UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Sistemas de Comunicaciones

TRABAJO DE FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE RTPS EN ARDUINO

Autor: Norbert Ludant
Director: Jaime José García Reinoso

Septiembre de 2015

Título: DISEÑO E IMPLEMENTACIÓN DE RTPS EN ARDUINO
Autor: Norbert Ludant
Director: Jaime José García Reinoso

EL TRIBUNAL

Presidente:	Bravo Santos, Ángel María
Vocal:	Martín Martínez, Ignacio
Secretario:	Choquehuanca Zevallos, Juan Jose

Realizado el acto de defensa y lectura del Trabajo de Fin de Grado el día 8 de Octubre en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

En primer lugar, quiero mostrar mi agradecimiento a la Universidad Carlos III de Madrid por estos cuatro años en los que he disfrutado aprendiendo.

A mi tutor del trabajo, Jaime García Reinoso, quiero agradecerle el apoyo y dedicación que me ha dado durante los últimos meses.

A Agustín Santos Méndez, le agradezco todo el apoyo que me ha brindado, el tiempo que me ha dedicado y todo lo que he aprendido de él, sin su ayuda esto no habría sido posible.

A mi familia, amigos y pareja tengo que darles las gracias por el apoyo que me han dado. Siempre que los he necesitado he podido contar con ellos.

Abstract

Technology and its relationship with the human being is changing constantly. After the Internet revolution, we are probably facing a new era led by new concepts, such as IoT.

IoT and the intelligent systems it enables will change our world deeply. The benefits of Internet of Things regarding a wide variety of economic areas, specially for emerging economies, are significant, and strategies to realise these benefits need to be found.

Those new intelligent systems are related to new schemes based on devices and data centric communication, they also need new distribution patterns, that is why many real-time publish-subscribe IoT protocols that can connect thousands of devices have emerged. One of these, Data Distribution Service (DDS), is really relevant because it targets devices that directly use device data, that is, it can distribute data to other devices in an effective way.

In this degree thesis we are going to implement the main functionalities of the Wire Protocol for DDS, RTPS, in an IoT platform, Arduino.

Keywords: Internet of Things, Publish-Subscribe, DDS, RTPS, Arduino

Table of Contents

Table of figures

Acronyms

CDR Common Data Representation

DDS Data Distribution Service

EDP Endpoint Discovery Protocol

GUID Globally Unique Indentifier

IDE Integrated Development Environment

IDL Interactive Data Language

JMS Java Messaging Service

M2M Machine To Machine

PDP Participant Discovery Protocol

PIM Platform Independent Model

PSM Platform Specific Model

RTPS Real-Time Publish-Subscribe

SEDP Simple Endpoint Discovery Protocol

SPDP Simple Participant Discovery Protocol

1 Introduction

We are experiencing deep changes in our lives in relation with technology, first came the Internet revolution, with a new communication and working model, transforming life in many ways, and now, the next wave of revolution from the Internet is not about people anymore, it is about intelligent, connected devices.

This new concept is still under development but arises with a great potential to provide us with a before' and an after', although this technology is having a big impact in the present, IoT will change the world even more profoundly in a future. IoT will change our daily lives, including the way we use and save energy, manage healthcare, or live and work, introducing a lot of 'smart' functionalities to daily objects. But especially this will imply a big revolution in industry, where there will be a big impact on cost savings, higher productivity, and connection on this scale will enable applications beyond imagination.

In addition, over the period of last two decades, data-centric design is emerging as a crucial element to develop advanced data-critical distributed systems that link diverse control oriented embedded devices and sensors to data processing systems usually in a real-time environment within large enterprises [1]. The IoT's opportunity will be to connect all these devices and sensors to deliver truly distributed machine-to-machine (M2M) applications.

As Kevin Ashton explained, "*If we had computers that knew everything there was to know about things—using data they gathered without any help from us—we would be able to track and count everything, and greatly reduce waste, loss and cost. We would know when things needed replacing, repairing or recalling, and whether they were fresh or past their best.*" He concluded, "*The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so.*" [2]

So in conclusion the vast new opportunity applying this concept is about infrastructure in industry. Smart machines will change the world economy more than anything since the industrial revolution. These smart machines will combine to form an Industrial Internet of Things (IIoT) that connects devices into truly intelligent distributed systems. Recent studies show the relevance of IoT nowadays, it is found that 75 percent of the enterprises are using IoT, so there is lots of work to do on this area. [3]

However, with the emergence of this new data-centric systems related to IoT, problems also appear, for this purpose of data distribution, most of the middleware options developed are being based in the same pattern of communication, the request-reply pattern. This communication pattern has been used for a long time with indubitable results, but now that a new communication approach has appeared, the typical communication where a Client interacts with a Server requesting a reply is not efficient.

For this purpose, it is better to use a different pattern, especially when we are dealing with a data-centric system, that is where the Publish-Subscribe pattern comes in. Publish/Subscribe offers a data distribution pattern where there are entities, publishers, who publish a type of data, and there are other entities, subscribers, that decide which type of data they want to receive, so we have a decoupled model where it is not required to ask periodically for the data, the subscriber just declare its interest in receiving the data updates. In addition, the communication is topic based, so this pattern provides greater network scalability and a more dynamic network topology just what we expect to interconnect small data devices, which will grow in number really fast.

In resume, we have the following problems:

- Data-centric systems are growing, so an efficient distribution mechanism is needed to handle all this data.
- IoT is yet a controversial term, standardization and developing is highly needed around this concept.

Lots of work was done around these problems, the biggest efforts on this area to achieve both standardization and more efficient distribution application systems were made by two organizations, OMG and Sun Microsystems, that have developed his own standard specifications for real time publish-subscribe middleware, DDS and JMS respectively. Of these two protocols we are more concerned about DDS, because it implements a powerful QoS mechanism, which is crucial for M2M IoT applications.

"*DDS is a protocol for the IoT which enables network interoperability for connected machines, enterprise systems, and mobile devices. It provides scalability, performance, and Quality of Service required to support IoT applications. DDS can be deployed in platforms ranging from low-footprint devices to the Cloud and supports efficient bandwidth usage as well as agile orchestration of system components. It provides a global data space for analytics and enables flexible real-time system integration*." [4]

This protocol has been deployed in a lot of industry systems but it lacks of interoperability between vendors, because the lower layer of communication had not been standardized. To unify implementations, guarantee vendor interoperability and provide to DDS all the needed functionalities, a Wire Protocol was also designed, DDSI-RTPS, The Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol.

RTPS protocol found its roots in industrial automation and was in fact approved by the IEC and is a field proven technology that is currently deployed worldwide in thousands of industrial devices. [5, pp. 6-7]

1.1 Motivation

As we saw at this point, we have a market that is growing exponentially, based on a concept that can revolutionize everyday life and our reality, IoT.

Having said that, the development of new applications for IoT and promoting the development of this technology is something really interesting with a view to the future. Having seen the problems of this new universe, and researching about the solutions proposed by different organizations, it is an urgent goal to deploy IoT related protocols, like, for example, the protocol that gives functionality to DDS and enables network interoperability between different devices, RTPS.

Both these protocols have been implemented on a large number of platforms, however, there are no implementations on low performance development boards, just on more robust and complex systems. In addition to the nature of the protocol designed for low power consumption and distributed systems, there is a special interest on implementing this protocol on an IoT oriented development board, matching those requirements.

Eventually, our goal is to implement the RTPS protocol on a low performance development board IoT-oriented to try to give a solution to this industry need.

1.2 Objectives

Once seen the whole environment we are working with, and what needs have to be covered, let us check the goals we set.

The main goal of this research is the deployment of the RTPS protocol in a low performance microcontroller. This research includes choosing the right development board, exploring its limitations, such as memory, communication limitations and viability of adaptation of the programming language from OMG's specification, in order to check which are the main features we can implement and if it is possible to maintain this protocol.

So the main objectives of this project are:

1. Choosing our IoT platform.
2. Choosing the RTPS functionalities we are going to implement
3. Implement those functionalities on the selected platform
4. Validation of this implementation

This project is focused around these objectives, we will make continuous reference to this objectives along the next chapters.

2 Estado del Arte

2.1 Modelo Publish/Subscribe

2.1.1 Características modelo Publish-Subscribe

El comportamiento de este esquema es simple y fácil de entender, en cuanto un Publisher tiene algún tipo de datos que pueden ser útiles para el resto de los usuarios, publica estos datos a los Suscribers interesados.

En resumen, Publish-Subscribe es una abstracción para la comunicación de uno a muchos que provee una comunicación anónima, desacoplada y asíncrona entre un Publisher y sus Subscribers. Los principales beneficios de este esquema son:

- Rendimiento: mejor latencia y tasas de envío. Los datos son enviados tan pronto como estén disponibles.
- Modelo altamente desacoplado: no se requiere de una petición de datos periódica, el Subscriber simplemente declara su interés en algún tipo de dato y estos son actualizados.
- Escalabilidad.
- Facilidad de adaptación a sistemas reales.

Este patrón de comportamiento provee una mayor escalabilidad en la red y una topología más dinámica, lo que también conlleva menor flexibilidad a la hora de modificar el Publisher y la estructura de los datos publicados.

Por otro lado, también presenta desventajas, como por ejemplo:

- Desconocimiento del estado de los Subscribers, que pueden estar caídos.
- Sobrecarga de la red ante un número muy elevado de pubs/subs.
- El estado del envío de los mensajes puede no ser notificado al Publisher, por lo tanto no hay conocimiento acerca del éxito o fallo en el envío.
- Problemas de seguridad debidos a accesos desautorizados.

Dependiendo del diseño y las funcionalidades a las que hay que amoldarse, se debe tener muy en cuenta las fortalezas y debilidades de este tipo de patrones de comunicación para no cometer errores graves de diseño, tanto para una comunicación cliente – servidor como en entornos donde se necesite una comunicación intensa bidireccional, típica de otro tipo de modelos de comunicación, no es conveniente aplicar estos patrones emergentes.

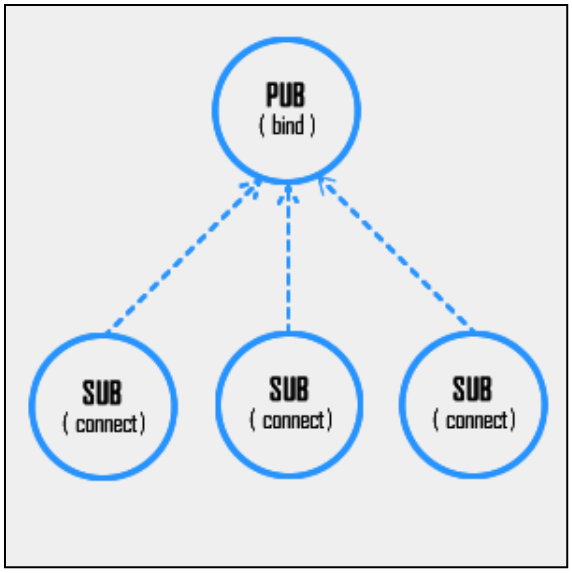


Figure 2.1. Publish-Subscribe pattern [6]

2.1.2 Middleware Publish-Subscribe para sistemas distribuidos

Pero antes, no podemos hablar de pub/sub sin mencionar un concepto que está en auge y que va a cobrar una gran importancia en un futuro muy cercano y ya en el presente. Ya hemos hablado de él, este concepto es Internet of Thigs (IoT) y en parte es lo que da sentido al desarrollo de nuevos mecanismos de distribución de datos.

Este concepto es aún ciertamente novedoso, y se está trabajando mucho sobre él, continuamente aparecen entidades con el fin de estandarizar este concepto, para poder aplicarlo de manera unánime y buscar soluciones interoperables entre los distintos proveedores, dispositivos, componentes, redes… Todo bajo unas cualidades mínimas tanto de seguridad, flexibilidad, integridad y fiabilidad. Además, al ser un concepto tan potente, hay que establecer qué implica IoT, qué es y no es IoT, y cómo debe funcionar.

Para este trabajo de estandarización de IoT y también de las comunicaciones M2M, que están muy ligadas, hay un gran número de entidades trabajando, incluso hay una sobrecarga de estándares al respecto, lo que dificulta su comprensión, sin embargo, destacaremos el trabajo de la ITU, con su iniciativa The Global Standards Initiative on Internet of Things (IoT-GSI), que finalizó en Julio de este mismo año, o la constante labor del ETSI, European Telecommunications Standard Institute, el cual ha desarrollado una serie de especificaciones tanto para IoT como M2M, tratando de esclarecer las bases.

Aún así, al tratarse de un concepto con tanta proyección en términos económicos, es constante la aparición de nuevos protocolos, como Threat, de Google, el cual tiene como objetivo crear un estándar para la comunicación entre los dispositivos domésticos conectados y se une a otros tantos protocolos ya creados.

Sin embargo, debido a la problemática de recuperar información de dispositivos como sensores u otros tipos de dispositivos con los protocolos actuales, aparecen nuevas soluciones y estándares.

Dos de estos estándares de middlewares API para cumplir la tarea de la distribución con el patrón pub/sub son Data Distribution Service (DDS) y Java Messaging Service (JMS), porque son sencillos de utilizar y ofrecen los beneficios del modelo de comunicación Publish-Subscribe, resultando en aplicaciones distribuidas escalables apenas acopladas.

Por supuesto, ambos sistemas están centrados en sistemas en tiempo real debido a su importancia. Algunas de las aplicaciones de los sistemas centrados en tiempo real se pueden encontrar en C4I, atomatización industrial, control distribuido y simulación, control de equipos de telecomunicación, redes de sensores o sistemas de gestión de red, un gran número de campos que se encuentran en continuo crecimiento y necesitan de diseños centrados en datos.

Sin embargo, retomando los estándares DDS y JMS, sus diferencias tienen un impacto a tener en cuenta para el tipo de diseño centrado en datos, debido a sus diferencias centraremos nuestra atención en DDS, que se encuentras más ligado a un sistema basado en datos que JMS.

2.1.3 DDS

En este contexto tenemos a Object Management Group (OMG), una de estas entidades con el fin de estandarizar las ventajas que nos ofrece mecanismos de distribución ligados a IoT para la industria y desarrollando un middleware en tiempo real capaz de aprovechar el esquema PubSub y teniendo muy en cuenta IoT.

De esta forma sus esfuerzos se han centrado en realizar una estandarización para la interoperabilidad entre distintos tipos de middleware, teniendo en cuenta las diferentes opciones que puede ofrecer, y también entre redes, máquinas conectadas o sistemas móviles. Además, se trata de un protocolo uy potente potente que provee escalabilidad, buen redimiento, fácil adaptación y QoS necesarios para soportar las aplicaciones IoT.

La escalabilidad está presente viendo en todos los dispositivos en los que se puede implementar, desde plataformas sobre dispositivos con bajos requisitos de memoria, hasta servicios en la Nube, soportando el uso de ancho de banda suficiente y así como una ágil interoperación entre componentes del sistema. También provee un espacio global de datos para análisis y permite la integración de sistemas real time flexibles.

Como se ha comentado previamente, QoS es una de las características que soporta DDS, es importante hacer hincapié en este hecho porque se trata de un tema crucial para IoT, DDS lo ha cubierto con una robusta implementación de QoS, al contrario que JMS, que carece de esta característica..

En conclusión, DDS especifica un middleware diseñado para hacer posible la distribución de datos en tiempo real con una patrón de comunicación eficaz adaptado a las exigencias.

La solución ofrecida por este estándar para los sistemas centrados en datos es la de proveer un middleware M2M potente, incluso capaz de implementar sistemas integrados completos "sensor-to-cloud", que conecten sistemas operacionales con la analítica propia de la nube, que puede ser clave para el desarrollo de IIoT (IoT en la industria).

La importancia de este protocolo se puede comprobar rápidamente al ver los sistemas en los que se encuentra operando, además de estar implementado en una gran cantidad de áreas de la industria, se encuentra presente en las más críticas áreas, debido a su gran fiabilidad y flexibilidad, como por ejemplo en entornos de tiempo real tan exigentes como sistemas de defensa o aeroespacial [7].

Adentrándonos técnicamente en DDS, para hacer uso del patrón de comunicación Publish-Subscribe, DDS define una interfaz a nivel de aplicación que soporta Data-Centric Publish-Subscribe (DCPS) en sistemas de tiempo real. Además, el estándar de DDS incluye la especificación de Interface definition language (IDL), de este modo una aplicación que trabaje con DDS puede intercambiar entre distintas implementaciones de DDS simplemente recompilando, tiene "application portability". El término de IDL es muy importante y haremos hincapié en él más adelante.

Todos estos esfuerzos de estandarización han conseguido crear una base muy potente de sistemas distribuidos de datos, pero aún quedan cabos sueltos a la hora de crear una interoperabilidad total, como veremos a continuación, aún queda mucho por desarrollar.

2.1.4 RTPS como solución al problema de interoperabilidad de DDS

Aunque hayamos visto que DDS ofrece gran interoperabilidad en la capa de aplicación del propio protocolo, no es el caso de la capa inferior, no tiene soporte de portabilidad para interoperabilidad entre protocolos como TCP/UDP/IP, si no que es el propio vendedor el que tiene que proveer el mecanismo de comunicación bajo el protocolo que se utilice.

A raíz del crecimiento de la implementación de DDS en sistemas de distribución grandes, ha surgido la necesidad de implementar un protocolo de la capa inferior que provea los mecanismos de comunicación para lograr la interoperabilidad entre protocolos bajo las condiciones que establece y necesita DDS para su funcionamiento, es ahí donde nace RTPS, el estándar de protocolo de comunicación definido por el mismo grupo, OMG.

Vemos que este problema de interconexión es análogo al que otros sistemas también han tenido y han tratado de dar solución, la relación entre RTPS y DDS es como IIOP con COBRA o lo que es JRMP para RMI.

El protocolo Real-Time Publish Subscribe (RTPS) es un protocolo de comunicación para dotar de interoperabilidad entre implementaciones, diseñado a priori para DDS. Su propósito es el de asegurar que aplicaciones basadas en las distintas implementaciones de DDS de diferentes proveedores puedan interoperar.

De este modo RTPS está centrado en proporcionar todos los requisitos que DDS necesita, tanto de servicios que soporta como de fiabilidad, funcionalidad u optimización.

Las principales características del protocolo, extraídas del estándar, incluyen:

“- Capacidad de multicast.

- Propiedades de calidad de servicio y rendimiento para dar soporte best-effort y posibilitar comunicaciones fiables publish-subscribe para aplicaciones en tiempo real sobre redes IP estándar.

- Tolerancia a fallos para permitir la creación de redes sin ningún tipo de fallo.

- Naturaleza best-effort orientada a no conexión.

- Distribución eficiente con la mínima sobrecarga.

- Configurabilidad para permitir equilibrar los requisitos de fiabilidad y puntualidad de cada entrega de datos.

- Extensibilidad para permitir la extensión y mejora del protocolo con nuevos servicios sin romper la compatibilidad e interoperabilidad con anteriores versiones.

- Modularidad para permitir que los dispositivos simples sean capaces de implementar un subconjunto del protocolo y sea posible su participación en la red.

- Escalabilidad para permitir a los sistemas escalar potencialmente a redes muy extensas.

-Conectividad plug-and-play para permitir que nuevas aplicaciones y servicios sean automáticamente descubiertos y permitir que las aplicaciones puedan unirse y abandonar la red sin necesidad de reconfiguración.” [5]

Es por esto que la implementación más común y la que vamos a desarrollar en detalle es sobre tecnologías de transporte intrínsecamente best-effort, que trabajen sin conexión, con capacidad multicast, como UDP.

2.1.5 RTPS en dispositivos de bajas prestaciones

Hasta ahora hemos visto la importancia de los sistemas centrados en datos, la importancia presente y sobretodo futura, el potencial que tendrán y la importancia de los protocolos DDS y RTPS para dar soporte a este último, todo englobado para ofrecer un estándar que encaje en ayuda de los problemas planteados por IoT. Ahora, una vez habiendo revisado dónde encaja todo, vamos a ver de manera concreta donde puede ser realmente útil la implementación de esta idea.

Las implementaciones tanto de RTPS como DDS son relativamente frecuentes, podemos encontrar bastantes implementaciones de distintos proveedores, como por ejemplo para DDS tenemos InterCOM® DDS de Gallium, IBM WebSphere R3 Services Asset, OpenSplice DDS de PrismTech, RTI Data Distribution Service de Real-Time Innovations o CoreDX DDS de Twin Oaks Computing. Todas ellas forman el principal grueso de implementaciones de DDS. Mientras que para RTPS tenemos RTPS Open-source: eProsima Fast RTPS o The Open Real-Time Ethernet (ORTE)

Sin embargo, todas estas implementaciones están orientadas a sistemas de procesado potentes, mientras que las implementaciones sobre pequeños dispositivos no han sido desarrolladas, el despliegue de estos protocolos despierta gran interés en microcontroladores o placas de desarrollo de bajas prestaciones.

Además, RTPS puede encajar dentro de las características que ofrecen las placas de desarrollo, ya que:

- Está diseñado para sistemas de distribución en tiempo real: Comunicaciones de bajo periodo de latencia y bajo uso de ancho de banda.
- Altamente personalizable: Permite utilizar diferentes parámetros de QoS adaptándose a diferentes tipos de datos o diferentes enlaces.
- Wireless enabled: Puede ser implementado sobre cualquier tipo de enlace de comunicación, incluyendo enlaces intermitentes de datos, como redes inalámbricas de bajo ancho de banda.
- Soporte multicast: Una característica muy relevante en este tipo de redes, además puede implementarse bajo enlaces no IP.
- Liviano: Requiere muy pocos recursos, puede ser implementado en dispositivos de baja potencia.

Como vemos RTPS va de la mano con las especificaciones de las placas de desarrollo o los microcontroladores y lo que estos pueden ofrecer, asique sería interesante ver que puede ser posible de realizar con un microcontrolador en relación a RTPS, si cumplirá los requisitos y aplicarlo a entornos reales.

Veamos qué ofrecen los microcontroladores de bajas prestaciones a RTPS:

- En primer lugar su coste es bastante reducido, lo que nos posibilita el despliegue de grandes sistemas de distribución con un coste bajo.
- Bajo consumo de potencia, posibilitando la inclusión de Arduino en sistemas donde la autonomía sea importante, como drones.

- Alta interoperabilidad con cualquier tipo de hardware que pueda proveernos de nuevas funcionalidades, las placas de desarrollo son capaces de comunicarse con el entorno y tomar datos a través de una gran cantidad de sensores que es posible añadir de manera sencilla, y además realizar acciones en consecuencia, como controlar motores, medios luminosos, o cualquier otro tipo de actuadores.
- Soportan múltiples tecnologías, como GSM, Ethernet, WiFi, y sobre diversos protocolos de comunicación, como UDP, TCP, USART.

2.2 Estudio de las principales plataformas para trabajar sobre IoT

Tras realizar el análisis de adaptar el protocolo RTPS a sistemas de bajas prestaciones, bajo consumo, baja potencia, se ha visto que ofrece un gran número de ventajas que van en consonancia con lo que es RTPS.

El desarrollo de microcontroladores con capacidad de interactuar con el entorno ha crecido de manera exponencial en los últimos años debido al avance de la automatización, electrónica, la reducción de costes y sobretodo la capacidad de IoT, tanto en la industria como en otros ámbitos, como la domótica. Estos microcontroladores están diseñados para reducir el costo económico y el consumo de energía, por lo que es perfecto en entornos industriales, además, nos permite realizar muchas más funciones que no sería posible sin ellos.

Por otro lado, es crucial el hecho de que estos microcontroladores posean la capacidad de interactuar con el entorno, pudiendo formar sistemas muy complejos y con unas funcionalidades a las que se puede aplicar que prácticamente parecen infinitas.

Pero no todos los microcontroladores son iguales, y aunque se basan en el mismo principio debemos tener en cuenta qué necesitamos y cómo va a cumplir estos requisitos la placa que decidamos utilizar. Algunos de estos parámetros que nos interesa analizar son por ejemplo la autonomía, el coste, la flexibilidad, escalabilidad, capacidad de adaptarse a otros sistemas, inclusión de nuevo hardware o soporte actual del software que utiliza.

Entre las opciones más accesibles de microcontroladores orientados a IoT encontramos:

Raspberry Pi

Se trata de una de las plataformas que integran hardware-software más conocidas, una placa de desarrollo con un gran número de periféricos y conexiones, por lo tanto ofrece muchísimas posibilidades a la hora de implementar nuestros proyectos, desde conexión USB a Ethernet, cubre gran parte de las conexiones que pudiésemos querer utilizar.

Sus ventajas son unas prestaciones medias, mucho más altas de lo esperado para una placa de desarrollo comparándolo con otros dispositivos de estas características:

Procesador ARM11 de hasta 1Ghz realizando overclocking.

Procesador gráfico.

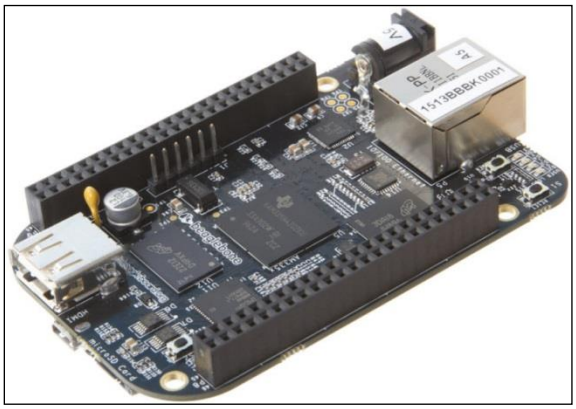
512 MB de RAM.

Todo ello en la placa más básica, Raspberry Pi, la sucesora Raspberry Pi 2 cuenta con unas especificaciones aún más potentes [8]. Además, ofrece un entorno de desarrollo muy potente y realmente fácil de usar.

Todas estas características son muy positivas, pero su misma ventaja se convierte en una desventaja para este fin concreto, un rendimiento tan alto parece un poco exagerado para la implementación que queremos realizar, además de no cumplirse el compromiso necesario de autonomía de la placa para el proyecto, no necesitamos tener unas prestaciones tan altas.

En definitiva esta placa no parece encajar dentro del contexto de lo que queremos realizar, está mas orientada a proyectos de domótica o proyectos propios de un alto potencial, no tan orientado a un sistema embebido en tiempo real. [8]

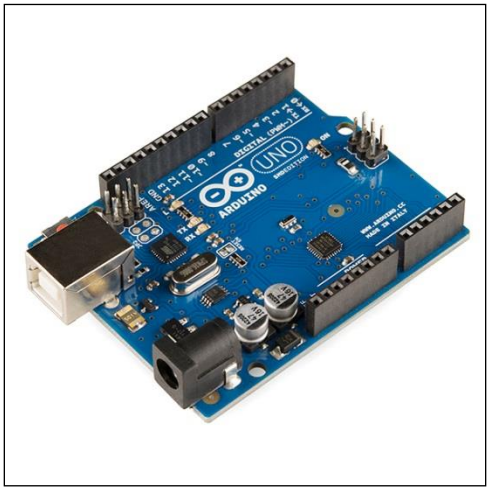
BeagleBone



En la misma línea se barajó la posibilidad de implementar RTPS en BeagleBone. Las prestaciones que ofrece esta placa son muy similares a las de Raspberry Pi, son prácticamente miniPC's y su potencia y características son excesivas, las desventajas son las mismas. [9]

En relación a estos dos anteriores sistemas tenemos por ejemplo UDOO Neo, que ofrece aún mas posibilidades y a un precio muy competititvo, pero presenta los mismos pros y contras que Raspberry Pi y BeagleBone [10]

Arduino



Arduino es mundialmente conocido, es una de las primeras opciones de open-source creadas y ha recibido una gran aceptación por parte de la comunidad investigadora y docente además de recibir muchos reconocimientos y premios.

Arduino es una placa de desarrollo basada en un microcontrolador Atmega28, es decir, incluye hardware pero también un entorno de desarrollo que lo hace muy potente, el IDE asociado a Arduino tiene una gran comunidad detrás que realiza grandes avances, es un proyecto que sigue muy vivo.

Algunas de sus ventajas son:

El Hardware ofrecido por la placa Arduino provee al usuario con un gran número de pines y periféricos que pueden ser utilizados para realizar diversos proyectos con una facilidad envidiable, tiene interoperabilidad con otros sistemas y placas, y además pueden incorporarse toda una serie de sensores que nos permitirán monitorizar el entorno.

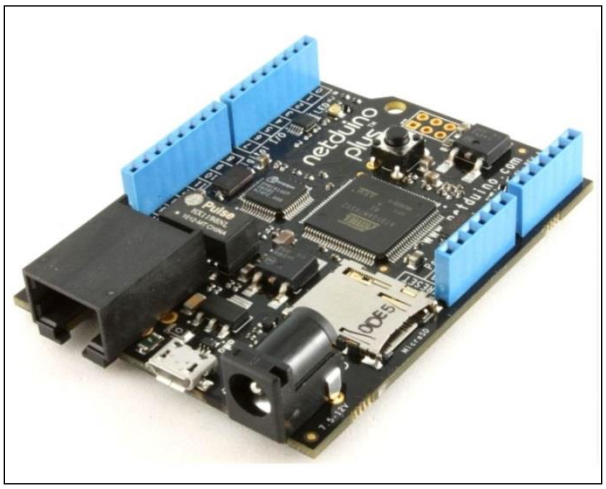
Además del Hardware, como hemos comentado, Arduino ofrece un Software que tiene una gran utilidad, nos permite interoperar con la placa de manera muy sencilla, intuitiva y rápida, en poco tiempo podremos hacer funcionar un proyecto sin quebrarnos la cabeza.

Vienen integradas un gran número de librerías que nos facilitan el trabajo y nos permiten operar con los elementos de la placa sin ninguna dificultad, puesto que vienen incluidos con el propio IDE.

Por último, Arduino tiene una gran escalabilidad y proyección de futuro con las placas añadidas que se le pueden incluir, conocidas como Shields. Se trata de módulos de Hardware que se adaptan a nuevas funcionalidades, y nos aumentan el potencial de nuestro propia placa. Estos Shields a su vez incluyen todo lo necesario para operar con ellos con extrema sencillez igual que lo hacemos con la propia placa Arduino. Algunos ejemplos de Shields son por ejemplo la Ethernet Shield que nos ofrece la posibilidad de comunicarnos vía Ethernet o la GSM Shield...

Comentar que aunque Arduino actualmente tiene placas muy limitados en "potencia", también posee placas con mayores especificaciones, como las placas basadas en microcontroladoras CortexM3 de ARM de 32 bits, que poseen las mismas funcionalidades y ventajas pero tienen mayor capacidad. [11]

Netduino



A partir de Arduino se generaron otro tipo de placas como por ejemplo la Netduino, que es también una plataforma opensource muy parecida a Arduino pero en vez de estar basada sobre C está basada en .NET Micro Framework. Posee como base un microcontrolador de 32 bits y un entorno de desarrollo bastante completo. [12]

Tiene más funcionalidades que Arduino, como por ejemplo GPS location, battery power control... y aún más pines I/O. Es compatible con algunas Arduino Shields.

PIC

Los PIC son microcontroladres puros que ofrecen unas funcionalidades y tienen una potencia de desarrollo mucho más grande que las placas de desarrollo que hemos nombrado anteriormente, que están limitadas por muchos factores, es un producto más especializado y no tan básico como PIC, al que tendríamos que añadirle todos los módulos necesarios para el funcionamiento con las conexiones y tecnologías que creyésemos conveniente, sin un IDE que nos sirva como apoyo y teniendo que tener unos conocimientos muy altos de electrónica y programación.

Sus ventajas son muchas y lo convierte en una opción óptima para el despliegue de grandes sistemas: precio muy bajo, infinidad de aplicaciones, muchos años siendo usado en el mundo de la electrónica, indudable escalabilidad, flexibilidad…

Pero todas estas ventajas implican una gran complejidad a la hora de desarrollar proyectos concretos sobre ella, lo que es una gran desventaja en este contexto.

En conclusión, podemos ver que hay grandes diferencias y similitudes entre todas las placas que hemos nombrado, por lo que para determinar qué placa es conveniente usar para este proyecto concreto, se ha recurrido a comparativas para conocer realmente cuáles son las especificaciones y limitaciones de cada una, de este modo, ha sido posible determinar que placa de desarrollo se adapta de manera conveniente a nuestras necesidades [13].

2.3 DDSI-RTPS

En los siguientes apartados vamos a comentar previamente qué es el protocolo RTPS, su relación directa con DDS a la hora de implementarlo y su estructura principal de funcionamiento.

La implementación que vamos a realizar estará guiada estrictamente por lo que nombra la última versión del estándar de OMG de DDS-RTPS, la versión 2.2 a la que haremos continua referencia. En el estándar se especifica que la comunicación puede ser tanto por TCP/UDP como por otros muchos medios, la implementación sugerida en el estándar es bajo UDP, e incluye el mapeado de la capa física a este protocolo, por lo tanto va a estar orientado hacia este protocolo IP a la hora de la implementación.

Posteriormente, una vez vistas las entrañas de RTPS y las características de su implementación, procederemos a elegir la plataforma de IoT para dicho fin.

2.3.1 DDS

El flujo de información mediante el esquema Pub/Sub es posible en DDS debido a los siguientes actores:

Los distribuidores de datos:

- Publisher: Se encarga de la distribución de datos. Puede publicar datos de diferente tipo.
- DataWriter: Es el objeto mediante el cual el protocolo DDS se comunica con un publisher, notificándole la existencia de nuevos datos de un determinado tipo. Cuando el publisher ha recibido estos datos se encarga de la distribución bajo unas condiciones, estipuladas mediante las condiciones de calidad de la comunicación, QoS. La publicación tal como es entendida en DDS es la asociación de un DataWriter a un publisher.

En la recepción de mensajes:

- Subscriber: Es una entidad que se encarga de la recepción de nuevos datos publicados y realizar la gestión necesaria para que llegue a la aplicación receptora. Igual que el publisher puede publicar datos de diferente tipo, obviamente el Subscriber debe ser capaz de recibir datos de diferente índole.
- DataReader: A nivel de aplicación es el objeto que accede a los datos recibidos por el subscriber. De manera análoga al concepto de publicación, la suscripción está definida como la asociación de un DataReader a un suscriber.

Por último es muy relevante el concepto de Topic, que está entre los conceptos de publicación y suscripción. Asocia un nombre único, en el propio dominio DDS bajo el que se trabaja, al tipo de datos y la QoS que necesita ese tipo de datos.

De esta forma tenemos una relación donde las condiciones de QoS del DataWriter asociado con un topic y las condiciones de QoS del Publisher asociado al DataWriter controlan el comportamiento de la publicación. Mientras que las condiciones de QoS de Topic, DataReader y Subscriber controlan el comportamiento de la comunicación en la parte de la suscripción.

2.3.2 RTPS sobre UDP/IP

Hasta ahora hemos comentado en numerosas ocasiones que el estándar incluye el mapeo para UDP/IP, vamos a ver que ventajas nos otorga y porqué se ha decidido implementar sobre este protocolo.

En primer lugar si hablamos de UDP hablamos de sistemas best-effort, concretamente el sistema de best-effort de UDP está muy bien enmarcado con los requisitos de Quality of Service de los flujos de datos en tiempo real. En caso de necesitar una conexión más comprometida que best-effort, es el protocolo RTPS quien se encarga de conseguir este tipo de comunicación con sus mecanismos descritos en la propia documentación.

En segundo lugar, al estar tratando con sistemas en muchas ocasiones, críticos en tiempo, o que deben operar bajo unas determinadas condiciones a priori desfavorables, necesitamos un protocolo que sea lo más simple y ligero posible con los mínimos costes y proporcionándonos el uso de la red IP, además, gracias a la comunicación connectionless, RTPS puede gestionar mejor los recursos de UDP, sockets y puertos.

Por último y para acabar, la funcionalidad de escalabilidad y multicast ofrecida nos permite una distribución de contenido muy eficiente teniendo en cuenta los tamaños de la red.

2.3.3 Encapsulación de datos

A lo largo de la descripción de las entrañas del protocolo RTPS, al realizar una implementación a un nivel tan bajo, hablaremos de la estructura de los datos, haremos uso de tipos de datos como bytes, long o incluso estructuras complejas, asimismo, al tratar con principios de orientación a objetos, necesitamos utilizar mecanismos que ayuden a esta flexibilidad y simplicidad en la estructuración de los datos.

Como veremos, la estructura de los mensajes que son intercambiados en este protocolo debe ser fija, y todo debe llegar en el orden correcto para que la interpretación sea la adecuada.

A raíz de ello es necesario definir cómo va a ser el "mapeo" de los datos, tener muy en cuenta términos como endianness, CDR, IDL, etc.

Veamos en primer lugar hasta dónde llega la jurisdicción del estándar respecto a la encapsulación de datos.

"Data encapsulation is not strictly part of the RTPS protocol… the RTPS protocol is agnostic to how the data in the SerializedData SubmessageElement is encapsulated. Instead, data encapsulation is the responsibility of the DDS type-plugin, which serializes and de-serializes the data." [5]**.**

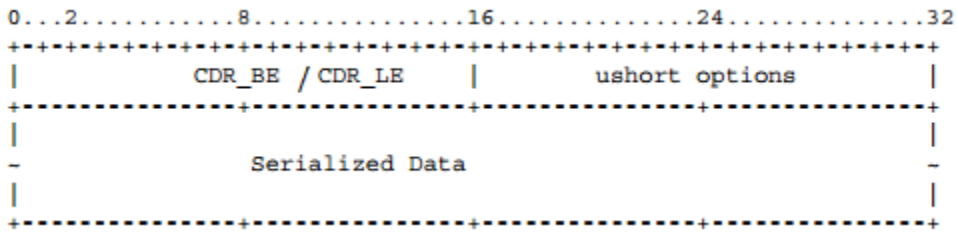
Es decir, la encapsulación de los datos no está especificada dentro del propio estándar de RTPS, pero sí lo especifica DDS, para conseguir interoperabilidad entre distintas implementaciones.

"For the purpose of interoperability, DDS implementations must support at least CDR encapsulation for application defined data types. The encapsulation of the data associated with built-in Topics must use a ParameterList" [5]**.**

Resumiendo, tenemos dos formas de encapsular los datos, o bien con la encapsulación CDR especificada por OMG y que tiene como base COBRA, o bien vía ParameterList, dependiendo del tráfico que estemos generando.

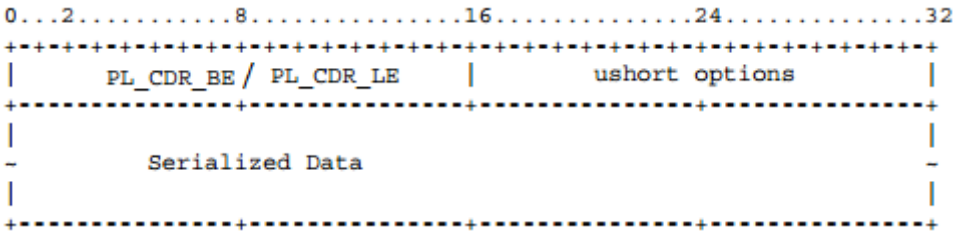
OMG CDR

Para enviar los datos serializados podemos especificar la opción CDR_BE para Big Endian o CDR_LE para Little Endian. Se suele utilizar para tráfico normal de RTPS, entre Endpoints simples.



ParameterList

Esta otra política de encapsulación de datos es utilizada para el metatráfico, se utiliza por ejemplo en el protocolo de descubrimiento cuando un Participant prefijado envía su información de descubrimiento. Incluye un marcador previo a los datos enviados que indica el campo de información que va a aparecer a continuación.



Es importante destacar que el estándar de RTPS de OMG establece una serie de estructuras de datos para su protocolo, hemos hecho uso de estas estructuras de datos en la medida de lo posible, para que el mapeo de los datos a la hora de enviar la información sea el correcto. El mapeo se encuentra en el Anexo 1, basado en el estándar, Table 9.4 - PSM mapping of the value types that appear on the wire. [5, pp. 154-158]

2.4 Arquitectura de RTPS

Este protocolo tiene una arquitectura definida dividida en cuatro diferentes módulos que controlan el intercambio de información entre las distintas aplicaciones de DDS.

Structure module: especifica los endpoints de RTPS y los relaciona con sus respectivos equivalentes en DDS.

Message module: especifica qué mensajes pueden intercambiar dichos endpoints y cómo se forman.

Behavior module: especifica las interacciones entre endpoints y cómo afectan los diferentes mensajes a éstos mismos.

Discovery module: especifica una serie de endpoints preestablecidos que nos permiten desarrollar el protocolo que nos descubre los endpoints de manera previa a la comunicación.

A continuación explicaremos lo imprescindible de cada uno de estos módulos para facilitar el entendimiento del resto del documento, para profundizar en el funcionamiento de RTPS es posible contrastar la información del propio estándar.

2.4.1 Structure Module

Como hemos comentado este módulo define los distintos endpoints. Al tratarse de un sistema publish-subscribe, RTPS encuentra grandes similitudes en los endpoints con DDS.

La estructura que presenta este protocolo no entraña extremada dificultad, tenemos una serie de RTPS Entities que están asociadas a un RTPS Domain, a su vez este dominio presenta una serie de RTPS Participants. Cada uno de estos dominios implica planos de comunicación diferentes.

Cada RTPS Participant contiene RTPS Endpoints. Hay dos tipos de endpoints, **Readers** y **Writers**.

Estos últimos son los que realizan la comunicación mediante paquetes RTPS.

Podemos ver esta relación en una figura bastante representativa:

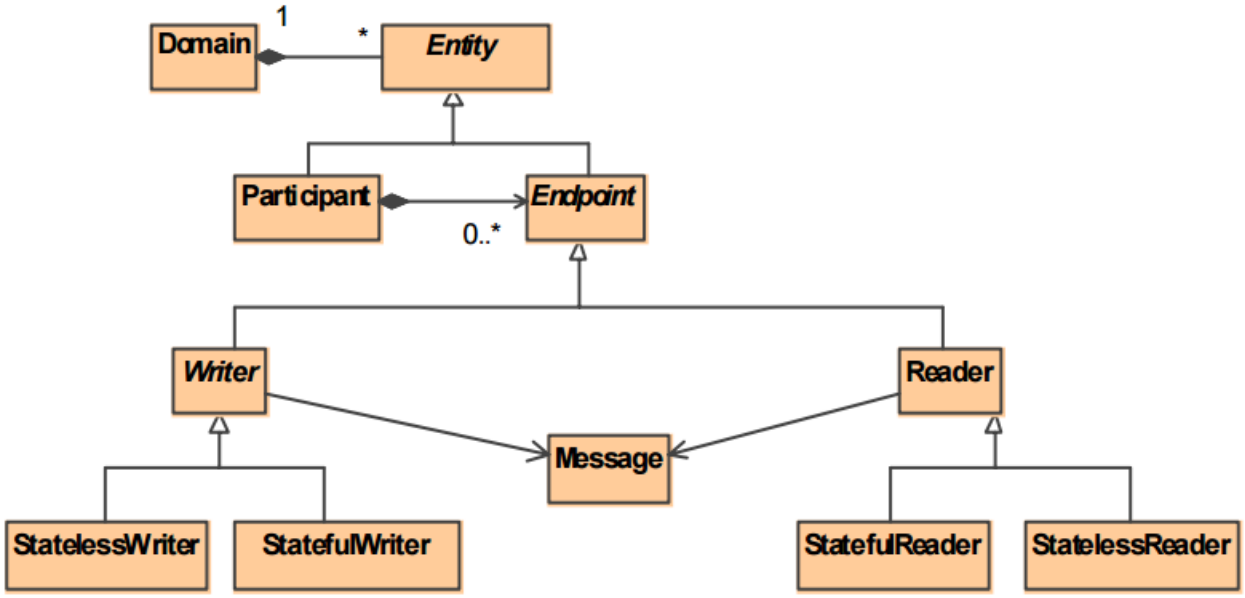


Figure 2.2. RTPS Structure [5, p. 8]

Veamos de manera más concreta qué es cada uno de estos conceptos y la relación de herencia entre ellos.

2.4.1.1 Clases utilizadas en la máquina virtual de RTPS.

Todas las entidades de RTPS derivan de la clase RTPS Entity.

Entity: Clase base para todas las entidades de RTPS. Representa la clase de objetos que son visibles a otros RTPS Entities en la red. Poseen un identificador único global (GUID).

Endpoint: Especialización de la clase anterior RTPS Entity. Representa los objetos que pueden ser puntos finales de comunicación, pueden enviar o recibir mensajes RTPS.

Participant: Se trata de una clase que contiene todos los RTPS Entities que comparten propiedades comunes y están en un mismo espacio de direccionamiento compartido. Tiene su análogo en DDS, DDS DomainParticipant.

Writer: Especialización esta vez de la clase anterior RTPS Endpoint. Representa los objetos que pueden ser fuentes de mensajes RTPS.

Reader: Especialización también de la clase RTPS Endpoint. Representa los objetos que pueden ser utilizados para la recepción de mensajes RTPS.

HistoryCache: Se trata de una clase utilizada de manera temporal como almacenamiento y tratamiento de los cambios efectuados en los datos. Está asociado o bien a un Writer o a un Reader, y dependiendo de en qué lado se encuentre tiene un tipo de uso. A continuación hablaremos más en profundidad de este concepto.

CacheChange: Representa un cambio de datos de un objeto. Es la unidad almacenada en HistoryCache.

Data: Representa los datos asociados a los cambios realizados sobre un objeto.

Cada uno de estas clases contiene una serie de atributos cuya configuración debe ser implementada. Será explicado en apartados posteriores.

2.4.1.2 GUID

Al describir de manera resumida las entidades que forman parte del protocolo RTPS hemos mencionado un concepto realmente importante, el GUID, veamos qué es y cómo está incluido en el protocolo RTPS.

Se trata de un identificador único global, globally unique identifier (GUID), este identificador es un número de 128 bytes (16 bytes/octetos), que cómo su nombre indica, nos ayuda a aplicar un identificador único a los equipos y redes con los que estamos trabajando.

Hay que matizar que no es realmente único, si no que la generación se realiza mediante un algoritmo especial que ha ido evolucionando y es tan improbable que se generen dos GUID's iguales que se considera prácticamente único.

El estándar inicial fue especificado por la Open Software Foundation (OSF) con el universally unique identifier (UUID), GUID es simplemente una adaptación e interpretación del estándar por parte de Microsoft.

Se utilizan 16 bytes divididos en 5 bloques con una determinada agrupación, un ejemplo de Guid es por ejemplo: {86fbde46-25a8-47e8-8085-c298be113d2e}. Este GUID ha sido generado por un generador online, pero la mayoría de herramientas de programación incluyen un generador aleatorio mediante algoritmos.

Aplicado a RTPS el GUID se encarga de identificar las entidades de RTPS. Como hemos visto se trata de un atributo de RTPS Entity e identifica esa entidad en un dominio de DDS. Está formado por un prefijo de GUID y un identificador de la entidad.

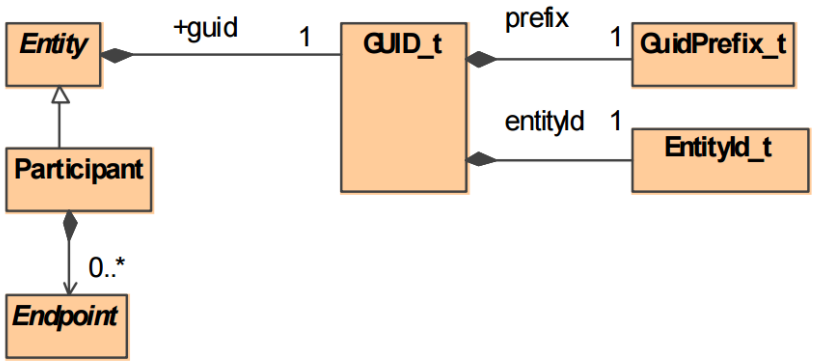


Figure 2.3. RTPS GUID [5, p. 20]

-Prefix: Identifica de manera única al Participant dentro del dominio de DDS.

-EntityId: Identifica de manera única al Entity dentro del dominio de DDS.

Como vemos tanto Participant como Entity tienen un GUID.

Para el RTPS Participant tenemos que su GUID está formado por el prefijo y una constante ENTITYID_PARTICIPANT, que se trata de un valor especial definido por el protocolo RTPS. Este valor depende de la implementación.

Para el RTPS Entity su GUID está formado por el mismo prefijo que el del Participant y un identificador del propio Entity. Esto tiene una serie de consecuencias:

- El GUID de todos los Endpoints dentro un mismo Participant tienen el mismo prefijo.
- Una vez conocido el GUID de un Endpoint, conocido también el GUID de su Participant.
- El GUID de cualquier endpoint puede ser deducido a partir del GUID del Participant al que pertenece y su identificador de entidad, entityId.

De igual modo que para el Participant, estos valores dependen de la implementación.

2.4.1.3 Relación DDS-RTPS y funcionamiento

Relacionando conceptos es realmente importante ver la relación directa entre DDS – RTPS para posteriormente entender cómo funciona RTPS en relación a DDS. Sobretodo la relación entre los endpoints y participants de RTPS con los de DDS, entre ellos encontraremos la puerta de comunicación entre sendos protocolos.

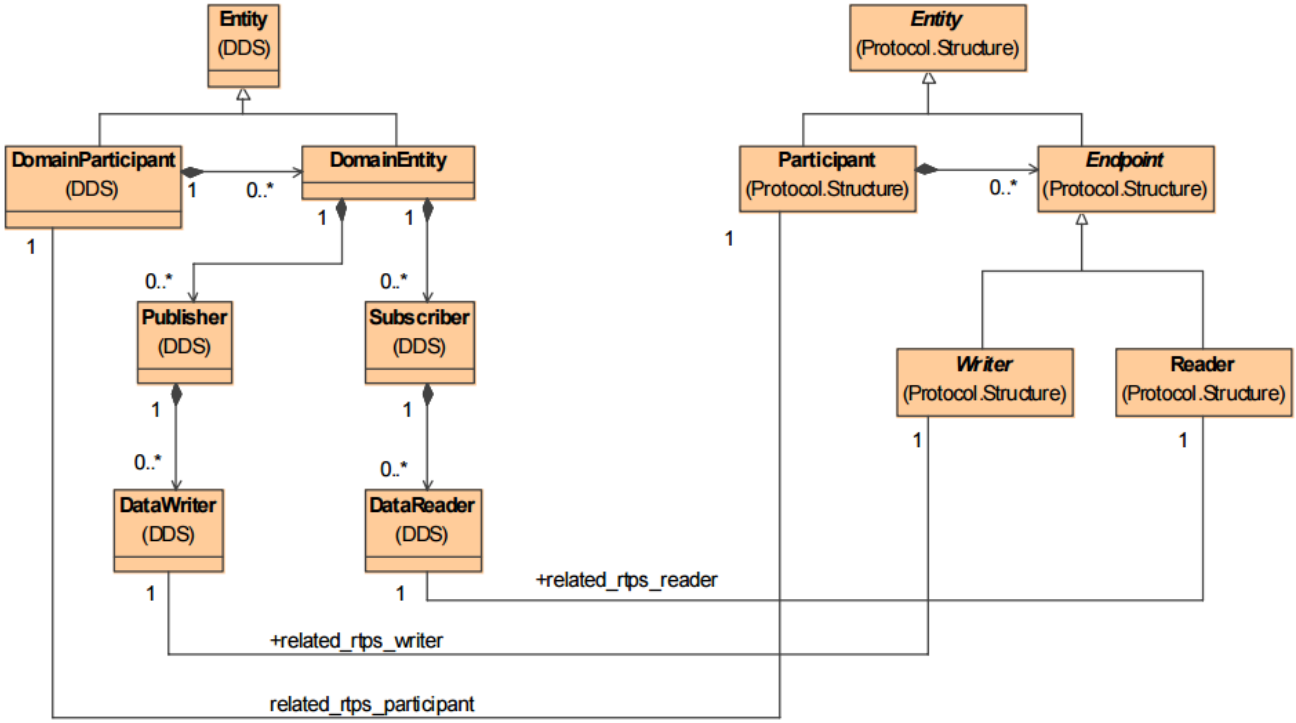


Figure 2.4. Correspondence between RTPS and DDS [5, p. 8]

Por último, y aunque no forme parte del módulo de estructura, pero para facilitar el entendimiento del resto del protocolo, a continuación se explica cómo funciona la relación DDS-RTPS, a través de qué entidad se gestiona y cómo se distribuyen los datos.

Hay una entidad muy importante que hemos mencionado previamente, HistoryCache, esta entidad es parte de la interfaz entre DDS y RTPS, contiene un historial parcial de los cambios en los datos de los objetos que han sido efectuados por su correspondiente DDS Writer y que necesitan ser emitidos a los RTPS Readers que se encuentran emparejados actualmente y aquellos que se unan en un futuro al sistema.

Este HistoryCache se encuentra presente tanto en el RTPS Writer como en el RTPS Reader, y juega un papel diferente dependiendo si estamos tratando con uno u otro.

Como hemos comentado en varias ocasiones es de gran relevancia el hecho de que DDS y RTPS tengan entidades análogas uno a uno, puesto que el funcionamiento está totalmente relacionado con este hecho.

Cada operación de escritura de un DDS Writer añade un nuevo CacheChange al HistoryCache del RTPS Writer asociado a dicho DDS Writer. Este RTPS Writer en consecuencia transfiere el CacheChange que acaba de ser añadido al HistoryCache de todos los RTPS Readers que estén actualmente emparejados con él en el sistema.

Por otro lado, en el receptor de este mensaje, el RTPS Reader, una vez tiene añadido este nuevo CacheChange a su HistoryCache, informa a su correspondiente DDS DataReader de los nuevos cambios que han llegado a su HistoryCache, y es ahora el DDS DataReader quien decide si acceder a esos datos.

La forma en la que las entidades de DDS interactúan con el HistoryCache no están especificadas en el propio estándar de RTPS y serán los clientes finales los que implementen esta funcionalidad.

La información que contenga el HistoryCache vendrá determinada según los criterios de QoS de DDS, y además debe cumplir los requisitos mínimos de interoperabilidad definidos en el Behavior Module.

2.4.2 Message Module

Este módulo centra sus esfuerzos en explicar simplemente cómo se forman los mensajes RTPS que intercambian Writers y Readers, su estructura y el contenido de éstos.

Cada mensaje tiene una estructura fija, formada por una cabecera del mensaje, y una serie de submensajes, cada uno con su cabecera y su contenido.

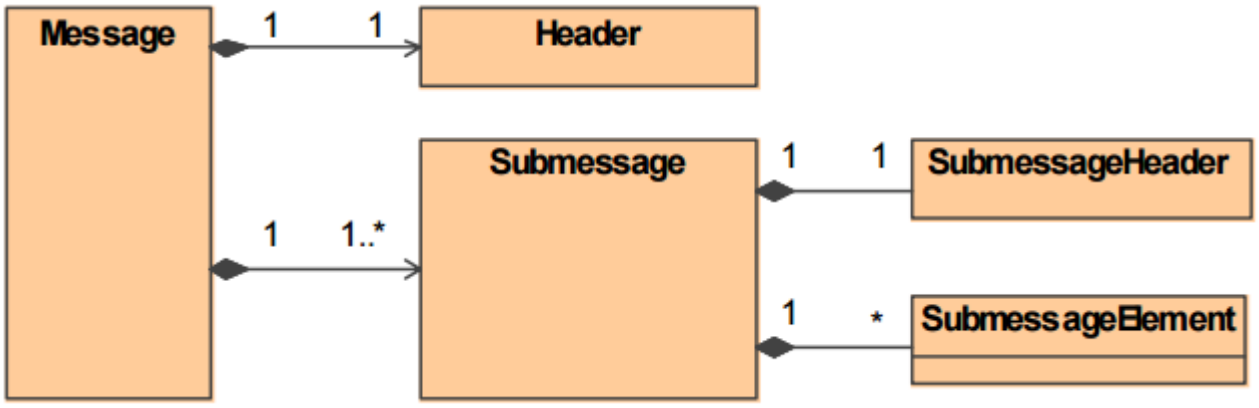


Figure 2.5. RTPS Message Structure [5, p. 9]

Es importante el hecho de que cada mensaje puede tener todos los submensajes que se crea conveniente, de esta forma se adapta el protocolo para todo tipo de dispositivos con mayor o menor memoria y ancho de banda.

La cabecera tiene una estructura definida, al igual que cada uno de los submensajes que forman parte el protocolo RTPS.

Los submensajes más importantes que comentaremos son los siguientes:

DATA: Este submensaje es enviado desde un Writer a un Reader con información asociada a cambios de datos del Writer. Puede añadir información o sobreescribirla, también hay otro submensaje que tiene la misma funcionalidad que DATA pero se encarga de enviarlo en paquetes fraccionados por si es necesario.

HEARTBEAT: De igual modo se trata de un submensaje enviado de Writer a Reader (como la mayoría), comunica los cambios que se encuentran disponibles en la Caché. Suele ir asociado a un submensaje de tipo DATA.

ACKNACK: Este submensaje también tiene gran importancia y es enviado en el sentido contrario, de Reader a Writer, sirve como en otros sistemas de comunicación para notificar acerca de los cambios de los que ha recibido información y acerca de cuáles no.

Como veremos más adelante es muy importante el hecho de que un mensaje RTPS tenga una estructura fija que hay que respetar para que los paquetes enviados y detectados por el endpoint remoto identifique un mensajes RTPS y pueda haber una comunicación entre los mismos.

Será en el apartado 3.4 de diseño donde veremos cómo está estructurada la parte genérica común, la cabecera de los mensajes, con su mapeo especificado en el estándar para el modelo específico de plataforma UDP/IP (PSM). En la figura Figure 2.6 podemos apreciar todos los submensajes existentes en el estándar.

Posteriormente veremos en detalle cómo se mapea también cada uno de estos submensajes que generaremos y se explicará en detalle los parámetros de los mensajes más importantes y cuya implementación es vital para el protocolo.

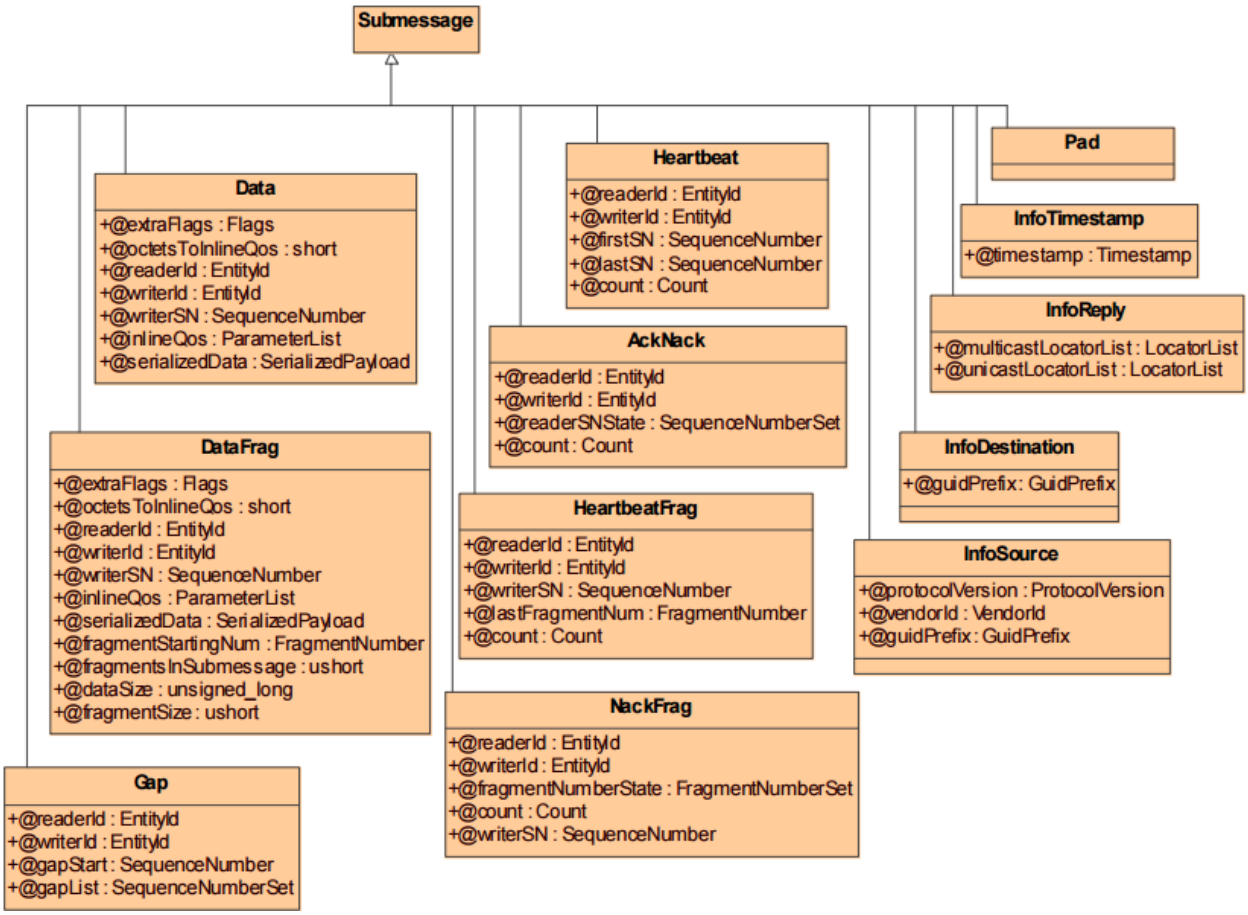


Figure 2.6. RTPS Submessages [5, p. 45]

2.4.3 Behavior Module

Este módulo describe las secuencias de mensajes que está permitido intercambiar entre RTPS Writers y RTPS Readers, así como los cambios que puede generar en cada uno de éstos los mensajes recibidos.

Este módulo especifica una serie de reglas mínimas que debe cumplir toda implementación de RTPS para poder cumplir los criterios de interoperabilidad. A su vez, ofrece distintas implementaciones en función de los requisitos del sistema, tratando de compensar o equilibrar el uso de la memoria, ancho de banda necesario y escalabilidad.

Como podemos apreciar en los esquemas anteriores en el Structure Module, tenemos dos tipos de Writers y de Readers, StatelessWriter/Reader y StatefulWriter/Reader, esta especialización de los conceptos Writer y Reader son las que realmente realizan la comunicación y están implementados, heredando del concepto de Writer y Reader. La principal diferencia entre éstos es la información que guardan los Endpoints de sus conexiones remotas.

En el caso del **Stateless** está optimizado para ser muy escalable y trabajar bajo unos bajos

requisitos de memoria, no mantiene información acerca del estado de los endpoints remotos, simplemente envía los mensajes RTPS. Es muy útil para sistemas muy grandes, aunque necesita unos mayores requisitos de ancho de banda. Esta implementación es ideal para trabajar bajo comunicaciones best-effort multicast.

Por otro lado la implementación **Stateful** mantiene información total acerca del estado de los endpoints remotos. Las ventajas y desventajas de esta implementación son opuestas a las del caso Stateless, un menor uso de ancho de banda pero aumento considerable de memoria y puede comprometer la escalabilidad. En contraposición esta implementación puede aplicar filtro en base a QoS o contenido, y además propone un esquema de comunicación fehaciente con muchas más funcionalidades y complejidad.

Ambas implementaciones son interoperables, la tendencia en un sistema real es tener una mezcla de ambos tipos dependiendo de los requisitos del sistema.

2.4.4 Discovery Module

Este módulo explica un apartado del protocolo RTPS realmente interesante, el protocolo de descubrimiento, que es extremadamente útil para DDS.

Los sistemas de datos distribuidos en tiempo real son muy dinámicos y cambiantes, por lo que este módulo se encarga de definir cómo modelar un protocolo de descubrimiento que consiga establecer la primera parte de la comunicación, el descubrimiento de las diferentes entidades que forman el sistema, teniendo en cuenta cuándo entran y salen del mismo.

Lo hace mediante dos protocolos diferentes, el Participant Discovery Protocol (PDP) y el Endpoint Discovery Protocol (EDP).

2.4.4.1 Participant Discovery Protocol

El dominio de RTPS está formado por diversos RTPS Participants, que a su vez incluyen varios RTPS Endpoints (Writers & Readers).

Esta subdivisión del protocolo es la encargada del descubrimiento entre los participantes, para posteriormente poder enlazar los Endpoints y llegar a la comunicación final.

Realmente la implementación mínima obligatoria es la del **Simple Particpant Discovery Protocol (SPDP)**, que funciona mediante metatráfico, un RTPS Participant prefabricado utiliza alguno de los RTPS Writers prefabricados que forman parte del Participant para enviar la información del propio participante dentro de un mensaje RTPS a una serie de puertos e IP's preestablecidos.

El envío de estos mensajes se realiza mediante RTPS Writers prefabricados, y de forma análoga para el caso de los RTPS Readers.

2.4.4.2 Endpoint Discovery Protocol

De forma análoga el estándar de RTPS simplemente especifica la mínima implementación para el correcto funcionamiento, al que se conoce como **Simple Endpoint Discovery Protocol (SEDP).**

Una vez realizado el descubrimiento de los otros participantes mediante metatráfico, el receptor del mensaje RTPS en el que está incluida la información del otro participante comienza el SEDP.

El mensaje recibido incluye toda la información relativa a los Endpoints que contiene el Participant emisor, por lo tanto simplemente basta con que envíe el RTPS Reader prefabricado su información a los Endpoints del emisor para que el protocolo de descubrimiento haya determinado completamente tanto el Participant como los Endpoints tanto de emisor como de receptor.

Técnicamente de manera más concreta a la realidad funciona como sigue:

Para funcionar el SPSP tenemos como base un Participant prefijado con los datos de: Guid, Protocolversion, VendorId, las direcciones IP y puertos por defecto de metatráfico (tanto UNICAST como MULTICAST) para comunicación de estos Participants prefijados, las direcciones IP y los puertos por defecto de los Endpoint finales, incluidos en el Participant (también UNICAST y MULTICAST) y por último los datos de las direcciones de los Endpoints que contiene, no por defecto si no las definidas que hay constancia de ellas (no es obligatoria este campo porque el tráfico se puede realizar mediante el default, aunque sea menos eficiente).

Este Participant prefijado envía un paquete de datos con toda su información a la dirección DEFAULT MULTICAST METATRAFFIC, y a partir de ahí pueden comunicarse por las otras vías para metatráfico, la unicast metatraffic u otras multicast que se definan etc.

Una vez hecho esto pasaríamos a utilizar la dirección por defecto para la comunicación entre endpoints, o bien coger las direcciones de estas mismas si están definidas para realizar finalmente el envío de los datos que nos interesan mediante mensajes DATA.

2.5 Elección de la plataforma de IoT

Una vez visto las principales diferencias y virtudes entre las distintas placas de desarrollo en el apartado 2.2, hemos decidido realizar la implementación bajo Arduino debido a las siguientes razones:

- En primer lugar y más importante, con ella tenemos capacidad de trabajar con IoT.
- Se trata de una placa de desarrollo de bajo consumo, que ofrece prestaciones que pueden adaptarse muy bien al protocolo que queremos implementar a un precio muy bajo.
- Su comunidad es muy amplia, lo que implica una posible futura cooperación y una proyección muy grande en trabajos futuros.
- La escalabilidad es absoluta, podemos incorporar nuevas funcionalidades gracias a su facilidad a la hora de añadir tanto Hardware como nuevo Software, ya sea con nuevos sensores para funcionalidades más simples o hasta nuevas tecnologías de comunicación utilizando las Shields oficiales y no oficiales de Arduino.

Una vez elegida la plataforma, cumpliendo el primer objetivo de los planteados en la sección 1.2, veamos las limitaciones que nos plantea, y entorno a las que tendremos que trabajar.

2.5.1 Arduino. Especificaciones técnicas. Limitaciones.

La placa de desarrollo con la que vamos a trabajar es la ARDUINO UNO, veamos sus especificaciones técnicas, bajo qué condiciones vamos a trabajar:

- Contiene 14 pines de entrada/salida, 6 de los cuales pueden configurarse como salida PWM, 6 entradas analógicas, reloj de 16 MHz, memoria FLASH de 32 KB, SRAM de 2KB y EEPROM de 1 KB. 8bit.
- Posee conexión USB, power jack puede ser conectado a una batería y con un adaptador AC-DC.

Adicionalmente, para nuestro cometido queremos realizar la conexión vía WiFi sobre UDP/IP, por lo tanto trabajaremos también con una WiFi Shield montada encima de la Arduino UNO, cuyas especificaciones técnicas son las siguientes:

- Posibilita la comunicación vía WiFi utilizando los protocolo WiFi 802.11b y 802.11g, y nos proporciona conectividad tanto UDP como TCP
- Puede trabajar en redes WiFi con encriptación WEP y WPA2 Personal.
- Contiene un micro-SD card slot, que puede ser utilizado para almacenar datos y es compatible con la placa Arduino UNO.

Ya conocidas las especificaciones técnicas, conocemos algunas de las limitaciones con las que tendremos que lidiar, como por ejemplo la baja memoria, pines limitados o una velocidad de procesado no muy alta.

Aunque no todo es el Hardware, el Software de Arduino es open-source, su Entorno de desarrollo integrado (IDE) facilita la escritura de código y la interacción del software con el hardware. Además funciona en los sistemas operativos más importantes, como Windows, Mac OS X y Linux.

El IDE está escrito en Java y la programación está basada en otros lenguajes de programación como Processing. También podemos utilizar otros programas con los que es más común trabajar, como Eclipse, que son más potentes y en determinados escenarios de programación pueden resultar ventajosos.

El IDE de Arduino ha ido evolucionando con nuevas versiones, comenzó con la versión 1.0 y actualmente para este proyecto vamos a utilizar la última versión del software que tiene soporte en la página oficial, la versión 1.6.5. Todo el código fuente relacionado con el software de Arduino se encuentra en GitHub.

Todo ello ha hecho que Arduino sea conocido y elegido por las opciones que ofrece su IDE, se trata de un entorno muy cómodo, junto con cada una de estas placas tendremos acceso a una serie de librerías que implementen las funcionalidades necesarias, por ejemplo con la placa Arduino UNO podemos hacer uso de librerías de comunicación serial, acceder y almacenar datos en una tarjeta SD, añadirle un LCD con el que interactuar o hacer uso del conversor digital-analógico.

Estas librerías también implican un gran número de limitaciones, asique hemos tenido que realizar una serie de pruebas para determinar la potencia y alcance de las librerías con las que vamos a trabajar, sobretodo la librería implementada de WiFi Shield para la comunicación UDP/IP. Estas pruebas se pueden contrastar en el apartado 3.2.1.1.

Tras varias pruebas hemos encontrado las siguientes limitaciones de la WiFi Shield:

- En las especificaciones técnicas vemos que podemos trabajar sobre redes WEP y WPA2, en cambio aún no es posible trabajar sobre redes WPA2 Enterprise, por lo que tendremos que tener en cuenta este hecho.
- Nos encontramos con una limitación muy grande en cuanto a buffer de datos que pueden ser enviados a través de un socket UDP según la librería integrada de WiFi Shield, tras realizar numerosas pruebas, actualización de firmware incluida, llegamos a la conclusión de que sólo es posible enviar paquetes de máximo 90 bytes, sin contar la cabecera UDP. Esto ha sido una limitación bastante dura que nos ha hecho tener que trabajar en torno a ella. Aunque a priori se pensó que no se podría implementar ni lo más básico, RTPS tiene en cuenta estos factores y aún con esta gran limitación se puede trabajar con el protocolo.

- Se han encontrado ciertas limitaciones también en términos de fiabilidad de la librería, no se trata de una implementación de UDP muy robusta, una frecuencia demasiado alta en el envío de paquetes vía UDP puede llegar a colapsar y Arduino deja de funcionar. [14] . Sin embargo pruebas realizadas para enviar paquetes con una frecuencia de 1 segundo no presentan complicaciones, se puede trabajar bajo estas circunstancias.
- En caso de querer resetear el WiFi Shield de manera programada, no hay manera de hacerlo, muchas de las funcionalidades incluidas en Arduino, como el WatchDog Timer no afectan al comportamiento de la WiFi Shield.
- No hay disponible una librería con soporte para plantillas (templates), extremadamente útiles para programación sin depender del tipo de datos con el que tratamos. Debemos tener muy en mente el IDL. El estudio de la inclusión de mecanismos de plantillas potentes en Arduino es un trabajo ajeno a nuestra implementación, pero tiene un gran interés puesto que tendría una influencia enorme en la librería. Hay algunas librerías realmente interesantes enfocadas a DDS y sistemas centrados en datos que podrían adaptarse, como la desarrollada por Sumant Tambe. [15]

2.5.2 Consecuencias de las limitaciones

Debido a estas limitaciones vamos a tener que limitar lo que podemos implementar del protocolo RTPS, esto nos compromete a:

- Es el cliente final quien debe implementar la serialización de los datos con los que desee trabajar al no poder hacer uso de plantillas. Se mostrará la implementación para tipos de datos sencillos a fin de presentar un producto final.
- Al tener que trabajar con un límite de 90 bytes, el tráfico que podemos generar se ve comprometido, de esta manera cada mensaje que mandemos tendrá un único submensaje, aunque la especificación de RTPS permita la inclusión de todos los que creamos convenientes.
- En consecuencia también de la anterior limitación, los datos serializados que podamos enviar en un paquete de tipo DATA también esta límitado a 52 bytes, descartando de los 90 bytes lo mínimo necesario de la cabecera.
- No es posible implementar el protocolo de comunicación. Como se verá a continuación en el apartado 3.6.

Arduino es un proyecto que tiene bastante movimiento y hay mucha gente involucrada en su desarrollo, por lo que es posible que en futuras actualizaciones del firmware de la WiFi Shield algunas de estas limitaciones se vean resueltas o almenos la capacidad del software aumente en

cierta medida, sobretodo el hecho de la limitación de 90 bytes (la limitación era aún mayor en firmwares anteriores).

Por ello aunque ahora mismo estemos bajo un marco de trabajo concreto, no debemos descartar ninguna implementación si no dejarla descrita para futuros trabajos de ampliación.

3 Diseño e Implementación

En este apartado será explicado el diseño y la implementación de las funcionalidades del protocolo RTPS que serán viables de implementar en la placa Arduino.

3.1 Claves del diseño

El propósito final se centra en implementar y probar las funcionalidades mínimas necesarias de RTPS sobre Arduino, concretamente el protocolo RTPS vía UDP sobre WiFi, explorando sus limitaciones, cumpliendo el segundo objetivo de los expuestos en el apartado 1.2.

- Generación de entidades de RTPS y sus funciones asociadas.

- Gestión de mensajes. Se divide en varios bloques:
    - Envío de los tres tipos de mensajes básicos de RTPS: Data, Heartbeat y Acknack.
    - Transmisión de objetos con poca información.
    - Transmisión de grandes bloques de información. No es el caso de IoT, así que su implementación no es necesaria.
    -
- Descubrimiento de equipos[1]
    - Mensajes para informar de la activación de dispositivos.
    - Resto de mensajes de descubrimiento. No obligatorios.

- XType. Añadir tipos de datos nuevos. No es necesario para aplicaciones IoT.

Estos han sido los principales objetivos de diseño, pero aún es necesario matizar las delimitaciones del diseño, las funcionalidades finales pueden presentar complejidad en la implementación.

Concretando, para la fase de envío de los diferentes mensajes de RTPS y la transmisión de objetos con poca información necesitamos:

- Generación de un publicador de información simple.
- Publicación de una serie de datos específicos por medio del publicador.

Adaptado al lenguaje del protocolo RTPS se trata de la implementación del RTPS StatelessWriter, que será también quien publique los datos al sistema PubSub.

[1] Como veremos más adelante, aunque se tratase de un objetivo inicial resultó no posible su implementacion. Apartado 3.5

Destacar que, aunque QoS sea una característica muy potente de DDS, no es viable de implementar debido a las limitaciones de la placa, ya que está ligada a la implementación Stateful y no forma parte de la implementación básica del protocolo, por ende no será implementada ni diseñada.

Por último, se busca realizar un escenario final en el que podamos publicar una serie de datos de interés al sistema que creamos que pueden ser útiles para un cliente.

La implementación consistirá en generar una librería para Arduino que dé soporte a las funcionalidades expuestas, sin haber ninguna otra librería previa realizada para este cometido, en ella se incluirá ejemplos donde publicará datos sencillos, y queda en manos del cliente final la adaptación del envío de otro tipo de datos a la librería.

3.1.1 Consideraciones respecto a la implementación

Para el cometido de la implementación del diseño en nuestra plataforma de IoT, Arduino, generaremos una librería de RTPS para dar soporte a la programación del código que se cargará en el dispositivo.

Cabe recordar que no se realizará la implementación de la librería que da soporte a la comunicación WiFi, si no que se utilizarán implementaciones previas debido a que es ajeno a este contexto.

La implementación de la librería de RTPS se realiza bajo el lenguaje de programación C++. Es un lenguaje de programación ampliamente utilizado, por lo que hay gran número de entornos de programación, en este caso debido a su simplicidad hemos utilizado CodeBlocks, aunque también habría sido posible utilizar otros IDE como Visual Studio o BloodShed Dev C++. Para mejor presentación se seguirán las recomendaciones de Arduino para la generación de librerías [16].

Por otro lado, la programación del código que se cargará en el propio Arduino se realizará sobre el propio IDE de Arduino, basado en Wiring y Processing, es en este código donde haremos uso de la librería generada.

Para realizar una comunicación bidireccional, o para que almenos haya un receptor de los mensajes RTPS, el escenario que se implementará será una comunicación vía WiFi entre el Arduino y un ordenador. Para generar este último escenario haremos uso de un programa que permita la comunicación, apertura de sockets y envío de información a una dirección IP.

El programa utilizado es realmente un sketchbook open-source llamado Processing, ha sido elegido debido a que la programación también se realiza bajo un lenguaje semejante al de Arduino y porque también posee una gran variedad de librerías, entre las que se encuentra una imprescindible para este escenario, basada en el envío utilizando UDP, de esta forma su implementación no genera más trabajo añadido.

La librería presenta un archivo de configuración (config.h) que podrá ser modificado por el cliente, en él se incluyen todos los valores prefijados que pueden ser variados, es responsabilidad del cliente el malfuncionamiento debido a la introducción de tipos de datos incorrectos.

Es importante destacar que las librerías tienen las mismas ventajas y limitaciones que si la implementación se realizase directamente sobre el IDE de Arduino, solo podemos hacer uso de las librerías de Arduino, y no las externas que podríamos utilizar en C++ como "standard input-output header", más conocida como stdio.h, u otras que nos podrían haber ahorrado bastante trabajo como las librerías “vector”, una clase contenedora muy útil, o incluso “string”.

3.2 Pasos previos a la implementación de la librería de RTPS

Si bien la implementación del protocolo RTPS es nuestra finalidad, no lo es la implementación de una librería que habilite la comunicación a nivel de capa física y capa de enlace.

3.2.1 Librerías alternativas para la comunicación

Por lo tanto tenemos que realizar un trabajo de investigación para encontrar una librería que nos ofrezca comunicación vía UDP como mínimo, aunque RTPS tolere también comunicación serial, TCP... para poder trabajar sobre ella, es decir, que se pueda invocar la transmisión o recepción de paquetes, sin importar el sistema de comunicación.

En primera instancia se trató de utilizar librerías ajenas a las de Arduino, tratando de buscar una que satisfaciese varios medios de comunicación, como el proyecto Apache MINA. [17]

Se trató de adaptar una librería muy potente realizada para OSC, puesto que nos proveía de compatibilidad con un gran número de protocolos de comunicación, EthernetUDP, Serial... [18]. Sin embargo, finalmente no fue posible utilizarla debido a que incluía cabeceras que arruinaban las posibilidades de realizar tráfico vía RTPS, puesto que lo que se pretendía enviar era ligeramente modificado añadiendo caracteres adicionales.

También se trató de adaptar las bases del protocolo de comunicación MaCaCo [19], con el que comparte grandes similitudes, pero el resultado fue el mismo.

Tras varios intentos fallidos, se tomó la decisión de utilizar la librería propia de Arduino para UDP WiFi, asumiendo las limitaciones.

3.2.1.1 Librería WiFi Arduino

A partir de este punto se optó por la utilización exclusiva de la librería WiFi de Arduino para la comunicación ocultando la capa de transporte.

Esta librería está totalmente documentada en la propia página de Arduino [11]. A continuación se verá de manera muy resumida las clases que serán de utilidad para la implementación.

WiFi Class

Esta clase es la base para inicializar la librería de Ethernet y configurar la red WiFi.

Mediante la función begin podremos conectarnos a las redes sin cifrado, con cifrado WEP o WPA2 Personal, y una vez conectados mantener la conexión o desconectarse para no generar tráfico.

WiFi UDP Class

Esta clase habilita la recepción y envío de mensajes UDP.

Se trata de una clase crucial para el proyecto, controla:

- El envío de información a las direcciones IP que deseemos, funcionando tanto para unicast como para broadcast/multicast. Se utilizará para el Writer
- La recepción de paquetes UDP en los sockets que se crea convenientes, permite la escucha en un puerto. Se utilizará para el Reader.

IP Address Class

Controla la gestión de IP's, será de utilidad porque el resto de funciones utilizan este tipo de objetos.

Por último tenemos las clases WiFi Server y WiFi Client, la primera para generar servidores que puedan recibir/enviar datos a clientes conectados y la segunda clase cuya finalidad es crear clientes para conectarse a servidores con el mismo fin.

Conexión a una red WiFi

El primer paso para tener una conexión válida es que todos los equipos que se quieran conectar a la red del sistema deben conectarse a la misma red WiFi.

Como hemos comentado se realiza mediante el uso de la clase WiFi y la función begin, tal como sigue:

```
char ssid[] = "MyWiFiNetwork";
char pass[] = "r63b4V4xpkwtQYJHUkJD";

// Connect to WPA2 Personal Network
status = WiFi.begin(ssid,pass);
```

Además hemos incluido impresión por pantalla del estado de la conexión a la red, obtendremos el siguiente texto:

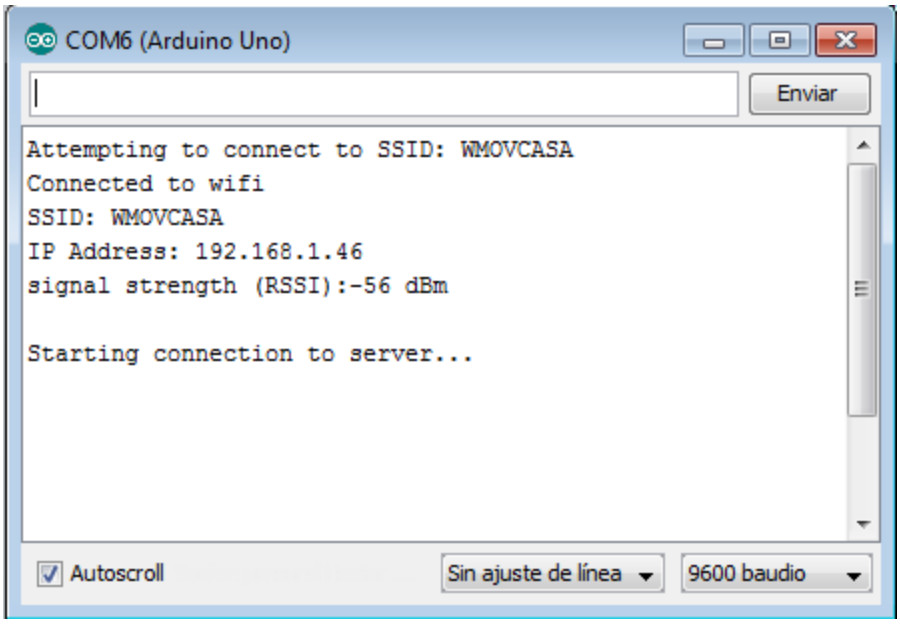


Figure 3.1. Arduino WiFi connection

Arduino incluye un ejemplo en la propia librería de la conexión a la red.

3.2.1.2 Envío de los primeros paquetes RTPS

El envío de los primeros paquete RTPS se realizó de manera muy rústica, pero se pudo llegar a la conclusión de que era posible el envío de paquetes de este tipo utilizando la librería de WiFi UDP, lo que supuso un gran avance.

La metodología utilizada fue el envío de una cadena de bytes de un paquete RTPS recuperado de otro sistema. Al abrir el archivo con un editor hexadecimal es posible recuperar y volver a generar la secuencia.

00000000	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	52	54	50	53	01	00	00	00	ac	10	00	80	00	d6	ab	02
00000010	07	03	18	00	00	00	00	00	00	00	01	c2	00	00	00	00
00000020	01	00	00	00	00	00	00	00	08	00	00	00	..	..	..	..

00000000	RTPS	....	¬	..	€	.	Ö	«	.							
00000010	.....	.....	Â	....												
00000020	.....	....														

A partir de aquí definimos la cadena en el propio programa que estemos realizando en Arduino.

```
unsigned char  RTPSMessage[] = {0x52,0x54,0x50,0x51,0x01,0x00,0x00,0x00,0xac,...};
```

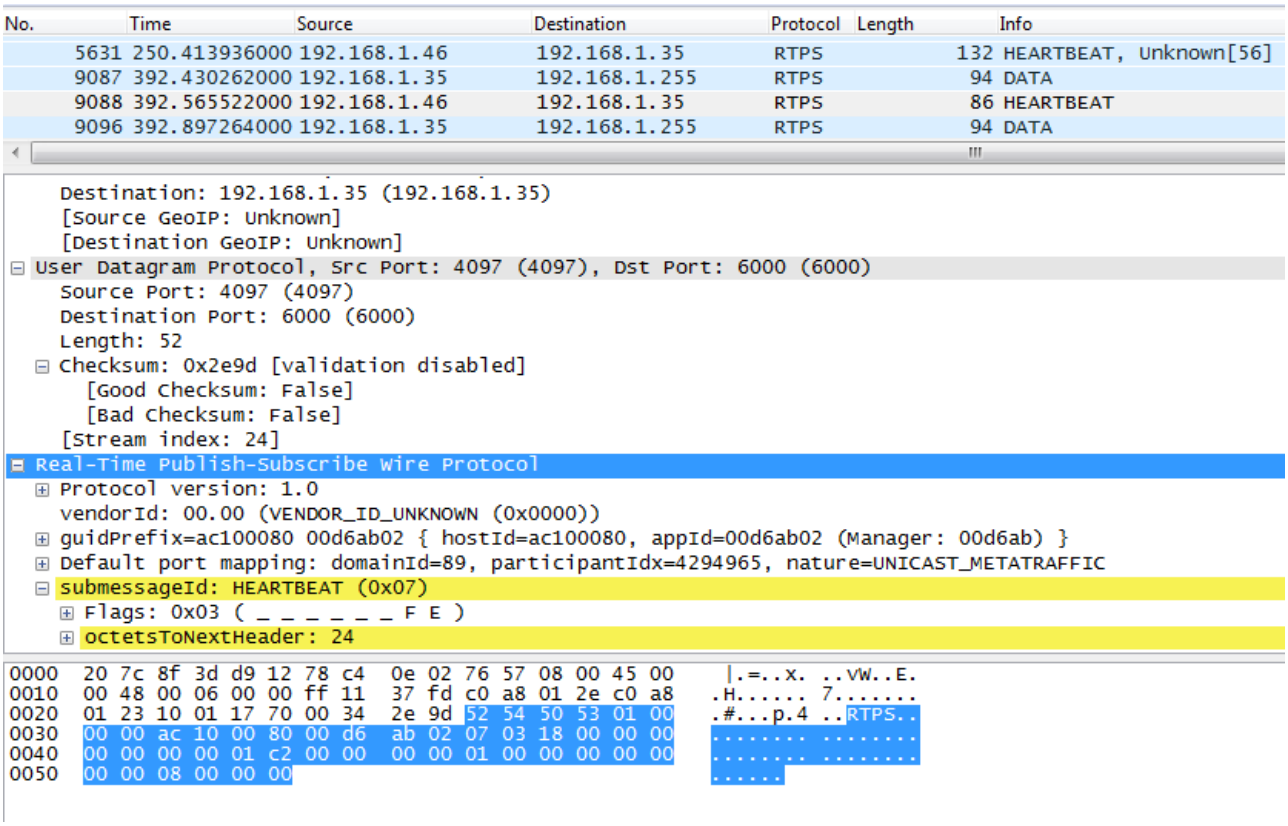
Cabe destacar que un unsigned char son 8 bits, lo que equivaldría a un octeto o un byte. El tipo de dato octeto no existe en Arduino, en cambio sí existe byte y es equivalente.

Damos por hecho el descubrimiento entre los dispositivos del sistema, dando por hecho que tienen conocimiento completo de toda la red.

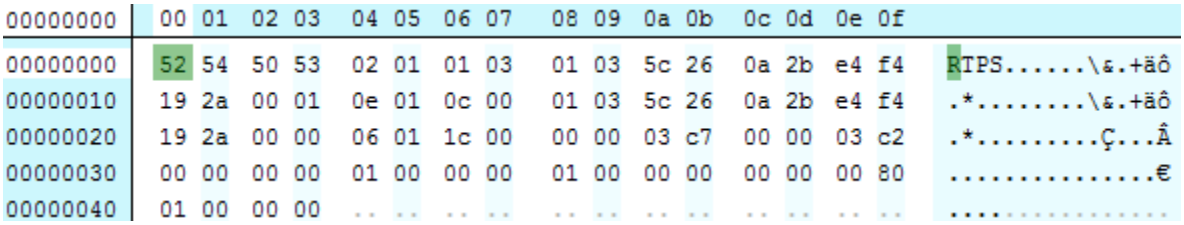
A continuación lo enviamos al otro Endpoint, en el caso de este sistema el envío de datos será del Arduino al ordenador.

```
Udp.beginPacket(remoteIP, remotePort);
Udp.write(RTPSMessage);
Udp.endPacket();
```

Finalmente comprobamos que el envío fue correcto, analizando todos los campos y el mapeo a hexadecimal apoyándonos en el estándar, se comparó con el análisis de Wireshark, confirmando la correcta transmisión.

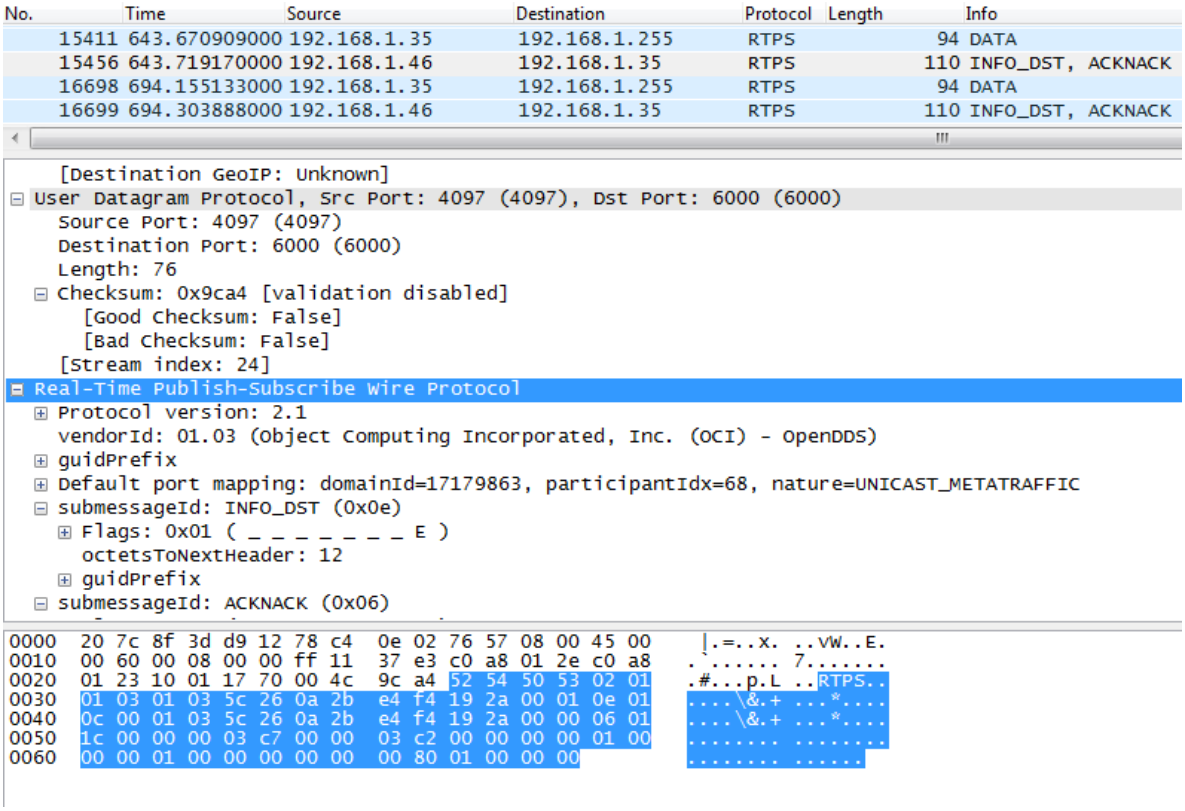


Para este otro paquete tenemos:



```
00000000   00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00000000   52 54 50 53  02 01 01 03  01 03 5c 26  0a 2b e4 f4   RTPS......\&.+äô
00000010   19 2a 00 01  0e 01 0c 00  01 03 5c 26  0a 2b e4 f4   .*........\&.+äô
00000020   19 2a 00 00  06 01 1c 00  00 00 03 c7  00 00 03 c2   .*.........Ç...Â
00000030   00 00 00 00  01 00 00 00  01 00 00 00  00 00 00 80   ...............€
00000040   01 00 00 00  .. .. .. ..  .. .. .. ..  .. .. .. ..   ................
```

Que mediante el mismo proceso podemos también capturar y comprobar cada uno de los campos:



```
No.   Time          Source          Destination     Protocol  Length  Info
15411 643.670909000 192.168.1.35    192.168.1.255   RTPS        94    DATA
15456 643.719170000 192.168.1.46    192.168.1.35    RTPS       110    INFO_DST, ACKNACK
16698 694.155133000 192.168.1.35    192.168.1.255   RTPS        94    DATA
16699 694.303888000 192.168.1.46    192.168.1.35    RTPS       110    INFO_DST, ACKNACK

    [Destination GeoIP: Unknown]
⊟ User Datagram Protocol, Src Port: 4097 (4097), Dst Port: 6000 (6000)
    Source Port: 4097 (4097)
    Destination Port: 6000 (6000)
    Length: 76
  ⊟ Checksum: 0x9ca4 [validation disabled]
      [Good Checksum: False]
      [Bad Checksum: False]
    [Stream index: 24]
⊟ Real-Time Publish-Subscribe Wire Protocol
  ⊞ Protocol version: 2.1
    vendorId: 01.03 (Object Computing Incorporated, Inc. (OCI) - OpenDDS)
  ⊞ guidPrefix
  ⊞ Default port mapping: domainId=17179863, participantIdx=68, nature=UNICAST_METATRAFFIC
  ⊟ submessageId: INFO_DST (0x0e)
    ⊞ Flags: 0x01 ( _ _ _ _ _ _ _ E )
      octetsToNextHeader: 12
    ⊞ guidPrefix
  ⊟ submessageId: ACKNACK (0x06)

0000  20 7c 8f 3d d9 12 78 c4  0e 02 76 57 08 00 45 00   |.=..x. ..vW..E.
0010  00 60 00 08 00 00 ff 11  37 e3 c0 a8 01 2e c0 a8   .`...... 7.......
0020  01 23 10 01 17 70 00 4c  9c a4 52 54 50 53 02 01   .#...p.L ..RTPS..
0030  01 03 01 03 5c 26 0a 2b  e4 f4 19 2a 00 01 0e 01   ....\&.+ ...*....
0040  0c 00 01 03 5c 26 0a 2b  e4 f4 19 2a 00 00 06 01   ....\&.+ ...*....
0050  1c 00 00 00 03 c7 00 00  03 c2 00 00 00 00 01 00   ........ ........
0060  00 00 01 00 00 00 00 00  00 80 01 00 00 00        ........ ......
```

Demostrado que es posible el envío de mensajes de tipo RTPS con esta librería, y la metodología necesaria, el siguiente objetivo es la creación de una librería que genere estos paquetes de manera dinámica según el estándar.

3.3 RTPS Entities

En primer lugar veamos cómo están definidas los atributos de los actores principales de RTPS, para ello necesitaremos hacer uso del sistema de herencia de RTPS, aunque ya fue nombrado de manera previa, ahondaremos en términos más técnicos. Todas estas entidades han sido implementadas en clases separadas de C++ con los atributos de cada entidad. Véase Anexo 1.

3.3.1 Actores principales

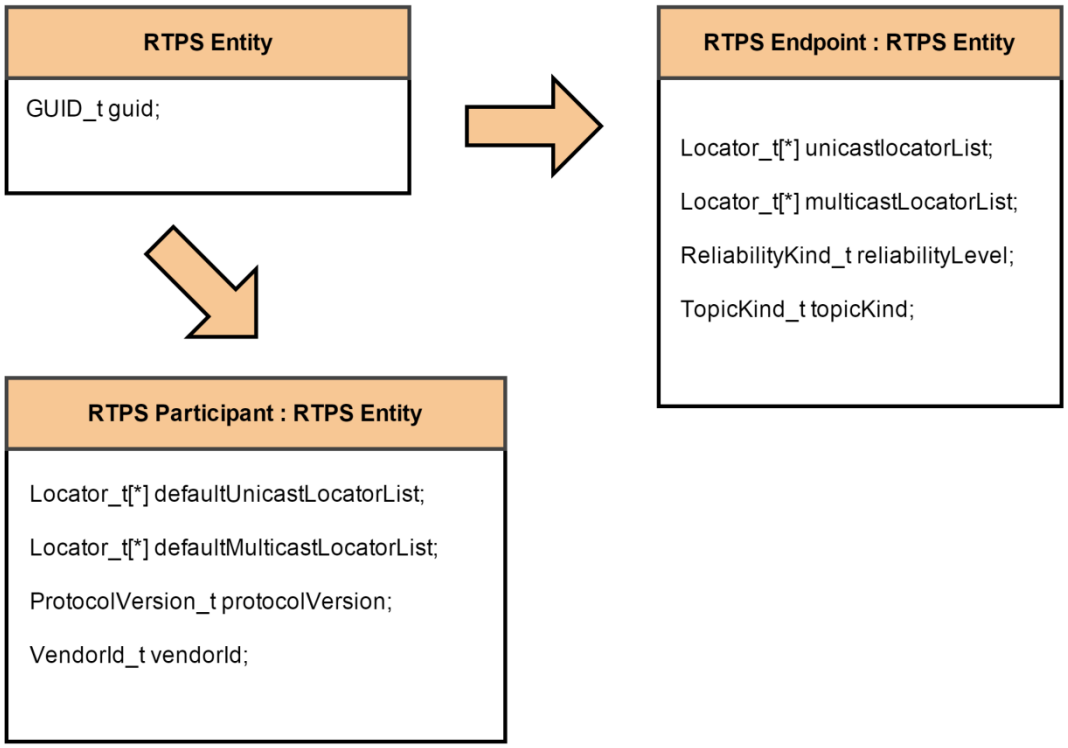


Figure 3.2. RTPS Entity Herency

Como podemos ver tenemos una clase-entidad raíz que es común a todas las demás, RTPS Entity, que posee el identificador único del que ya hablamos en el apartado 2.4. A partir de él heredan los conceptos de Participant, y Endpoint.

RTPS Participant

El Participant tiene como una de las funciones principales el descubrimiento de equipos del sistema, los campos que incluye esta entidad, a parte de la heredada Guid son las direcciones IP por defecto definidas por el estándar de OMG, la versión del protocolo que está siendo utilizada y la identificación del vendedor.

Al estar tan íntimamente ligado a la función de descubrimiento de equipos, hablaremos de la implementación de esta entidad de manera más extensa en el apartado 3.5.

Por otro lado, como bien sabemos, según lo establecido en el mismo apartado al que acabamos de hacer referencia, un Endpoint puede o bien ser un Writer o un Reader.

RTPS StatelessWriter

A continuación se procede con el desglose del concepto heredado de Endpoint, RTPS Writer:

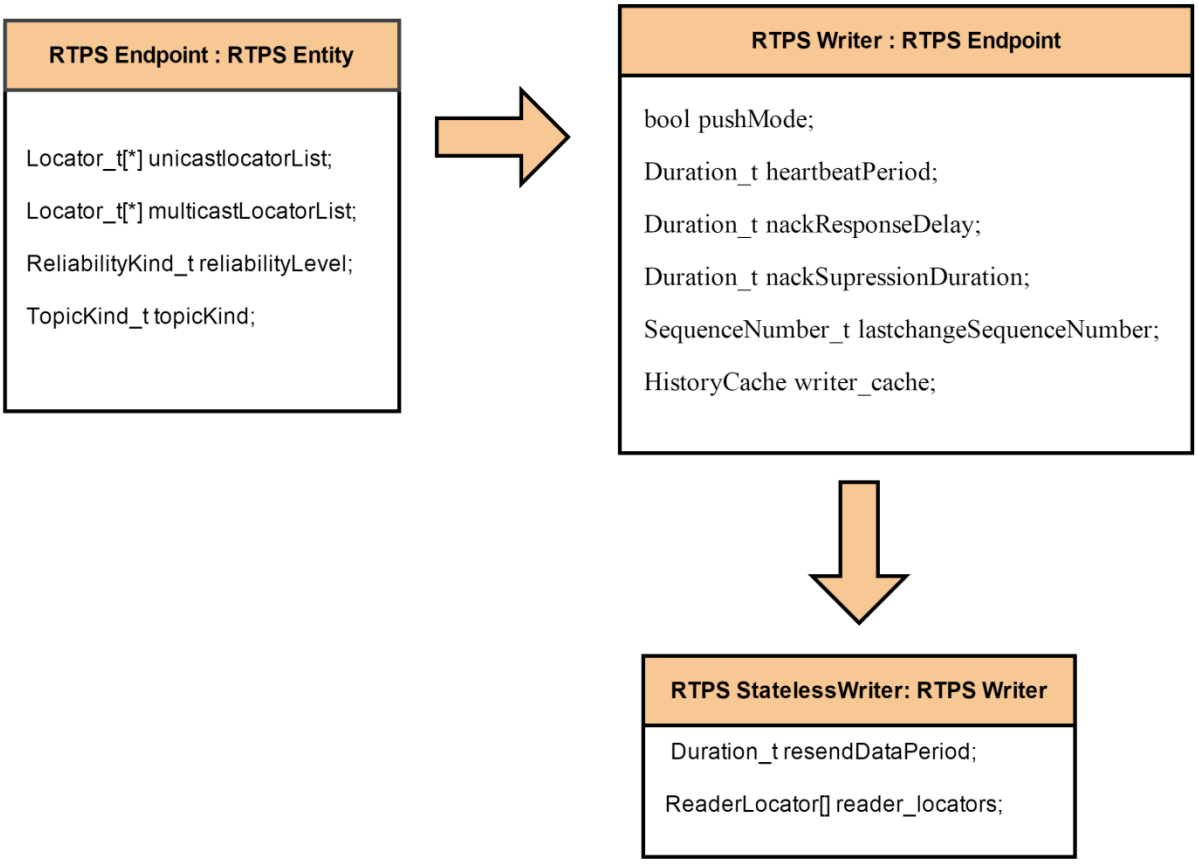


Figure 3.3. RTPS StatelessWriter

También comentamos con anterioridad que, tanto un Reader como Writer pueden seguir una implementación Stateless o Stateful en función de si tienen constancia del estado de sus endpoints remotos.

En el diseño solamente planteamos la posibilidad de realizar la implementación de las funcionalidades Stateless tal como están definidas en el estándar. [5, p. 83].

El diseño contempla la inclusión de todos los parámetros tanto de los RTPS StatelessWriter, como Reader como Participant, pero muchos de estos atributos no serán utilizados para la

implementación final. Aún así, es incluido para futuros trabajos.

En este caso, el **Writer** incluye una serie de campos que nombraremos:

- pushMode: Indica cómo actúa el Writer, en caso de que el valor sea TRUE, enviará los cambios en su caché a los Readers, en caso contrario, sólo anunciará los cambios y serán los Readers quienes deberán pedir el envío de los datos. En nuestro casó estará fijado a true puesto que siempre enviaremos datos.
- heartbeatPeriod: Indica cada cuanto tiempo enviará el Writer la información de su caché.
- nackResponseDelay: Indica cuanto puede retrasar el Writer el envío de los datos ante la recepción de un submensaje de tipo nack.
- nackSuppressionDuration: Indica a partir de que tiempo transcurrido desde el envío de un cambio se admite la recepción de nacks para dicho cambio.
- lastChangeSequenceNumber: Se trata de un contador interno que se utiliza para asignar el valor del número de secuencia a los cambios enviados.
- writer_cache: Contiene el historial de cambios en la caché para este Writer, la longitud de este historial dependerá de la implementación, siendo fijado a 1 cambio en esta versión.

Todos estos valores han sido fijado a su valor por defecto según el estándar, pero el cliente es libre de realizar modificaciones según crea conveniente en el archivo de configuración incluido.

Su clase heredada, **StatelessWriter**, posee estos atributos y además añade dos más:

- resendDataPeriod: Indica el tiempo tras el cual envía todos sus cambios a todos sus Readers asociados.
- reader_locators: Contiene una lista de todos los Readers a los que envía información. Su implementación no se contempla en esta versión.

Los campos heredados de Endpoint se verán con detalle en el protocolo de descubrimiento junto con el Participant, porque es su principal cometido es el SEDP.

En cuanto a las funciones incluidas en esta clase tenemos:

- La generación de un nuevo StatelessWriter, un constructor, al que se le pasan todos sus nuevos atributos por parámetro.
```
void newRTPSWriter(GUID_t wguid, Locator_t unicastLocator, Locator_t multicastLocator)
```
- La generación de un nuevo cambio que es añadido a su caché, en este caso dato de temperatura.
```
RTPS_CacheChange new_Change(float tempData);
```
- También gestiona el envío de los mensajes RTPS al puerto indicado de su lista de unicastLocator.
```
void sendDataPacket(int port, unsigned char* identif);
```

RTPS StatelessReader

Por otro lado veamos qué atributos y funciones posee el StatelessReader:

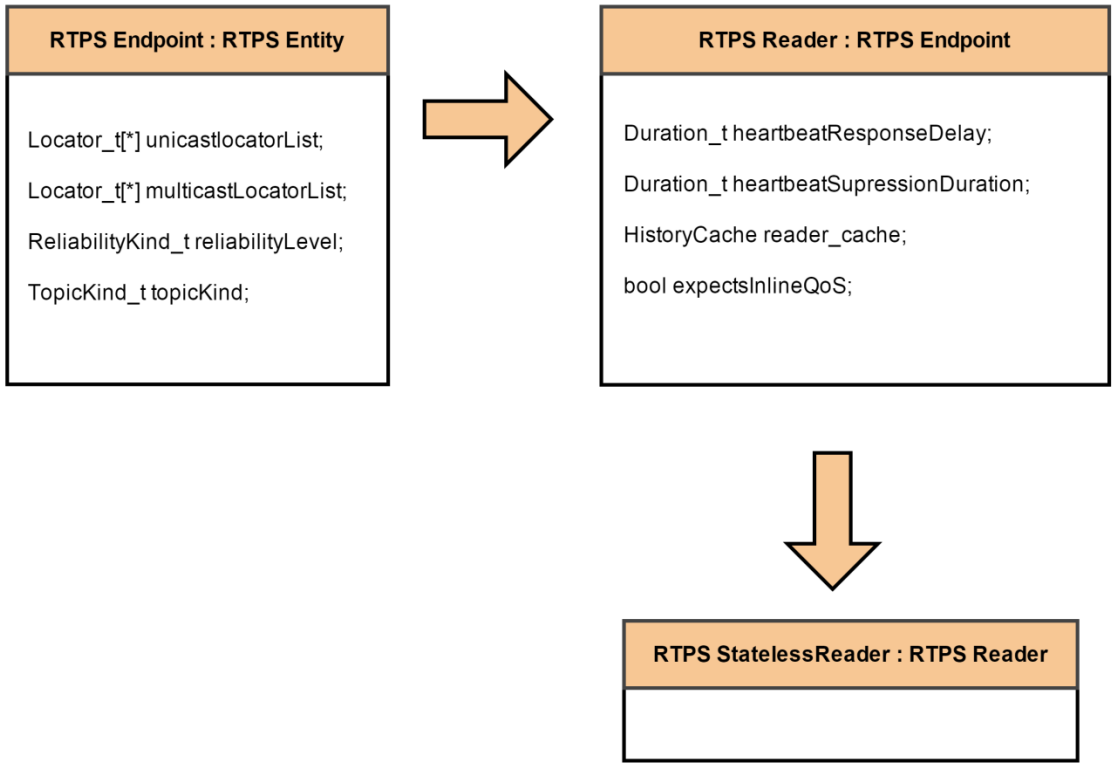


Figure 3.4. RTPS StatelessReader

El Reader hereda los atributos de Endpoint, que serán analizados en el apartado de Discovery Protocol, como ya comentamos.

Por otro lado en esta versión del protocolo RTPS, la versión 2.2, la clase heredada RTPS StatelessReader y RTPS Reader son exactamente iguales, no añade ningún atributo, por lo tanto simplemente explicaremos los atributos y funciones del RTPS Reader, que son muy similares a los del RTPS Writer.

- heartbeatResponseDelay: Indica cuánto puede retrasarse el Reader en el envío de un ack/nack en respuesta a un mensaje de heartbeat.
- heartbeatSupressionDuration: Indica a partir de qué espacio temporal desde la recepción de un mensaje de tipo heartbeat se puede recibir otro mensaje de este tipo.
- expectsInlineQoS: Indica si se espera parámetro de QoS, estará fijado a false.
- reader_cache: Contiene el historial de los cambios realizados en la caché. Para esta implementación al igual que en el Writer el tamaño está fijado a 1.

Los parámetros temporales heartbeatResponseDelay y heartbeatSupressionDuration tienen su valor fijado al definido por el estándar, pero puede ser modificado por el cliente a través del archivo de configuración.

Por otro lado esta clase según la definición del estándar incluye solamente el constructor entre sus funciones, y es el propio sistema DDS quien decide cómo y cuándo hacer uso de los datos que recibe un Reader.

El diseño e implementación de esta entidad ha sido siguiendo las instrucciones del estándar, sin embargo, con el propósito de poder interactuar con el reader, se han incluido una serie de funciones.

- receivePacket: Se encarga de gestionar la recepción de mensajes Udp y trata de analizar el paquete RTPS recibido. Su finalidad final es extraer los datos serializados.
```
void receivePacket(int localPort,unsigned char* packetBuffer);
```
- add_change: Añade un nuevo cambio en la caché a su historial.
```
void add_change(GUID_t wGUID,SequenceNumber_t w_seq_num,unsigned char* datatemp);
```

Aunque la implementación ha sido realizada, no ha sido posible hacer que funcione debido a la poca potencia de la librería de Arduino y la falta de memoria.

RTPS HistoryCache

Se trata de la clase contenedora de todos los cambios realizados en los objetos.
Tanto Writer como Reader poseen uno como acabamos de ver. Simplemente se trata de un array de CacheChanges, con las funciones necesarias para añadir y borrar cambios y para dar información de importancia para el protocolo, como son los números de secuencia.

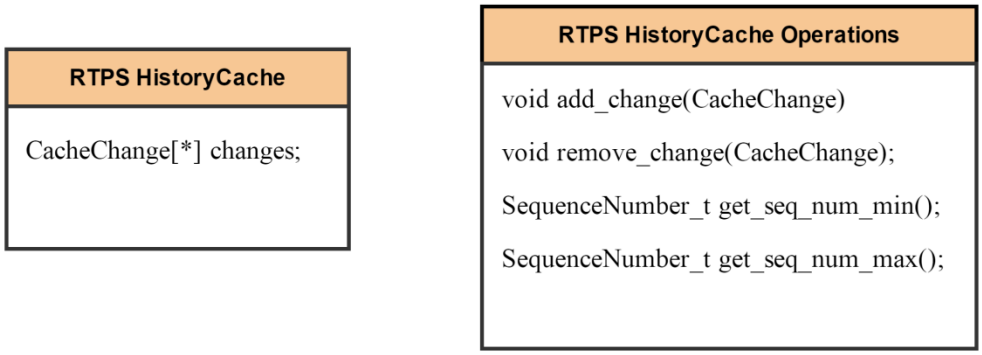


Figure 3.5. RTPS HistoryCache

RTPS CacheChange

Continuando el desglose de los actores de RTPS, llegamos a la expresión más pequeña que analizamos.

Se trata de una clase que genera los objetos de los cambios producidos, es decir, contiene los datos, número de secuencia, tipo de cambio, el GUID del writer, etcétera.

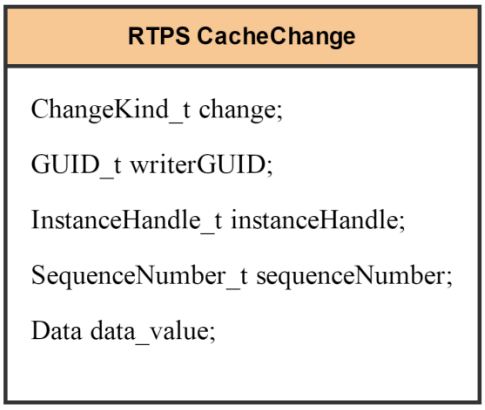


Figure 3.6. RTPS CacheChange

La interconexión entre todos estos conceptos para el funcionamiento del protocolo y la implementación de la funcionalidad final se verá en el apartado 3.6.

3.3.2 Guid

Se trata de un concepto bastante importante para poder identificar de manera única cada uno uno de los dispositivos y entidades del protocolo dentro de nuestra red.

Arduino no incluye una librería que añada la posibilidad de generar nuevos GUID/UUID, por ello hemos barajado diversas opciones para su implementación:

1. Inclusión de una lista realmente grande de GUID's en el propio código a partir de la cual se coja un número aleatorio probablemente único.
2. Inclusión de la misma lista en un medio de almacenamiento externo.
3. Acceso a un GUID único aleatorio vía Internet.
4. GUID fijado por el propio cliente a la hora de compilar.

Entre todas estas opciones se han ido descartando las que no son posibles de implementar o presentan grandes desventajas o limitaciones.

La opción 1 ha sido totalmente descartada debido a la capacidad de memoria del dispositivo.

La opción 2 ha sido también descartada, debido a que podría causar un gran delay el acceso a la memoria externa. Si bien es cierto que la propia placa WiFi Shield tiene un compartimento dedicado para una tarjeta SD, su implementación es compleja y a priori no parece eficiente. Puede ser posible su implementación en futuras versiones.

La opción 3 fue implementada, al estar en conexión mediante un protocolo WiFi, en algunos escenarios sería posible la conexión a Internet. Por lo tanto, para su implementación, se generó una página web gratuita con capacidad de código HTML y PHP, http://guidmicrocontroller.zz.mu/guid.php la cual genera un código GUID nuevo mediante el mecanismo pseudoaleatorio definido por el estándar de GUID (esta función forma parte del propio lenguaje).

```
<guid>7CA1C485-38A6-42B1-A661-0ED6B4829880</guid>
```

Mediante el WiFi Client nos conectamos como cliente a este servidor y extraemos el valor.
Finalmente esta opción también fue descartada debido a que restringe el escenario a la conexión a Internet, además de tener un gran delay.

Finalmente la opción adoptada es la 4, es el propio cliente quien da el valor del GUID a la entidad que genere a la hora de compilar.

3.4 RTPS Messages

En este apartado explicaremos cómo están estructurados los submensajes más importantes y cómo se ha realizado el mapeo de los datos a los mensajes, qué valor tiene cada campo. Cada uno de los diferentes tipos de mensaje que utilizaremos para la generación dinámica está implementado en clases distintas.

Toda la implementación de atributos tanto de mensajes como de las propias entidades utiliza el Common Data Representation (CDR) de OMG, el cual será utilizado en la medida de lo posible. Véase Anexo 1.

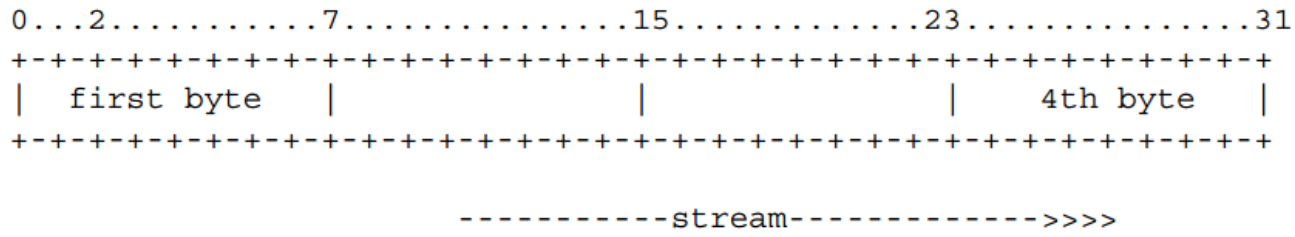
3.4.1 Message Mapping

En primer lugar comentar que el estándar utiliza un tipo de notación gráfica para explicar el mapeo muy simple, donde:

```
+-+-+-+-+-+-+-+-+
|7|6|5|4|3|2|1|0|
+-+-+-+-+-+-+-+-+
```

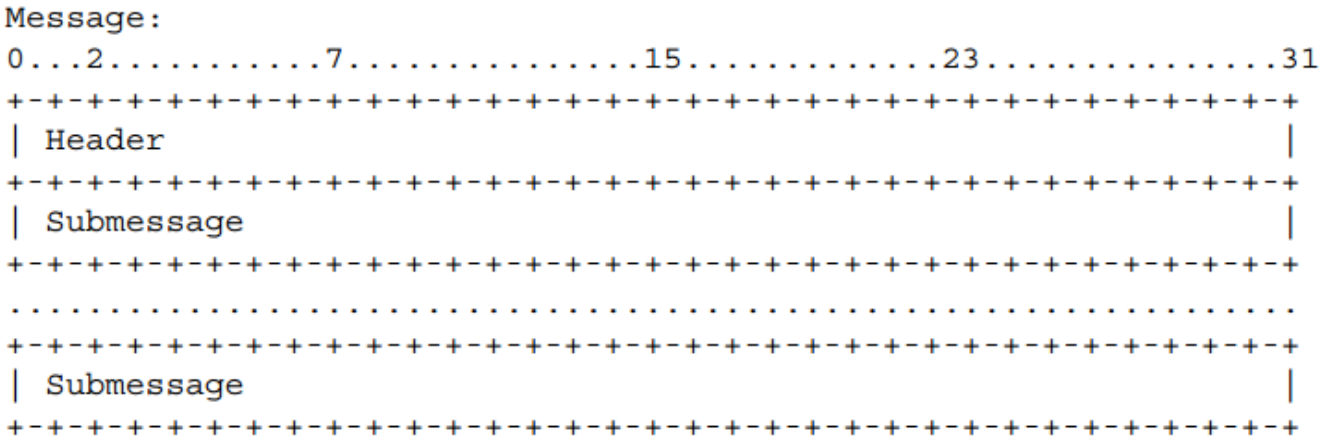
Representa un byte u octeto. El bit 7 representa el “most significant bit” (MSB) y el 0 el “less significant bit” (LSB).

Los flujos de bytes están representados de esta manera:



En este documento nos adaptaremos a la representación del estándar para que sea más sencillo el entendimiento a la hora de consultar el estándar.

Ahora sí estableciendo estas bases explicaremos cómo está definida la estructura de cada mensaje RTPS y los elementos de la cabecera.



Por otro lado la cabecera de cada mensaje RTPS consta de 4 campos:

-Protocol: Identifica el mensaje como un mensaje RTPS.

-Version: Identifica la versión del protocolo (existe interoperabilidad entre versiones).

-VendorId: Indica el vendor que provee la implementación de RTPS en uso.

-GuidPrefix: Define un prefijo por defecto que puede ser utilizado para reconstruir el GUID y que es común para todos los submensajes, de esta forma se ahorra repetirlo con motivos de no sobrecarga.

Veamos una representación gráfica de cómo está ordenado y cuánto ocupa cada campo:

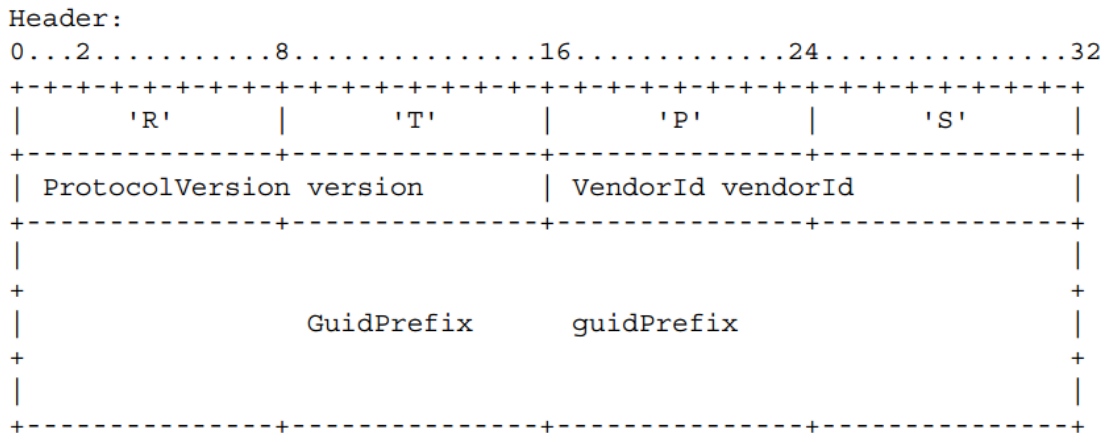


Figure 3.7. Header mapping

Es realmente importante respetar la estructura general, y además teniendo una visión global de lo que implica el proyecto, la implementación en una placa de desarrollo concreta, se ha debido realizar un estudio previo para comprobar si se respeta totalmente la estructura en el envío de mensajes, lo que se quiera enviar debe ser lo que llegue al receptor, y el sistema de comunicación no puede incluir ningún tipo de cabecera o datos a nuestro mensaje.

El valor de estos campos es estático:

- El campo de protocolo es RTPS, expresado en 4 bytes hexadecimales: {0x52,0x54,0x50,0x53}
- El campo de versión tiene el valor 2.2, la versión más reciente. Su valor está definido según las estructuras de datos de CDR de OMG. (Anexo 1)
- El campo vendorId tiene el valor 5.5, definido por el creador para identificación propia.
- Por último el valor del campo guidPrefix está definido según varias opciones y depende del diseño utilizado, explicadas en el apartado 3.3.2.

3.4.2 Submessage Mapping

En adelante centraremos los esfuerzos en explicar la estructura de los submensajes.

Cada submensaje posee una cabecera y una serie de elementos propios de cada submensaje, veamos la cabecera común y qué variaciones presenta cada uno de los tipos de submensaje más relevantes del protocolo.

Esta cabecera, al igual que la cabecera del mensaje en sí, también tiene una estructura fija que debe ser respetada. Posee una serie de campos comunes:

- SubmessageId: Identifica el tipo de mensaje, Data, Heartbeat,AckNack, etcétera. El valor de este octeto será el correspondiente al submensaje según la tabla Figure 3.8. Submessage ID.
- Flags: Todos los submensajes poseen este campo, pero su contenido varía en función del submensaje, por eso su valor será determinado de manera separada para cada caso.
- OctetsToNextHeader: Indica cuántos bytes/octetos restan para llegar al inicio del siguiente submensaje, de esta forma el receptor podrá omitir mensajes si es necesario. Este valor está fijado a 0 al trabajar con solo un submensaje por mensaje.

El resto del contenido del submensaje varía en función del tipo.

```
PAD              = 0x01, /* Pad */
ACKNACK          = 0x06, /* AckNack */
HEARTBEAT        = 0x07, /* Heartbeat */
GAP              = 0x08, /* Gap */
INFO_TS          = 0x09, /* InfoTimestamp */
INFO_SRC         = 0x0c, /* InfoSource */
INFO_REPLY_IP4   = 0x0d, /* InfoReplyIp4 */
INFO_DST         = 0x0e, /* InfoDestination */
INFO_REPLY       = 0x0f, /* InfoReply */
NACK_FRAG        = 0x12, /* NackFrag */
HEARTBEAT_FRAG   = 0x13, /* HeartbeatFrag */
DATA             = 0x15, /* Data */
DATA_FRAG        = 0x16, /* DataFrag */
```

Figure 3.8. Submessage ID [5, p. 168]

Data Submessage

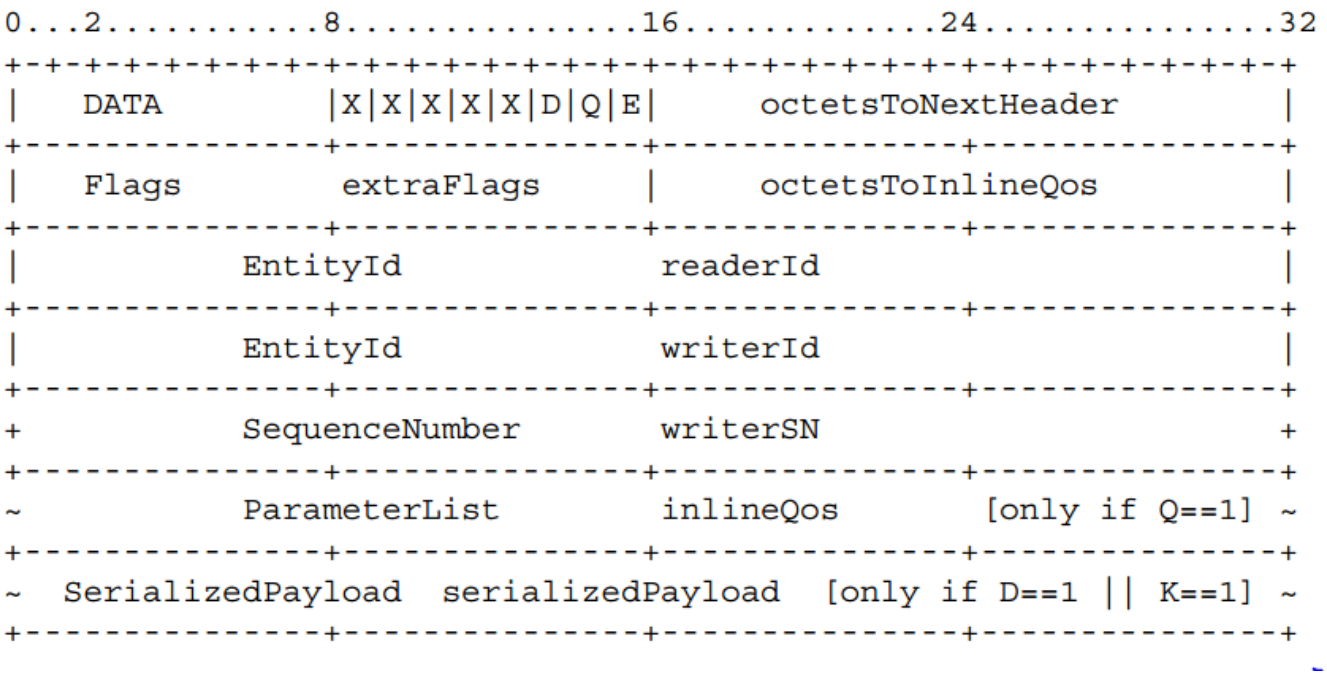


Figure 3.9. Data Submessage mapping [5, p. 170]

Esta es la estructura de un submensaje de tipo DATA.

Este submensaje es enviado desde un RTPS Writer a un RTPS Reader. Notifica al destinatario de algún cambio de datos realizado en algún objeto relativo al RTPS Writer. Los cambios pueden ser o bien del propio valor del dato del objeto o algún otro tipo de información.

Estos cambios se comunicación mediante el campo serializedPayload del submensaje.

En cuanto a su estructura, tiene los elementos de la cabecera de submensaje; el tipo de submensaje en primera instancia ocupando un octeto, e indicando que es de tipo DATA, los flags correspondientes, que como podemos observar varía dependiendo del tipo de submensaje, y por último el campo de octetsToNextHeader indicando cuantos octetos restan para llegar al siguiente submensaje.

Por otro lado tenemos los elementos del submensaje, que son específicos del tipo, tenemos:

- extraFlags: Se trata de un campo que ofrece indicadores adicionales a los que aparecen siempre en la cabecera del submensaje. Estos nuevos bits adicionales de indicadores serán útiles en versiones futuras para añadir nuevas funcionalidades sin comprometer la compatibilidad entre protocolos diferentes. En esta versión del protocolo el campo permanece a valor 0.
- octetsToInlineQos: Al igual que el campo de octetsToNextHeader nos indica los octetos restantes, esta vez, hasta el campo inlineQos del mismo submensaje. También se utiliza

para la compatibilidad entre distintos protocolos porque este campo debe ser omitido o saltado en caso de que el receptor crea conveniente que no puede interpretarlo o él no es el destinatario. Al no implementar QoS nos indicará el valor hasta el campo de datos, este valor estará fijado a 16, contando el número de octetos de la cabecera.

- readerId: Indica el identificador del reader destinatario al que va destinado el mensaje, en caso de no estar destinado a un reader concreto se pone el valor por defecto, de este modo el mensaje tiene como destinatario todos los Readers con el mismo prefijo de GUID, es decir, dentro del mismo Participant.. Su valor estará fijado por el cliente.
- writerId: De manera análoga indica el identificador del writer que envía el mensaje. Su valor estará fijado por el cliente.
- writerSN: Este campo indica el SequenceNumber, el número de secuencia del submensaje, este valor se incrementa cada vez que se envía un nuevo submensaje. Es útil para los entornos en los que el reader realice la petición de algún submensaje en concreto que no tenga en su caché. Su valor comenzará en 0 y será aumentado de manera sucesiva en cada nuevo cachechange.
- inlineQos: Este campo provee los medios necesarios para que los submensajes de este tipo puedan incluir políticas de QoS. Estas opciones de QoS son encapsuladas mediante ParameterList. No haremos uso de este campo al no implementar QoS.
- serializedPayload: Está presente en función de los indicadores de la cabecera, que veremos a continuación. Estos datos van encapsulados y pueden contener el valor de los datos asociados a un cambio o la clave del objeto al que hace referencia. A la hora de generar los datos a envíar se generar el paquete de forma dinámica con los datos. Hablaremos del tipo de dato más adelante.

Respecto a los flags:

- EndiannessFlag: Aparece en la cabecera del submensaje, indica el endianness. Un bit a 1 indica el uso de Big-endian, 0 en caso de utilizar Little-endian
- InlineQosFlag: De igual modo aparece en los indicadores del submensaje e indica al receptor (Reader) de la presencia del campo inlineQos, que contiene los parámetros de QoS que debe tener en cuenta. Su valor estará fijado a 0 al no implementar QoS.
- DataFlag: Este indicador indica al Reader que el submensaje contiene el valor serializado de un objeto de datos. Su valor estará fijado a 1 al enviar datos.
- KeyFlag: Este indicador indica al Reader que el submensaje contiene el valor serializado de la clave del objeto de datos. Su valor estará fijado a 0 puesto que no implementamos dicha funcionalidad.

Heartbeat Submessage

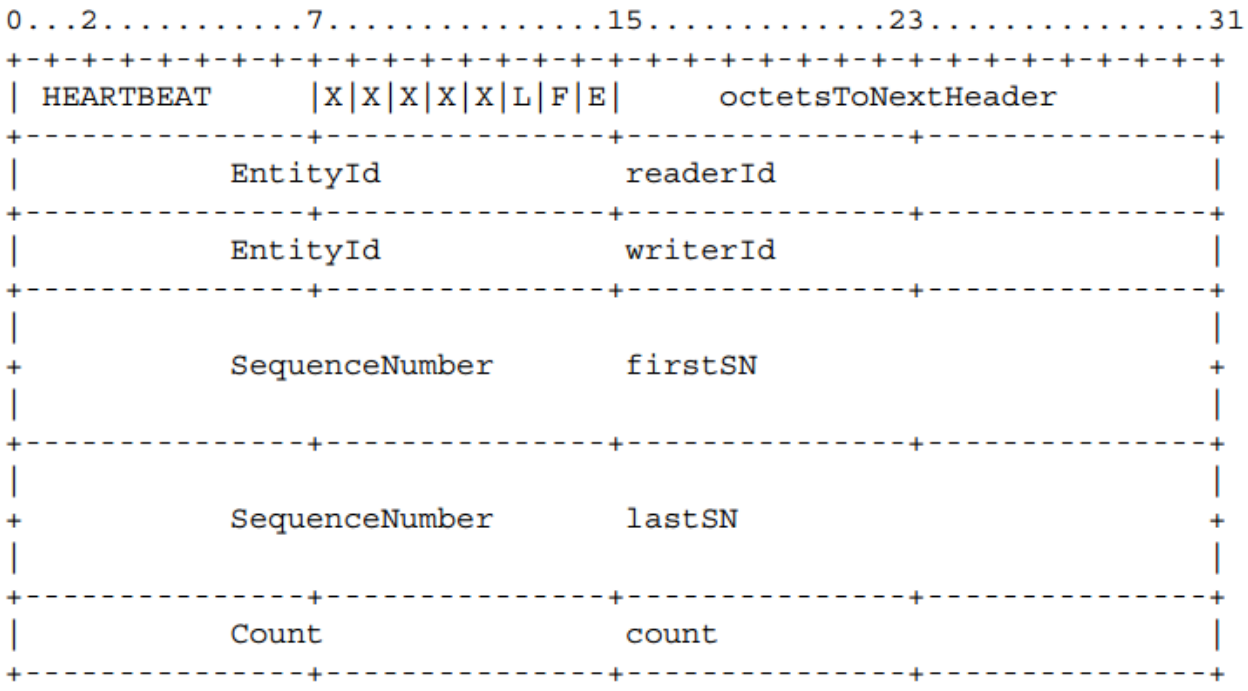


Figure 3.10. Heartbeat Submessage mapping [5, p. 172]

Este submensaje, al igual que el anterior que hemos descrito, es enviado de un RTPS Writer a un RTPS Reader.This message is sent from an RTPS Writer to an RTPS Reader to communicate the sequence numbers of changes that the Writer has available.

En este caso ofrece dos funciones: informar al Reader de los números de secuencia que están disponibles en el Writer para que el Reader realice una petición de los que tenga necesidad o bien realiza una petición de envío de confirmación de los cambios previos realizados para que el Writer conozca su estado.

En cuanto a su estructura sólo presenta tres campos que difieren del caso anterior:

- firstSN: Identifica el primer número de secuencia disponible en el Writer.
- lastSN: Identifica el último número de secuencia disponible en el Writer.
- count: Se trata de un contador que incrementa en uno su valor cada vez que genera y envía un nuevo mensaje de tipo Heartbeat. De esta manera el Reader puede detectar un mensaje duplicado e identificar un camino de comunicación redundante.4

El resto de campos poseen el mismo valor que en el caso del submensaje Data.

Respecto a los flags de la cabecera de un paquete de tipo Heartbeat tenemos:

- EndiannessFlag.
- FinalFlag: Indica si el Reader debe responder al mensaje Heartbeat con un ACKNACK o si sólo implica una finalidad meramente informativa con un bit a True o False respectivamente.
- LivelinessFlag: Indica que el DDS DataWriter asociado al RTPS Writer ha reafirmado su estado activo. Su valor estará fijado a 0.

AckNack Submessage

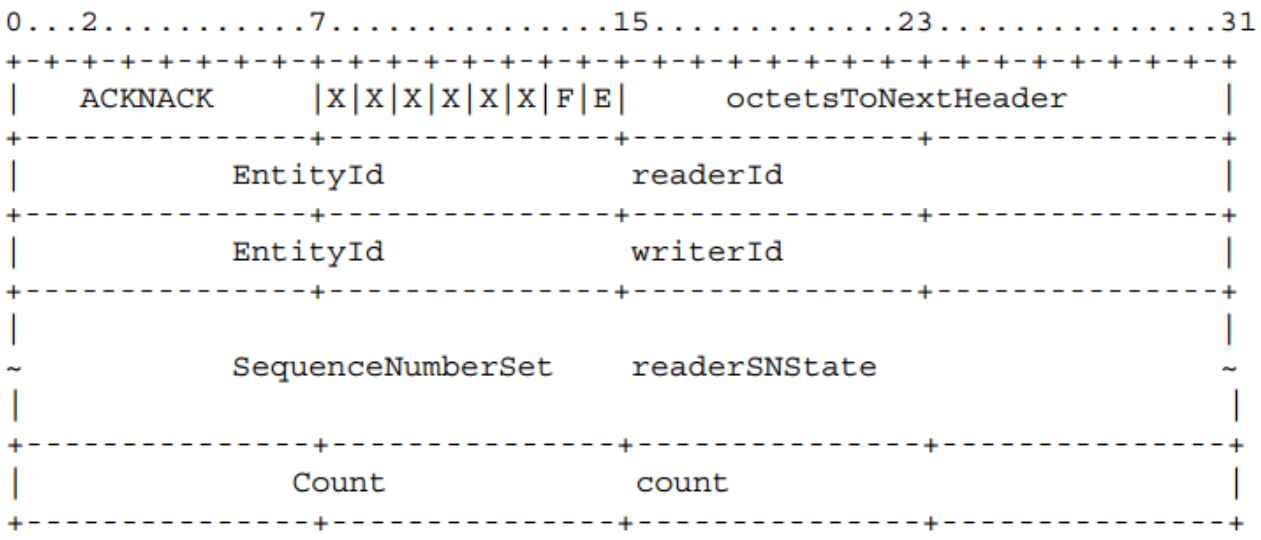


Figure 3.11. AckNack Submessage mapping [5, p. 168]

Por último este submensaje se utiliza para la comunicación en el sentido contrario, es el RTPS Reader quien envía el mensaje a un RTPS Writer.

Permite que el Reader informe al Writer de los números de secuencia que ha recibido y cuales necesita o se han extraviado.

En su estructura tenemos los mismos parámetros que en los submensajes anteriores, el único cambio nuevo que aparece es:

- readerSNState: Este campo comunica el estado del Reader al Writer. Todos los números de secuencia hasta el valor de este campo se confirman como recibidos por el Reader.

El valor de estos campos será dado a la hora de generar el objeto de la clase del tipo de submensaje correspondiente.

3.4.3 Generación dinámica

Estos mensajes serán generados de manera dinámica en función de lo que sea necesario enviar, y gran parte de los datos del mensaje serán tomados de la propia entidad que genere el mensaje.

La estrategia utilizada para la generación dinámica es mediante la programación orientada a objetos, la librería incluye clases para los diferentes submensajes, todas ellas heredadas de una clase raíz que incluye los parámetros que tienen en común todos estos submensajes, para ser más eficientes en la implementación.

De esta forma tenemos una clase RTPSPacket con la cabecera de mensaje y submensaje, y las clases heredadas RTPSDataPacket, RTPSAckNackPacket y RTPSHeartbeatPacket, cada una para definir los submensajes que usaremos.

La cabecera y subcabeceras serán parámetros comunes, así como la identificación de los readers y writers.

Por último, según el apartado 3.2.1.2, el envío de los datos vía WiFi Udp se realiza enviando un array de bytes, todos estos mensajes deben ser transformados en una cadena de bytes/hexadecimal para su posterior envío.

Para este cometido se ha implementado una función que vuelque los datos del objeto a un array, ya que no es posible usar clases contenedoras complejas de C++ que ya implementan esta función.

Esta función creada se favorece de la herencia de clases y cada submensaje solo debe convertir la parte que difiere de la base común.

```
unsigned char* RTPStoBuffer(unsigned char* RTPSbuffer,int bufferlen);
```

Todas las entidades de RTPS, las estructuras de datos necesarias para el protocolo, los diferentes submensajes y todas las funciones necesarias para la funcionalidad final han sido implementadas en una librería realizada en C++ que consta de 36 archivos.

3.5 RTPS Discovery Protocol

En este punto retomaremos los conceptos de Endpoint y Participant vistos en el apartado 3.3.1 y los relacionaremos con la explicación realizada del protocolo de descubrimiento en el apartado 2.4.4.

Por un lado, el Participant posee una serie de atributos, los cuales son utilizados para hacer posible el SPDP.

Este Participant a su vez posee un Writer preestablecido, encargado de mandar los datos del Participant a las direcciones IP que se encuentran en los campos “defaultUnicastLocatorList” y “defaulMulticastLocatorList” mediante un submensaje de tipo DATA. Estos valores, así como

los puertos a los que deben enviar la información están delimitados por el estándar en el apartado 9.6.2 Data representation for the built-in Endpoints. [5, p. 180]

A su vez otro Participant que reciba estos datos mediante un Reader es capaz de analizar los datos y proceder a realizare el SEDP de manera análoga con los datos incluidos en los Endpoints.

Retomando la información dada acerca del protocolo de descubrimiento en el Discovery Module, punto 2.4 más atrás, y teniendo en cuenta la limitación GRAVE de 90 bytes por paquete que comentamos en el apartado 2.5.1, hemos llegado a la conclusión de que el Participant prefijado con los datos necesarios para el descubrimiento enviaría lo mínimo imprescindible para su funcionamiento.

El metatráfico es enviado utilizando encapsulación de datos mediante parameterList [5, p. 182], los datos que tiene que llevar este paquete son los siguientes, acompañados de su tamaño real a la hora de enviarlo:

PROTOCOLVERSION: 8 bytes

GUID: 20 bytes

VENDORID: 8 bytes

DEFAULT MULTICAST METATRAFFIC: 28 bytes

DEFAULT MULTICAST LOCATOR: 28 bytes

Teniendo en cuenta que debe ir en el campo de serializedData de un paquete de tipo DATA y que ya de por si el resto del paquete sin este campo ocupa 48 bytes, nos quedamos con que necesitamos una capacidad mínima de paquete completo de 140 bytes sin contar el campo “Sentinel”, que indica el fin del Participant prefijado y ocupa 8 bytes más, por lo tanto **concluimos que la implementación del discovery protocol no es posible de realizar en el Arduino** mientras que no se aumente la limitación de 90 bytes mejorando las prestaciones de la WiFi Shield y la librería de WiFi Shield.

3.6 Entorno final de comunicación

Una vez entrado en los detalles de la implementación de cada uno de los elementos en los apartados anteriores y cómo funciona la comunicación con esta implementación del protocolo, más adelante veremos la implementación de un escenario final que se ofrece como ejemplo al cliente y servirá como validación de la implementación.

Haciendo uso de la librería RTPS, tenemos la capacidad de crear las entidades que participan en la comunicación, es decir, para la publicación de datos disponemos de un RTPS StatelessWriter, veamos cómo es la publicación de datos.

Para la publicación de datos necesitamos crear un nuevo objeto de tipo RTPS StatelessWriter,

quien se encargará de enviar mensajes. Esta entidad se generará con los valores estándar a priori, y mediante una función se le asignarán los valores deseados.

```
void newRTPSWriter(GUID_t wguid, Locator_t unicastLocator, ...);
```

Esta función inicializa todos los datos tal como deseamos, y como lo hace el estándar.

```
this->setGUID(wguid);
this->addUnicastLocator(unicastLocator);
this->addMulticastLocator(multicastLocator);
this->reliabilityLevel=BEST_EFFORT;
this->topicKind=NO_KEY;
this->pushMode=true;
this->setHeartbeatPeriod(hbmillis);
this->setNackResponseDelay(NACKDEFAULTRESPONSEDELAY);
this->setNackSupressionDuration(SUPPRESSIONDEFAULTDURATION);
this->setLastChangeSequenceNumber();
RTPS_HistoryCache* wCache= new RTPS_HistoryCache();
writer_cache=*wCache;
```

En caso de enviar datos en un submensaje de tipo Data, se realizará enviando los datos pendientes en su HistoryCache.

Por lo tanto, se generarán nuevos cambios en la caché con los datos deseados, por ejemplo, para el envío de datos de temperatura, usaremos:

```
RTPS_CacheChange new_Change(float tempData);
```

Una vez que el Writer tiene cambios en su caché, para enviar todos los cambios que tenga, hacemos uso de la función:

```
void sendDataPacket(int port, unsigned char* identif);
```

Que enviará los datos al puerto deseado y con el formato de CDR indicado en el segundo campo.

Para el envío de otro tipo de datos se debe realizar la sobrecarga de la función new_Change con los tipos de datos deseados. Se realizará en futuras implementaciones.

También se incluye la implementación para el envío de otro tipo de mensajes que no sean de datos, para ello se hará uso de las funciones del Writer para dicho fin.

```
void sendHeartbeatPacket(int port);
void sendAckNackPacket(int port);
```

Con esto finalizamos el tercer objetivo del proyecto, expuesto en el apartado 1.2.

4 Validación

La validación y comprobación del funcionamiento correcto ha sido respaldado por el uso de un potente analizador de tráfico, el muy conocido WireShark. Gracias a un proyecto llevado a cabo hace unos años se añadió a WireShark la funcionalidad de captura y análisis de paquetes RTPS. Los detalles acerca de cómo es la captura y qué funciones del protocolo RTPS incluye se pueden consultar en la propia documentación de la página web. [20] [21]

El objetivo de la validación es la comprobación de que los datos que queremos enviar son reconocidos por WireShark como un paquete RTPS con los valores que se deseaba enviar. De esta forma se verá cumplido el último objetivo de los planteados en el apartado 1.2.

4.1 Esquema de validación

Para realizar la validación hemos generado un entorno real de sensorización de datos de temperatura que serán enviados al endpoint remoto.

Como no es posible la implementación y uso del protocolo de descubrimiento según vimos en el apartado 3.5, se considerará preestablecido el descubrimiento, en donde todos los elementos de la red tienen constancia de los demás.

Aunque el protocolo esté adaptado para distintos tipos de datos y para un gran número de situaciones, realizaremos el diseño de un entorno en el que la placa Arduino tome una medida que creemos interesante, la temperatura del ambiente, y publique este dato. Es realmente importante el hecho de que Arduino pueda adaptarse a distintos sensores para poder ofrecer funcionalidades de monitorización muy útiles para la industria o para tareas cotidianas.

En primer lugar realizaremos la conexión física de la placa Arduino con el sensor utilizado, el LM35, un sensor de temperatura con una precisión calibrada de 1ºC con el que podremos medir en un rango desde -55°C hasta 150°C. Su utilización es realmente sencilla siguiendo las instrucciones de conexión del Datasheet [22]. Anexo 3.

La conexión física final es la siguiente:

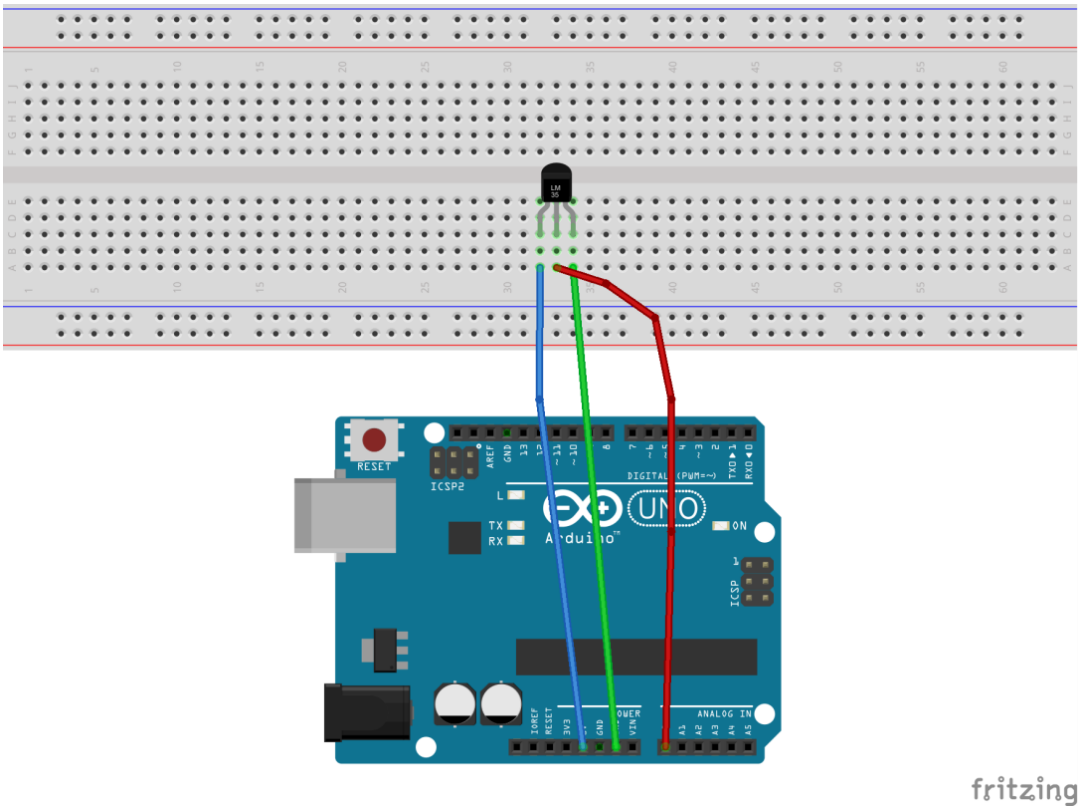


Figure 4.1a LM35 Connection Scheme

Una vez tenemos nuestra conexión física realizada, haremos uso de la librería RTPS implementada y del código incluido en Arduino, podemos verlo completo en el Anexo 2.
Realizaremos la configuración inicial de los parámetros para la conexión. En primer lugar la inclusión de la librería:

```
#include <SPI.h>
#include <WiFi.h>
#include <WiFiUdp.h>
#include <RTPSPacket.h>
#include <RTPSDataPacket.h>
#include <RTPSHeartbeatPacket.h>
#include <RTPS_Writer.h>
```

Ahora es posible el uso de la librería, generaremos el RTPS Writer que enviará los datos:

```
RTPS Writer Writer = RTPS Writer();
```

Los constructores en Arduino funcionan de manera diferente, el valor que se le asigna es por defecto, y no es posible realizar constructores con valores, por eso, después de generar un nuevo RTPS Writer, llamamos a la función que da valor a sus parámetros:

```
Writer.newRTPSWriter(myGuid,pcIP,bcIP,4000);
```

Donde myGuid es el GUID que desee dar el valor el cliente y los tres parámetros restantes son las direcciones IP y puertos para establecer la conexión y envío de mensajes, en este caso se ha utilizado la dirección de broadcast.

Tras tener la conexión iniciada, habiendo previamente realizado la conexión a la red WiFi con SSID y password, se procede a la toma de datos mediante el sensor y se añade el nuevo valor en un nuevo cambio en la caché, que se incluye en el historial del RTPS Writer, que al tener algún valor lo enviará a su dirección de destino, el endpoint remoto. Todas estas acciones se realizan dentro del bucle infinito, la toma de datos se realiza como sigue:

```
void loop() {
  temperatura = analogRead(analog_pin);
  temperatura = 5.0*temperatura*100.0/1024.0;
  Serial.print(temperatura);
  Serial.println(" oC");
```

Como vemos simplemente tomamos el valor análogico y realizamos la transformación a temperatura. Esta fórmula se encuentra en el propio Datasheet [22]. Como añadido mostraremos el valor de temperatura por el puerto serie, para comparar el valor real con el valor que es enviado en el paquete RTPS por el RTPS Writer al realizar la comprobación en Wireshark.

La otra parte del código, el tratamiento de los datos de temperatura para su envío se realiza como sigue:

```
Writer.new_Change(temperatura);
Writer.sendDataPacket(6000,CDR_LE);
delay(2000);
```

Donde hemos indicado el envío con formato Little Endian, al estar trabajando con Arduino, indicando el formato en el que está serializados los datos, y el puerto al que se enviará el mensaje.

El envío es periódico, hemos incluido un delay de 2 segundos, por lo tanto tomaremos datos de temperatura cada 2 segundos y los enviaremos mediante un mensaje RTPS.

4.2 Pruebas

El envío de datos fue realizado por un RTPS StatelessWriter, con un tiempo entre mensajes de dos segundos, enviando el valor de las mediciones del sensor.

Comprobaremos que el valor enviado en el paquete tipo Data es coherente con la temperatura que debe ser medida, comparándolo con el valor mostrado mediante el puerto Serie y además viendo si es coherente con la temperatura ambiente.

Podemos observar la memoria del dispositivo en uso mediante este programa al compilar, tanto de las variables globales como del propio código:

Sketch uses 15.472 bytes (47%) of program storage space. Maximum is 32.256 bytes.

Global variables use 1.007 bytes (49%) of dynamic memory, leaving 1.041 bytes for local variables. Maximum is 2.048 bytes.

Una vez compilado el código, procedemos a cargar el programa en arduino para que empiece a realizar la función que deseamos, obteniendo lo siguiente:

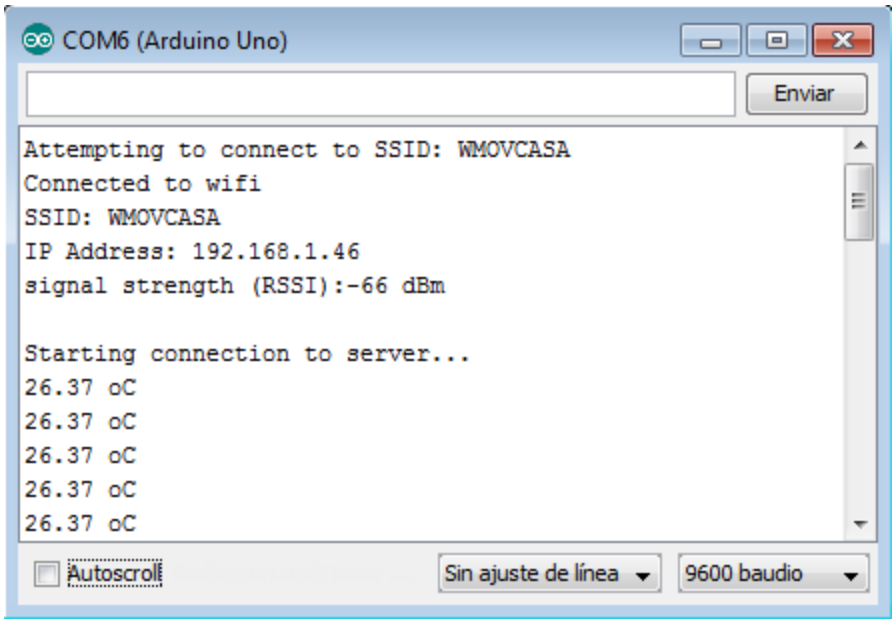


Figure 4.2. Serial port data

Obtenemos por pantalla mediante puerto serie la confirmación de conexión a la red WiFi y a continuación se muestran los datos de temperatura actual de manera periódica.

Analicemos con detalle uno de los mensajes RTPS capturados.

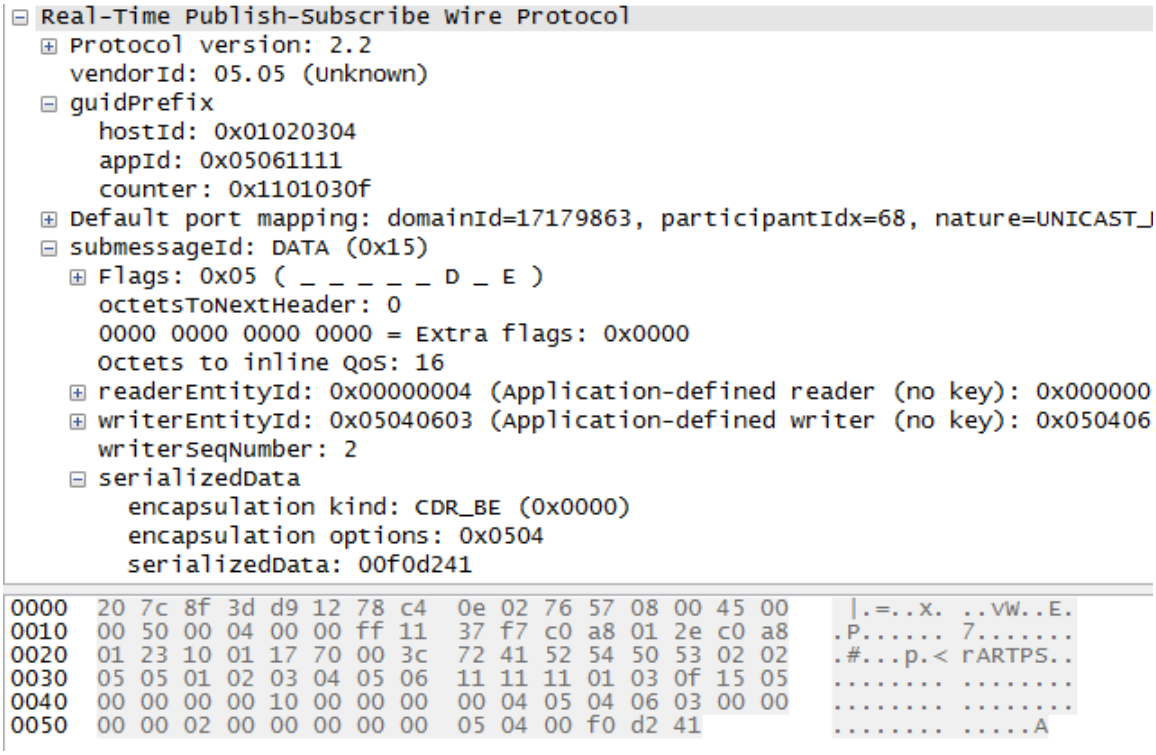


Figure 4.3. RTPS Data temperature submessage

Veamos la parte que nos interesa realmente:

```
serializedData
    encapsulation kind: CDR_BE (0x0000)
    encapsulation options: 0x0504
    serializedData: 00f0d241
```

4.3 Resultados

Una vez realizadas las pruebas a la validación, hemos podido comprobar que el valor del campo SerializedPayload del mensaje enviado. Como vemos estamos en un formato little endian, el valor 0x00f0d241 corresponde a 26.367188. En caso de tratarse de un sistema big endian estaría invertido.

Por otro lado, analizando los datos del puerto serie, podemos comprobar que coincide con el valor enviado por puerto serie de manera bastante precisa, 26.37 grados, que además es la temperatura del ambiente en el que ha sido realizada la comunicación.

Con esto hemos podido demostrar que:

- El sensor LM35 realiza correctamente la medida de temperatura.
- El mensaje RTPS enviado cumple las expectativas de diseño al generarse de manera dinámica con todos los datos deseados.

Ahora sí, con estos resultados podemos concluir que se ha completado el último objetivo de los definidos en el apartado 1.2.

5 Conclusions

After the implementation and the tests realized, we are able to discern what can be carried out, what cannot be carried out and what could be implemented even though we have not tried to implement it because it was out of the scope of the project.

Let us remember that the interconnection with a real DDS system has not been executed, but it has been proven to which extent the implementation of the RTPS was possible, and the most important thing, proving that messages can be sent correctly.

Firstly, we have followed the action scheme of section 1.1, accomplishing the objectives planned, choosing Arduino as platform for IoT and building a library for giving the possible functionalities of RTPS.

Within the more concrete frame of objectives that we plan in section 3.2 related to the library, we have been able to accomplish the following:

- The sending of the three simplest messages for communication through RTPS, Heartbeat, AckNack and Data. Let us remember that each RTPS message shall only include one submessage due to the board limitations.
- With regard to the previous point, having shown that the sending of these three sub-message types is possible, the inclusion in the library of the remaining RTPS messages shall also be possible as long as the limit frame of the board is accomplished.
- The sending of simple RTPS messages has been possible. It is the library's clients responsibility to adapt the new types of messages to the library, dealing with its serialization.
- The communication via RTPS messages is bidirectional, the Arduino board has received and sent RTPS packets. Sending messages to Arduino has been possible by the application Processing from a computer.
- The implementation of all the RTPS entities was carried out from the basic one, RTPS Entity, and with the inheritance structure marked by RTPS.
- Specifying the previous point, it has been able to generate a RTPS Writer which is able to send RTPS messages to other IP directions in the net.

On the other hand, we have proved whatever could not be achieved due the limitations of the board:

- The implementation of the discovery protocol has not been possible due to the limitations of the Arduino.
- The implementation of a Writer with the capacity of sending different types of data has not been possible because Arduino does not include templates.
- The interconnection of the library with a real DDS system has not been possible but it was not one of the objectives and its accomplishment was not planned.
- Even though both endpoints have been implemented, Writer and Reader, it has not been possible to accomplish a stable functionality, so its implementation has not been validated.

5.1 Future work

Having worked on the library for some time and having carried out several analysis of what can be achieved, the future work lines are aimed at improving whatever could not be done on this platform and to work in another framework.

Implementing the library under Arduino, it has been possible to carry out whatever was possible for its better completeness. However, there are a few points which could be improved or which could add more value to the library, in accordance with the points mentioned above:

- The three most important types of sub-messages have been included, but RTPS includes a big variety of them. It is possible to assume that having implemented the three most important ones, the other can also be included in the library.
- By implementing a library for giving templates support, one of the characteristics that give sense to C++ language, we could send different types of data more easily, so it would be interesting to either generate a new library or adapt an existing one for Arduino.
- As previously mentioned, the operation of the RTPS Reader in a stable way has not been achieved due to different problems, such as memory problems. However, the basis for a particular operation are implemented as well as the extraction of date from a Data kind package that reaches RTPS Reader and the inclusion of those data in its HistoryCache. It would be interesting to operate it in a stable way.

However, there is a deeper problem which is in fact a base of the design and implementation.

As we have mentioned before, in our work we had to adapt the design due to some limitations which make extremely difficult the implementation.

Many functionalities that we have to implement require a more powerful communication mechanism. Hence, for further work lines on this library, it would be more convenient to think in deeper changes:

- Use of another library for WiFi communications: the biggest limitation is the maximum 90 bytes of data to transfer and the absence of multicast capacity. Using other libraries or improving the one we have been using, could let us implement several more functionalities, and more important ones.
- Use of other Shield attached to Arduino for IoT: even though we are using the WiFi Shield board, it is also possible to use other boards such as Ethernet Shield, which includes another library of Arduino that does not present this size limits in the sending of packages and presents similar characteristics, capacity over UDP, and is more trustful. In addition, the adaptation of the library to this other platform would be immediate.

In any case, it has been demonstrated that it is possible to employ low performance IoT platforms, such as the ones described along this degree thesis, to design and implement functionalities of a strong industrial protocol, such as RTPS

6 Planificación

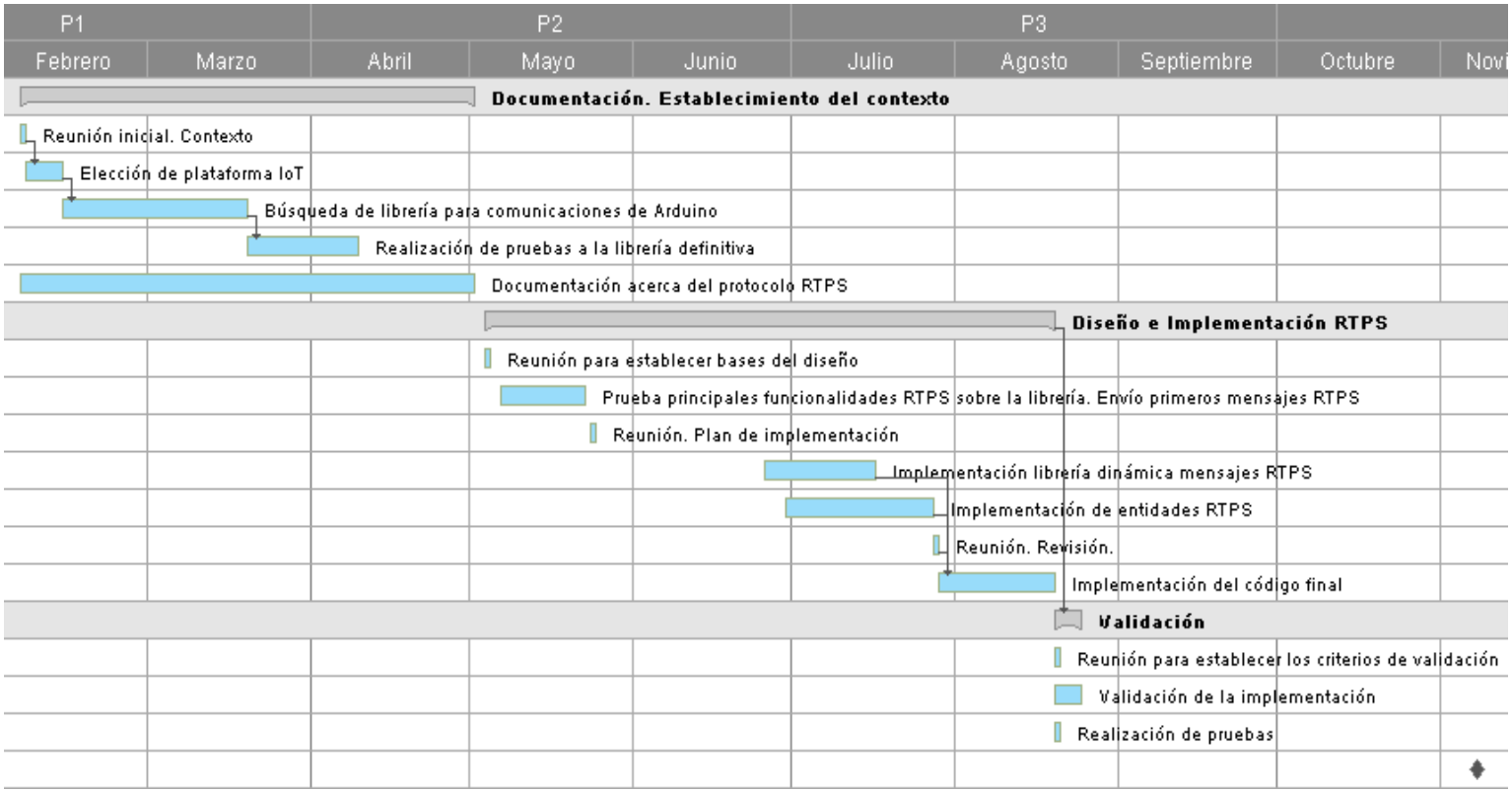


Figure 6. Gantt Chart

7 Bibliografía

[1] G. Tarek y R. Rojdi, *Design and Performance of DDS-based Middleware for Real- Time Control Systems,* CiteSeerX, 2007.

[2] K. Ashton, «That 'Internet of Things' Thing,» *RFID Journal,* p. 1, 2009.

[3] C. Witchalls y J. Chambers, «The Internet of Things business index: A quiet revolution gathers,» Economist Intelligence Unit, USA, 2013.

[4] Object Management Group, «OMG And The IIoT,». Available: http://www.omg.org/hot-topics/iot-standards.htm.

[5] Object Management Group (OMG) ; Real-time Innovations, Inc., *The Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification Version 2.2,* 2014.

[6] E. SatterWhite, «Summer of Sockets part 3: Pub Sub With ZeroMQ,» Codedependant, 13 September 2014. Available: http://www.codedependant.net/2014/09/13/pub-sub-with-zeromq/.

[7] A. Corsaro, «The Data Distribution Service for Real-Time Systems,» *Dr. Dobbs,* vol. I, pp. 1-2, 2010.

[8] Raspberry Pi Foundation, «Raspberry Pi,» University of Cambridge, 2008. Available: https://www.raspberrypi.org/.

[9] The BeagleBoard.org Foundation, «BeagleBoard.org,». Available: http://beagleboard.org/.

[10] http://www.udoo.org/udoo-neo/, «UDOO,» SECO USA Inc. ; Aidilab, 2015. Available: http://www.udoo.org/udoo-neo/.

[11] Arduino , «Arduino,». Available: https://www.arduino.cc/.

[12] Netduino, «Netduino,». Available: http://www.netduino.com/hardware/.

[13] A. Alasdair, «Arduino Uno vs BeagleBone vs Raspberry Pi,» *Make:,* 2015.

[14] T. K., «Toby K build log,» 29 April 2014. Available: http://buildlog.tobyk.com.au/reliability-test-uno-r3-wifi-shield-udp/.

[15] T. Sumant, «A C++ Template Library for Data-Centric Type Modeling for DDS-XTypes,» Real-Time Innovations, 2014.

[16] Arduino, «Writing a Library for Arduino,» November 2007. Available: https://www.arduino.cc/en/Hacking/LibraryTutorial.

[17] The Apache Software Foundation, «Apache Mina,». Available: https://mina.apache.org/.

[18] University of California at Berkeley., «Center for New Music & Audio Technologies,». Available: http://cnmat.berkeley.edu/oscuino.

[19] Souliss Team, «Soulis,». Available: http://www.souliss.net/p/about.html#getit.

[20] Wireshark Wiki, «Wireshark Wiki,» 18 June 2013. Available: https://wiki.wireshark.org/FrontPage.

[21] Wireshark Foundation, «Wireshark RTPS docs,». Available: https://www.wireshark.org/docs/dfref/r/rtps.html.

[22] Texas Instruments, «LM35 Precision Centigrade Temperature Sensors,» 1999, Revised January 2015. Available: http://www.ti.com/lit/ds/symlink/lm35.pdf.

8 Anexo 1. Estructura de datos

Type	PSM UDP-IP Mapping

Time_t	**Mapping of the type**
	struct Time_t {
	long seconds; // time in seconds
	unsigned long fraction; // time in sec/2^32
	};
	
	Mapping of the reserved values:
	#define TIME_ZERO {0, 0}
	#define TIME_INVALID {-1, 0xffffffff}
	#define TIME_INFINITE {0x7fffffff, 0xffffffff}

VendorId_t	**Mapping of the type**
	typedef octet OctetArray2[2];
	struct VendorId_t {
	OctetArray2 vendorId;
	};
	
	Mapping of the reserved values:
	#define VENDORID_UNKNOWN {0,0}

SequenceNumber_t	**Mapping of the type**
	struct SequenceNumber_t {
	long high;
	unsigned long low;
	};
	
	Mapping of the reserved values:
	#define SEQUENCENUMBER_UNKNOWN {-1,0}

TopicKind_t	**Mapping of the type**
	struct TopicKind_t {
	long value;
	};
	
	Mapping of the reserved values:
	#define NO_KEY 1
	#define WITH_KEY 2

Type	PSM UDP-IP Mapping

Locator_t	**Mapping of the type**
	typedef octet OctetArray16[16];
	struct Locator_t {
	long kind;
	unsigned long port;
	OctetArray16 address;
	};
	
	Mapping of the reserved values:
	#define LOCATOR_INVALID \
	{LOCATOR_KIND_INVALID, LOCATOR_PORT_INVALID, LOCATOR_ADDRESS_INVALID}
	#define LOCATOR_KIND_INVALID -1
	#define LOCATOR_ADDRESS_INVALID {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
	#define LOCATOR_PORT_INVALID 0
	#define LOCATOR_KIND_RESERVED 0
	#define LOCATOR_KIND_UDPv4 1
	#define LOCATOR_KIND_UDPv6 2

FragmentNumber_t	**Mapping of the type**
	struct FragmentNumber_t {
	unsigned long value;
	};

ReliabilityKind_t	**Mapping of the type**
	struct ReliabilityKind_t {
	long value;
	};
	
	Mapping of the reserved values:
	#define BEST_EFFORT 1
	#define RELIABLE 3

Count_t	**Mapping of the type**
	struct Count_t {
	long value;
	};

Type	PSM UDP-IP Mapping

ParameterId_t	**Mapping of the type**
	struct ParameterId_t {
	short value;
	};

EntityName_t	**Mapping of the type**
	struct EntityName_t {
	string name;
	};

ProtocolVersion_t	**Mapping of the type**
	struct ProtocolVersion_t {
	octet major;
	octet minor;
	};
	
	Mapping of the reserved values:
	#define PROTOCOLVERSION_1_0 {1,0}
	#define PROTOCOLVERSION_1_1 {1,1}
	#define PROTOCOLVERSION_2_0 {2,0}
	#define PROTOCOLVERSION_2_1 {2,1}
	#define PROTOCOLVERSION_2_2 {2,2}
	#define PROTOCOLVERSION PROTOCOLVERSION_2_2
	
	The Implementations following this version of the document implement protocol version 2.2 (major = 2, minor = 2).

GUID_t	**Mapping of the type** struct GUID_t { GuidPrefix_t guidPrefix; EntityId_t entityId; } ; **Mapping of the reserved values:** #define GUID_UNKNOWN{ GUIDPREFIX_UNKNOWN, ENTITYID_UNKNOWN }

Table 1. PSM UDP/IP Structure Mapping

9 Anexo 2. Código fuente

Arduino main code. This is only the sketch that is being loaded into the Arduino board, the RTPS library is a 36 files library which is not attached.

```
/*  WiFi UDP Send RTPS Messages This sketch takes a temperature value and
sends it as an RTPS message to a remote endpoint via WiFi Shield UDP.
Circuit: * WiFi shield and LM35 sensor attached
Created 30 August 2015 by Norbert Ludant   */

#include <SPI.h>
#include <WiFi.h>
#include <WiFiUdp.h>
#include <RTPSPacket.h>
#include <RTPSDataPacket.h>
#include <RTPSHeartbeatPacket.h>
#include <RTPS_Writer.h>
#include <string.h>


int status = WL_IDLE_STATUS;
char ssid[] = "NetworkSSIDname"; //  your network SSID (name)
char pass[] = "Networkpassword"; // your network password (use for WPA, or
use as key for WEP)
int keyIndex = 0;            // your network key Index number (needed only
for WEP)

GUID_t myGuid;
IPAddress pcIP = IPAddress(192,168,1,35);
IPAddress bcIP = IPAddress(192,168,1,255);
unsigned char MyGuidPrefix[] =
{0x01,0x02,0x03,0x04,0x05,0x06,0x11,0x11,0x11,0x01,0x03,0x0f};
unsigned char myId[] = {0x05,0x04,0x06};
unsigned char noId[] = {0x00,0x00,0x01};
unsigned char CDR_LE[]={0x00,0x01};
unsigned char CDR_BE[]={0x00,0x00};
unsigned char IP[]={192,168,1,35};
int port = 6000;
int analog_pin = 0;
float temperatura;

RTPS_Writer Writer = RTPS_Writer();

unsigned char RTPSbuffer[60];


void setup() {
  //Initialize serial and wait for port to open:
  Serial.begin(9600);
//  while (!Serial) {
//    ; // wait for serial port to connect. Needed for Leonardo only
//  }

  // check for the presence of the shield:
```

```
  if (WiFi.status() == WL_NO_SHIELD) {
    Serial.println("WiFi shield not present");
    // don't continue:
    while (true);
  }
 String fv = WiFi.firmwareVersion();
  if ( fv != "1.1.0" )
    Serial.println("Please upgrade the firmware");

  // attempt to connect to Wifi network:
  while ( status != WL_CONNECTED) {
    Serial.print("Attempting to connect to SSID: ");
    Serial.println(ssid);
    // Connect to WPA/WPA2 network.
    status = WiFi.begin(ssid,pass);

    // wait 10 seconds for connection:
    delay(10000);
  }
  Serial.println("Connected to wifi");
  printWifiStatus();

  Serial.println("\nStarting connection to server...");
 memcpy(myGuid.guidPrefix, MyGuidPrefix, sizeof(MyGuidPrefix));
  memcpy(myGuid.entityId.entityKey, myId, sizeof(myId));

  myGuid.entityId.entityKind=0x03;
  Writer.newRTPSWriter(myGuid,pcIP,bcIP,4000);
}


void loop() {

  //Medicion de la temperatura
  temperatura = analogRead(analog_pin);
  temperatura = 5.0*temperatura*100.0/1024.0;
  Serial.print(temperatura);
  Serial.println(" oC");

  //A CacheChange is added to RTPS Writer's HistoryCache, then this change is
sent as an RTPS message to the remote endpoint
  Writer.new_Change(temperatura);
  Writer.sendDataPacket(6000,CDR_BE);
  delay(2000);

}
void printWifiStatus() {
  // print the SSID of the network you're attached to:
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());

  // print your WiFi shield's IP address:
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);
```

```
  // print the received signal strength:
  long rssi = WiFi.RSSI();
  Serial.print("signal strength (RSSI):");
  Serial.print(rssi);
  Serial.println(" dBm");
}
```

10 Anexo 3. Esquema de conexión

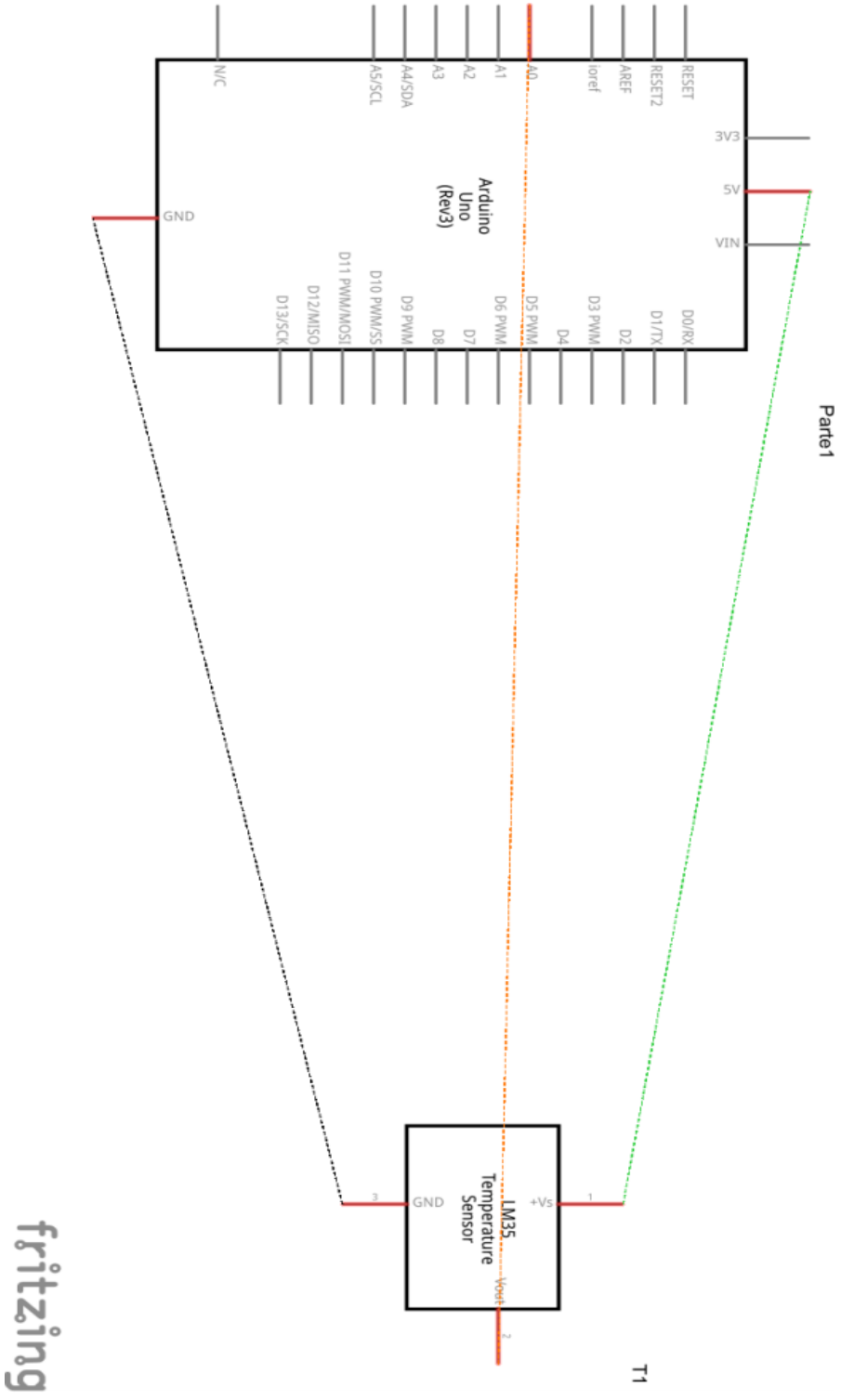


Figure 10.1.b LM35 Connection Scheme

11 Anexo 4. Summary

This annex provides a brief summary of the degree thesis, from contextualization to conclusions, including implementation and validation, and without delving into specific factors, which may be contrasted with the full documentation.

Context

With increasing interconnection of machines and people and the huge exchange of information it involves, we are experiencing many revolutionary changes. This concept, known as Internet of Things (IoT), brings amazing benefits in a wide variety of aspects, from profound changes in the daily lives of citizens, offering "smart" functionalities to everyday objects, to industrial areas, where this concept makes the real difference, providing distinct advantages, such as increase in productivity, cost reduction, and other intelligent processes.

But there is a lack of support for this new technology, many techniques still in use today are now out of place. This is the case of data distribution patterns, IoT's data flow needs concrete specifications of communication, that the typical server-client schemes cannot provide.

In this context emerges Publish-Subscribe, a data distribution pattern that tries to give a solution to that problem, it is based on a highly decoupled system of communication. In this message pattern there are two main entities, senders of information, called Publishers, that are not programed to send messages directly to other endpoints, but label published messages with a topic. At the other extreme, Subscribers express interest in those topics, and receive information as soon as it is available.

This situation enables a new industrial environment, where new middlewares based on the described pattern emerge, aimed to deploy new features in the industry and to solve the difficulty of extracting data and interoperating various devices.

We will focus on DDS, one of the main middlewares for data-centric systems, it presents numerous advantages, such as an efficient distribution system, scalability, ease of use and provides Quality of Service. It has been deployed over multitude of critical distribution systems, so it is a field proven technology.

On the other hand, one of its disadvantages or problems is the lack of interoperability between different vendors implementations. In order to tackle these problems, OMG (Object Management Group) has developed a wire protocol standard that supports DDS, RTPS.

RTPS (Real Time Publish Subscribe Protocol) is a protocol for best effort and reliable pub-sub communications over unreliable transports such as UDP in both unicast and multicast.

Besides the RTPS implementations embedded in the different DDS implementations, there are standalone lightweight RTPS implementations.

Both RTPS and DDS have had great success in the industry, being deployed over many critical real-time systems, specially in the aerospace and defense sectors. However, all these implementations have been run over powerful devices, not over a low-cost low-power development board, so there is an opportunity to implement RTPS on a small IoT platform, such as Arduino.

Let us take a look at the main features of RTPS, to see how they fit development board performance:

- Capability of exploiting multicast.
- Plug-and-play connectivity so that new applications and services are automatically discovered and applications can join and leave the network at any time without the need for reconfiguration.
- Extensibility to allow the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- Modularity to allow simple devices to implement a subset of the protocol and still participate in the network.
- Scalability to enable systems to potentially scale to very large networks. [5]

As we can see, the requirements and specifications of both RTPS protocol and common development boards match really well. The last step was choosing the right IoT-oriented development board, after a deep research, Arduino was chosen because of its strengths.

RTPS Architecture

The RTPS protocol is based on four different modules, controlling all parameters of the communication and exchange of information.

- The Structure module: defines the RTPS inner structure and its relation with the DDS equivalents.

- The Message module: defines the structure of the RTPS messages. The three most important messages are Heartbeat, that notifies new changes on the publisher, Data, that sends information to its matched endpoints, and acknack, whose function is to acknowledge a message.

- The Behavior module: defines the possible interactions between endpoints. This module describes something very important, the Stateless and Stateful implementation, two different implementations based on the knowledge about the state of its remote endpoints.

- The discovery module defines a set of built-in endpoints that can be used to allow the automatic discovery, this module cannot be implemented because of the Arduino memory limitations.

Design and implementation

As we have seen, there is a lot of interest in deploying RTPS on a low power microcontroller, the main purpose is to design and implement those RTPS features and parameters that is possible to include in a low-performance device, such as Arduino, and generating a final data sensing environment where communication takes place via the RTPS protocol, for efficient data distribution. All functions and entities of the protocol have been implemented strictly according to the OMG specifications.

As mentioned previously, RTPS is oriented to best effort and reliable pub-sub communications over unreliable transports, so our implementation will be based over UDP IP in a WiFi network. For this purpose, an Arduino library have had to be found in order to enable WiFi communications. After testing several libraries and encountering problems to make RTPS message exchange compatible, finally it was decided to use the official WiFi Arduino library, which has a lot of limitations, such as:

- There is a 90 byte transfer limit.
- It is not very reliable sending packets for long periods of time or without enough time between them.
- Software limitation, such as poor templates support.

We will have to work in this particular circumstances, therefore, we have to adapt the desired functionalities to this limitations.

The final implementation has the following features:

- Implementation of all the RTPS Entities, following the inheritance scheme. Only the Stateless implementation has been included due to Arduino limitations. All the attributes have been implemented following the Common Data Representation (CDR) specifications. Finally, the Entities related to the discovery protocol are not included neither.

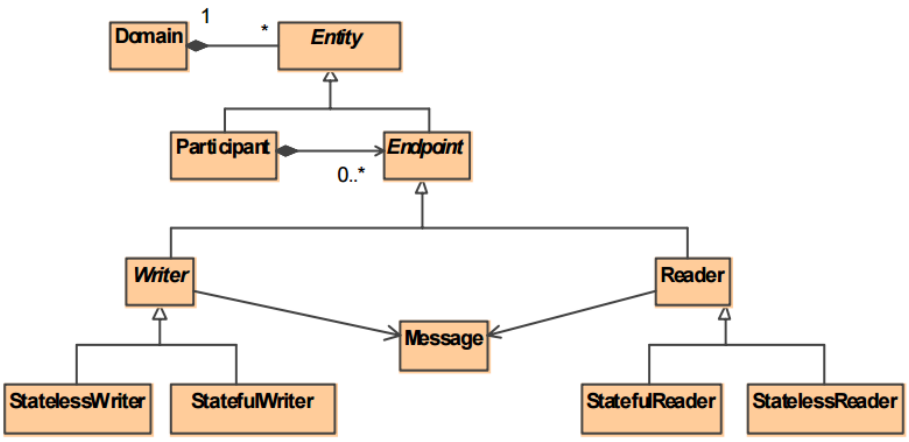


Figure 1. RTPS Entities Scheme [5]

- Dynamic generation of the three most important messages, Heartbeat, AckNack and Data messages.
- Implementation of the RTPS StatelessWriter functionalities, generating new StatelessWriter, adding CacheChanges to his HistoryCache, removing CacheChanges from his HistoryCache and sending the three most important messages of RTPS to his matched endpoints.
- Implementation of the RTPS StatelessReader functionalities, however, a stable operation has not been achieved, so the validation has been discarded.

It is important to note that sending Data submessages is in line with the protocol specification procedure; a CacheChange is added to his HistoryCache and the RTPS StatelessWriter sends all the changes from his HistoryCache to the matched endpoints that have previously subscribed to this type of data.

Arduino operation is similar to any other development board, a sketch is loaded and runs continuously in the same loop. Obviously, we will not make such a large deployment of code in the Arduino sketch itself, we will just write a new whole library for more robustness and simplicity.
This library will be written in C++ language, but with the limitations of Arduino programming, consequently, typical C++ libraries cannot be used, such as stdio.h, we are limited to using only the ones programmed specifically for Arduino, for this purpose we are going to use a cross-platform IDE, CodeBlocks.

Furthermore, all RTPS entities have been implemented in different classes, respecting the inheritance principles imposed by RTPS, and the dynamic generation of RTPS messages is also performed by object-oriented programming.

It is important to note that RTPS includes its own data types, following a fixed structure to make sure that the message is generated correctly and it is a real RTPS message. Both messages and entities parameters make use of this structures, and the default values have been assigned to many parameters. In addition, a configuration file has been included to modify communication conditions and fixed values.

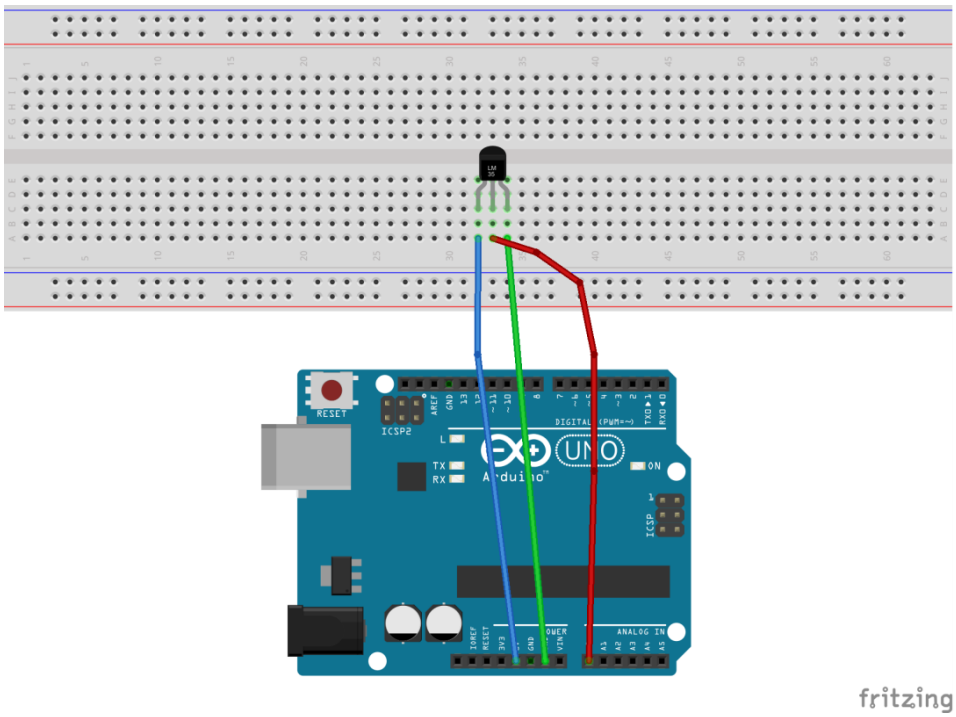
Validation. Final environment

The last step is the validation of the implementation, testing the whole map of functionalities we deployed. For this purpose we use a traffic analyzer, Wireshark, which includes RTPS capture and analysis addons.

The main goal of this validation is testing if the RTPS Stateless Writer functionalities are working as it should and if the dynamic generated messages have all the fields correctly generated.

This can be done generating a final communication scheme and checking through Wireshark display if the message we are trying to send is really reaching its destination. If so, we proceed to analyze the whole RTPS message.

The final communication scheme was just built attaching a temperature sensor, LM35, to our Arduino UNO + Arduino WiFi Shield boards.



After displaying on the serial port the sensor value, and comparing it with the room temperature and with the value included on the SerializedData field on the RTPS message captured in Wireshark, we can conclude, checking previously if all RTPS fields are correctly generated too, that the RTPS message was generated correctly based on what we wanted to send.

Conclusions

As we could see, some of the main RTPS functionalities implemented on a low cost IoT-oriented development board have been validated, specifically the correct sending of the different RTPS messages and the mapping of the RTPS entities. This opens up new perspectives to focus in working on small footprint implementations of emerging distribution protocols, trying to give a solution to IoT problems.

Future work includes new functionalities, such as the ones related to the RTPS StatelessReader, or including all type of RTPS messages and new types of data for data messages. Of course, migrating the implementation to another development board with less limitations is also contemplated.

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor: Norbert Ludant

2.- Departamento: Telemática

3.- Descripción del Proyecto:

- Titulo: **Diseño e implementación de RTPS en Arduino**
- Duración (meses): **4**
- Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

20.051,00 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar - solo a titulo informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
García Reinoso, Jaime Jose		Ingeniero Senior	1	4.289,54	4.289,54	Jaime
Santos Méndez, Agustín		Ingeniero Senior	1	4.289,54	4.289,54	Agustín
Norbert Ludant		Ingeniero	3	2.694,39	8.083,17	Norbert
					0,00	
					0,00	
		Hombres mes 5		**Total**	16.662,25	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Ordenador portátil	600,00	100	4	60	40,00
Arduino UNO	22,00	100	3	60	1,10
Arduino WiFi Shield	69,90	100	3	60	3,50
Router WiFi	51,40	100	3	60	2,57
		100		60	0,00
					0,00
				Total	47,17

[d] Fórmula de cálculo de la Amortización:

$$\frac{A}{B} \text{xCxD}$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
	Total	0,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	16.662
Amortización	47
Subcontratación de tareas	0
Costes de funcionamiento	0
Costes Indirectos	3.342
Total	20.051