ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

DESARROLLO DE UN MÓDULO QUE IMPLEMENTE LAS FUNCIONALIDADES DEL PROTOCOLO RTPS PARA SER UTILIZADO EN APLICACIONES DISTRIBUIDAS DE TIEMPO REAL

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y REDES DE INFORMACIÓN

RUBIO PROAÑO ANDRÉS XAVIER

TELLO GONZÁLEZ ALEJANDRA BEATRIZ

DIRECTOR: ING. XAVIER CALDERÓN, MSc.

Quito, Noviembre 2015

DECLARACIÓN

Nosotros, Andrés Xavier Rubio Proaño, Alejandra Beatriz Tello González, declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que hemos consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedemos nuestros derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

Andrés Xavier Rubio Proaño

Alejandra Beatriz Tello González

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Andrés Xavier Rubio Proaño y Alejandra Beatriz Tello González,bajo mi supervisión.

Ing. Xavier Calderón. MSc.
DIRECTOR DEL PROYECTO

AGRADECIMIENTOS

Doy las Gracias primeramente a Jesús, quien es digno de toda la honra, toda la adoración.

A Mercedes y Xavier, quienes son mis Padres, y a los cuales admiro demasiado, y amo con todo mi corazón.

A Jose, mi hermano, quien ha estado en cada etapa de mi vida, compartiendo momentos felices y tristes y a quien amo demasiado.

A mis abuelos Miguel y Rosa, quienes han sido parte fundamental en mi vida, y han estado pendientes de mi siempre.

A mis abuelos José y Luz, a quienes admiro por su amor hacia Dios, y quienes fueron usados por Dios para llegar a los pies de Cristo.

A Alejandra, por su amistad y entrega en el proyecto.

A la Escuela Politécnica Nacional, en especial a Xavier Calderón, Agustin Méndez, y a Ernesto Jiménez quienes nos han guiado y han estado pendiente de nosotros en el desarrollo del proyecto.

Andrés Xavier Rubio Proaño

AGRADECIMIENTOS

A Dios, por siempre estar presente en todo momento de mi vida.

A mis padres: Luz Beatriz González Arciniega y Luis Guilberto Tello Vicuña que me han enseñado a no darme por vencida, porque gracias a ellos pude alcanzar esta meta profesional y por el apoyo y confianza que me brindaron a lo largo de mi vida, porque si no los tuviera a mi lado nunca hubiera llegado hasta aquí.

A mi abuelita Celita por siempre darme su amor incondicional y sentirme apoyada siempre.

A mis hermanos: Luis y Jorge que siempre estuvieron conmigo apoyándome en todo momento, dándome ánimos y que con su cariño pude culminar mis estudios.

A Miguel mi novio, que gracias a su amor, apoyo y confianza he podido culminar esta etapa universitaria, le agradezco por siempre estar conmigo ayudándome siempre.

A toda mi familia porque han sido un pilar importante en el transcurso de mi carrera.

A los amigos con los cuales he llegado a formar un lazo muy importante a lo largo de mi vida, los quiero mucho.

A Andrés el coautor de este proyecto quien con su ayuda y apoyo se logró culminar con este proyecto exitosamente.

A la Escuela Politécnica y a mis profesores en especial al Ingeniero Xavier Calderón que gracias a el me pude involucrar en el proyecto de investigación, a Agustín Méndez por siempre estar dispuesto a darnos una mano con el proyecto de titulación.

Alejandra Beatriz Tello González

DEDICATORIA

Dedicado al Dios eterno,a mi madre, a mi padre y a mi hermano quien siempre a estado conmigo.

Andrés Xavier Rubio Proaño

DEDICATORIA

Dedicado a mi Dios y a las personas que más amo: Celita, Luis, Beatriz, Jorge, Luis A. y Miguel.

Alejandra Beatriz Tello González

CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE ESPACIOS DE CÓDIGO

RESUMEN

El presente Proyecto comprende el desarrollo de un módulo que implementa las funcionalidades del protocolo Real-Time Publish-Suscribe o RTPS para aplicaciones distribuidas de tiempo real.

El módulo contempla una implementación básica del Middleware Data Distributed System o DDS, la implementación del RTPS y la interacción de los mismos, para el funcionamiento adecuado del middleware.

El DDS es el encargado, por medio de operaciones de escritura tal como *write*, de almacenar los datos que el usuario requiera, serializar los datos y anunciar por medio de RTPS que hay una publicación de datos. Además, el DDS es el encargado de leer o interpretar la información que RTPS le provee, por medio de suscripciones a información publicada, es decir, utilizando operaciones de lectura tal como *take* y/o *read*.

El RTPS es el encargado de transportar los datos que genera el DDS; este primeramente anuncia la presencia de entidades y por medio del protocolo de descubrimiento las demás entidades se conocen entre sí. Además, se encarga de encapsular la información provista por el DDS para enviarla a los diferentes destinatarios.

La interacción de DDS con RTPS es proporcionada por un submódulo dentro del RTPS, el cual se encarga del comportamiento del protocolo de comunicación RTPS ante lo que DDS requiera.

En el primer capítulo se presenta el estado actual de los Middlewares de comunicaciones en tiempo real, donde se realiza un estudio de los Middlewares de comunicaciones actuales incluyendo al Middleware DDS y se realizar una comparación entre ellos para determinar las ventajas y desventajas de cada uno. Finalmente se analizan las características y funcionalidades más específicas definidas en el estándar publicado por el Object Management Group u OMG sobre DDS y su interoperabilidad con el protocolo RTPS.

En el segundo capítulo se definen los requisitos necesarios para integrar el protocolo RTPS con el middleware DDS; además, usando la herramienta Wiresharrk se realiza un análisis previo con capturas de paquetes de los diferentes mensajes RTPS y mensajes de descubrimiento RTPS.

En el tercer capítulo se diseña y luego se adapta un módulo que permita la interacción entre RTPS y DDS, que trabaje con el modelo Publicador/Suscriptor, y que se basen en la publicación del estándar DDS-RTPS de la OMG; el cual permite intercambiar mensajes RTPS. Además, se describe el intercambio de mensajes por medio de diagramas de secuencia. También se implementa de acuerdo al diseño cada submódulo que RTPS y DDS requieran para poder interactuar adecuadamente.

En el cuarto capítulo se presentan los resultados tanto pruebas unitarias como una prueba entre 4 computadoras comunicándose por medio del protocolo RTPS. En las pruebas unitarias se comprueba el funcionamiento correcto de las clases y métodos; y en las pruebas con computadoras se crea un ambiente de comunicación intercambiando mensajes de descubrimiento RTPS, para luego intercambiar mensajes RTPS.

Finalmente se presentan las principales conclusiones y recomendaciones que arrojaron el desarrollo y la implementación del sistema.

Adicionalmente en los anexos se incluye un glosario de términos, códigos fuentes del middleware, códigos fuente de la aplicación de escritorio y del chat CORBA, estándares del DDS y del RTPS, y manuales de usuario de las diferentes implementaciones.

PRESENTACIÓN

El presente Proyecto de Titulación se ha realizado con el objetivo de desarrollar un módulo que implemente las funcionlidades del protocolo RTPS, el cual será utilizado en aplicaciones de tiempo real. Las comunicaciones actualmente buscan descentralizar la información por lo que los Middlewares basado en DDS trabajando con comunicación RTPS, proveen una arquitectura capaz de cumplir este objetivo.

Este proyecto forma parte de la investigación realizada por la Universidad Técnica Particular de Loja, la Universidad Politécnica Salesiana, la Escuela Politécnica Nacional y el Consorcio Ecuatoriano para el Desarrollo de Internet Avanzado, el cual se titula “Middleware en tiempo real basado en el modelo publicación/suscripción”, del cual se puede obtener mayor información en la siguiente dirección electrónica: http://www.utpl.edu.ec/proyectomiddleware/. Dentro del proyecto de investigación, este proyecto de titulación se encarga del protocolo de comunicación RTPS y como este debe interactuar con DDS.

En el presente documento el lector tendrá a su disposición la información básica para comprender el funcionamiento de los diferentes middlewares de comunicación incluyendo DDS-RTPS. Además, se presenta la interacción entre las tecnologías DDS y RTPS.

Se tendrá a disposición información relevante sobre el proceso de comunicación que provee la tecnología DDS-RTPS, se analiza profundamente diferentes escenarios por medio de diagramas de secuencia que detallan el funcionamiento y comportamiento del protocolo RTPS.

Finalmente, se tiene una aplicación de comunicación que utiliza las librerías generadas en el proyecto, donde se puede observar tangiblemente el uso del protocolo de comunicaciones RTPS.

CAPÍTULO 1.
MARCO TEÓRICO

1.1. INTRODUCCIÓN

En el presente capítulo se presenta el fundamento teórico necesario para el desarrollo de este proyecto. Inicialmente se describe el estado actual de los Middlewares de comunicación en tiempo real y se presenta un estudio de cada tecnología encontrada incluyendo al Middleware DDS [1]. Posteriormente se realiza una comparación entre las tecnologías descritas anteriormente, donde se detalla las ventajas y desventajas del uso de cada una. Finalmente se analizan características y funcionalidades más específicas definidas en el estándar publicado por el Object Management Group u OMG sobre el Data Distributed System o DDS y su interoperabilidad con el protocolo Real-Time Publish-Suscribe o RTPS.

1.2. MIDDLEWARES

El Middleware es una capa intermedia de software, el cual se encarga de simplificar el manejo y la programación de aplicaciones.El Middleware mantiene transparente al usuario la complejidad de las redes y de sistemas heterogéneos.

En la *Tabla 1.1* se presenta los tipos de middleware que existen en la actualidad, y una breve descripción.

Tabla 1.1. Tipos de Middleware **[2]**

Middleware	Descripción
De Comunicaciones	Abstrae detalles de comunicación y de distribución de datos
De Componentes	Desarrollo de sistemas por medio del uso de módulos se software reutilizables(componentes).
Basado en Modelos	Se basas en un proceso de desarrollo que incluye middlewares de componentes
Adaptativo	Permite la reconfiguración de aplicaciones distribuidas, en términos de uso de recurso, configuraciones de seguridad, etc.
Sensible al contexto	Permite la reconfiguración de aplicaciones distribuidad en tiempo de ejecución.

Se profundiza en el Middlewares de Comunicaciones, el cual proporciona las bases para el desarrollo de Middlewares de alto nivel. Este describible el proceso de comunicación entre nodos que incluye las siguientes características:

- Direccionamiento o asignación de identificadores a entidades con la finalidad de indicar su ubicación.
- Marshalling[1] o transformación de los datos en una representación adecuada para la transmisión sobre la red.
- Envío o la asignación de cada solicitud a un recurso de ejecución para su procesamiento.
- Transporte o establecimiento de un enlace de comunicaciones para el intercambio de mensajes entre redes vía unicast o multicast.

En la Figura 1.1 se puede apreciar los servicios básicos que provee un Middleware.

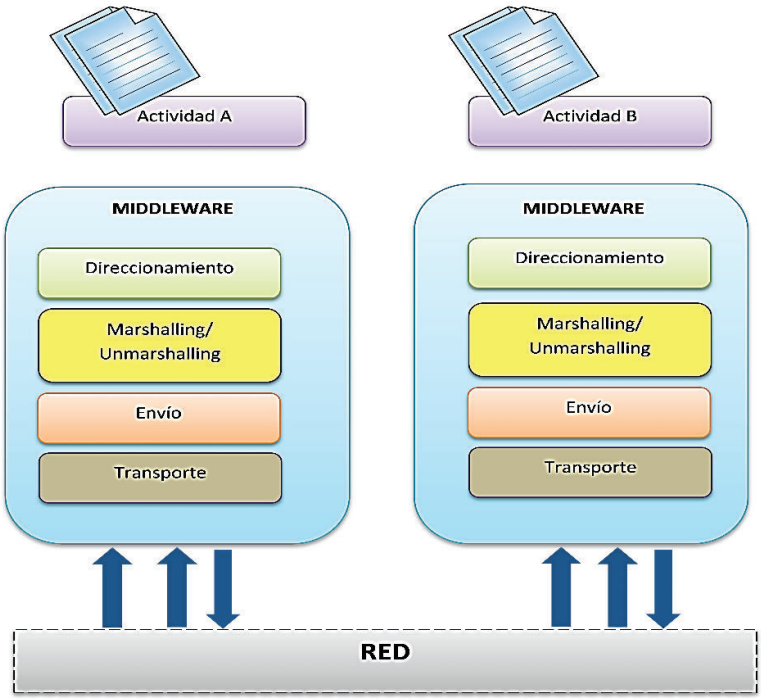


Figura 1.1. Servicios básicos provistos por el Middleware de Comunicación [2]

1.3. SISTEMAS DISTRIBUIDOS

Existen varios modelos de sistemas distribuidos donde la comunicación se abstrae dentro de los Middlewares, entre estos modelos tenemos a:

- *Remote Procedure Calls* o RPC
- *Distribution based on objects* o DOM
- *Distribution based on messages* o MOM

[1] Marshalling, es un mecanismo ampliamente usado para transportar objetos a través de una red.

- *Data-Centric Model*
- *Tuplespaces paradigm*

Dentro de estos modelos de sistemas distribuidos se tiene como ejemplos más representativos las tecnologías detalladas en la Tabla 1.2.

Tabla 1.2. Ejemplos de los modelos de sistemas distribuidos

Modelos de Sistemas Distribuidos	Ejemplos
RPC	• Open Software Foundation/Distributed Computing Environment (OSF/DCE) • Distributed Systems Annex of Ada (DSA) [6]
DOM [7]	• Common Object Request Broker Architecture (CORBA) [8] • Java Remote Method Invocation (RMI) [9] • Distributed Systems Annex of Ada (DSA)
MOM	• Java Message Service (JMS) [10] • Data Distribution Service for Real-Time Systems (DDS) [11]
Data-Centric Model	• Data Distribution Service for Real-Time Systems (DDS)
Tuplespaces paradigm	• JavaSpaces [12] • Simple Scalable Streaming System (S4) [13] • S-NET [14]

1.4. MIDDLEWARES DE TIEMPO REAL

Los sistemas en tiempo real a diferencia de los sistemas de propósito general, se basan en la exactitud y en la disminución de retardos en el intercambio de datos. Por tanto las aplicaciones en tiempo real deben administrar adecuadamente el uso de recursos en el sistema.

Obtener predicciones temporales factibles y fiables dentro de sistemas en tiempo real distribuidos es un reto ya que se debe administrar adecuadamente tanto recursos de red y procesadores.

La tecnología de Middlewares proporciona una abstracción de alto nivel de los servicios ofrecidos por los sistemas operativos, sobre todo los relacionados con la comunicación.

Existen varias fuentes dentro del proceso de distribución de datos que pueden afectar al determinismo dentro de aplicaciones en tiempo real, tal como son los algoritmos de marshalling, la transmisión y recepción en las colas de mensajes, los retraso en la transmisión, etc.

"El Middleware de tiempo real apunta a resolver estos problemas mediante la implementación de mecanismos previsibles, tales como el uso de redes de comunicación en tiempo real de propósito especial o la gestión de parámetros de

planificación[2]. En consecuencia, este tipo de Middleware se dirige no sólo a los problemas de distribución, sino también debe proporcionar a los desarrolladores los mecanismos que permitan que el comportamiento temporal de la aplicación distribuida sea determinístico". [2]

1.4.1. CORBA Y RT-CORBA[3]

"Common Object Request Broquer Architecture o CORBA es un Middleware basado en el modelo de sistema distribuido DOM, el cual utiliza el paradigma Cliente-Servidor y cuya característica principal es facilitar la interoperabilidad entre aplicaciones heterogéneas[4]" [2].

Una visión general de la arquitectura CORBA se muestra en la Figura 1.2, la cual esta integrada por los siguientes componetes:

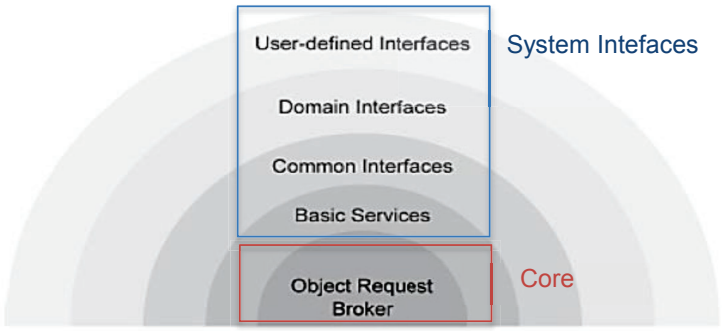


Figura 1.2. Arquitectura de CORBA [2]

- *Object Request Broker (ORB)*, es responsable de coordinar la comunicación entre los nodos cliente y servidor.
- *Interfaces del Sistema*, son un conjunto de interfaces agrupadas en función de su ámbito de aplicación, que incluyen:
    - Una colección de servicios básicos o *Basic Services*, como la concurrencia , la persistencia y la ubicación de los objetos, los mismos que dan soporte al ORB.
    - Un conjunto de interfaces comunes o *Common Interfaces* a través de una amplia gama de dominios de aplicación, por ejemplo, la

[2] Planificación, se refiere al término scheduling.
[3] RT-CORBA, CORBA de tiempo real
[4] Aplicaciones Heterogéneas, se refiere a las aplicaciones codificadas en diferentes lenguajes de programación, ejecución en diferentes plataformas y/o las implementaciones de middlewares desarrolladas por diferentes empresas.

administración de bases de datos, la compresión de la información y la autenticación.

- Un conjunto de interfaces para un dominio particular de la aplicación o *Domain Interfaces*, por ejemplo, las telecomunicaciones, la banca y las finanzas.
- Interfaces definidas por Usuario o *User-defined Interfaces*, las cuales no se encuentran estandarizadas.

- CORBA no proporciona soporte para aplicacione en tiempo real, por lo tanto, esta fue abordada por Real Time CORBA o RT-CORBA [3] por medio de extensiones. Las mas importantes extensiones se describen acontinuacion:
- *Real Time Object Request Broquer o RT-ORB*, una extensión ORB, que permite la creación y la destrucción de entidades en tiempo real..
- *Real Time Portable Object Adapter o RT-POA*, representa una extensión del POA [4] y provee soporte para la configuración de las políticas de tiempo real definidas por RT-CORBA.
- *Prioridad y Asignación de Prioridad*, proporcionan operaciones para asignar prioridades nativas dentro de las prioridades RT-CORBA y viceversa.
- *Servicio de Planificación*, es un servicio que simplifica la configuración de aspectos de sincronización del sistema.

Estas entidades permiten desarrollos de sistemas en tiempo real utilizando CORBA.

1.4.2. THE ADA DISTRIBUTED SYSTEMS ANNEX

El lenguaje de programacion Ada tiene como principal fortaleza que el codigo fuente esta escrito sin tener en cuenta de si va a ser ejecutado en una plataforma distribuida o en un solo procesador. En el lenguaje de programación Ada, cada parte de la aplicación se asigna de forma independiente a cada nodo la cual es llamada partición. Formalmente, de acuerdo al manual de referencia de Ada, “una partición es un programa o parte de un programa que puede ser invocado desde fuera de la aplicación Ada. [5] [2]”.

La DSA solamente define dos tipos de particiones: activos, los cuales se ejecutan paralelamente uno con otro en direcciones de memoria separadas,

además, permiten la comunicación por medio del subsistema de comunicación de particiones; y pasivos, los cuales no tienen un hilo de control.

Los componentes de alto nivel del modelo de distribución propuesto por la DSA están ilustrados en la Figura 1.3. Esta figura representa el diagrama de secuencia de una llamada remota síncrona entre dos particiones

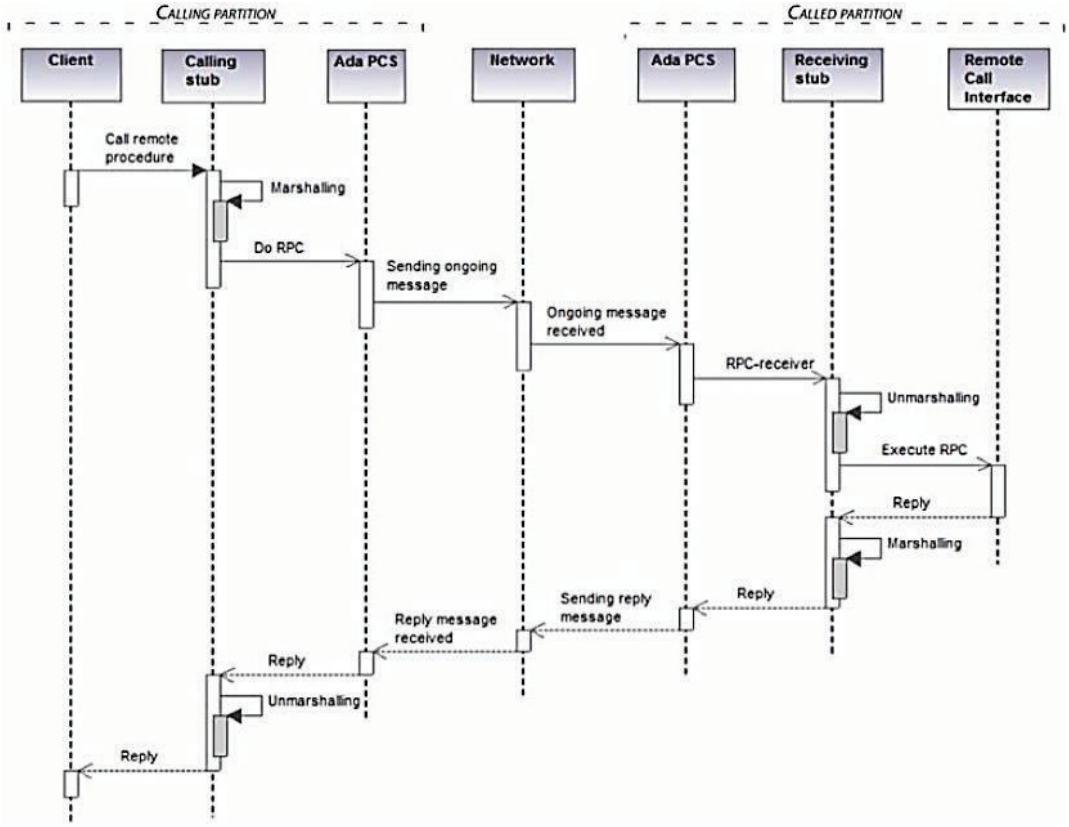


Figura 1.3. Diagrama de secuencia de una llamada remota síncrona [2]

La DSA simplifica la manera en que se crean los sistemas distribuidos, pero tiene todavía varios problemas que afectan al determinismo, el cual es indispensable en aplicaciones en tiempo real.

1.4.3. THE DISTRIBUTED REAL-TIME SPECIFICATION FOR JAVA

Java fue diseñado para sistemas de proposito general, lo cual genera varios inconvenientes al momento de desarrollar aplicaciones en tiempo real. El mayor inconveniente se relaciona a la gestión de recursos de procesador.

Distributed Real-Time Specification for Java o DRTSJ [2] es la propusta que integra dos tecnologías de de Java y que permite tener sistemas distribuidos en tiempo real, estas dos tecnologías son:

- *Real-Time Specification for Java o RTSJ* [2], define nuevas funcionalidades para que el lenguaje Java se pueda usar en sistemas en tiempo real, para esto se generaron nuevas bibliotecas que permitieron mejorar a la maquina virtual de Java y de esta manera convertirla en una maquina virtual de Java en tiempo real.
- *Remote Method Invocation o RMI* [2], brinda una visión general del alto nivel de los componentes implicados en la arquitectura. A continuación se muestra en la Figura 1.4 el diagrama de secuencia de una llamada remota asíncrona entre un cliente y un servidor.

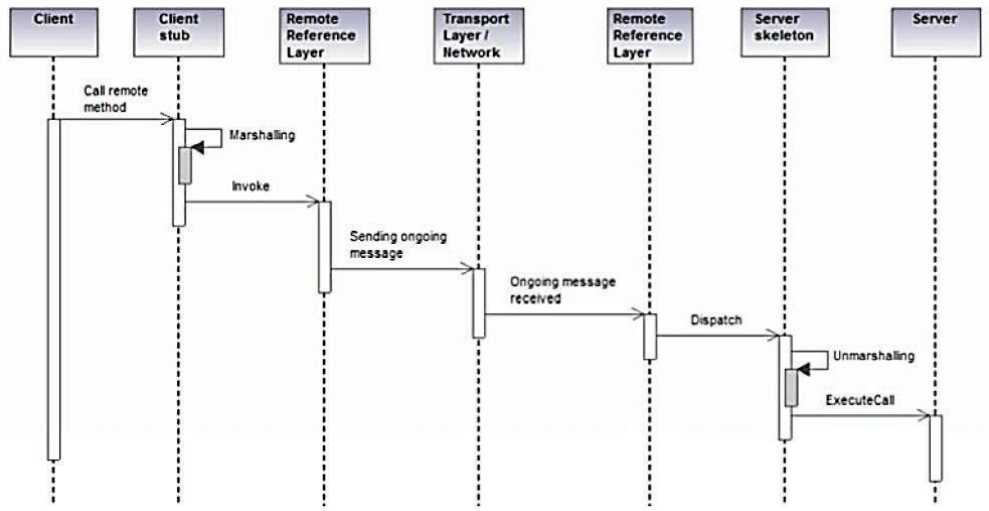


Figura 1.4. Diagrama de secuencia de una llamada remota asíncrona [2]

Java no define formalmente las especificación DRTSJ aunque existe un primer borrador en el sitio web del grupo de trabajo.

1.4.4. THE DATA DISTRIBUTION SERVICE FOR REAL-TIME SYSTEMS

El DDS [1] tiene como objetivo facilitar el intercambio de datos a través del paradigma publicador-suscriptor, es decir la comunicación se centra en los datos.

DDS pertenece al tipo de middleware de comunicación que utiliza la arquitectura data-centric, por lo tanto existe un enfoque centrado en los datos, donde el middleware es “consciente” de la información intercambiada.

El DDS se encuentra definido dentro de la OMG, el cual proporciona un sistema de datos estructurales, nuevas representaciones de tipos de datos, diferentes formatos de serialización o de codificación, y una nueva API para la gestión de los tipos de datos en tiempo de ejecución.

Conceptualmente, DDS abstrae los tipos de datos y donde se pueden producir y consumir información por medio de un Publicador y un Suscriptor respectivamente. Para que el intercambio de datos sea adecuado, se definen varias entidades que deben participar en el proceso de comunicación. Aquellos participantes que deseen publicar información lo deben realizar por medio de la entidad *DataWriter* o DW. De igual manera, los participantes que deseen recibir o suscribirse a infomacion lo deben realizar por medio de la entidad *DataReader* o DR. Estas dos entidades descritas son contenidas por dos entidades superiores los *Publisher* y los *Suscriber*. Además, estas entidades deben compartir información en cuanto a políticas de calidad de servicio o QoS y deben compartir un mismo dominio o canal de comunición para poder comunicarse entre sí.

Algo muy importante en cuanto a las entidades de DDS, es que estas basan sus publicaciones y suscripciones en base a un *Topic*, el cual define un tipo de dato que podría llegar a ser de interés para otros suscriptores, es decir que se puede publicar información de interés y por medio del *Topic* o tema, otros suscriptores pueden buscar suscribirse a esta información.

En la Figura 1.5 se muestra un sistema distribuido que consta de tres participantes en un sólo dominio y dos tópicos.

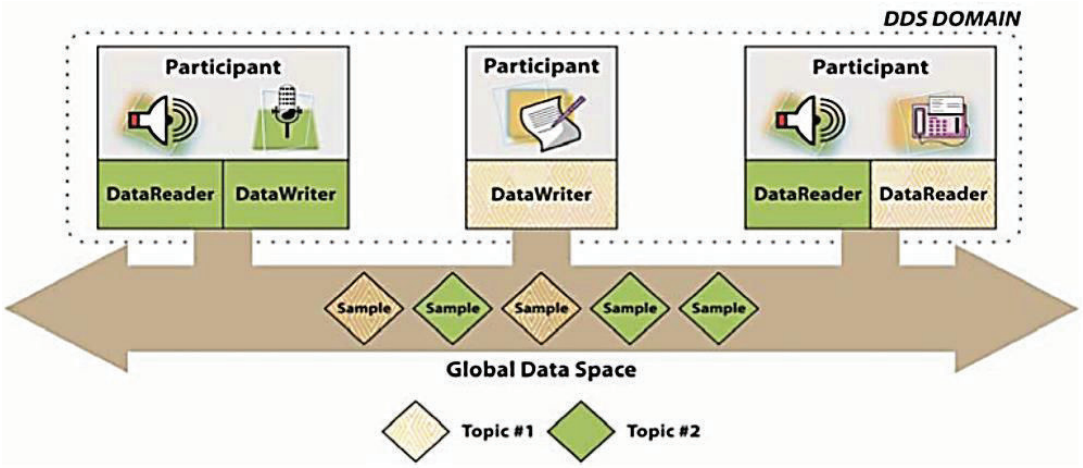


Figura 1.5. Sistema Distribuido que consta de tres participantes en un sólo dominio [2]

Ambos tópicos tienen un sólo DW, el cual es el encargado de generar nuevos grupos de datos. Sin embargo, las actualizaciones sucesivas del Tópico #1 sólo serán recibidos por un sólo DR, mientras que las nuevas muestras del Tópico #2 serán recibidos por dos DR.

Los Publicadores y Suscriptores no están obligados a comunicarse directamente entre sí, pero están relacionados de forma flexible en función de lo siguiente [2]:

- *Time,* los datos pueden ser almacenados y recuperados después, por ejemplo, cuando un nuevo Suscriptor se une al sistema distribuido y requiere información sobre el anterior estado del sistema.
- *Space,* porque para los Publicadores de datos no es necesario saber acerca de cada receptor individual, mientras que los Suscriptores no necesitan conocer la fuente de origen de los datos, los Publicadores y Suscriptores no se conocen entre sí.

Como se mencionó anteriormente, el desarrollo de sistemas distribuidos con DDS está unido a otra especificación que establece las principales directrices para la realización de la comunicación entre las entidades: el DDSI. Este protocolo tiene como objetivo garantizar la interoperabilidad entre diferentes implementaciones, utilizando el estándar del protocolo RTP [6] en tiempo real del Publicador-Suscriptor, junto con la Representación de Datos Común o CDR, el cual se define en CORBA [4]. Aunque esta especificación se centra en las redes IP, ningún otro protocolo de red en tiempo real puede ser usado. Por último, aunque DDS ha sido diseñado para ser escalable, eficiente y predecible, pocos investigadores han evaluado sus capacidades en tiempo real [7]. Sin embargo, se considera una tecnología madura y ya se ha desplegado en varios escenarios en tiempo real, tales como sistemas de Defensa Nacional [8], Automatización [9], o Espacio [10].

1.5. COMPARACIÓN ENTRE LAS TECNOLOGÍAS DE MIDDLEWARES DE COMUNICACIÓN EN TIEMPO REAL.

Después de analizar los diferentes estándares de distribución orientadas al desarrollo de aplicaciones en tiempo real, se hallan las semejanzas y diferencias entre estas tecnologías, así como se evalúa su capacidad para ser utilizados en los sistemas de tiempo real. Para una mejor comparación, primeramente se explica una serie de requisitos que un estándar de distribución para sistemas de tiempo real deben considerar [2]:

- *El soporte para la planificación en procesadores y redes* [2], es necesario dentro los middlewares de tiempo real tener el mayor control sobre los proesos y la planificación de uso de recursos en procesadores y en redes.

Específicamente, se debe evaluar como se ordena el acceso simultaneo de los hilos a los procesadores, y de mensajes a la red de comunicación.

- *Control de parámetros de planificación,* debe proveer control sobre los hilos en los procesadores y sobre los mensajes en la red, todo por medio de parámetros que permitan esto.
- *Gestión de hilos o patrones de concurrencia,* se refiere a la forma como el middleware gestiona a los hilos, si este requiere de un paradigma de multi-hilos y como se permite tener concurrencia.
- *El acceso controlado a recursos compartidos,* esto se puede lograr a través de la aplicación de protocolos de sincronización, como locks, mutex y semáforos.

Por tanto se puede observar que la gestión de los hilos para procesadores y los mensajes para redes de comunicaciones son requisitos muy importantes al momento de comparar tecnologías de middleware.

Tabla 1.3. Recursos del Procesador dentro de los Middlewares

Middleware	Recursos del Procesador			
	Políticas de Planificación	Configuración de los Parámetros de Planificación	Patrones de Concurrencia	Protocolos de Sincronismo
RT-CORBA	FPS EDF LLF MAU	Client_Propagated Server Declared Priority_Transfer	Threadpool	Requerido
Ada DSA	FPS No apropiable Round-robin EDF	Implementación definida	Implementación definida	Priority Ceiling
DDS	Implementación definida	Implementación definida	Implementación definida	Implementación definida
Java DRTSJ	FPS Usuario definido	Implementación definida	Implementación definida	Priority inheritance Priority Ceiling

1.5.1. GESTIÓN DE LOS RECURSOS DEL PROCESADOR

Dentro de la gestión de los recursos del procesador, es importante conocer como cada tecnología maneja las políticas de planificación, los patrones de concurrencia, saber como se realiza la configuración de parámetros de planificación y los protocolos de sincronismo. En el caso de las políticas de planificación, se debe conocer cuales mecanismo son provistos por el middleware para seleccionar una política específica de planificación para las entidades planificables responsables de atender a los servicios remotos. En el caso de los patrones de concurrencia se

busca determinar cuál hilo es responsable para el envío o recepción de peticiones/datos remotos.

En la Tabla 1.3, se muestra como los diferentes middlewares soportan cada una de las características presentadas anteriormente en cuanto a los recursos del procesador.

1.5.2. GESTIÓN DE RECURSOS DE RED

Dentro de la gestión de los recursos de red, es importante conocer como cada tecnología maneja las políticas de planificación y la configuración de parámetros de planificación.

En la Tabla 1.4, se muestra como los diferentes middlewares soportan cada una de las características presentadas anteriormente en cuanto a los recursos de red.

Tabla 1.4. Recursos de Red dentro de los Middlewares

Middleware	Recursos de Red	
	Políticas de Planificación	Configuración de los Parámetros de Planificación
RT-CORBA	Implementación definida	Implementación definida
Ada DSA	Implementación definida	Implementación definida
DDS	FPS	Prioridad de Transporte
Java DRTSJ	Implementación definida	Implementación definida

En el caso de DDS, la especificación sólo considera redes basadas en políticas de planificación con prioridad fija o FPS y excluye redes como time-triggered networks[5] que se utiliza en industrias. En cuando a la configuración de parámetros de planificación esta puede ser tratada mediante la modificación de la definición del parámetro *Transport_Priority*.

Un factor que es importante es el tamaño de los mensajes de red ya que en DDS es fundamental para el diseño de aplicaciones, pero cuando se agrego la capa *DDS Interoperability Wire Protocol o* DDSI a DDS se pudo soportar multiples mensajes , que incluyen no solo al metatráfico, sino también a datos de usuario.

[5] *Time-triggered networks*, es una red basada en un protocolo abierto para sistemas controlados, diseñada para aplicaciones industriales y redes de vehículos.

Este mecanismo es poco eficiente para minimizar el tiempo de respuesta promedio, no suele ser adecuado para sistemas de tiempo real que tienen por objetivo garantizar los límites de la latencia en cada flujo de red. Por lo tanto, depende de la implementación proporcionar los medios para definir el tamaño máximo de un mensaje *DDS Interoperability Wire Protocol* o DDSI.

Finalmente, *"la presencia de los mensajes y operaciones que pertenecen al Middleware puede provocar un incremento en los tiempos de respuesta de aplicaciones críticas. Aunque, esta sobrecarga depende casi exclusivamente de cada aplicación, este efecto es más significativo en estándares tal como DDS, el cual define un conjunto de entidades que pueden consumir recursos del procesador y de la red"* [2].

1.5.3. CUADRO COMPARATIVO DE LAS DIFERENTES TECNOLOGÍAS

Para resumir las características de cada tecnología presentada, en la Figura 1.6 se muestra la evolución de estos estándares de distribución de tiempo real.

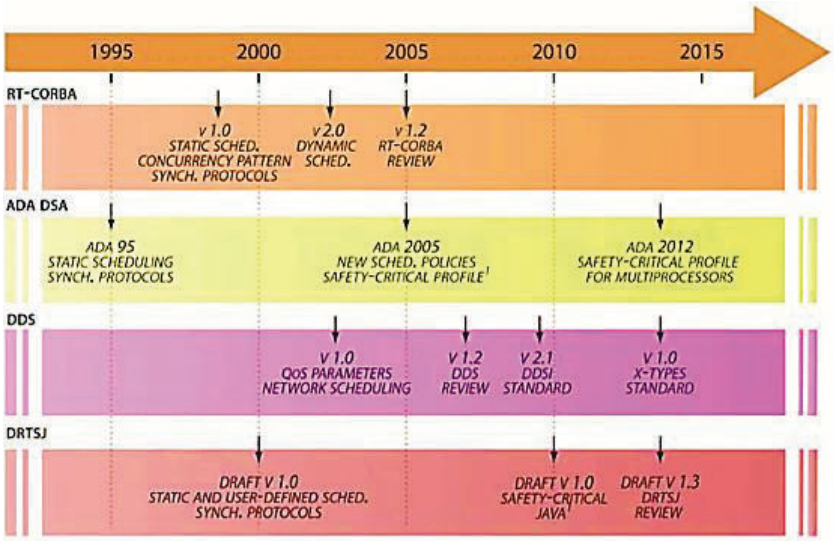


Figura 1.6. Línea de tiempo en los estándares de tiempo real [2]

En la Tabla 1.5 se resume el análisis de acuerdo al grado de soporte de los requerimientos propuestos en la sección anterior, como las políticas de planificación, los parámetros de configuración de la planificación, los patrones de concurrencia, y el acceso controlado a los recursos compartidos.

La mayoría de estas características, son requeridas en el peor caso del comportamiento temporal, aunque otras permanecerán abiertas a las implementaciones, como las mostradas en la Tabla 1.5. En consecuencia, la

elección de un Middleware particular, determina no sólo el rendimiento de las aplicaciones, sino también su previsibilidad, y por lo tanto la capacidad de cumplir con la separación máxima de tiempo entre dos actualizaciones. La elección del patrón de concurrencia para el procesamiento de llamadas remotas o datos entrantes es particularmente relevante, aunque esta característica dependa de las implementaciones. Para lo cual el Middleware escogido de acuerdo a la priorización de transporte, es DDS.

Tabla 1.5. Capacidades de Tiempo-Real de los Estándares de Distribución [2].

Middleware	Recursos del Procesador				Recursos de Red	
	Políticas de Planificación	Configuración de los Parámetros de Planificación	Patrones de Concurrencia	Protocolos de Sincronismo	Políticas de Planificación	Configuración de los Parámetros de Planificación
RT-CORBA	FPS EDF LLF MAU	Client_Propagated Server Declared Priority_Transfer	Threadpool	Requerido	Implementación definida	Implementación definida
Ada DSA	FPS No apropiable Round-robin EDF	Implementación definida	Implementación definida	Priority Ceiling	Implementación definida	Implementación definida
DDS	Implementación definida	Implementación definida	Implementación definida	Implementación definida	FPS	Prioridad de Transporte
Java DRTSJ	FPS Usuario definido	Implementación definida	Implementación definida	Priority inheritance Priority Ceiling	Implementación definida	Implementación definida

1.6. CARACTERÍSTICAS Y FUNCIONALIDADES DEL DDS

1.6.1. CARACTERÍSTICAS

1.6.1.1. Arquitectura

Una arquitectura Publicador-Suscriptor promueve un bajo acoplamiento en la arquitectura de datos y es flexible y dinámica; esta es fácil de ser adaptada y extendida a sistemas basados en DDS. El modelo Publicador-Suscriptor conecta a los generadores de información anónima o el Publicador con los consumidores de información o Suscriptor. La mayoría de aplicaciones distribuidas que utilizan el modelo Publicador-Suscriptor están compuestos de procesos, cada uno corriendo por espacios de memoria separados y comúnmente en diferentes computadoras. Se denominará a cada uno de estos procesos como participantes. Un participante puede simultáneamente publicar y suscribir información. El aspecto definido en el modelo Publicador-Suscriptor se refiere al desacoplamiento tanto en espacio, tiempo y flujo entre publicador y suscriptor.

La información transferida por comunicaciones de tipo *data-centric* pueden ser clasificadas en: señales, flujos, y estados.

- Las señales, representan los datos que están en continuo cambio, por ejemplo la lectura de un sensor. Las señales pueden trabajar análogamente como IP , es decir, haciendo su mejor esfuerzo.
- Los flujos, representan capturas de valores de un *data-object* que pueden ser interpretados en el contexto de una captura previa. Los flujos necesitan tener confiabilidad.
- Los estados, representan el estado de un conjunto de objetos o sistemas codificados como el valor más actual de un conjunto de atributos de datos o de estructuras de datos. El estado de un objeto no cambia necesariamente en cualquier periodo. Los rápidos cambios pueden ser seguidos por intervalos largos sin cambios en el estado. Los participantes que requieren información del estado, se encuentran típicamente interesados en el valor más actual. Sin embargo, como el estado puede no cambiar a lo largo del tiempo, el Middleware puede necesitar asegurar que el estado más actual entregue confiabilidad. En otras palabras, si el valor se ha perdido, entonces no es siempre aceptable esperar hasta que el valor vuelva a cambiar.

El objetivo del estándar DDS, es facilitar una distribución eficiente de la información en un sistema distribuido. Los participantes usando DDS pueden leer y escribir datos eficientemente y naturalmente[6] con un tipo de interfaz. Por debajo, el Middleware DDS distribuye los datos, por lo tanto cada lector puede acceder a los valores más actuales. Por tanto, el servicio crea un espacio de datos global donde cualquier participante puede leer como escribir. Este también crea un nombre de espacio que permite a los participantes encontrar y compartir objetos.

El objetivo de DDS son los sistemas de tiempo real, el API y QoS han sido escogidos para balancear el comportamiento predecible y la eficiencia/rendimiento de la implementación. Una visión general de la arquitectura se muestra en la **¡Error! o se encuentra el origen de la referencia.**.

El estándar DDS describe dos niveles de interfaces y un nivel de comunicaciones:

[6] Naturalmente, se refiere a que la interfaz puede ser similar a una ya usada por variables locales lectura/ escritura.

- Un nivel bajo denominado *Data-Centric Publish-Subscribe o DCPS,* el cual está orientado a la entrega eficiente de información adecuada a los destinatarios correctos.
- Un nivel opcional alto denominado *data-local reconstruction layer o DLRL,* el cual permite una integración simple a la capa de aplicación.
- El nivel de comunicaciones se denomina DDS *Interoperability Wire Protocol*, el cual opera con el protocolo RTPS.

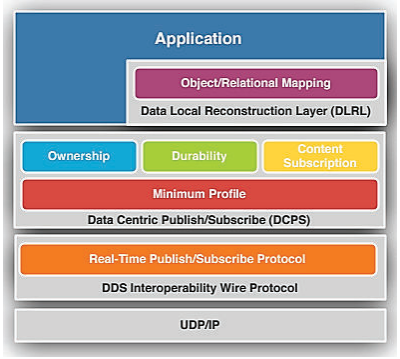


Figura 1.7. Arquitectura del Middleware DDS [11]

1.6.1.1.1. DCPS o Data-Centric Publish-Subscribe

La comunicación se lleva a cabo con la ayuda de las siguientes entidades: *DomainParticipant, DataWriter, DataReader, Publisher, Subscriber, y Topic.* Todas estas entidades son representadas por clases que extienden a la clase *DCPSEntity,* la cual muestra su capacidad de ser configurada a través de las políticas de QoS, ser notificado por medio de eventos, y soportar condiciones que pueden ser esperadas por la aplicación. Cada especialización de una clase base *DCPSEntity* tiene un correspondiente *Listener* especializado, el cual se utiliza para escuchar el canal, y un conjunto de valores de *Políticas de Calidad de Servicio* o *QoSPolicy* que son deseables.

El Publicador será el responsable de la distribución de datos. Podrá publicar los datos de los diferentes tipos de datos. Un *DataWriter* actuará como un *typed*[7] de acceso a una publicación. Un *DataWriter* se encargará de comunicar a un publicador la existencia de datos de un tipo dado, es decir, cuando los valores de los datos hayan sido comunicados al publicador a través de un *DataWriter* apropiado, será responsabilidad del publicador realizar la distribución.

[7] Typed, significa que cada objeto DataWriter es dedicado a una aplicación de tipo de dato

Un *DataWriter* es la cara al Publicador, los participantes usan *DataWriter* para comunicar el valor y los cambios en los datos de un determinado tipo. Una vez que la nueva información ha sido comunicada al Publicador, es la responsabilidad del Publicador determinar cuándo es apropiado emitir el correspondiente mensaje y llevar a cabo realmente la emisión, es decir, el Publicador hará esto de acuerdo a su calidad de servicio o a la QoS asociada al correspondiente *DataWriter,* y/ o a su estado interno.

El Suscriptor será el responsable de recibir los datos publicados y podrá recibir y despachar datos de los diferentes tipos especificados. Para acceder a los datos recibidos, la aplicación deberá utilizar un *typed DataReader* adjunto al suscriptor.

La asociación de un objeto *DataWriter*, el cual representa a una publicación, con el objeto *DataReader*, que representa la suscripción, es hecha por la entidad *Topic.*

Un Topic representará la unidad de información que puede ser producida o consumida; estará compuesta por un tipo, un nombre único y un conjunto de políticas de calidad de servicio, como se muestra en la Figura 1.8, que se utilizará para controlar las propiedades no funcionales asociadas con el Topic. Es decir, que si no se especifica de manera explícita las políticas de QoS, la aplicación DDS utilizará valores predeterminados por la norma.

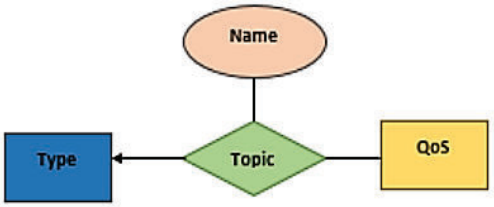


Figura 1.8. Objeto Topic y sus componentes. [11]

Los Topic encajarán entre las publicaciones y suscripciones. Las publicaciones deberán ser conocidas de tal manera que las suscripciones puedan referirse a ellas sin ambigüedades. El Topic tendrá el propósito de:

- Asociar un nombre único en el dominio, es decir, el conjunto de aplicaciones que se comunican entre sí
- Asociar un tipo de datos, y la calidad de servicio en relación con los datos en sí.

Un tipo Topic DDS se describe por una estructura IDL que contiene un número arbitrario de campos cuyos tipos podrían ser: tipo *primitivo*, como se muestra en la

Tabla 1.6, un tipo *template,* como se muestra en la Tabla 1.7, o un tipo *compuesto,* como se muestra en la Tabla 1.8.

Tabla 1.6. Tipos IDL primitivos.

Tipos primitivos
boolean
wchar
short
unsigned short
long
unsigned long long
float
double
long double

Tabla 1.7. Tipos IDL template.

Tipos Template	Ejemplos
string <length= UNBOUNDED>	string s1; string<32> s2;
wstring<length= UNBOUNDED>	wstring ws1; wstring<64> ws2;
sequence<T, length = UNBOUNDED>	sequence<octect> oseq; sequence<octect, 1024> oseq1k;
	Sequence<MyType> mtseq; Sequence<MyType, 10> mtseq10;
fixed<digits, scale>	fixed<5, 2> fp; //d1d2d3.d4d5

Tabla 1.8. Tipos IDL compuestos.

Tipos construidos	Ejemplos
enum	enum Dimension {1D, 2D, 3D, 4D}
struct	struct Coord1D {long x;}; struct Coord2D {long x; long y;}; struct Coord3D {long x; long y; long z;}; struct Coord4D {long x; long y; long z; unsigned long long t;};

union	union Coord switch (Dimension) {
	case 1D:
	Coord1D c1d;
	case 2D:
	Coord2D c2d;
	case 3D:
	Coord3D c3d;
	case 4D:
	Coord4D c4d;
	};

Los *Topic* deberán ser conocidos por el Middleware DDS. Los objetos del *Topic* serán creados utilizando las operaciones de creación proporcionadas por el *DomainParticipant*.

DCPS puede también soportar suscripciones con filtrado del tipo *content-based*[8] por medio de un filtro el cual corresponde a las políticas de QoS. Esta es una característica opcional ya que el filtrado del tipo *content-based* ocupa muchos recursos e introduce retardos que son difíciles de predecir.

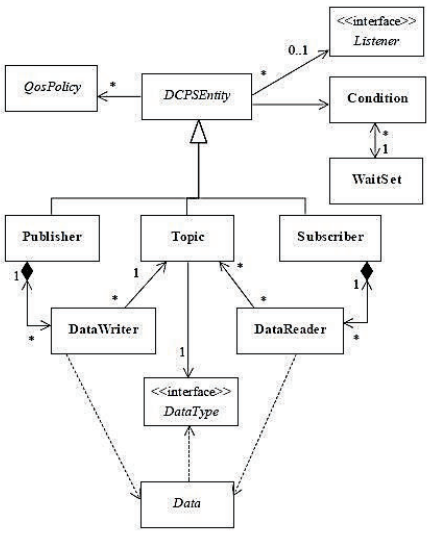


Figura 1.9. Modelo DCPS y sus relaciones [12]

La capa DCPS se encuentra compuesta de cinco módulos: infraestructura, tópico, publicación, suscripción, y dominio.

- El módulo Infraestructura contiene las clases *DCPSEntity, QoSPolicy*, *Listener, Condition,* y *WaitSet.* Esta abstracción de clases soporta dos estilos de interacción: *notification-based* y w*ait-based.* Estos implementan interfaces que se mejoran en otros módulos.
- El módulo *Topic-Definition* contiene a las clases *Topic* y al *TopicListener* y generalmente todo lo que es necesario para que la aplicación defina sus tipos de datos, cree tópicos, y asocie QoS a estos.
- El módulo publicación contiene las clases Publicador, el *DataWriter* y el *PublisherListener*, y generalmente todo lo que es necesario en el lado de la publicación.
- El módulo suscripción contiene las clases Suscriptor, el *DataReader*, y el *SubscriberListener,* y generalmente todo lo que es necesario en el lado de la suscripción.

[8] *Content-based filtering*, es una técnica de filtro de información, utilizando distintos ítems de información

- El módulo de dominio contiene la clase *DomainParticipantFactory*, que actúa como una entrada al servicio, así también como la clase *DomainParticipant,* la cual es un contenedor para otros objetos.
  En la Figura 1.9 se muestra el modelo DCPS y sus relaciones.

Mecanismos y Técnicas para el alcance de la información

Los dominios y las particiones serán la manera de organizar los datos. El Topic DDS permitirá crear *topics*, que limitará los valores que pueden tomar sus instancias. Al suscribirse a un *topic* una aplicación, sólo recibirá, entre todos los valores publicados aquellos valores que coincidan con el filtro del *topic*. Los operadores de los filtros y condiciones de consultas se muestran en la siguiente Tabla 1.9.

Tabla 1.9. Operadores para Filtros DDS y Condiciones de Consulta

Operador	Descripción
=	Igual
<>	Diferente
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
BETWEEN	Entre y rango inclusivo
LIKE	Búsqueda para un patrón

Estos limitarán la cantidad de memoria utilizada por el Middleware.

1.6.1.1.2. DLRL o Data Local Reconstruction Layer

La capa DLRL es opcional, el propósito de esta es proveer un acceso más directo para el intercambio de datos, perfectamente integrado con constructores y lenguaje nativo. La orientación a objetos ha sido seleccionada por todos los beneficios de ingeniería de software.

DLRL está diseñada para permitir al desarrollador de aplicaciones usar características subyacentes de DCPS. Sin embargo, este puede tener conflicto con el propósito principal de esta misma capa, ya que es de fácil uso y permite una integración completa dentro de la aplicación. Por lo tanto, algunas características de DCPS pueden ser sólo utilizadas a través de DCPS y no ser accesibles al DLRL.

DLRL permite que una aplicación describa objetos por medio de: métodos y atributos; los atributos pueden ser locales, es decir, que no participan en la distribución de datos; o pueden ser compartidos, es decir, que participan en la

distribución de datos y estos se encuentran asociados a las entidades DCPS. DLRL gestionará los objetos DLRL en una caché, por ejemplo dos diferentes referencias a un mismo objeto o un objeto con la misma identidad no apuntará a la misma dirección de memoria.

DLRL define dos tipos de relaciones entre objetos DLRL:

- *Herencia,* la cual organiza las clases DLRL
- *Asociaciones,* las cuales organizan las instancias DLRL.

Una herencia simple es permitida entre objetos DLRL. Cualquier objeto que herede de un objeto DLRL se convierte en un objeto DLRL. Los objetos DLRL pueden, además, heredar de cualquier lenguaje de objetos nativos.

El extremo de la asociación se lo denomina relación.

Las asociaciones soportadas son:

- *Una relación a uno*, es llevada a cabo por un atributo de un sólo valor, es decir, hace referencia a un objeto.
- *Una relación a varias,* es llevada a cabo por atributos de varios valores, es decir, una colección de referencias a varios objetos.

Las relaciones soportadas son:

- *Relaciones de uso plano,* las cuales no tienen impacto en el ciclo de vida del objeto.
- *Composiciones,* constituyen el ciclo de vida del objeto.

La

Figura 1.10 representa el modelo del DLRL

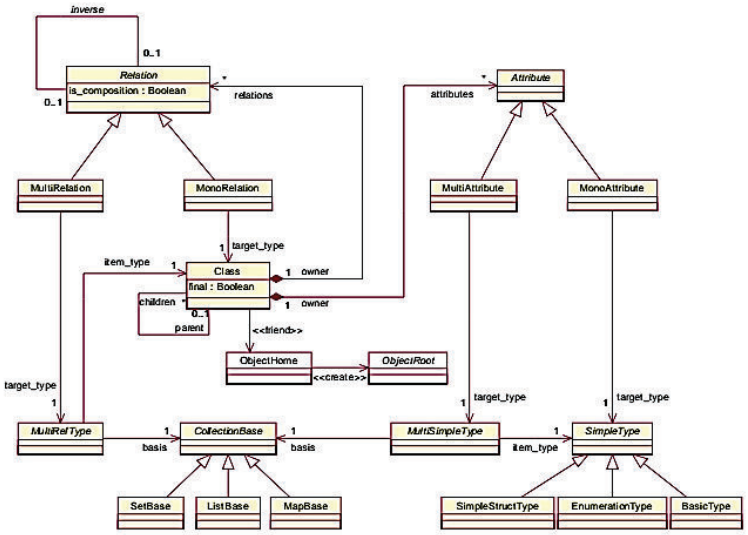


Figura 1.10. Modelo DLRL [1].

1.6.1.1.3. DDS Interoperability Wire Protocol

"RTPS fue específicamente desarrollado para soportar los requerimientos únicos de los sistemas de datos distribuidos. Como uno de los dominios de aplicación a los que apunta DDS, es la comunidad de automatización industrial que define requerimientos para un protocolo de Publicación-Suscripción, el cual es compatible con DDS" [13].

El protocolo RTPS soportar transporte multicast y transporte no orientado a la conexión tal como UDP.

Las principales características del protocolo RTPS son descritas en la *Tabla 1.10*.

Tabla 1.10. Características de RTPS

Caracteristicas	Explicación
Propiedades para el rendimiento y calidad de servicio	Permiten tener comunicación segura entre el Publicador-Suscriptor y que haga el mejor esfuerzo para aplicaciones de tiempo real sobre redes IP.
Tolerancia a fallos	Permite la creación de redes sin puntos de fallos
Extensibilidad	Permite que el protocolo sea extendido y mejorado con nuevos servicios con compatibilidad hacia atrás e interoperabilidad.
Conectividad plug-and-play	Permite que las nuevas aplicaciones y servicios sean automáticamente descubiertos y las aplicaciones puedan unirse y dejar la red en cualquier momento sin necesidad de reconfiguración.
Configurabilidad	Permite el balanceo de requerimientos para la confiabilidad y la puntualidad de cada entrega de datos.
Implementación parcial	Capacidad para permitir que los dispositivos implementen un submodulo del protocolo y que aun así participen en la red.
Escalabilidad	Permite a sistemas que potencialmente escalen en redes extensas.
Seguridad de tipo de datos	Previene errores en la programación de aplicaciones que puedan comprometer las operaciones en los nodos remotos.

Dentro del estándar del protocolo RTPS se describe en términos de un Plataform Independent Model o PIM, y un Platform Specific Model o PSM. El PIM RTPS contiene cuatro módulos los cuales son [13]:

- *Estructura,* el cual define los actores del protocolo.

- *Mensajes,* define un conjunto de mensajes que cada extremo puede intercambiar.
- *Comportamiento,* define un conjunto de interacciones legales en el intercambio de mensajes y como estos afectan el estado de la comunicación en los extremos.
- *Descubrimiento,* define como las entidades son automáticamente descubiertas y configuradas.

En la Figura 1.11 se puede observar la interacción del protocolo RTPS con DDS.

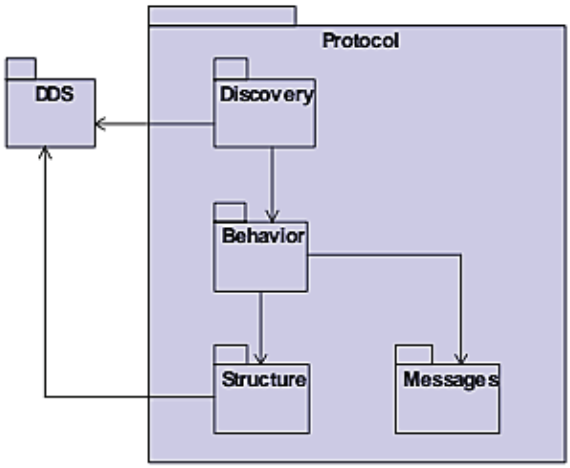


Figura 1.11. Interacción del protocolo RTPS con DDS [13].

1.6.1.2. Descubrimiento

DDS provee descubrimiento dinámico en los Publicadores y Suscriptores. Este descubrimiento dinámico hace que las aplicaciones de DDS sean extensibles. Esto significa que la aplicación no tiene que conocer o configurar a los extremos para que se comuniquen porque estos son descubiertos por DDS. DDS descubrirá que en un extremo está publicando datos, suscribiéndose a datos, o ambos. Este descubrirá el tipo de datos que está siendo publicado o suscrito. También se descubrirá las características de comunicación ofrecida por los publicadores y las características solicitadas por los suscriptores. Todos estos atributos se toman en consideración durante el descubrimiento dinámico y la asociación de participantes DDS.

Los participantes DDS pueden estar en la misma máquina o a través de la red: la aplicación usa el mismo API DDS para la comunicación. Porque no hay

necesidad de conocer o configurar direcciones IP o tomar en cuenta las diferentes arquitecturas de computadores.

1.6.1.3. Políticas de Calidad de Servicio

El servicio de Distribución de Datos confía en el uso de Calidad de Servicio a medida de los requerimientos de una aplicación para el servicio. La Calidad de Servicio actualmente tiene un conjunto de características que maneja un comportamiento dado del servicio.

La descripción de todas las políticas de QoS soportadas por el servicio DCPS se encuentran definidas en el estándar.

Tabla 1.11. Políticas de QoS del DDS [11].

QoS Policy	Applicability	Modifiable	
DURABILITY	T, DR, DW	N	Data Availability
DURABILITY SERVICE	T, DW	N	
LIFESPAN	T, DW	Y	
HISTORY	T, DR, DW	N	
PRESENTATION	P, S	N	Data Delivery
RELIABILITY	T, DR, DW	N	
PARTITION	P, S	Y	
DESTINATION ORDER	T, DR, DW	N	
OWNERSHIP	T, DR, DW	N	
OWNERSHIP STRENGTH	DW	Y	
DEADLINE	T, DR, DW	Y	Data Timeliness
LATENCY BUDGET	T, DR, DW	Y	
TRANSPORT PRIORITY	T, DW	Y	
TIME BASE FILTER	DR	Y	Resources
RESOURCE LIMITS	T, DR, DW	N	
USER_DATA	DP, DR, DW	Y	Configuration
TOPIC_DATA	T	Y	
GROUP_DATA	P, S	Y	

Una política de QoS puede ser establecida en todos los objetos *DCPSEntity.* En varios casos para que la comunicación funcione apropiadamente, una política de QoS en el lado del Publicador debe ser compatible con la política correspondiente en el lado del Suscriptor. Por ejemplo, si un suscriptor pide confiabilidad en la información recibida, mientras que el correspondiente publicador define una política de mejor esfuerzo, la comunicación no se establecerá tal como fue requerida. Para abordar este problema y mantener el desacople deseable de la publicación y la suscripción en medida de lo posible, la especificación para políticas de QoS sigue el modelo Suscriptor-Solicitado, Publicador- Ofertado. En la Tabla 1.11 se muestra las políticas de calidad de servicio, donde se puede observar en que entidades las QoS son aplicables como *DataWriter*, *DataReader*, *Topic*,

Pubisher, *Subscriber*; además de cuales de las QoS pueden ser modificadas en tiempo de ejecución.

En este modelo en el lado del Suscriptor se puede especificar una lista ordenada de las peticiones para una política particular de QoS en un orden decreciente. En el lado del Publicador se especifica un conjunto de valores ofertados para esta política de QoS. El Middleware escogerá el valor que concuerde lo más posible a lo solicitado en el lado del Suscriptor, que está ofertado en el lado del Publicador; o puede rechazar el establecimiento de la comunicación entre los dos objetos *DCPSEntity*, sin la QoS solicitada y ofertada no pueden llegar a un acuerdo. En la Figura 1.12 se muestra el Modelo Suscriptor-Solicitado y el Publicador-Ofertado.

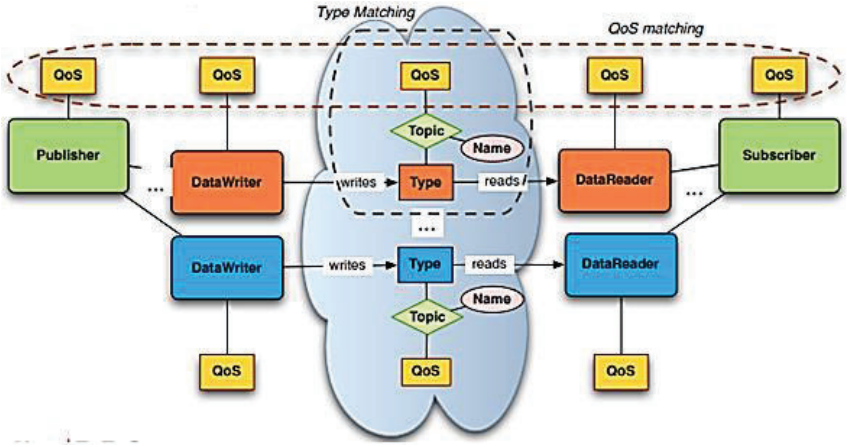


Figura 1.12. Modelo Suscriptor-Solicitado y Publicador-Ofertado [11].

1.6.1.4. Interoperabilidad

Existen múltiples aspectos de interoperabilidad en el Middleware. Estos incluyen al API, el *Wire Protocol* o Protocolo de conexión y la cobertura de QoS. Todos estos elementos tienen un rol importante en la interoperabilidad dentro de las tecnologías de Middleware. En el caso del DDS, hay estándares que especifican el API, el protocolo de conexión y la cobertura de QoS, que debe ser adherida a todos los participantes en las implementaciones DDS.

1.6.1.4.1. API y su interoperabilidad

El API es la interfaz entre el DDS y la aplicación. Este comprende los tipos de datos específicos y llamadas a funciones para que la aplicación interactúe con el Middleware, ya que el API está estandarizado, los clientes del estándar DDS pueden reemplazar implementaciones DDS con una recompilación simple, para no

cambiar el código. Un API estandarizado permite portabilidad del Middleware DDS y elimina el bloqueo de un propietario.

La

Figura 1.13 muestra la interoperabilidad del API.

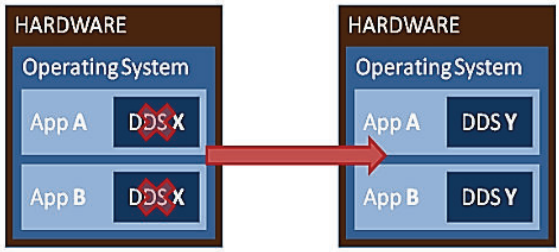


Figura 1.13. Interoperabilidad del API [14].

1.6.1.4.2. Protocolo de Conexión y su Interoperabilidad

El protocolo RTPS es responsable de la interoperabilidad del DDS sobre la conexión. La OMG también gestiona el estándar RTPS y adhiere a este múltiples proveedores del DDS. RTPS es usado como el protocolo de transporte de datos subyacente a la comunicación dentro del DDS. Este provee soporte para todas las tecnologías DDS, como el descubrimiento dinámico, las comunicaciones seguras, la independencia de plataforma, y las asociaciones de QoS. Este aspecto de interoperabilidad permite a un usuario del DDS fácilmente extender su sistema distribuido añadiendo diferentes implementaciones DDS. DDS utiliza estratégicamente comunicación de datos basada en multicast y unicast para las necesidades de las aplicaciones. DDS provee una implementación nativa de RTPS, es decir, no existen *gateways RTPS,* ni demonios, ni aplicaciones en otro plano, ya que se busca el mejor rendimiento posible. En la Figura 1.14 se puede ver la interoperabilidad del protocolo de conexión.

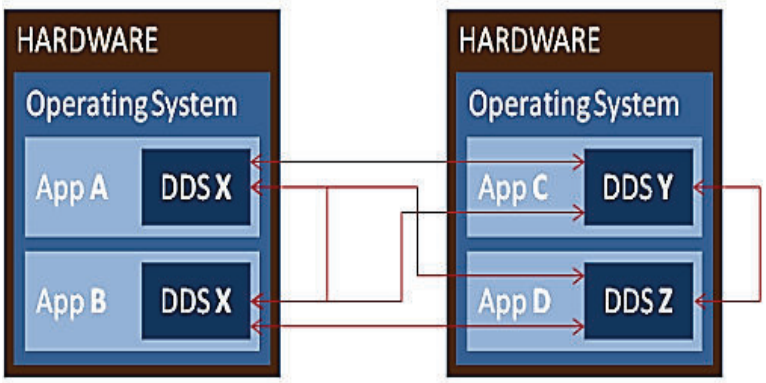


Figura 1.14. Interoperabilidad del Protocolo de Conexión [14].

1.6.1.4.3. Cobertura de la QoS y su interoperabilidad

Las políticas de QoS permiten a la aplicación medir el comportamiento específico de la comunicación. Esta medición incluye diferentes aspectos de la comunicación. Ejemplos de las políticas de QoS incluyen: la confiabilidad, es decir, ¿qué requerimientos de confiabilidad necesitan esos datos?; la durabilidad, es decir, ¿cuán grande son los datos guardados para publicaciones de datos posteriores?; historial y límites de recursos, es decir, ¿cuáles son los requerimientos de almacenamiento?; filtrado y presentación, es decir, ¿qué información debe ser presentada al suscriptor, y cómo?; y la propiedad es decir, ¿existen requisitos para la redundancia o para la desconexión?. Estas son solamente una pequeña parte de las 22 distintas políticas de QoS definidas por el estándar DDS. Estas políticas de QoS proveen un conjunto completo de opciones de configuración, permitiendo a las aplicaciones tomar fácilmente ventaja de estrategias de comunicación muy complejas y poderosas. Las características de QoS tienen un rol a través de todos los aspectos de interoperabilidad. Por supuesto, el API para la configuración de QoS y los protocolos de conexión para la interacción del QoS deben estar estandarizados para proveer implementaciones portables y conexiones interoperables dentro del DDS. Sin embargo, la cobertura de estas políticas de QoS depende del proveedor.

El estándar DDS además de especificar el API DDS, también categoriza las características de QoS dentro de perfiles que definen diferentes niveles de conformidad. El perfil mínimo tiene más de 22 políticas de QoS, y define un conjunto mínimo de políticas de QoS que deben estar cubiertas para una implementación DDS para ser compatible con el estándar, es decir, interoperable.

Todos estos aspectos de interoperabilidad puestos juntos permiten una gran flexibilidad a los clientes del Middleware.

1.6.2. FUNCIONALIDADES

El estándar DDS fue diseñado explícitamente para construir sistemas distribuidos en tiempo real. Para este fin, la especificación añade un conjunto de parámetros de calidad de servicio para configurar propiedades no funcionales. En este caso, el DDS provee una alta flexibilidad en la configuración de sistemas por medio de la asociación del conjunto de parámetros de calidad de servicio a cada entidad.

Además, el DDS permite la modificación de algunos parámetros en tiempo de ejecución, mientras se realiza una reconfiguración dinámica del sistema. Este conjunto de parámetros de calidad de servicio, permite varios aspectos de los datos, referidos a los recursos de red y recursos informáticos a ser configurados y pueden ser clasificados en las siguientes categorías, como se muestra en la Figura 1.15:

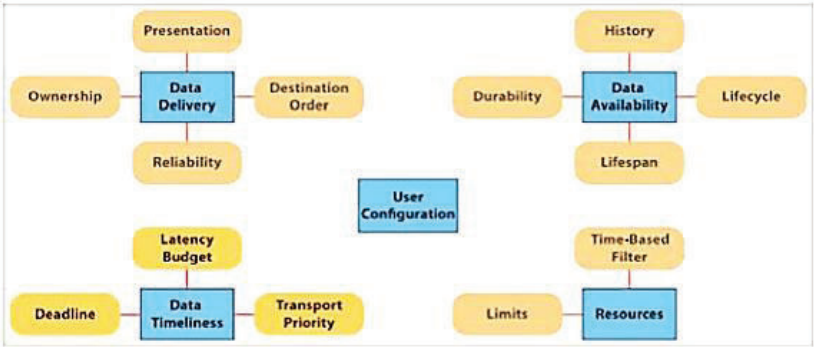


Figura 1.15. Parámetros de QoS definidos por DDS [11].

A continuación en la Tabla 1.12 , se explica cada política de calidad de servicio. Esta tabla fue generada por todo el grupo del proyecto de investigación.

Tabla 1.12. Políticas de Calidad de Servicio

QosPolicy	Valor	Significado	Cambiable	DDS	RTPS
USER_DATA		Los datos de usuario no conocidos por el middleware, pero distribuidos por medio de temas integrados. El valor por defecto es una secuencia vacía (de tamaño cero).	SI	X	
TOPIC_DATA		Los datos de usuario no conocidos por el middleware, pero distribuidos por medio de temas integrados. El valor por defecto es una secuencia vacía (de tamaño cero).	SI	X	
GROUP_DATA		Los datos de usuario no conocidos por el middleware, pero distribuidos por medio de temas integrados. El valor por defecto es una secuencia vacía (de tamaño cero).	SI	X	
DURABILITY	A “kind”: VOLATILE, TRANSIENT _LOCAL, TRANSIENT, or PERSISTENT	Esta política expresa si los datos deben 'sobrevivir' su tiempo de escritura.	NO		X
	VOLATILE	El servicio no es necesario para mantener las muestras de instancias de datos en nombre de cualquier DataReader que no es conocido por el DataWriter en el momento que la instancia escrita. En otras palabras, el Servicio sólo tratará de proporcionar los datos a los suscriptores existentes. Este es el tipo por defecto			

Tabla 1.12. Políticas de Calidad de Servicio

	TRANSIENT _ LOCAL, TRANSIENT	El servicio intentará mantener algunas muestras para que puedan ser entregados a cualquier potencial DataReader que participara más tarde. ¿Qué muestras en particular, se guardaron depende de otros QoS como la HISTORY y RESOURCE_LIMITS. Para TRANSIENT_LOCAL, el servicio sólo está obligado a mantener los datos en la memoria de la DataWriter que escribió los datos y los datos no se requiere para que sobreviva el DataWriter. Para TRANSIENT, el servicio sólo está obligado a mantener los datos en la memoria y no en el almacenamiento permanente; pero los datos no está ligado al ciclo de vida del DataWriter y que, en general, sobrevivirá. El soporte para el tipo TRANSIENT es opcional			
	PERSISTENT	[opcional] Datos son almacenados en memoria permanente, para que puedan sobrevivir a una sesión del sistema			
DURABILITY_ SERVICE	A duration "service_clean up _ delay" A HistoryQosPol icy Kind "history_kind" And three integers: history_depth, max_samples, max_instances , max_samples_ per_instance	Especifica la configuración del servicio de durabilidad. Esto es, el servicio que implementa el tipo DURABILITY de TRANSIENT y PERSISTENT	NO		X
	service_cleanu p_ delay	Control cuando el servicio es capaz de eliminar toda la información relativa a una instancia de datos. Por defecto, cero			
	history_kind, history_depth	Controla el HISTORY QoS del DataReader ficticio que almacena los datos dentro del servicio de durabilidad. La configuración predeterminada es history_kind = KEEP_LAST history_depth = 1			
	max_samples, max_instances , max_samples_ per_instance	Controla los RESOURCE_LIMITS QoS del DataReader implícito que almacena los datos en el servicio de durabilidad. Por defecto todos están LENGTH_UNLIMITED			
PRESENTATION	An “access_scope ”: INSTANCE, TOPIC, GROUP And two booleans: “coherent_acc ess” “ordered_acce ss”	Especifica cómo se presentan las muestras que representan los cambios a instancias de datos para la aplicación de suscripción. Esta política afecta a la capacidad de la aplicación para especificar y recibir cambios coherentes y para ver el orden relativo de los cambios. access_scope determina el mayor alcance que abarca las entidades para las que el orden y la coherencia de los cambios pueden ser preservados. Los dos booleanos controlan si el acceso coherente y acceso ordenado son compatibles dentro del ámbito access_scope.	NO	X	

Tabla 1.12. Políticas de Calidad de Servicio

	INSTANCE	Ámbito de aplicación se extiende por una única instancia. Indica que los cambios en una instancia no necesitan ser coherente ni ordenada con respecto a cambios en cualquier otra instancia. En otras palabras, el orden y cambios coherentes se aplican a cada instancia por separado. Este es el access_scope por defecto.			
	TOPIC	Ámbito de aplicación se extiende a todas las instancias dentro del mismo DataWriter (o DataReader), pero no en todos los casos en diferente DataWriter (o DataReader)			
	GROUP	[opcional] Ámbito abarca a todas las instancias que pertenecen a entidades DataWriter (o DataReader) del mismo editor (o suscriptor).			
	coherent_access	Especifica apoyar el acceso coherente. Es decir, la capacidad de agrupar un conjunto de cambios como una unidad en el extremo de publicación de tal manera que se reciben como una unidad en el extremo de suscribsión. La configuración predeterminada de coherent_access es FALSO.			
	ordered_access	Especifica el apoyo para el acceso ordenado a las muestras recibidas al final del suscriptor. Es decir, la capacidad del suscriptor para ver los cambios en el mismo orden en que se produjeron en el extremo de publicación. La configuración predeterminada de ordered_access es FALSO.			
DEADLINE	A duration “period”	DataReader espera una nueva muestra de la actualización del valor de cada instancia de al menos una vez cada período de plazo. DataWriter indica que la aplicación se compromete a escribir un nuevo valor (utilizando el DataWriter) para cada instancia gestionada por la DataWriter al menos una vez cada período de plazo. Es incoherente que un DataReader para tener un período DEADLINE menos de TIME_BASED_FILTER´s minimum_separation. El valor por defecto del período límite es el infinito.	SI		X
LATENCY_ BUDGET	A duration “duration”	Especifica el máximo retardo aceptable desde el momento que los datos se escriben hasta que se han insertado en la cache de la aplicación receptora y la aplicación que recibe es notificado del hecho. Esta política es una pista para el Servicio, no es algo que debe ser monitoreado o ejecutado. El servicio no está obligado a realizar un seguimiento o alertar al usuario de cualquier violación. El valor predeterminado de la duración es de cero que indica que el retardo debe ser minimizado.	SI		X
OWNERSHIP	A “kind” SHARED EXCLUSIVE	[opcional] Especifica si se permite que múltiples DataWriters escriban la misma instancia de los datos y si es así, cómo deben ser arbitradas estas modificaciones	NO	X	
	SHARED	Indica la titularidad compartida para cada instancia. A múltiples escritores se les permite actualizar la misma instancia y todas las actualizaciones están disponibles para los lectores. En otras palabras, no existe el concepto de un "propietario" de las instancias. Este es el comportamiento por defecto si no se especifica o apoyó la política OWNERSHIP QoS.			

Tabla 1.12. Políticas de Calidad de Servicio

	EXCLUSIVE	Indica que cada instancia sólo puede ser propiedad de un DataWriter, pero el dueño de una instancia puede cambiar dinámicamente. La selección del propietario se controla por el ajuste de la política de QoS OWNERSHIP_STRENGTH. El propietario siempre está dispuesto a ser el objeto DataWriter de mayor fuerza entre los que actualmente "activos" (según lo determinado por la QoS LIVELINESS).			
OWNERSHIP_ STRENGTH	An integer “value”	[opcional] Especifica el valor de la "fuerza" que se utiliza para arbitrar entre múltiples objetos DataWriter que intentan modificar la misma instancia de un objeto de datos (identificados por Topic + key). Esta política sólo se aplica si la política de QoS OWNERSHIP es de tipo EXCLUSIVE. El valor por defecto de la ownership_strength es cero.	SI	X	
LIVELINESS	A “kind”: AUTOMATIC, MANUAL_BY_ PARTICIPANT, MANUAL_BY_ TOPIC and a duration “lease_duration”	Determina el mecanismo y los parámetros utilizados por la aplicación para determinar si una entidad está "activa" (viva). El estado de "liveliness" de la Entidad se utiliza para mantener la propiedad de instancia en combinación con el ajuste de la política QoS OWNERSHIP. La aplicación también se informó a través de listener cuando la entidad ya no está viva. El DataReader solicita que el liveliness de los escritores es mantenida por los medios solicitados y pérdida de liveliness es detectada con un retraso que no exceda el lease_duration. El DataWriter compromete a señalización su liveliness utilizando los medios que, a intervalos que no excedan el lease_duration. Los listeners son utilizados para notificar el DataReader de pérdida de liveliness y DataWriter de violaciónes en el contrato de liveliness. El tipo predeterminado es AUTOMATIC y el valor por defecto de la lease_duration es infinito.	NO		X
	AUTOMATIC	La infraestructura indicará automáticamente la vivacidad de los DataWriters al menos con la frecuencia que requiera el lease_duration			
	MANUAL modes	La aplicación de usuario asume la responsabilidad de señalar liveliness al Servicio. Liveliness debe afirmarse por lo menos una vez cada lease_duration de lo contrario el servicio asumirá la Entidad correspondiente dejando de estar "activa / con vida."			
	MANUAL_BY_PARTICIPANT	El Servicio asumirá que mientras al menos una entidad dentro del DomainParticipant ha afirmado su liveliness las otras entidades de ese mismo DomainParticipant también están vivas.			
	MANUAL_BY_TOPIC	El Servicio sólo asumirá el liveliness de la DataWriter si la aplicación ha afirmado el liveliness de que la propia DataWriter.			
TIME_BASED_ FILTER	A duration "minimum_separation"	Filtro que permite a un DataReader especificar que sólo está interesado (potencialmente) en un subconjunto de los valores de los datos. El filtro	SI		X

Tabla 1.12. Políticas de Calidad de Servicio

		establece que el DataReader no quiere recibir más de un valor cada minimum_separation, independientemente de qué tan rápido se producen los cambios. Es incoherente para un DataReader tener un minimum_separation más largo que su período DEADLINE. Por defecto minimum_separation = 0 indicando que el DataReader esta potencialmente interesados en todos los valores.			
PARTITION	A list of strings “name”	Conjunto de strings que introduce una partición lógica entre los temas visibles por el Publisher y el Subscriber. Un DataWriter dentro de un Publisher sólo se comunica con un DataReader en un Subscriber si (además de coincidir el tema y tener QoS compatibles) el publicador y el suscriptor tiene una partición común string nombre. La cadena vacía ("") se considera una partición válida que se corresponde con otros nombres de partición utilizando las mismas reglas de string correspondientes y de expresiones regulares que coinciden usadas para ningún otro nombre de la partición. El valor por defecto para la PARTITION QoS es una secuencia de longitud cero. La secuencia de longitud cero se trata como un valor especial equivalente a una secuencia que contiene un solo elemento que consiste en la cadena vacía.	SI	X	
RELIABILITY	A “kind”: RELIABLE, BEST_EFFORT and a duration “max_blocking_time”	Indica el nivel de fiabilidad que ofrecido/solicitado por el Servicio.	NO		X
	RELIABLE	Especifica el servicio que intentará entregar todas las muestras de su historial. Muestras perdidas pueden ser reintendas. En el estado de steady (sin modificaciones comunicadas a través de la DataWriter) las garantías de middleware que todas las muestras del historial del DataWriter eventualmente se entregan a todos los objetos DataReader. Fuera de estado steady las políticas de HISTORY y RESOURCE_LIMITS determinarán cómo las muestras se convierten en parte del historial y si las muestras pueden ser desechados de la misma. Este es el valor por defecto para DataWriters.			
	BEST_EFFORT	Indica que es aceptable para no volver a intentar la propagación de las muestras. Presumiblemente nuevos valores para las muestras se generan bastante a menudo que no es necesario volver a enviar o reconocer las muestras. Este es el valor por defecto para DataReaders y Topics.			
	max_blocking_time	El valor de la max_blocking_time indica el tiempo máximo que la operación DataWriter::write se permite bloquear si el DataWriter no tiene espacio para almacenar el valor escrito. El valor por defecto del max_blocking_time = 100 ms.			
TRANSPORT_PRIORITY	An integer “value”	Esta política es una pista para la infraestructura en cuanto a la forma de establecer la prioridad del transporte subyacente utilizado para enviar los	SI		X

Tabla 1.12. Políticas de Calidad de Servicio

		datos. El valor por defecto del transport_priority es cero.			
LIFESPAN	A duration “duration”	Especifica la duración máxima de validez de los datos escritos por el DataWriter El valor predeterminado del lifespan duration es infinita.	SI		X
DESTINATION_ ORDER	A “kind”: BY_RECEPTION_TIMESTAMP, BY_SOURCE_TIMESTAMP	Controla los criterios utilizados para determinar el orden lógico de los cambios realizados por entidades de Publisher a la misma instancia de datos (es decir, correspondiente Tema y clave). El tipo predeterminado es BY_RECEPTION_TIMESTAMP.	NO		X
	BY_RECEPTION_ TIMESTAMP	Indica que los datos se ordena en base a la hora de recepción en cada suscriptor. Dado que cada suscriptor puede recibir los datos en diferentes momentos no hay garantía que los cambios se verán en el mismo orden. En consecuencia, es posible para cada suscriptor terminar con un valor final diferente para los datos.			
	BY_SOURCE_TIMESTAMP	Indica que los datos se ordena sobre la base de una marca de tiempo colocada en la fuente (por el Servicio o por la aplicación). En cualquier caso, esto garantiza un valor final consistente para los datos de todos los suscriptores.			
HISTORY	A “kind”: KEEP_LAST, KEEP_ALL And an optional integer “depth”	Especifica el comportamiento del servicio en el caso donde el valor de las muestras cambie (una o más veces) antes de que pueda ser comunicada con éxito a uno o más suscriptores existentes. Esta politica de QoS controla si el servicio debe entregar sólo el valor más reciente, intenta el liberar a todos los valores intermedios, o hacer algo en el medio. Por el lado del publicador esta política controla las muestras que deben mantenerse por el DataWriter en nombre de entidades DataReader existentes. El comportamiento con respecto a las entidades DataReader descubiertos después de un ejemplo está escrito es controlado por la política DURABILITY QoS. En el lado suscripción controla las muestras que deben mantenerse hasta que la aplicación las "toma" desde el Servicio.	NO		X
	KEEP_LAST and optional integer “depth”	Por el lado del publicador, el Servicio sólo tratará de mantener las más recientes muestras de "fondo" de cada instancia de datos (identificados por su clave), gestionadas por el DataWriter. Por el lado de suscripción, el DataReader sólo tratará de mantener las más recientes muestras de "fondo" recibidas para cada instancia (identificada por su clave) hasta que la aplicación las "toma" a través de la operación take del DataReader. KEEP_LAST es el tipo por defecto. El valor predeterminado de fondo es 1. Si se especifica un valor distinto de 1, debe ser coherente con la configuración de la política RESOURCE_LIMITS QoS.			
	KEEP_ALL	Por el lado del publicador, el servicio intentará mantener todas las muestras (que representa cada valor escrito) de cada instancia de datos (identificados por su clave), gestionada por el DataWriter hasta que puedan ser entregados a todos los suscriptores. Por el lado de suscripción, el servicio intentará mantener todas las muestras			

Tabla 1.12. Políticas de Calidad de Servicio

		de cada instancia de datos (identificados por su clave), gestionados por el DataReader. Estas muestras se conservarán hasta que la aplicación las "toma" desde el servicio a través de la operación de take. El ajuste de la profundidad no tiene ningún efecto. Su valor implícito es LENGTH_UNLIMITED2.			
RESOURCE_ LIMITS	Three integers: max_samples, max_instances , max_samples_ per_instance	Especifica los recursos que el Servicio puede consumir a fin de satisfacer la QoS solicitada.	NO	X	
	max_samples	Especifica el número máximo de datos-muestras que el DataWriter (o DataReader) puede gestionar a través de todos los casos asociados a estos. Representa el máximo de muestras que el middleware puede almacenar para cualquier DataWriter (o DataReader). Es incoherente que este valor sea menor que max_samples_per_instance. Por defecto, LENGTH_UNLIMITED			
	max_instances	Representa el número máximo de instancias DataWriter (o DataReader) que se pueden gestionar. Por defecto, LENGTH_UNLIMITED3.			
	max_samples_ per_instance	Representa el número máximo de muestras que cualquier instancia de DataWriter (o DataReader) pueden gestionar. Es incoherente que este valor sea mayor que max_samples. Por defecto, LENGTH_UNLIMITED4.			
ENTITY_ FACTORY	A boolean: “autoenable_c reated_entities ”	Controla el comportamiento de la entidad cuando actúa como una fábrica para otras entidades. En otras palabras, configura los efectos secundarios de las operaciones create_* y delete_*.	SI	X	
	autoenable_cr eated_entities	Especifica si la entidad que actúa como una fábrica habilita automáticamente las instancias que crea. Si autoenable_created_entities == TRUE la fábrica habilitará automáticamente cada Entidad creada de lo contrario no lo hará. De forma predeterminada, TRUE.			
WRITER_ DATA_ LIFECYCLE	A boolean: “autodispose_ unregistered_ instances”	Especifica el comportamiento del DataWriter con respecto al ciclo de vida de las instancias de datos que gestiona.	SI	X	
	autodispose_u nregistered_ instances	Controla si un DataWriter dispondrá automáticamente instancias cada vez que no están registradas. La configuración autodispose_unregistered_instances = TRUE indica que las instancias no registradas también serán consideradas dispuestos. De forma predeterminada, TRUE.			
READER_ DATA _ LIFECYCLE	Two durations “autopurge_ nowriter_sam ples _delay” and “autopurge_ disposed_ samples_delay ”	Especifica el comportamiento del DataReader en relación con el ciclo de vida de la instancia de datos que gestiona.	SI	X	

Tabla 1.12. Políticas de Calidad de Servicio

	autopurge_ nowriter_sam ples _delay	Indica el tiempo que el DataReader debe retener información sobre las instancias que tienen el instance_state NOT_ALIVE_NO_WRITERS. De forma predeterminada, infinito.			
	autopurge_ disposed_ samples_delay	Indica el tiempo que el DataReader debe retener información sobre las instancias que tienen la instance_state NOT_ALIVE_DISPOSED. De forma predeterminada, infinito			

Finalmente, esta especificación sigue el modelo Suscriptor-Solicitado, y el Publicador-Ofertado para establecer los parámetros de QoS. Mediante el uso de este modelo, ambos el Publicador y el Suscriptor deben especificar parámetros compatibles de QoS para establecer la comunicación. De otra manera, el Middleware debe indicar a la aplicación que la comunicación no es posible.

1.6.2.1. Gestión de Recursos del Procesador

El estándar DDS no aborda explícitamente la planificación de hilos en los procesadores, ya que esto es un aspecto de implementación definida. Sin embargo, un subconjunto de parámetros de QoS, definidos por el estándar están enfocados al control del comportamiento temporal y el mejoramiento de la previsibilidad de la aplicación. Los tres parámetros de la puntualidad de datos, que están resaltados en la Figura 1.15, son importantes en la gestión de recursos de sistemas de tiempo real. En particular, el estándar ha definido los siguientes parámetros para la gestión de recursos de procesador:

- *Deadline,* este parámetro indica la cantidad máxima de tiempo disponible para enviar/ recibir muestras de datos pertenecientes a un tópico particular. Sin embargo, este no define ningún mecanismo asociado para asegurar estos requerimientos temporales; por lo tanto, este parámetro de QoS sólo representa un servicio de notificación en el cual el Middleware informa a la aplicación que el tiempo límite se ha perdido.
- *Latency_Budget,* este parámetro es definido como el retardo máximo aceptable en la entrega de mensajes. Sin embargo, el estándar hace énfasis en que este parámetro no puede ser llevado a cabo o controlado por el Middleware; por lo tanto, este puede ser usado para optimizar el comportamiento interno del Middleware.

Estos dos parámetros de QoS, incluso si ambos comparten objetivos similares, se aplican a diferentes niveles, como se ilustra en la Figura 1.16. Esta

figura muestra cómo el parámetro *deadline* es monitorizado dentro de la capa DDS, mientras que el parámetro *Latency_Budget* se aplica dentro de la capa DDSI.

Figura 1.16. Control del tiempo en DDS [11].

El DDS define diferentes mecanismos para permitir la comunicación entre entidades. En el lado del Publicador, el mecanismo de comunicación es sencillo: cuando los nuevos datos están disponibles, el DW realiza una llamada de escritura simple, como por ejemplo, escribir o eliminar, para publicar datos dentro del dominio del DDS. Entonces, la muestra de datos es transmitida usando modos de comunicación de los tipos asincrónicos, uno a uno o uno a varios. Sin embargo, el DDS también da soporte a hilos de llamadas bloqueadas hasta que la muestra de datos haya sido entregada y confirmada por los DR asociados.

En el lado del Suscriptor, la recepción de los datos puede ser realizada con *polling*[9] [15], modo sincrónico, y asincrónico. Estos modelos no sólo son válidos para la recepción de datos sino también para la notificación de cualquier cambio en el estado de la comunicación, por ejemplo, para el no cumplimiento de las peticiones de calidad de servicio. En particular, la aplicación podría ser notificada a través de los siguientes modelos:

- *Polling,* como los hilos de una aplicación pueden invocar operaciones no bloqueantes para obtener datos o cambios en el estado de la comunicación.
- *Listeners,* adjunta una función de devolución de llamada para las modificaciones de acceso asincrónico en el estado de la comunicación mientras que la aplicación se sigue ejecutando, es decir, que los hilos del Middleware son responsables de la gestión de cualquier cambio en el estado de la comunicación.

[9] Polling, hace referencia a una operación de consulta constante

- *Conditions y Wait-Sets*, los cuales permiten que los hilos de la aplicación sean bloqueados hasta conocer una o varias condiciones. Ambas representan el mecanismo de sincronización para gestionar cualquier cambio en el estado de la comunicación.

1.6.2.2. Gestión de Recursos de Red

En materia de redes, esta especificación define un conjunto de características enfocadas a garantizar el determinismo en las comunicaciones, tal como el uso de parámetros de planificación en redes y la definición del formato para el intercambio de mensajes.

El paso de parámetros de planificación para las redes de comunicación se lleva a cabo a través de otro parámetro de QoS incluido en la categoría de *data timeless,* mostrado en la Figura 1.15:

- *Transport-Priority,* a diferencia de *Latency-Budget*, que intenta optimizar el comportamiento interno del Middleware, este parámetro prioriza el acceso a la red de comunicación, como se muestra en la Figura 1.16. Además, mientras que las comunicaciones son unidireccionales, este sólo está asociado con entidades DW.

Por otra parte, la especificación DDSI define el conjunto de normas y características requeridas para habilitar la comunicación entre entidades DDS. Aunque esta especificación no está particularmente orientada al uso de redes en tiempo real, esta no opone su uso y sólo muestra un conjunto de requisitos para las redes subyacentes. El punto más importante tratado por la especificación se encuentra descrito en el protocolo RTPS, el cual es responsable específicamente de cómo se difunden los datos entre los nodos. Esto requiere la definición de los protocolos de intercambio de mensajes y formatos de mensajes. En particular, la estructura de un mensaje RTPS consiste de una cabecera de tamaño fijo seguida por un número variable de submensajes. Al procesar cada submensaje independientemente, el sistema puede descartar mensajes desconocidos o erróneos lo cual facilita futuras extensiones del protocolo.

Otra característica clave de DDS es la sobrecarga introducida por las operaciones internas del Middleware. En este caso, el estándar define una serie de operaciones a ser llevadas a cabo por las implementaciones que puedan consumir recursos tanto del procesador como de la red. En particular, DDS proporciona un

servicio para la gestión de entidades remotas llamadas *Discovery* o Descubrimiento. Este servicio describe como se obtiene información sobre la presencia y características de cualquier otra entidad inmersa en el sistema distribuido. Aunque el estándar describe un protocolo específico para el descubrimiento con el propósito de la interoperabilidad, este permite que otros protocolos de descubrimiento puedan ser aplicados. Bajo el protocolo de descubrimiento requerido, las implementaciones deben crear un conjunto de entidades DDS por defecto. Estas entidades presentes son responsables del establecimiento transparente de la comunicación con el usuario y del descubrimiento de la presencia o ausencia de entidades remotas, por ejemplo, un sistema de *plug-and-play*[10]. Este tipo de tráfico en la red, el cual es interno en el Middleware, es llamado metatráfico y puede ser considerado en los análisis de tiempo.

1.6.3. LECTURA Y ESCRITURA DE DATOS

1.6.3.1. Escritura de Datos

Escribir datos dentro del DDS será un proceso simple ya que sólo se deberá llamar al método *write* del *DataWriter*. Este permitirá a una aplicación establecer el valor de los datos para ser publicados bajo un determinado Topic.

El ciclo de vida del *topic-instances*, podrá ser manejado implícitamente a través de la semántica implicada por el *DataWriter*, o podrá consultarse explícitamente por la API *DataWriter*. La transición del ciclo de vida de *topic-instances* puede tener implicaciones en el uso de recursos locales y remotos.

1.6.3.2. Ciclo de Vida de los Topic-Instances

Los estados disponibles en los *topic-instances serán*:

- *ALIVE*, si por lo menos hay un *DataWriter* escribiendo.
- *NOT_ALIVE_NO_WRITERS*, cuando no haya más *DataWriters*, escribiendo.
- *NOT_ALIVE_DISPOSED*, si el *DataWriter* ha sido eliminado ya sea de manera implícita debido a un defecto en la calidad de servicio, o de manera explícita mediante una llamada específica en el API *DataWriter*. Este estado indicará que la instancia ya no es relevante para el sistema y que no debe ser escrita más por cualquier escritor. Como resultado, los recursos asignados por el sistema para almacenar la instancia podrán liberarse.

[10] Plug-and-Play, se refiere a un conjunto de protocolos de comunicación que permiten descubrir de manera transparente la presencia de otros dispositivos en la red.

1.6.3.2.1. Administración del ciclo de vida automática

El ciclo de vida automática de las instancias será explicado por medio de un ejemplo. Al observar el código de una supuesta aplicación, el cual se muestra en la Tabla 1.13, se puede notar que solamente es para escribir datos, y existen tres operaciones *write*, donde se crean tres nuevas instancias de *topic* en el sistema, y están asociados a los id=1, 2, 3.

Tabla 1.13. Administración del Ciclo de Vida Automática

Código de ejemplo del uso del método *write()*

```
int main(int, char**) {
   dds::Topic<TempSensorType>
tsTopic("TempSensorTopic");
   dds::DataWriter<TempSensorType>
dw(tsTopic);
   TempSensorType ts;
   //[NOTE #1]: Instances implicitly
registered as part
   // of the write.
   // {id, temp hum scale};
   ts = {1, 25.0F, 65.0F, CELSIUS};
   dw.write(ts);
   ts = {2, 26.0F, 70.0F, CELSIUS};
   dw.write(ts);
   ts = {3, 27.0F, 75.0F, CELSIUS};
   dw.write(ts);
   sleep(10);
   //[NOTE #2]: Instances automatically
unregistered and
   // disposed as result of the destruction
of the dw object
   return 0;
}
```

Mientras estas instancias se encuentren en el estado *ALIVE*, estas serán registradas automáticamente, es decir, que están asociadas con el *Writer*.

Por tanto el comportamiento del DDS será eliminar las instancias del *topic* una vez que se destruya el objeto *DataWrite*, es decir, llevando a las instancias a estado *NOT_ALIVE_DISPOSED*.

1.6.3.2.2. Administración de Ciclo de Vida Explícita

El ciclo de vida de *topic-instances* podrá manejarse explícitamente por el API definido en el *DataWriter*. En este caso el programador de la aplicación tendrá el control sobre cuándo las instancias son registradas, se dejan de registrar o se eliminan. En esencia, el acto de registrar explícitamente una instancia permitirá al Middleware reservar recursos, así como optimizar la búsqueda de instancias. Finalmente, eliminar un *topic-instances* dará una manera de comunicar al DDS que la instancia no es más relevante para el sistema distribuido, por lo tanto, los recursos asignados a las instancias específicas deberán ser liberados tanto de forma local como remota.

1.6.3.2.3. Topic sin claves

Los *topic* sin claves se convierten en únicos. En los *topic* sin clave el estado de transición será vinculado al ciclo de vida del *DataWriter*.

1.6.3.3. Lectura de Datos

El DDS tendrá dos mecanismos diferentes para acceder a los datos, cada uno de ellos permitirá controlar el acceso a los datos y la disponibilidad de los mismos. El acceso a los datos se realizará a través de la clase *DataReader* exponiendo dos semánticas para acceder a los datos: *read* y *take*.

1.6.3.3.1. Read y Take

La semántica del *read* implementado por el método *DataReader: read*, dará acceso a los datos recibidos por el *DataReader* sin sacarlo de su caché local, la cual corresponde a una memoria de almacenamiento local. Por lo cual estos datos serán nuevamente legibles mediante una llamada apropiada al *read*. Así mismo, el DDS proporcionará una semántica de *take*, implementado por los métodos *DataReader::take* que permitirá acceder a los datos recibidos por el *DataReader* para removerlo de su caché local. Esto significa que una vez que los datos están tomados, no estarán disponibles para subsecuentes operaciones *read* o *take*.

El *read* y el *take* DDS se encontrarán siempre en modo no bloqueado. Si los datos no están disponibles para leerse, la llamada retornará inmediatamente. Además, si hay menos datos que requieren llamadas reunirán lo disponible y retornará de inmediato. En este modo se asegurará que estos puedan utilizarse con seguridad por aplicaciones que sondean datos.

1.6.3.4. Datos y Metadatos

El ciclo de vida de *topic-instances* junto con otra información que describe las propiedades de las muestras de los datos recibidos estará a disposición del *DataReader* y podrán ser utilizadas para seleccionar los datos accedidos a través del *read* o ya sea del *take*.

Especialmente, para cada muestra de datos recibidos un *DataWriter* será asociado a una estructura, llamada *SampleInfo* describiendo la propiedad de esa muestra. Estas propiedades incluyen información como:

- **Estado de la muestra.** El estado de la muestra podrá ser *READ* o *NOT_READ* dependiendo si la muestra ya ha sido leída o no.

- **Estado de la instancia.** Este indicará el estado de la instancia como *ALIVE*, *NOT_ALIVE_NO_WRITERS*, o *NOT_ALIVE_DISPOSED*.
- **Estado de la vista.** Este podrá ser *NEW* o *NOT_NEW* dependiendo de si es la primera muestra recibida por el *topic-instance*.

El *SampleInfo* también contiene un conjunto de contadores que permitirán reconstruir el número de veces que el *topic-instance* haya realizado cierta transición de estado, convirtiéndose en *alive* después del *disposed*.

Finalmente, el *SampleInfo* contendrá un *timestamp* para los datos y una bandera que dice si la muestra es asociada o no. Esta bandera es importante ya que el DDS podrá generar información válida de las muestras con datos no válidos para informar acerca de las transiciones de estado como una instancia a ser desechada.

1.6.3.5. Notificaciones

Una forma de coordinar con DDS será tener un sondeo de uso de datos mediante la realización de un *read* o un *take* de vez en cuando. El sondeo podría ser el mejor enfoque para algunas clases de aplicaciones, el ejemplo más común es en las aplicaciones de control. En general, las aplicaciones podrán ser notificadas de la disponibilidad de datos o tal vez esperar su disponibilidad. El DDS apoyará la coordinación por medio de *waitsets* y los *listeners*.

1.6.3.5.1. Waitsets

El DDS proporcionará un mecanismo genérico para la espera de eventos. Uno de los tipos soportados de eventos son las condiciones de lectura, las cuales podrán ser usadas para esperar la disponibilidad de los datos de uno o más *DataReaders*.

1.6.3.5.2. Listeners

Otra manera de encontrar datos a ser leídos, será aprovechar al máximo los eventos planteados por el DDS y asincrónicamente notificar a los *handler* (controladores) registrados. Por lo tanto, si se quiere un *handler* para ser notificado de la disponibilidad de los datos, se deberá conectar el *handler* apropiado con el evento *on_data_available* planteado por el *DataReader*.

Los mecanismos de control de eventos permitirán enlazar cualquier cosa que se quiera a un evento DDS, lo que significa que se puede enlazar una función, un método de clase, etc.. Por ejemplo, cuando se trata con el evento

on_data_available se tendrá que registrar una entidad exigible que acepte un único parámetro de tipo *DataReader*.

Finalmente, cabe mencionar que el *handler* se ejecutará en un *thread* de Middleware. Como resultado, cuando se utilizan *listeners* se debería minimizar el tiempo del *listener* empleado en el mismo.

CAPÍTULO 2.
ANÁLISIS DE REQUISITOS PARA LA IMPLEMENTACIÓN DE UN MÓDULO QUE SOPORTE EL PROTOCOLO RTPS

2.1. INTRODUCCIÓN

En el presente capítulo se definen los requisitos necesarios para integrar el protocolo RTPS con el Middleware DDS que se describió en el capítulo anterior. Primeramente, se realizan capturas de paquetes con la herramienta Wireshark de los diferentes mensajes RTPS y mensajes de descubrimiento RTPS, para lo cual se utiliza el software proporcionado por OpenDDS y RTI Connext DDS Professional, los cuales están descritos en los anexos. Finalmente, se obtiene un análisis detallado de los paquetes RTPS y se definen los requisitos necesarios para su implementación.

2.2. ANÁLISIS DE PAQUETES DE LOS DIFERENTES MENSAJES RTPS

2.2.1. ESTRUCTURA DE LOS MENSAJES RTPS

2.2.1.1. Estructura general

En la Figura 2.1 se muestra la estructura general del mensaje RTPS incluyendo el tamaño en bytes de cada campo.

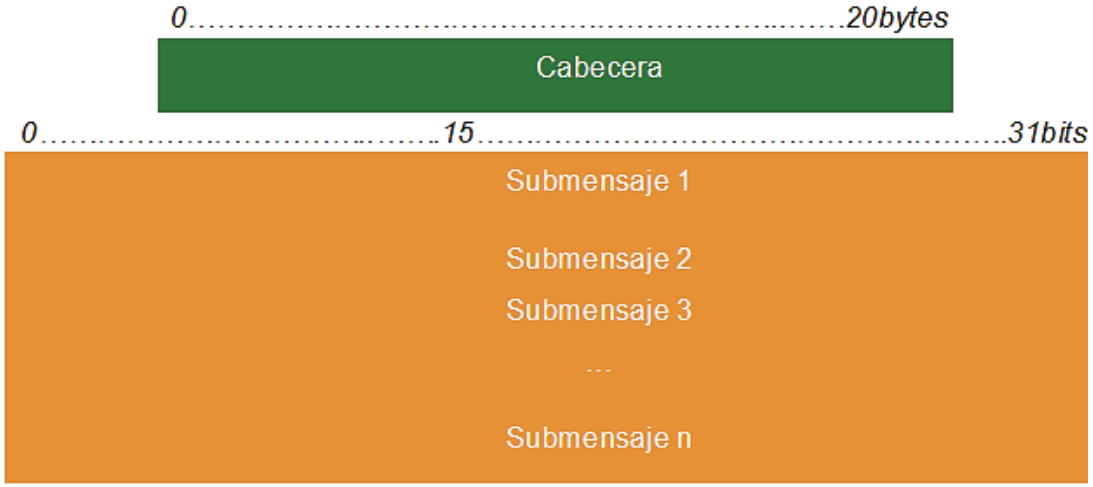


Figura 2.1. Estructura general mensaje RTPS [13]

Los mensajes RTPS explícitamente no envían el tamaño, pero utilizan el transporte con el cual se envían los mensajes, para el caso del tamaño, se envía en la carga UDP/IP.

2.2.1.2. Cabecera

La cabecera del mensaje RTPS está formada por los campos mostrados en la Figura 2.3, los cuales se encargan de proporcionar información de la versión del protocolo y de identificar al mensaje.

Figura 2.2. Cabecera del Mensaje RTPS [13]

2.2.2. ESTRUCTURA DE LOS SUBMENSAJES RTPS

En la Figura 2.3 se muestra la estructura del submensaje, la cual está compuesta por una cabecera submensaje y el contenido de submensaje. El submensaje tal como se muestra tiene un tamaño en múltiplos de 4 bytes.

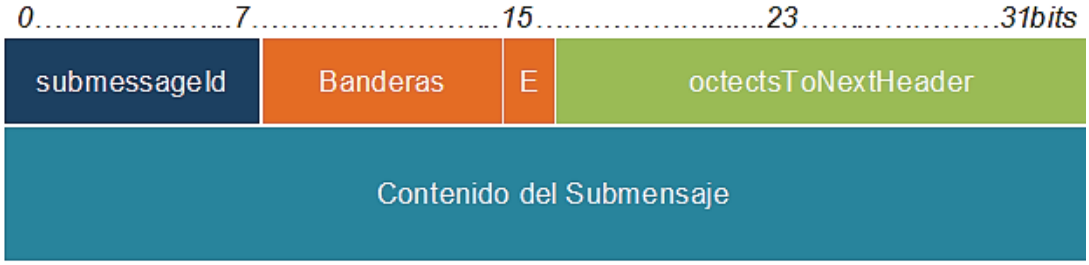


Figura 2.3. Estructura de los submensajes RTPS [13]

2.2.2.1. Lista de submensajes

A continuación se muestra el listado de submensajes posibles dentro de un mensajes RTPS.

- AckNack
- Data
- DataFrag
- GAP
- Heartbeat
- HeartbeatFrag
- InfoDestination
- InfoReply
- InfoSource
- InfoTimeStamp
- NackFrag
- InfoReplyIp4

- Pad

2.2.3. ACKNACK SUBMESSAGE

En la Figura 2.4 se muestra la estructura del submensaje *AckNack*.

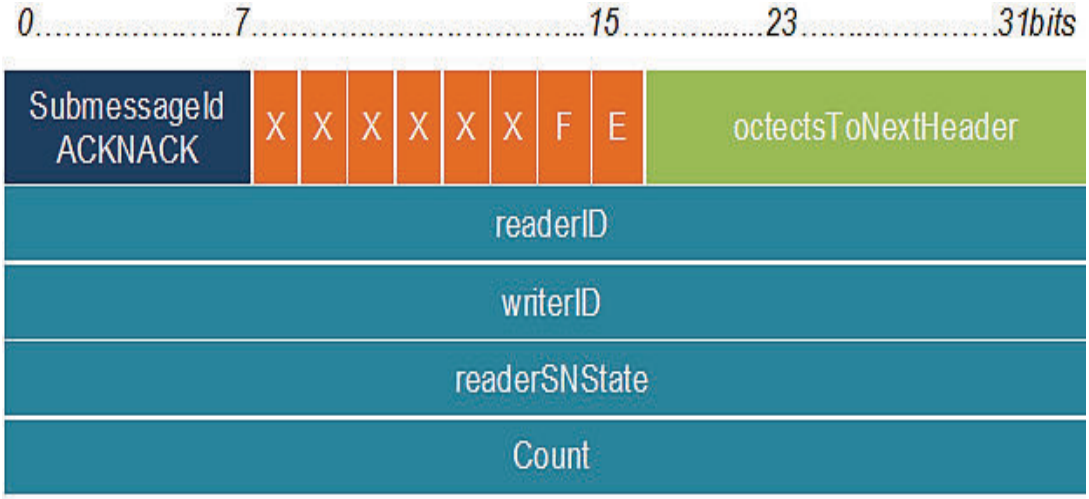


Figura 2.4. Estructura del submensaje AckNack [13]

Este submensaje se utiliza para comunicar el estado de un *lector* a un *escritor*. El submensaje permite al lector dar a conocer los números de secuencia que ha recibido y los que le faltan todavía al escritor. Este submensaje puede utilizarse para hacer acuses de recibo tanto positivos como negativos.

2.2.3.1. Banderas en el encabezado de submensaje

En el **FinalFlag**, cuando este campo tiene el valor de 1 significa que el lector no requiere una respuesta del escritor, sin embargo, si este se establece en 0 significa que el escritor debe responder al mensaje de *AckNack*.

En el **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de bits.

2.2.3.2. Otros elementos en el encabezado del submensaje

El ***Reader*ID**, identifica la entidad lectora que realiza el acuse recibo de cierto número de secuencia o las solicitudes para recibir ciertos números de secuencia.

El ***Writer*ID**, identifica la entidad escritora a la cual tiene como objetivo el submensaje AckNack. Es la entidad del escritor que pide reenviar algunos números de secuencia o está siendo informado de la recepción de ciertos números de secuencia.

El ***Reader*SNState**, comunica el estado del lector al escritor, enviando números de secuencia que confirman la llegada de los datos en el lector.

El **Count**, se incrementa cada vez que se envía un mensaje *AckNack*. Proporciona los medios para que un escritor detecte los mensajes duplicados de *AckNack* que pueden derivarse de los diferentes canales de comunicación.

2.2.3.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el *submessageLength* en el encabezado de submensaje es demasiado pequeño.
- Cuando el *ReaderSNState* no es válido.

2.2.3.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor.

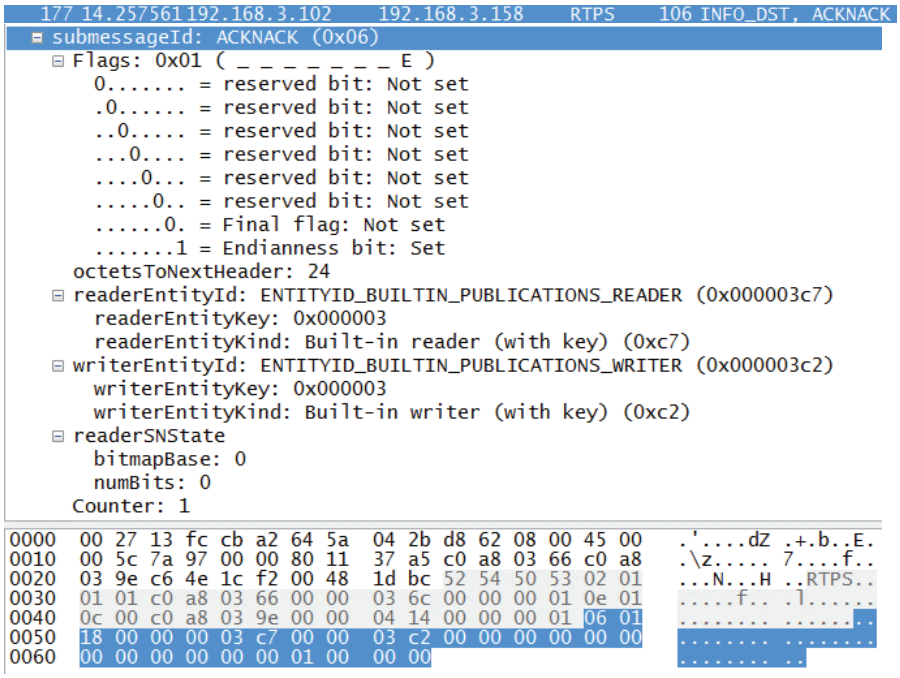


Figura 2.5. Uso del submensaje ACKNACK.

2.2.3.5. Interpretación lógica

El lector envía el submensaje AckNack al escritor para comunicar su estado con respecto a los números de secuencia utilizados por el escritor.

El escritor se identifica únicamente por su GUID, el cual es un atributo de todas las Entidades RTPS, e identifica de manera única a las entidades RTPS en el dominio DDS. El GUID del escritor se obtiene utilizando el estado del receptor.

El lector se identifica únicamente por su GUID. El GUID del lector se obtiene utilizando el estado del receptor.

El mensaje tiene dos propósitos simultáneamente:

- Reconocer en el submensaje todos los números de secuencia dentro del *SequenceNumberSet* el cual se encuentra dentro del *ReaderSNState.*
- Reconocer los acuses de recibo negativos que aparecen explícitamente en el *ReaderSNState*.

2.2.3.5.1. Ejemplo

En la Figura 2.5 se muestra la captura del submensaje *AckNack*, se puede observar claramente los campos previamente descritos, con valores en las banderas y en los identificadores.

Una explicación del uso del *AckNack* ha sido descrita en el capítulo 3 en el ejemplo 2 de la sección 3.3.5

2.2.4. DATA SUBMESSAGE

En la Figura 2.6 se muestra la estructura del submensaje *Data*.

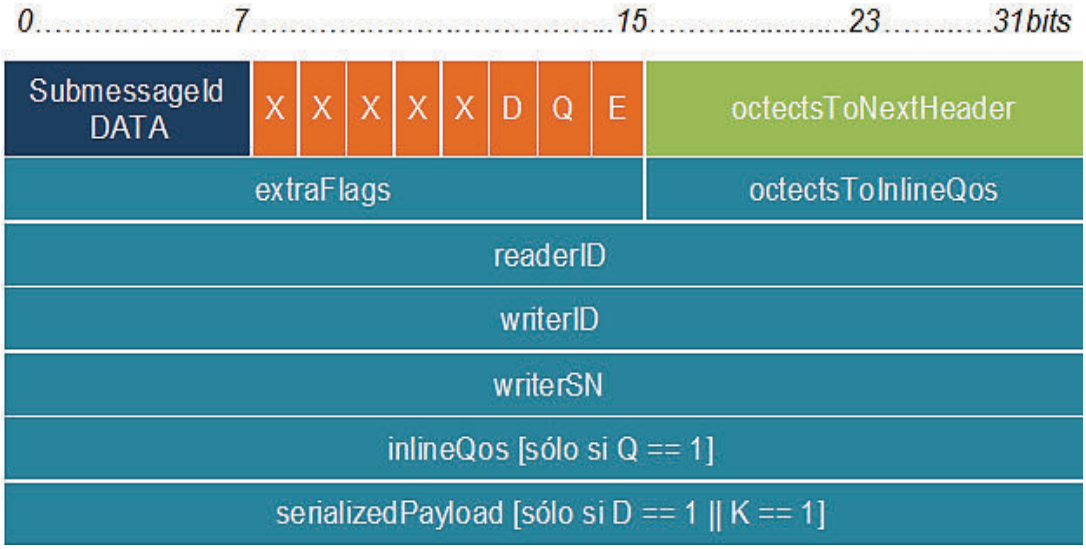


Figura 2.6. Estructura del submensaje Data [13]

El submensaje notifica al lector RTPS los cambios en un objeto de datos pertenecientes al escritor RTPS. Los posibles cambios incluyen el valor y el ciclo de vida del objeto de datos.

2.2.4.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de bits.

La **InlineQosFlag**, indica al lector la presencia de un *ParameterList* que contiene los parámetros de calidad de servicio que debe utilizarse para interpretar

el mensaje. Cuando Q = 1 significa que el submensaje data contiene información dentro del inlineQos.

El **DataFlag**, indica al lector que el elemento de submensaje *dataPayload* contiene el valor serializado del objeto de datos. Se representa con el literal 'D'.

La **KeyFlag**, indica al lector que el elemento de submensaje *dataPayload* contiene el valor de la clave del objeto de datos serializado. Se representa con el literal 'K '.

El *DataFlag* se interpreta en combinación con el *KeyFlag* de la siguiente manera:

- D = 0 y K = 0 significa que no hay ningún dato dentro del *payload*.
- D = 1 y K = 0 significan que el *payload* contiene datos serializados.
- D = 0 y K = 1 significan que el *payload* contiene la clave serializada.
- D = 1 y K = 1 es una combinación no válida en esta versión del Protocolo.

2.2.4.2. Otros elementos en el encabezado del submensaje

El ***Reader*ID**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El ***Writer*ID** , identifica la entidad RTPS escritora que realizó el cambio.

El ***Writer*SN**, identifica el cambio y el orden relativo de todos los cambios realizados por el **escritor** RTPS identificados por el *Writer*Guid. Cada cambio obtiene un número de secuencia consecutiva. Cada escritor RTPS mantiene su propio número de secuencia.

El **inlineQos**, presenta información de QoS solamente si el *InlineQosFlag* está situado en la cabecera.

El ***serialize*dPayload**, presenta información solamente si el *DataFlag* o el *KeyFlag* cumplen las siguientes condiciones.

- Si el DataFlag tiene el valor de 1, entonces contiene el valor encapsulado del cambio.
- Si el KeyFlag tiene el valor de 1, entonces contiene la encapsulación de la clave del submensaje.

El campo **extraFlags**, ofrece espacio para 16 bits adicionales para banderas a parte de los 8 bits ya proporcionados. Estos bits adicionales apoyarán la evolución del protocolo sin comprometer la compatibilidad hacia atrás. Esta versión del protocolo debe establecer todos los bits en el *extraFlags* en cero.

El campo **octetsToInlineQos,** es representado por un *shortUnsignedCDR*. contiene el número de octetos que representan al *inlineQos*. Si el *inlineQos* no está presente (es decir, el InlineQosFlag no está establecido), entonces el octetsToInlineQos contiene el desplazamiento al siguiente campo después de la *inlineQos*.

Las implementaciones del protocolo que están procesando un submensaje recibido siempre deben usar el *octetsToInlineQos* para omitir cualquier elemento de encabezado submensaje.

2.2.4.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando en el encabezado del submensaje el *submessageLength* es demasiado pequeño.
- Cuando el *WriterSN.value* no es estrictamente positivo o es un numero de secuencia desconocido.
- Cuando el *inlineQos* no es válido.

2.2.4.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

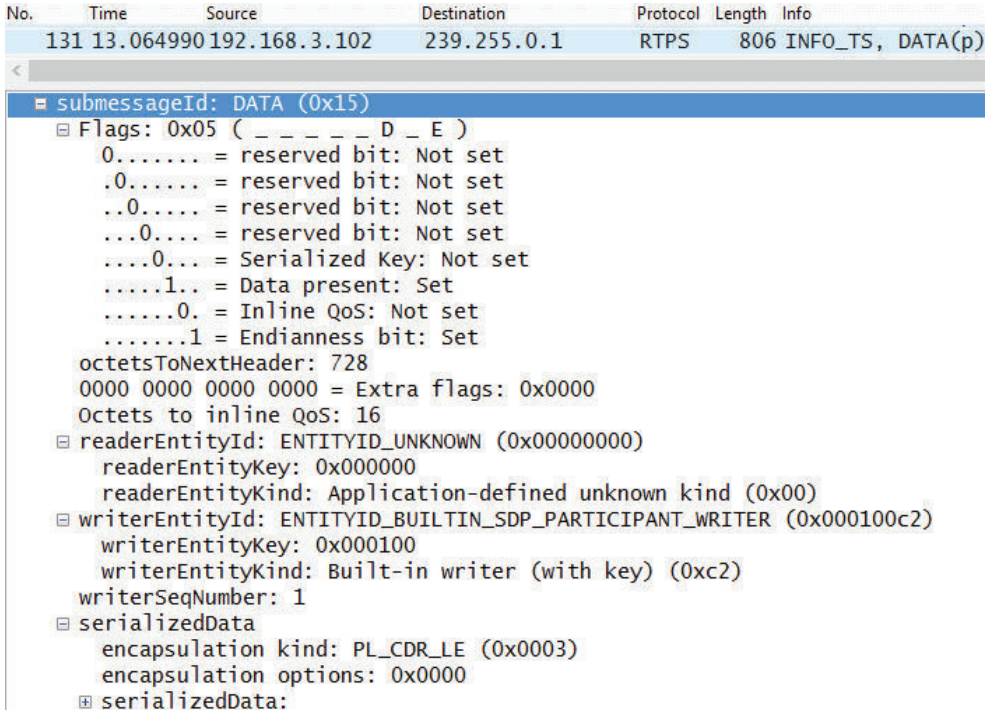


Figura 2.7. Uso del submensaje DATA.

2.2.4.5. Interpretación lógica

El escritor envía el submensaje data al lector para comunicar los cambios realizados. Estos incluyen los cambios en el valor, así como los cambios en el ciclo de vida del objeto de datos. Éstos se comunican por la presencia del *serializedPayload*. Cuando están presentes, el *serializedPayload* se interpreta como el valor del objeto de datos o como la clave que identifica el objeto de datos del conjunto de objetos registrados.

- Si el DataFlag es igual a 1 y el KeyFlag es igual a 0, el elemento *serializedPayload* se interpreta como el valor del objeto *dataobject*.
- Si el *DataFlag* es igual a 0 y el KeyFlag es igual a 1, el elemento *serializedPayload* se interpreta como el valor de la clave que identifica la instancia del objeto de datos registrada.
- Si el InlineQosFlag es igual a 1, el elemento inlineQos contiene los valores de QoS que reemplazan a los del escritor RTPS.

2.2.4.5.1. Ejemplo

En la Figura 2.7 se muestra la captura del submensaje *DATA*, en el cual se puede visualizar los campos previamente descritos con sus valores.

Una explicación del uso del Data ha sido descrita dentro en el capítulo 3 en el ejemplo 2 de la sección 3.3.5

2.2.5. DATAFRAG SUBMESSAGE

En la Figura 2.8 se muestra la estructura del submensaje *DataFrag*.

El submensaje *DataFrag* es un submensaje de datos que es fragmentado y enviado como múltiples submensajes. Los fragmentos contenidos en los submensajes *DataFrag* son reensamblados por el lector RTPS posteriormente.

Un submensaje *DataFrag*, ofrece las siguientes ventajas:

- Mantiene las variaciones en el contenido y estructura de cada submensaje al mínimo.
- Evita tener que agregar información de fragmentación como parámetros del *inlineQoS* en el submensaje *data*.

2.2.5.1. Banderas en el encabezado de submensaje

El **InlineQosFlag**, indica que la entidad lectora está siendo informada del cambio: cuando se establece en 1 significa que el submensaje DataFrag contiene el inlineQos.

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **KeyFlag**, cuando es igual a 0, significa que el *serializedPayload* contiene los datos serializados, y cuando es igual 1 significa que el *serializedPayload* contiene la clave serializada.

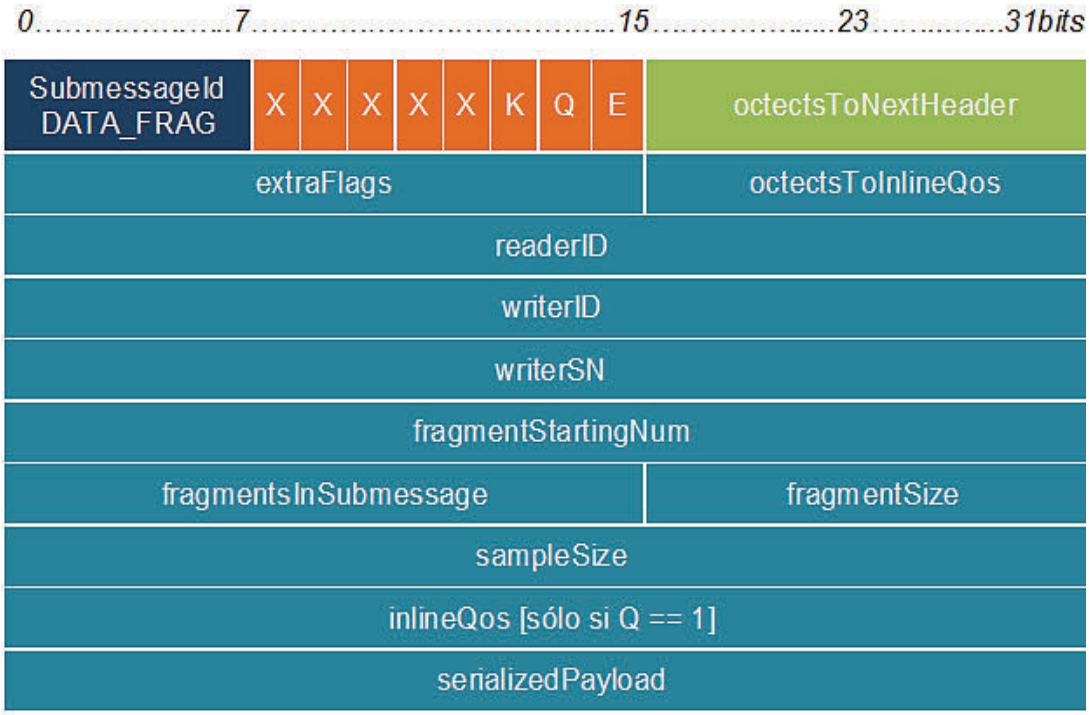


Figura 2.8. Estructura del submensaje DataFrag [13]

2.2.5.2. Otros elementos en el encabezado del submensaje

El ***Reader*ID**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El ***Writer*ID** , identifica la entidad RTPS escritora que realizó el cambio.

El ***Writer*SN**, identifica el cambio y el orden relativo de todos los cambios realizados por el escritor RTPS identificados por el *Writer*Guid. Cada cambio obtiene un número de secuencia consecutiva. Cada escritor RTPS mantiene su propio número de secuencia.

El **fragmentStartingNum**, indica el fragmento inicial de la serie de fragmentos de *serializedData*. La numeración de los fragmentos comienza con el número 1.

El **fragmentInSubmessage**, indica el número de fragmentos consecutivos contenidos en este submensaje, a partir de las *fragmentStartingNum*.

El **dataSize**, indica el tamaño total en bytes de los datos originales antes de la fragmentación.

El **fragmentSize**, indica el tamaño de un fragmento individual en bytes. El tamaño del fragmento máximo equivale a 64K.

El **inlineQos**, presenta información de QoS solamente si el *InlineQosFlag* está situado en la cabecera.

El ***serialize*dPayload**, presenta información sólo si el *DataFlag* es igual a 1.

El *DataFrag* representa parte del valor del objeto de datos-nuevo después del cambio.

- Si el *DataFlag* es igual a 1, entonces contiene un conjunto de fragmentos de la encapsulación del nuevo valor del objeto *dataobject* después del cambio consecutivo.
- Si el *KeyFlag* es igual a 1, contiene un conjunto de fragmentos de la encapsulación de la clave del mensaje.

En cualquier caso, el conjunto consecutivo de fragmentos contiene partes de *fragmentsInSubmessage* y comienza con el fragmento identificado por *fragmentStartingNum*.

2.2.5.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el *WriterSN.value* no es estrictamente positivo o es un numero de secuencia desconocido.
- Cuando *fragmentStartingNum* no es estrictamente positivo o excede el número total de fragmentos.
- Cuando el *fragmentSize* excede al *dataSize*.
- Cuando el tamaño del *serializedData* es superior al producto *fragmentsInSubmessage * fragmentSize*.
- Cuando el *inlineQos* no es válido.

2.2.5.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.5.5. Interpretación lógica

El submensaje *DataFrag* extiende el submensaje data usando el *serializedData* a ser fragmentado y enviado como múltiples submensajes. Una vez que el *serializedData* llega al lector RTPS, la interpretación de los submensajes DataFrag es idéntica a la del submensaje Data.

A continuación, se describe cómo volver a unir la información dentro de los submensaje DataFrag.

El tamaño total de los datos que se reensambla está dada por el *dataSize*. Cada submensaje DataFrag contiene un segmento contiguo de estos datos en su elemento *serializedData*. El tamaño del segmento se determina por la longitud del elemento *serializedData*. Durante la desfragmentación, el desplazamiento de cada segmento se determina por:

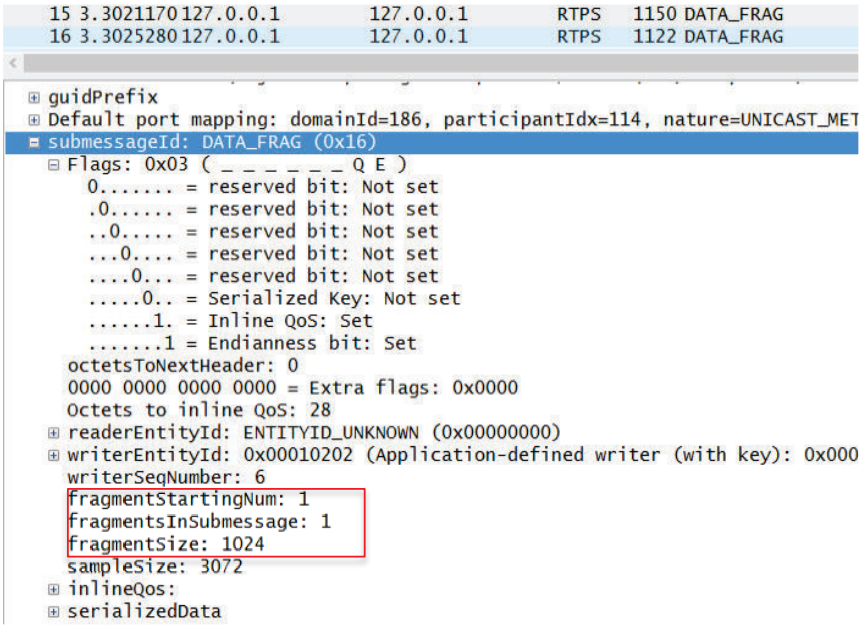


Figura 2.9. Uso del submensaje DATA_FRAG (parte I).

(*fragmentStartingNum* - 1) * *fragmentSize*

Los datos son reensamblados completamente cuando se han recibido todos los fragmentos. El número total de fragmentos esperado equivale a:

(*dataSize* / *fragmentSize*) + ((*dataSize* % *fragmentSize*))

Tenga en cuenta que cada submensaje DataFrag puede contener múltiples fragmentos. Un escritor RTPS seleccionará al *fragmentSize* basándose en el tamaño más pequeño de mensaje soportado a través de todos los transportes. Si algunos lectores RTPS pueden ser alcanzados a través de un transporte que

soporte mensajes más grandes, el escritor RTPS puede empaquetar los múltiples fragmentos en un sólo submensaje DataFrag o incluso puede enviar un submensaje regular de datos si la fragmentación ya no es necesaria.

2.2.5.5.1. Ejemplo

En las siguientes figuras se muestra dos mensajes RTPS fragmentados. Dentro de la Figura 2.9 se puede observar los campos previamente descritos, y además se puede observar en el contorno rojo la fragmenación de la información.

```
⊟ submessageId: DATA_FRAG (0x16)
  ⊟ Flags: 0x01 ( _ _ _ _ _ _ _ E )
      0....... = reserved bit: Not set
      .0...... = reserved bit: Not set
      ..0..... = reserved bit: Not set
      ...0.... = reserved bit: Not set
      ....0... = reserved bit: Not set
      .....0.. = Serialized Key: Not set
      ......0. = Inline QoS: Not set
      .......1 = Endianness bit: Set
    octetsToNextHeader: 0
    0000 0000 0000 0000 = Extra flags: 0x0000
    Octets to inline QoS: 28
  ⊞ readerEntityId: ENTITYID_UNKNOWN (0x00000000)
  ⊞ writerEntityId: 0x00010202 (Application-defined writer (with key): 0x000102)
    writerSeqNumber: 6
    fragmentStartingNum: 3
    fragmentsInSubmessage: 1
    fragmentSize: 1024
    sampleSize: 3072
  ⊞ serializedData
```

Figura 2.10. Uso del submensaje DATA_FRAG (parte II).

2.2.6. GAP SUBMESSAGE

En la Figura 2.11 se muestra la estructura del submensaje *GAP*.

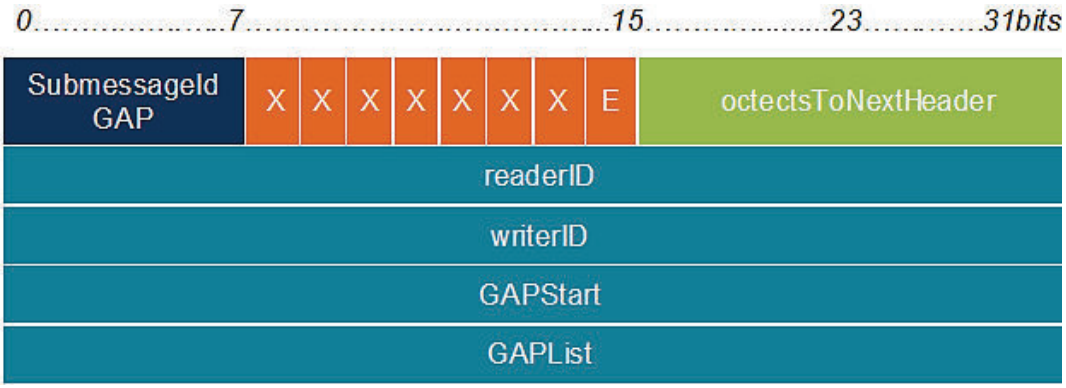


Figura 2.11. Estructura del submensaje GAP [13]

Este submensaje es enviado de un escritor RTPS a un lector RTPS e indica al lector que un rango de números de secuencia ya no es relevante. El conjunto puede ser un rango contiguo de números de secuencia o un conjunto específico de números de secuencia.

2.2.6.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.6.2. Otros elementos en el encabezado de submensaje

El ***Reader*ID**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El ***Writer*ID** , identifica la entidad RTPS escritora que realizó el cambio.

El **GAPStart**, identifica el primer número de secuencia.

El **GAPList**, tiene dos propósitos:

- Identifica el último número de la secuencia en el intervalo.
- Identifica una lista adicional de números de secuencia que son irrelevantes.

2.2.6.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el *GAPStart* es cero o negativo.
- Cuando el *GAPList* no es válido.

2.2.6.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.6.5. Interpretación lógica

El escritor RTPS envía el submensaje *GAP* al lector RTPS para comunicar que ciertos números de secuencia ya no son relevantes. Esto es causado típicamente en el lado del escritor. En este escenario, nuevos valores de los datos podrán sustituir a los viejos valores de los objetos de datos que fueron representados por los números de secuencia que aparecen como irrelevantes en el submensaje.

Los números de secuencia irrelevantes comunicados por el submensaje *GAP* se componen de dos grupos:

- Toda la secuencia de números en el rango *GAPStart* < = *sequence_number* < = *GAPList*.*base* -1
- Todos los números de secuencia que aparecen explícitamente enumerados en el *GAPList*.

2.2.6.5.1. Ejemplo

En la Figura 2.12 se muestra la captura del submensaje *GAP*. Dentro de a figura se observa los diferentes campos descritos previamente.

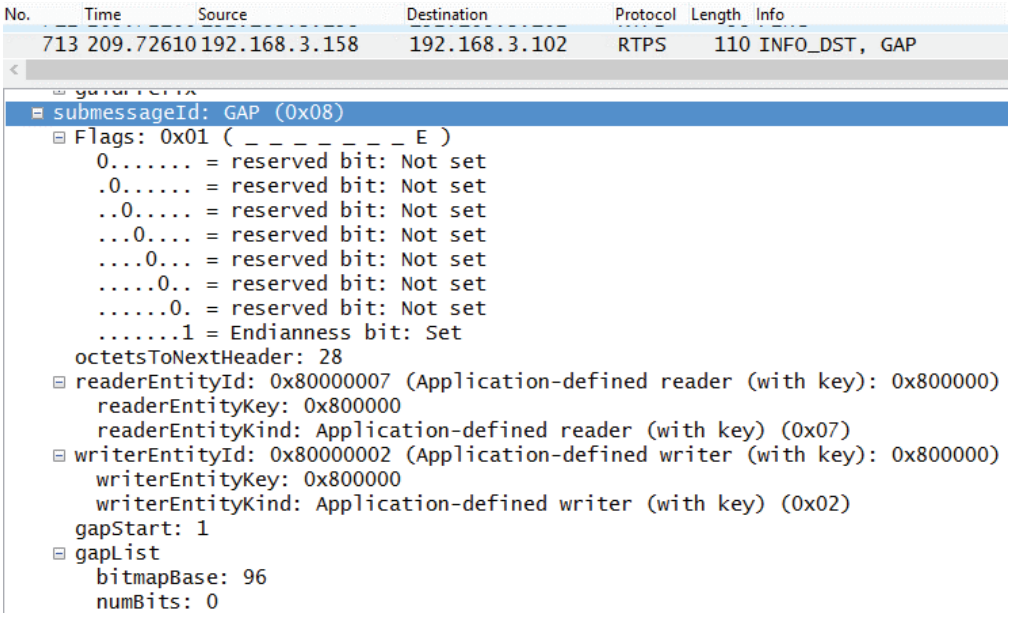


Figura 2.12. Uso del submensaje GAP.

2.2.7. HEARTBEAT SUBMESSAGE

En la Figura 2.13 se muestra la estructura del submensaje *Heartbeat*.

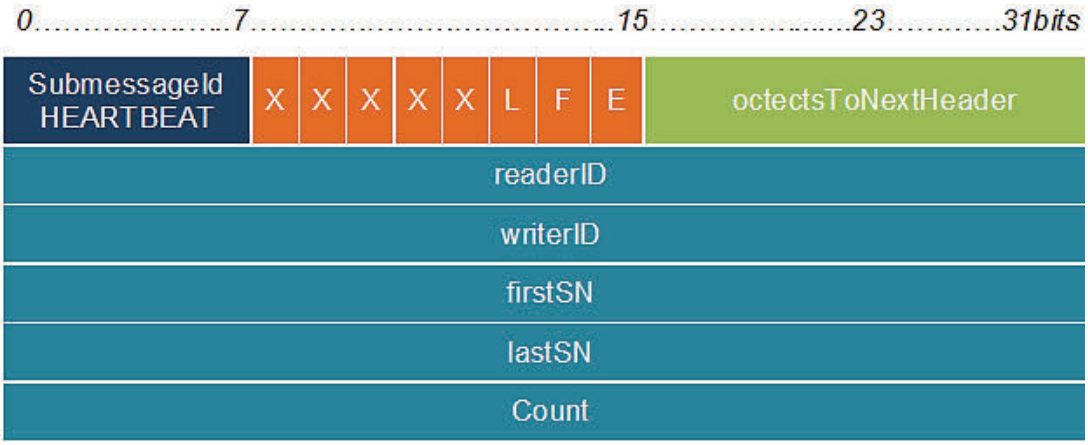


Figura 2.13. Estructura del submensaje Heartbeat [13]

Este mensaje se envía desde un escritor RTPS a un lector RTPS para comunicar la secuencia de cambios que el escritor tiene disponibles.

2.2.7.1. Banderas en el encabezado del submensaje

El **FinalFlag**, indica si el lector es necesita para responder al submensaje o si es sólo *Heartbeat* de control. El FinalFlag se representa con el literal 'F'. Cuando

F = 1 significa que el escritor no requiere una respuesta desde el lector y cuando es igual a 0 significa que el lector debe responder al submensaje Heartbeat.

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **LivelinessFlag**, indica que el escritor de datos DDS asociado con el escritor RTPS se encuentra presente. El LivelinessFlag se representa con el literal 'L'. Cuando L = 1 significa que el *DataReader* DDS asociado con el lector de RTPS debe actualizar manualmente el *liveliness* del *DataWriter* DDS asociado con el escritor RTPS del mensaje.

2.2.7.2. Otros elementos en el encabezado de submensaje

El ***Reader*ID**, identifica la entidad RTPS lectora que está siendo informada de los cambios. En este submensaje se puede configurar en ENTITYID_*UNKNOWN* para indicar a todos los lectores que el escritor envió el submensaje.

El ***Writer*ID** , identifica la entidad RTPS escritora que realizó el cambio.

El **lastSN**, identifica el último número de secuencia que está disponible en el escritor.

El **Count**, se incrementa cada vez que se envía un nuevo submensaje *Heartbeat*. Este proporciona los medios para que un lector detecte los submensajes *Heartbeat* duplicados que pueden derivarse de la presencia de las diferentes vías de comunicación.

2.2.7.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el valor del *firstSN* es cero o negativo.
- Cuando el valor del *lastSN* es cero o negativo.
- Cuando el valor *lastSN* es menor que el valor del *firstSN*.

2.2.7.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.7.5. Interpretación lógica

El submensaje Heartbeat tiene dos propósitos:

- Informar al lector de los números de secuencia que están disponibles en el *HistoryCache* del escritor para que el lector pueda solicitar (mediante un *AckNack*) cuales le han faltado.
- Pedir al lector enviar un acuse de recibo para los cambios *CacheChange* que han sido introducidos en la *HistoryCache* del lector para que el escritor conozca el estado del lector.

El lector encontrará siempre el estado del *HistoryCache* del escritor y puede solicitar lo que ha perdido. Normalmente, el lector RTPS sólo enviaría un mensaje *AckNack* si le falta un cambio.

El escritor utiliza el *FinalFlag* para solicitar al lector el envío de un acuse de recibo para los números de secuencia que ha recibido. Si el *Heartbeat* tiene el *FinalFlag* es igual a 1, no es necesario que el lector envíe un mensaje *AckNack*. Sin embargo, si el *FinalFlag* es igual a 0, entonces el lector debe enviar mensaje de AckNack que indica que ha recibido el cambio, aunque el *AckNack* indica que ha recibido todos los cambios en *HistoryCache* del escritor.

2.2.7.5.1. Ejemplo

En la Figura 2.5 se muestra la captura del submensaje *Heartbeat*., y en la cual se puede observar los diferentes campos fueron descritos previamente.

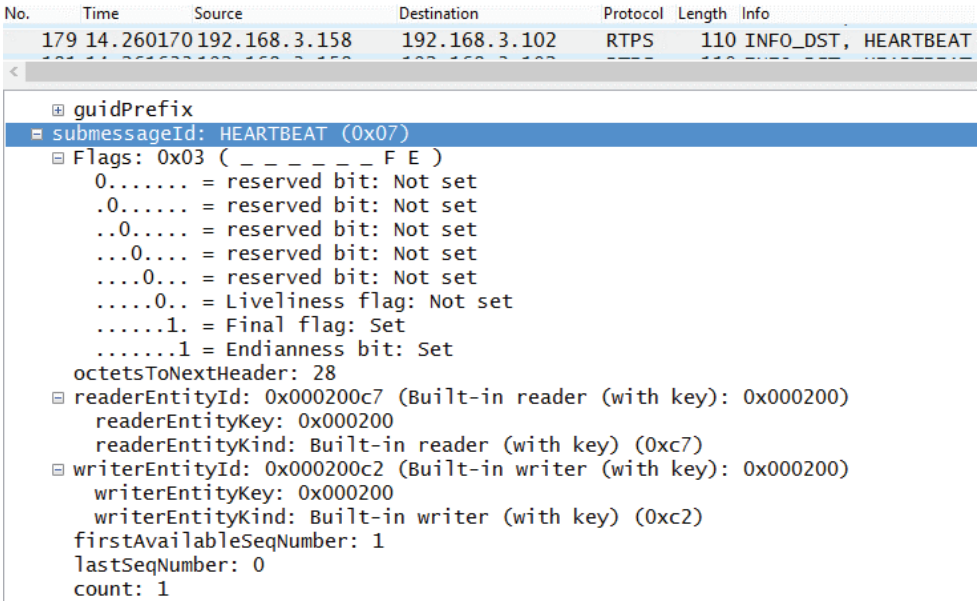


Figura 2.14. Uso del submensaje HEARTBEAT.

Una explicación del uso del *Heartbeat* se ha descrito dentro del ejemplo 2 y 3 en la sección 3.3.5 de ejemplos de RTPS.

2.2.8. HEARTBEATFRAG SUBMESSAGE

En la Figura 2.15 se muestra la estructura del submensaje *HeartBeatFrag*.

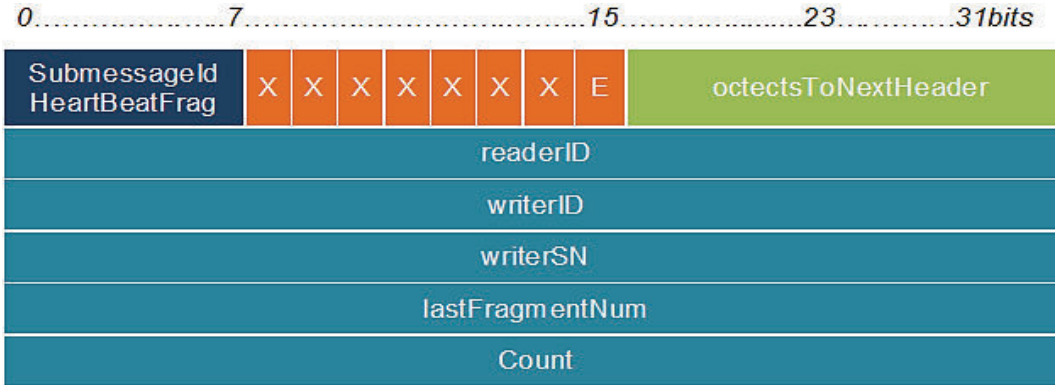


Figura 2.15. Estructura del submensaje HeartBeatFrag [13]

El submensaje *HeartbeatFrag* se envía desde un escritor a un lector para comunicar que los fragmentos del escritor están a su disposición. Esto permite una comunicación segura a nivel de fragmento. Una vez que todos los fragmentos están disponibles, se utiliza un mensaje regular de *Heartbeat*.

2.2.8.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.8.2. Otros elementos en el encabezado de submensaje

El ***Reader*ID**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El ***Writer*ID** , identifica la entidad RTPS escritora que realizó el cambio.

El ***Writer*SN**, identifica el número de secuencia de los cambios para que los fragmentos estén disponibles.

El **lastFragmentNum**, indica todos los fragmentos están disponibles en el escritor para el cambio identificado por el *WriterSN*.

El **Count**, se incrementa cada vez que se envía un mensaje HeartbeatFrag. Este proporciona los medios para que un lector detecte los submensajes *HeartbeatFrag* duplicados que pueden derivarse de la presencia de las diferentes vías de comunicación.

2.2.8.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el *WriterSN*.value es cero o negativo.
- Cuando el *lastFragmentNum*.value es cero o negativo.

2.2.8.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

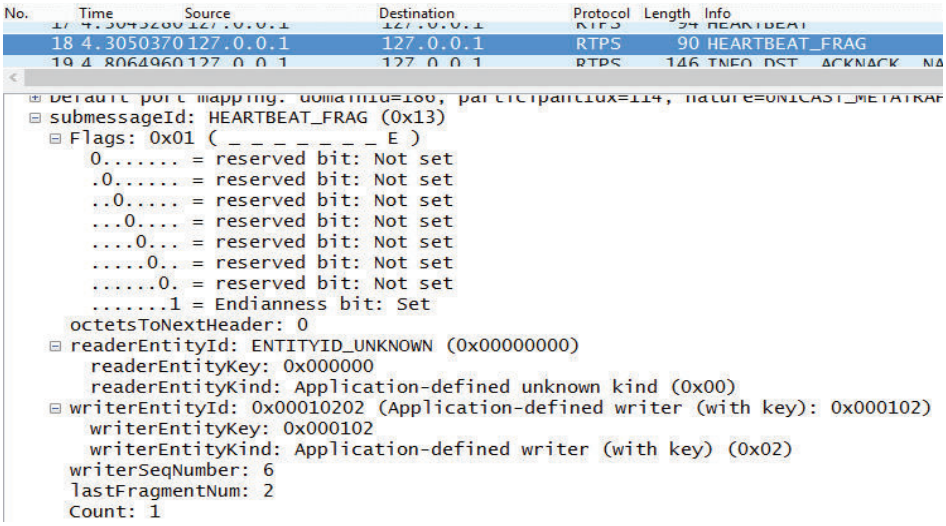


Figura 2.16. Uso del submensaje HEARTBEAT_FRAG.

2.2.8.5. Interpretación lógica

El submensaje de *HeartbeatFrag* tiene el mismo propósito que un submensaje *Heartbeat* regular, pero en lugar de lo que indica la disponibilidad de una gama de números de secuencia, indica la disponibilidad de una gama de fragmentos para el cambio de datos con el número de secuencia *WriterSN*.

El lector RTPS responderá enviando un mensaje *NackFrag*, pero sólo si le falta alguno de los fragmentos disponibles.

2.2.8.5.1. Ejemplo

En la Figura 2.16 se muestra la captura del submensaje *HeartbeatFrag*, y en donde se puede observar los campos que fueron descritos previamente.

2.2.9. INFODESTINATION SUBMESSAGE

En la Figura 2.17 se muestra la estructura del submensaje *InfoDestination*.

Este mensaje se envía desde un escritor RTPS a un lector RTPS para modificar el *GuidPrefix* utilizado para interpretar el *entityId* del lector que aparece en los siguientes submensajes.

2.2.9.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

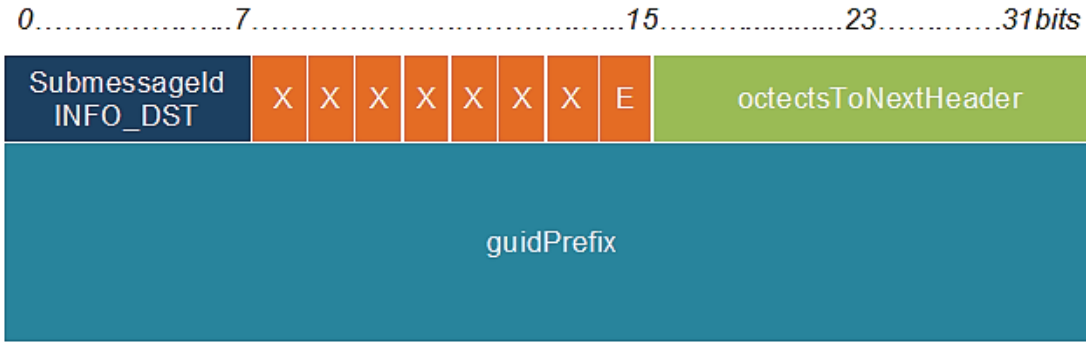


Figura 2.17. Estructura del submensaje InfoDestination [13]

2.2.9.2. Otros elementos en el encabezado del submensaje

El **guidPrefix**, proporciona la identificación que debe utilizarse para reconstruir los GUID de todos los lectores RTPS cuyo *EntityId* aparece en los siguientes submensajes.

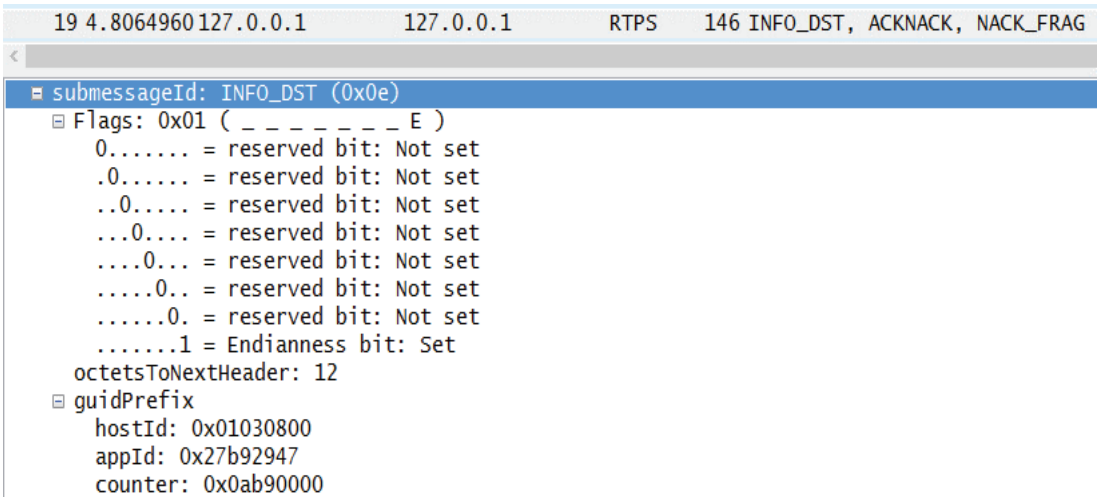


Figura 2.18. Uso del submensaje INFO_DESTINATION.

2.2.9.3. Validez del submensaje

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.9.4. Cambio en el estado del receptor

Existe un cambio en el estado del recepto cuando se cumple con (InfoDestination.guidPrefix! = GUIDPREFIX_UNKNOWN)

2.2.9.4.1. Ejemplo

En la Figura 2.18 se muestra la captura del submensaje *INFO_DESTINATION*, y se puede observar los campos descritos previamente.

Una explicación del uso del *InfoDestination* se encuentra descrito dentro del ejemplo 2 en la sección 3.3.5 de ejemplos de RTPS.

2.2.10. INFOREPLY SUBMESSAGE

En la Figura 2.19 se muestra la estructura del submensaje *InfoReply*.

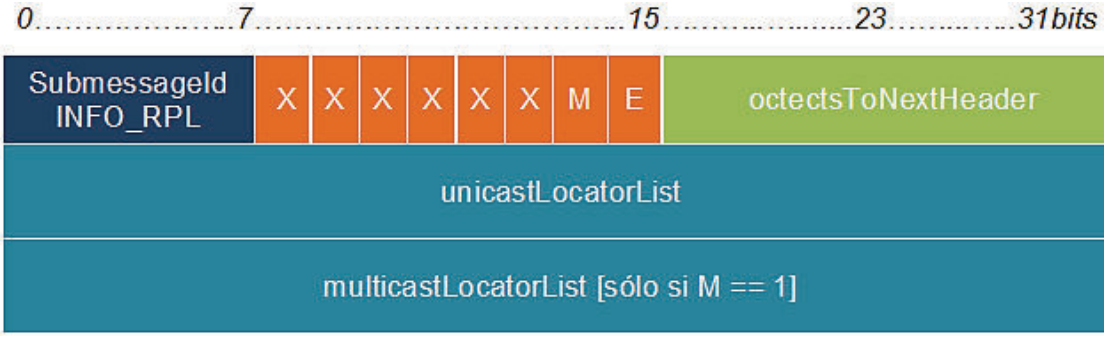


Figura 2.19. Estructura del submensaje InfoReply [13]

Este submensaje se envía desde un lector RTPS a un escritor RTPS. Contiene información explícita sobre dónde enviar una respuesta a los submensajes que le siguen en el mismo mensaje.

2.2.10.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **MulticastFlag**, indica si el submensaje también contiene una dirección multicast. El MulticastFlag está representado con el literal M. Si este se encuentra en 1 significa que el submensaje InfoReply también incluye un *multicastLocatorList*.

2.2.10.2. Otros elementos en el encabezado del submensaje

La **unicastLocatorList**, indica un conjunto alternativo de direcciones unicast que el escritor debe utilizar para alcanzar los lectores cuando se replican los submensajes que le siguen.

La **multicastLocatorList**, indica un conjunto alternativo de direcciones de multidifusión que el escritor debe utilizar para llegar a los lectores cuando se replican los submensajes que siguen.

2.2.10.3. Validez del submensaje

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.10.4. Cambio en el estado del receptor

Se produce un cambio en el estado del receptor cuando la siguiente condición es cumplida:

Receiver.unicastReplyLocatorList = InfoReply.unicastLocatorList if (MulticastFlag) {Receiver.multicastReplyLocatorList = InfoReply.multicastLocatorList} más {Receiver.multicastReplyLocatorList = < vacío >}

2.2.11. INFOSOURCE SUBMESSAGE

En la Figura 2.20 se muestra la estructura del submensaje *InfoSource*.

Figura 2.20. Estructura del submensaje InfoSource [13]

Este mensaje modifica la fuente de los submensajes que siguen. Es decir, este contiene la dirección fuente de los submensajes que se encuentran junto al mismo.

2.2.11.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.11.2. Otros elementos en el encabezado del submensaje

El **protocolVersion**, indica la versión del protocolo utilizado para encapsular submensajes posteriores.

El **identifier,** indica el identificador del fabricante del producto que encapsula submensajes posteriores.

El **guidPrefix**, identifica el participante, es el contenedor del escritor RTPS que representa la fuente de los submensajes posteriores.

2.2.11.3. Validez del submensaje

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.11.4. Cambio en el estado del receptor

Se produce un cambio en el estado del receptor cuando se cumpla cualquiera de las siguientes condiciones:

- Receiver.sourceGuidPrefix = InfoSource.guidPrefix
- Receiver.sourceVersion = InfoSource.protocolVersion
- Receiver.sourceVendorId = InfoSource.vendorId
- Receiver.unicastReplyLocatorList = {LOCATOR_INVALID}
- Receiver.multicastReplyLocatorList = {LOCATOR_INVALID}
- haveTimestamp = false

2.2.12. INFOTIMESTAMP SUBMESSAGE

En la Figura 2.21 se muestra la estructura del submensaje *InfoTimestamp*.

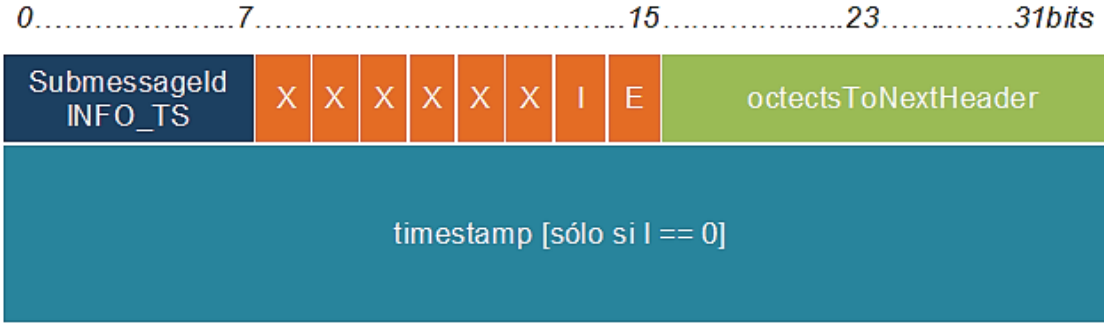


Figura 2.21. Estructura del submensaje InfoTimestamp [13]

Este submensaje se utiliza para enviar una marca de tiempo que se aplica a los submensajes que siguen dentro del mismo mensaje.

2.2.12.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **InvalidateFlag**, indica si los submensajes posteriores deben considerar la marca de tiempo o no. El *InvalidateFlag* se representa con el literal 'I'. Cuando este es igual a 0 significa que el *InfoTimestamp* también incluye una marca de tiempo. Mientras que si es igual a 1 significa que los submensajes posteriores no deben considerar que tienen una marca de tiempo válida.

2.2.12.2. Otros elementos en el encabezado del submensaje

El **Timestamp**, presenta solamente la marca de tiempo si el *InvalidateFlag* no se encuentra en la cabecera. La marca de tiempo debe utilizarse para interpretar los submensajes posteriores.

2.2.12.3. Validez del submensaje

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.12.4. Cambio en el estado del receptor

El cambio en el estado del receptor se produce cuando la siguiente condición es verdadera:

if (! InfoTimestamp.InvalidateFlag) {Receiver.haveTimestamp = true Receiver.timestamp = InfoTimestamp.timestamp} más {Receiver.haveTimestamp = false}

2.2.12.4.1. Ejemplo

En la Figura 2.22 se muestra la captura del submensaje *INFO_TIMESTAMP*, y donde se observa los campos descritos previamente.

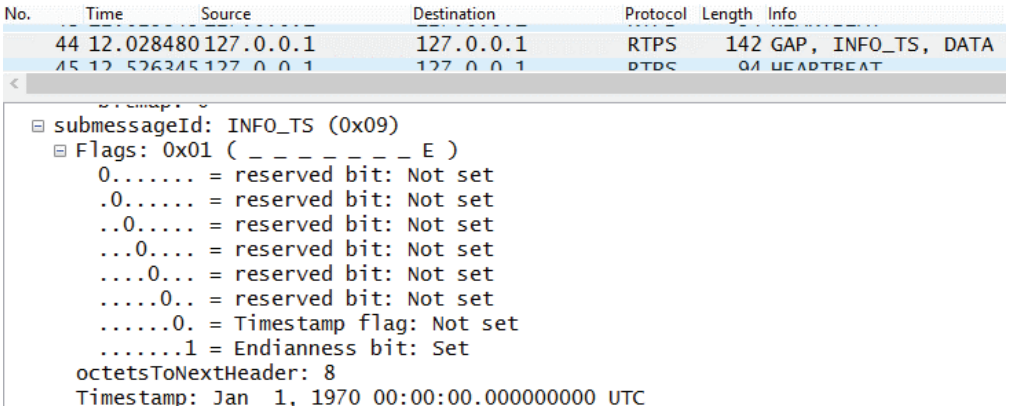


Figura 2.22. Uso del submensaje INFO_TS.

Una explicación del uso del InfoTimeStamp se ha descrito dentro del ejemplo 2 en la sección 3.3.5 de ejemplos de RTPS.

2.2.13. NACKFRAG SUBMESSAGE

En la Figura 2.23 se muestra la estructura del submensaje *NackFrag*.

El submensaje *NackFrag* se utiliza para comunicar el estado de un lector a un escritor. Cuando se envía un cambio con una serie de fragmentos, es decir, se envía un dato con un submensaje *DataFrag*, el submensaje *NackFrag* permite al

lector dar a conocer al escritor asociado los números fragmento específicos que aún le faltan.

Este submensaje sólo puede contener acuses de recibo negativos. Tenga en cuenta que esto difiere de un submensaje *AckNack*, el cual incluye acuses de recibo tanto positivos como negativos. Las ventajas de este enfoque incluyen:

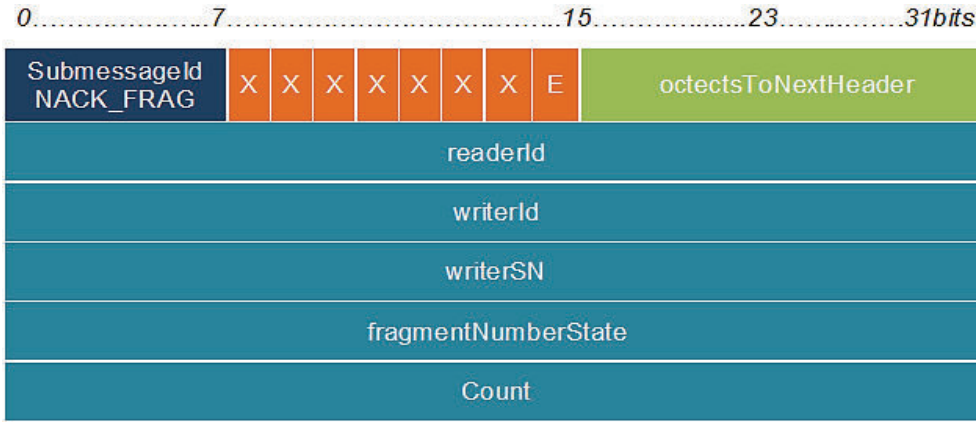


Figura 2.23. Estructura del submensaje NackFrag [13]

- Eliminar la limitación de ventanas introducida por el submensaje *AckNack*. Dado que el tamaño de un *SequenceNumberSet* se limita a 256 bits, un submensaje *AckNack* se limita solamente a acusar de forma negativa a aquellas muestras cuyo número de secuencia no exceda a aquellos submensajes que primero se perdieron por más de 256 bits. Cualquier muestra que se encuentre por debajo de aquellas que se perdieron primero, son confirmadas. Los submensajes *NackFrag* por otro lado pueden ser utilizados para acusar negativamente cualquier fragmento, incluso fragmentos que tengan más 256 bits aparte de aquellos que ya han sido acusados negativamente por un submensaje *AckNack* anterior. Esto llega a ser importante cuando manejo las muestras contienen una gran cantidad de fragmentos.
- Los Fragmentos pueden ser acusados negativamente en cualquier orden.

2.2.13.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.13.2. Otros elementos en el encabezado de submensaje

El ***Reader*ID**, identifica a la entidad lectora que pide recibir ciertos fragmentos.

El ***Writer*ID**, identifica la entidad escritora. Este es el identificador del escritor que pide el reenvío de algunos fragmentos

El ***Writer*SN**, es el número de secuencia de los fragmentos faltantes.

El **fragmentNumberState**, comunica el estado del lector al escritor. Los números de fragmento que aparecen en el conjunto indican fragmentos perdidos en el lado del lector. Los que no aparecen en el conjunto podrían haber sido ya recibidos o no.

El **Count**, se incrementa cada vez que se envía un nuevo submensaje *NackFrag*. Proporciona los medios para que un escritor detecte los submensajes duplicados de *NackFrag* que pueden derivarse de la presencia de las distintas vías de comunicación.

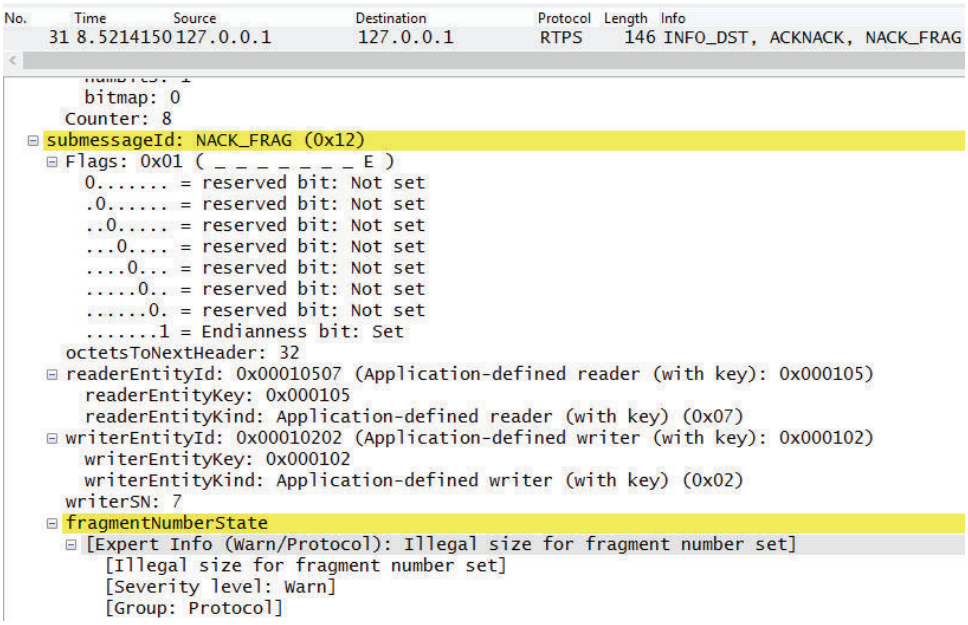


Figura 2.24. Uso del submensaje NACK_FRAG.

2.2.13.3. Validez del submensaje

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el valor del *WriterSN* es cero o negativo.
- Cuando el *fragmentNumberState* no es válido.

2.2.13.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor.

2.2.13.5. Interpretación lógica

El lector envía el submensaje *NackFrag* al escritor para solicitar fragmentos.

2.2.13.5.1. Ejemplo

En la Figura 2.24 se muestra la captura del submensaje *NACK_FRAG*, y en donde la figura se observa los campos descritos previamente.

2.2.14. PAD SUBMESSAGE

En la Figura 2.25 se muestra la estructura del submensaje *Pad*.

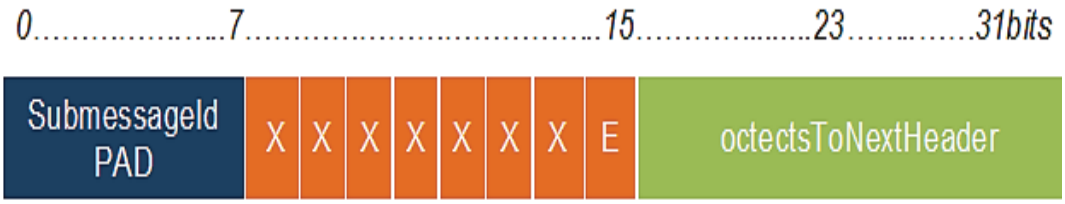


Figura 2.25. Estructura del submensaje Pad [13]

El propósito de este submensaje es permitir la introducción de cualquier relleno necesario para satisfacer cualquier requisito de alineamiento en la memoria que sea deseado.

2.2.14.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.14.2. Otros elementos en el encabezado del submensaje

Este submensaje no tienen otros elementos.

2.2.14.3. Validez del submensaje

Este submensaje siempre es válido.

2.2.14.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor.

2.2.15. INFOREPLYIP4 SUBMESSAGE

Este submensaje se envía desde un lector RTPS a un escritor RTPS. Contiene información explícita sobre dónde enviar una respuesta a los submensajes que le siguen en el mismo mensaje.

Este submensaje por motivos de eficiencia, puede utilizarse en lugar del submensaje *InfoReply* para proporcionar una representación más compacta.

2.2.15.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **MulticastFlag**, indica si el submensaje también contiene una dirección multicast. El MulticastFlag está representado con el literal M. Si este se encuentra en 1 significa que el submensaje *InfoReplyIp4* también incluye un *multicastLocatorList*.

2.2.15.2. Otros elementos en el encabezado del submensaje

La **unicastLocatorList**, indica un conjunto alternativo de direcciones unicast que el escritor debe utilizar para alcanzar los lectores cuando se replican los submensajes que le siguen.

La **multicastLocatorList**, indica un conjunto alternativo de direcciones de multidifusión que el escritor debe utilizar para llegar a los lectores cuando se replican los submensajes que siguen. Sólo está cuando se establece la MulticastFlag en 1.

En la Figura 2.26 se muestra la estructura del submensaje *InfoReplyIp4*.

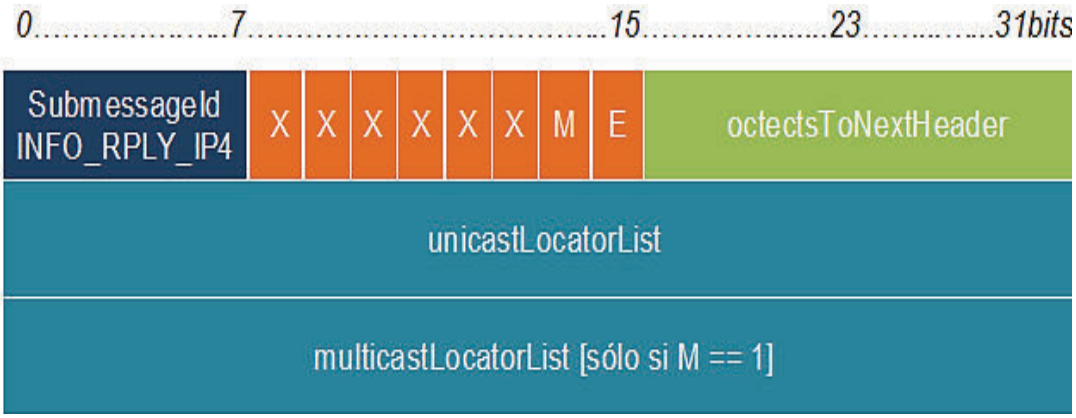


Figura 2.26. Estructura del submensaje InfoReplyIp4 [13]

2.2.15.3. Validez del submensaje

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.15.4. Cambio en el estado del receptor

Se produce cambios en el estado del receptor cuando la siguiente condición se cumple:

Receiver.unicastReplyLocatorList = InfoReply.unicastLocatorList if (MulticastFlag) {Receiver.multicastReplyLocatorList = InfoReply.multicastLocatorList} más {Receiver.multicastReplyLocatorList = < vacío >}

2.3. ANÁLISIS DE REQUISITOS

Una vez realizado el análisis de los paquetes RTPS y en conjunto con los participantes del proyecto *CEPRA VIII-2014-06 Middleware en Tiempo Real basado en el modelo publicación/suscripción*, se definió los requerimientos necesarios para soportar el protocolo RTPS con el Middleware DDS, por lo que se llegó a la conclusión que es necesario cumplir con los siguientes requisitos:

- Se deberá implementar los componentes necesarios del protocolo RTPS y el mecanismo de descubrimiento proporcionado por el mismo. Además, se deberá implementar un submódulo de transporte, un submódulo de mensajes y encapsulación, un submódulo de descubrimiento, un submódulo de comportamiento, y un submódulo de configuración.
- Se deberá implementar los componentes básicos que conforman el DDS, tales como el Publicador, Suscriptor y el *Topic*. Para este requisito se necesita la implementación del DDS que se encuentra incluida en los anexos y fue parte de la implementación realizada por el proyecto *CEPRA VIII-2014-06 Middleware en Tiempo Real basado en el modelo publicación/suscripción*.
- Se deberá implementar los mecanismos y técnicas para el alcance de la información, es decir, se deberá organizar los datos dentro de cada dominio. Para cumplir con este requisito se requiere la implementación del submódulo comportamiento y además se requiere la implementación del *Topic* encargado del proyecto *CEPRA VIII-2014-06 Middleware en Tiempo Real basado en el modelo publicación/suscripción* que se encuentra en los anexos.
- Se deberá implementar los mecanismos de Lectura y Escritura de datos, y el ciclo de vida de los *Topic*. Para este cumplir con este requisito se requiere la implementación del submódulo mensaje y encapsulación y de DDS el cual se encuentra descrito en los anexos.

CAPÍTULO 3.
DISEÑO E IMPLEMENTACIÓN DE UN MÓDULO QUE PERMITA INTERACTUAR AL PROTOCOLO RTPS CON DDS

3.1. INTRODUCCIÓN

En este capítulo primeramente se diseña un módulo que permita la interacción entre RTPS y DDS y que trabaje con el modelo Publicador/Suscriptor, por medio de diagramas de clases del módulo RTPS. Para poder realizar el diseño del módulo se utiliza como guía al API-RTPS, para luego adaptar los diseños de los submódulos a las necesidades del proyecto.

Seguidamente, se presenta la interacción entre DDS y RTPS con el modelo publicador/suscriptor por medio de diagramas de secuencia.

Finalmente, se muestra la implementación del diseño, utilizando normas de convención, implementación del protocolo RTPS, implementaciones necesarias DDS, implementación de archivos de configuración.

En base a los requisitos obtenidos en el capitulo 2, se decidió organizar tanto el diseño como la implementación, en base a submódulos.

3.2. DISEÑO DEL MÓDULO

Una vez recopilada la información necesaria, y establecidos los requerimientos para soportar el protocolo RTPS con el Middleware DDS, en esta sección se realiza el diseño del módulo que permita la interacción entre el RTPS y el DDS.

En el desarrollo del sistema no se sigue una metodología rígida, sino una metodología que se consolida a medida que se avanza en la investigación, es por eso que se presentan dentro de esta sección los diagramas de clases que se han obtenido al final del desarrollo del proyecto *CEPRA VIII-2014-06 Middleware en Tiempo Real basado en el modelo publicación/suscripción*.

En base a los submódulos descritos en el API-RTPS, se adaptarán los mismos y además se agrega al diseño dos submódulos extras, los cuales son: el submódulo de transporte y el submódulo de configuración. El submódulo de transporte es el encargado del envío y recepción de mensajes RTPS y de mensajes

de descubrimiento RTPS.El submódulo de configuración es el encargado de interpretar un archivo de configuración que tiene parámetros configurables tanto del módulo DDS como del módulo RTPS, este submódulo no se encuentra especificado en el API-RTPS. A continuación se muestra en la Tabla 3.1 los submódulos definidos en el API-RTPS.

Tabla 3.1. Submódulos definidos por el API-RTPS

Submódulos	API-RTPS
Submódulo de mensaje y encapsulamiento	X
Submódulo de descubrimiento	X
Submódulo de comportamiento	X
Submódulo de configuración	-
Submódulo de transporte	-

3.2.1. API-RTPS

A partir del API proporcionado por la OMG, el cual es el API-RTPS, se muestran los diagramas de clase correspondientes a los módulos que se describen posteriormente.

El API se encuentra organizado por medio de un modelo con varios niveles lógicos, lo que permite tener un mejor control sobre el código, disminuir su complejidad y dividirlos en submódulos. Estos submódulos son: el submódulo de mensaje y encapsulación, el submódulo de descubrimiento y el submódulo de comportamiento.

El submódulo de mensaje y encapsulación es el encargado de los codificadores, del encapsulamiento y de la administración del encapsulamiento. El submódulo de descubrimiento es el encargado de los mensajes de descubrimiento y de finalizar el descubrimiento de los *participantes.* Finalmente, el submódulo de comportamiento es el encargado de la interfaz con el módulo DDS, el cual es desarrollado por el grupo de investigación a partir del API-DDS; dentro del submódulo de comportamiento se observa específicamente la interfaz hacia los *Writer* y *Reader* y el funcionamiento con estado y sin estado.

3.2.2. DISEÑO MÓDULO RTPS

Los módulos RTPS están definidos por la PIM. La PIM describe el protocolo en términos de una “máquina virtual.” La estructura de la máquina virtual está construida por clases, las cuales están descritas en el módulo de estructura, además este incluye a los *endpoints* de los *Writer* y *Reader*. Estos extremos se comunican usando los mensajes descritos en el módulo de mensajes. También es necesario describir el comportamiento de la máquina virtual, por medio del módulo de comportamiento, en el cual se observa el intercambio de mensajes que debe tomar lugar entre los extremos. Por último, se encuentra el protocolo de descubrimiento usado para configurar la máquina virtual con la información que esta necesita para comunicar a los pares remotos, este protocolo se encuentra descrito en el módulo de descubrimiento.

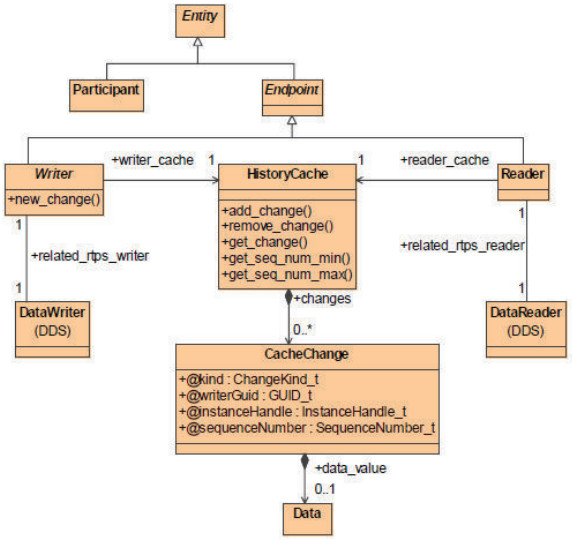


Figura 3.1. Módulo Estructura [13]

Por lo tanto el número de módulos que contiene RTPS, depende de la PIM, es decir, que está tiene especificado el propósito de cada uno de los módulos que son explicados a continuación.

3.2.2.1. Módulo Estructura

El propósito principal de este módulo es describir las clases principales que serán utilizadas por el protocolo RTPS, como se puede observar en la Figura 3.1.

Las entidades RTPS muestran los *endpoints* que serán utilizados por las entidades DDS para que se comuniquen entre sí. Cada Entidad RTPS tendrá correspondencia uno a uno con las Entidades DDS. El *HistoryCache* proporcionará

una interfaz entre las entidades DDS y su correspondiente entidad RTPS. Por ejemplo, cada operación *write* en un *DataWriter* DDS añadirá un *CacheChange* al *HistoryCache* en su correspondiente RTPS *Writer*. El RTPS *Writer* subsecuentemente transferirá el *CacheChange* al *HistoryCache* de cada *Reader* RTPS asociado. En el lado del receptor, el *DataReader* DDS será notificado por el *Reader* RTPS que un nuevo *CacheChange* ha llegado al *HistoryCache*.

Tabla 3.2. Clases y Entidades RTPS

Entidades y Clases RTPS	
Clase	**Propósito**
Entity	Es la clase base para todas las entidades RTPS. El *Entity* representa la clase de objetos que son visibles para otras entidades RTPS en la Red. Estos objetos *Entity* tienen un identificador único y global llamado *GUID* y pueden ser referenciados dentro de los mensajes RTPS.
Endpoint	Es la representación de objetos *Entity* RTPS que pueden ser extremos de comunicación. Es decir, los objetos que pueden ser orígenes o destinos de los mensajes RTPS.
Participant	Es el contenedor de todas las entidades RTPS que comparten propiedades en común y son localizadas en un espacio de dirección simple.
Writer	Es la representación de objetos *Endpoints* que puede ser origen de mensajes que comunican *CacheChanges*.
Reader	Es la representación de objetos *Endpoints* que puede ser destino de mensajes que comunican *CacheChanges*.
HistoryCache	Es la clase contenedora usada para almacenar temporalmente y gestionar grupos de cambios. En el lado del escritor este contiene la historia de los cambios en el objeto de datos hechos por el escritor. No todos los cambios deben ser conservados en memoria, por tanto existen reglas para la superposición y acumulamiento para la historia requerida, y también dependerá del estado de comunicaciones y de las políticas de QoS del DDS. En la Figura 3.2 se puede observar un diagrama de clases del *HistoryCache.*
CacheChange	Representa un cambio individual que se ha hecho al objeto de datos. Estos incluyen la creación, la modificación, y la eliminación de objetos de datos.
Data	Representa a los datos que deben ser asociados con un cambio hecho al objeto de datos.

3.2.2.1.1. Resumen de las clases usadas por la máquina virtual RTPS

Como se puede observar todas las entidades de RTPS son derivadas de la clase *Entity* del RTPS, como se muestra en la Tabla 3.2.

3.2.2.2. Módulo Mensajes y Encapsulamiento

El submódulo de mensaje y encapsulamiento es el encargado de los codificadores, del encapsulamiento y de la administración del encapsulamiento,

para lo cual el API-RTPS define una serie de clases que permiten la codificación y decodificación de los diferentes mensajes RTPS, cabeceras, e identificadores. Así como se muestra en la Figura 3.3, Figura 3.4 y Figura 3.5.

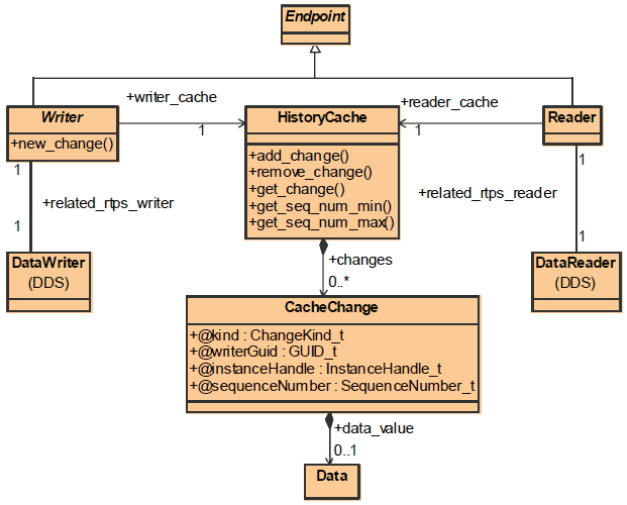


Figura 3.2. HistoryCache [13]

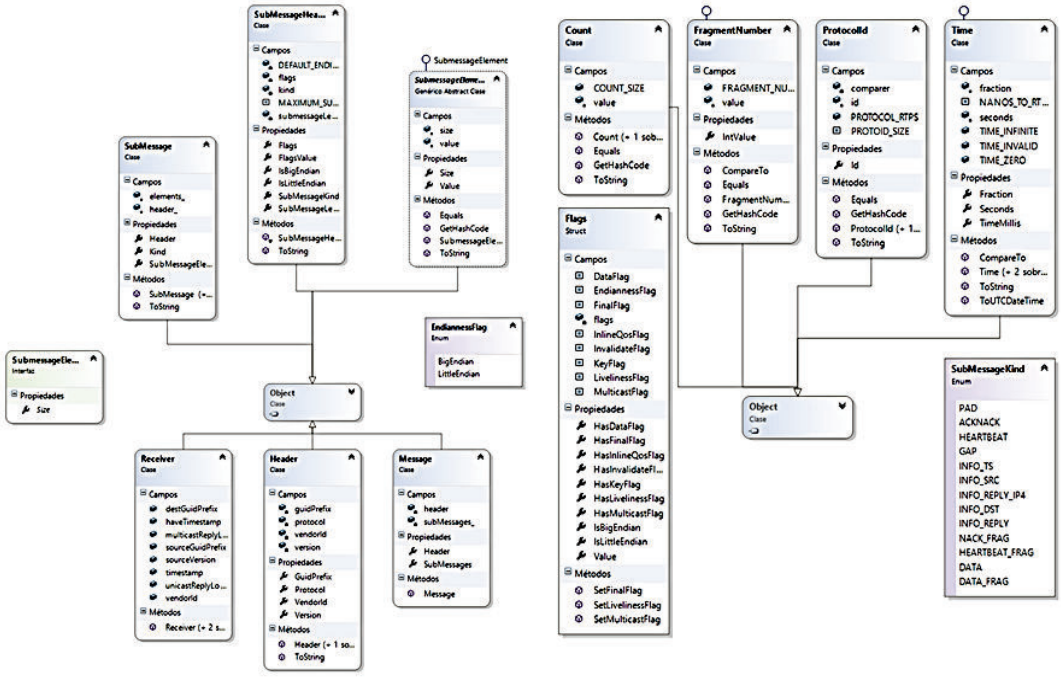


Figura 3.3. Diagrama de Clases del API-RTPS para los mensaje y el encapsulamiento I

Dentro de la Figura 3.3, se puede observar clases referentes a *Header*, *Message*, y clases que representan subcampos de cada una de estas; donde se describe claramente que atributos deben tener cada una de estas.

Dentro de la Figura 3.4, se describen las clases correspondientes a los diferentes tipos de mensaje RTPS que fueron descritos en la sección 2.2.2.1, donde se observa los atributos que cada uno de estos submensajes debe tener.

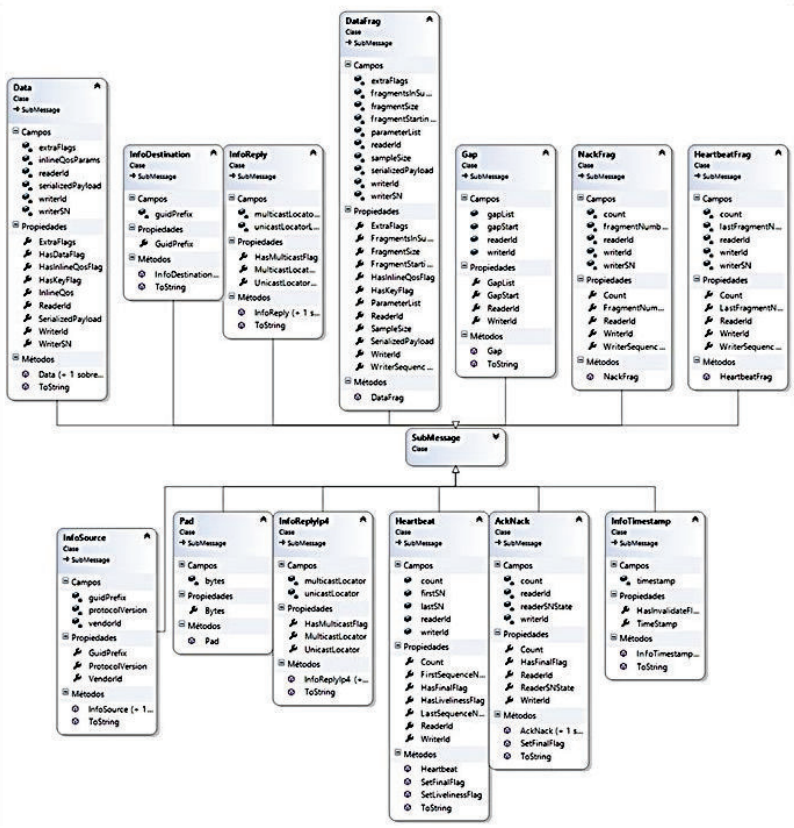


Figura 3.4. Diagrama de Clases del API-RTPS para los mensaje y el encapsulamiento II

Dentro de la Figura 3.5, se observa las clases correspondientes a los proceso de encapsulación y serialización de la información.

Este submódulo describe la estructura y contenidos lógicos globales de los mensajes que se intercambian entre los puntos finales del *Writer* RTPS y los puntos finales del *Reader* RTPS. Los mensajes RTPS son de diseño modular y se pueden ampliar fácilmente para apoyar tanto nuevas características del protocolo, así como extensiones específicas del proveedor.

3.2.2.2.1. Estructura general del mensaje RTPS

Consta de una cabecera RTPS de tamaño fijo, seguido de un número variable de Submensajes RTPS. Cada submensaje a su vez consta de un *SubmessageHeader* y un número variable de *SubmessageElements.* Esto se muestra en la Figura 3.6.

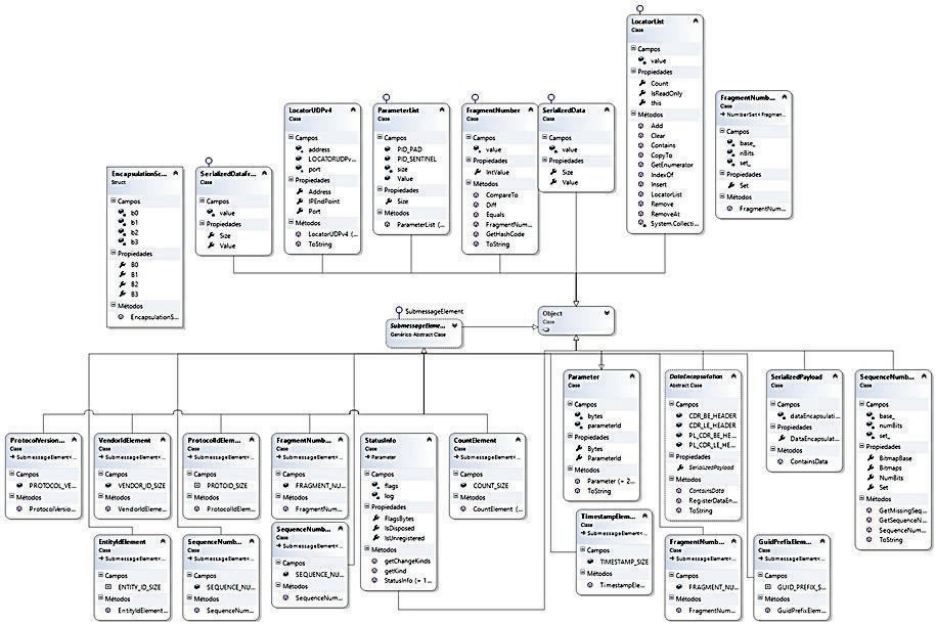


Figura 3.5. Diagrama de Clases del API-RTPS para los mensaje y el encapsulamiento III

Cada mensaje enviado por el protocolo RTPS tendrá una longitud finita. Esta longitud no se envía explícitamente por el protocolo RTPS pero es parte del transporte subyacente con la que se enviarán los mensajes RTPS. En el caso de un transporte orientado a paquetes (como UDP/ IP), la longitud del mensaje ya es proporcionada por la encapsulación del transporte.

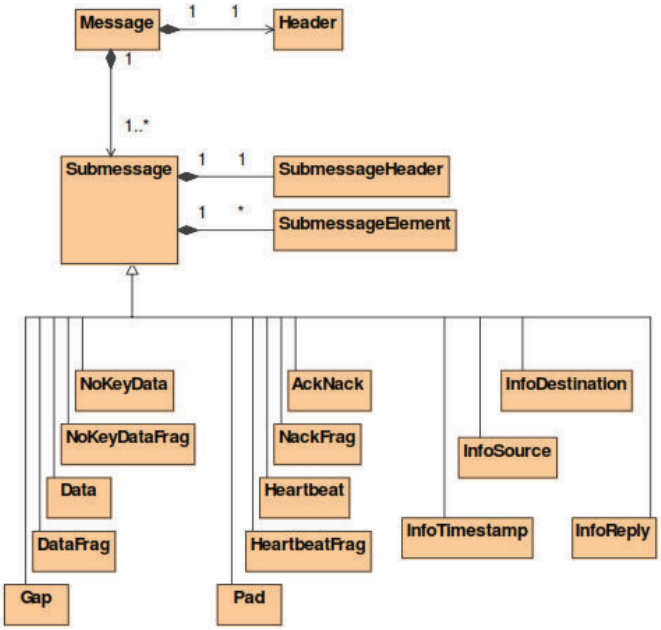


Figura 3.6. Estructura del mensaje RTPS. [13]

Estructura del Encabezado

El *header* RTPS debe aparecer al principio de cada mensaje. El *header* identifica el mensaje como perteneciente al protocolo RTPS. La cabecera identifica la versión del protocolo y el *vendor* que envío el mensaje. El *header* contiene los campos que se muestran en la Figura 3.7.

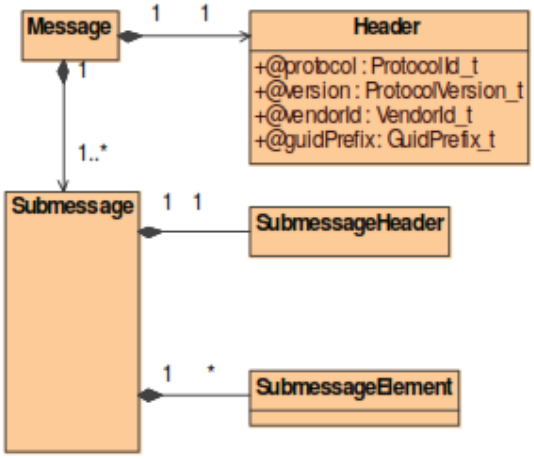


Figura 3.7. Estructura de la cabecera del mensaje RTPS. [13]

Estructura del submensaje

Cada submensaje RTPS consistirá de un número variable de partes del submensaje RTPS. Todos los submensajes RTPS cuentan con la misma estructura idéntica como se muestra en la Figura 3.8.

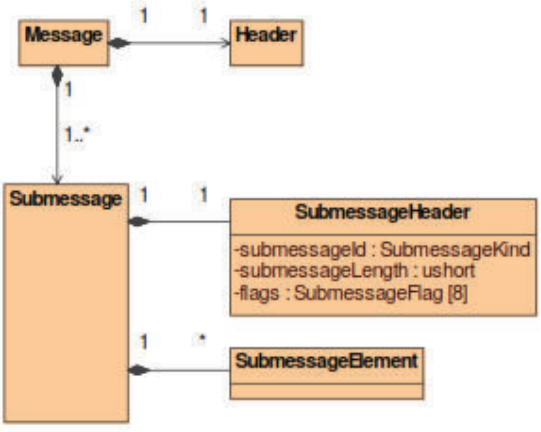


Figura 3.8. Estructura de los submensajes RTPS. [13]

Todos los submensajes empiezan con un *SubmessageHeader* seguido por un *SubmessageElement*. El *header* del submensaje identifica el tipo de mensajes y los elementos opcionales dentro del mismo.

SubmessageId

El *SubmessageId* identifica el tipo del submensaje. A fin de mantener la interoperabilidad con versiones futuras, la plataforma de asignaciones específicas debe reservar un rango de valores destinados a extensiones de protocolo y un rango de valores que son reservados por el *vendor* de submensajes específicos que nunca serán utilizados por futuras versiones del protocolo RTPS.

Flags

Las *Flags* en la cabecera del submensaje contienen ocho valores booleanos. La primera bandera, el *EndiannessFlag,* está presente y se encuentra en la misma posición en todos los submensajes y representa el orden de bits utilizados para codificar la información en el submensaje.

SubmessageLength

El *SubmessageLength* indica la longitud del submensaje.

Cuando el *SubmessageLength* es mayor a 0, significa que:

- La longitud se mide desde el comienzo de los contenidos del submensaje hasta el comienzo de la cabecera del siguiente submensaje.
- Es la longitud del mensaje restante.

Un intérprete del mensaje puede distinguir entre estos dos casos, ya que conoce la longitud total del mensaje.

Cuando el *SubmessageLength* es igual 0, el submensaje es el último en el mensaje y se extiende hasta el final del mensaje. Esto hace que sea posible enviar submensajes mayores a 64 KB (longitud máxima que se puede almacenar en el campo *submessageLength*), siempre que sean el último submensaje en el mensaje.

3.2.2.2.2. RTPS Message Receiver

El receptor de un mensaje deberá mantener el estado de los submensajes deserealizados previamente en el mismo mensaje. Este estado se modela como el estado de un receptor RTPS que se reestablece cada vez que un nuevo mensaje se procesa y proporciona un contexto para la interpretación de cada submensaje. El receptor RTPS se muestra en la Figura 3.9.

Reglas seguidas por el Receptor del mensaje

El siguiente algoritmo, el cual esta propuesto en la OMG describe las reglas que un receptor de cualquier mensaje debe seguir:

- Si el encabezado del submensaje no se puede leer, el resto del mensaje se considera no válido.
- El campo *submessageLength* define dónde comienza el siguiente submensaje o indica que el submensaje se extiende hasta el final del mensaje. Si este campo no es válido, el resto del mensaje no es válido.
- Un submensaje con un *SubmessageId* desconocido debe ser ignorado y el análisis debe continuar con el siguiente submensaje.

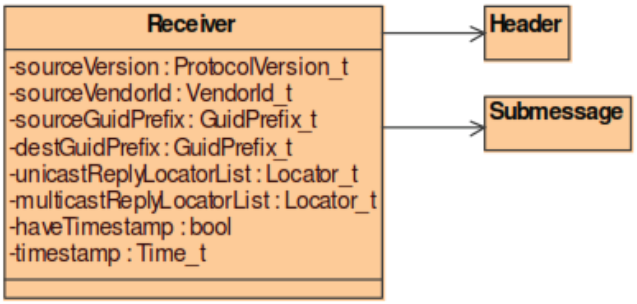


Figura 3.9. Receptor RTPS. [13]

- En las banderas del submensaje, el receptor de un submensaje debe ignorar banderas desconocidas.
- Un campo *submessageLength* válido siempre debe ser utilizado para encontrar el siguiente submensaje, incluso para submensajes con ID conocidos.
- Un submensaje conocido, pero no válido, invalida al resto del mensaje.

La recepción de una cabecera válida y/ o submensaje tiene dos efectos:

- Se puede cambiar el estado del receptor; este estado influye en cómo se interpretan los siguientes submensajes en el mensaje. En esta versión del protocolo, sólo la cabecera y los submensajes *InfoSource*, *InfoReply*, *InfoDestination* e *InfoTimestamp* cambian el estado del receptor.
- Se puede afectar el comportamiento del punto final al que está destinado el mensaje. Esto se aplica a los mensajes básicos RTPS, tales como: *Data, DataFrag, HeartBeat, AckNack, GAP, HeartbeatFrag, NackFrag*.

3.2.2.2.3. Elementos del submensaje RTPS

Cada mensaje RTPS contiene un número variable de submensajes RTPS. Cada submensaje RTPS a su vez, se construye a partir de un conjunto de bloques

llamados *SubmessageElements*. El RTPS versión 2.2 define los siguientes elementos: submensaje *GuidPrefix, entityId, sequenceNumber, SequenceNumberSet, FragmentNumber, FragmentNumberSet, VendorID, ProtocolVersion, LocatorList, TimeStamp, Count, SerializedData* y *ParameterList*. A continuación se muestra los elementos del submensaje RTPS en la Figura 3.10.

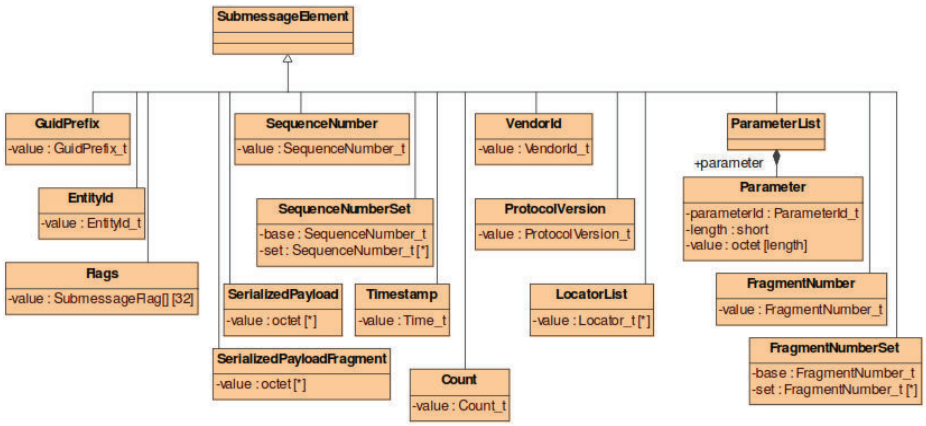


Figura 3.10. Elementos de submensaje RTPS.

El detalle de los elementos del submensaje RTPS se encuentra en la sección del Módulo Mensajes de la norma [13].

3.2.2.2.4. Submensaje RTPS

El protocolo RTPS de la versión 2.2 define varios tipos de submensajes. Se clasifican en dos grupos: EntitySubmessages e Interpreter-Submessages.

El *Entity submessage* se dirige a una Entidad RTPS. El *Interpreter-Submessages* modifica el estado del receptor RTPS y proporcionará el contexto que ayuda a los procesos posteriores del *Entity submessage*.

Las entidades del submensaje son:

- El submensaje *Data*, contiene información sobre el valor de un objeto fecha de la aplicación. Los submensajes de datos son enviados por el *Writer* a un *Reader*.
- El *DataFrag*, equivale a los datos, pero sólo contiene una parte del valor (uno o más fragmentos). Permite que los datos se transmitan como varios fragmentos para superar las limitaciones de tamaño de mensajes de transporte.
- El *HeartBeat*, describe la información que está disponible en un *Writer*. Los mensajes *HeartBeat* son enviados por un *Writer* a uno o más *Reader*.

- El *HeartbeatFrag*, sirve para los datos fragmentados, describe que fragmentos están disponibles en un *Writer*. Los submensajes *HeartbeatFrag* son enviados por un *Writer* a uno o más *Reader*.
- El *GAP*, describe la información que ya no es relevante para el *Reader*. Los mensajes *GAP* son enviados por un *Writer* a uno o más *Reader*.
- El *AckNack*, proporciona información sobre el estado de un *Reader* a un *Writer*. Los mensajes *AckNack* son enviados por un *Reader* a uno o más *Writer*.
- El *NackFrag,* proporciona información sobre el estado de un *Reader* a un *Writer*, específicamente los fragmentos de información que siguen perdidos en el *Reader*. Los submensajes *NackFrag* son enviados por un *Reader* a uno o más *Writer*.

Los submensajes de interpretación son:

- El *InfoSource,* proporciona información acerca de la fuente de donde se originaron los Entity Submessage posteriores. Este submensaje se utiliza principalmente para la retransmisión de los submensajes RTPS.
- El *InfoDestination*, proporciona información sobre el destino final de los submensajes que le acompañan. Este submensaje se utiliza principalmente para la retransmisión de submensajes RTPS.
- El *InfoReply*, proporciona información sobre donde responder a las entidades que figuran en submensajes posteriores.
- El *InfoTimestamp*, proporciona una marca de tiempo a los submensajes que le acompañan.
- El *Pad,* se utiliza para agregar relleno a un mensaje, siempre y cuando sea necesaria la alineación de la memoria.

A continuación se muestran los diferentes submensajes RTPS en la Figura 3.11.

El detalle de los tipos de submensajes RTPS se encuentra en la sección del Módulo Mensajes de la norma [13].

3.2.2.3. Módulo Comportamiento

El submódulo de comportamiento es el encargado de la interfaz con el módulo DDS, de los *Writer* y *Reader* y del funcionamiento con estado y sin estado, para lo cual el API-RTPS define una serie de clases que permiten mostrar el

comportamiento en la comunicación. A continuación se presenta en la Figura 3.12 el diagrama de clases del submódulo comportamiento.

Una ver que el *Writer* RTPS sea asociado a un *Reader* RTPS, es responsabilidad de ambos, asegurarse que los cambios en el *CacheChange* que existen en la *HistoryCache* de los diferentes *Writers* sean propagados a la *HistoryCache* de los diferentes *Readers*.

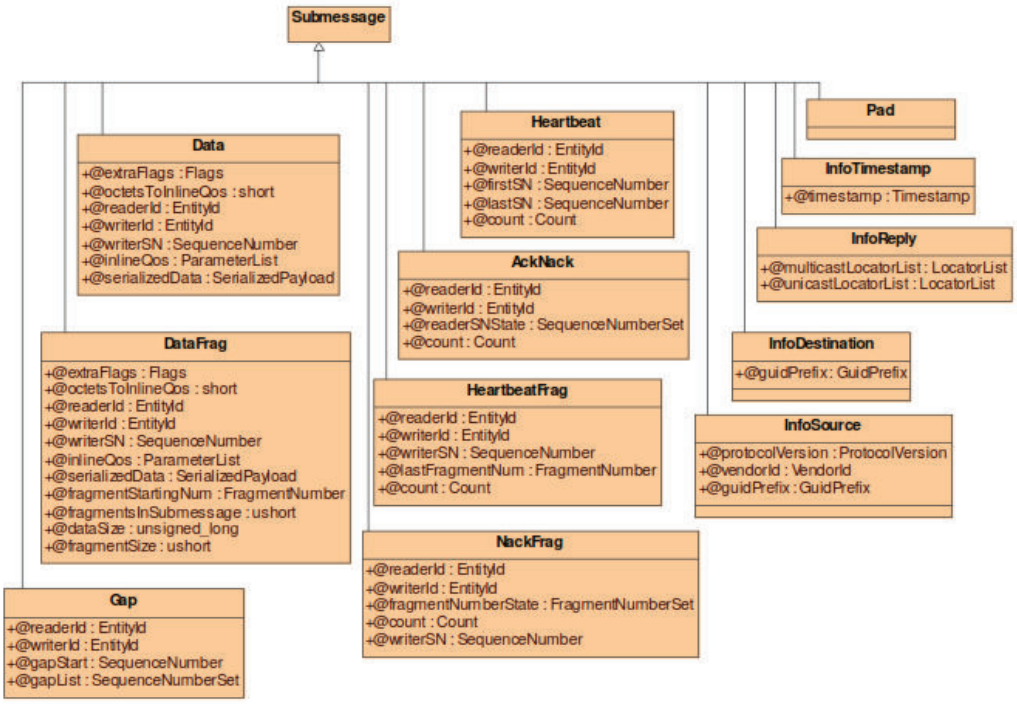


Figura 3.11. Submensajes RTPS.

Este módulo describe como los pares de *Writer y Reader* RTPS asociados deben comportarse para propagar los cambios en el *CacheChange*. Este comportamiento está definido en términos de los mensajes intercambiados usando a los mensajes RTPS que ya fueron descritos en el punto anterior.

3.2.2.3.1. Requerimientos Generales dentro del Módulo Comportamiento proporcionados por la OMG

Los siguientes requerimientos aplican a todas las entidades RTPS.

- Todas la comunicaciones deberán tomar lugar usando mensajes RTPS, es decir, que ningún otro mensaje que no está definido en los mensajes RTPS puede ser usado.
- Se debe implementar un *Message Receiver* RTPS, es decir, que para interpretar a los submensajes RTPS se deberá usar esta implementación.

- Las características de tiempos en todas las implementaciones deben ser configurables.
- Se debe implementar el protocolo de descubrimiento denominado *Simple Participant and Endpoint Discovery Protocols*, es decir, a los protocolos de descubrimientos que cubre el estándar.

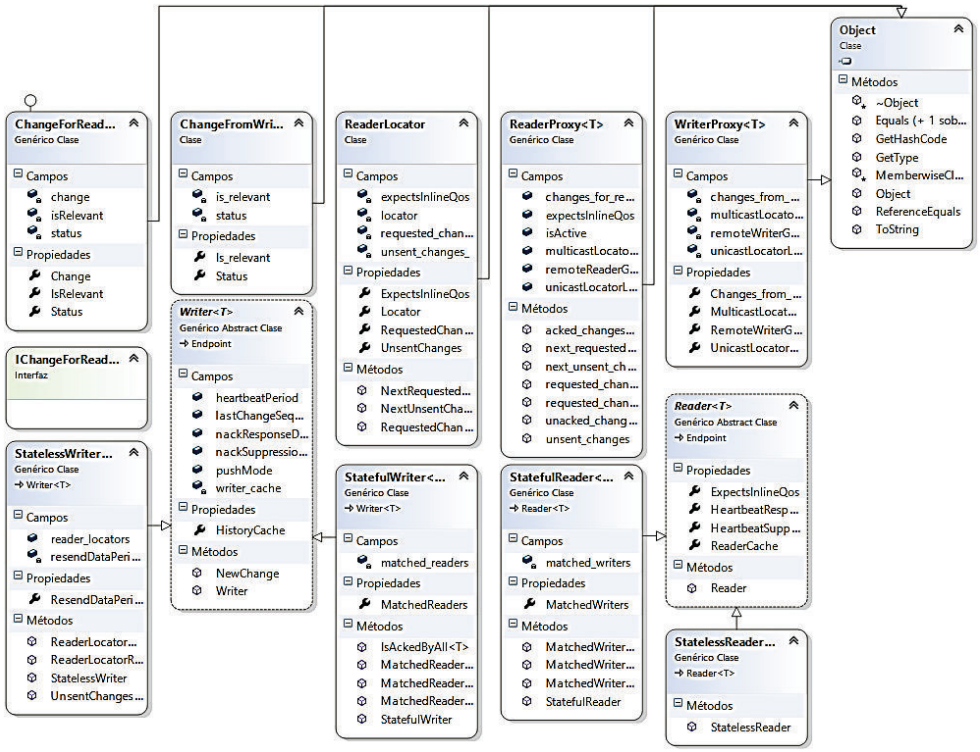


Figura 3.12. Diagrama de Clases del API-RTPS para el comportamiento

3.2.2.3.2. Comportamiento requerido de los Writer RTPS proporcionado por la OMG

- Los *Writer* no deben enviar datos fuera de orden, es decir, que estos deben enviar datos en el mismo orden en el que fueron añadidos en la *HistoryCache*.
- Los *Writer* deben incluir valores *in-line QoS* si es requerido por un *Reader*, es decir, que un *Writer* debe respetar las solicitudes de los *Reader* para recibir mensajes de datos con QoS.
- Los *Writer* deben enviar mensajes *Heartbeat* periódicamente cuando se trabaja en modo confiable, un escritor debe periódicamente informar a cada lector asociado de su disponibilidad de datos, enviando *HeartBeat* periódicos que incluyen el número de secuencia del dato disponible. Si no hay datos disponibles, ningún *Heartbeat* debe ser enviado. Para comunicaciones

estrictamente confiables, los escritores deben continuar enviando mensajes Heartbeat a los lectores hasta que los lectores hayan confirmado la recepción de los datos o hayan desparecido.

- Los *Writers* deben eventualmente responder a los acuses de recibo negativos; un acuse de recibo negativo indica que parte de la información se ha perdido, el escritor debe responder también enviando nuevamente los datos perdidos, enviando un mensaje GAP cuando esta información no es relevante, o enviando un mensaje Heartbeat cuando esta información ya no está disponible.

3.2.2.3.3. Comportamiento requerido de los Reader RTPS proporcionado por la OMG

- Los *Reader* deben responder eventualmente después de recibir un mensaje *Heartbeat* con bandera *final* no establecida con un mensaje *AckNack*, este mensaje debe acusar el recibo de información cuando toda la información ha sido recibida o también podría indicar que algunos datos se han perdido. Además, esta respuesta debe ser retardada para evitar ráfagas de mensajes.
- Los *Reader* deben responder eventualmente después de recibir *heartbeats* los cuales indican que un dato se ha perdido, hasta recibir un mensaje *Heartbeat*; un lector que está perdiendo información debe responder con un mensaje *AckNack* indicando que información ha perdido. Este requerimiento solamente es aplicado si el lector puede acomodar los datos perdidos en su caché y es independiente de la configuración de la bandera final del submensaje *Heartbeat*.
- Una vez acusado positivamente un mensaje, no se puede acusar negativamente el mismo mensaje.
- Los *Reader* solamente pueden enviar mensajes *AckNack* en respuesta a los mensajes *Heartbeat*.

3.2.2.3.4. Implementación del Protocolo RTPS

La especificación RTPS establece que una implementación funcional del protocolo debe solamente satisfacer los requerimientos presentados en los dos puntos anteriores. Sin embargo, existen dos implementaciones definidas por el módulo de comportamiento.

- **Implementación sin estado**, la cual esta optimizada para escalabilidad. Esta mantiene virtualmente un no estado en las entidades remotas y por lo

tanto escala sin gran problema en sistemas grandes. Además, esto implica una escalabilidad mejorada y una disminución en el uso de la memoria, pero un ancho de banda adicional. La implementación sin estado es ideal para comunicaciones en modo *best-effort* sobre multicast.

- **Implementación con estado**, la cual mantiene el estado de las entidades remotas. Esta minimiza el uso del ancho de banda, pero requiere más memoria y tiene una reducida escalabilidad. También esta garantiza estrictamente comunicaciones confiables y puede aplicar políticas de QoS.

Tabla 3.3. Combinación de atributos posibles en lectores asociados con escritores [13]

Writer properties	Reader properties	Combination name
topicKind= WITH_KEY reliabilityLevel=BEST_EFFORT or reliabilityLevel= RELIABLE	topicKind= WITH_KEY reliabilityLevel=BEST_EFFORT	WITH_KEY Best-Effort
topicKind= NO_KEY reliabilityLevel=BEST_EFFORT or reliabilityLevel=RELIABLE	topicKind=NO_KEY reliabilityLevel=BEST_EFFORT	NO_KEY Best-Effort
topicKind=WITH_KEY reliabilityLevel=RELIABLE	topicKind=WITH_KEY reliabilityLevel=RELIABLE	WITH_KEY Reliable
topicKind=NO_KEY reliabilityLevel=RELIABLE	topicKind=NO_KEY reliabilityLevel=RELIABLE	NO_KEY Reliable

3.2.2.3.5. Comportamiento de un Writer respecto a Reader asociados proporcionado por la OMG

El comportamiento de un escritor RTPS con respecto a sus lectores asociados depende de:

- La configuración del nivel de confiabilidad del escritor y el lector.
- La configuración del tipo de topic usado en el lector y escritor. Es decir controla si los datos que están siendo comunicados corresponden a un topic DDS con una clave definida

No todas las combinaciones de niveles de confiabilidad son posibles con el tipo de topic. En la Tabla 3.3 se muestran las combinaciones posibles.

3.2.2.3.6. Comportamiento de Writer sin estado proporcionado por la OMG

Comportamiento de Writer sin estado con mejor esfuerzo

En la siguiente Figura 3.13 se puede observar el comportamiento de este tipo de escritor.

El listado de estados se encuentra descrito en la Tabla 3.4.

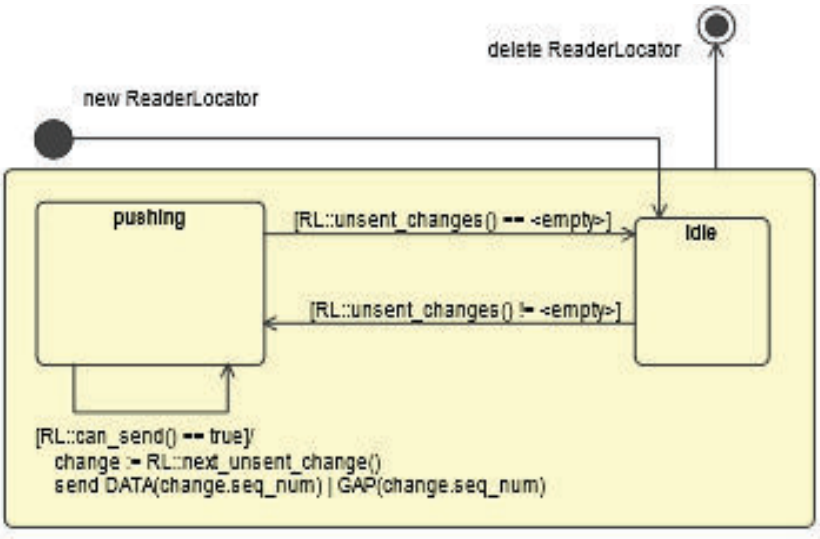


Figura 3.13. Comportamiento de un Writer sin estado con WITH_KEY Best-Effort con respecto a cada ReaderLocator [13]

Tabla 3.4. Transiciones del comportamiento en mejor esfuerzo de un Writer sin estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El escritor RTPS es configurado con un *ReaderLocator*.	*Idle*
T2	*Idle*	Se indica que hay algunos cambios en el *HistoryCache* del escritor que aún no han sido enviados al *ReaderLocator*.	*Pushing*
T3	*Pushing*	Se indica que todos los cambios en el *HistoryCache* del escritor han sido enviados al *ReaderLocator*.	*Idle*
T4	*Pushing*	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al *ReaderLocator*.	*Pushing*
T5	*Any state*	El escritor RTPS es configurado para no mantener más al *ReaderLocator*.	*Final*

Comportamiento de Writer sin estado confiable

En la siguiente Figura 3.14 se puede observar el comportamiento de este tipo de escritor.

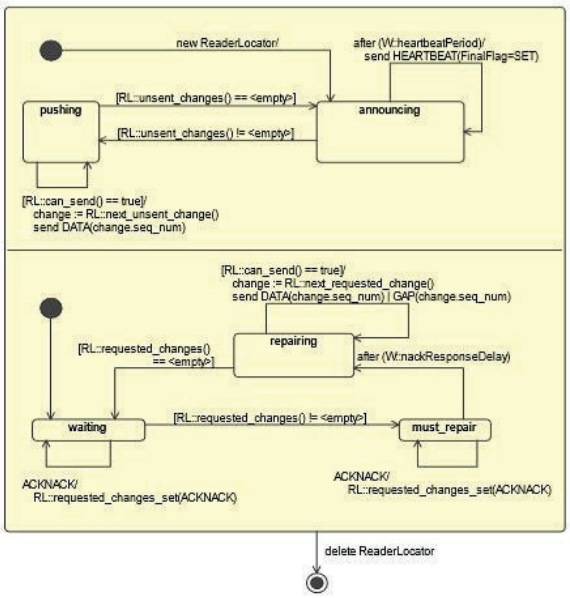


Figura 3.14. Comportamiento de un Writer sin estado con WITH_KEY Reliable con respecto a cada ReaderLocator [13]

El listado de estados se encuentra descrito en la Tabla 3.5.

Tabla 3.5. Transiciones del comportamiento en confiable de un Writer sin estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El escritor RTPS es configurado con un *ReaderLocator*.	*Announcing*
T2	*Announcing*	Se indica que hay algunos cambios en el *HistoryCache* del escritor que no han sido enviados al *ReaderLocator*.	*Pushing*
T3	*Pushing*	Se indica que todos los cambios del *HistoryCache* del *Writer* han sido enviados al *ReaderLocator*.	*Announcing*
T4	*Pushing*	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al *ReaderLocator*.	*Pushing*
T5	*Announcing*	Se busca enviar con un temporizador periódico cada Heartbeat.	*Announcing*
T6	*Waiting*	Se recepta ACKNACK que han sido destinados al escritor sin estado	*Waiting*
T7	*Waiting*	Se indica que hay cambios que han sido solicitados por algún lector RTPS alcanzable hacia el *ReaderLocator*.	*Must_repair*
T8	*Must_repair*	Se recepta ACKNACK que han sido destinados al escritor sin estado	*Must_repair*
T9	*Must_repair*	Se busca enviar con un temporizador que la duración del ACKNACK ha caducado mientras se ha entrado a este modo.	*Repairing*

Tabla 3.5. Transiciones del comportamiento en confiable de un Writer sin estado con respecto a cada ReaderLocator

T10	*Repairing*	Se indica que el escritor RTPS tiene los recursos necesitados para enviar un cambio al *ReaderLocator*.	*Repairing*
T11	*Repairing*	Se indica que no hay más cambios solicitados por un lector alcanzable para el *ReaderLocator*.	*Waiting*
T12	*Any state*	El escritor RTPS es configurado para no mantener más al *ReaderLocator*.	*Final*

3.2.2.3.7. Comportamiento de Writer con estado proporcionado por la OMG

Comportamiento de Writer con estado con mejor esfuerzo

En la siguiente Figura 3.15 se puede observar el comportamiento de este tipo de escritor.

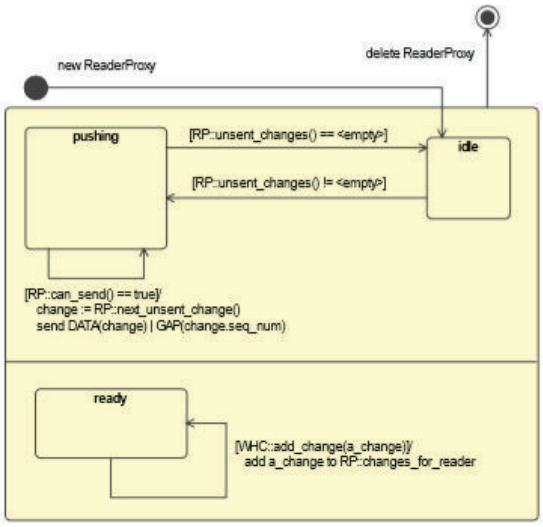


Figura 3.15. Comportamiento de un Writer con estado con WITH_KEY Best-Effort con respecto a cada ReaderLocator [13]

El listado de estados se encuentra descrito en la Tabla 3.6.

Tabla 3.6. Transiciones del comportamiento en mejor esfuerzo de un Writer con estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El escritor RTPS es asociado con un *Reader*.	*Idle*
T2	*Idle*	Se indica que hay algunos cambios en el *HistoryCache* del escritor que aún no han sido enviados al *Reader* representado por el *ReaderProxy*.	*Pushing*

Tabla 3.6. Transiciones del comportamiento en mejor esfuerzo de un Writer con estado con respecto a cada ReaderLocator

T3	*Pushing*	Se indica que todos los cambios en el *HistoryCache* del escritor han sido enviados al *Reader* representado por el *ReaderProxy*.	*Idle*
T4	*Pushing*	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al *Reader* representado por el *ReaderProxy*.	*Pushing*
T5	*Ready*	Un nuevo cambio fue añadido al *HistoryCache* del *Writer*.	*Ready*
T6	*Any state*	El escritor RTPS es configurado para no mantener más al *Reader* representado por el *ReaderProxy*.	*Final*

Comportamiento de Writer con estado confiable

En la siguiente Figura 3.16 se puede observar el comportamiento de este tipo de escritor.

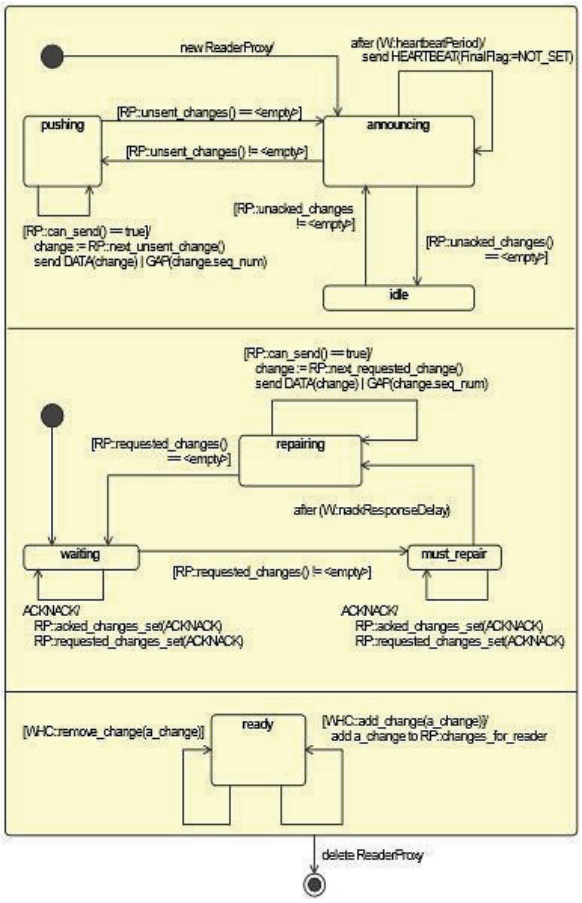


Figura 3.16. Comportamiento de un Writer con estado con WITH_KEY Reliable con respecto a cada ReaderLocator [13]

El listado de estados se encuentra descrito en la Tabla 3.7.

Tabla 3.7. Transiciones del comportamiento en confiable de un Writer con estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El escritor RTPS es asociado con un *Reader*.	*Announcing*
T2	*Announcing*	Se indica que hay algunos cambios en el *HistoryCache* del escritor que no han sido enviados al *Reader* representado por un *ReaderProxy*.	*Pushing*
T3	*Pushing*	Se indica que todos los cambios del *HistoryCache* del *Writer* han sido enviados al *Reader* representado por el *ReaderProxy*.	*Announcing*
T4	*Pushing*	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al *Reader* representado por el *ReaderProxy*.	*Pushing*
T5	*Announcing*	Se indica que todos los cambios en el *HistoryCache* del *Writer* han sido confirmados por el *Reader* representado por el *ReaderProxy*.	*Idle*
T6	*Idle*	Se indica que hay cambios en el *HistoryCache* en el *Writer* que no han sido confirmados por el *Reader* representado por el *ReaderProxy*.	*Announcing*
T7	*Announcing*	Se busca enviar con un temporizador periódico cada Heartbeat.	*Announcing*
T8	*Waiting*	Se recepta ACKNACK que han sido destinados al escritor con estado	*Waiting*
T9	*Waiting*	Se indica que hay cambios que han sido solicitados por algún lector RTPS alcanzable hacia el *Reader* representado por el *ReaderProxy*.	*Must_repair*
T10	*Must_repair*	Se recepta ACKNACK que han sido destinados al escritor con estado	*Must_repair*
T11	*Must_repair*	Se busca enviar con un temporizador que la duración del ACKNACK ha caducado mientras se ha entrado a este modo.	*Repairing*
T12	*Repairing*	Se indica que el escritor RTPS tiene los recursos necesitados para enviar un cambio al *Reader* representado por el *ReaderProxy*.	*Repairing*
T13	*Repairing*	Se indica que no hay más cambios solicitados por un lector alcanzable para el *Reader* representado por el *ReaderProxy*.	*Waiting*
T14	*Ready*	Se añade un nuevo *CacheChange* al *HistoryCache* del *Writer* correspondiente al DDS *Writer*.	*Ready*
T15	*Ready*	Se remueve un *CacheChange* al *HistoryCache* del *Writer* correspondiente al DDS *Writer*.	*Ready*
T16	*Any state*	El escritor RTPS es configurado para no mantener más al *Reader* representado por el *ReaderProxy*.	*Final*

3.2.2.3.8. Comportamiento de Reader sin estado proporcionado por la OMG

Comportamiento de Reader sin estado con mejor esfuerzo

En la siguiente Figura 3.17 se puede observar el comportamiento de este tipo de lector.

El listado de estados se encuentra descrito en la Tabla 3.8.

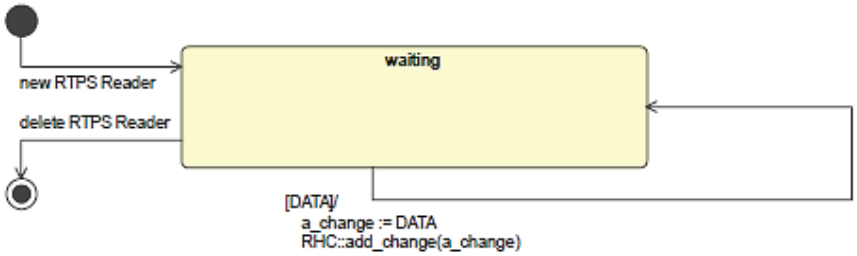


Figura 3.17. Comportamiento de un Reader sin estado con WITH_KEY Best-Effort [13]

Tabla 3.8. Transiciones del comportamiento en mejor esfuerzo de un Reader sin estado

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El lector RTPS es creado	*Waiting*
T2	*Waiting*	El mensaje DATA es recibido	*Waiting*
T3	*Waiting*	El lector RTPS es borrado.	*Final*

Comportamiento de Reader sin estado confiable

Esta combinación no es soportada por el protocolo RTPS, es decir, que para tener una implementación confiable, el lector RTPS debe mantener algún estado.

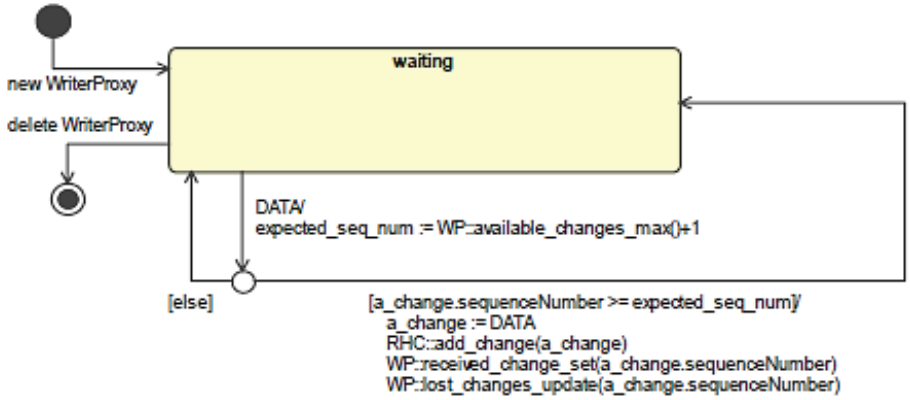


Figura 3.18. Comportamiento de un Reader con estado con WITH_KEY Best-Effort con respecto a cada Writer asociado [13]

3.2.2.3.9. Comportamiento de Reader con estado proporcionado por la OMG

Comportamiento de Reader con estado con mejor esfuerzo

En la siguiente Figura 3.18 se puede observar el comportamiento de este tipo de lector.

El listado de estados se encuentra descrito en la Tabla 3.9.

Tabla 3.9. Transiciones del comportamiento en mejor esfuerzo de un Reader con estado con respecto a cada Writer asociado

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El lector RTPS es configurado con su escritor asociado.	*Waiting*
T2	*Waiting*	El mensaje DATA es recibido desde el escritor asociado.	*Waiting*
T3	*Waiting*	El lector RTPS es configurado para no estar más asociado con el escritor.	*Final*

Comportamiento de Reader con estado con mejor esfuerzo

En la siguiente Figura 3.19 podremos observar el comportamiento de este tipo de lector.

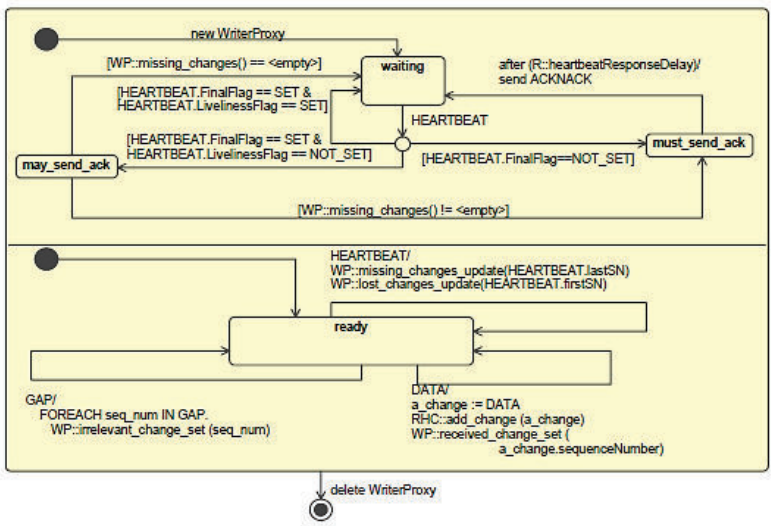


Figura 3.19. Comportamiento de un Reader con estado con WITH_KEY Reliable con respecto a cada Writer asociado [13]

El listado de estados se encuentra descrito en la Tabla 3.10.

Tabla 3.10. Transiciones del comportamiento en confiable de un Reader con estado con respecto a su Writer asociado

Transición	Estado	Evento	Siguiente Estado
T1	*Initial*	El lector RTPS es configurado con un escritor asociado.	*Waiting*
T2	*Waiting*	El mensaje HEARTBEAT es recibido.	*Si la finalflag está en 0 Must_send_ack*

Tabla 3.10. Transiciones del comportamiento en confiable de un Reader con estado con respecto a su Writer asociado

			Si la livelinessflag está en 0 May_send_ack Sino Waiting
T3	*May_send_ack*	Se indica que todos los cambios conocidos a estar en el *HistoryCache* del *Writer* representado por el *WriterProxy* han sido recibidos por el *Reader*.	*Waiting*
T4	*May_send_ack*	Se indica que hay algunos cambios conocidos a estar en el *HistoryCache* del *Writer* representado por el *WriterProxy*, que no han sido recibidos por el *Reader*.	*Must_send_ack*
T5	*Must_send_ack*	Se busca enviar con un temporizador periódico cada Heartbeat.	*Waiting*
T6	*Initial2*	Similar a la transición 1.	*Ready*
T7	*Ready*	Se recepta un mensaje HEARTBEAT que ha sido destinado al lector con estado.	*Ready*
T8	*Ready*	Se recepta un mensaje DATA destinado al lector con estado.	*Ready*
T9	*Ready*	Se recepta un mensaje GAP destinado al lector con estado .	*Ready*
T10	*Any state*	El *Reader* RTPS no se encuentra más asociado con el *Writer* RTPS representado por el *WriterProxy*.	*Final*

3.2.2.4. Módulo Descubrimiento

El submódulo de descubrimiento es el encargado de los mensajes de descubrimiento y de finalizar el descubrimiento de los participantes, para lo cual el API-RTPS define una serie de clases que permiten el descubrimiento. A continuación en la Figura 3.20 se muestra el diagrama de clase basado en el API-RTPS del submódulo de descubrimiento.

Dentro de la Figura 3.20, se observa clases como SPDPbuiltinParticipan, las cuales corresponden al protocolo de descubrimiento que se encuentra explicado en la sección 3.2.2.4

El módulo descubrimiento definirá el protocolo de descubrimiento RTPS. El propósito del protocolo de descubrimiento permitirá que cada *participante* RTPS descubra otros participantes relevantes y sus *endpoint*. Una vez que el *endpoint* ha

sido descubierto, las implementaciones podrán configurar *endpoint* locales para establecer la comunicación.

La especificación RTPS divide al protocolo de descubrimiento en dos protocolos independientes:

- *Participant* Discovery Protocol (PDP)
- *Endpoint* Discovery Protocol (EDP)

Un PDP especifica como los participantes se descubren entre sí en la red. Una vez que dos *participantes* se han descubierto, intercambian información sobre los *endpoint* que los contienen utilizando un EDP. Aparte de esta relación de causalidad, ambos protocolos se pueden considerar independientes.

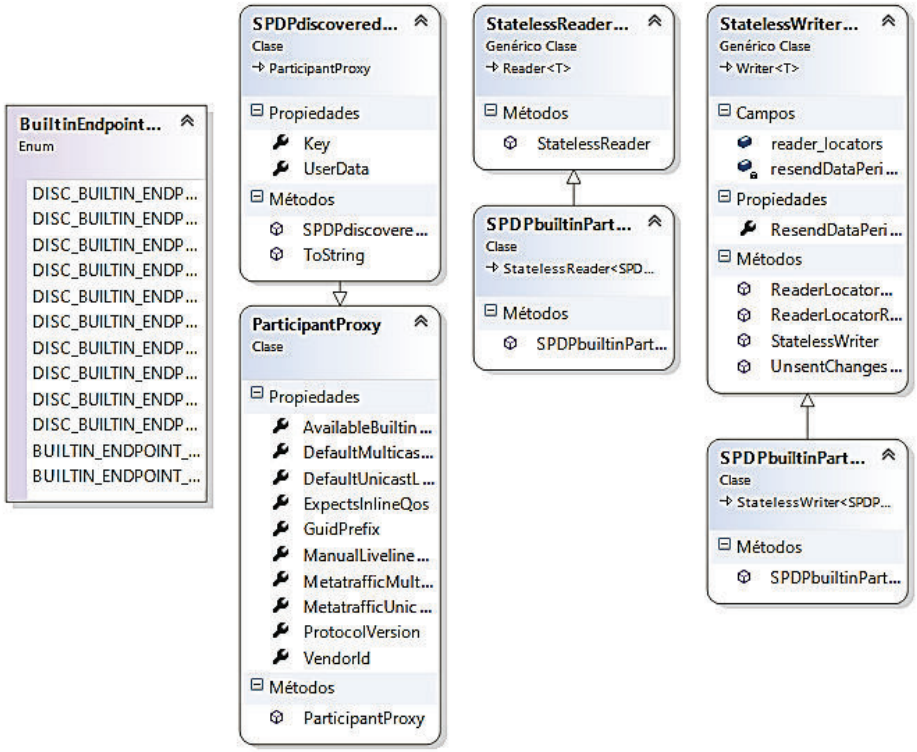


Figura 3.20. Diagrama de Clases del API-RTPS para el descubrimiento

A fin de la interoperabilidad, todas las implementaciones RTPS deben proporcionar al menos los siguientes protocolos de descubrimiento:

- Simple Participant Discovery Protocol (SPDP)
- Simple Endpoint Discovery Protocol (SEDP)

Ambos son protocolos básicos de descubrimiento que bastan para pequeñas redes de mediana escala. Los PDP adicionales y EDP que están orientados hacia las redes más grandes se pueden añadir a las futuras versiones de la especificación.

Finalmente, el rol de un protocolo de descubrimiento será proporcionar información sobre *endpoint* remotos descubiertos.

3.2.2.4.1. Simple Participant Discovery Protocol

El propósito de este protocolo será descubrir la presencia de otros participantes en la red y sus propiedades.

Un participante puede soportar varios PDP, pero para el propósito de interoperabilidad, todas las implementaciones deberían soportar al menos SPDP.

El RTPS SPDP utiliza un enfoque simple para anunciar y detectar la presencia de *participantes* en un dominio.

3.2.2.4.2. Simple Endpoint Discovery Protocol

Un EDP define la información intercambiada requerida entre dos *participantes* para descubrir *Writer* y *Reader Endpoint.*

Un participante puede soportar varios EDP, pero para el propósito de interoperabilidad, todas las implementaciones soportarían para el caso de *Simple Endpoint Discovery Protocol*.

3.2.2.4.3. Apoyos alternativos para Protocolos de Descubrimiento

Los requisitos sobre el Participante y Protocolos de Descubrimiento *Endpoint* pueden variar en función del escenario de implementación.

Por ejemplo, un protocolo optimizado para la velocidad y simplicidad (como un protocolo que sea desplegado en dispositivos de una LAN) no pueden escalar bien a los grandes sistemas en un entorno WAN.

Por esta razón, la especificación RTPS permite implementaciones que soportan múltiples PDP y EDP. Hay muchos enfoques posibles para la implementación de un protocolo de descubrimiento que incluye el uso de descubrimiento estático, el descubrimiento de archivos, servicio de consulta central, etc. El único requisito impuesto por RTPS con el propósito de interoperabilidad es que todas las implementaciones RTPS soporten al menos el SPDP y SEDP.

Si una aplicación soporta múltiples PDP, cada PDP puede ser inicializado de manera diferente y descubrir un conjunto diferente de los participantes remotos. Los *participantes* remotos mediante la implementación RTPS de un *vendor* diferente deben ser contactados usando al menos el SPDP para garantizar la interoperabilidad. No existe tal requisito cuando el participante remoto utiliza la misma implementación RTPS.

Incluso cuando el SPDP es utilizado por todos los participantes, los participantes remotos pueden todavía utilizar diferentes EDP. Los EDP soportan participantes que incluye en la información intercambiada por el SPDP. Todos los participantes deben soportar al menos el SEDP, por lo que siempre tienen al menos un EDP en común. Sin embargo, si dos participantes apoyan a otro EDP, este protocolo alternativo puede ser utilizado en su lugar.

3.2.3. ADAPTACIÓN DEL API-RTPS PARA INTERACTUAR CON DDS

A partir de la sección 3.2.1, se procedió con el diseño y adaptación de todos los submódulos, por medio de la creación de las clases y métodos necesarios para que RTPS interactúe con DDS.

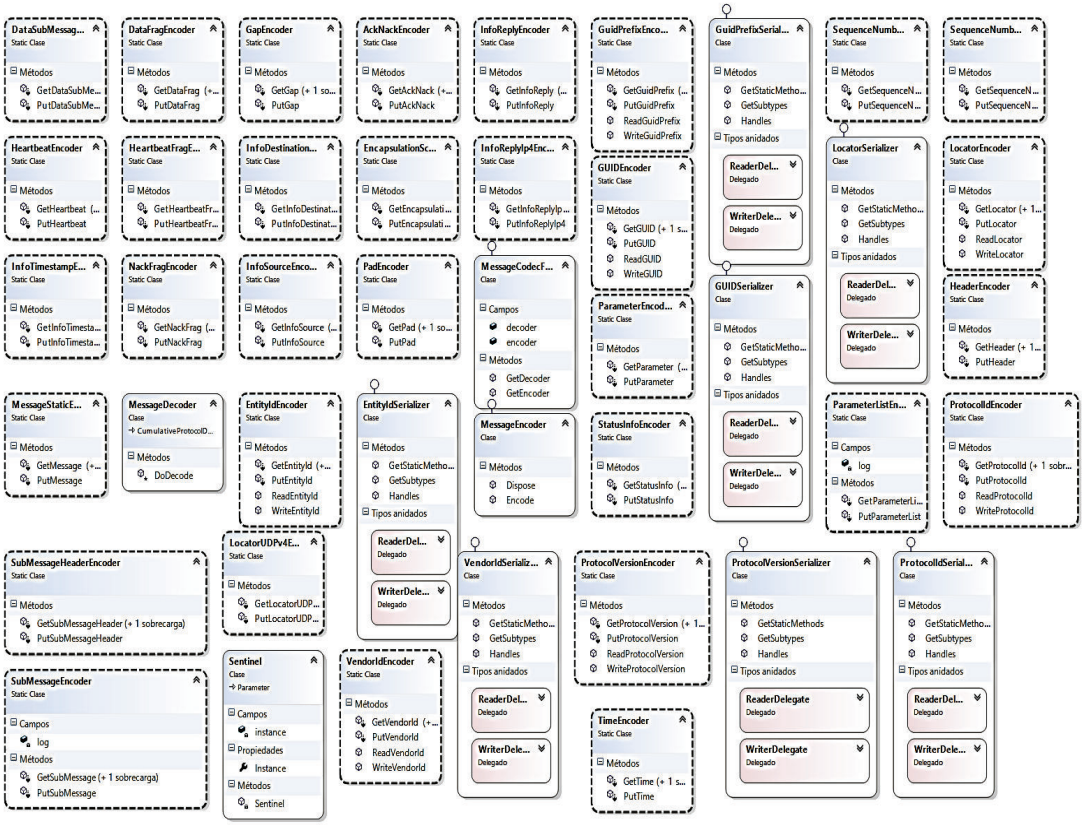


Figura 3.21. Diagrama de clases de la encapsulación de mensajes, cabeceras e identificadores

3.2.3.1. Submódulo de mensaje y encapsulamiento

El submódulo de mensaje y encapsulamiento es el encargado de los codificadores, del encapsulamiento y de la administración del encapsulamiento, para lo cual se define una serie de clases que permiten la codificación y

decodificación de los diferentes mensajes RTPS, cabeceras, e identificadores. Así como se muestra en la Figura 3.21 y la Tabla 3.11.

A partir del submódulo de mensaje y encapsulación del API-RTPS, se creó nuevas clases, las cuales son necesarias para la correcta implementación del submódulo, por ejemplo se definió los codificadores y decodificadores de cada tipo de submensaje, como también las clases necesarias para serializar los campos de un mensaje RTPS.

Tabla 3.11. Métodos de las clases para encapsulamiento de mensajes, cabeceras e identificadores

Clase	Método	Descripción
DataSub*MessageEncoder* **HeartbeatEncoder** **InfoTimestampEncoder** **MessageStaticEncoder** **SubmessageHeaderEncoder** **Sub*MessageEncoder*** **DataFragEncoder** **HeartbeatFragEncoder** **NackFragEncoder** **GapEncoder** **InfoDestinationEncoder** **InfoSourceEncoder** **LocatorUDPv4Encoder** **AckNackEncoder** **EncapsulationSchemeEncoder** **PadEncoder** **InfoReplyEncoder** **InfoReplyIp4Encoder** **ParameterEncoder** **StatusInfoEncoder** **TimeEncoder** **ParameterListEncoder** **HeaderEncoder** **SequenceNumber** **SequenceNumberSet** **MessageCodecFactory**	Get	Obtiene el objeto del tipo (nombre de la clase) del buffer
	Put	Pone un objeto del tipo (nombre de la clase) en el buffer
MessageDecoder	DoDecode	Empieza el proceso de decodificación del mensaje
EntityIdEncoder **ProtocolIdEncoder** **GuidPrefixEncoder** **VendorIdEncoder**	Get	Obtiene el objeto (nombre de la clase) del buffer
	Put	Pone el objeto (nombre de la clase) en el buffer
	Read	Leer el (nombre de la clase) del buffer
	Write	Escribir o cambia un (nombre de la clase) del buffer
Sentinel	Sentinel	Crea una instancia en el espacio de memoria correspondiente a la instancia Sentinel
EntityId*Serializer* **VendorId*Serializer*** **GuidPrefix*Serializer*** **GUID*Serializer*** **ProtocolId*Serializer***	GetStaticMethods	Crea los métodos delegados (nombre de la clase) de escritura y lectura
	GetSubtypes	Obtiene un subtipo
	Handles	Compara al tipo del tipo de (nombre de la clase)

Una vez que se tiene la posibilidad de codificar y decodificar mensajes, cabeceras e identificadores dentro de un buffer. Es necesario tener un esquema de serialización el cual se encuentra mostrado en la Figura 3.22 y Tabla 3.12 que contempla la adaptación de ciertas clases que han sido propuestas en la OMG.

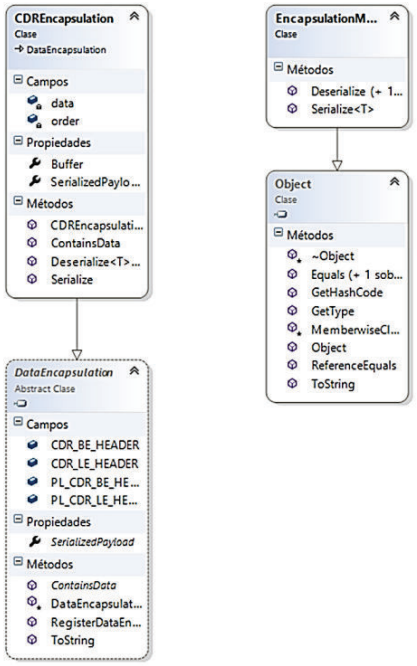


Figura 3.22. Diagrama de clases para la serialización de mensajes.

Tabla 3.12. Métodos de las clases para la serialización y deserialización

Clase	Método	Descripción
CDREncapsulation	CDREncapsulation	Es el método que sirve para serializar mensajes de datos
	ContainsData	Informa si contiene Datos o no
	De*serialize*	Deserializa desde un buffer
	Serialize	Serializa
EncapsulationManager	De*serialize*	Gestiona la deserialización
	Serialize	Gestiona la serialización

3.2.3.2. Submódulo de descubrimiento

El submódulo de descubrimiento es el encargado de los mensajes de descubrimiento y de finalizar el descubrimiento de los participantes, para lo cual se define una serie de clases que permiten el descubrimiento de mensajes RTPS a partir del API-RTPS. A continuación en la Figura 3.23 se muestra el diagrama de clases del submódulo de descubrimiento y en la Tabla 3.13 se realiza una explicación de cada método utilizado en el submódulo.

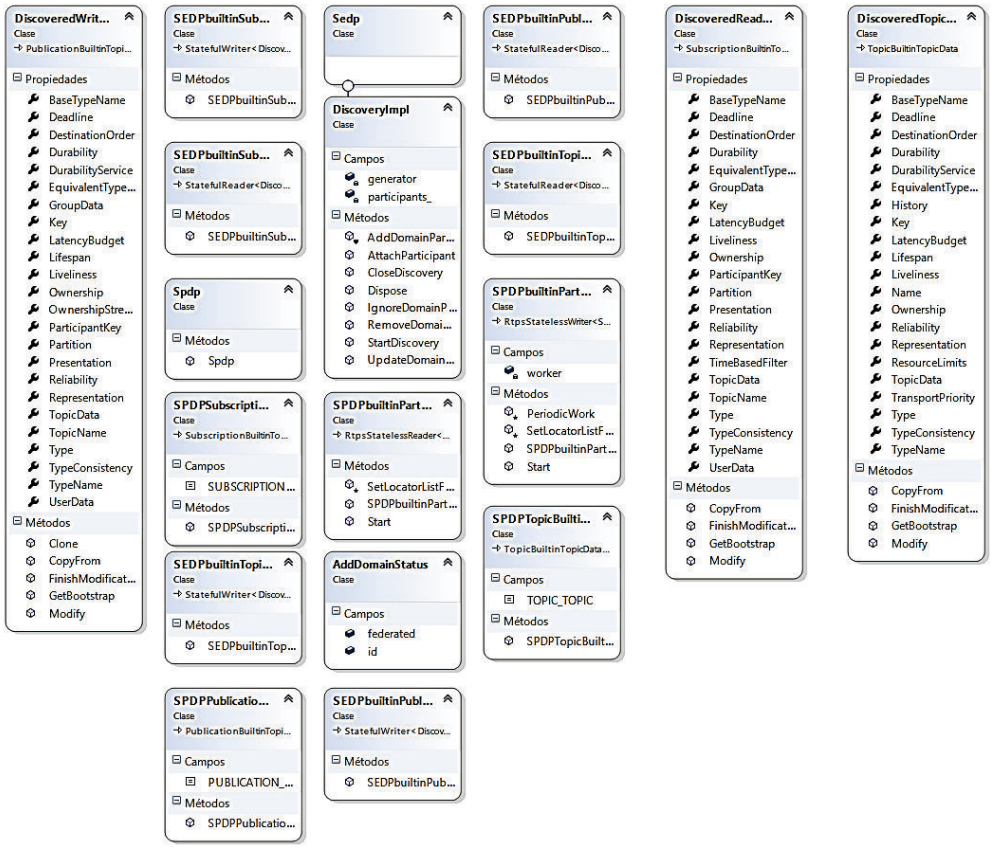


Figura 3.23. Diagrama de clases del submódulo descubrimiento.

Tabla 3.13. Métodos de las clases para el descubrimiento.

Clases	Método	Descripción
SEDPbuiltinSubscriptions*Writer*	SEDPbuiltinSubscriptions*Writer*	Se refiere a la suscripción de un *Writer* por medio del protocolo de descubrimiento SEDP.
SEDPbuiltinSubscriptions*Reader*	SEDPbuiltinSubscriptions*Reader*	Se refiere a la suscripción de un *Reader* por medio del protocolo de descubrimiento SEDP.
SEDPbuiltinTopics*Writer*	SEDPbuiltinTopics*Writer*	Se refiere a la representación de un topic *Writer* en el protocolo SEDP,
SEDPbuiltinTopics*Reader*	SEDPbuiltinTopics*Reader*	Se refiere a la representación de un topic *Reader* en el protocolo SEDP.
Sedp		Se refiere a la representación de una instancia del protocolo SEDP en la implementación del protocolo RTPS.
SEDPbuiltinPublications*Reader*	SEDPbuiltinPublications*Reader*	Se refiere a las publicaciones descubiertas por el *Reader* por medio del protocolo SEDP.

Tabla 3.13. Métodos de las clases para el descubrimiento.

SEDPbuiltinPublications*Writer*	SEDPbuiltinPublications*Writer*	Se refiere a las publicaciones descubiertas por el *Writer* por medio del protocolo SEDP.
Spdp	Spdp	Se refiere a la representación de una instancia del protocolo SPDP en la implementación del protocolo RTPS.
SPDPSubscriptionBuiltinTopicData	SPDPSubscriptionBuiltinTopicData	Se refiere a la suscripción en el protocolo SPDP de un topic.
SPDPPublicationBuiltinTopicData	SPDPPublicationBuiltinTopicData	Se refiere a las publicaciones descubiertas por el topic por medio del protocolo SPDP.
SPDPbuiltinParticipant*Writer*Impl	PeriodicWork	Envía periódicamente objetos de datos.
	SetLocatorListFromConfig	Lista pre configurada con objetos de datos para anunciar la presencia de una participante en la red.
	Start	Inicia el envío de objetos de datos.
SPDPbuiltinParticipant*Reader*Impl	SetLocatorListFromConfig	Lista pre configurada con objetos de datos para anunciar la presencia de una participante en la red.
	Start	Inicia la recepción de objetos de datos.

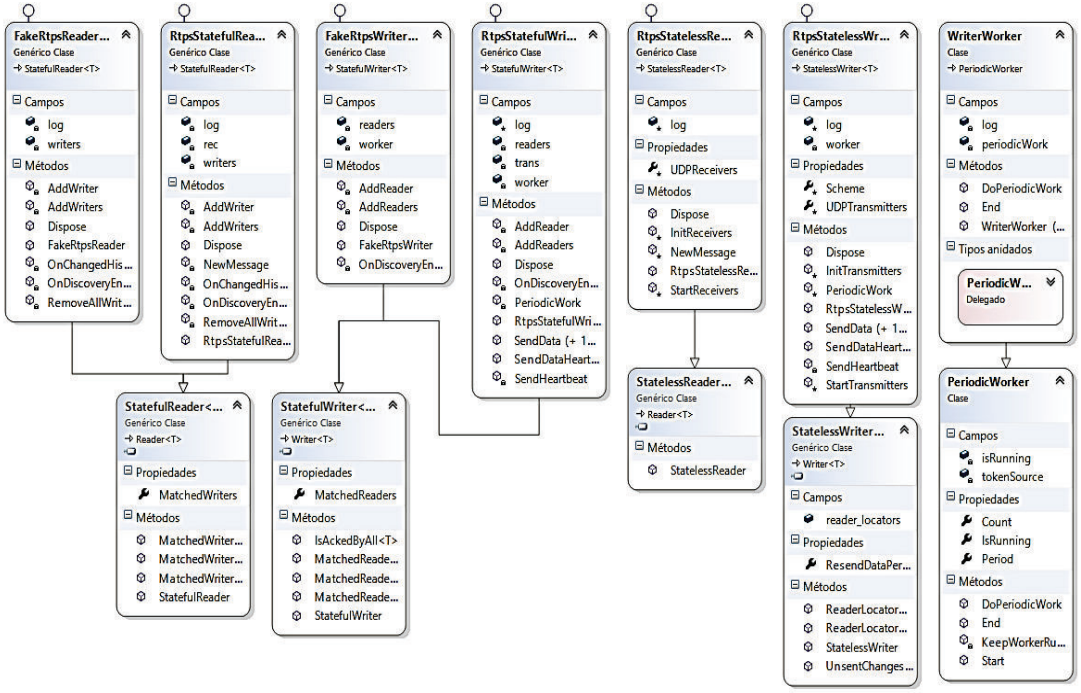


Figura 3.24. Diagrama de clases del submódulo comportamiento

3.2.3.3. Submódulo de comportamiento

El submódulo de comportamiento es el encargado de la interfaz con el módulo DDS, de los *Writer* y *Reader* y del funcionamiento con estado y sin estado, para lo cual se define una serie de clases en base al API-RTPS que permiten mostrar el comportamiento de la comunicación. A continuación se presenta en la Figura 3.24 el diagrama de clases del submódulo comportamiento y en la Tabla 3.14 se muestran los métodos necesarios para las clases para la comunicación entre el protocolo RTPS y el Middleware DDS.

Tabla 3.14. Métodos de las clases para la comunicación del RTPS con el DDS.

Clases	Métodos	Descripción
FakeRtps*Reader*	Add*Writer*	Añade un *Writer* a las lista de *Writer* en el lado del *Reader*.
	Add*Writer*s	Añade al grupo de *Writer* en los Endpoint por medio de un protocolo de descubrimiento.
	Dispose	Dispone o desregistra todos los *Writer* de los Endpoint.
	FakeRtps*Reader*	Representa al *Reader* RTPS falso.
	OnChanged*HistoryCache*	Permite añadir, detectar y remover cambios en el *HistoryCache*.
	OnDiscoveryEndpoints	Añade automáticamente los *Writer* encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
	RemoveAll*Writer*s	Limpia la información del *HistoryCache* de los *Writer*.
FakeRtps*Writer*	Add*Reader*	Añade un *Reader* a las lista de *Reader* en el lado del *Writer*.
	Add*Reader*s	Añade al grupo de *Reader* en los Endpoint por medio de un protocolo de descubrimiento.
	Dispose	Dispone o desregistra todos los *Reader* de los Endpoint.
	FakeRtps*Writer*	Representa al *Writer* RTPS falso.
	OnDiscoveryEndpoints	Añade automáticamente los *Reader* encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
Rtps*StatefulReader*	Add*Writer*	Añade un *Writer* a las lista de *Writer* en el lado del *Reader*.
	Add*Writer*s	Añade al grupo de *Writer* en los Endpoint por medio de un protocolo de descubrimiento.
	Dispose	Dispone o desregistra todos los *Writer* de los Endpoint.
	NewMessage	Crea los mensajes RTPS.
	OnChanged*HistoryCache*	Permite añadir, detectar y remover cambios en el *HistoryCache*.
	OnDiscoveryEndpoints	Añade automáticamente los *Writer* encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
	RemoveAll*Writer*s	Limpia la información del *HistoryCache* de los *Writer*.
	Rtps*StatefulReader*	Representa al *Reader* RTPS con estado.
Rtps*StatefulWriter*	Add*Reader*	Añade un *Reader* a las lista de *Reader* en el lado del *Writer*.
	Add*Reader*s	Añade al grupo de *Reader* en los Endpoint por medio de un protocolo de descubrimiento.

Tabla 3.14. Métodos de las clases para la comunicación del RTPS con el DDS.

	Dispose	Dispone o desregistra todos los *Reader* de los Endpoint.
	OnDiscoveryEndpoints	Añade automáticamente los *Reader* encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
	PeriodicWork	Anuncia repetidamente la disponibilidad de datos enviando un mensaje HeartBeat.
	Rtps*StatefullWriter*	Representa al *Writer* RTPS con estado.
	SendData	Envía el mensaje RTPS.
	SendDataHeartbeat	Crea un mensaje InfoSource y lo envía.
RtpsStateless*Writer*	Dispose	Dispone o desregistra todos los *Reader* de los Endpoint.
	InitTransmitters	Añade transmisores UDP a una lista de transmisión multicast.
	PeriodicWork	Anuncia repetidamente la disponibilidad de datos enviando un mensaje HeartBeat.
	SendData	Envía el mensaje RTPS.
	SendDataHeartbeat	Crea un mensaje InfoSource y lo envía.
	SendHeartbeat	Crea un mensaje HeartBeat y lo envía.
	StartTransmitters	Inicia la transmisión UDP.
	RtpsStateless*Writer*	Representa al *Writer* RTPS sin estado.
RtpsStateless*Reader*	Dispose	Dispone o desregistra todos los *Writer* de los Endpoint.
	InitReceivers	Añade receptores UDP a una lista de transmisión multicast.
	NewMessage	Crea los mensajes RTPS.
	RtpsStateless*Reader*	Representa al *Reader* RTPS sin estado.
	StartReceivers	Inicia la recepción UDP.
*Writer*Worker	DoPeriodicWork	Inicia el PeriodicWork.
	End	Finaliza el PeriodicWork.
	*Writer*Worker	Publica un delegado de un PeriodicWorker.

3.2.3.4. Submódulo de transporte

En un principio para realizar la codificación de los demás submódulos se diseñó un sistema de transporte local o falso, lo que quiere decir que en realidad se simulaba el transporte de datos, trabajando a nivel local. Esto se realizó tanto para los mensajes de descubrimiento como para los mensajes RTPS. A continuación se presenta en la Figura 3.25 los diagramas de clases del sistema falso de transporte local, los cuales representan a las clases correspondientes *FakeDiscovery* el cual está descrito en la Tabla 3.15, y a los eventos de descubrimiento que interactúan con DDS; a pesar que estos pertenecen al mismo submódulo de transporte no tienen relación entre sí ya que los eventos de descubrimiento son manejados por el DDS.

Tabla 3.15. Métodos de la clase FakeDiscovery.

Clase	Método	Descripción
FakeDiscovery	NotifyEndpointsChanges	Notifica los cambios en los endpoint.
	NotifyParticipantChanges	Notifica los cambios en el participante.
	RegisterEndpoint	Registra los nuevos endpoint.
	RegisterParticipant	Registra los nuevos participantes.
	UnregisterEndpoint	Borra del registro a los endpoint.
	UnregisterParticipant	Borra del registro a los participantes.

Una vez completada la codificación de los demás submódulos se procedió a diseñar el transporte sobre la red en sí. Tomando en cuenta que el protocolo de transporte RTPS trabaja sobre UDP se procedió, en primer lugar a implementar las clases necesarias para enviar y recibir información sobre UDP.

A continuación se presenta en la Figura 3.26 los diagramas de clases del transporte sobre la red, los cuales representan a las clases correspondientes Transmisor y Receptor UDP el cual está descrito en la Tabla 3.16, y a los eventos de entrada y salida de datos; a pesar que estos pertenecen al mismo submódulo de transporte no tienen relación entre sí ya que algunos representan la interfaz al protocolo RTPS.

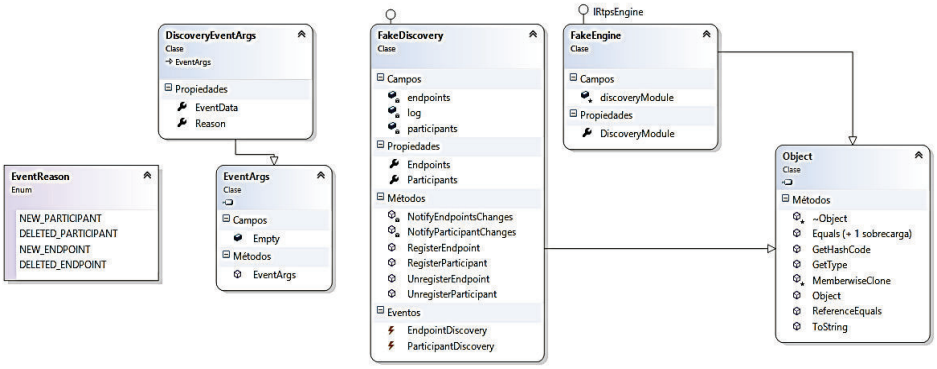


Figura 3.25. Diagrama de clases del sistema falso de transporte.

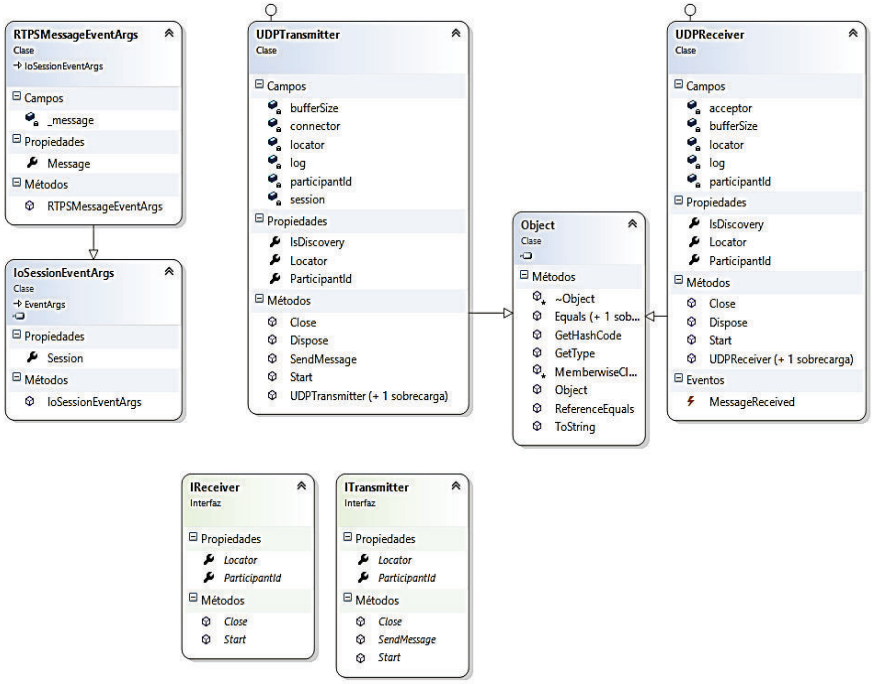


Figura 3.26. Diagrama de clases del sistema de transporte sobre la red.

Tabla 3.16. Métodos de las diferentes clases para la implementación del sistema transporte en red.

Clase	Método	Descripción
UDPTransmitter	Close	Cierra la sesión en el lado del transmisor.
	Dispose	Dispone la sesión en el lado del transmisor.
	SendMessage	Envía un mensaje UDP.
	Start	Inicia la sesión.
	UDPTransmitter	Obtiene la dirección IP desde un DNS para enviar la información en el caso de que no se la tenga.
UDPReceiver	Close	Cierra la sesión en el lado del receptor.
	Dispose	Dispone la sesión en el lado del receptor.
	Start	Acepta una sesión unicast o multicast.
	UDPReceiver	Obtiene direcciones IP desde un DNS antes de recibir información.
ITransmitter	Close	Cierra el transmisor.
	SendMessage	Envía mensajes a su destino.
	Start	Inicia el transmisor.
IReceiver	Close	Cierra el receptor.
	Start	Inicia el receptor.

Finalmente teniendo las clases necesarias para poner mensajes sobre la red se procedió a implementar las clases para enviar mensajes RTPS y mensajes de descubrimiento RTPS trabajando en conjunto con el envío de mensajes sobre la red.

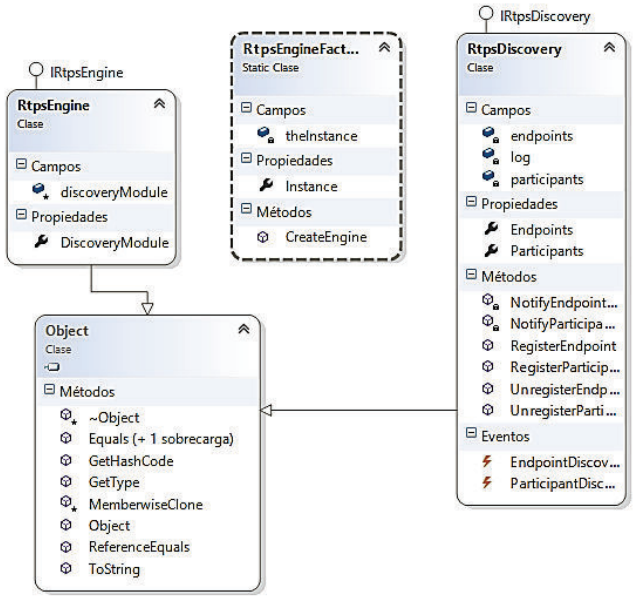


Figura 3.27. Diagrama de clases de los mensajes de descubrimiento y mensajes RTPS.

A continuación se presenta en la Figura 3.27 los diagramas de clases para el envío de mensajes RTPS y mensajes de descubrimiento RTPS, los cuales representan a las clases correspondientes al *EngineFactoryRTPS* y a las interfaces

las cuales están descritas en la Tabla 3.17; a pesar que estos pertenecen al mismo submódulo de transporte no tienen relación entre sí ya que algunas clases pertenecen a la interfaz al RTPS.

Tabla 3.17. Métodos de los mensajes de descubrimiento y mensajes RTPS

Clase	Método	Descripción
RtpsDiscovery	NotifyEndpointsChanges	Notifica los cambios en los endpoint.
	NotifyParticipantChanges	Notifica los cambios en el participante.
	RegisterEndpoint	Registra los nuevos endpoint.
	RegisterParticipant	Registra los nuevos participantes.
	UnregisterEndpoint	Borra del registro a los endpoint.
	UnregisterParticipant	Borra del registro a los participantes.
RtpsEngineFactory	CreateEngine	Crea toda la maquinaria necesaria para la comunicación a partir de los parámetros DDS y RTPS encontrados en el archivo de configuración.

3.2.3.5. Submódulo de configuración

El submódulo de configuración es el encargado de interpretar un archivo de configuración, el cual tiene parámetros configurables tanto del módulo DDS como del módulo RTPS. A continuación se muestran en la Figura 3.28 y Figura 3.29 la configuración básica de DDS y RTPS respectivamente.

El submódulo de configuración se lo diseñó con el objetivo de tener facilidad al configurar parámetros, y facilitar al usuario el uso de RTPS y DDS.

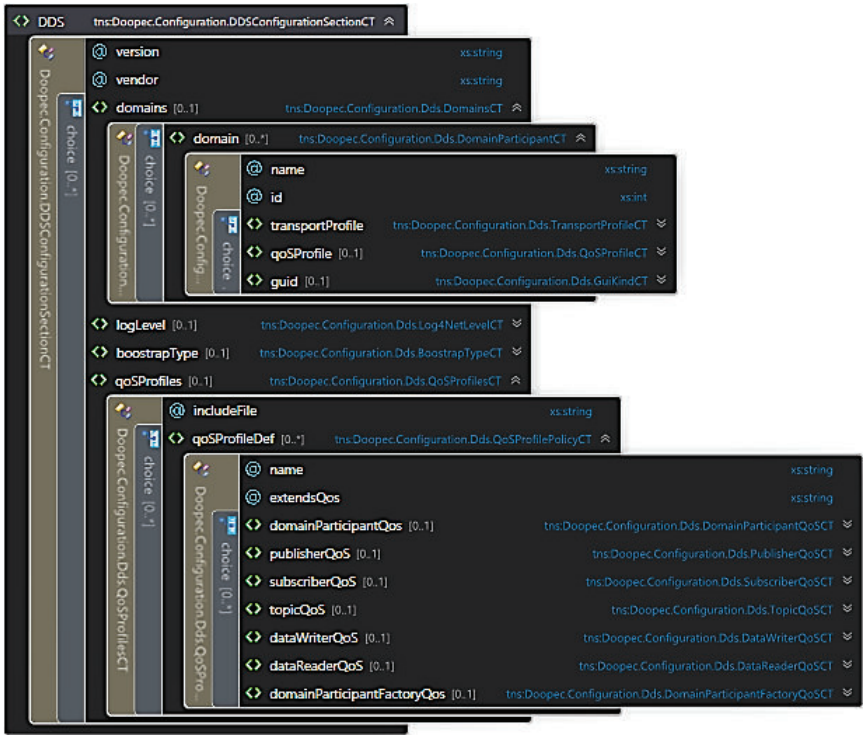


Figura 3.28. Diagrama del Archivo de configuración sección DDS

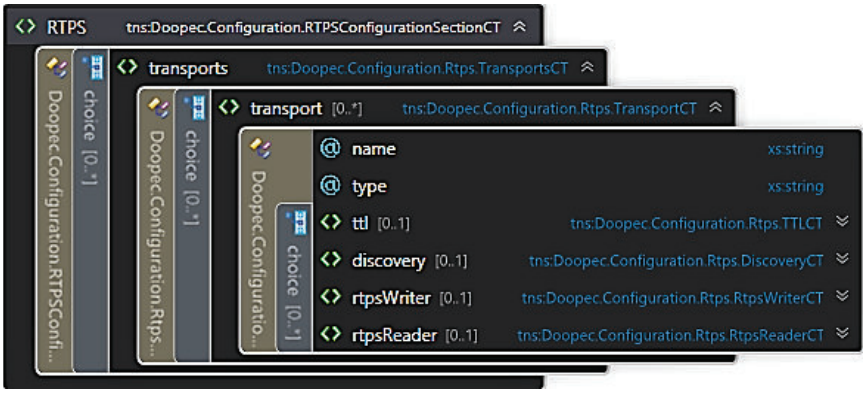


Figura 3.29. Diagrama del Archivo de configuración sección RTPS

3.3. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS

A continuación se muestra la descripción de cada diagrama mostrado en la sección 3.4

3.3.1. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS CON ESTADO

3.3.1.1. Interacción en base a la QoS Best Effort

3.3.1.1.1. Best Effort Reader – Best Effort Writer

La siguiente descripción corresponde a la Figura 3.35.

1) El usuario DDS escribe datos por medio de la llamada a la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS llama a la operación *new_change* en el *Writer* RTPS para crear un nuevo *CacheChange*. Cada uno de estos cambios es identificado únicamente por un *SequenceNumber*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS utiliza la operación *add_change* para almacenar el *CacheChange* dentro de *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica el cambio por medio de la operación *notify_change* al *Publisher* DDS.
6) La operación *notify_change* termina.
7) La operación *add_change* termina.
8) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
9) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.

10) La operación *unsent_changes* termina.
11) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
12) La operación *can_send* termina.
13) El *DataWriter* DDS utiliza la operación *remove_change* en el *HistoryCache* del *Writer* DDS para limpiar la cache. Esta operación puede ser realizada posteriormente.
14) La operación *remove_change* termina.
15) El *ReaderProxy* serializa la información mediante la operación *serialize* en el *Serializer*.
16) La operación *serialize* termina.
17) El *ReaderProxy* envía el submensaje DATA al *MessageEncoder* para que sea encapsulado.
18) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
19) La operación *encoded_message* termina.
20) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
21) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
22) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
23) La operación *doDecode* termina.
24) El *MessageDecoder* envía el submensaje DATA al *WriterProxy*.
25) El *WriterProxy* llama a la operación *deserialize_data* al *Deserializer*
26) La operación *deserialize_data* termina.
27) El *WriterProxy* llama a la operación *available_change* dentro del *HistoryCache* del *Reader* RTPS, para la verificación de números de secuencia recibidos.
28) La operación *available_change* termina.
29) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
30) La operación *new_change* termina.
31) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*

32) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
33) La operación *notify_change* termina.
34) La operación *add_change* termina.
35) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
36) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
37) La operación *get_change* termina.
38) La operación *take* termina. Los datos recibidos son entregados al usuario.
39) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
40) La operación *remove_change* termina.

3.3.1.1.2. Best Effort Reader – BestEffort Writer (Packet Failure)

La siguiente descripción corresponde a la Figura 3.36.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) La operación *notify_change* termina.
7) La operación *add_change* termina.
8) La operación *write* termina.
9) El *HistoryCache* del *Writer* RTPS envía los cambios no enviados mediante la operación *unsent_changes* al *ReaderProxy*.
10) La operación *unsent_changes* termina.
11) El *HistoryCache* del *Writer* RTPS le informa que todos los cambios han sido enviados por medio de la operación *can_send*.

12) La operación *can_send* termina.
13) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
14) La operación *serialize* termina.
15) El *DataWriter* DDS elimina el cambio enviado mediante la operación *remove_change.*
16) La operación *remove_change* termina.
17) El *ReaderProxy* envía el submensaje DATA al *MessageEncoder*, para que el mensaje sea encapsulado.
18) El *MessageEncoder* envía el mensaje encapsulado mediante la operación *encoded_message*.
19) La operación *encoded_message* termina.
20) El UDPTransmitter envía el mensaje UDP a la red.
21) El usuario intenta obtener datos mediante la operación *take* al *DataReader* DDS.
22) El *DataReader* DDS intenta obtener los cambios mediante la operación *get_change* al *HistoryCache* del *Reader* RTPS.
23) La operación *get_change* termina.
24) La operación *take* termina.

3.3.1.2. Interacción en base a la QoS Reliable

3.3.1.2.1. Reliable Reader—Reliable Writer

La siguiente descripción corresponde a la Figura 3.37.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) La operación *notify_change* termina.
7) La operación *add_change* termina.

8) La operación *write* termina.
9) El *HistoryCache* del *Writer* RTPS envía los cambios no enviados mediante la operación *unsent_changes* al *ReaderProxy*.
10) La operación *unsent_changes* termina.
11) El *HistoryCache* del *Writer* RTPS le informa que todos los cambios han sido enviados por medio de la operación *can_send*.
12) La operación *can_send* termina.
13) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
14) La operación *serialize* termina.
15) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
16) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
17) La operación *unacked_changes* termina.
18) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT.
19) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
20) La operación *encoded_message* termina.
21) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
22) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
23) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder.*
24) La operación *doDecode* termina.
25) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
26) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
27) La operación *deserialize_data* termina.
28) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader* RTPS.

29) La operación *missing_changes* termina.
30) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize.*
31) La operación *serialize* termina.
32) El *WriterProxy* envía los submensajes INFO_DESTINATION y ACKNACK con la confirmación de recepción o pérdida de paquetes.
33) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
34) La operación *encoded_message* termina.
35) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
36) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
37) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
38) La operación *doDecode* termina.
39) El *ReaderProxy* recibe los submensajes INFO_DESTINATION y ACKNACK desde el *MessageEncoder.* El submensaje INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
40) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
41) La operación *deserialize_data* termina.
42) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.
43) La operación *acked_changes* termina.
44) El *DataWriter* DDS consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
45) La operación *is_acked_by_all* termina.
46) El *DataWriter* DDS elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer* RTPS.
47) La operación *remove_change* termina.
48) Esta literal toma lugar después del punto 25, luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder.*

49) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
50) La operación *new_change* termina.
51) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
52) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
53) La operación *notify_change* termina.
54) La operación *add_change* termina.
55) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
56) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
57) La operación *get_change* termina.
58) La operación *take* termina. Los datos recibidos son entregados al usuario.
59) El usuario indica al *DataReader* DDS que ya obtuvo el cambio mediante la operación *return_loan.*
60) El *DataReader* DDS pregunta al *HistoryCache* del *Reader* RTPS si el cambio indicado es relevante mediante la operación *a_change_is_relevant.*
61) La operación *a_change_is_relevant* termina.
62) Dependiendo si el cambio es relevante el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
63) La operación *remove_change* termina.
64) La operación *return_loan* termina.

3.3.1.2.2. Reliable Reader—Reliable Writer con fragmentación

La siguiente descripción corresponde a la Figura 3.38.

1) DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.

5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) La operación *notify_change* termina.
7) La operación *add_change* termina.
8) La operación *write* termina.
9) El *HistoryCache* del *Writer* RTPS envía los cambios no enviados mediante la operación *unsent_changes* al *ReaderProxy*.
10) La operación *unsent_changes* termina.
11) El *HistoryCache* del *Writer* RTPS le informa que todos los cambios han sido enviados por medio de la operación *can_send*.
12) La operación *can_send* termina.
13) El *ReaderProxy* serializa los datos mediante la operación *serialize*, y se realiza el proceso de fragmentación de la información.
14) La operación *serialize* termina.
15) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA_FRAG y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
16) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos a los submensajes HEARTBEAT_FRAG.
17) La operación *unacked_changes* termina.
18) El *ReaderProxy* envía al *MessageEncoder* los submensajes HEARTBEAT_FRAG.
19) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
20) La operación *encoded_message* termina.
21) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
22) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
23) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
24) La operación *doDecode* termina.
25) *WriterProxy* recibe los submensajes HEARTBEAT_FRAG desde el *MessageDecoder.*

26) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
27) La operación *deserialize_data* termina.
28) El *WriterProxy* obtiene una lista de los cambios que se han perdido con la operación *missing_changes* al *HistoryCache* del *Reader* RTPS.
29) La operación *missing_changes* termina.
30) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize.*
31) La operación *serialize* termina.
32) El *WriterProxy* envía los submensajes INFO_DESTINATION y NACK_FRAG con la confirmación de recepción o pérdida de paquetes.
33) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
34) La operación *encoded_message* termina.
35) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
36) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
37) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
38) La operación *doDecode* termina.
39) El *ReaderProxy* recibe los submensajes INFO_DESTINATION y NACK_FRAG desde el *MessageEncoder.* INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
40) El *ReaderProxy* deserializa por medio de la operación *deserialize_data.*
41) La operación *deserialize_data* termina.
42) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.
43) La operación *acked_changes* termina.
44) El *DataWriter* DDS consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
45) La operación *is_acked_by_all* termina.
46) El *DataWriter* DDS elimina los cambios cuando todos suscriptores han recibido por medio de la operación *remove_change* al *HistoryCache* del *Writer* RTPS.

47) La operación *remove_change* termina.
48) Esta literal toma lugar después del punto 25, luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA_FRAG del *MessageDecoder.*
49) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
50) La operación *new_change* termina.
51) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
52) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
53) La operación *notify_change* termina.
54) La operación *add_change* termina.
55) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
56) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
57) La operación *get_change* termina.
58) La operación *take* termina. Los datos recibidos son entregados al usuario.
59) El usuario indica al *DataReader* DDS que ya obtuvo el cambio mediante la operación *return_loan.*
60) El *DataReader* DDS pregunta al *HistoryCache* del *Reader* RTPS si el cambio indicado es relevante mediante la operación *a_change_is_relevant.*
61) La operación *a_change_is_relevant* termina.
62) Dependiendo si el cambio es relevante el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
63) La operación *remove_change* termina.
64) La operación *return_loan* termina.

3.3.1.2.3. Reliable Reader—Reliable Writer (Communication Error)

La siguiente descripción corresponde a la Figura 3.39.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) El *Publisher* DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación *data_available.*
7) El *ReaderProxy* serializa la notificación mediante la operación *serialize.*
8) La operación *serialize* termina.
9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder.*
10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
11) La operación *encoded_message* termina.
12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
15) La operación *doDecode* termina.
16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data.*
18) La operación *deserialize_data* termina.
19) La operación *notify_change* termina.
20) La operación *add_change* termina.
21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.

22) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
23) La operación *unsent_changes* termina.
24) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
25) La operación *can_send* termina.
26) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
27) La operación *serialize* termina.
28) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
29) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
30) La operación *unacked_changes* termina.
31) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
32) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
33) La operación *encoded_message* termina.
34) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos. En este punto se ha simulado una falla en la comunicación por tanto el mensaje no llega a su destino.
35) El *ReaderProxy* reenvía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
36) El *ReaderProxy* reenvía al *MessageEncoder* el submensaje HEARTBEAT.
37) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
38) La operación *encoded_message* termina.
39) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
40) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.

41) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
42) La operación *doDecode* termina.
43) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
44) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
45) La operación *deserialize_data* termina.
46) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader* RTPS.
47) La operación *missing_changes* termina.
48) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize.*
49) La operación *serialize* termina.
50) El *WriterProxy* envía los submensajes INFO_DESTINATION y ACKNACK con la confirmación de recepción o pérdida de paquetes.
51) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
52) La operación *encoded_message* termina.
53) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
54) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
55) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
56) La operación *doDecode* termina.
57) El *ReaderProxy* recibe los submensajes INFO_DESTINATION y ACKNACK desde el *MessageEncoder.* El submensaje INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
58) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
59) La operación *deserialize_data* termina.
60) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.

61) La operación *acked_changes* termina.
62) El *DataWriter* DDS consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
63) La operación *is_acked_by_all* termina.
64) El *DataWriter* DDS elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer* RTPS.
65) La operación *remove_change* termina.
66) Esta literal toma lugar después del punto 43, luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder.*
67) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
68) La operación *new_change* termina.
69) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
70) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
71) La operación *notify_change* termina.
72) La operación *add_change* termina.
73) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
74) El *DataReader* DDS solicita los cambios con la operación *get_change*.
75) La operación *get_change* termina.
76) La operación *take* termina. Los datos recibidos son entregados al usuario.
77) El usuario indica al *DataReader* DDS que ya obtuvo el cambio mediante la operación *return_loan.*
78) El *DataReader* DDS pregunta al *HistoryCache* del *Reader* RTPS si el cambio indicado es relevante mediante la operación *a_change_is_relevant.*
79) La operación *a_change_is_relevant* termina.
80) Dependiendo si el cambio es relevante el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.

81) La operación *remove_change* termina.
82) La operación *return_loan* termina.

3.3.1.2.4. Reliable Reader—Reliable Writer (Packet Failure)

La siguiente descripción corresponde a la Figura 3.40.

1) El usuario DDS escribe datos con operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) El *Publisher* DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación *data_available.*
7) El *ReaderProxy* serializa la notificación mediante la operación *serialize.*
8) La operación *serialize* termina.
9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder,* que será dirigido al participante uno.
10) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder,* que será dirigido al participante dos.
11) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
12) La operación *encoded_message* termina.

Del punto 13 al 18 pertenecen al participante 2

13) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos, dirigido hacia el participante dos.
14) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
15) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
16) La operación *doDecode* termina.
17) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.

18) El *WriterProxy* deserializa el submensaje con operación *deserialize_data.*
19) La operación *deserialize_data* termina.

Del punto 20 al 26 pertenecen al participante 1

20) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos, dirigido hacia el participante dos.
21) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
22) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
23) La operación *doDecode* termina.
24) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
25) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data.*
26) La operación *deserialize_data* termina.
27) La operación *notify_change* termina.
28) La operación *add_change* termina.
29) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
30) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
31) La operación *unsent_changes* termina.
32) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
33) La operación *can_send* termina.
34) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
35) La operación *serialize* termina.
36) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP, va dirigido al suscriptor uno.
37) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP, va dirigido al suscriptor dos.

38) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
39) La operación *unacked_changes* termina.
40) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones, va dirigido al suscriptor uno.
41) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones, va dirigido al suscriptor dos.

Del punto 42 al 59 pertenece al suscriptor 2

42) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
43) La operación *encoded_message* termina.
44) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
45) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
46) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
47) La operación *doDecode* termina.
48) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
49) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
50) La operación *deserialize_data* termina.
51) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader* RTPS.
52) La operación *missing_changes* termina.
53) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize.*
54) La operación *serialize* termina.
55) El *WriterProxy* envía los submensajes INFO_DESTINATION y ACKNACK con la confirmación de recepción o pérdida de paquetes.

56) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
57) La operación *encoded_message* termina.
58) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
59) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.

Del punto 60 al 77 pertenece al suscriptor 1

60) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
61) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos. El mensaje se corrompe.
62) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
63) La operación *doDecode* termina.
64) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
65) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
66) La operación *deserialize_data* termina.
67) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader* RTPS.
68) La operación *missing_changes* termina.
69) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize.*
70) La operación *serialize* termina.
71) El *WriterProxy* envía los submensajes INFO_DESTINATION y ACKNACK con la confirmación de recepción o pérdida de paquetes.
72) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
73) La operación *encoded_message* termina.
74) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
75) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
76) Una vez recibido los mensajes UDP, el *UDPReceiver* desencapsula los mensaje mediante la operación *doDecode* en el *MessageDecoder.*

77) La operación *doDecode* termina.
78) El *ReaderProxy* recibe los submensajes INFO_DESTINATION y ACKNACK desde el *MessageEncoder,* del suscriptor uno. El submensaje INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
79) El *ReaderProxy* recibe los submensajes INFO_DESTINATION y ACKNACK desde el *MessageEncoder,* del suscriptor dos. El submensaje INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
80) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data.*
81) La operación *deserialize_data* termina.
82) El *HistoryCache* del *Writer* RTPS pregunta si hay cambios confirmados mediante la operación *requested_changes* al *ReaderProxy.*
83) El *ReaderProxy* envía los cambios confirmados y no confirmados al *Stateful Writer* mediante la operación *acked_changes.*
84) La operación *acked_changes* termina.
85) La operación *requested_changes* termina.

Del punto 86 al 102 pertenece al suscriptor 2

86) Luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder.*
87) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
88) La operación *new_change* termina.
89) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
90) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
91) La operación *notify_change* termina.
92) La operación *add_change* termina.
93) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
94) El *DataReader* DDS solicita los cambios con la operación *get_change*.
95) La operación *get_change* termina.

96) La operación *take* termina. Los datos recibidos son entregados al usuario.
97) El usuario indica al *DataReader* DDS que ya obtuvo el cambio mediante la operación *return_loan.*
98) El *DataReader* DDS pregunta al *HistoryCache* del *Reader* RTPS si el cambio indicado es relevante mediante la operación *a_change_is_relevant.*
99) La operación *a_change_is_relevant* termina.
100) Dependiendo si el cambio es relevante el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
101) La operación *remove_change* termina.
102) La operación *return_loan* termina.

Desde el punto 103 pertenece al suscriptor 1

103) Luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder.*
104) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
105) La operación *unacked_changes* termina.
106) El *ReaderProxy* reenvía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
107) El *ReaderProxy* reenvía al *MessageEncoder* el submensaje HEARTBEAT.
108) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
109) La operación *encoded_message* termina.
110) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
111) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
112) El *UDPReceiver* desencapsula el mensaje con la operación *doDecode* en el *MessageDecoder*.
113) La operación *doDecode* termina.
114) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*

115) El *WriterProxy* deserializa el submensaje con operación *deserialize_data.*
116) La operación *deserialize_data* termina.
117) El *WriterProxy* obtiene una lista de los cambios que se han perdido con la operación *missing_changes* al *HistoryCache* del *Reader* RTPS.
118) La operación *missing_changes* termina.
119) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize.*
120) La operación *serialize* termina.
121) El *WriterProxy* envía los submensajes INFO_DESTINATION y ACKNACK con la confirmación de recepción o pérdida de paquetes.
122) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
123) La operación *encoded_message* termina.
124) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
125) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
126) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
127) La operación *doDecode* termina.
128) El *ReaderProxy* recibe los submensajes INFO_DESTINATION y ACKNACK desde el *MessageEncoder.* El submensaje INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
129) El *ReaderProxy* deserializa el submensaje con la operación *deserialize_data.*
130) La operación *deserialize_data* termina.
131) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.
132) La operación *acked_changes* termina.
133) El *DataWriter* DDS consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
134) La operación *is_acked_by_all* termina.
135) El *DataWriter* DDS elimina los cambios cuando todos suscriptores han recibido los cambios con la operación *remove_change* al *HistoryCache* del *Writer* RTPS.

136) La operación *remove_change* termina.
137) Esta literal toma lugar después del punto 114, luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder.*
138) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
139) La operación *new_change* termina.
140) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
141) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
142) La operación *notify_change* termina.
143) La operación *add_change* termina.
144) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
145) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
146) La operación *get_change* termina.
147) La operación *take* termina. Los datos recibidos son entregados al usuario.
148) El usuario indica al *DataReader* DDS que ya obtuvo el cambio mediante la operación *return_loan.*
149) El *DataReader* DDS pregunta al *HistoryCache* del *Reader* RTPS si el cambio indicado es relevante mediante la operación *a_change_is_relevant.*
150) La operación *a_change_is_relevant* termina.
151) Dependiendo si el cambio es relevante el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
152) La operación *remove_change* termina.
153) La operación *return_loan* termina.

3.3.1.3. Interacción en base a la combinación QoS reliable – Best Effort

3.3.1.3.1. Reliable Writer—Best Effort Reader

La siguiente descripción corresponde a la Figura 3.41.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) El *Publisher* DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación *data_available.*
7) El *ReaderProxy* serializa la notificación mediante la operación *serialize.*
8) La operación *serialize* termina.
9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder.*
10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
11) La operación *encoded_message* termina.
12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder.*
15) La operación *doDecode* termina.
16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data.*
18) La operación *deserialize_data* termina.
19) La operación *notify_change* termina.
20) La operación *add_change* termina.
21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.

22) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
23) La operación *unsent_changes* termina.
24) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
25) La operación *can_send* termina.
26) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
27) La operación *serialize* termina.
28) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
29) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT. Esta operación asume que todos los mensajes son confirmados cuando se trabaja con suscriptores con mejor esfuerzo.
30) La operación *unacked_changes* termina.
31) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
32) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
33) La operación *encoded_message* termina.
34) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
35) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
36) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
37) La operación *doDecode* termina.
38) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
39) El *WriterProxy* recibe el submensaje GAP Y DATA desde el *MessageDecoder.*
40) El *WriterProxy* llama a la operación *deserialize_data* al *Deserializer*

41) La operación *deserialize_data* termina.
42) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.
43) La operación *new_change* termina.
44) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
45) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
46) La operación *notify_change* termina.
47) La operación *add_change* termina.
48) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
49) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
50) La operación *get_change* termina.
51) La operación *take* termina. Los datos recibidos son entregados al usuario.
52) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
53) La operación *remove_change* termina.
54) El *DataWriter* DDS consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
55) La operación *is_acked_by_all* termina.
56) El *DataWriter* DDS elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer* RTPS.
57) La operación *remove_change* termina.

3.3.1.3.2. Reliable Writer—Best Effort Reader (Packet Failure)

La siguiente descripción corresponde a la Figura 3.42.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.

4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) El *Publisher* DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación *data_available.*
7) El *ReaderProxy* serializa la notificación mediante la operación *serialize.*
8) La operación *serialize* termina.
9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder.*
10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
11) La operación *encoded_message* termina.
12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
15) La operación *doDecode* termina.
16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data.*
18) La operación *deserialize_data* termina.
19) La operación *notify_change* termina.
20) La operación *add_change* termina.
21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
22) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
23) La operación *unsent_changes* termina.
24) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
25) La operación *can_send* termina.

26) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
27) La operación *serialize* termina.
28) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
29) El *HistoryCache* del *Writer* RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT. Esta operación asume que todos los mensajes son confirmados cuando se trabaja con suscriptores con mejor esfuerzo.
30) La operación *unacked_changes* termina.
31) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
32) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
33) La operación *encoded_message* termina.
34) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
35) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos. Se simula un paquete corrupto.
36) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
37) La operación *doDecode* termina.
38) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
39) El *WriterProxy* recibe el submensaje GAP Y DATA desde el *MessageDecoder.*
40) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
41) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
42) La operación *get_change* termina.
43) La operación *take* termina. Los datos recibidos son entregados al usuario.

44) El *DataWriter* DDS consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
45) La operación *is_acked_by_all* termina.
46) El *DataWriter* DDS elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer* RTPS.
47) La operación *remove_change* termina.

3.3.2. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS SIN ESTADO

3.3.2.1. Interacción en base a la QoS Best Effort

3.3.2.1.1. BestEffort Reader –- Best Effort Writer

La siguiente descripción corresponde a la Figura 3.43.

1) El usuario DDS escribe datos por medio de la llamada a la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS llama a la operación *new_change* en el *Writer* RTPS para crear un nuevo *CacheChange*. Cada uno de estos cambios es identificado únicamente por un *SequenceNumber*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS utiliza la operación *add_change* para almacenar el *CacheChange* dentro de *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica el cambio por medio de la operación *notify_change* al *Publisher* DDS.
6) La operación *notify_change* termina.
7) La operación *add_change* termina.
8) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
9) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderLocator* que hay cambios o información no enviada.
10) La operación *unsent_changes* termina.
11) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderLocator* que puede enviar los cambios.
12) La operación *can_send* termina.

13) El *DataWriter* DDS utiliza la operación *remove_change* en el *HistoryCache* del *Writer* DDS para limpiar la cache. Esta operación puede ser realizada posteriormente.
14) La operación *remove_change* termina.
15) El *ReaderLocator* serializa la información mediante la operación *serialize* en el *Serializer*.
16) La operación *serialize* termina.
17) El *ReaderLocator* envía al *MessageEncoder* el submensaje DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
18) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
19) La operación *encoded_message* termina.
20) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
21) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
22) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder.*
23) La operación *doDecode* termina.
24) El *MessageDecoder* envía el submensaje DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP al *Reader* RTPS.
25) El *Reader* RTPS llama a la operación *deserialize_data* en el *Deserializer.*
26) La operación *deserialize_data* termina.
27) El *Stateless Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
28) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
29) La operación *notify_change* termina.
30) La operación *add_change* termina.
31) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
32) El *DataReader* DDS solicita los cambios por medio de la operación *get_change* al *HistoryCache* del *Reader* RTPS.
33) La operación *get_change* termina.

34) La operación *take* termina. Los datos recibidos son entregados al usuario.
35) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
36) La operación *remove_change* termina.

3.3.2.2. Interacción en base a la QoS Reliable – Best Effort

3.3.2.2.1. Reliable Writer – Best Effort Reader

La siguiente descripción corresponde a la Figura 3.44.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) El *Publisher* DDS indica la disponibilidad de datos al *ReaderLocator* mediante la operación *data_available.*
7) El *ReaderLocator* serializa la notificación mediante la operación *serialize.*
8) La operación *serialize* termina.
9) El *ReaderLocator* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder.*
10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
11) La operación *encoded_message* termina.
12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
15) La operación *doDecode* termina.
16) El *Stateless Reader* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.

17) El *Stateless Reader* deserializa el submensaje mediante la operación *deserialize_data.*
18) La operación *deserialize_data* termina.
19) La operación *notify_change* termina.
20) La operación *add_change* termina.
21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
22) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
23) La operación *unsent_changes* termina.
24) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderLocator* que puede enviar los cambios.
25) La operación *can_send* termina.
26) El *ReaderLocator* serializa los datos mediante la operación *serialize*.
27) La operación *serialize* termina.
28) El *ReaderLocator* envía al *MessageEncoder* los submensajes DATA.
29) El *ReaderLocator* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
30) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
31) La operación *encoded_message* termina.
32) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
33) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
34) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
35) La operación *doDecode* termina.
36) El *Stateless Reader* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
37) El *Stateless Reader* recibe el submensaje DATA desde el *MessageDecoder.*
38) El *Stateless Reader* llama a la operación *deserialize_data* al *Deserializer*
39) La operación *deserialize_data* termina.

40) El *Stateless Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
41) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
42) La operación *notify_change* termina.
43) La operación *add_change* termina.
44) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
45) El *DataReader* DDS solicita los cambios con la operación *get_change*.
46) La operación *get_change* termina.
47) La operación *take* termina. Los datos recibidos son entregados al usuario.
48) El *HistoryCache* del *Writer* RTPS solicita los cambios no confirmados al *ReaderLocator* mediante la operación *requested_changes.* Como se está trabajando con un *Reader* sin estado con mejor esfuerzo, este no confirma ningún cambio por lo cual en esta operación se asume que todo está confirmado.
49) La operación *requested_changes* termina.
50) El *ReaderLocator* envía al *MessageEncoder* el submensaje GAP. Este submensaje en este caso no solicitará ningún número de secuencia.
51) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
52) La operación *encoded_message* termina.
53) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
54) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
55) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
56) La operación *doDecode* termina.
57) El *Stateless Reader* recibe el submensaje GAP desde el *MessageDecoder.*
58) El *Stateless Reader* llama a la operación *deserialize_data* al *Deserializer*.
59) La operación *deserialize_data* termina.
60) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.

61) La operación *remove_change* termina.

3.3.3. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS HÍBRIDOS (CON ESTADO Y SIN ESTADO)

3.3.3.1.1. Reliable Stateless Writer – Reliable Stateful Reader

La siguiente descripción corresponde a la Figura 3.45.

1) El usuario DDS escribe datos mediante la operación *write* en el *DataWriter* DDS.
2) El *DataWriter* DDS crea un *CacheChange* mediante la operación *new_change* al *Stateful Writer*.
3) La operación *new_change* termina.
4) El *DataWriter* DDS añade el cambio mediante la operación *add_change* al *HistoryCache* del *Writer* RTPS.
5) El *HistoryCache* del *Writer* RTPS notifica al *Publisher* mediante la operación *notify_change*.
6) El *Publisher* DDS indica la disponibilidad de datos al *ReaderLocator* mediante la operación *data_available.*
7) El *ReaderLocator* serializa la notificación mediante la operación *serialize.*
8) La operación *serialize* termina.
9) El *ReaderLocator* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder.*
10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
11) La operación *encoded_message* termina.
12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
15) La operación *doDecode* termina.
16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder.* El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data.*

18) La operación *deserialize_data* termina.
19) La operación *notify_change* termina.
20) La operación *add_change* termina.
21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
22) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
23) La operación *unsent_changes* termina.
24) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderLocator* que puede enviar los cambios.
25) La operación *can_send* termina.
26) El *ReaderLocator* serializa los datos mediante la operación *serialize*.
27) La operación *serialize* termina.
28) El *ReaderLocator* envía al *MessageEncoder* los submensajes DATA.
29) El *ReaderLocator* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
30) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
31) La operación *encoded_message* termina.
32) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
33) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
34) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
35) La operación *doDecode* termina.
36) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder.*
37) El *WriterProxy* recibe el submensaje DATA desde el *MessageDecoder.*
38) El *WriterProxy* deserializa los datos mediante la operación *deserialize_data.*
39) La operación *deserialize_data* termina.
40) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader* RTPS.

41) La operación *new_change* termina.
42) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader* RTPS por medio de la operación *add_change.*
43) El *HistoryCache* del *Reader* RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change.*
44) La operación *notify_change* termina.
45) La operación *add_change* termina.
46) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
47) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
48) La operación *get_change* termina.
49) La operación *take* termina. Los datos recibidos son entregados al usuario.
50) El *WriterProxy* envía la confirmación de los datos mediante un submensaje ACKNACK e indica el destinatario mediante el submensaje INFO_REPLY. No se utiliza el submensaje INFO_DESTINATION ya que el publicador es sin estado.
51) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
52) La operación *encoded_message* termina.
53) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
54) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
55) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
56) La operación *doDecode* termina.
57) El *ReaderLocator* recibe los submensajes INFO_REPLY y ACKNACK desde el *MessageEncoder.* El submensaje INFO_DESTINATION contiene el destino el cual ha confirmado el cambio.
58) El *ReaderLocator* deserializa el submensaje por medio de la operación *deserialize_data.*
59) La operación *deserialize_data* termina.
60) El *HistoryCache* del *Writer* RTPS solicita los cambios no confirmados al *ReaderLocator* mediante la operación *requested_changes.*

61) La operación *requested_changes* termina.
62) El *ReaderLocator* envía al *MessageEncoder* el submensaje GAP. Este submensaje en este caso no solicitará ningún número de secuencia.
63) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
64) La operación *encoded_message* termina.
65) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
66) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
67) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
68) La operación *doDecode* termina.
69) El *WriterProxy* recibe el submensaje GAP desde el *MessageDecoder.*
70) El *WriterProxy* llama a la operación *deserialize_data* al *Deserializer*
71) La operación *deserialize_data* termina.
72) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
73) La operación *remove_change* termina.

3.3.4. PROTOCOLO DESCUBRIMIENTO

3.3.4.1. Tráfico de Descubrimiento

3.3.4.1.1. Fase de Descubrimiento de Participantes.

La siguiente descripción corresponde a la Figura 3.46.

1) El participante 1 ha sido creado.
2) El participante 1 se anuncia enviando mensajes SPDP.
3) El Participante 2 es creado.
4) El participante 2 se anuncia enviando mensajes SPDP.
5) El participante 1 recibe los paquetes SPDP y en este caso añade al participante 2 a la base de datos.
6) El participante 1 se anuncia enviando mensajes SPDP.
7) El participante 2 recibe los paquetes SPDP y en este caso añade al participante 1 a la base de datos.
8) El participante 2 se anuncia enviando mensajes SPDP.
9) El participante 1 crea un *DataWriter.*
10) El participante 1 envía su publicación por medio de mensajes SEDP.

11) El participante 2 recibe el mensaje SEDP y añade al *DataWriter* remoto a su base de datos.
12) El participante 1 continúa enviando su publicación mediante mensajes SEDP.
13) El participante 1 destruye su *DataWriter.*
14) El participante 1 informa que el *DataWriter* ha sido eliminado enviando mensajes SEDP.
15) El participante 1 es destruido.
16) El participante 1 informa que ha sido eliminado enviando mensajes SPDP.
17) El participante 2 recibe el mensaje SPDP y remueve al participante 1 de la base de datos.
18) El participante 2 es destruido.
19) El participante 2 informa que ha sido eliminado enviando mensajes SPDP.

3.3.5. INTERCAMBIO DE MENSAJES RTPS SOBRE LA RED

Una vez presentado el comportamiento mediante los diagramas de secuencia de la interacción del protocolo RTPS con DDS, se muestra varios ejemplos sobre el intercambio de mensajes RTPS, con el objetivo de ilustrar gráficamente por medio de una herramienta para capturar tráfico, el flujo de datos que generan programas que utiliza el protocolo RTPS como base, como por ejemplo *OpenDDS* y RTI. A partir de estas capturas se tendrá una idea más clara de como debe comportarse el Middleware RTPS-DDS.

3.3.5.1. Ejemplo 1

El siguiente gráfico ilustra el flujo de datos que han generado un par de entidades que se comunican entre sí.

- Los primeros 3 paquetes capturados que se muestran en el cuadro rojo representan la interacción entre entidades independientes con el DDS.
- Los 3 paquetes siguientes que se muestran en el cuadro azul representan los mensajes equivalentes a un saludo.
- El paquete resaltado en azul representa al suscriptor buscando un servicio por medio del *Topic*, el mensaje no va al publicador en primer lugar, porque no sabe que una de las entidades ya ha publicado ese servicio, el mensaje va directamente al Middleware para consultar si alguna entidad dispone de un servicio para la solicitud.

- El Middleware anuncia al publicador que una entidad requiere sus servicios, y envía la información de la entidad que quiere el servicio, de forma transparente.
- El siguiente mensaje, el cual esta en el cuadro anaranjado representa un mensaje ping que anuncia la presencia de la entidad que tiene el servicio.
- El siguiente mensaje tiene 2 submensajes se encuentra en el cuadro negro, representa primero un submensaje *InfoDestination*, que indica el identificador del servicio publicado a la entidad que ha consultado el servicio, y el otro submensaje corresponde a un submensaje *GAP*, que indica que los siguientes mensajes vienen con un número de secuencia y tienen que mantener un orden.
- El siguiente grupo de submensajes que se encuentran en el cuadro café, representan los datos enviados por el servicio y que tienen un orden.
- El último grupo de mensajes que se encuentran en el cuadro gris representa varios submensajes, con el *Heartbeat* se informa de una entidad a la otra de cualquier cambio en la continuidad de los mensajes, mientras que con el submensaje *AckNack* la entidad que tiene el servicio indica que este ya no se encuentra disponible.

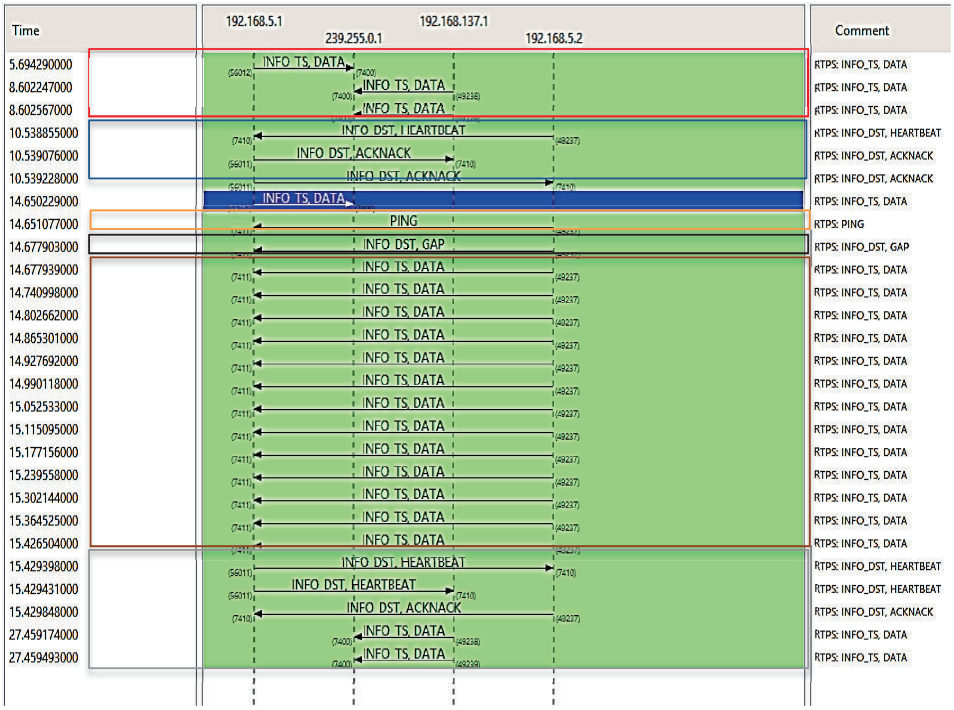


Figura 3.30. Intercambio de mensajes RTPS sobre la red Ejemplo 1

3.3.5.2. Ejemplo 2

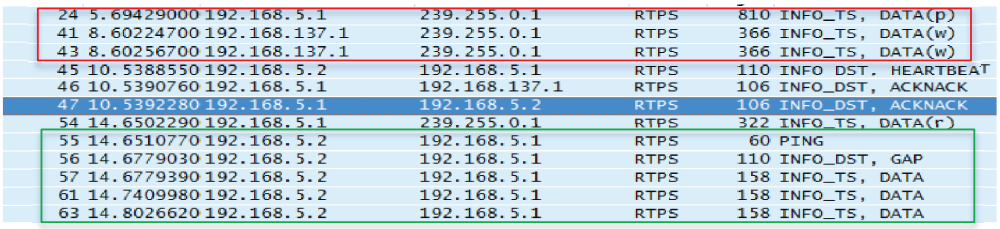


Figura 3.31. Intercambio de Mensajes RTPS Ejemplo 2

La captura corresponde al flujo de datos intercambiados por 2 participantes que trabajan con RTPS sobre el Middleware DDS.

Al inicio de la captura en el cuadro rojo, se puede observar que todos los participantes envían un mensaje con el mismo formato, y que todos son enviados a la IP del Middleware, 239.255.0.1, este mensaje incluye los submensajes *InfoTimeStamp* y *Data*. El submensaje *InfoTimeStamp* tiene el propósito de dar una referencia de tiempo o marca a los siguientes submensajes. El submensaje *Data* sólo envía cambios en los objetos de datos.

En el mensaje resaltado, se puede observar que el *host* con IP 192.168.5.1 envía al host con IP 192.168.5.2 un mensaje RTPS con los submensajes, que corresponden a un *InfoDestination*, el cual tiene el único propósito de enviar información sobre el *guidPrefix* para ser identificado, y un submensaje *AckNack*.

Como ya fue explicado anteriormente el submensaje *AckNack* es usado para comunicar el estado de un Lector a un Escritor, y anteriormente se pudo observar que el *host* 192.168.5.2, envía al *host* 192.168.5.1 un submensaje *Heartbeat*, lo que significa que el participante está saludando y el otro participante responde, con el submensaje *AckNack*, diciendo ¿necesitas algo?, y como la bandera estaba seteada en 1, el *host* con 192.168.5.1 no está obligado a responder.

Dentro del cuadro verde se muestra el intercambio de mensajes que ya fue explicado en el ejemplo 1.

3.3.5.3. Ejemplo 3

La captura corresponde a un flujo de datos intercambiados trabajando de manera local.

Al inicio de la captura como se muestra en cuadro rojo, se puede observar un Submensaje *GAP*, el cual indica que el siguiente submensaje viene un número de secuencia para mantener un orden. A continuación, se puede observar el

submensaje *HeartBeat* que significa que el participante está saludando y responden con los submensajes *InfoDestination* y *AckNack*; el primero, tiene como propósito enviar información sobre el *guidPrefix* para ser identificado localmente y el submensaje *AckNack* es usado para comunicar el estado del Lector al Escritor.

En el cuadro café se muestra un grupo de mensajes, el submensaje es el *DataFrag*, el cual fragmenta los datos y los envía en varios submensajes D*ataFrag*. Los fragmentos contenidos en los submensajes *DataFrag* se vuelven a ensamblar en el *Reader* RTPS. El siguiente submensaje es el *HeartbeatFrag,* el cual se envía desde un *Writer* RTPS a un *Reader* RTPS, lo que indica cuales fragmentos están disponibles.

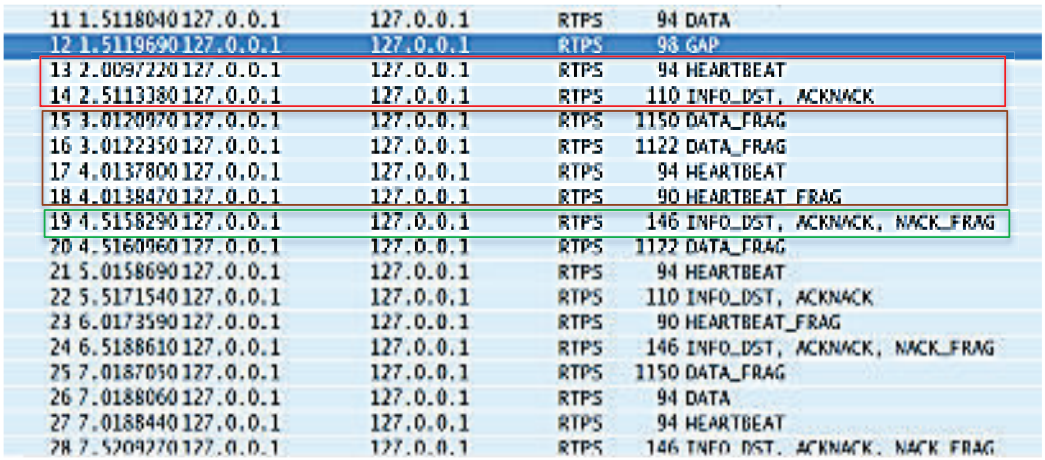


Figura 3.32. Intercambio de Mensajes RTPS Ejemplo 3

Por último en el cuadro verde que representa un grupo de submensajes, se puede observar al *InfoDestination, AckNack* y al *NackFrag.* El submensaje *NackFrag* permite al lector informar al *Writer* acerca del número de fragmentos que se han perdido.

3.3.5.4. Ejemplo 4

En este ejemplo se procede a realizar la misma prueba del ejemplo 1, con la diferencia que en este caso se trabajará con 3 *hosts.*

Como se puede observar en la primera figura, el comportamiento es el mismo que en el ejemplo1, al suscribirse un segundo computador al servicio, como se observa en la segunda imagen, primeramente, el publicador hace un ping al subscriptor, luego le informa con el *GAP*, que los datos que vienen deben mantener un orden. Y el mensaje que antes se enviaba a un sólo *host*, ahora es enviado a dos *hosts*, es decir, se enviarán tantos mensajes como suscriptores hayan.

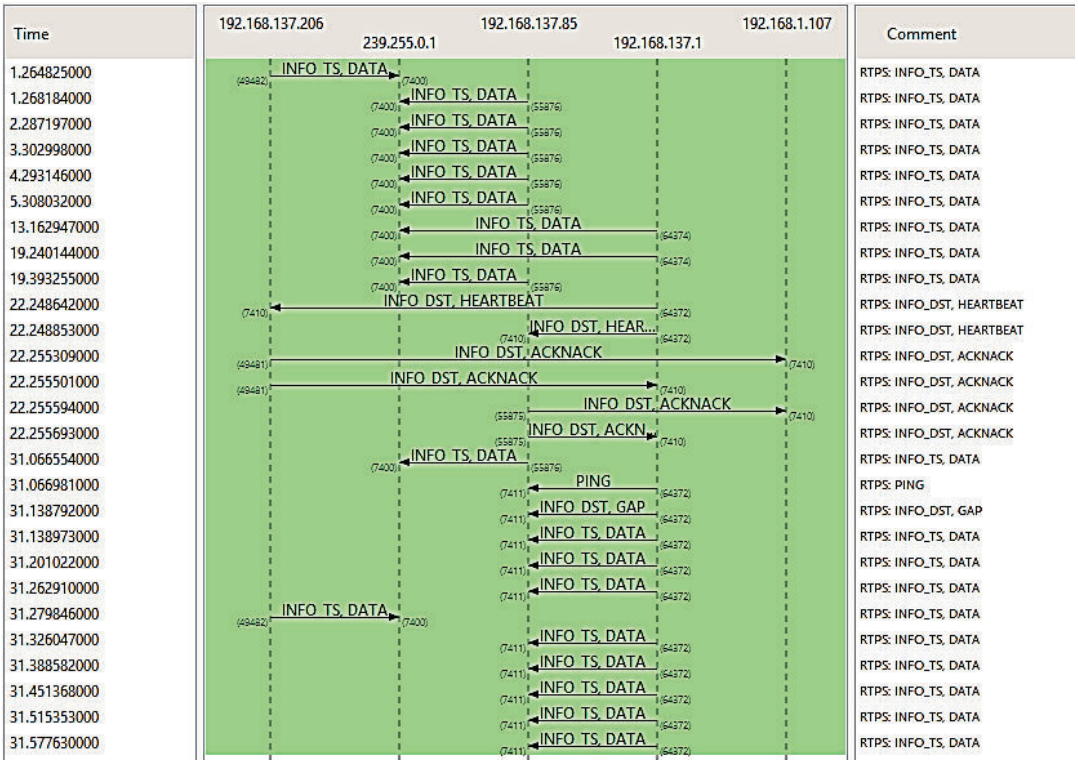


Figura 3.33. Intercambio de Mensajes RTPS Ejemplo 4.1

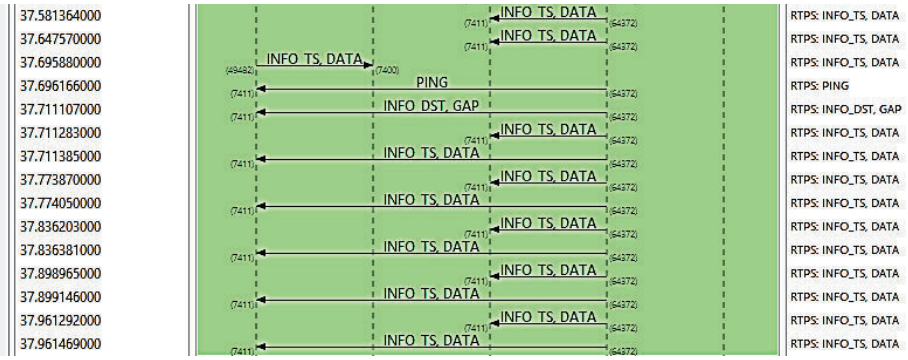


Figura 3.34. Intercambio de Mensajes RTPS Ejemplo 4.2

3.4. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS

3.4.1. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS CON ESTADO

3.4.1.1. Diagramas de secuencia basados en la QoS Best Effort

3.4.1.1.1. Best Effort Reader – Best Effort Writer

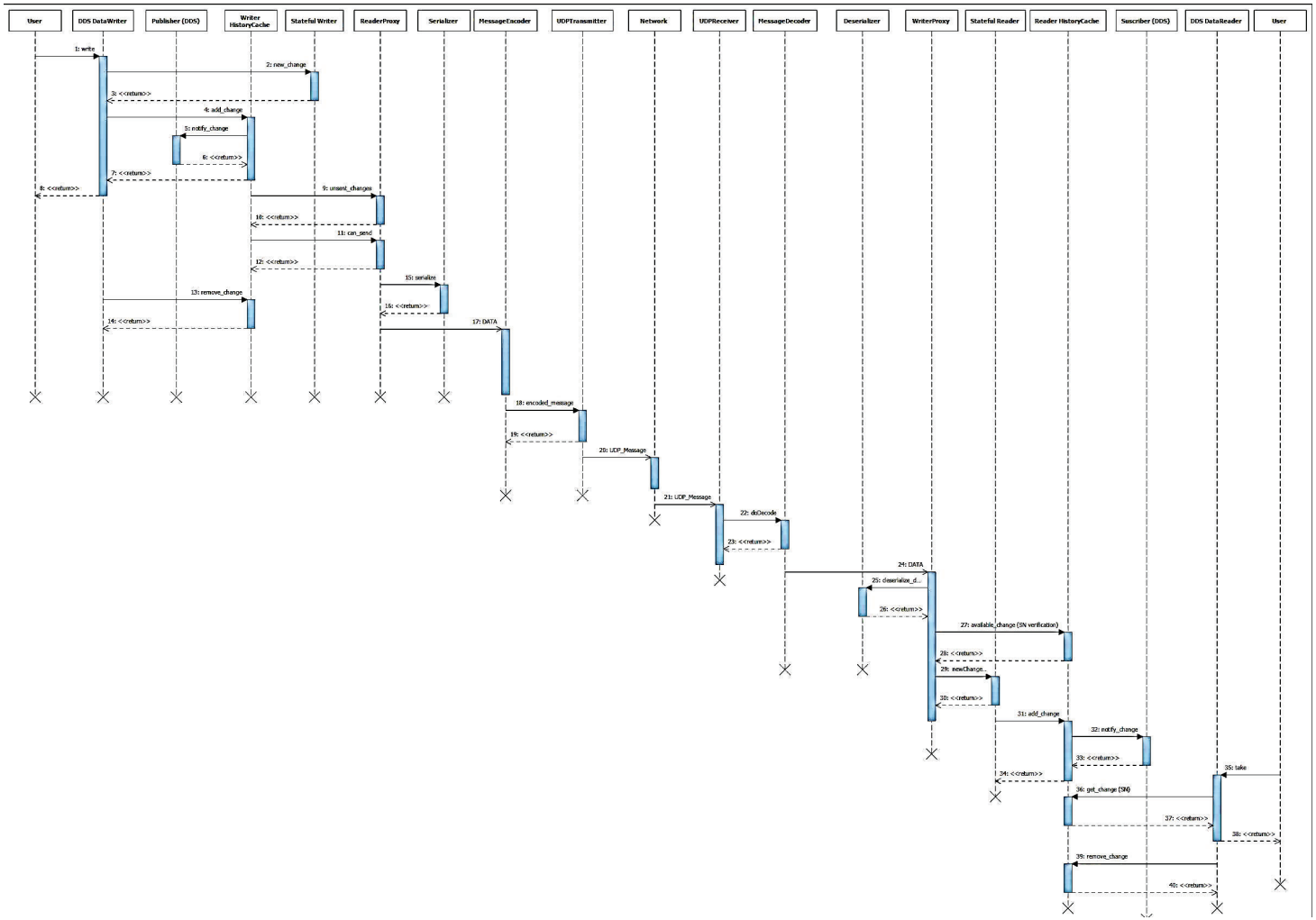


Figura 3.35. Comportamiento Best Effort Reader – Best Effort Writer en interacción con estado.

3.4.1.1.2. Best Effort Reader – BestEffort Writer (Packet Failure)

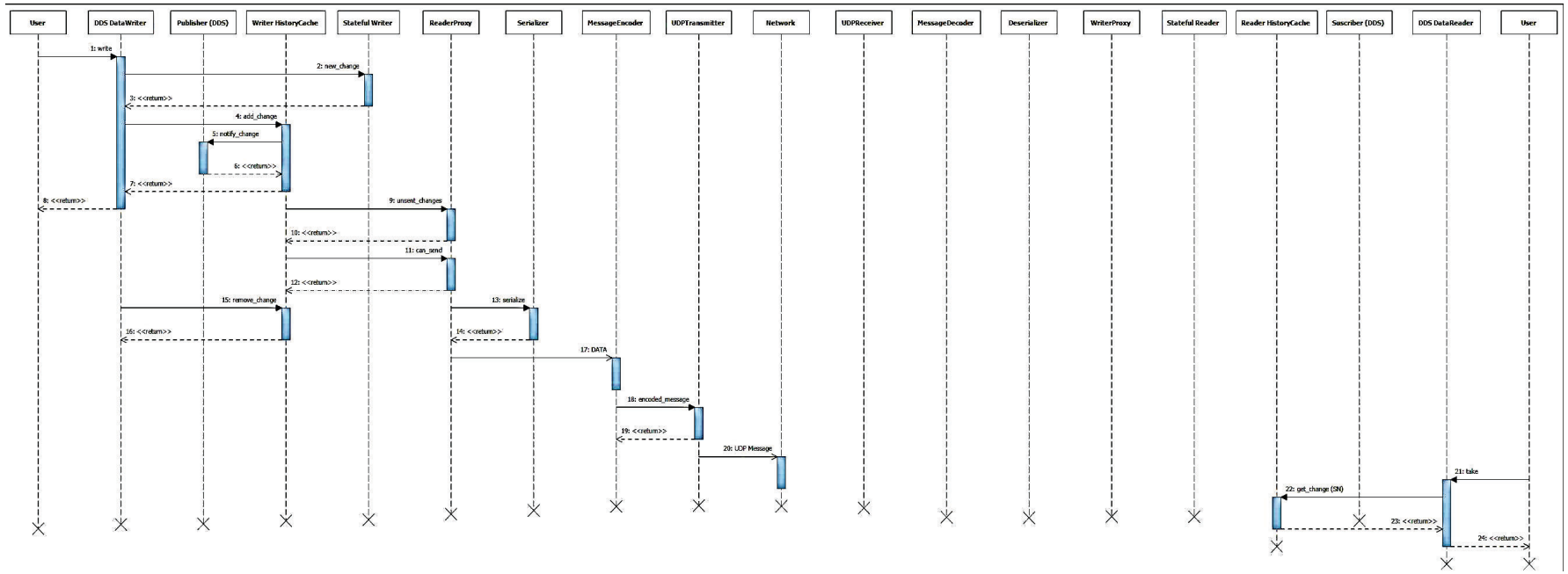


Figura 3.36. Comportamiento Best Effort Reader – Best Effort Writer en interacción con estado con falla de envío de paquete.

3.4.1.2. Diagramas de secuencia basados en la QoS Reliable

3.4.1.2.1. Reliable Reader – Reliable Writer

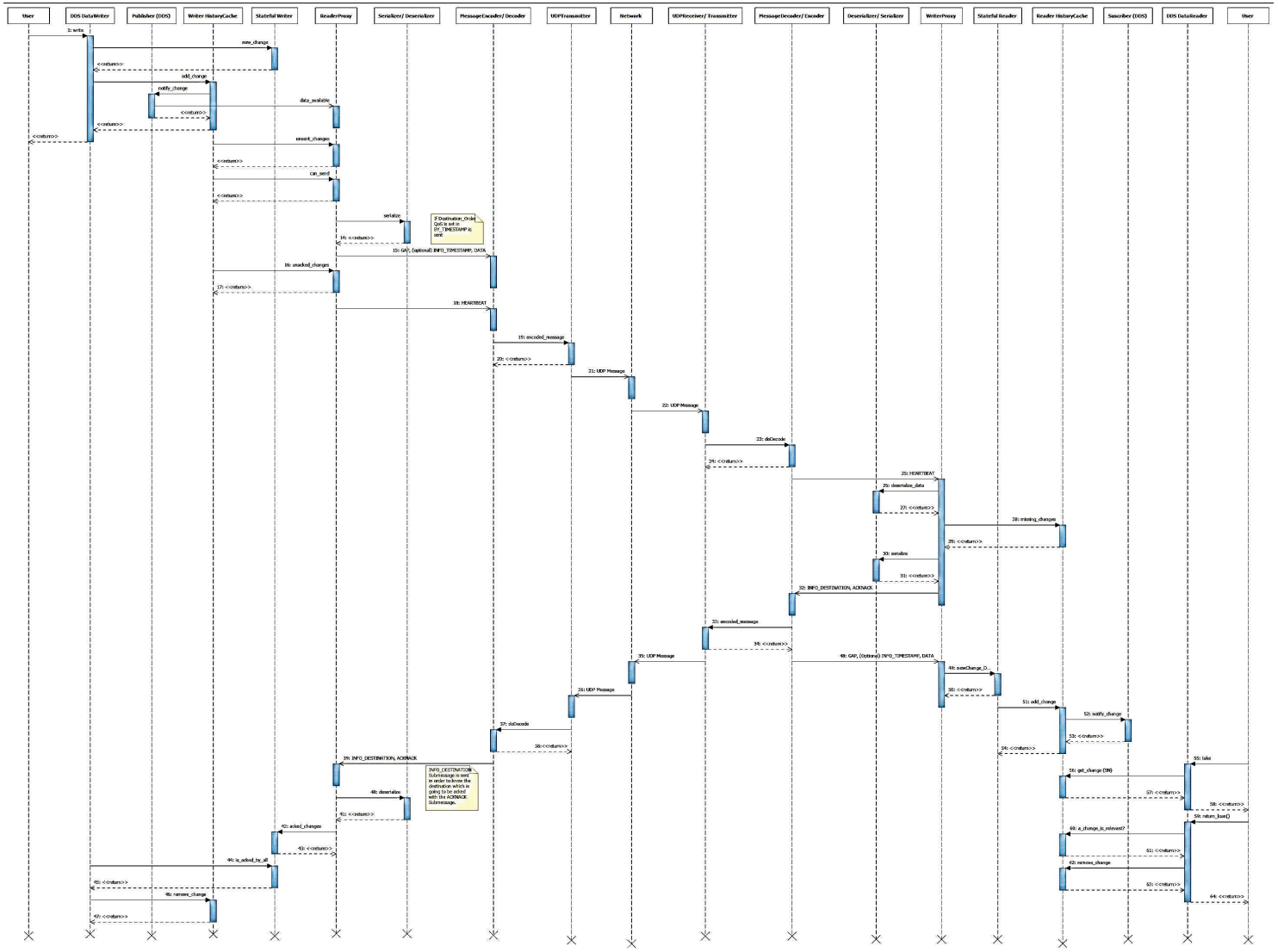


Figura 3.37. Comportamiento Reliable Reader – Reliable Writer en interacción con estado.

3.4.1.2.2. Reliable Reader – Reliable Writer con fragmentación

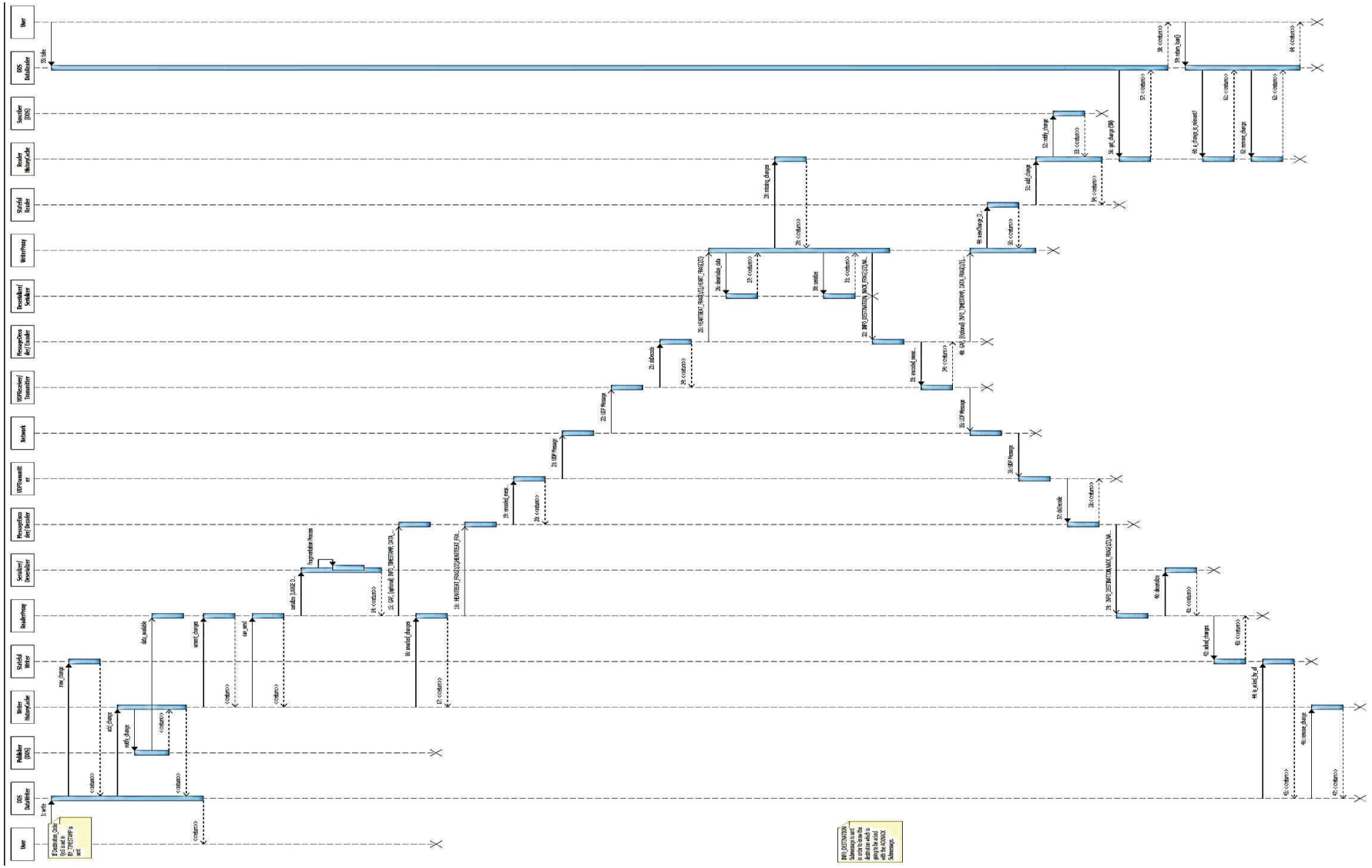


Figura 3.38. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con fragmentación de datos.

3.4.1.2.3. Reliable Reader – Reliable Writer (Communication Error)

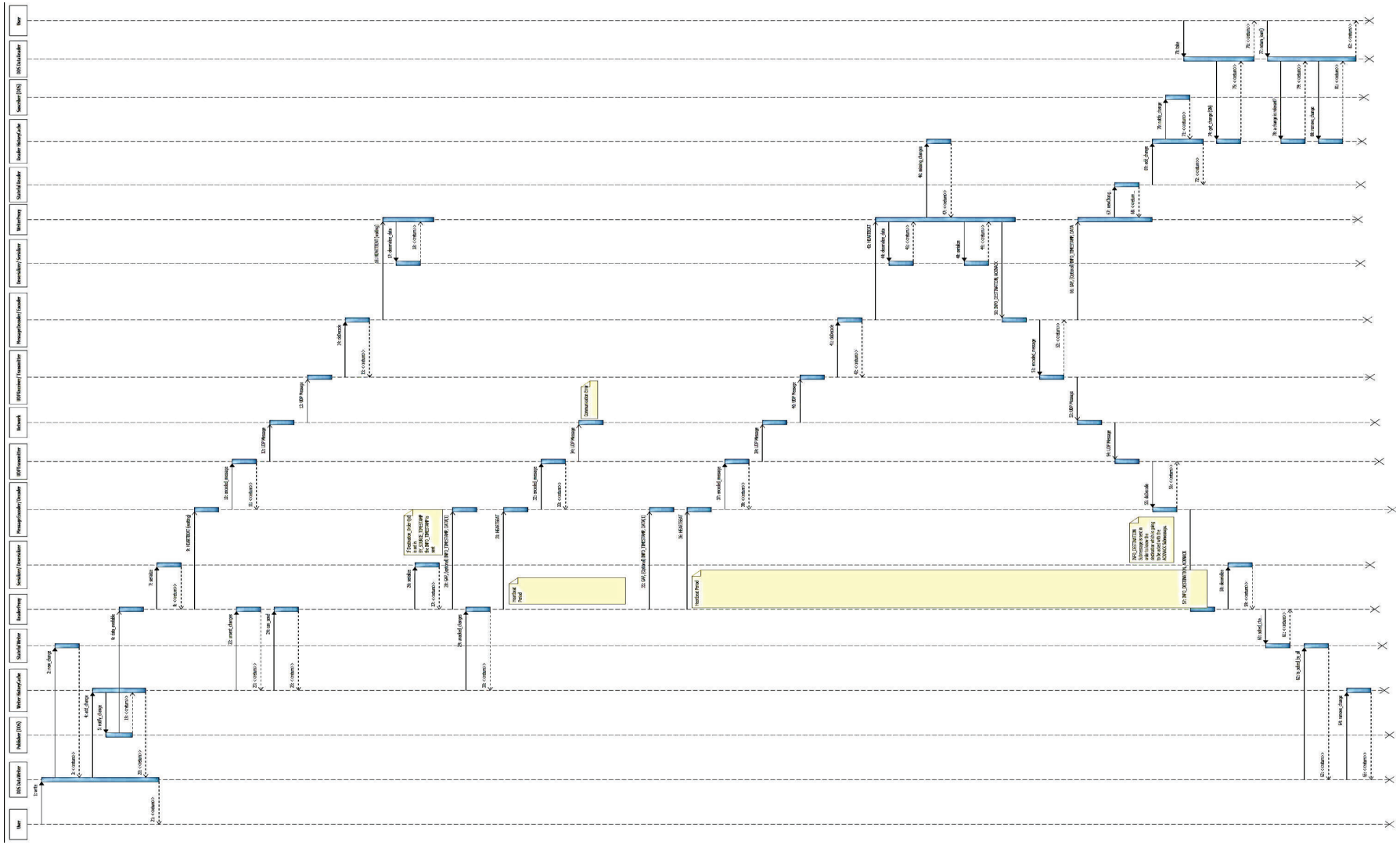


Figura 3.39. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con falla en la comunicación.

3.4.1.2.4. Reliable Reader – Reliable Writer (Packet Failure)

Figura 3.40. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con falla de envío de paquete y con tres participantes.

3.4.1.3. Diagramas de secuencia basados en la combinación de la QoS Reliable – Best Effort

3.4.1.3.1. Reliable Writer—Best Effort Reader

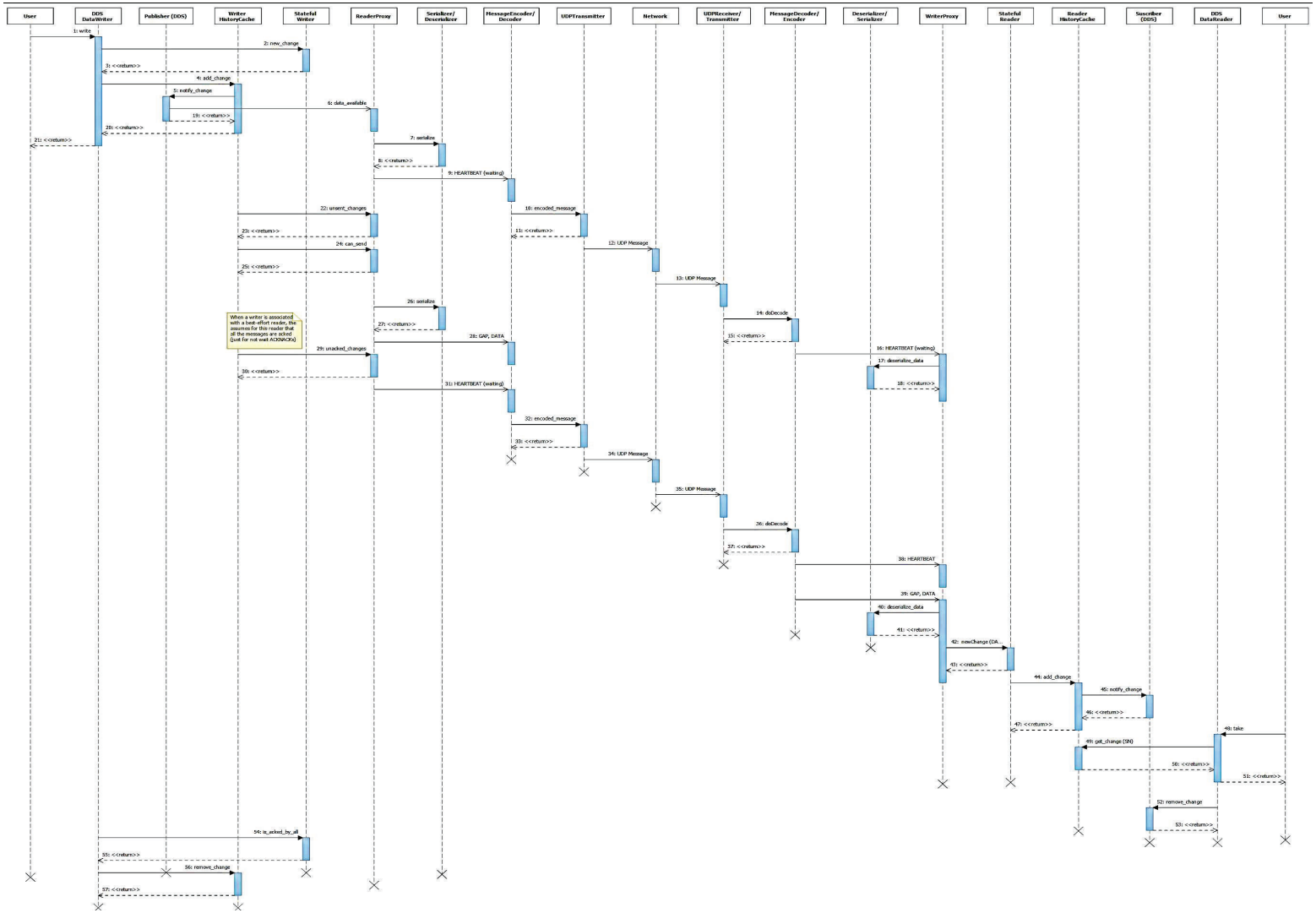


Figura 3.41. Comportamiento Reliable Writer – Best Effort Reader en interacción con estado.

3.4.1.3.2. Reliable Writer—Best Effort Reader (Packet Failure)

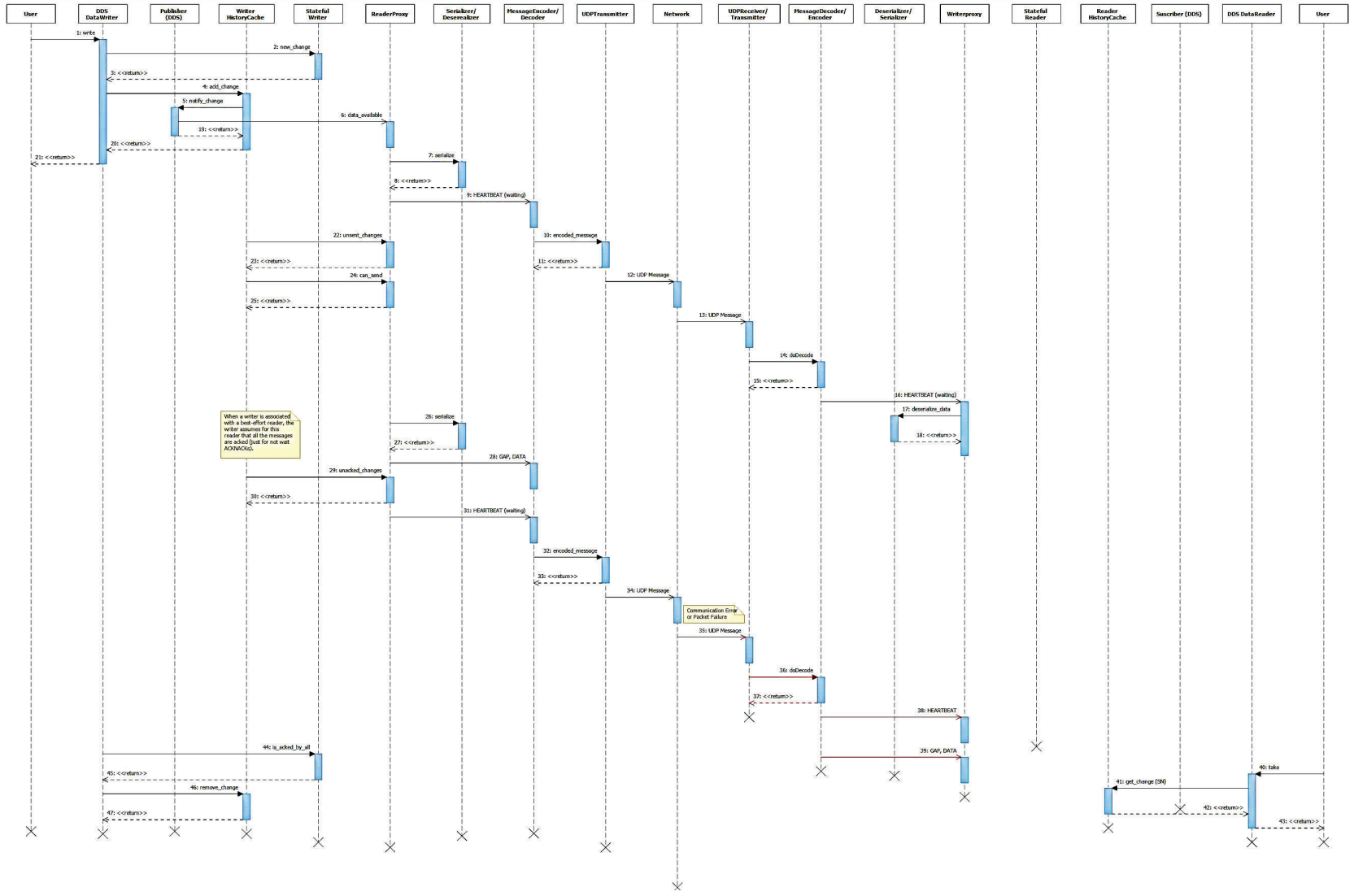


Figura 3.42. Comportamiento Reliable Writer – Best Effort Reader en interacción con estado con falla en el envío de paquetes.

3.4.2. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS SIN ESTADO

3.4.2.1. Diagrama de secuencia basado en la QoS Best Effort

3.4.2.1.1. Best Effort Reader –- Best Effort Writer

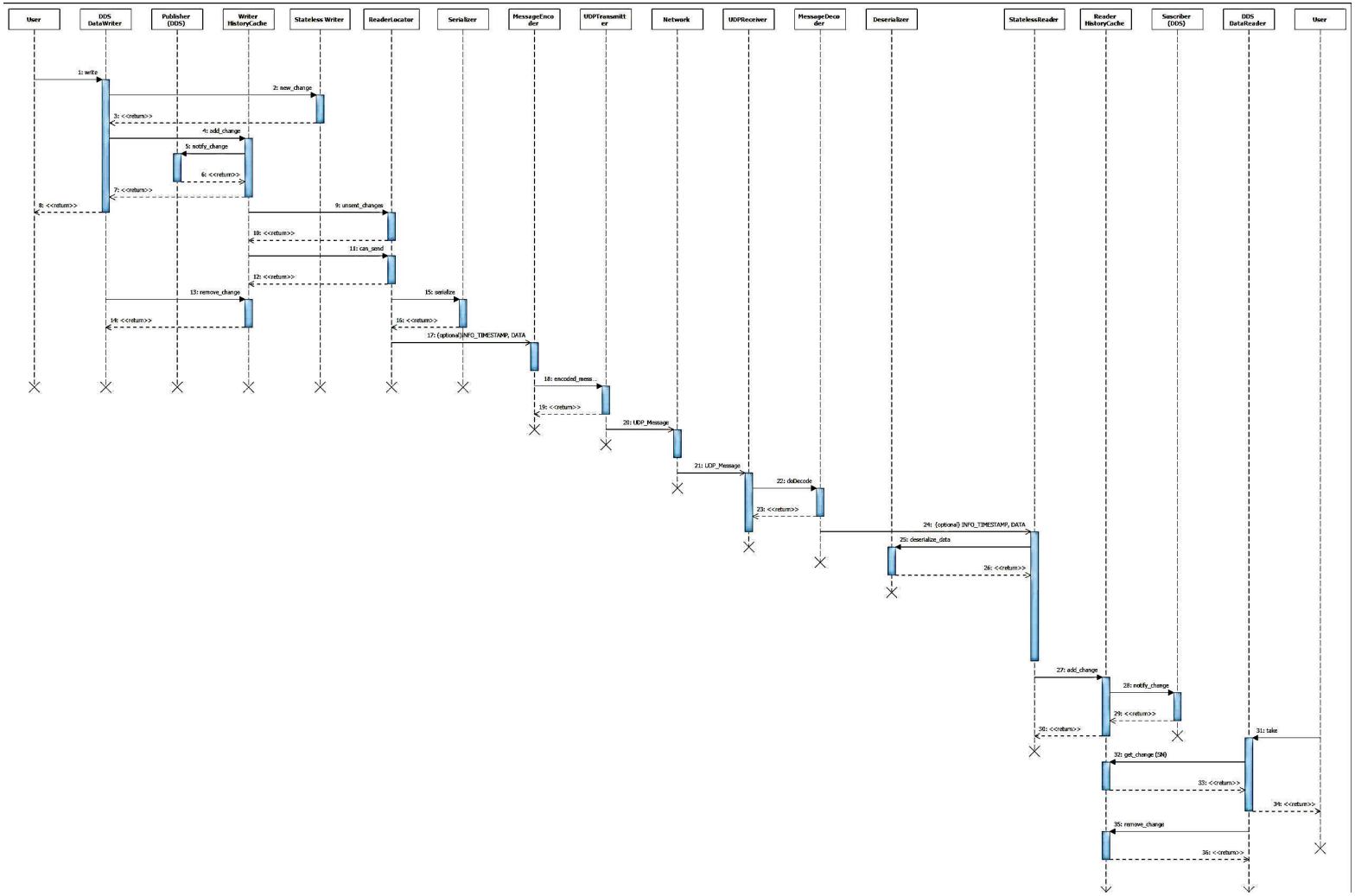


Figura 3.43. Comportamiento Best Effort Reader – Best Effort Writer en interacción sin estado.

3.4.2.2. Diagrama de secuencia basado en la QoS Reliable – Best Effort

3.4.2.2.1. Reliable Writer – Best Effort Reader

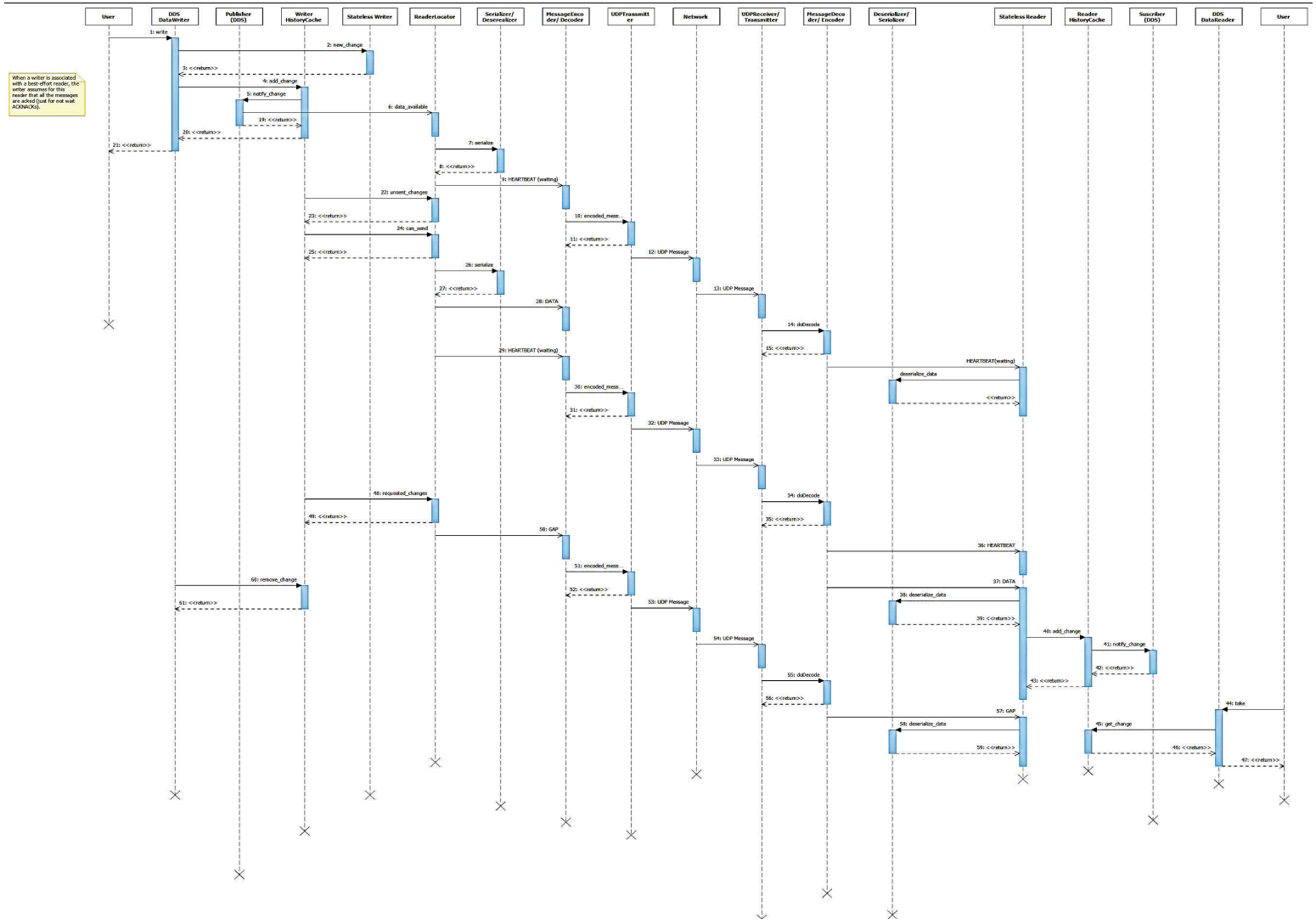


Figura 3.44. Comportamiento Reliable Writer – Best Effort Reader en interacción sin estado.

3.4.3. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS HÍBRIDOS (CON ESTADO Y SIN ESTADO)

3.4.3.1.1. Reliable Stateless Writer – Reliable Stateful Reader

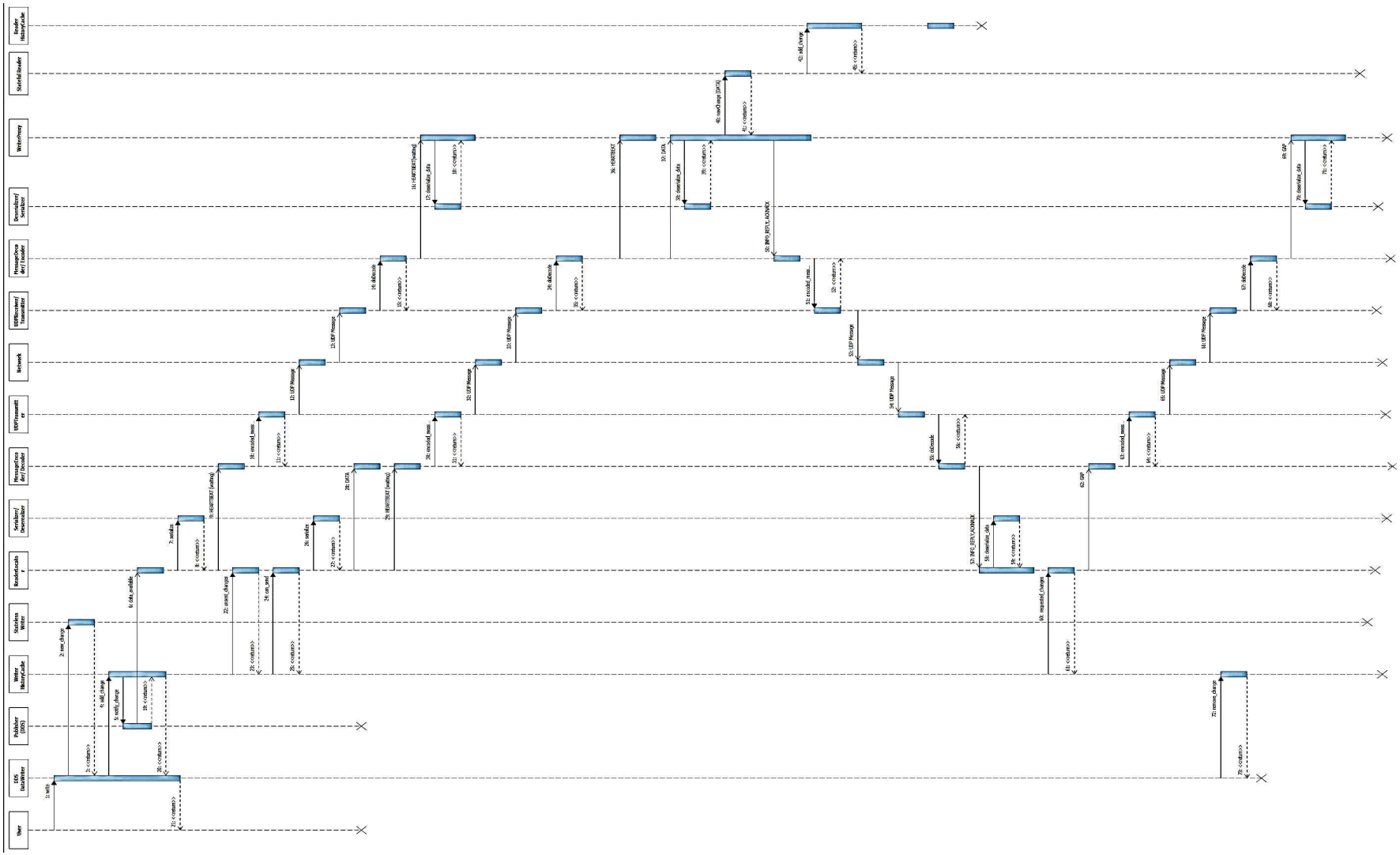


Figura 3.45. Comportamiento Reliable Stateless Writer sin estado – Reliable Stateful Reader con estado.

3.4.4. PROTOCOLO DESCUBRIMIENTO

3.4.4.1. Tráfico de Descubrimiento

3.4.4.1.1. Fase de Descubrimiento de Participantes.

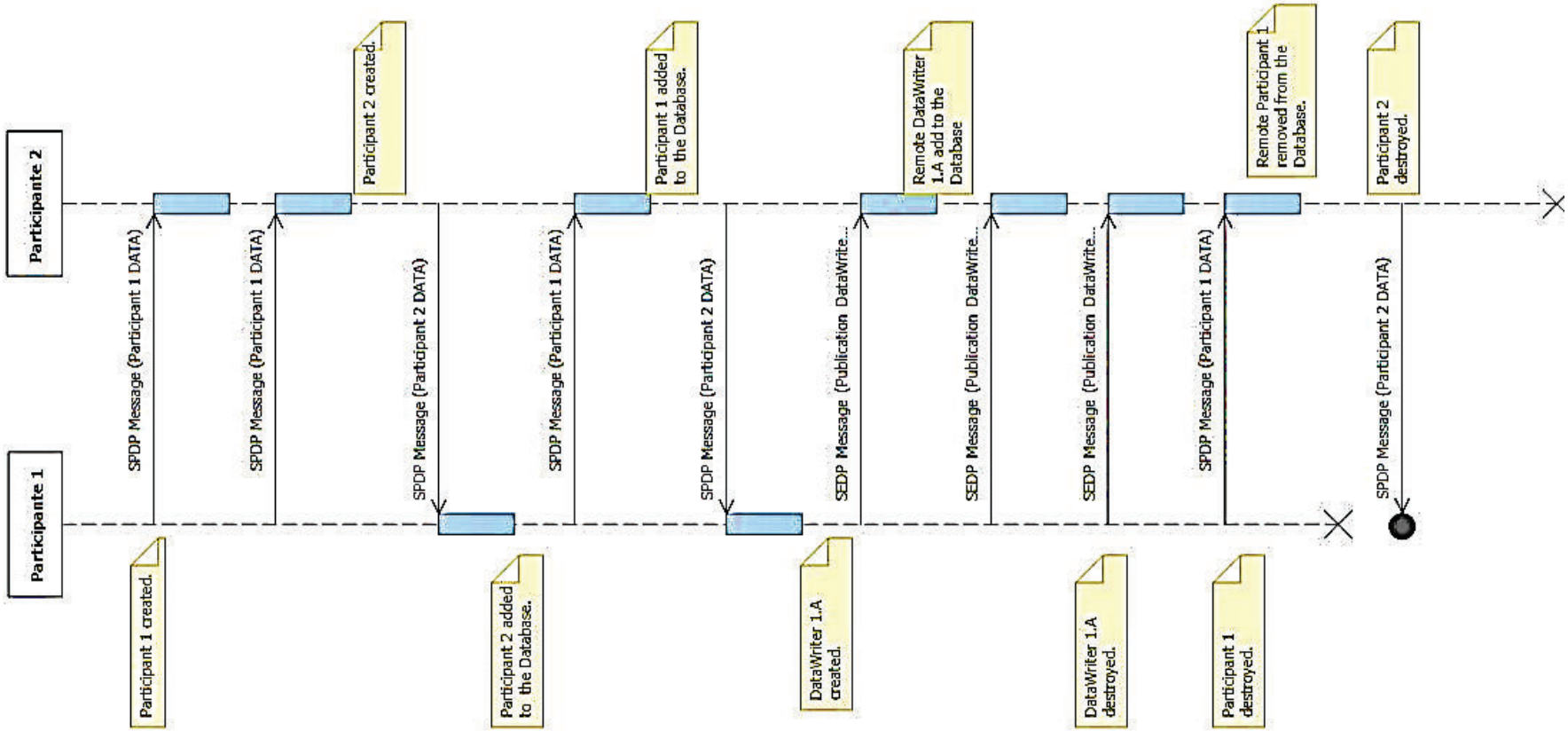


Figura 3.46. Fases de descubrimiento de participantes.

3.5. IMPLEMENTACIÓN DEL MÓDULO

En esta sección se explica el desarrollo de todos los submódulos que componen el sistema, tomando en cuenta los diseños y adaptaciones presentados en la sección 3.2.

3.5.1. SUBMÓDULO DE MENSAJE Y ENCAPSULAMIENTO

3.5.1.1. Implementación de los mensajes RTPS

Para poner mensajes en la red se utiliza un método que permite alinear el mensaje a los 32 bits como define la norma, luego se define el *Endianess* y finalmente el tipo de submensaje a enviar. En el Espacio de Código 3.1 se muestra como el submensaje es alineado a los 32 bits con respecto al inicio del mensaje por medio del manejo del *buffer.*

```
public static int PutSubMessage(this IoBuffer buffer, SubMessage msg)
        {
            // The PSM aligns each Submessage on a 32-bit boundary
with respect to the start of the Message (page 159).
            buffer.Align(4);
            buffer.Order = (msg.Header.IsLittleEndian ?
ByteOrder.LittleEndian : ByteOrder.BigEndian); // Set the endianess
            buffer.PutSubMessageHeader(msg.Header);
            int position = buffer.Position;
            switch (msg.Kind)
            {
                case SubMessageKind.PAD:
                    buffer.PutPad((Pad)msg);
                    break;
                case SubMessageKind.ACKNACK:
                    buffer.PutAckNack((AckNack)msg);
                    break;
```

Espacio de Código 3.1. Implementación mensajes RTPS

3.5.1.2. Implementación de la Encapsulación

Para la encapsulación de los mensajes, se obtiene el mensaje luego de alinearlo y se procede a almacenarlo en el buffer de salida como se muestra en el Espacio de Código 3.2.

```
public void Encode(IoSession session, object message,
IProtocolEncoderOutput output)
        {
            Message msg = (Message)message;
            IoBuffer buffer = IoBuffer.Allocate(1024);
            buffer.AutoExpand = true;
            buffer.PutMessage(msg);
            buffer.Flip();
            output.Write(buffer);
        }
```

Espacio de Código 3.2. Implementación de la Encapsulación.

Ya en el buffer de salida descrito en color rojo dentro del Espacio de Código 3.2, dependiendo de su *Endianess* se serializa la información como se muestra en el Espacio de Código 3.3.

```
public static void Serialize(IoBuffer buffer, object dataObj,
ByteOrder order)
        {
            buffer.Order = order;
            if (order == ByteOrder.LittleEndian)
                buffer.PutEncapsulationScheme(CDR_LE_HEADER);
            else
                buffer.PutEncapsulationScheme(CDR_BE_HEADER);
            Doopec.Serializer.Serializer.Serialize(buffer, dataObj);
        }
```

Espacio de Código 3.3. Implementación Serializador.

3.5.1. SUBMÓDULO DE DESCUBRIMIENTO

3.5.1.1. Implementación del descubrimiento RTPS

Para poder realizar el descubrimiento es necesario primeramente añadir a los participantes, estos tienen asociada una QoS, y se envían dentro de algún protocolo de descubrimiento. Estos participantes deben tener una identificación que es autogenerada en el programa. En el Espacio de Código 3.4 se muestra como se añade un participante al dominio y como se autogenera una identificación al participante.

```
internal virtual AddDomainStatus AddDomainParticipant(int domain,
DomainParticipantQos qos)

        {
            lock (this)
            {
                AddDomainStatus ads = new AddDomainStatus() { id
= new GUID(), federated = false };
                generator.Populate(ref ads.id);
                ads.id.EntityId = EntityId.ENTITYID_PARTICIPANT;

                try
                {
                    if (participants_.ContainsKey(domain) &&
participants_[domain] != null)
                    {
                        participants_[domain][ads.id] = new
Spdp(domain, ads.id, qos, this);

                    }

                    else
                    {
                        participants_[domain] = new
Dictionary<GUID, Spdp>();
```

```
                        participants_[domain][ads.id] = new
Spdp(domain, ads.id, qos, this);
                    }
                }
```

Espacio de Código 3.4. Implementación del Descubrimiento RTPS.

3.5.1.2. Implementación de los protocolos SPDP y SEDP

Tanto para los lectores como para los escritores y sus *topics*, es necesario tener la correspondiente clase que implemente el protocolo SEDP en base a un identificador, tal como se muestra en el Espacio de Código 3.5.

```
namespace Doopec.Rtps.Discovery
{
    public class SEDPbuiltinPublicationsWriter :
StatefulWriter<DiscoveredWriterData>
    {
        public SEDPbuiltinPublicationsWriter(GUID guid)
            : base(guid)
        {
            this.guid = new GUID(guid.Prefix,
EntityId.ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER);
        }

    }
}
```

Espacio de Código 3.5. Implementación SEDP.

Así como para el protocolo SEDP, es necesario tener la correspondiente clase que implemente el protocolo SPDP en base a una configuración de transporte predefinida, la cual configura varios parámetros de configuración de calidad de servicio. En el Espacio de Código 3.6 se implementa el descubrimiento de un *Writer* mediante el protocolo SPDP.

```
public SPDPbuiltinParticipantWriterImpl(Transport
transportconfig, ParticipantImpl participant)
            : base(participant.Guid)
        {
            SetLocatorListFromConfig(transportconfig,
participant);
            participant.DefaultMulticastLocatorList =
this.MulticastLocatorList as List<Locator>;
            participant.DefaultUnicastLocatorList =
this.UnicastLocatorList as List<Locator>;
            SPDPdiscoveredParticipantData data = new
SPDPdiscoveredParticipantData(participant);
            // TODO Assign UserData from configuration
            CacheChange<SPDPdiscoveredParticipantData> change =
this.NewChange(ChangeKind.ALIVE, new Data(data), null);
            this.HistoryCache.AddChange(change);

            this.TopicKind = TopicKind.WITH_KEY;
            this.ReliabilityLevel = ReliabilityKind.BEST_EFFORT;
            this.ResendDataPeriod = new
Duration(transportconfig.Discovery.ResendPeriod.Val);
```

```
            this.heartbeatPeriod = new
Duration(transportconfig.RtpsWriter.HeartbeatPeriod.Val);

            //The following timing-related values are used as the
defaults in order to facilitate
            // 'out-of-the-box' interoperability between
implementations.
            this.nackResponseDelay = new
Duration(transportconfig.RtpsWriter.NackResponseDelay.Val); //200
milliseconds
            this.nackSuppressionDuration = new
Duration(transportconfig.RtpsWriter.NackSuppressionDuration.Val);
            this.pushMode =
transportconfig.RtpsWriter.PushMode.Val;

            InitTransmitters();
            foreach (var trans in this.UDPTransmitters)
            {
                trans.IsDiscovery = true;
            }
            this.Scheme = Encapsulation.PL_CDR_BE;

            worker = new WriterWorker(this.PeriodicWork);
```

Espacio de Código 3.6. Implementación SPDP.

3.5.1. SUBMÓDULO DE COMPORTAMIENTO

Lo que se observa a continuación muestra lo esencial del código de cada *Reader y Writer*.

3.5.1.1. Implementación del *Reader* RTPS con estado

Para la implementación de un lector RTPS con estado, el Espacio de Código 3.7 demuestra la creación de un mensaje, la cual permite escoger el tipo de mensaje, después de almacenar el mensaje en un *buffer*, seguidamente se lo encapsula y serializa. Luego del proceso de creación del mensaje, el *Reader* se encarga de informar al *HistoryCache* que se ha generado un cambio por medio del objeto *CacheChange*.

```
private void NewMessage(object sender, RTPSMessageEventArgs e)
        {
            Message msg = e.Message;
            log.DebugFormat("New Message has arrived from {0}",
e.Session.RemoteEndPoint);
            log.DebugFormat("Message Header: {0}", msg.Header);
            foreach (var submsg in msg.SubMessages)
            {
                switch (submsg.Kind)
                {
                    case SubMessageKind.DATA:
                        Data d = submsg as Data;

                        IoBuffer buf =
IoBuffer.Wrap(d.SerializedPayload.DataEncapsulation.SerializedPayload);
```

```
                        buf.Order = ByteOrder.LittleEndian;
//(d.Header.IsLittleEndian ? ByteOrder.LittleEndian :
ByteOrder.BigEndian);
                        object obj =
Doopec.Serializer.Serializer.Deserialize<T>(buf);
                        CacheChange<T> change = new
CacheChange<T>(ChangeKind.ALIVE, new GUID(msg.Header.GuidPrefix,
d.WriterID), d.WriterSN, new DataObj(obj), new InstanceHandle());
                        ReaderCache.AddChange(change);
                        break;
```

Espacio de Código 3.7. Implementación Reader RTPS con estado.

3.5.1.2. Implementación del *Reader* RTPS sin estado

Una parte esencial de un *Reader* sin estado, es configurar a los receptores, inicializándolos como se muestra en el Espacio de Código 3.8. Esto se realiza de igual manera tanto para tráfico *Unicast* como para tráfico *Multicast.*

```
protected void InitReceivers()
        {
            foreach (var locator in MulticastLocatorList)
            {
                UDPReceiver rec = new UDPReceiver(locator, 1024);
                rec.ParticipantId = this.Guid;
                rec.MessageReceived += NewMessage;
                UDPReceivers.Add(rec);
            }

            foreach (var locator in UnicastLocatorList)
            {
                UDPReceiver rec = new UDPReceiver(locator, 1024);
                rec.ParticipantId = this.Guid;
                rec.MessageReceived += NewMessage;
                UDPReceivers.Add(rec);
            }
        }
```

Espacio de Código 3.8. Implementación del Reader RTPS sin estado.

3.5.1.3. Implementación del *Writer* RTPS con estado

Tanto para los *Writer* con estado y sin estado, es importante la implementación del método *PeriodicWork*, como se muestra en el Espacio de Código 3.9, el cual se encarga de anunciar repetitivamente la disponibilidad de datos por medio del envío de submensajes *Heartbeat.* Con la diferencia que en los escritores sin estado no se guarda más que un cambio.

```
private void PeriodicWork()
        {
            // the RTPS Writer to repeatedly announce the
availability of data by sending a Heartbeat Message.
            log.DebugFormat("I have to send a Heartbeat Message,
at {0}", DateTime.Now);
            SendHeartbeat();
```

```
            if (HistoryCache.Changes.Count > 0)
            {
                foreach (var change in HistoryCache.Changes)
                {
                    //SendHeartbeat();
                    //SendData(change);
                    SendDataHeartbeat(change);
                }
                HistoryCache.Changes.Clear(); //TODO
            }
        }
```

Espacio de Código 3.9. Implementación del Writer RTPS con estado.

3.5.1.4. Implementación del *Writer* RTPS sin estado

Para los *Writer* sin estado aparte de la implementación del método *PeriodicWork*, se implementa un método que permite él envió manual de submensajes *Heartbeat*, como se muestra en el Espacio de Código 3.10.

```
private void SendHeartbeat()
        {
            // Create a Message with Heartbeat
            Message m1 = new Message();

            Heartbeat heartbeat = new Heartbeat();
            EntityId id1 = EntityId.ENTITYID_UNKNOWN;
            EntityId id2 = EntityId.ENTITYID_PARTICIPANT;

            heartbeat.readerID = id1;
            heartbeat.writerID = id2;
            heartbeat.firstSN = new SequenceNumber(10);
            heartbeat.lastSN = new SequenceNumber(20);
            heartbeat.count = 5;
            m1.SubMessages.Add(heartbeat);

            SendData(m1);
```

Espacio de Código 3.10. Implementación del Writer RTPS sin estado.

3.5.1.5. Implementación del Publicador y el *Writer* DDS

3.5.1.5.1. Writer

En el Espacio de Código 3.11, se muestra el constructor de un *DataWriter* DDS, y en su inicialización se especifica si el *Writer* RTPS es con estado o sin estado.

```
public DataWriterImpl(Publisher pub, Topic<TYPE> topic,
DataWriterQos qos, DataWriterListener<TYPE> listener,
ICollection<Type> statuses)
        {
            this.pub_ = pub;
            this.topic_ = topic;
            this.listener = listener;
```

```
            this.rtpsWriter = new
RtpsStatefulWriter<TYPE>((pub.GetParent() as
DomainParticipantImpl).ParticipantGuid);
        }
```

Espacio de Código 3.11. Implementación del Writer DDS.

3.5.1.5.2. Publicador

A continuación en el Espacio de Código 3.12 se muestra la creación de los *DataWriter* DDS que son agregados a una lista de *DataWriter* y la lista es retornada al publicador.

```
public DataWriter<TYPE> CreateDataWriter<TYPE>(Topic<TYPE> topic)
        {
            DataWriter<TYPE> dw = null;
            dw = new DataWriterImpl<TYPE>(this, topic);
            datawriters.Add(dw);
            return dw;
        }
        public DataWriter<TYPE>

CreateDataWriter<TYPE>(Topic<TYPE> topic, DataWriterQos qos,
DataWriterListener<TYPE> listener, ICollection<Type> statuses)
        {
            DataWriter<TYPE> dw = null;
            dw = new DataWriterImpl<TYPE>(this, topic, qos,
listener, statuses);
            datawriters.Add(dw);
            return dw;
        }
```

Espacio de Código 3.12. Implementación del Publicador.

3.5.1.6. Implementación del Suscriptor y del *Reader* DDS

3.5.1.6.1. Reader

En el Espacio de Código 3.13, se muestra el constructor de un *DataReader* DDS, y en su inicialización se especifica si el *Reader* RTPS es *con estado* o sin estado.

```
public DataReaderImpl(Subscriber sub, TopicDescription<TYPE>
topic, DataReaderQos qos, DataReaderListener<TYPE> listener,
ICollection<Type> statuses)
        {
            this.sub_ = sub;
            this.topic_ = topic;
            this.listener = listener;


            RtpsStatefulReader<TYPE> reader = new
RtpsStatefulReader<TYPE>((sub.GetParent() as
DomainParticipantImpl).ParticipantGuid);
            reader.ReaderCache.Changed += NewMessage;
            this.rtpsReader = reader;
        }
```

Espacio de Código 3.13. Implementación del Reader DDS.

3.5.1.6.2. Suscriptor

En el Espacio de Código 3.14, se muestra la creación de los *DataReader* DDS que son agregados a una lista de *DataReader,* y la lista es retornada al suscriptor.

```
public DataReader<TYPE>
CreateDataReader<TYPE>(TopicDescription<TYPE> topic)
        {
            DataReader<TYPE> dw = null;
            dw = new DataReaderImpl<TYPE>(this, topic);
            datawriters.Add(dw);
            return dw;
        }

        public DataReader<TYPE>
CreateDataReader<TYPE>(TopicDescription<TYPE> topic,
DataReaderQos qos, DataReaderListener<TYPE> listener,
ICollection<Type> statuses)


        {
            DataReader<TYPE> dw = null;
            dw = new DataReaderImpl<TYPE>(this, topic, qos,
listener, statuses);
            datawriters.Add(dw);
            return dw;
        }
```

Espacio de Código 3.14. Implementación de Suscriptor.

3.5.2. SUBMÓDULO DE TRANSPORTE

A continuación, se explica el código más relevante de la implementación de este módulo.

3.5.2.1. Implementación del transmisor UDP

Dentro de la implementación del transmisor UDP, el inicio de la comunicación del lado del transmisor constituye parte vital del proyecto.

Se observa que tanto en el transmisor, como en el receptor se utiliza el protocolo que trabaja sobre IP, denominado UDP, el cual sólo es útil por su facilidad de uso, ya que RTPS es un protocolo que trabaja en un nivel superior al de la capa transporte.

A continuación, se observa parte del código para iniciar la comunicación dentro del transmisor.

Dentro del código mostrado a continuación, se inicia la comunicación. Primeramente, se detecta si esta trabaja con direcciones IPv4 o IPv6, y también si se trabaja con tráfico unicast o multicast.

```
public void Start()
        {
            IPEndPoint ep = new IPEndPoint(locator.SocketAddress,
locator.Port);
            bool isMultiCastAddr;
            if (ep.AddressFamily == AddressFamily.InterNetwork)
//IP v4

            {
                byte byteIp = ep.Address.GetAddressBytes()[0];
                isMultiCastAddr = (byteIp >= 224 && byteIp <
240) ? true : false;
            }
            else if (ep.AddressFamily ==
AddressFamily.InterNetworkV6)
            {
                isMultiCastAddr = ep.Address.IsIPv6Multicast;
            }
            else
            {
                throw new NotImplementedException("Address family
not supported yet: " + ep.AddressFamily);
            }
            connector = new AsyncDatagramConnector();
            connector.FilterChain.AddLast("RTPS", new
ProtocolCodecFilter(new MessageCodecFactory()));
```

Espacio de Código 3.15. Inicio de la comunicación con UDP

En el caso que se trabaje con tráfico multicast, se define un objeto que usa un socket[11], donde finalmente se fuerza a asociar la dirección local IP y obtener una sesión multicast.

```
if (isMultiCastAddr)
            {
                // Set the local IP address used by the listener
and the sender to
                // exchange multicast messages.
                connector.DefaultLocalEndPoint = new
IPEndPoint(IPAddress.Any, 0);
                // Define a MulticastOption object specifying the
```

Espacio de Código 3.16. Comunicación Multicast

Dentro del conector para el caso de UDP se hace referencia al asincrónico, para obtener los posibles eventos de entrada o salida, se utiliza el concepto de futuros dentro de C# donde este, es un sustituto de un cálculo que es inicialmente desconocido, pero vuelve a estar disponible en un momento posterior [16].

[11] Socket, Asociación de una dirección IP con un Puerto

```
                IoSession session =
connector.Connect(ep).Await().Session;
            }
            connector.ExceptionCaught += (s, e) =>
            {
                log.Error(e.Exception);
            };
            connector.MessageReceived += (s, e) =>
            {
                log.Debug("Session recv...");
            };
            connector.MessageSent += (s, e) =>
            {
                log.Debug("Session sent...");
            };
            connector.SessionCreated += (s, e) =>
            {
                log.Debug("Session created...");
            };

            connector.SessionOpened += (s, e) =>
            {
                log.Debug("Session opened...");
            };

            connector.SessionClosed += (s, e) =>
            {
                log.Debug("Session closed...");
            };

            connector.SessionIdle += (s, e) =>
            {
                log.Debug("Session idle...");
            };

            IConnectFuture connFuture = connector.Connect(new
IPEndPoint(locator.SocketAddress, locator.Port));
            connFuture.Await();

            connFuture.Complete += (s, e) =>
            {
                IConnectFuture f = (IConnectFuture)e.Future;
                if (f.Connected)
                {
                    log.Debug("...connected");
                    session = f.Session;
                }
                else
                {
                    log.Warn("Not connected...exiting");
                }
            };
        }
```

Espacio de Código 3.17. Conexión mediante futuros.

Como se puede observar en el Espacio de Código 3.17, existen muchas expresiones lambda, las cuales se definen como funciones anónimas [17], las cuales son muy útiles a la hora de utilizar futuros.

3.5.2.2. Implementación del receptor UDP

El código mostrado a continuación hace referencia a la definición de un objeto UDP *Receiver*, en el cual se puede hacer hincapié en que se utiliza un *buffer* para almacenar los mensajes recibidos. También se puede encontrar dentro del código del programa al transmisor.

```
public UDPReceiver(Uri uri, int bufferSize)
        {
            this.bufferSize = bufferSize;
            var addresses =
System.Net.Dns.GetHostAddresses(uri.Host);
            int port = (uri.Port < 0 ? 0 : uri.Port);
            if (addresses != null && addresses.Length >= 1)
                this.locator = new Locator(addresses[0], port);
        }
```

Espacio de Código 3.18. Receiver Buffer.

3.5.2.3. Implementación de la maquinaria RTPS

La maquinaria RTPS, se refiere a toda la configuración de perfiles necesarios para trabajar con RTPS, para lo cual primeramente se obtiene del archivo de configuración las instancias, es decir, los parámetros del funcionamiento del protocolo, donde se encuentran tiempos, retardos y QoS.

```
public static IRtpsEngine CreateEngine(IDictionary<string,
Object> environment)
        {
            DDSConfigurationSection ddsConfig =
Doopec.Configuration.DDSConfigurationSection.Instance;

            RTPSConfigurationSection rtpsConfig =
Doopec.Configuration.RTPSConfigurationSection.Instance;

            string transportProfile =
ddsConfig.Domains[0].TransportProfile.Name;

            string className =
rtpsConfig.Transports[transportProfile].Type;
            if (string.IsNullOrWhiteSpace(className))
            {
                // no implementation class name specified
                throw new ApplicationException("Please Set the
RTPS engine type property in the settings.");

            }
            Type ctxClass = Type.GetType(className, true);
```

Espacio de Código 3.19. Implementación maquinaria.

Finalmente luego de configurar el RTPS, se llama a la instancia creada por medio de una interfaz.

```
// --- Instantiate new object --- //
            try
            {
                // First, try a constructor that will accept the
environment.
                object newInstance =
Activator.CreateInstance(ctxClass, environment);
                if (newInstance != null)
                    return (IRtpsEngine)newInstance;
```

Espacio de Código 3.20. Interfaz RTPS Engine.

3.5.3. SUBMÓDULO CONFIGURACIÓN

3.5.3.1. Sección DDS

3.5.3.1.1. Etiqueta DDS

```
<DDS xmlns="urn:Configuration" vendor="Doopec" version="2.1">
```

Espacio de Código 3.21. Etiqueta DDS.

Como se observa en el Espacio de Código 3.21 la etiqueta DDS, necesita tener establecido el *namespace xml* o *xmlns*. Además, se define el fabricante dentro del *vendor* y finalmente se define la versión con la que se trabaja.

Etiqueta Boostrap

Como se observa en el Espacio de Código 3.22, la etiqueta *BoostrapType*, debe tener configurada los atributos *name y type*.

```
<boostrapType name="default" type="Doopec.Dds.Core.BootstrapImpl,
Doopec"/>
```

Espacio de Código 3.22. Etiqueta Boostrap.

Etiqueta Domains

Como se observa en el Espacio de Código 3.23 la etiqueta Domains, anuncia a los participantes presentes, y con cuales se podrá interactuar por medio del protocolo RTPS, en este caso se puede observar a 3 *domain* configurados, cada uno de estos tiene dentro de su etiqueta *domain*, el atributo *name*, es decir, un nombre indistinto, y un atributo *id*, para la identificación única del participante

Etiqueta Transport Profile

Esta etiqueta define el perfil de transporte de información, con el cual trabaja el DDS, para nuestro caso será *defaultRTPS*, es decir, que se utiliza el protocolo RTPS como transporte.

Etiqueta QoS Profile

Esta etiqueta definirá el perfil de calidad de servicio, con el cual trabaja DDS, para nuestro caso será *defaultQoS,* es decir, que existirá un perfil de calidad de servicio por defecto.

Etiqueta Guid

Para la etiqueta Guid, se tienen 5 posibles valores a tomar:

- *Fixed*
- *Random*
- *AutoId*
- *AutoIdFromIp*
- *AutoIdFromMac*

```
<domains>
        <domain name="Servidor" id="0">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
      </domain>
      <domain name="Servidor" id="3">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
      </domain>
      <domain name="Cliente1" id="1">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
      </domain>
      <domain name="Cliente2" id="2">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
      </domain>
    </domains>
```

Espacio de Código 3.23. Etiqueta Domains.

Etiqueta logLevel

Como se observa en el Espacio de Código 3.24 la etiqueta *logLevel*, define un nivel mínimo y un máximo de *logs* con los siguientes posibles valores:

- *DEBUG*
- *ALL*
- *WARN*

- *INFO*
- *ERROR*
- *FATAL*
- *OFF*

```
<logLevel levelMin="DEBUG" levelMax="FATAL"/>
```

Espacio de Código 3.24. Etiqueta LogLevel.

Etiqueta QoS Profiles

QoS Profiles, está compuesto, de un *QosProfileDef*, el cual tiene un *domainParticipantFactory*, un *domainParticipantQoS*, un *topicQoS*, un *PublisherQoS*, un *suscriberQoS*, un *DataWriterQos*, y un *DataReaderQoS*.

Etiqueta domainParticipantFactoryQos

En el Espacio de Código 3.25, además de asignar un nombre a esta etiqueta, tiene una etiqueta interna *autoenableCreatedEntities*, la cual define si se auto habilita a los participantes creados.

```
<qoSProfiles>
      <qoSProfileDef name="defaultQoS">
        <domainParticipantFactoryQos
name="defaultDomainParticipantFactoryQoS">
          <entityFactory autoenableCreatedEntities="true"/>
        </domainParticipantFactoryQos>
```

Espacio de Código 3.25. Etiqueta domainParticipantFactoryQoS.

Etiqueta domainParticipantQos

En el **¡Error! No se encuentra el origen de la referencia.**, además de signar un nombre a esta etiqueta, se tiene una etiqueta interna *autoenableCreatedEntity*, la cual define si se auto habilita a los participantes creados, y tiene una etiqueta *userData*, en la cual se puede poner valores de acuerdo a la política *userDataQoS*.

```
<domainParticipantQos name="defaultDomainParticipantQoS">
          <entityFactory autoenableCreatedEntities="true"/>
          <userData  value=""/>
        </domainParticipantQos>
```

Espacio de Código 3.26. Etiqueta domainParticipantQoS.

Etiqueta topicQos

En el Espacio de Código 3.27, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna llamada *topicData*, a la cual se le puede asignar un valor, aparte se puede agregar varias políticas de QoS, como en este caso *deadline* y *durability*.

```
<topicQoS name="defaultTopicQoS">
          <topicData value=""/>
          <deadline  period="100"/>
          <durability  kind="VOLATILE"/>
        </topicQoS>
```

Espacio de Código 3.27. Etiqueta topicQoS.

Etiqueta PublisherQos

En el Espacio de Código 3.28, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna llamada *entityFactory*, a la cual se le puede asignar un valor, aparte se puede agregar varias políticas de QoS, como en este caso *groupData*, *partition*, y *presentation*.

```
<publisherQoS name="defaultPublisherQoS">
          <entityFactory autoenableCreatedEntities="true"/>
          <groupData value=""/>
          <partition value=""/>

          <presentation accessScope="INSTANCE" coherentAccess="true"
orderedAccess="true"/>
        </publisherQoS>
```

Espacio de Código 3.28. Etiqueta PublisherQoS.

Etiqueta suscriberQos

En el Espacio de Código 3.29, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna llamada *entityFactory*, a la cual se le puede asignar un valor, aparte se puede agregar varias políticas de QoS, como en este caso *groupData*, *partition*, y *presentation*.

```
<subscriberQoS name="defaultSubscriberQoS">
          <entityFactory autoenableCreatedEntities="true"/>
          <groupData value=""/>
          <partition value=""/>
          <presentation accessScope="INSTANCE" coherentAccess="true"
orderedAccess="true"/>
        </subscriberQoS>
```

Espacio de Código 3.29. Etiqueta suscriberQoS.

Etiqueta DataWriterQos

En el Espacio de Código 3.30, además de asignar un nombre a esta etiqueta, se puede agregar todas políticas de QoS utilizadas en un escritor.

```
<dataWriterQoS name="defaultDataWriterQoS">
          <deadline period="1"/>
          <destinationOrder  kind="BY_SOURCE_TIMESTAMP"/>
          <durability  kind="VOLATILE"/>
          <durabilityService historyDepth="0" historyKind="KEEP_LAST"
maxInstances="1" maxSamples="1" maxSamplesPerInstance="1"
serviceCleanupDelay="100"/>
          <history kind="KEEP_LAST" depth="1"/>
          <latencyBudget duration="100"/>
          <lifespan duration="100"/>
          <liveliness kind="AUTOMATIC" leaseDuration="100"/>
          <ownership kind="SHARED"/>
          <ownershipStrength value="100"/>
          <reliability kind="BEST_EFFORT" maxBlockingTime="1000"/>
          <resourceLimits maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1"/>
          <transportPriority value="1"/>
          <userData value=""/>
          <writerDataLifecycle
autodisposeUnregisteredInstances="true"/>
        </dataWriterQoS>
```

Espacio de Código 3.30. Etiqueta DataWriterQoS.

Etiqueta DataReaderQos

En el Espacio de Código 3.31, además de asignar un nombre a esta etiqueta, se puede agregar todas políticas de QoS utilizadas en un lector.

```
<dataReaderQoS name="defaultDataReaderQoS">
          <deadline period="1"/>
          <destinationOrder  kind="BY_SOURCE_TIMESTAMP"/>
          <durability  kind="VOLATILE"/>
          <history kind="KEEP_LAST" depth="1"/>
          <latencyBudget duration="100"/>
          <liveliness kind="AUTOMATIC" leaseDuration="100"/>
          <ownership kind="SHARED"/>
          <reliability kind="BEST_EFFORT" maxBlockingTime="1000"/>
          <resourceLimits maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1"/>
          <readerDataLifecycle autopurgeDisposedSamplesDelay="1000"
autopurgeNowriterSamplesDelay="1000"/>
          <timeBasedFilter minimumSeparation="1000"/>
          <userData value=""/>
        </dataReaderQoS>
      </qoSProfileDef>
    </qoSProfiles>
  </DDS>
```

Espacio de Código 3.31. Etiqueta DataReaderQoS.

3.5.3.2. Sección RTPS

3.5.3.2.1. Etiqueta RTPS

```
<RTPS xmlns="urn:Configuration">
```

Espacio de Código 3.32. Etiqueta RTPS.

Como se observa en el Espacio de Código 3.32 la etiqueta RTPS, necesita tener establecido el *namespace xml o xmlns*.

Etiqueta Transports

La etiqueta *transport* trae los atributos *name* y *type*, el segundo especifica el Motor RTPS, y además hay una etiqueta *ttl*, la cual cumple las funciones de *time to live.*

```
<transports>
      <transport name="defaultRtps"
type="Doopec.Rtps.RtpsTransport.RtpsEngine, Doopec">
        <ttl val="1"/>
```

Espacio de Código 3.33. Etiqueta transport.

Etiqueta Discovery

En el Espacio de Código 3.34 la etiqueta Discovery introduce la configuración de los paquetes de descubrimiento, aquí se puede configurar el periodo de reenvío, además se define si se usan paquetes multicast del tipo SEDP, el puerto base con el que se trabaja, el domainGain y el participantGain; también se configura *offsets* tanto para tráfico unicast y multicast, y se define una lista de IP con las que se trabaja tanto en modo multicast y unicast.

```
<discovery name="defaultDiscovery">
          <resendPeriod val="30000"/>
          <useSedpMulticast val="true"/>
          <portBase val="7400"/>
          <domainGain val="250"/>
          <participantGain val="2"/>
          <offsetMetatrafficMulticast val="0"/>
          <offsetMetatrafficUnicast val="10"/>
          <metatrafficUnicastLocatorList val="localhost"/>
          <metatrafficMulticastLocatorList val="239.255.0.1"/>
        </discovery>
```

Espacio de Código 3.34. Etiqueta Discovery.

Etiqueta rtpsWriter

En el Espacio de Código 3.35 la etiqueta *rtpsWriter* introduce la configuración de los *Writer* RTPS y su comportamiento, aquí se puede configurar el periodo de envío de los submensajes *Heartbeat*, se define los retardos de las respuestas por

medio de los submensajes *Nack*, también el tiempo para que se suprima una respuesta *Nack*, y si estamos trabajando con modo *push*.

```
<rtpsWriter>
          <heartbeatPeriod val="1000"/>
          <nackResponseDelay val="200"/>
          <nackSuppressionDuration val="0"/>
          <pushMode val="true"/>
        </rtpsWriter>
```

Espacio de Código 3.35. Etiqueta rtpsWriter.

Etiqueta rtpsReader

En el Espacio de Código 3.36 la etiqueta *rtpsReader* introduce la configuración de los *Reader* RTPS y su comportamiento, se define los retardos de las respuestas por medio de los submensajes *Heartbeat*, también el tiempo para que se suprima una respuesta *Heartbeat*.

```
        <rtpsReader>
          <heartbeatResponseDelay val="500"/>
          <heartbeatSuppressionDuration val="0"/>
        </rtpsReader>
      </transport>
    </transports>
  </RTPS>
  <appSettings>
    <add key="org.omg.dds.serviceClassName"
value="Doopec.Dds.Core.BootstrapImpl, Doopec" />
  </appSettings>
  <startup>
    <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5.1" />
  </startup>
</configuration>
```

Espacio de Código 3.36. Etiqueta rtpsReader.

CAPÍTULO 4.

PRUEBAS

4.1. INTRODUCCIÓN

En este capítulo primeramente se realizarán pruebas unitarias para la comprobación de distintos componentes del programa tales como clases y métodos, las cuales estarán documentadas mediante una estructura de tablas donde se incluye la llamada, la descripción, la entrada, la referencia, el código y la salida.

A continuación se describirá el ambiente de pruebas, el cual tendrá como base varios computadores comunicándose entre sí por medio del protocolo RTPS y además se realizará una prueba con el protocolo RT-CORBA o Ada-DSA o DRTSJ, esta prueba obtiene los paquetes correspondientes al protocolo y verifica el tiempo de respuesta del mismo. Además se adjunta capturas de pantalla tanto de la aplicación utilizando los protocolos y de capturas del flujo de datos con la herramienta WireShark, y se presenta un manual de usuario de las aplicaciones y del protocolo RTPS.

Finalmente, se realizará una comparación midiendo tiempos de respuesta y eficiencia del protocolo dentro de nuestro ambiente de pruebas.

4.2. PRUEBAS UNITARIAS DE LA IMPLEMENTACIÓN

Las pruebas unitarias consisten en verificar que dos datos son iguales. El primer dato corresponde a un valor esperado el cual arbitrariamente se lo asigna. El segundo dato correponde al valor calculado por un método y este deberá ser igual al primer dato para que la prueba sea superada.

El objetivo de realizar las pruebas unitarias es probar que los métodos se encuentran funcionando adecuadamente.

Para realizar estas pruebas dentro de un proyecto en visual studio se cargo las librerías generadas, es decir, la librería del DDS y del RTPS, para luego probar el funcionamiento de los métodos y clases.

Todo el stack de pruebas unitarias se encuentra descrito en los anexos.

4.2.1. CODIFICADORES

4.2.1.1. Prueba de los Elementos de los Mensajes.

En estas pruebas se verifica el funcionamiento del serializador y deserializador de los elementos del módulo de mensajes y encapsulación; los

elementos a comprobar son los siguientes: *Locator, ClassWithLocator, EntityId, ProtocolVersion, ClassWithProtocolVersion.*

- *Locator,* representa la información de la ubicación del *endpoint* RTPS para el envío de los mensajes usando una dirección IPv4 y un puerto.
- *ClassWithLocator,* representa la información de la ubicación del *endpoint* RTPS para el envío de los mensajes usando una dirección IPv6 y un puerto.
- *EntityId,* representa al identificador de cada entidad a la cual se envía los mensajes.
- *ProtocolVersion,* representa la versión del protocolo RTPS.
- ClassWithProtocolVersion, es una clase que presenta a la versión del protocolo.

Tabla 4.1. TestLocatorIpV4CDR_BE

Llamada: `public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE)` `public static DataEncapsulation Deserialize(IoBuffer buffer, int length)`	
Descripción	En esta prueba se verifica los *Locator* con el *Serializador* y el *Deserealizador.*
Entrada	Inicialmente no existe ninguna instancia del *DataEncapsulation.*

Código	`[TestMethod]`
	`public void TestLocatorIpV4CDR_BE()`
	`{`
	`Encapsulation Scheme = Encapsulation.CDR_BE;`
	`int bufferSize = 16 + 4 + 4 + CDRHeaderSize;`
	
	`Locator v1 = new Locator(IPAddress.Parse("10.20.30.40"), 2700);`
	`SerializedPayload payload = new SerializedPayload();`
	`payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme);`
	`IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload);`
	`Assert.AreEqual(bufferSize, buffer.Remaining);`
	
	`Assert.AreEqual("00 00 00 00 00 00 00 01 00 00 0A 8C 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28", buffer.GetHexDump());`
	`Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer);`
	
	`Assert.AreEqual(v1, v2);`
	`}`

Salida	Nombre de la prueba: *TestLocatorIpV4CDR_BE*

Tabla 4.1. TestLocatorIpV4CDR_BE

	Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:02.9120332

4.2.1.2. Prueba de Mensajes

En estas pruebas se verifica el funcionamiento de todos los submensajes, comprobando que se pueda escribir y leer el submensaje correctamente y verificando que no haya pérdida de información.

Tabla 4.2. TestInfoDestination

Llamada: `public InfoDestination(GuidPrefix guidPrefix)`	
Descripción	En esta prueba se verifica el submensaje *InfoDestination.*
Entrada	Inicialmente no existe ninguna instancia del *InfoDestination.*

Código	`[TestMethod]`
	`public void TestInfoDestination()`
	`{`
	`// Create a Message with InfoDestination`
	`Message m1 = new Message();`
	`m1.SubMessages.Add(new InfoDestination(GuidPrefix.GUIDPREFIX_UNKNOWN));`
	
	`// Write Message to bytes1 array`
	`byte[] bytes1 = Write(m1);`
	
	`// Read from bytes1 array - tests reading`
	`Message m2 = Read(bytes1);`
	
	`// Write the message Read to bytes2`
	`byte[] bytes2 = Write(m2);`
	
	`// Test, that bytes1 and bytes2 are equal`
	`AssertArrayEquals(bytes1, bytes2);`
	`}`

Salida	Nombre de la prueba: *TestInfoDestination* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00.1653231

4.2.2. TRANSPORTE

4.2.2.1. Prueba de Detección de paquetes RTPS.

En esta prueba se verifica el funcionamiento de los UDP *Receiver,* por medio de la generación de mensajes RTPS.

Tabla 4.3. TestPublishData

Llamada: `public UDPReceiver(Uri uri, int bufferSize)`

Tabla 4.3. TestPublishData

Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS, a los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert
Entrada	Inicialmente no se tiene inicializado al *Receiver UDP*

Código	`[TestMethod]`
	`public void TestPublishData()`
	`{`
	`object key = new object();`
	`UDPReceiver rec = new UDPReceiver(new Uri("udp://" + Host + ":" + Port), 1024);`
	
	`rec.MessageReceived += (s, m) =>`
	`{`
	`Message msg = m.Message;`
	`Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint);`
	`Debug.WriteLine("Message Header: {0}", msg.Header);`
	`Assert.AreEqual(ProtocolId.PROTOCOL_RTPS, msg.Header.Protocol);`
	`Assert.AreEqual(VendorId.OCI, msg.Header.VendorId);`
	
	`Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1, msg.Header.Version);`
	`Assert.AreEqual(2, msg.SubMessages.Count);`
	`foreach (var submsg in msg.SubMessages)`
	`{`
	`Debug.WriteLine("SubMessage: {0}", submsg);`
	`if (submsg is Data)`
	`{`
	`Data d = submsg as Data;`
	`foreach (var par in d.InlineQos.Value)`
	`Debug.WriteLine("InlineQos: {0}", par);`
	`}`
	`}`
	`lock (key) Monitor.Pulse(key);`
	`};`
	
	`rec.Start();`
	
	`simulator.SendUDPPacket("SamplePackets/packet1.dat", Host, Port);`
	`lock (key)`
	`{`
	`Assert.IsTrue(Monitor.Wait(key, 1000), "Time-out. Message has not arrived or there is an error on it.");`
	`}`
	`rec.Close();`
	`}`
	
	`}`

Salida	Nombre de la prueba: *TestPublishData*

Tabla 4.3. TestPublishData

	Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,640815 Salida estándar de Result: Trace du débogage : no configuration section <common/logging> found - suppressing logging output Sent 613/613 bytes to 224.0.1.111:9999 New Message has arrived from 172.30.82.26:56951 Message Header: [RTPS, 2.1, 01-03, 01-03-00-00-01-23-45-67-89-AB-CD-EF] SubMessage: InfoTimestamp:header[9, 1, 8], 12/12/1944 16:04:37 [1418400277:3853715240] SubMessage: Data:header[21, 7, 577], Payload[Rtps.Messages.Submessages.Elements.*Serialize*d Payload] InlineQos: ParameterId=PID_KEY_HASH, Content=09-23-09-23-00-00-00-00-00-00-00-00-00-00-00-00 InlineQos: ParameterId=PID_SENTINEL, Content=

4.2.2.2. Pruebas de paquetes RTPS.

En estas pruebas se verifica el funcionamiento de los UDP *Receiver,* pero a diferencia de la prueba anterior en estas se utilizan transmisores de otros fabricantes, es decir, se comprueba que los receptores pueden recibir de una manera adecuada los mensajes RTPS. El fabricante que se utiliza para las pruebas corresponde a *OpenDDS.*

Tabla 4.4. TestPublishPacket2

Llamada: `public UDPReceiver(Uri uri, int bufferSize)`	
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS de otros vendors con los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert
Entrada	Inicialmente no se tiene inicializado al *Receiver UDP*

Código	`[TestMethod]`
	`public void TestPublishPacket2()`
	`{`
	`object key = new object();`
	`UDPReceiver rec = new UDPReceiver(new Uri("udp://" + Host + ":" + Port), 1024);`
	
	`rec.MessageReceived += (s, m) =>`
	`{`
	`Message msg = m.Message;`
	`Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint);`

Tabla 4.4. TestPublishPacket2

```
                Debug.WriteLine("Message Header: {0}",
msg.Header);
                Assert.AreEqual(ProtocolId.PROTOCOL_RTPS,
msg.Header.Protocol);
                Assert.AreEqual(VendorId.OCI,
msg.Header.VendorId);

Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1,
msg.Header.Version);
                Assert.AreEqual(2, msg.SubMessages.Count);
                foreach (var submsg in msg.SubMessages)
                {
                    Debug.WriteLine("SubMessage: {0}", submsg);
                    switch (submsg.Kind)
                    {
                        case SubMessageKind.DATA:
                            Data d = submsg as Data;
                            foreach (var par in
d.InlineQos.Value)
                                Debug.WriteLine("InlineQos:
{0}", par);
                            break;
                        case SubMessageKind.INFO_TS:
                            InfoTimestamp its = submsg as
InfoTimestamp;
                            Debug.WriteLine("The TimeStampFlag
value state is: {0}", its.HasInvalidateFlag);
                            Debug.WriteLine("The EndiannessFlag
value state is: {0}", its.Header.Flags.IsLittleEndian);
                            Debug.WriteLine("The
octetsToNextHeader value is: {0}",
its.Header.SubMessageLength);
                            if (its.HasInvalidateFlag == false)
                            {
                                Debug.WriteLine("The Timestamp
value is: {0}", its.TimeStamp);

                            }
                            break;
                        default:
                            Assert.Fail("Only Timestamp and
Data submesages are expected");
                            break;
                    }
                }
                lock (key) Monitor.Pulse(key);
            };

            rec.Start();


simulator.SendUDPPacket("SamplePackets/packet3.dat", Host,
Port);
            lock (key)
            {
                Assert.IsTrue(Monitor.Wait(key, 10000), "Time-
out. Message has not arrived or there is an error on it.");
            }
            rec.Close();
```

Tabla 4.4. TestPublishPacket2

	}
Salida	Nombre de la prueba: *TestPublishPacket2* Resultado de la prueba:  Duración de la prueba: 0:00:00,2431361 Salida estándar de Result: Trace du débogage : no configuration section <common/logging> found - suppressing logging output Sent 196/196 bytes to 224.0.1.111:7400 New Message has arrived from 172.30.82.26:63179 Message Header: [RTPS, 2.1, 01-03, 01-03-00-00-01-23-45-67-89-AB-CD-EF] SubMessage: InfoTimestamp:header[9, 1, 8], 01/01/1900 00:00:00 [0:0] The TimeStampFlag value state is: False The EndiannessFlag value state is: True The octetsToNextHeader value is: 8 The Timestamp value is: 01/01/1900 00:00:00 [0:0] SubMessage: Data:header[21, 11, 0], Payload[Rtps.Messages.Submessages.Elements.*Serialize*dPayload] InlineQos: ParameterId=PID_STATUS_INFO, Content=00-00-00-01 InlineQos: ParameterId=PID_TOPIC_NAME, Content=0A-00-00-00-4D-79-20-54-6F-70-69-63-20-00-00-00 InlineQos: ParameterId=PID_PRESENTATION, Content=E7-03-00-00-00-00-00-00 InlineQos: ParameterId=PID_PARTITION, Content=01-00-00-00-06-00-00-00-48-65-6C-6C-6F-00-00-00 InlineQos: ParameterId=PID_OWNERSHIP_STRENGTH, Content=0C-00-00-00 InlineQos: ParameterId=PID_LIVELINESS, Content=02-00-00-00-FF-FF-FF-7F-FF-FF-FF-7F InlineQos: ParameterId=PID_RELIABILITY, Content=00-00-00-00-00-00-00-00-00-E1-F5-05 InlineQos: ParameterId=PID_TRANSPORT_PRIORITY, Content=0D-00-00-00 InlineQos: ParameterId=PID_LIFESPAN, Content=0E-00-00-00-FF-FF-FF-7F InlineQos: ParameterId=PID_DESTINATION_ORDER, Content=01-00-00-00 InlineQos: ParameterId=PID_SENTINEL, Content=

4.2.3. UTILS

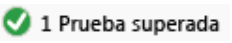
4.2.3.1. Pruebas del generador de identidad.

En estas pruebas se verifica el correcto funcionamiento del *Guid Generator,* el cual es el generador de identidades.

Tabla 4.5. TestGenerator1

Llamada: `static GuidGenerator()`	
Descripción	En esta prueba se verifica el correcto funcionamiento del Guid Generator.
Entrada	Inicialmente no se tiene inicializado al *GuidGenerator*

Código	`[TestMethod]`
	`public void TestGeneration1()`
	`{`
	`GuidGenerator generator = new GuidGenerator();`
	`GUID guid = generator.GenerateGuid();`
	
	`Assert.AreEqual(GuidGenerator.VENDORID_DOOPEC[0], guid.Prefix.Prefix[0]);`
	
	`Assert.AreEqual(GuidGenerator.VENDORID_DOOPEC[1], guid.Prefix.Prefix[1]);`
	`}`

Salida	Nombre de la prueba: *TestGeneration1* Resultado de la prueba:  Duración de la prueba: `0:00:00,172692`

4.2.3.2. Pruebas del PeriodicWorker.

En estas pruebas se verifica el funcionamiento del *PeriodicWorker,* el cual está encargado del envío de mensajes con submensajes *Heartbeat* los cuales sirven para mantener la conexión entre entidades.

Tabla 4.6. TestWorkerVerySlow

Llamada: `private void KeepWorkerRunning()`	
Descripción	En esta prueba se verifica el correcto funcionamiento del Worker en el cual se realiza tareas de actualización y descubrimiento
Entrada	Inicialmente no se tiene inicializado al Worker
Código	`[TestMethod]` `public void TestWorkerVerySlow()`

Tabla 4.6. TestWorkerVerySlow

	{
	int period = 2 * 1000;
	int sleepTime = 20 * 1000+90;
	PeriodicWorker worker = new PeriodicWorker();
	worker.Start(period);
	Thread.Sleep(sleepTime);
	worker.End();
	Assert.AreEqual(sleepTime / period, worker.Count);
	}

Salida	Nombre de la prueba: *TestWorkerVerySlow* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:20,2217009

4.2.3.3. Prueba de tiempo

En esta prueba se verifica la conversión del tipo *timeMillis* hacia el tipo de dato *long*.

Tabla 4.7. TestTimeSeconds

Llamada: public Time(long systemCurrentMillis)	
Descripción	En esta prueba se verifica el correcto funcionamiento del temporizador
Entrada	Inicialmente no se tiene inicializado al *Time*

Código	[TestMethod]
	public void TestTimeSeconds()
	{
	long timeMillis = 1000; // 1 sec
	Time t = new Time(timeMillis);
	long timeConverted = t.TimeMillis;
	
	Assert.AreEqual(timeMillis, timeConverted);
	}

Salida	Nombre de la prueba: *TestTimeSeconds* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:02,8221745

4.2.4. SERIALIZADOR

4.2.4.1. Pruebas del BuiltinTopic

En estas pruebas se verifica el correcto funcionamiento del serializador de la implementación del API-DDS, la cual es encapsulada en los mensajes RTPS.

Tabla 4.8. TestParticipantBuiltinTopicData

Llamada:

Tabla 4.8. TestParticipantBuiltinTopicData

public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador del DDS en el Builtin Data Participant
Entrada	Inicialmente no se tiene inicializado al ddsType

Código	[TestMethod]
	public void TestParticipantBuiltinTopicData()
	{
	var ddsType = TypeExplorer.ExploreType(typeof(ParticipantBuiltinTopicData));
	Assert.IsNotNull(ddsType);
	
	Assert.IsNotNull(ddsType.GetProperty());
	var propInfo = ddsType.GetProperty();
	
	Assert.AreEqual("org.omg.dds.topic.ParticipantBuiltinTopicData", propInfo.Name);
	
	Assert.IsInstanceOfType(ddsType, typeof(StructureType));
	StructureType structType = ddsType as StructureType;
	var members = structType.GetMember();
	Assert.IsNotNull(members);
	Assert.AreEqual(2, members.Count);
	Assert.AreEqual("Key", members[0].GetProperty().Name);
	Assert.AreEqual("UserData", members[1].GetProperty().Name);
	
	Assert.AreEqual((uint)0x0050, members[0].GetProperty().MemberId);
	Assert.AreEqual((uint)0x002C, members[1].GetProperty().MemberId);
	}

Salida	Nombre de la prueba: *TestParticipantBuiltinTopicData* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1871316

4.2.4.2. Pruebas de Encapsulación CDR.

En estas pruebas se procede a verificar la encapsulación CDR, es decir, que se comprueba la forma en la que se ordenan los *bytes* en los diferentes tipos de datos que se muestran de la forma *LittleEndian* y *BigEndian.*

Tabla 4.9. TestBoolPacketLE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order)

Tabla 4.9. TestBoolPacketLE

public static void *Serialize*(IoBuffer buffer, object dataObj, ByteOrder order) public static T De*serialize*<T>(IoBuffer buffer)	
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el BoolPacket Little Endian
Entrada	Dentro del Test Initialize se inicializa el serializador

Código	[TestMethod]
	public void TestBoolPacketLE()
	{
	BoolPacket v1 = new BoolPacket(true);
	int bufferSize = sizeof(bool) + CDRHeaderSize;
	var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize);
	CDREncapsulation.*Serialize*(buffer, v1, ByteOrder.LittleEndian);
	Assert.AreEqual(bufferSize, buffer.Position);
	
	buffer.Rewind();
	Assert.AreEqual("00 01 00 00 01", buffer.GetHexDump());
	BoolPacket v2 = CDREncapsulation.De*serialize*<BoolPacket>(buffer);
	Assert.AreEqual(v1, v2);
	Assert.AreEqual(bufferSize, buffer.Position);
	}

Salida	Nombre de la prueba: *TestBoolPacketLE* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1651349

4.2.4.3. Pruebas de exploración de tipo.

En esta prueba se verifica que la serialización de la clase *ddsType* funciona de manera adecuada.

Tabla 4.10. TestExploreMyClass1

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador de DDS en una Clase
Entrada	Inicialmente no se tiene inicializado al ddsType
Código	[TestMethod] public void TestExploreMyClass1() {

Tabla 4.10. TestExploreMyClass1

	var ddsType = TypeExplorer.ExploreType(typeof(XMyClass1));
	Assert.IsNotNull(ddsType);
	
	Assert.IsNotNull(ddsType.GetProperty());
	var propInfo = ddsType.GetProperty();
	
	Assert.AreEqual("*Serializer*Tests.XMyClass1", propInfo.Name);
	
	Assert.IsInstanceOfType(ddsType, typeof(StructureType));
	StructureType structType = ddsType as StructureType;
	var members = structType.GetMember();
	Assert.IsNotNull(members);
	Assert.AreEqual(3, members.Count);
	Assert.AreEqual("m_byte", members[0].GetProperty().Name);
	Assert.AreEqual("m_int", members[1].GetProperty().Name);
	Assert.AreEqual("m_short", members[2].GetProperty().Name);
	
	Assert.AreEqual((uint)0x8001, members[0].GetProperty().MemberId);
	Assert.AreEqual((uint)0x8002, members[1].GetProperty().MemberId);
	Assert.AreEqual((uint)0x8003, members[2].GetProperty().MemberId);

Salida	Nombre de la prueba: *TestExploreMyClass1* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1996965

4.2.4.4. Pruebas de Paquetes.

En estas pruebas se verifica que los diferentes tipos de datos se están serializando y deserializando de manera adecuada.

Tabla 4.11. TestBoolPacket

Llamada: public BoolPacket(bool v)	
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el *BoolPacket.*
Entrada	En la prueba no se tiene inicializado el fichero.

Código	[TestMethod]
	public void TestBoolPacket()
	{
	BoolPacket v1 = new BoolPacket(true);
	var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(bool));
	Serializer.*Serialize*(buffer, v1);

Tabla 4.11. TestBoolPacket

	Assert.AreEqual(sizeof(bool), buffer.Position);
	
	buffer.Rewind();
	BoolPacket v2 = *Serializer*.De*serialize*<BoolPacket>(buffer);
	Assert.AreEqual(v1, v2);
	Assert.AreEqual(sizeof(bool), buffer.Position);
	}

Salida	Nombre de la prueba: *TestBoolPacket* Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:39.9332615

4.2.4.5. Pruebas de Primitivas.

En estas pruebas se verifica que los tipos de datos primitivos están siendo serializados y deserializados de manera adecuada.

Tabla 4.12. TestDouble

Llamada: public struct Double : IComparable, IFormattable, IConvertible, IComparable<double>, IEquatable<double>	
Descripción	En esta prueba se verifica el estado de la primitiva *doublé*.
Entrada	En la prueba no se tiene inicializado el fichero.

Código	[TestMethod]
	public void TestDouble()
	{
	double v1 = 1.0;
	var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(double));
	
	Doopec.*Serializer*.Primitives.WritePrimitive(buffer, v1);
	
	buffer.Rewind();
	double v2;
	
	Doopec.*Serializer*.Primitives.ReadPrimitive(buffer, out v2);
	Assert.AreEqual(v1, v2);
	}

Salida	Nombre de la prueba: *TestDouble* Resultado de la prueba: 1 Prueba superada Duración de la prueba: < 1 ms

4.3. PRUEBA APLICACIÓN RTPS

A continuación, se muestra los requerimientos necesarios para el uso de un chat trabajando bajo el Middleware DDS-RTPS, se presenta además la interfaz de aplicación.

El objetivo de esta prueba es demostrar que las librerías RTPS y DDS que fueron generadas en el capitulo 3 se encuentran funcionando adecuadamente y asi

comprobar que estas librerías sirven para que varias computadoras se comuniquen entre sí.

El código del chat desarrollado y el manual de usuario del chat se encuentran en el Anexo C.

4.3.1. ESCENARIO PARA LA PRUEBA

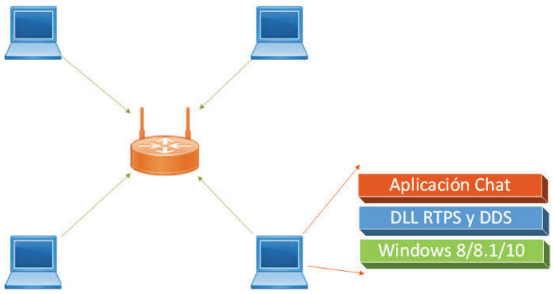


Figura 4.1. Escenario prueba Aplicación Chat DDS-RTPS

En la Figura 4.1, se muestra un ambiente con 4 computadoras las cuales tienen previamente cargado el software Aplicación Chat; estas se conectan a través de un Access point, el cual permite la comunicación entre las computadoras. Una vez realizada las configuraciones necesarias se procede a probar el programa.

4.3.2. REQUERIMIENTOS PARA EL USO DE LA APLICACIÓN

Para usar la aplicación se debe tener en cuenta los siguientes requerimientos:

- Sistema Operativo Windows 7, 8, 8.1 y 10 o superior.
- Copiar el paquete del CD.

4.3.3. APLICACIÓN CHAT CON TECNOLOGÍA DDS-RTPS

Se muestra la interfaz de usuario y las capturas de paquetes.

4.3.3.1. Interfaz de Usuario.

1) Ejecutar la aplicación

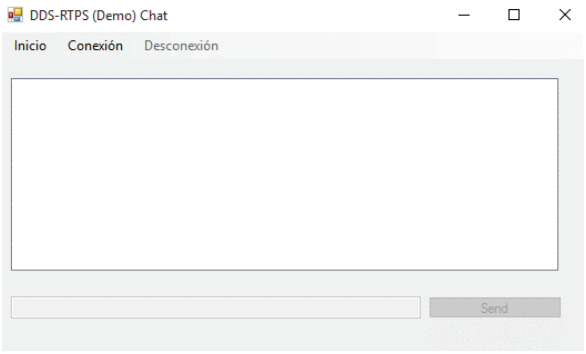


Figura 4.2. Ejecución del Chat RTPS.

2) Conectarse al servicio de chat

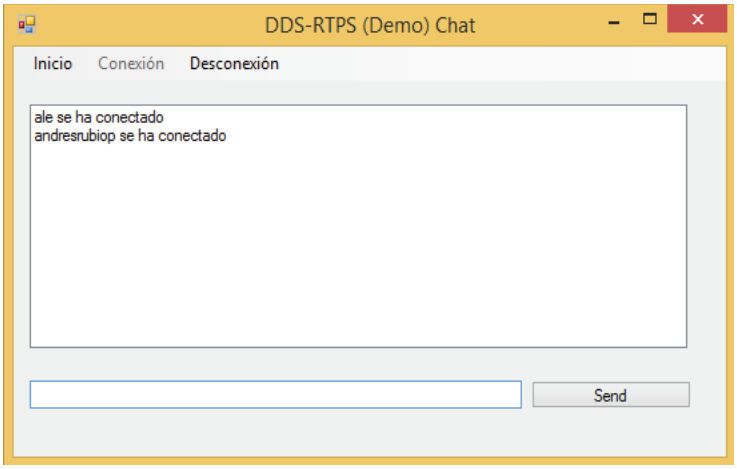


Figura 4.3. Conexión al servicio de Chat.

3) Intercambio de mensajes

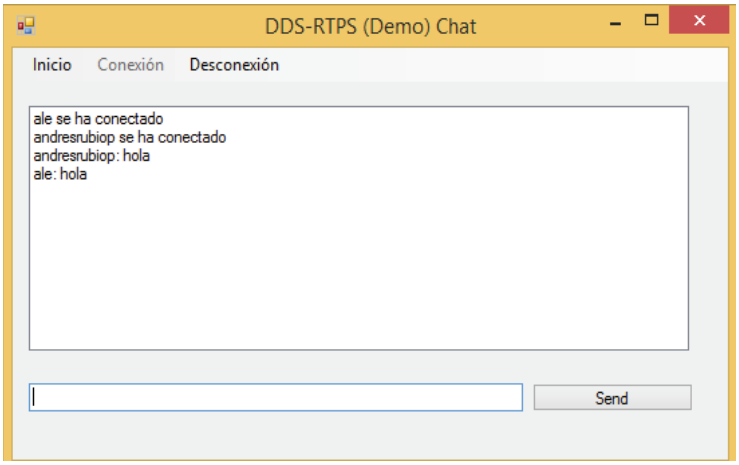


Figura 4.4. Intercambio de mensajes.

4) Desconexión del servicio de chat

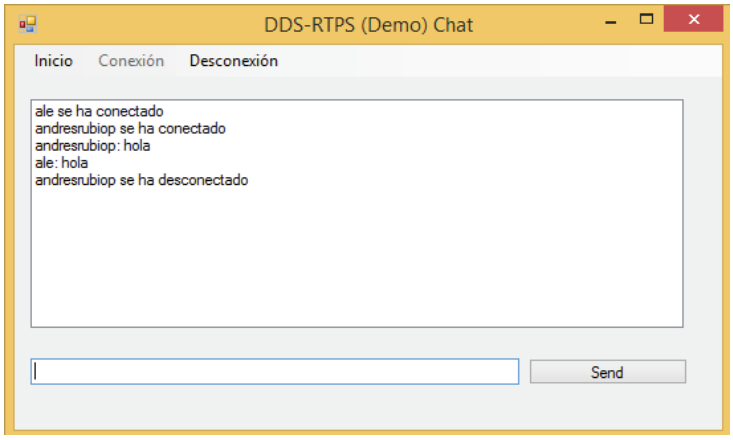


Figura 4.5. Desconexión del servicio de Chat.

5) Salida de la aplicación

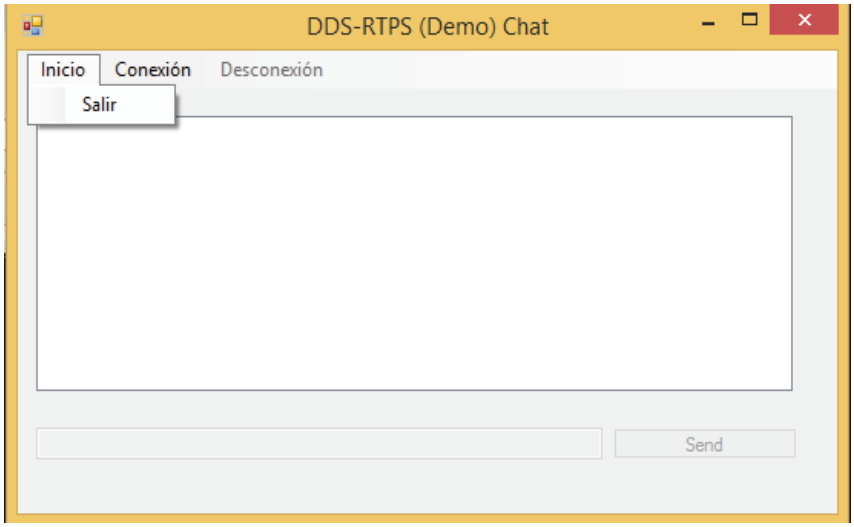


Figura 4.6. Salida de la aplicación.

4.3.3.2. Captura de paquetes.

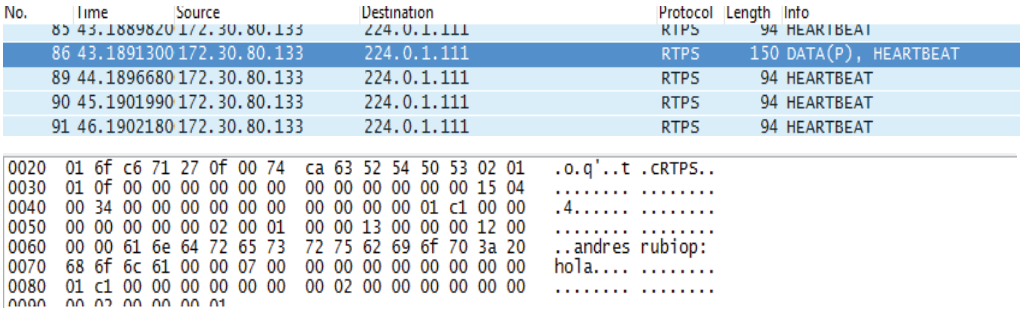


Figura 4.7. Paquete RTPS dentro del Chat.

En la Figura 4.7, se muestra un paquete RTPS capturado, que proviene de la Aplicación Chat DDS-RTPS, esl cual indica que el usuario andresrubiop esta eviando el mensaje ‘hola’.

4.4. MANUAL DE USUARIO DE LAS LIBRERIAS DDS y RTPS

4.4.1. MANUAL DE USUARIO

A continuación, se presenta el manual de usuario para la implementación del DDS – RTPS.

4.4.1.1. Requerimientos

Para utilizar la implementación DDS – RTPS se debe cumplir con varios requisitos necesarios para su uso:

- Sistema Operativo Windows 8 y 8.1
- Microsoft Visual Studio Ultimate 2013

- Descargar las librerías Common.Login.Core.dll, Common.Login.dll, DDS.dll, Doopec.dll, DoopecSerializer.dll, log4net.dll, Mina.NET.dll, RTPS.dll, SerializerDebug.dll

4.4.1.2. Proceso de Creación de un Proyecto en Lenguaje C#

1) Crear un proyecto en Visual Studio, seleccionando Archivo – Nuevo – Proyecto.

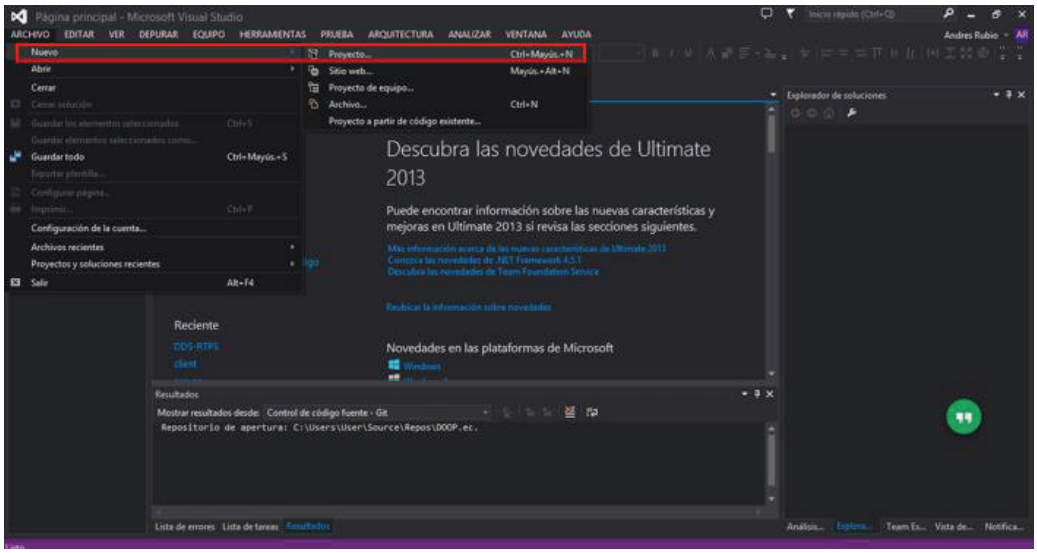


Figura 4.8. Creación de un Proyecto en Lenguaje C# 1

2) Escoger un proyecto Aplicación de Consola o de Aplicación de Windows Forms (Se ha escogido aplicación de consola). Escribiendo el nombre que se desee y seleccionar Aceptar.

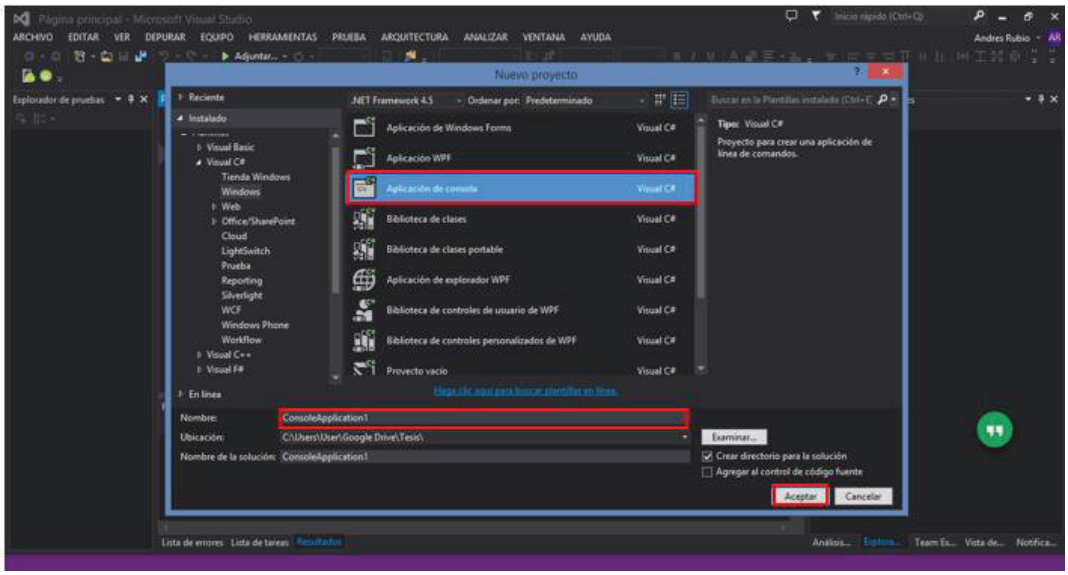


Figura 4.9. Creación de un Proyecto en Lenguaje C# 2

3) Una vez creada la solución se procede a hacer clic derecho en References y Agregar Referencia.

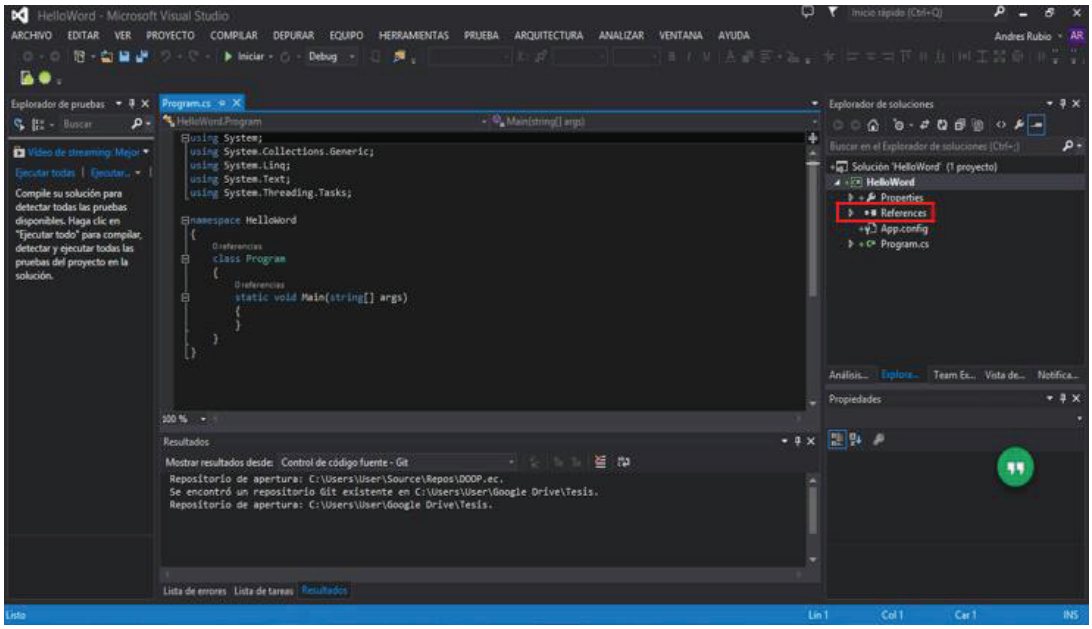


Figura 4.10. Creación de un Proyecto en Lenguaje C# 3

4) Una vez en el Administrador de Referencias, diríjase a Examinar.

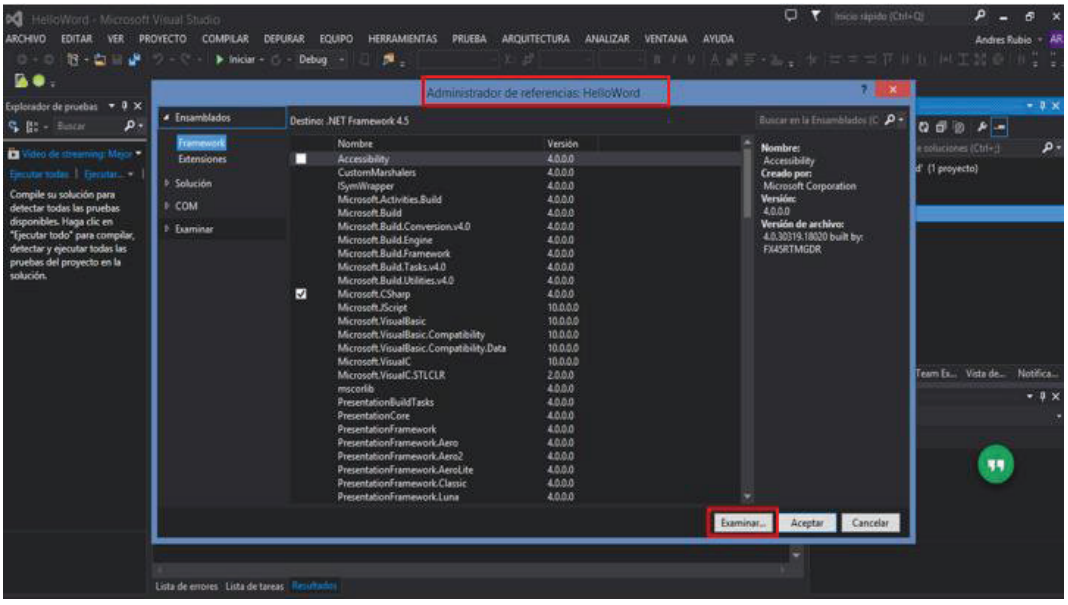


Figura 4.11. Creación de un Proyecto en Lenguaje C# 4

5) Dentro del botón Examinar, buscar la carpeta donde se encuentran las librerías dll y agregarlas.

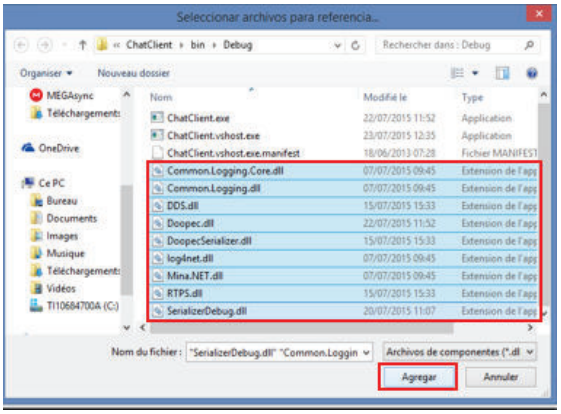


Figura 4.12. Creación de un Proyecto en Lenguaje C# 5

6) Una vez agregadas las librerías, marcarlas con un visto a todas y aceptar.

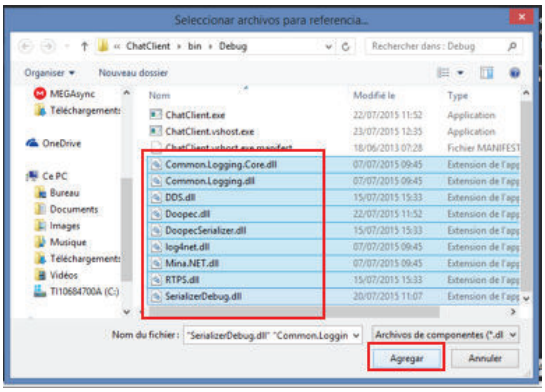


Figura 4.13. Creación de un Proyecto en Lenguaje C# 6

7) Se verifica en la pestaña de References que se encuentren todas las librerías agregadas.

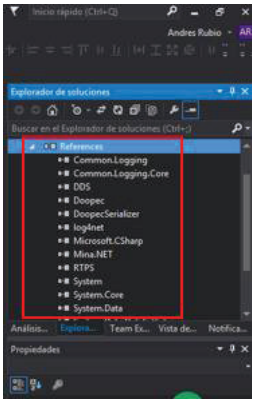


Figura 4.14. Creación de un Proyecto en Lenguaje C# 7

8) Agregar las referencias a la clase principal del proyecto, tal como se muestra en el gráfico.

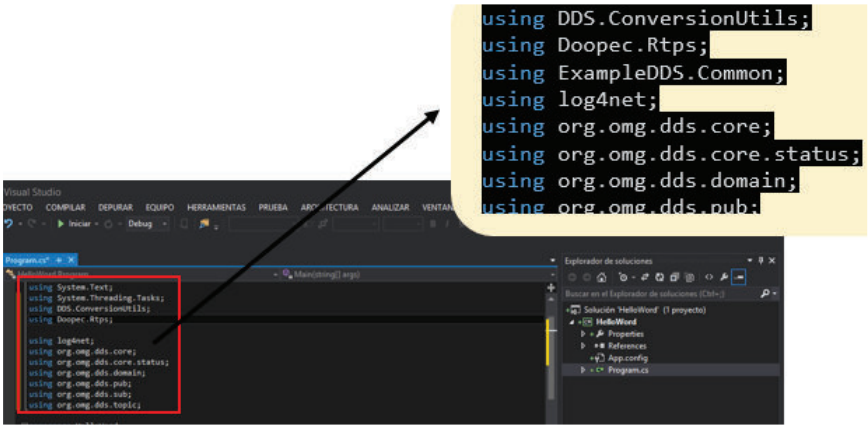


Figura 4.15. Creación de un Proyecto en Lenguaje C# 8

4.5. MANUAL PARA LA CREACION DE UN PROGRAMA "HOLA MUNDO" CON EL MIDDLEWARE DDS-RTPS

4.5.1. MANUAL DE USUARIO

A continuación se presenta el manual de usuario para la implementación de un ejemplo básico de la utilización de DDS - RTPS.

4.5.1.1. Requerimientos

Para poder realizar un ejemplo se debe anteriormente seguir el Manual de Usuario del Protocolo.

4.5.1.2. Proceso de Creacion del ejemplo "hola mundo"

Dentro de este programa se genera el envío y recepción de la secuencia Hola Mundo.

4.5.1.2.1. Creación de clases adicionales

Creación de la clase ExampleApp.cs

1) Clic derecho en el proyecto creado anteriormente, clic en agregar - Clase.

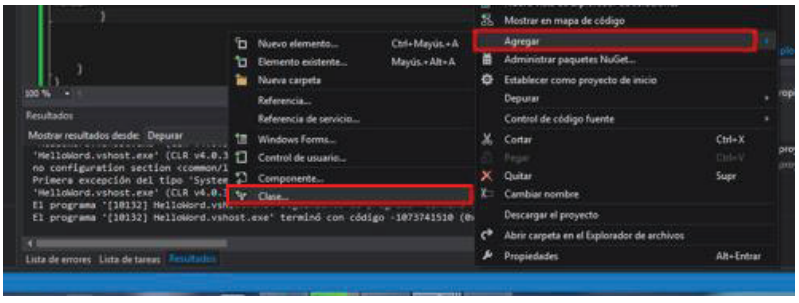


Figura 4.16. Creación de la clase ExampleApp.cs 1

2) Dar el nombre a la clase con *ExampleApp* y agregar.

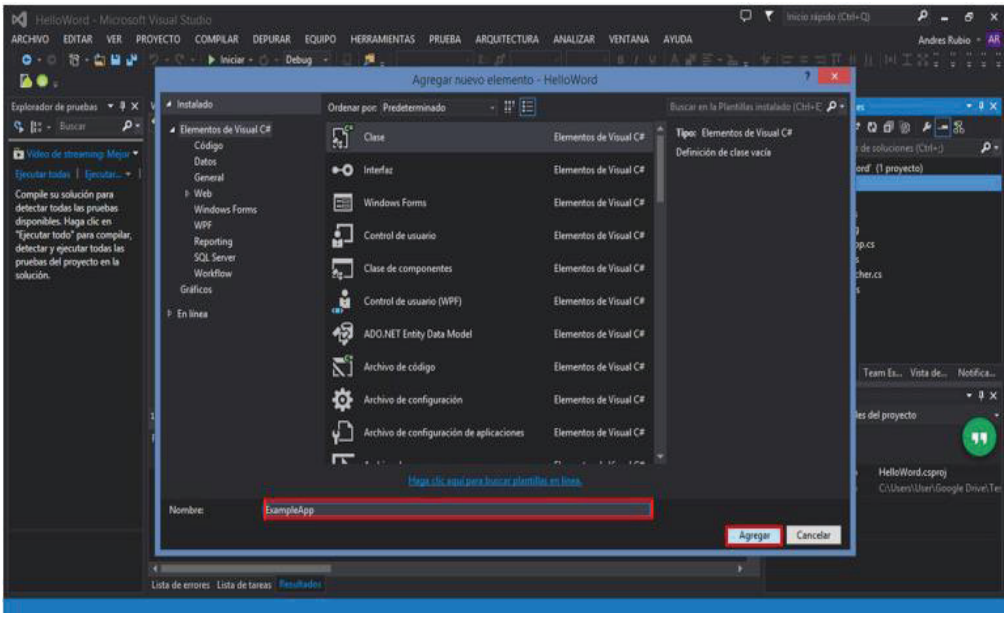


Figura 4.17. Creación de la clase ExampleApp.cs 2

3) Dentro de la clase *ExampleApp* crear el método *Public Virtual RunExample*, como se muestra en la figura. Verificar que las siguientes referencias se encuentren añadidas.

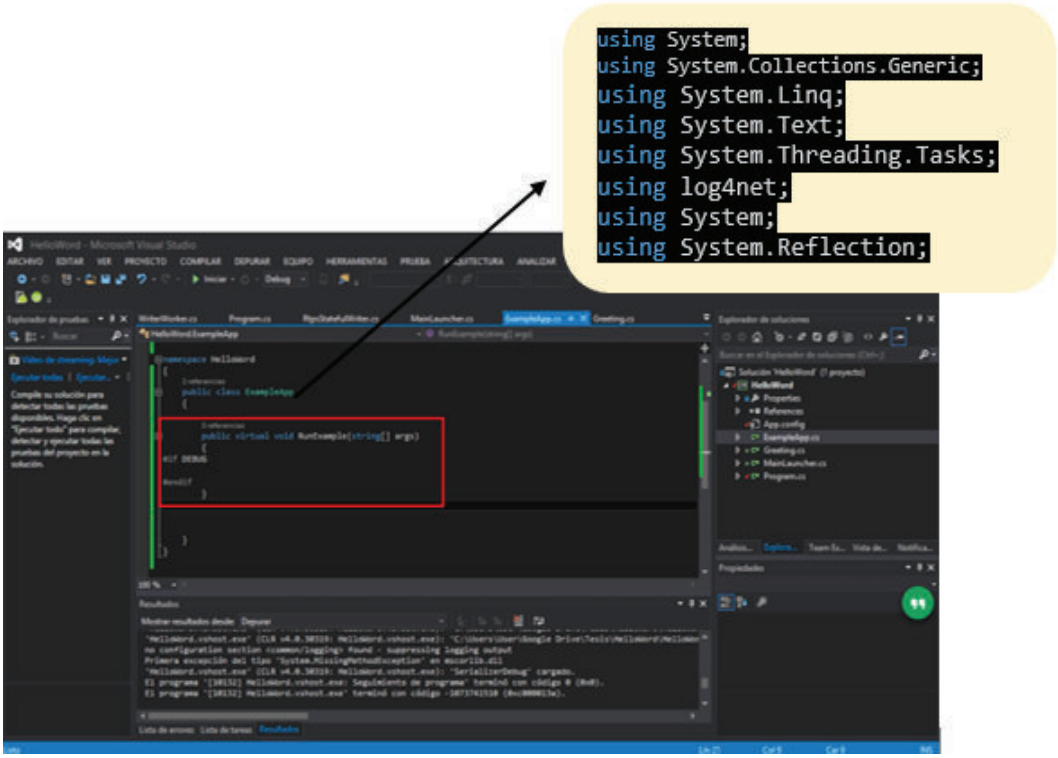


Figura 4.18. Creación de la clase ExampleApp.cs 3

Creación de la clase Greeting.cs

1) Clic derecho en el proyecto creado anteriormente, clic en agregar - Clase.

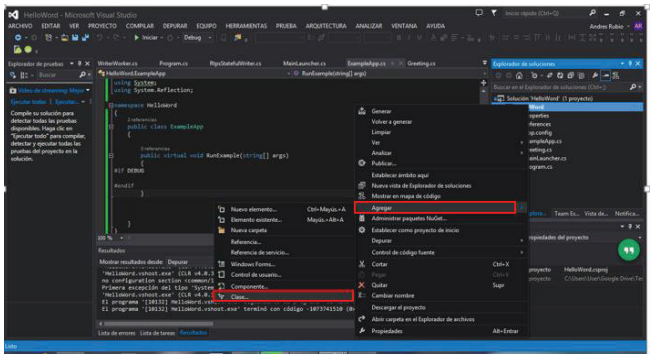


Figura 4.19. Creación de la clase Greeting.cs 1

2) Dar el nombre a la clase con *Greeting* y agregar.

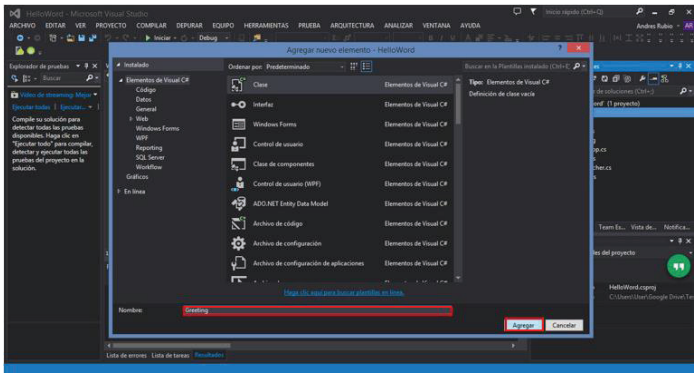


Figura 4.20. Creación de la clase Greeting.cs 2

3) Dentro de la clase *Greeting* agregar las instancias básicas de los atributos de las clases.

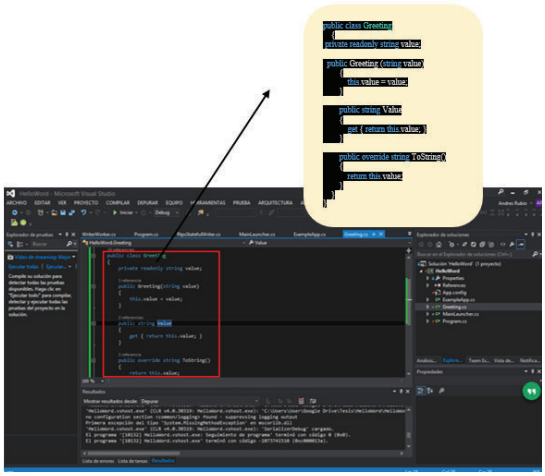


Figura 4.21. Creación de la clase Greeting.cs 3

Creación de la clase MainLauncher

1) Clic derecho en el proyecto creado anteriormente, clic en agregar - Clase.

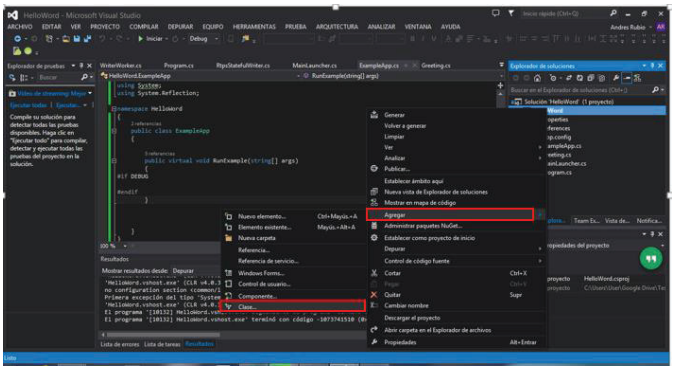


Figura 4.22. Creación de la clase MainLauncher.cs 1

2) Dar el nombre a la clase con *MainLauncher* y agregar.

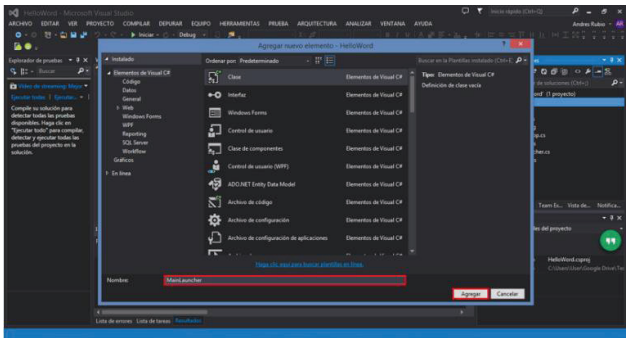


Figura 4.23. Creación de la clase MainLauncher.cs 2

3) Dentro de la clase *MainLauncher* agregar el siguiente código para poder correr la clase *Program.cs* (Esta clase se crea automáticamente al generar el proyecto).

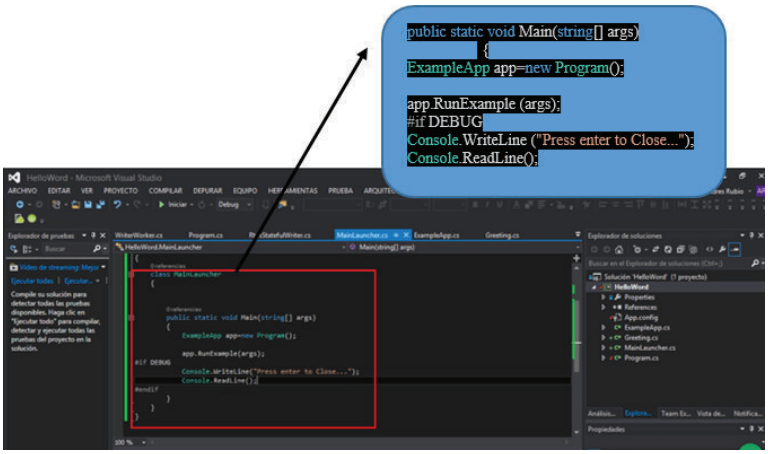


Figura 4.24. Creación de la clase MainLauncher.cs 3

Modificación del archivo de configuración App.config

El archivo de configuración se modifica de acuerdo a los siguientes parámetros.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="DDS" type="Doopec.Configuration.DDSConfigurationSection,
Doopec, Version=0.1.*, Culture=neutral, PublicKeyToken=null"/>
    <section name="RTPS" type="Doopec.Configuration.RTPSConfigurationSection,
Doopec, Version=0.1.*, Culture=neutral, PublicKeyToken=null"/>
  </configSections>
  <DDS xmlns="urn:Configuration" vendor="Doopec" version="2.1">
    <!--
              This is just a minimal sample configuration file that shows how to
declare
              the configuration sections.

              Because an XML Schema Definition (XSD) is generated for each
configuration
              section, it should be trivial to edit these files because you have
              IntelliSense on the XML definition.
              -->
    <boostrapType name="default" type="Doopec.Dds.Core.BootstrapImpl, Doopec"/>
    <domains>
      <domain name="Servidor" id="0">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-7D9376EA061C"/>
      </domain>
      <domain name="Servidor" id="3">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-7D9376EA061C"/>
      </domain>
      <domain name="Cliente1" id="1">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-7D9376EA061C"/>
      </domain>
      <domain name="Cliente2" id="2">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-7D9376EA061C"/>
      </domain>
    </domains>

    <logLevel levelMin="DEBUG" levelMax="FATAL"/>
    <qoSProfiles>
      <qoSProfileDef name="defaultQoS">
        <domainParticipantFactoryQos name="defaultDomainParticipantFactoryQoS">
          <entityFactory autoenableCreatedEntities="true"/>
        </domainParticipantFactoryQos>

        <domainParticipantQos name="defaultDomainParticipantQoS">
          <entityFactory autoenableCreatedEntities="true"/>
          <userData  value=""/>
        </domainParticipantQos>

        <topicQoS name="defaultTopicQoS">
```

```
          <topicData value=""/>
          <deadline  period="100"/>
          <durability  kind="VOLATILE"/>
        </topicQoS>

        <publisherQoS name="defaultPublisherQoS">
          <entityFactory autoenableCreatedEntities="true"/>
          <groupData value=""/>
          <partition value=""/>
          <presentation accessScope="INSTANCE" coherentAccess="true"
orderedAccess="true"/>
        </publisherQoS>

        <subscriberQoS name="defaultSubscriberQoS">
          <entityFactory autoenableCreatedEntities="true"/>
          <groupData value=""/>
          <partition value=""/>
          <presentation accessScope="INSTANCE" coherentAccess="true"
orderedAccess="true"/>
        </subscriberQoS>

        <dataWriterQoS name="defaultDataWriterQoS">
          <deadline period="1"/>
          <destinationOrder  kind="BY_SOURCE_TIMESTAMP"/>
          <durability  kind="VOLATILE"/>
          <durabilityService historyDepth="0" historyKind="KEEP_LAST"
maxInstances="1" maxSamples="1" maxSamplesPerInstance="1"
serviceCleanupDelay="100"/>
          <history kind="KEEP_LAST" depth="1"/>
          <latencyBudget duration="100"/>
          <lifespan duration="100"/>
          <liveliness kind="AUTOMATIC" leaseDuration="100"/>
          <ownership kind="SHARED"/>
          <ownershipStrength value="100"/>
          <reliability kind="RELIABLE" maxBlockingTime="1000"/>
          <resourceLimits maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1"/>
          <transportPriority value="1"/>
          <userData value=""/>
          <writerDataLifecycle autodisposeUnregisteredInstances="true"/>
        </dataWriterQoS>

        <dataReaderQoS name="defaultDataReaderQoS">
          <deadline period="1"/>
          <destinationOrder  kind="BY_SOURCE_TIMESTAMP"/>
          <durability  kind="VOLATILE"/>
          <history kind="KEEP_LAST" depth="1"/>
          <latencyBudget duration="100"/>
          <liveliness kind="AUTOMATIC" leaseDuration="100"/>
          <ownership kind="SHARED"/>
          <reliability kind="RELIABLE" maxBlockingTime="1000"/>
          <resourceLimits maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1"/>
          <readerDataLifecycle autopurgeDisposedSamplesDelay="1000"
autopurgeNowriterSamplesDelay="1000"/>
          <timeBasedFilter minimumSeparation="1000"/>
          <userData value=""/>
        </dataReaderQoS>
      </qoSProfileDef>
    </qoSProfiles>
  </DDS>
```

```
  <RTPS xmlns="urn:Configuration">
    <!--
              This is just a minimal sample configuration file that shows how to
declare
              the configuration sections.

              Because an XML Schema Definition (XSD) is generated for each
configuration
              section, it should be trivial to edit these files because you have
              IntelliSense on the XML definition.
              -->
    <transports>
      <transport name="defaultRtps" type="Doopec.Rtps.RtpsTransport.RtpsEngine,
Doopec">
        <ttl val="1"/>
        <discovery name="defaultDiscovery">
          <resendPeriod val="30000"/>
          <useSedpMulticast val="true"/>
          <portBase val="7400"/>
          <domainGain val="250"/>
          <participantGain val="2"/>
          <offsetMetatrafficMulticast val="0"/>
          <offsetMetatrafficUnicast val="10"/>
          <metatrafficUnicastLocatorList val="localhost"/>
          <metatrafficMulticastLocatorList val="239.255.0.1"/>
        </discovery>
        <rtpsWriter>
          <heartbeatPeriod val="1000"/>
          <nackResponseDelay val="200"/>
          <nackSuppressionDuration val="0"/>
          <pushMode val="true"/>
        </rtpsWriter>
        <rtpsReader>
          <heartbeatResponseDelay val="500"/>
          <heartbeatSuppressionDuration val="0"/>
        </rtpsReader>
      </transport>
    </transports>
  </RTPS>
  <appSettings>
    <add key="org.omg.dds.serviceClassName" value="Doopec.Dds.Core.BootstrapImpl,
Doopec" />
  </appSettings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.1" />
  </startup>
</configuration>
```

Espacio de Código 4.1. Archivo de Configuración

Codificación de la lógica del programa en la clase *Program.cs*

1) Dentro de la clase *Program.cs,* adicionalmente a las referencias agregadas se debe agregar *using HelloWorld*.
2) A la declaración de la clase *Program.cs* se le extiende la clase *ExampleApp.cs,* por lo cual es necesario tener el método

RunExample. Dentro del método *RunExample* primeramente inicializamos al participante, al topic y al *DomainParticipantFactory.*

```
public class Program : ExampleApp
    {

        public override void RunExample(string[] args)
        {
            base.RunExample(args);

            DomainParticipantFactory factory =
DomainParticipantFactory.GetInstance(Bootstrap.CreateInstance());
            DomainParticipant dp = factory.CreateParticipant();

            // Implicitly create TypeSupport and register type:
            Topic<Greeting> tp = dp.CreateTopic<Greeting>("Greetings Topic");
```

Espacio de Código 4.2. Código de la clase Program.cs 1

3) A continuación se crea el Publicador y el *DataWriter.*

```
            // Create the publisher
            Publisher pub = dp.CreatePublisher();
           /* DataWriter<Greeting> dw = pub.CreateDataWriter(tp);

*/

            DataWriter<Greeting> dw = pub.CreateDataWriter<Greeting>(tp,

pub.GetDefaultDataWriterQos(),null,  null);
```

Espacio de Código 4.3. Código de la clase Program.cs 2

4) Se crea el Suscriptor, el *DataReader* y el *DataReaderListener,* el cual servirá para la escucha de mensajes.

```
                        // Create the subscriber
            Subscriber sub = dp.CreateSubscriber();
            DataReaderListener<Greeting> ls = new MyListener();
            /*DataReader<Greeting> dr = sub.CreateDataReader(tp);*/

            DataReader<Greeting> dr = sub.CreateDataReader<Greeting>(tp,

sub.GetDefaultDataReaderQos(), ls, null  );
            /*
            // Now Publish some piece of data
            Greeting data = new Greeting("Hello, World with DDS.");
            Console.WriteLine("Sending data:\"{0}\"", data.Value);
            dw.Write(data);
            //and check that the reader has this data
            dr.WaitForHistoricalData(10, TimeUnit.SECONDS);
```

Espacio de Código 4.4. Código de la clase Program.cs 3

5) Se crea el código de publicación de datos utilizando método del *DataWriter write (),* y se espera en el *DataReader* un tiempo de 1500 ms hasta la recepción del mensaje.

```
*/
            int i = 0;
            // Now Publish some piece of data
            //Greeting data = new Greeting("Hola Mundo"+ i.ToString());

            for (i = 0; i < 1; i++)
            {
                Greeting data = new Greeting("Hola Mundo" + i.ToString());

                Console.WriteLine("Sending data:\"{0}\"", data.Value);
                dw.Write(data);
                dr.WaitForHistoricalData(1500, TimeUnit.MILLISECONDS);
            }

            //and check that the reader has this data
            //dr.WaitForHistoricalData(10000, TimeUnit.SECONDS);
            dp.Close();
        }
```

Espacio de Código 4.5. Código de la clase Program.cs 4

6) Dentro del *Listener* se genera un evento llamado *OnDataAvailable* que sirve para la escritura de mensajes.

```
private class MyListener : DataReaderAdapter<Greeting>
        {
            //private static readonly ILog log =
LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);

            public override void OnDataAvailable(DataAvailableStatus<Greeting>
status)
            {
                DataReader<Greeting> dr = status.GetSource();
                SampleIterator<Greeting> it = dr.Take();
                foreach (Sample<Greeting> smp in it)
                {
                    // SampleInfo stuff is built into Sample:
                    // InstanceHandle inst = smp.GetInstanceHandle();
                    // Data accessible from Sample; null if invalid:
                    Greeting dt = smp.GetData();
                    Console.WriteLine("Received data:\"{0}\"", dt.Value);
                }
            }
        }
    }
```

Espacio de Código 4.6. Código de la clase Program.cs 5

4.5.1.3. Resultado

En el siguiente gráfico se muestra el resultado de la compilación.

Figura 4.25. Resultado Programa Hola Mundo

4.6. PRUEBA APLICACIÓN CORBA

A continuación, se muestra los requerimientos necesarios para el uso de un chat con la tecnología CORBA y se presenta además la interfaz de aplicación.

El objetivo de esta prueba es comparar el funcionamiento, tiempos de transmisión de la tecnología CORBA con RTPS.

4.6.1. ESCENARIO PARA LA PRUEBA

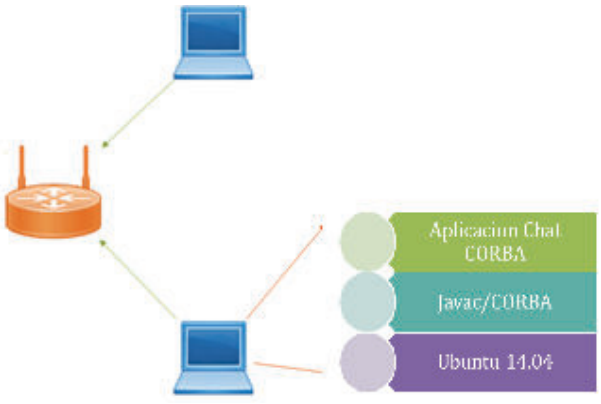


Figura 4.26. Escenario para la prueba del Chat CORBA

En la Figura 4.26, se muestra un ambiente con 2 computadoras trabajando con Ubuntu 14.04 las cuales tienen instalado la maquina virtual de java y CORBA; estas se conectan a través de un Access point, el cual permite la comunicación

entre las computadoras. Una vez realizada las configuraciones necesarias se procede a probar el programa. El programa se lo puede encontrar en el Anexo D.

4.6.2. REQUERIMIENTOS PARA EL USO DE LA APLICACIÓN

Para usar la aplicación se debe tener en cuenta los siguientes requerimientos:

- Sistema Operativo Ubuntu 14.04 o superior.
- Instalar el comando *make.* Usando el comando *apt-get install built-essential,* dentro del terminal.
- Instalar la máquina virtual de Java llamada *Javac.* Usando el comando *apt-get install openjdk-7-jdk,* dentro del terminal.

4.6.3. APLICACIÓN CHAT CON TECNOLOGÍA CORBA

Se muestra la interfaz de usuario y las capturas de paquetes.

4.6.3.1. Interfaz de Usuario

1) Inicialización del objeto CORBA.

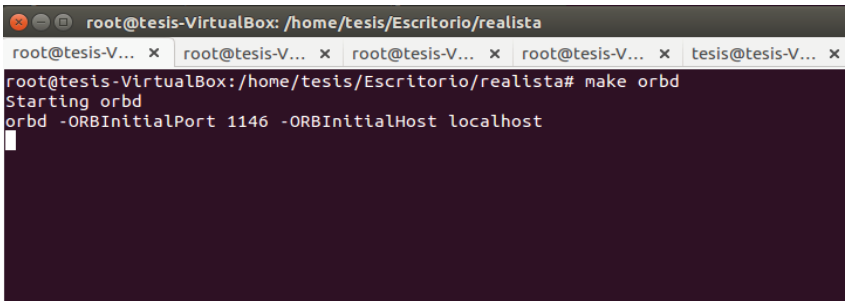


Figura 4.27. Chat CORBA 1

2) Inicialización del servidor.

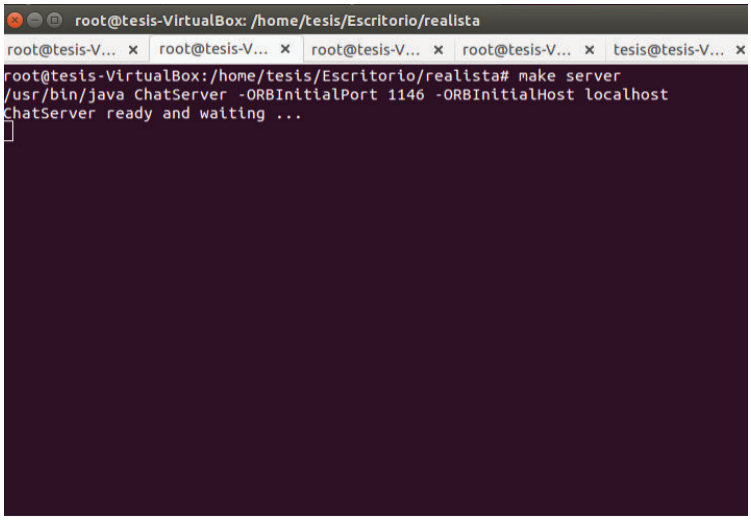


Figura 4.28. Chat CORBA 2

3) Inicialización de clientes.

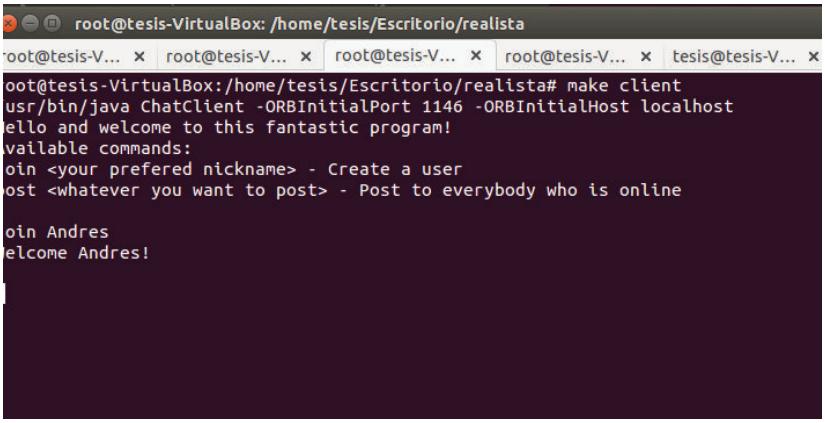


Figura 4.29. Chat CORBA 3

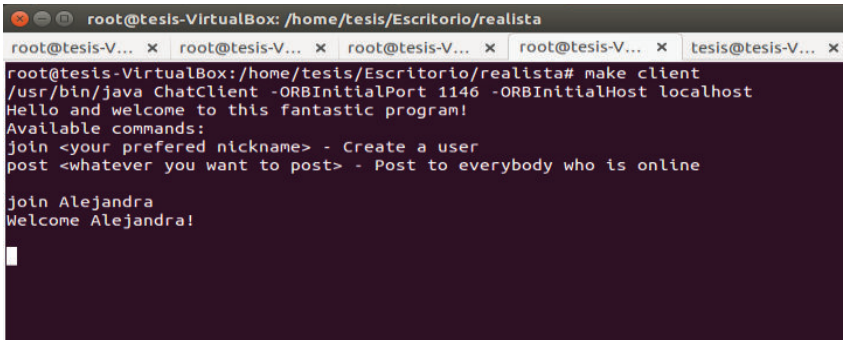


Figura 4.30. Chat CORBA 4

4) Intercambio de mensajes entre el cliente 1 y el cliente 2.

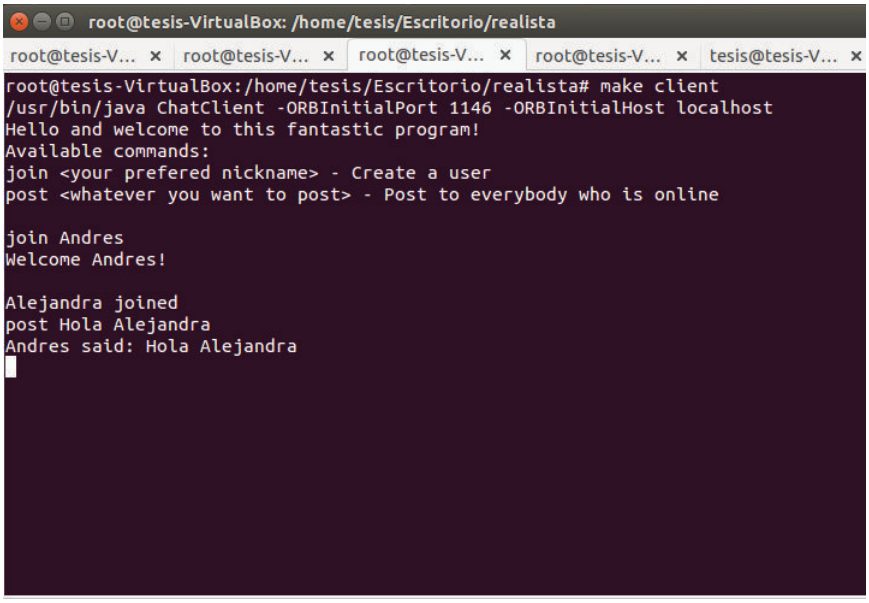


Figura 4.31. Chat CORBA 5

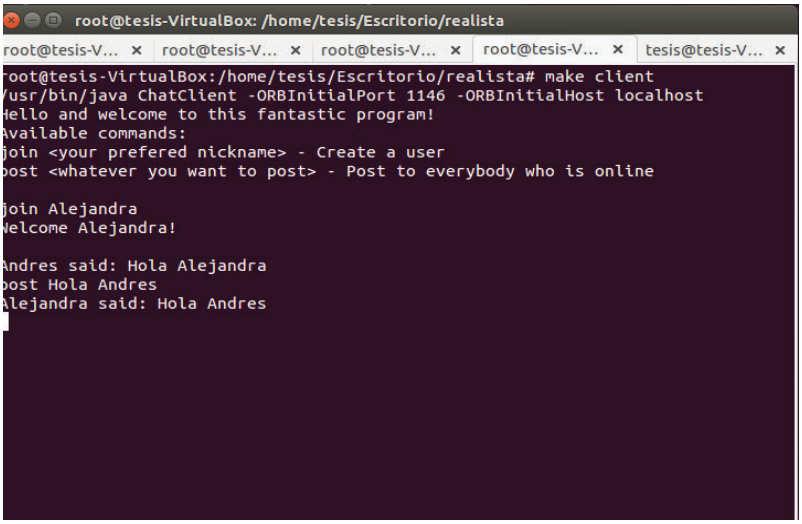


Figura 4.32. Chat CORBA 6

4.6.3.2. Captura de paquetes.

1) Paquete de envío del cliente 1. En la figura se puede observar como el cliente 1 ha enviado el mensaje 'Hola Alejandra' al cliente 2

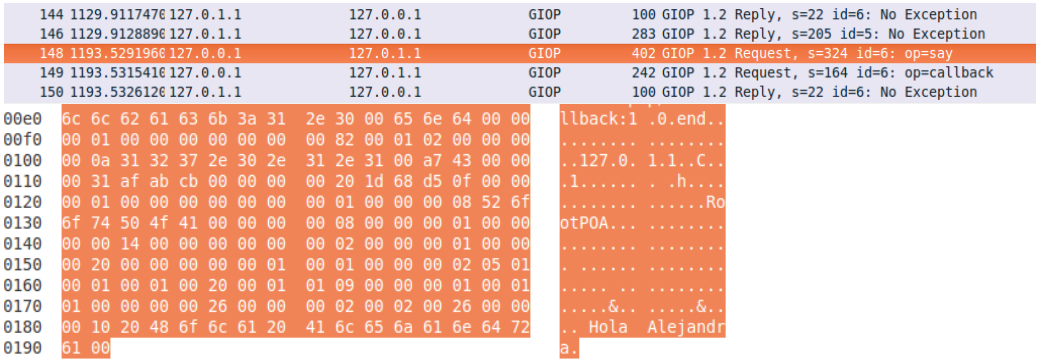


Figura 4.33. Captura de Paquetes CORBA 1

2) Paquete de Recepción del cliente 1 hacia el cliente 2. En la figura se muestra la recepción del mensaje enviado por el cliente 1 que dice 'Hola Alejandra'.

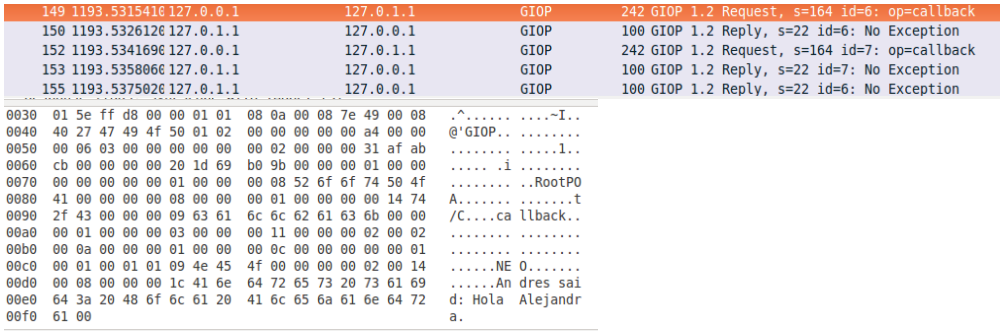


Figura 4.34. Captura de Paquetes CORBA 2

3) Paquete de envío del cliente 2. En la figura se puede observar que el Cliente 2 ha respondido con el mensaje ‘Hola Andres’ al cliente 1.

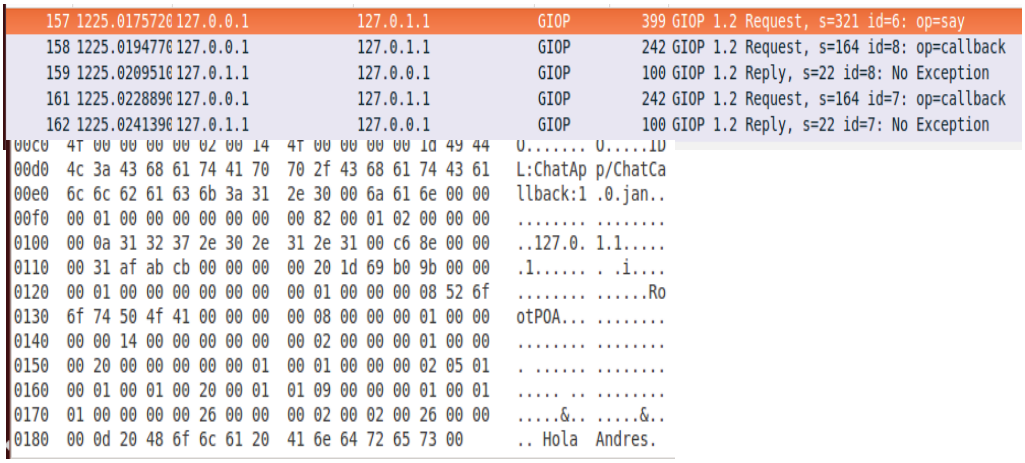


Figura 4.35. Captura de Paquetes CORBA 3

4) Paquete de recepción del cliente 2 hacia el cliente 1. En la figura se muestra el mensaje que recibe el cliente 1 ‘Hola Andres’.

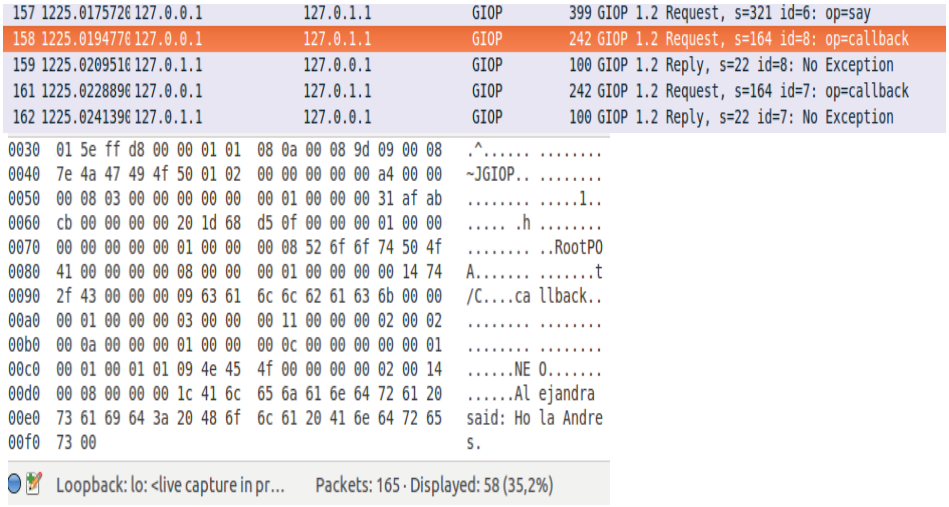


Figura 4.36. Captura de Paquetes CORBA 4

4.7. COMPARACIÓN DE APLICACIONES

A continuación, se muestra la comparación de la tecnología DDS-RTPS y CORBA-RT.

4.7.1. CAPTURAS DE PAQUETES DDS-RTPS Y CORBA-RT

En las siguientes imágenes se muestra un diagrama de flujo generado por la herramienta Wireshark para las dos tecnologías.

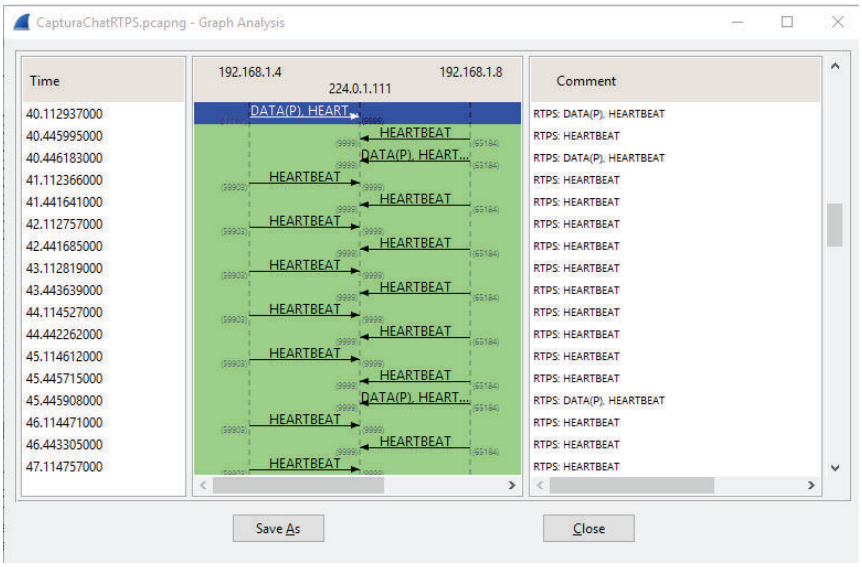


Figura 4.37. Flujo de datos DDS-RTPS

Para transmitir un mensaje de datos en RTPS es: 13.77ms y se requiere de 2 paquetes los cuales son HEARTBEAT y el DATA.

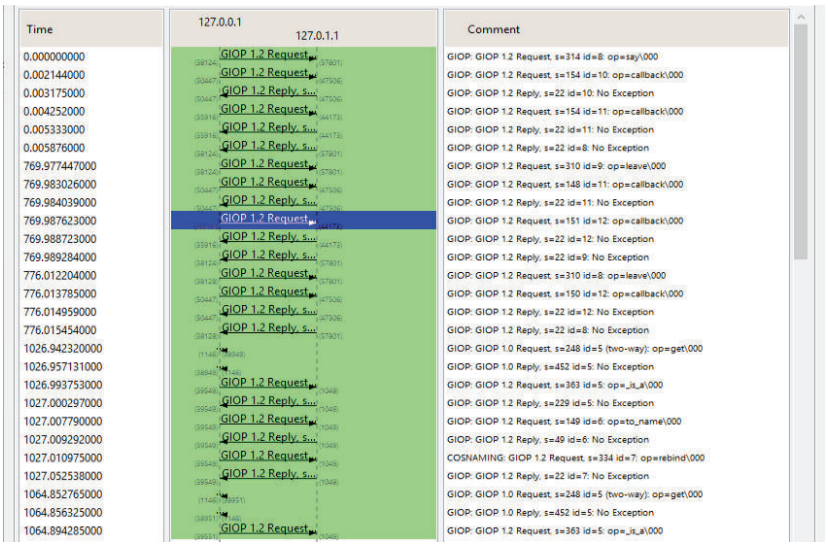


Figura 4.38. Flujo de datos CORBA-RT

Para transmitir un mensaje de datos en CORBA-RT es: 3.25ms y se requiere de 5 paquetes REQUEST y REPLY.

Tabla 4.13. Comparación de las tecnologías DDS-RTPS con CORBA-RT.

CARACTERÍSTICAS	DDS-RTPS	CORBA-RT
	Envío de 1 dato	
Tiempo de transmisión de un mensaje de datos	13,77ms	3,25ms
Número de paquetes para la transmisión	2	5

Tabla 4.13. Comparación de las tecnologías DDS-RTPS con CORBA-RT.

Paquetes necesarios para transmitir a dos suscriptores	2	10
Tiempo de transmisión de un mensaje a dos suscriptores	13,77ms	6,50ms

La tecnología CORBA-RT permite tener transmisiones de datos del tipo *unicast* sumamente rápidas, pero a medida que la tecnología requiere más escalabilidad el tiempo de transmisión de datos se muestra afectado; en cambio, la tecnología DDS-RTPS no se ve afectada en gran manera cuando se requiere escalabilidad, sin embargo en redes pequeñas esta no llega a superar a CORBA-RT.

Dentro de las pruebas unitarias, se ha podido evaluar al Middleware, superando *tests* que son necesarios para el funcionamiento de DDS y RTPS, tanto para que estos se comuniquen entre sí, como también para que la infomación que estos van a transportar se mantenga íntegra y no sufra alteraciones. Tambien se realizó una prueba global por medio de un escenario real con varias computadoras comunicándose entre si, el cual con éxito probó que se puede utilizar la librería DDS, para programar un sistema simple de comunicación como en este caso ha sido un Chat, e implícitamente se comprobó la librería de comunicación RTPS, donde con éxito se realizaron los pasos de mensajes.

CAPÍTULO 5.

CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

- La gestión de recursos de red presentes en el middleware DDS-RTPS, puede provocar un incremento en los tiempos de respuesta de las aplicaciones, aunque esta sobrecarga depende de casi exclusivamente de cada aplicación, este efecto es más significativo dentro de DDS-RTPS ya que define un conjunto de entidades que consumen recursos del procesador y de la red.
- El estándar DDS fue diseñado explícitamente para construir sistemas distribuidos en tiempo real, añadiendo un conjunto de parámetros de calidad de servicio para configurar propiedades del sistema y a su vez permiritr la reconfiguración dinámica del sistema, es decir, modificar parámetros en tiempo de ejecución.
- Los usos de pruebas unitarias permiten comprobar que los componentes de la aplicación trabajen de la manera esperada, además esto permite mejorar el código y tener procedimientos mas eficaces, y se las puede realizar independientemente del lenguaje de programación o de la plataforma de desarrollo utilizada.
- Dentro del comportamiento de la interacción entre las entidades DDS y sus correspondientes entidades RTPS, en lo concerniente a la escritura de datos el *DataWriter* DDS es el encargado de añadir y remover cambios del tipo *CacheChange* desde y hacia el *HistoryCache* del *Writer* RTPS asociado, es decir, el *Writer* RTPS no está en control cuando un cambio es removido desde *HistoryCache.*
- La implementación sin estado está optimizada para la escalabilidad, esta mantiene virtualmente un estado sumamente simple en las entidades remotas y por lo tanto esta puede escalar de manera adecuada en sistemas grandes. La implementación sin estado es ideal para las comunicaciones que requieran el modo *besteffort*, ya que al trabajar sin estado se requiere menos uso de memoria y por lo

tanto la comunicación más rápida, ya que no se requieren confirmaciones.

- La implementación con estado mantiene un total estado en las entidades remotas, esto minimiza el uso de ancho de banda, ya que los mensajes son confirmados y sólo se reenvían aquellos que han tenido algún problema, pero requiere una mayor capacidad de la memoria, y la escalabilidad es reducida, por lo tanto, garantiza una comunicación confiable.
- Dentro del descubrimiento existen dos fases, la primera concerniente al protocolo SPDP el cual se encarga de descubrir y anunciar de los participantes y la segunda al protocolo SEDP el cual se encarga de descubrir y anunciar de los servicios que publica el participante.
- El *DataWriter* es la cara del Publicador, el cual representa a los objetos responsables de la emisión de datos, lo usan los participantes para comunicar el valor y los cambios de los datos; una vez que la nueva información ha sido comunicada al Publicador, es responsabilidad de este determinar cuándo es apropiado emitir el correspondiente mensaje, es decir, lo realiza de acuerdo a su calidad de servicio asociada al correspondiente *DataWriter* o a su estado interno.
- Para acceder a los datos recibidos, el participante debe utilizar un tipo *DataReader* asociado al suscriptor, este recibe los datos publicados y los hace disponibles al participante. Un Suscriptor debe recibir y despachar datos de diferentes tipos especificados y asociar a un objeto *DataWriter*, el cual representa a una publicación, con el objeto *DataReader*, que representa la suscripción, es hecha por la entidad *Topic.*
- El Topic tiene el propósito de asociar un nombre único en el dominio, es decir, el conjunto de aplicaciones que se comunican entre sí.
- La forma de trabajar proporcionada por las operaciones *read* y *take* permite usar el DDS como una caché distribuida, es decir, una cache que se la va a encontrar en todos lo participantes que se encuentren en la comunicación o como un sistema de cola, o ambos. Esta es una

poderosa combinación que raramente se encuentra en la misma plataforma Middleware. Esta es una de las razones porque DDS es usado en una variedad de sistemas, algunas veces como una caché distribuida de alto rendimiento, y que la información es replicada, otras como tecnología de mensajería de alto rendimiento, y sin embargo, otras veces como una combinación de las dos.

- El uso del lenguaje de programación C#, a pesar de no ser un lenguaje común para programar middleware de comunicación, este nos permite tener una interacción más directa con las herramientas que comúnmente son requeridas al momento de desarrollar aplicaciones dirigidas a los usuarios. Además, permite la creación simple de aplicaciones multi tarea y permite utilizar fácilmente las sobrecargas en métodos.
- La tecnología DDS-RTPS permite desarrollar aplicaciones de comunicación que no se ven afectadas de una manera significativa al momento en que la red de datos deba crecer, es decir, que a comparación de otras tecnologías como CORBA-RT al tener un mayor número de usuarios la eficiencia de la comunicación es mínimamente afectada, ya que se comprobó dentro de las pruebas de las aplicaciones el comportamiento de las dos tecnologías. Ademas esto es importante ya que para un sistema distribuido de tiempo real, también es importante que al escalar la red, sus tiempos minimos de retardo no se vean afectado mayormente.

5.2. RECOMENDACIONES

- Para implementaciones críticas que utilicen el Middleware DDS-RTPS: como transmisión de flujos de video y audio; se recomienda el uso de calidad de servicio que proporciona el Middleware y el uso adecuado de los diferentes tipos de escritores y lectores, para obtener la mejor calidad en transmisión.
- Se sugiere contar con versiones recientes de software, ya que permite el uso de nuevas funcionalidades a las aplicaciones y obtener productos de calidad. Particularmente se recomienda el uso de Visual Studio 2013 y 2015 ya que a la fecha es una herramienta que cuenta con varias características que permiten crear una gran variedad de aplicaciones.
- Se recomienda revisar las publicaciones de la OMG con respecto a futuras actualizaciones o al estado actual de DDS y RTPS, con el propósito de tener un panorama más amplio al momento de realizar actualizaciones dentro Middleware.
- Una mejora al Middleware que se podría incluir, es la interoperabilidad con otros sistemas que trabajen con DDS-RTPS como OpenDDS, realizando pruebas y mejorando la compatibilidad con los perfiles de calidad de servicio.
- Se sugiere que este proyecto de titulación pueda ser la base para futuros proyectos que trabajen con el modelo Publicador-Subscriptor, se sugiere la implementaion de un sistema de transmisión de video que trabaje bajo las librerías de DDS-RTPS.
- Se sugiere para futuros proyector analizar los modelos matemáticos sobre el protocolo de comunicaciones RTPS.
- Se sugiere que al implementar un API primeramente se realice el *stack* de pruebas unitarias ya que estas permitirán tener un mayor control en la implementación.
- Se sugiere realizar aplicaciones de tiempo real, que puedan ser usadas a gran escala, teniendo como base las librerías generadas en

este proyecto, por ejemplo, video conferencia, sistemas de comunicación inalámbricas.

REFERENCIAS BIBLIOGRÁFICAS

[1] OMG, «Data Distribution Service for Real-Time Systems. v1.2.,» 2007.

[2] H. Pérez y J. J. Gutiérrez, «A survey on standards for real-time distribution middleware,» *ACM Computing Surveys,* vol. 46, nº 49, p. 39, Marzo 2014.

[3] OMG, «Realtime Corba Specification. v1.2.,» 2005.

[4] OMG, «Corba Core Specification. v3.2.,» 2011.

[5] ISO/IEC, «Ada 2012 Reference Manual. Language and Standard Libraries—International Standard,» *ISO/IEC,* vol. 8652, 2012.

[6] OMG, «The Real-Time Publish-Subscribe Wire Protocol. DDS interoperability wire protocol specification. v2.1.,» 2009. [En línea]. Available: http://www.omg.org/spec/DDSI/2.1/. [Último acceso: 6 Marzo 2015].

[7] H. Pérez y J. J. Gutiérrez, «On the schedulability of a data-centric real-time distribution middleware.,» *Computer Standards and Interfaces 34.,* pp. 203-211, 2012.

[8] D. C. Schimidt, A. Corsaro y H. V. Hag, «Addressing the challenges of tactical information management in net-centric systems with DDS.,» de *Journal of Defense Software Engineering*, 2008, pp. 24-29.

[9] M. Ryll y S. Ratchev, «Application of the data distribution service for flexible manufacturing automation.,» *International Journal of Aerospace and Mechanical Engineering ,* pp. 193-200, 2008.

[10] M. Gillen, J. Loyall, K. Z. Haigh, R. Walsh, C. Partridge, G. Lauer y T. Strayer, «Information dissemination in disadvantaged wireless communications using a data dissemination service and content data network.,» de *In Proceedings of the SPIE Conference on Defense Transformation and Net-Centric Systems*, 2012.

[11] A. Corsaro, «Advanced DDS Tutorial,» [En línea]. Available: http://www.prismTech.com/dds-community.

[12] G. Pardo-Castellote, «OMG Data-Distribution Service: Architectural Overview,» Real-Time Innovations, Inc..

[13] OMG, «The Real-time Publish-Suscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification Version 2.2,» 2014.

[14] Twin Oaks Computing, Inc., «Interoperable DDS Strategies,» Diciembre 2011. [En línea]. Available: http://www.twinoakscomputing.com. [Último acceso: 17 Marzo 2015].

[15] WIKIPEDIA, «Polling,» 8 Marzo 2013. [En línea]. Available: http://es.wikipedia.org/wiki/Polling. [Último acceso: 11 Marzo 2015].

[16] MSDN, «Futures,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/ff963556.aspx.

[17] MSDN, [En línea]. Available: https://msdn.microsoft.com/es-es/library/bb397687.aspx.

[18] T. Vergnaud, J. Hugues, F. Kordon y L. Pautet, «PolyORB: A schizophrenic middleware to build versatile reliable distributed applications,» *Lecture Notes in Computer Science,* vol. 3063, pp. 106-119, 4 Mayo 2004.

[19] J. L. Campos, J. J. Gutiérrez y M. G. Harbour, «Interchangeable scheduling policies in real-time middleware for distribution,» *Lecture Notes in Computer Science,* vol. 4006, pp. 227-240, 2006.

[20] Y. Kermarrec, «CORBA vs. Ada 95 DSA: A programmer's view,» vol. XIX, pp. 39-46, 1999.

[21] ISO/IEC, S. T. Taft, R. A. Duff, R. Brukardt, E. Ploedereder y P. Leroy, «Ada 2005 Reference Manual. Language and Standard Libraries—International Standard ISO/IEC 8652 (E) with Technical Corrigendum 1 and Amendment 1,» *Lecture Notes in Computer Science,* vol. 4348, 2006.

[22] M. Amoretti, S. Caselli y M. Reggiani, «Designing distributed, component-based systems for industrial robotic applications,» *In Industrial Robotics: Programming, Simulation and Applications,* 2006.

[23] J. Bard y V. J. Kovarik, «Software Defined Radio: The Software Communications Architecture,» 2007.

[24] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst y M. G. Harbour, «Influence of different system abstractions on the performance analysis of distributed real-time systems.,» de *In*

Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07), New York, 2007.

[25] R. I. Davis y A. Burns, «A survey of hard real-time scheduling for multiprocessor systems,» 2011.

[26] C. L. Liu y J. W. Layland, «Scheduling algorithms for multiprogramming in a hard-real-time environments,» de *Journal of the ACM*, 1973.

[27] L. Sha, R. Rjkumar y J. P. Lehoczky, «Priority inheritance protocols: An approach to real-time synchronization,» de *IEEE Transactions on Computers 39*, 1990.

[28] C. Grelck, J. Julju y F. Penczek, «Distributed S-Net: Cluster and grid computing without the hassle,» de *In Proceedings of the 12th IEEE /ACM International Symposium on Cluster, Cloud and Grid Computing(CCGrid)*, 2012.

[29] L. Neumeyer, B. Robbins, A. Nair y A. Kesari, «S4: Distributed stream computing platform,» de *In Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2010.

[30] E. Freeman, S. Hupfer y K. Arnold, «JavaSpaces: Principles, Patterns, and Practice,» 1999.

[31] Sun Microsystems, «JavaTM Message Service Specification. v1.1.,» 2002.

[32] Sun Microsystems, «Java Remote Method Invocation (RMI),» 2004.

[33] K. H. Kim, «Object-oriented real-time distributed programming and support middleware.,» *In Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS),* pp. 10-20, 2000.

[34] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli y U. Scholz, «MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Software Engineering for Self-Adaptive Systems,» *Lecture Notes in Computer Science,* vol. 5525, pp. 164-182, 2009.

[35] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrik, L. Johnston, R. Moreira, N. Parlavantzas y K. Saikoski,

«The design and implementation of open ORB 2.,» *In IEEE Distributed Systems Online,* vol. 2, 2001.

[36] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Tukay, J. Parsons y D. C. Schmidt, «Model driven middleware: A new paradigm for developing distributed real-time and embedded systems.,» *Science of Computer Programming,* vol. 73, pp. 39-58, 2008.

[37] R. Klefstad, D. C. Schmidt y C. O'Ryan, «Towards highly configurable real-time object request brokers,» de *In Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002.

[38] Sun Microsystems, «JSR-50: Distributed Real-Time Specification,» 2000.

[39] G. Bollella y J. Gosling, «The real-time specification for Java,» IEEE Computer, 2000.

[40] Sun Microsystems, «Distributed Real-Time Specification (Early draft),» 2012. [En línea]. Available: http://jcp.org/en/egc/download/drtsj.pdf?id=50&fileId=5028. [Último acceso: 06 Marzo 2015].

[41] D. Tejera, A. Alonso y M. A. de Miguel, «RMI-HRT: Remote method invocation—hard real time.,» *In Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'07).,* pp. 113-120, 2007.

[42] P. Basanta-Val, M. García-Valls y I. Estévez-Ayres, «An architecture for distributed real-time Java based on RMI and RTSJ.,» de *In Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, 2010.

[43] OMG, «Extensible and Dynamic Topic Types for DDS. v1.0.,» 2012. [En línea]. Available: http://www.omg.org/spec/DDS-XTypes/1.0/.

[44] H. Pérez, J. J. Gutiérrez, D. Sangorrín y M. Harbour, «Real-time distribution middleware from the Ada perspective. In Proceedings of the 13th Ada-Europe International Conference on Reliable Software Technologies,» de *Lecture Notes in Computer Science*, 2008.

[45] «Wikipedia,» 09 Marzo 2015. [En línea]. Available: http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling. [Último acceso: 09 Marzo 2015].

[46] «WIKIPEDIA,» 2 Mayo 2014. [En línea]. Available: http://it.wikipedia.org/wiki/Priority_ceiling_protocol. [Último acceso: 9 Marzo 2015].

[47] IEEE, «The Institute of Electrical and Electronics Engineers STD 802.1Q. 2006. Virtual bridged local area networks. Annex G.,» 2006. [En línea]. Available: http://www.ieee802.org/1/pages/802.1Q.html.

[48] M. Aldea, G. Bernat, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. González Harbour, G. Guidi, J. J. Gutiérrez, T. Lennvall, G. Lipari, J. M. Martínez, J. L. Medina, J. C. P. Gutiérrez y M. Trimarchi, «FSF: A real-time scheduling architecture framework.,» *n Proceedings of the IEEE Real Time Technology and Applications Symposium.,* pp. 113-124, 2006.

[49] FRESCOR, «Framework for Real-Time Embedded Systems Based on COntRacts. Project Web page. Retrieved September 2013,» 2006. [En línea]. Available: http://www.frescor.org. [Último acceso: 10 Marzo 2015].

ANEXOS

ANEXO A: GLOSARIO

A

API
: Application Programming Interface · 30, 37, 45, 47, 48, 49, 60, 61, 93, 94, 97, 98, 99, 104, 106, 116, 117, 119, 120, 121, 124, 209, 240

C

CDR
: Common Data Representation · 32, 202, 210

D

DDS
: Data Distributed System · 24, 30, 32, 34, 35, 36, 37, 38, 39, 40, 42, 44, 45, 46, 47, 48, 49, 50, 57, 58, 59, 60, 61, 62, 63, 64, 65, 68, 79, 92, 93, 94, 95, 96, 104, 108, 119, 124, 125, 128, 129, 130, 131, 132, 134, 135, 137, 138, 139, 140, 142, 143, 144, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 167, 170, 178, 180, 187, 188, 189, 193, 194, 209, 210, 211, 213, 214, 234, 235, 236, 237, 238, 239, 240, 249

DDSI
: DDS Interoperability Wire Protocol · 32, 35, 58, 59

DOM
: Distribution based on objects · 25, 27

DR
: Data Reader · 31, 58

DW
: Data Writer · 31, 58, 59

G

GUID
: Globally Unique Identifier · 68, 83, 96, 208

I

IDL
: Lenguaje de definición de interfaces · 40

IP
: Internet Protocol · 32, 37, 44, 46, 65, 99, 167, 189, 190, 198

M

MOM
: Distribution based on messages · 25

O

OMG
: Object Management Group · 24, 48, 94, 240

P

PDP
: Participant Discovery Protocol · 117, 118

PIM
: Plataform Independent Model · 44, 95

Q

QoS
: Quality of Service · 37, 38, 39, 41, 46, 47, 48, 49, 50, 57, 59, 70, 72, 74, 96, 106, 108, 129, 132,

R

U

ANEXO B: MIDDLEWARE

Los anexos se incluyen en el DVD adjunto al presente documento.

- **ANEXO B.1: CÓDIGO FUENTE DEL MIDDLEWARE**
- **ANEXO B.2: MANUAL DE USO DE LA LIBRERÍAS DDS-RTPS**

ANEXO C: APLICACIÓN DE ESCRITORIO

Los anexos se incluyen en el DVD adjunto al presente documento.

- **ANEXO C.1: CÓDIGO FUENTE DEL CHAT RTPS**
- **ANEXO C.2: MANUAL DE USUARIO DEL CHAT RTPS**

ANEXO D: APLICACIÓN CORBA

Los anexos se incluyen en el DVD adjunto al presente documento.

- **ANEXO D.1: CÓDIGO FUENTE CHAT CORBA**

ANEXO E: ESTÁNDAR

Los anexos se incluyen en el DVD adjunto al presente documento.

- **ANEXO E.1: THE REAL-TIME PUBLISH-SUSCRIBE PROTOCOL (RTPS) DDS INTEROPERABILITY WIRE PROTOCOL SPECIFICATION.**
- **ANEXO E.2: DATA DISTRIBUTION SERVICE FOR REAL-TIME SYSTEM SPECIFICATION (DDS).**

ANEXO F: PRUEBAS UNITARIAS

Los anexos se incluyen en el DVD adjunto al presente documento.

- **ANEXO F.1: STACK DE PRUEBAS UNITARIAS DE CLASES Y MÉTODOS DEL PROYECTO DE TITULACIÓN.**