, pero con el protocolo UDP, y para que fuese correcto, el protocolo que debería salir es el de RTPS como se puede ver en la captura de la *figura 4.7*. Además, en el espacio del kernel no podemos hacer uso de las herramientas adecuadas que poder crear y enviar un paquete con el protocolo RTPS, ya que no podemos especificar a los sockets que creamos en el módulo que el protocolo utilizado para el envío sea RTPS (aunque por debajo se encuentre UDP).

Capítulo 5

Experimentación

En este capítulo se realizará una explicación detallada de los experimentos realizados y sus resultados, con el objetivo de ver las repercusiones que están teniendo los diferentes cambios en el código de los programas.

5.1. Moviéndonos entre el user space y el kernel space

A fin de ver más claramente los efectos de los cambios en el código introduciendo las primitivas de tiempo real se plantearán tres escenarios diferentes.

Pero antes un pequeño inciso, en la parte izquierda de las figuras de los escenarios que se muestran a continuación se puede ver que también se hace uso de *iperf*, (tanto dentro de nuestro propio equipo como desde un equipo externo) que es una herramienta muy utilizada para testear y medir el rendimiento de las redes informáticas. En este caso se utilizará para añadir tráfico UDP externo a la red y así intentar reproducir una situación de mucho tráfico y sobrecargar el equipo.

5.1.1. Escenario 1

Simple programa cliente-servidor UDP ejecutado en el *user space*, escrito en *C* y que hace envío de un paquete con el mensaje *Hello!*. Kernel no-RT.

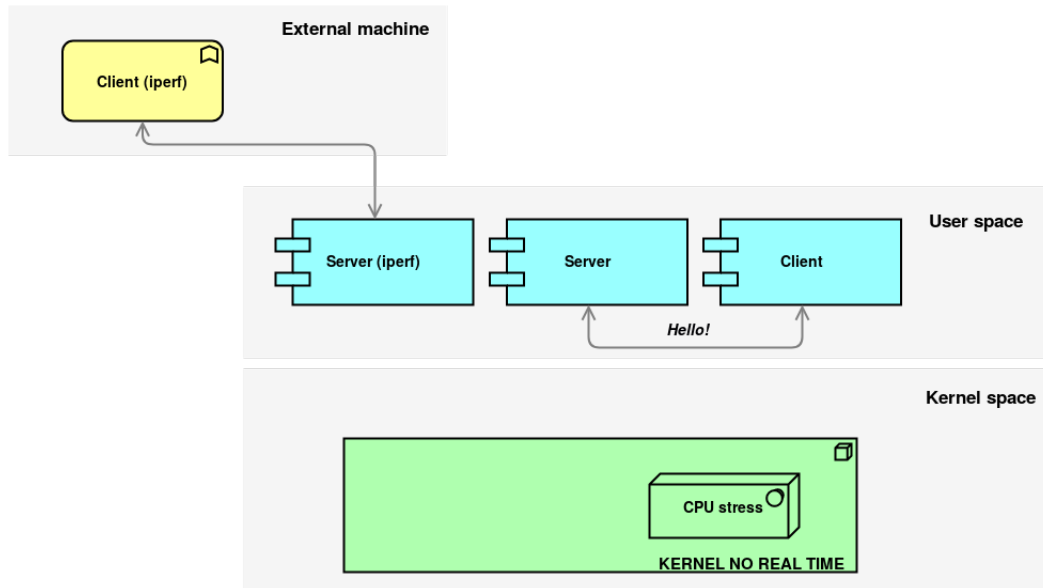


Figura 5.1: Escenario 1.

5.1.2. Escenario 2

Programa cliente-servidor UDP ejecutado en el *user space*, escrito en *C* haciendo uso de las herramientas para convertir aplicaciones normales a aplicaciones de tiempo real descritas en el apartado 4.4 y que realiza el envío de un paquete con el mensaje *Hello!*. Kernel RT.

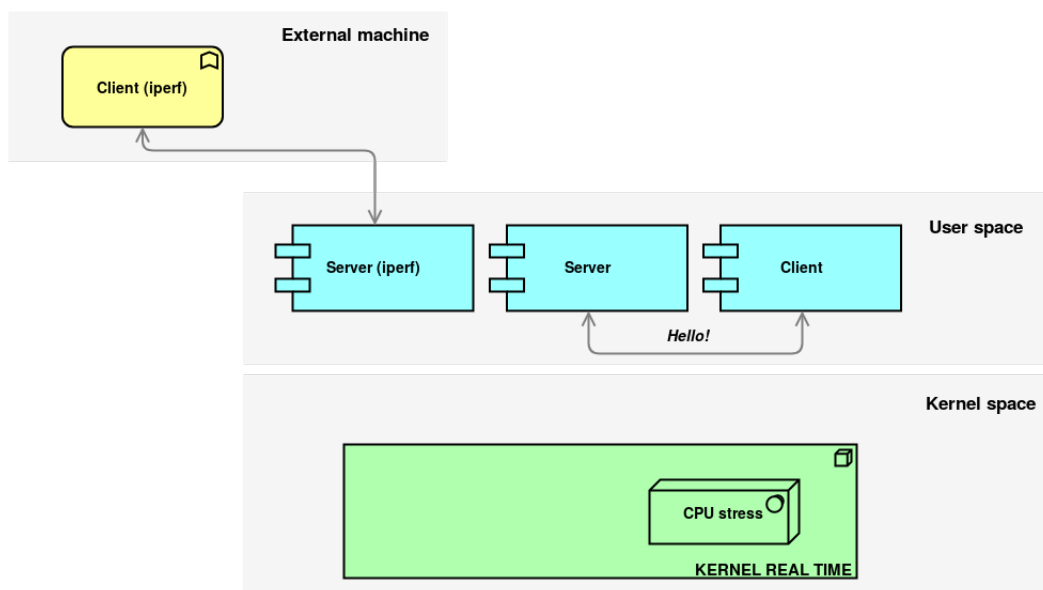


Figura 5.2: Escenario 2.

5.1.3. Escenario 3

Programa servidor UDP escrito en *C* y ejecutado en el *user space* haciendo uso de las herramientas para convertir aplicaciones normales a aplicaciones de tiempo real descritas en el apartado 4.4 y módulo del kernel que actúa como cliente UDP y que realiza el envío de un paquete con el mensaje *Hello!* (también haciendo uso de las herramientas de tiempo real en el espacio del kernel). Kernel RT.

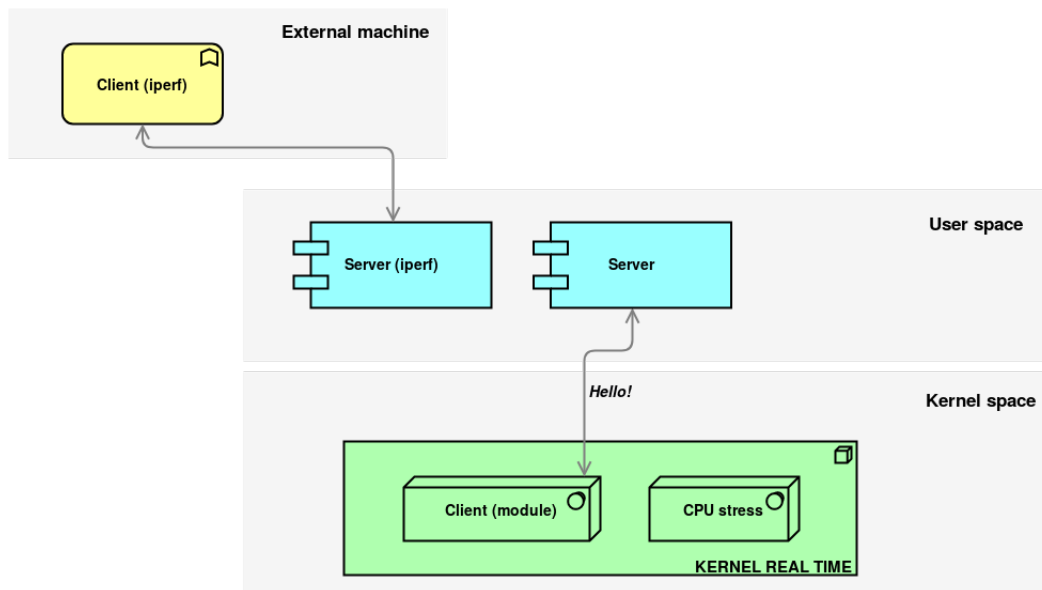


Figura 5.3: Escenario 3.

5.1.4. Gráfica comparativa entre escenarios

Antes de comenzar la comparación entre los distintos escenarios, conviene destacar algunas características importantes, se han utilizado las primitivas de tiempo real, las prioridades asignadas han sido 90 para el servidor y 91 para el cliente. El motivo por el que se han usado distintas prioridades es para que no se produzca una competición entre los dos programas.

Respecto a la reserva de memoria, se ha ejecutado el comando `ps -l eyf` en una terminal y se ha determinado que la cantidad de memoria a reservar para las dos aplicaciones es de 1K aproximadamente.

Para hacerlo más interesante añadimos estrés a la CPU¹ y a la red² en los tres escenarios.

¹`stress --cpu 4` (en el equipo local).

²Inyección de tráfico externo y constante con un cliente `iperf iperf -c 192.168.1.162 -u -P 10 -i 1 -p 5001 -f -k -b 10M -t 1200 -T 1` y un servidor `iperf -s -u -i 1`.

Escenario/Iteración	1	2	3	4	5	6	7	8	9	10
1	0.121	0.060	0.453	0.068	0.354	0.263	0.100	0.048	0.159	0.318
2	0.084	0.075	0.085	0.189	0.121	0.103	0.112	0.088	0.058	0.074
3	0.0034	0.0016	0.0028	0.0031	0.0091	0.0076	0.0080	0.0033	0.0052	0.0058

Cuadro 5.1: Latencias de los escenarios 1, 2 y 3 (s).

Escenario	Media	Mínimo	Máximo
1	0.1944	0.048	0.453
2	0.0989	0.058	0.189
3	0.00499	0.0016	0.0091

Cuadro 5.2: Media, mínimo y máximo de los escenarios 1, 2 y 3 (s).

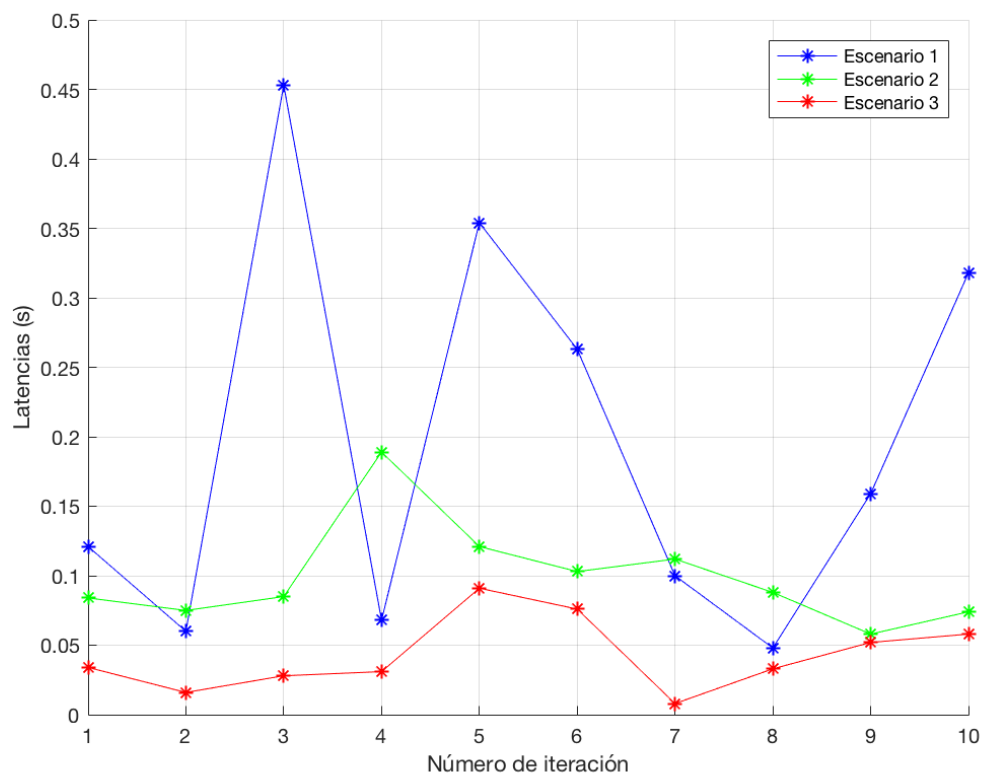


Figura 5.4: Gráfica comparativa entre escenarios.

Como se puede ver en la figura 5.4, a medida que se añaden más primitivas de tiempo real y se trabaja a más bajo nivel se obtiene un sistema más determinista sin latencias imprevisibles y además en este caso concreto también se nota una reducción de las latencias.

5.2. Experimentación con el LatencyTest de Fast-RTPS

Como ya se comentó en apartados anteriores, para hacer una medición precisa de las latencias en *Fast-RTPS* la forma más eficaz y sencilla es utilizando el *LatencyTest* que proporciona el proveedor de *Fast-RTPS*.

Para la selección de las siguientes tablas se realizaron 10 iteraciones para cada uno de los escenarios y se eligió el peor de los diez casos.

5.2.1. Escenario 1

Ejecución del *LatencyTest* proporcionado por *Fast-RTPS*, para 11 tamaños de paquetes distintos y 10000 muestras, en modo *best effort* con un **kernel no-RT**. (Resultados en μs).

Bytes	Samples	stdev	mean	min	50 %	90 %	99 %	99.99 %	max
16	10000	9.00	40.05	26.24	38.75	50.15	65.03	166.34	180.82
32	10000	15.00	37.58	25.74	36.84	49.16	62.56	284.63	1209.88
64	10000	13.00	40.83	27.33	39.81	50.98	61.94	385.42	1010.06
128	10000	9.00	37.55	25.45	36.18	47.81	57.93	153.44	421.30
256	10000	13.00	39.35	25.92	37.74	49.78	75.72	293.89	304.93
512	10000	32.00	38.41	26.46	35.32	49.65	66.82	348.67	3048.83
1024	10000	11.00	42.86	26.28	43.39	53.23	69.65	234.08	291.09
2048	10000	11.00	51.76	33.75	48.17	61.77	77.73	224.79	637.94
4096	10000	11.00	64.58	37.13	66.90	75.77	91.71	219.23	308.25
8192	10000	15.00	65.14	44.58	63.75	82.83	107.98	254.12	598.12
16384	10000	14.00	80.55	58.01	77.23	93.33	125.60	419.16	501.45

Cuadro 5.3: Latencias del escenario 1 (μs).

5.2.2. Escenario 2

Ejecución del *LatencyTest* proporcionado por *Fast-RTPS*, para 11 tamaños de paquetes distintos y 10000 muestras, en modo *best effort* con un **kernel RT**. (Resultados en μs).

Bytes	Samples	stdev	mean	min	50 %	90 %	99 %	99.99 %	max
16	10000	26.00	52.07	35.50	47.62	55.71	238.85	375.35	377.23
32	10000	18.00	48.66	36.81	46.66	52.52	102.07	482.65	490.94
64	10000	25.00	52.43	35.95	48.31	55.26	160.27	435.15	447.76
128	10000	32.00	54.36	37.52	47.90	59.52	240.43	423.57	462.33
256	10000	30.00	53.24	37.20	47.69	57.26	239.53	376.89	446.45
512	10000	11.00	51.47	39.67	50.13	58.88	93.00	257.35	311.62
1024	10000	20.00	52.09	38.02	48.57	57.52	124.53	479.38	502.15
2048	10000	19.00	53.64	41.32	51.31	58.11	113.06	394.21	452.71
4096	10000	50.00	69.55	42.92	57.46	68.84	324.08	434.29	440.55
8192	10000	101.00	114.61	61.43	70.14	352.54	398.38	564.18	577.92
16384	10000	128.00	169.72	75.33	90.25	414.98	440.38	595.76	599.16

Cuadro 5.4: Latencias del escenario 2 (μs).

5.2.3. Comparativa gráfica entre escenarios

Anteriormente, ya se explicó que un kernel RT no garantiza que se vaya a obtener una mejora de las latencias, de hecho, observando la *figura 5.5* las latencias son ligeramente superiores a las de un kernel no-RT, sin embargo, como se ilustra en la *figura 5.6*, con un kernel RT, se obtienen unos resultados más deterministas y se eliminan los molestos "picos" que podrían producir fallos inesperados y hacer que un sistema se viniese abajo.

Estos últimos resultados, junto con la demostración de que portar código al *kernel space* proporciona una mejora en cuanto a rendimiento visto en el *apartado 5.1*, fueron los que nos llevaron a la idea de intentar portar un módulo de *Fast-RTPS* al kernel RT³.

³Idea desarrollada y explicada en el *apartado 4.6* del proyecto

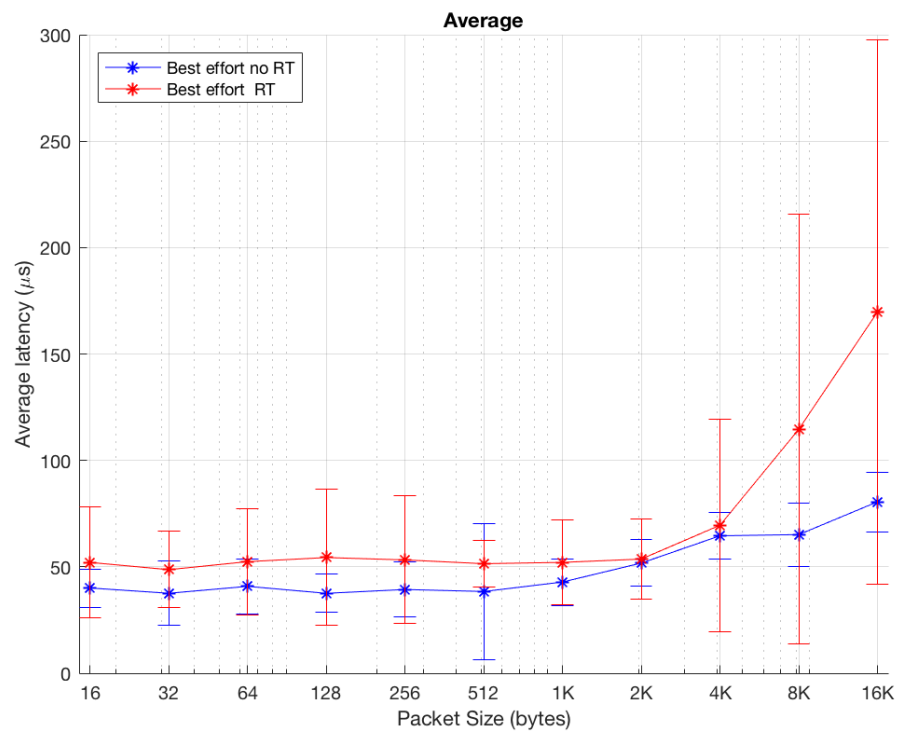


Figura 5.5: Comparativa de la media en modo *best effort* entre un kernel RT y no-RT.

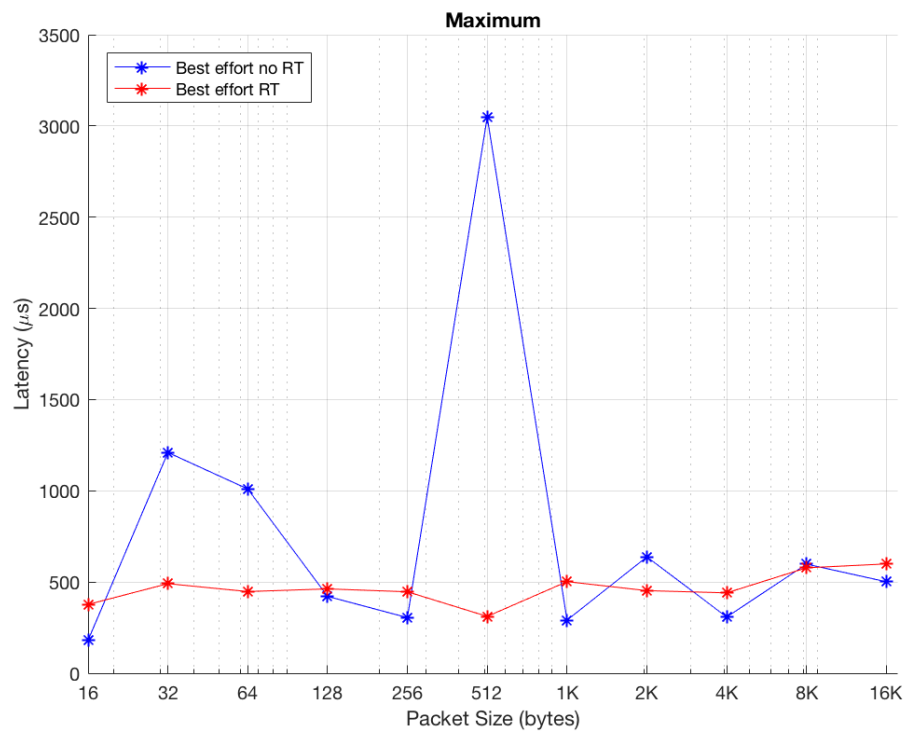


Figura 5.6: Comparativa de máximos en modo *best effort* entre un kernel RT y no-RT.

Capítulo 6

Conclusiones

En esta memoria se ha definido el proceso para crear aplicaciones *Real Time* tanto en el espacio de usuario como del kernel.

En este capítulo se analizará la consecución de los objetivos propuestos. También se enumerarán las competencias del grado de las cuales se ha surtido el proyecto y las adquiridas mientras se llevaba a cabo su desarrollo. Y finalmente, se presentarán una serie de ideas para mejorar el proyecto descrito y sus implicaciones.

6.1. Consecución de objetivos

Al comienzo de este proyecto se propusieron los siguientes objetivos:

- Demostrar que un kernel de tiempo real es más determinista que un kernel normal.
- Usar herramientas de tiempo real en aplicaciones del espacio de usuario.
- Usar herramientas de tiempo real en el espacio de kernel.
- Demostrar que hay una mejora del rendimiento al portar código de una aplicación del espacio de usuario al espacio del kernel.
- Reproducir el punto anterior con *Fast-RTPS*.

Estos objetivos se han cumplido en gran medida, ya que el objetivo más importante de este proyecto era demostrar que un kernel de tiempo real proporciona un sistema determinista para

el desarrollo de aplicaciones *Real Time* en robots y además también hacer una mejora en los tiempos de ejecución de las aplicaciones utilizando otras herramientas también *Real Time*. La consecución de todos estos objetivos anteriormente descritos se llevan a cabo y están explicados detalladamente en el *apartado 5.1* y *5.2* de esta memoria.

Sin embargo, un punto importante de este proyecto era reproducir todos los objetivos anteriores en *Fast-RTPS*. La imposibilidad de conectar el nuevo módulo de nuevo con *Fast-RTPS* unido al tiempo limitado para la realización de este último punto me lleva a decir que ha sido un objetivo no completado.

En cualquier caso, este punto ha servido para demostrar que la implementación de *Fast-RTPS*, además de ser muy compleja también es difícil de optimizar, o al menos en lo que se refiere a la parte de transporte, ya que al final la llamada al sistema *sendmsg* realizaba el envío de los paquetes de la forma más eficaz.

6.2. Competencias utilizadas y adquiridas

Durante el grado se han adquirido unas competencias que han servido de apoyo a la hora de realizar este proyecto, asignaturas como Sistemas Operativos y Sistemas Distribuidos han sido fundamentales ya que el proyecto se ha desarrollado en los lenguajes aprendidos en dichos cursos. Otra base crucial del proyecto ha sido la asignatura Arquitectura de Redes de Ordenadores, ya que en ella se aprenden gran parte de los protocolos de los que *Fast-RTPS* hace uso. Y por último, la asignatura de Robótica que es en cierto modo la unión de todo lo aprendido trasladado al mismo entorno.

Por otra parte, durante el desarrollo de este proyecto se han adquirido otros conocimientos como la programación de módulos del kernel, *ROS2*, el uso de nuevas librerías y bibliotecas como puede ser *Asio* o el enfrentarse a una cantidad de código considerable como es el caso de *Fast-RTPS*.

6.3. Trabajos futuros

A continuación se dan una serie de ideas que se podrían incluir en el proyecto para mejorarlo:

- **Eliminar el fichero intermedio /dev/aux .** Es posible que en este paso intermedio se estuviera perdiendo información, además de introducir mucha latencia.
- **Estudiar otros módulos de *Fast-RTPS*.** Aunque portar código del módulo de transporte no haya tenido buenos resultados por la complejidad en dicho módulo, tal vez mover otra parte del código de *Fast-RTPS* sí sea viable, un posible candidato es el proceso de encapsulación del mensaje.
- **Cambiar la librería *Asio* por otra.** Para notar una mejora de rendimiento en la aplicación, un cambio en la librería que se usa para la creación de sockets y sincronización de procesos sea un posible candidato. Aunque actualmente es posible que pocas librerías sean tan completas como *Asio*.
- **Trabajar directamente sobre el código de *Fast-RTPS* en el *user space*.** Para intentar conseguir una mejora del rendimiento, cambiar algunas primitivas, eliminación de código innecesario...etc.
- **Convertir el código de *Fast-RTPS* en código de una aplicación de tiempo real.** Usando las herramientas descritas en el apartado 4.4.

Bibliografía

[1] CURSO UNIX.

https://bioinf.comav.upv.es/courses/unix/unix_intro.html.

[2] ROS ON DDS.

http://design.ros2.org/articles/ros_on_dds.html.

[3] E. Brown. REAL-TIME LINUX EXPLAINED, AND CONTRASTED WITH XENOMAI AND RTAI.

<http://linuxgizmos.com/real-time-linux-explained/>,

10 de Febrero de 2017.

[4] X. Calbet. BREVE TUTORIAL PARA ESCRIBIR DRIVERS EN LINUX.

<https://www.ibiblio.org/pub/linux/docs/LuCaS/>

Presentaciones/200103hispalinux/calbet/html/t1.html,

http://www.exa.unicen.edu.ar/catedras/rtlinux/material/apuntes/driv_tut_last.pdf.

[5] X. Calbet. WRITING DEVICE DRIVERS IN LINUX: A BRIEF TUTORIAL.

http://freesoftwaremagazine.com/articles/drivers_linux/.

[6] I. K. Center. DEVICE NAMES, DEVICE NODES, AND MAJOR/MINOR NUMBERS.

https://www.ibm.com/support/knowledgecenter/en/linuxonibm/com.ibm.linux.z.lgdd/lgdd_c_udev.html.

[7] W. W. Deanna Hood. ROS 2 UPDATE, ROSCON 2016.

<https://roscon.ros.org/2016/presentations/ROSCon%202016%20-%20ROS%202%20Update.pdf>,

8 de octubre de 2016.

- [8] P. J. Echaiz. PROCESOS, PLANIFICACIÓN 3.
<http://cs.uns.edu.ar/~pmd/sosd/downloads/Slides/03-Procesos%20BW.pdf>.
- [9] eProxima. FAST-RTPS GITHUB.
<https://github.com/eProxima/Fast-RTPS>.
- [10] eProxima. FAST-RTPS MANUAL.
<http://docs.eprosima.com/en/latest/introduction.html>,
2016.
- [11] B. O. Gallmeister. PROGRAMMING FOR THE REAL WORLD POSIX.4.
- [12] A. L. S. Iglesias. ¿QUÉ ES LA MEMORIA VIRTUAL Y POR QUÉ ES IMPORTANTE?
<https://www.aboutespanol.com/que-es-la-memoria-virtual-\y-por-que-es-importante-841348>,
17 de junio de 2016.
- [13] C. M. Kohlhoff. ASIO C++ LIBRARY.
<https://think-async.com/>,
2013-2015.
- [14] E. C. Kucukoglu. KERNEL PREEMPTION AND REALTIME LINUX KERNEL WITH PREEMPT_RT.
http://eckucukoglu.com/linux/kernel-preemption-and-realtime-\linux-kernel-with-preempt_rt-part-1/,
31 de agosto de 2015.
- [15] I. Linux Kernel Organization. THE LINUX KERNEL ARCHIVES.
<https://www.kernel.org>.
- [16] J. M. Losa. COMMUNICATING WITH ROS ROBOTS USING FAST RTPS.
<https://www.slideshare.net/JaimeMartin-eProxima/fiware-communicating-with-ros-robots-using-fast-rtps>,
30 de mayo de 2017.

- [17] D. Molloy. LINUX KERNEL PROGRAMMING.
<http://derekmolloy.ie/writing-a-linux-kernel-\module-part-1-introduction/>.
- [18] B. Nichols, D. Buttlar, and J. P. Farrell. PTHREADS PROGRAMMING.
- [19] OMG. THE REAL-TIME PUBLISH-SUBSCRIBE PROTOCOL(RTPS) DDS INTEROPERABILITY WIRE PROTOCOL SPECIFICATION.
www.omg.org/spec/DDS-RTSPS/2.2/PDF,
Septiembre de 2014.
- [20] I. Open Source Robotics Foundation. ROS ON DDS.
http://design.ros2.org/articles/ros_on_dds.html.
- [21] I. Open Source Robotics Foundation. WHY ROS 2.0?
http://design.ros2.org/articles/why_ros2.html.
- [22] E. Robotics. DDS.
http://docs.erlerobotics.com/robot_operating_system/ros2/basic_concepts/dds.
- [23] E. Robotics. MIDDLEWARE INTERFACE.
http://docs.erlerobotics.com/robot_operating_system/ros2/basic_concepts/middleware_interface.
- [24] E. Robotics. ROS 2.
http://docs.erlerobotics.com/robot_operating_system/ros2.
- [25] A. Rubini, J. Corbet, and G. Kroah-Hartman. LINUX DEVICE DRIVERS.
- [26] A. G. Serrano. DESARROLLO DE MÓDULOS PARA EL KERNEL LINUX.
<http://www.ellaberintodefalken.com/2014/03/desarrollo-modulo-kernel-linux.html>.
- [27] D. Thomas. ROS 2 MIDDLEWARE INTERFACE.
http://design.ros2.org/articles/ros_middleware_interface.html.

- [28] J. Torres. BSD SOCKETS.

<https://medium.com/jmtorres/bsd-sockets-7b50fccf71e8>,

1 de Marzo de 2013.

- [29] T. Ts'o, D. Hart, and J. Kacur. REAL-TIME LINUX WIKI.

https://rt.wiki.kernel.org/index.php/Main_Page,

22 de agosto de 2016.

- [30] L. Vallespín. ¿CUÁL ES LA DIFERENCIA ENTRE C, C++ Y C?

<https://es.quora.com/Cu%C3%A1l-es-la-diferencia-entre-C-C++-y-C>,

27 de Diciembre de 2016.

- [31] L. F. Wiki. HOWTO BUILD A SIMPLE RT APPLICATION.

https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base.

- [32] L. F. Wiki. MEMORY FOR REAL-TIME APPLICATIONS.

<https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory#memory-locking>,

<https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory#stack-memory-for-rt-threads>.