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

ALEJANDRA BEATRIZ TELLO GONZÁLEZ
alejitat_28@hotmail.com

ANDRÉS XAVIER RUBIO PROAÑO
andresrubiop@msn.com

DIRECTOR: ING. XAVIER CALDERÓN, MSc.
xavier.calderon@epn.edu.ec

Quito, Agosto 2015

ESCUELA POLITÉCNICA NACIONAL – EPN

CARRERA DE INGENIERÍA EN ELECTRÓNICA Y REDES DE INFORMACIÓN

AUTORÍA DE RESPONSABILIDAD

Alejandra Beatriz Tello González

Andrés Xavier Rubio Proaño

El proyecto de titulación denominado **"Desarrollo de un módulo que implemente las funcionalidades del protocolo RTPS para ser utilizado en aplicaciones distribuidas de tiempo real."** ha sido desarrollado con base a una investigación exhaustiva, respetando derechos intelectuales de terceros, cuyas fuentes se incorporan en la bibliografía.

Consecuentemente este trabajo es de nuestra autoría.

En virtud de esta declaración, nos responsabilizamos del contenido, veracidad y alcance científico del proyecto de grado en mención.

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

Quito, Agosto de 2015

Andrés Xavier Rubio Proaño	Alejandra Beatriz Tello González

ESCUELA POLITÉCNICA NACIONAL - EPN

CARRERA DE INGENIERÍA EN ELECTRÓNICA Y REDES DE INFORMACIÓN

CERTIFICADO

Ing. Xavier Calderón, MSc.

CERTIFICA

Que el trabajo titulado **"Desarrollo de un módulo que implemente las funcionalidades del protocolo RTPS para ser utilizado en aplicaciones distribuidas de tiempo real."**, realizado por Alejandra Beatriz Tello González y Andrés Xavier Rubio Proaño, ha sido guiado y revisado periódicamente y cumple normas estatuarias establecidas por la EPN, en el Reglamento de Estudiantes de la Escuela Politécnica Nacional - EPN.

Debido a que este trabajo cumple con los requisitos establecidos por la institución, se recomienda su publicación.

El mencionado trabajo consta de dos documentos empastados y dos discos compactos (CD) los cuales contienen los archivos en formato PDF. Autorizan a Alejandra Beatriz Tello González y Andrés Xavier Rubio Proaño que lo entregue al Ing. David Mejía, en su calidad de Coordinador de la Carrera de Ingeniería en Electrónica y Redes de Información.

Quito, Agosto de 2015

_____

Ing. Xavier Calderón. MSc.

Director

ESCUELA POLITÉCNICA NACIONAL – EPN

CARRERA DE INGENIERÍA ELECTRÓNICA Y REDES DE INFORMACIÓN

AUTORIZACIÓN

Nosotros, Alejandra Beatriz Tello González y Andrés Xavier Rubio Proaño

Autorizamos a la Escuela Politécnica Nacional - EPN la publicación, en la biblioteca virtual de la institución, el trabajo **"Desarrollo de un módulo que implemente las funcionalidades del protocolo RTPS para ser utilizado en aplicaciones distribuidas de tiempo real."**, cuyo contenido, ideas y criterios son de nuestra exclusiva responsabilidad y autoría.

Quito, Agosto de 2015

Andrés Xavier Rubio Proaño

Alejandra Beatriz Tello González

DEDICATORIA

Alejandra Beatriz Tello González

DEDICATORIA

Alejandra Beatriz Tello González

AGRADECIMIENTO

AGRADECIMIENTO

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

Aquí va el resumen

Palabras Clave:

1. CAPÍTULO I

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 comunicaciones de tiempo real y se presenta un estudio de cada tecnología encontrada incluyendo al middleware DDS. Posteriormente se realiza una comparación entre las tecnologías descritas anteriormente, donde se detalla las ventajas y desventajas del uso de cada una de las mismas. Finalmente se analiza características y funcionalidades más específicas definidas en el estándar publicado por la OMG sobre DDS y su interoperabilidad con el protocolo RTPS.

1.2. MIDDLEWARES

El middleware es una capa de software intermedio, el cual se encarga de simplificar el manejo y la programación de aplicaciones, tratando de mantener la complejidad de redes y sistemas heterogéneos transparentes al usuario, por lo que se ha convertido en una herramienta esencial para el desarrollo de sistemas distribuidos.

El concepto de middlewares es muy amplio y cuenta con varias funcionalidades:

- Middleware de Comunicaciones, el cual es una abstracción de los detalles de bajo nivel relacionados con la distribución y la comunicación.
- Middleware de Componentes (Klefstad, Schmidt, & O'Ryan, 2002), el cual se basa en un modelo formal que permite el desarrollo de sistemas mediante el ensamblaje de módulos de software reutilizables (componentes), los cuales han sido desarrollados previamente por otros, independientemente de la aplicación que será utilizada.

- Middleware basado en modelos o Middleware MD[1] (Gokhale, et al., 2008), el cual se centra principalmente en la consecución de un proceso de desarrollo sostenible en términos de costos, tiempos de desarrollo, y la calidad, combinando un middleware de componentes con el desarrollo de software basado en modelos.
- Middleware Adaptativo (Blair, et al., 2001), el cual permite la reconfiguración de aplicaciones distribuidas para modificar funcionalidades, el uso de recursos, configuraciones de seguridad, etc.
- Middleware Sensible al Contexto (Rouvoy, et al., 2009), el cual es capaz de interactuar con el entorno en el que las aplicaciones distribuidas se ejecutan y toman acción para hacer cambios en el tiempo de ejecución.

Profundizando en el ámbito de comunicaciones, se toma el primer grupo anteriormente descrito, que corresponde a los Middleware de Comunicaciones, el cual proporciona las bases para el desarrollo de middlewares de alto nivel. Este tipo de middlewares maneja internamente los detalles del proceso de interconexión entre nodos que por lo general incluye las siguientes características básicas:

- Direccionamiento o asignación de identificadores a entidades con la finalidad de indicar su ubicación.
- Marshalling[2] o transformación de los datos en una representación adecuada para la transmisión sobre la red.

[1] Middleware MD, corresponde al Model-Driven Middleware en su traducción al español.
[2] Marshalling, es un mecanismo ampliamente usado para transportar objetos a través de una 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.

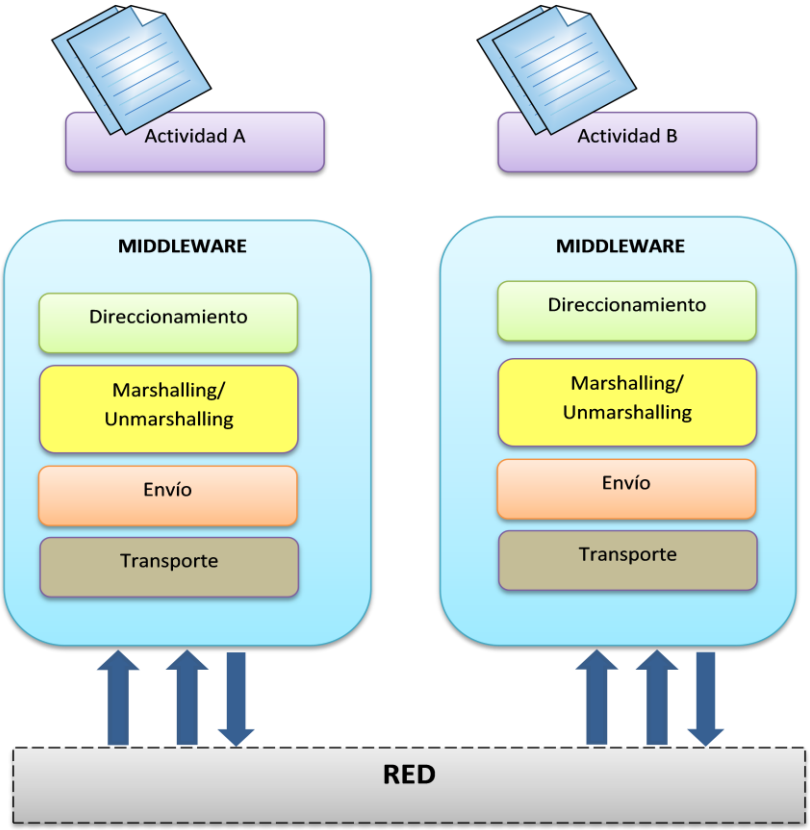


Figura 1-1. Servicios básicos provistos por el middleware de distribución

1.3. SISTEMAS DISTRIBUIDOS

Debido a la estabilidad y a su impacto en la industria, los middlewares de distribución están estandarizados. Existen varios modelos de sistemas distribuidos donde la comunicación ha sido abstraída dentro de middlewares, entre estos modelos tenemos a:

- RPC[3]

[3] RPC. Remote Procedure Calls

- DOM[4]
- MOM[5]
- Data-Centric Model
- Tuplespaces paradigm[6]

Dentro de estos modelos de sistemas distribuidos se tiene como ejemplos más representativos las siguientes tecnologías detalladas en la Tabla 1-1.

Tabla 1-1. 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) (ISO/IEC, et al., 2006)
DOM (Kim, 2000)	• Common Object Request Broker Architecture (CORBA) (OMG, Corba Core Specification. v3.2., 2011) • Java Remote Method Invocation (RMI) (Sun Microsystems, 2004) • Distributed Systems Annex of Ada (DSA)
MOM	• Java Message Service (JMS) (Sun Microsystems, 2002) • Data Distribution Service for Real-Time Systems (DDS) (OMG, Data Distribution Service for Real-Time Systems. v1.2., 2007)
Data-Centric Model	• Data Distribution Service for Real-Time Systems (DDS)

[4] DOM. Distribution based on objects
[5] MOM. Distribution based on messages
[6] Tuplespaces paradigm. Distributed Stream Computing

Tuplespaces paradigm	• JavaSpaces (Freeman, Hupfer, & Arnold, 1999) • Simple Scalable Streaming System (S4) (Neumeyer, Robbins, Nair, & Kesari, 2010) • S-NET (Grelck, Julju, & Penczek, 2012)

1.4. MIDDLEWARES DE TIEMPO REAL

A diferencia de los sistemas de propósito general, un sistema de tiempo real se define como un tipo especial de sistema cuya corrección lógica se basa tanto en la exactitud como también en la disminución de retardos en la información. Sin embargo, no es suficiente que el software haya sido programado con una lógica correcta; las aplicaciones también deben satisfacer determinadas restricciones temporales. Para este fin, las aplicaciones en tiempo real se basan en un esquema de planificación para especificar un criterio para ordenar el uso de recursos del sistema.

El problema de obtener predicciones temporales computacionalmente factibles, fiables y precisas, puede ser resuelto mediante la aplicación de diferentes técnicas analíticas para un solo procesador, multiprocesador o sistemas de tiempo real distribuido (Davis & Burns, 2011).

En el caso de los sistema distribuidos, este proceso es un reto incluso para los sistemas distribuidos aparentemente simples en donde las dependencias complejas entre los datos o subprocesos asignados en diferentes procesadores podrían estar presentes, y por lo tanto, las redes y los procesadores se debe programar juntos (Perathoner, et al., 2007) (Liu & Layland, 1973) (Sha, Rjkumar, & Lehoczky, 1990).

En los sistemas de propósito general, el uso de la tecnología de middlewares tiene como objetivo facilitar la programación de aplicaciones distribuidas. Con este fin, el middleware proporciona una abstracción de alto nivel de los servicios ofrecidos por los sistemas operativos, sobre todo los relacionados con la comunicación. Por lo tanto, los desarrolladores solo son

responsables de definir que parte de la aplicación puede ser accesible de forma remota, mientras el middleware establece y gestiona transparentemente la comunicación entre los nodos del sistema distribuido. Además, los sistemas en tiempo real también se benefician de estas abstracciones de alto nivel.

Sin embargo, los middlewares de propósito general no se pueden aplicar directamente a sistemas de tiempo real. En general, el proceso de distribución presenta varias posibles fuentes de indeterminismo, incluyendo los algoritmos de marshalling/unmarshalling, transmisión y recepción de colas para mensajes de red, retrasos en el servicio de transporte, o en él envió de solicitudes.

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[7]. En consecuencia, este tipo de middleware se dirige no solo 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.

1.4.1. CORBA y RT-CORBA[8]

CORBA (OMG, Corba Core Specification. v3.2., 2011) es un middleware basado en el modelo de sistema distribuido basado en DOM, el cual utiliza el paradigma Cliente-Servidor y cuya característica principal es facilitar y la interoperabilidad entre aplicaciones heterogéneas [9]. CORBA fue desarrollado por un consorcio industrial llamado OMG[10], una visión general de la arquitectura CORBA se muestra en la figura 1-2.

[7] Planificación, se refiere al término scheduling.
[8] RT-CORBA, CORBA de tiempo real
[9] 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.
[10] OMG, Object Management Group.

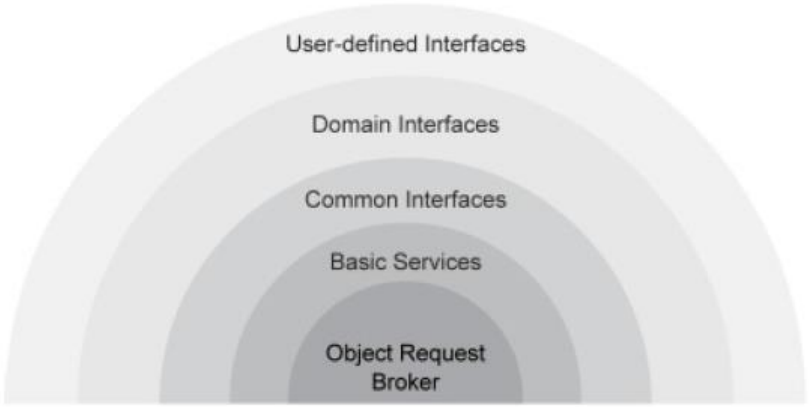


Figura 1-2. Arquitectura de CORBA

(Pérez & Gutiérrez, 2014)

Esta arquitectura está integrada por los siguientes componentes:

- *Object Request Broker*, ORB representa el núcleo del middleware y es responsable de coordinar la comunicación entre los nodos cliente y servidor.
- *Interfaces del Sistema*, Estas consisten en un conjunto de interfaces agrupadas en función de su ámbito de aplicación, que incluyen:
    - Una colección de servicios básicos, que dan soporte al ORB, por ejemplo, la ubicación de objetos remotos, la concurrencia y la persistencia.
    - Un conjunto de interfaces comunes a través de una amplia gama de dominios de aplicación, por ejemplo, la administración de bases de datos, la compresión de la información y la autenticación.
    - Un conjunto de interfaces para un dominio particular de la aplicación, por ejemplo, las telecomunicaciones, la banca y las finanzas.
    - Interfaces definidas por Usuario, las cuales no se encuentran estandarizadas.

Puesto que no hay software, sistema operativo, o lenguaje de programación que reúna todos los requisitos industriales, el objetivo principal de CORBA es proporcionar soluciones para dar soporte a los sistemas heterogéneos, basándose en dos aspectos básicos:

- *Middleware con independencia de lenguaje o Multilenguaje*, Los objetos CORBA están definidos por el uso de un lenguaje de descripción llamado Interface Definition Languaje o IDL [11]. Actualmente, dentro del estándar CORBA existen las normas para la asignación de tipos de datos para varios lenguajes de programación tales como Ada, java o C.
- *Middleware con independencia de plataforma o Interoperabilidad*, CORBA define un protocolo de transporte genérico denominado General Inter-ORB Protocol o GIOP. Este protocolo garantiza la interoperabilidad entre objetos CORBA independientemente de si se asignan a los ORB de diferentes fabricantes o para diferentes plataformas. El protocolo Internet Inter-ORB o IIOP es la especificación de asignación del protocolo GIOP sobre redes TCP/IP, que son consideradas el transporte base para las implementaciones en CORBA.

La comunicación entre los nodos es llevada a cabo mediano el uso de varias entidades CORBA como se ilustra en la Figura 1-3 y se describe a continuación.

[11] IDL, Lenguaje de definición de interfaces.

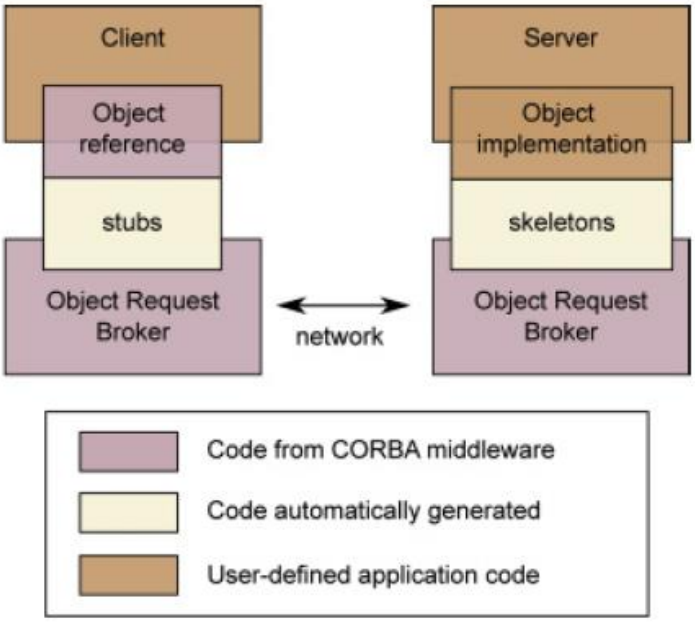


Figura 1-3. Comunicación entre entidades CORBA

(Pérez & Gutiérrez, 2014)

- *Object Request Broker*, El ORB proporciona mecanismos para invocar de forma transparente un método remoto como si se tratara de un método local. Por lo tanto el ORB abstrae la ubicación de los objetos remotos y el método de comunicación con ellos.
- *Cliente Stubs y Servidor Skeletons*, Estos representan las partes del código, que por lo general son generados automáticamente, a cargo de redirigir la llamada remota a través del ORB, así como la realización de las operaciones de marshalling y unmarshalling.
- *Object Reference*, Esta no es una referencia clara que únicamente determina la localización de un objeto remoto y se la conoce como Interoperable Object Referenc o IOR. El IOR incluye detalles de todos los protocolos de red y puertos receptores que el ORB puede utilizar para procesar las solicitudes entrantes. Esta referencia es generada y gestionada por el Portable Object Adapter o POA.

- *Redes de Comunicación*, Los nodos , el cliente y el servidor se comunican a través del ORB usando el protocolo GIOP. Este protocolo está en la parte superior de la capa transporte del modelo OSI[12] y puede ser implementado en la parte superior de varios protocolos de red; sin embargo, el estándar CORBA solo incluye directrices para implementarlo en redes basadas en IP.

Aunque CORBA, proporciona soporte completo para objetos distribuidos, este estándar no incluye soporte para aplicaciones en tiempo real. Por lo tanto, esta falta de soporte fue abordada por la OMG a través de un conjunto opcional de extensiones para CORBA que fueron llamadas RT-CORBA (OMG, Realtime Corba Specification. v1.2., 2005). Estas extensiones se describen a continuación:

- *RT-ORB*, Una extensión ORB, que añade funciones para la creación y la destrucción de entidades específicas de tiempo real, por ejemplo, los Mutex, los threadpools, o las políticas planificación y permite la asignación de prioridades para su uso por hilos internos ORB.
- *RT-POA*, Representa una extensión del POA (OMG, Corba Core Specification. v3.2., 2011) y provee soporte para la configuración de las políticas de tiempo real definidas por RT-CORBA. Estas políticas manejan los modelos de prioridad de propagación de extremo a extremo, la gestión de llamadas remotas, la prioridad en conexiones agrupadas, o la selección y configuración de los protocolos de red disponibles.
- *Prioridad y Asignación de Prioridad*, estas representan un interfaz que definen un tipo de datos de prioridad genérica, es decir independientemente del sistema operativo, y proporcionan operaciones para asignar prioridades nativas dentro de las prioridades RT-CORBA y viceversa.

[12] OSI, Open System Interconnection

- *Mutex*, es una interfaz portable para acceder a los Mutex proporcionados por la RT-ORB. Esta proporciona mecanismos de sincronización para controlar el acceso a los recursos compartidos.
- *RTCurrent*, es una interfaz para determinar la prioridad de la invocación actual, es decir, permite a la prioridad de los hilos de aplicaciones que sean manejados.
- *ThreadPool*, es un mecanismo para controlar el grado de concurrencia durante la ejecución de las llamadas remotas en el lado del servidor.
- *Servicio de Planificación*, es un servicio que simplifica la configuración de aspectos de sincronización del sistema. Por medio de este servicio, RT-CORBA permite que la aplicación especifique sus requerimientos en función de diversos parámetros tales como las prioridades, lazos, o plazos de ejecución previstos, mientras que el middleware será responsable de la creación de los recursos necesarios para cumplir con ellos.

El uso de estas entidades RT-CORBA permite el desarrollo de sistemas de tiempo real críticos como sistemas de control en tiempo real, así como sistemas no críticos como el de las agencias de viajes o sistemas de compra en línea. Actualmente RT-CORBA se emplea en una amplia gama de escenarios tales como radios definidos por software (Bard & Kovarik, 2007) o robótica industrial (Amoretti, Caselli, & Reggiani, 2006) y puede considerarse una tecnología madura.

1.4.2. The Ada Distributed Systems Annex

El lenguaje de programación Ada (ISO/IEC, Ada 2012 Reference Manual. Language and Standard Libraries—International Standard, 2012) es un estándar internacional que incluye un anexo dedicado al desarrollo de aplicaciones distribuidas, el cual corresponde al anexo E o Ada DSA. La principal fortaleza de DSA es que el código fuente está escrito sin tener en cuenta de si va a ser ejecutado en una plataforma distribuida o en un solo procesador.

En el diseño de sistemas distribuidos, una aplicación diseñada para un solo procesador puedes ser dividido en diferentes funcionalidades tales que, cuando actúen juntos puedan proporcionar una servicio particular para usuarios finales. La ejecución de cada una de estas funcionalidades pueden ser distribuidas a través de varios nodos interconectados, mientras que en los usuarios finales se invoca de forma transparente el servicio. 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." (ISO/IEC, Ada 2012 Reference Manual. Language and Standard Libraries—International Standard, 2012)

El particionamiento de una aplicación de la DSA no está definido en el estándar, pero su implementación está definida. Las particiones se comunican entre sí mediante el intercambio de datos a través de las RPC y de los objetos distribuidos. La DSA define dos tipos de particiones: activos, los cuales pueden ser ejecutados en paralelo uno con otro, posiblemente en direcciones separadas de memoria y en computadores independientes; y pasivos, los cuales son particiones sin una tarea o hilo de control, por ejemplo, los nodos de almacenamiento.

Las particiones activas se comunican a través del subsistema de comunicación de particiones o PCS, un interfaz definido por lenguaje es responsable del subprograma de enrutamiento de llamadas de una partición a otra. El acceso a la PCS no debe hacerse directamente desde la capa aplicación, sino este debe hacerse por medio de los Stubs de llamadas y recepción. El PCS soporta compiladores usados para generar Stubs de una interfaz estándar sin preocuparse de la implementación. A pesar de este esfuerzo de normalización , una reciente revisión del lenguaje de programación (ISO/IEC, et al., 2006) permite el uso de interfaces alternativas para PCS con el fin de facilitar la interoperabilidad con otro middleware.

Los componentes de alto nivel del modelo de distribución propuesto por la DSA están ilustrados en la Figura 1-4. Esta figura representa el diagrama de secuencia de una llamada remota síncrona entre dos particiones: una partición que requiere servicios remotos y una partición que proporciona estos servicios a través de un interfaz de llamada remota.

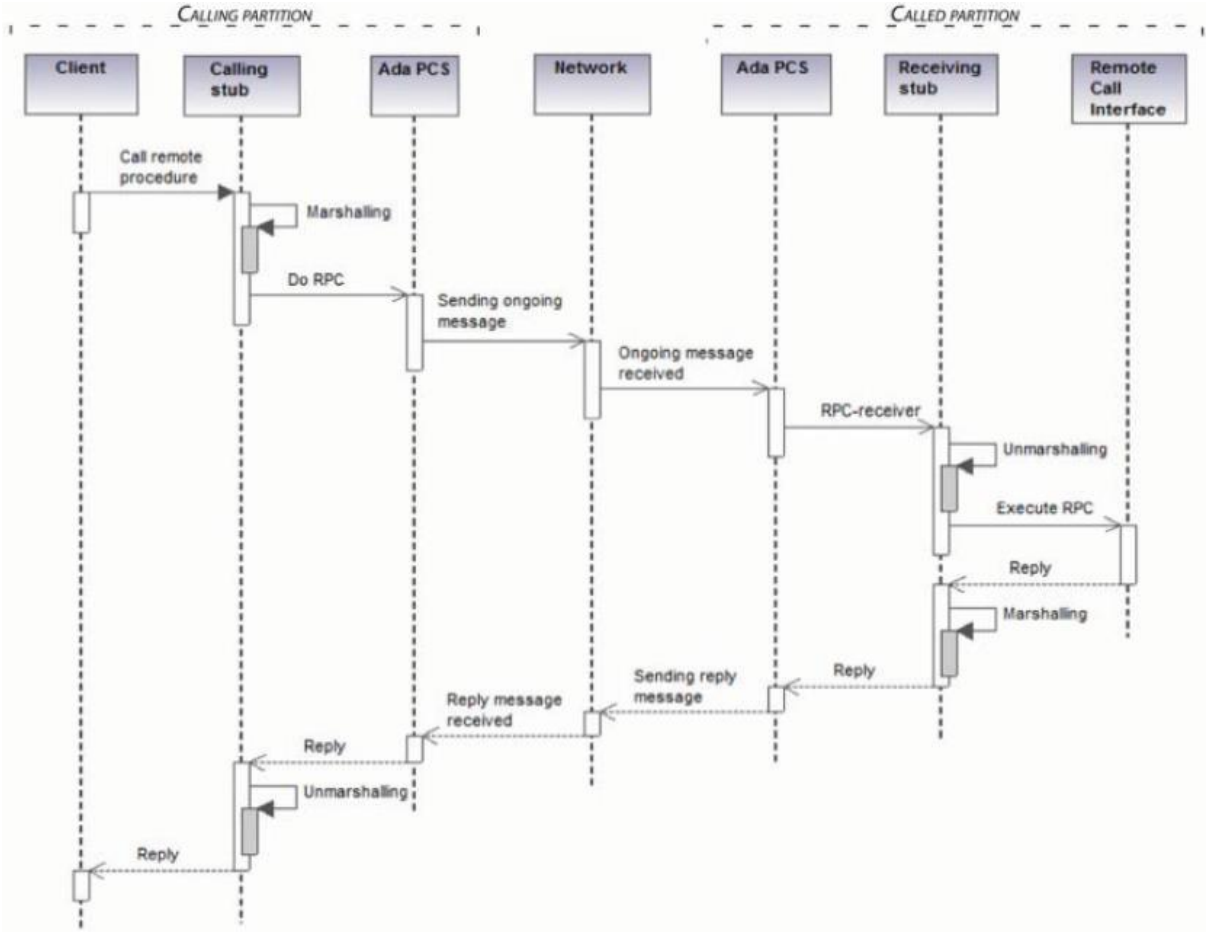


Figura 1-4. Diagrama de secuencia de una llamada remota síncrona.

(Pérez & Gutiérrez, 2014)

Aunque la DSA permite que los sistemas distribuidos sean construidos de manera simple, no están específicamente diseñados para soportar aplicaciones previsibles, y la mayoría de los problemas que afectan el determinismo se han quedado en la implementación. Sin embargo, hay existen algunas investigaciones previas en este campo, y hay implementaciones que muestran que pueden ser usadas para aplicaciones de tiempo real. Aunque este anexo no ha tenido un impacto comercial muy significativo (Vergnaud, Hugues, Kordon, & Pautet, 2004)

(Campos, Gutiérrez, & Harbour, 2006). Aunque este anexo no ha tenido un impacto comercial muy significativo (Kermarrec, 1999), Ada ha sido tradicionalmente usado y aun se usa para construir los sistemas de un solo procesador en tiempo real, así que vale la pena considerar el análisis de esta norma y su desarrollo futuro.

1.4.3. The Distributed Real-Time Specification for Java

Además de las normas de distribución, hay otras soluciones no estandarizadas que han despertado gran interés entre los desarrolladores. Este es el caso del lenguaje de programación Java y sus extensiones para sistemas distribuidos en tiempo real, el cual es un estándar de facto. Java fue diseñado inicialmente como un lenguaje de programación para sistemas de propósito general y, por lo tanto, tiene varios inconvenientes para el desarrollo de aplicaciones previsibles, especialmente aquellos aspectos relacionados con la gestión de los recursos internos como la memoria o la programación del procesador (Basanta-Val, García-Valls, & Estévez-Ayres, 2010). Para los sistemas distribuidos de tiempo real, uno de los trabajos de investigación más notable es la Distributed Real-Time Specification for Java o DRTSJ (Sun Microsystems, 2000), que integra dos tecnologías existentes de Java:

- *Real-Time Specification for Java o RTSJ* (Bollella & Gosling, 2000), el cual define una especificación de Java para abordar las limitaciones del lenguaje cuando se usa en sistemas de tiempo real. Como una de las principales directrices fue evitar hacer extensiones sintácticas del lenguaje, se logró el soporte en tiempo real a través de nuevas bibliotecas, un mejor mecanismo de Java, y una Máquina Virtual de Java en tiempo real o JVM con soporte tanto para de propósito general y aplicaciones en tiempo real. Sin embargo, esta especificación fue concebida solo para sistemas de un solo procesador.
- *Remote Method Invocation o RMI* (Sun Microsystems, 2004), el cual define un modelo DOM basado en objetos Java que definen una nueva interfaz, llamada remota, lo que

permite la diferenciación de los objetos distribuidos de los locales. Un visión general del alto nivel de los componentes implicados en la arquitectura RMI, la cual se muestra en la Figura 1-5, la cual representa el diagrama de secuencia de una llamada remota asíncrona entre un cliente y un servidor. Esta arquitectura tiene los siguientes componentes.

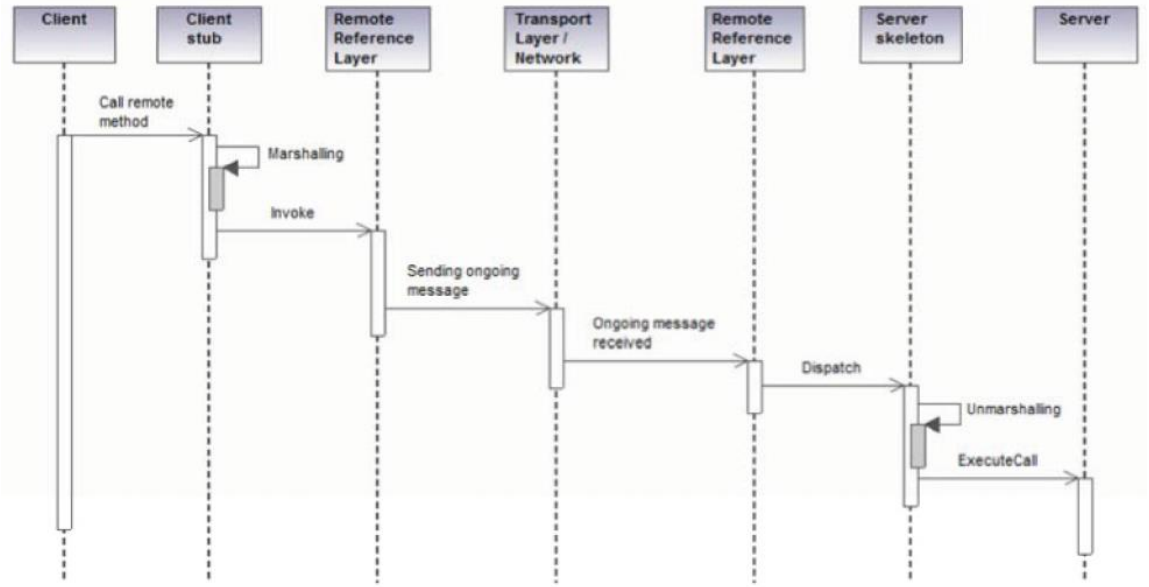


Figura 1-5. Diagrama de secuencia de una llamada remota asíncrona.

- *Cliente Stubs o Proxy y Servidor Skeleton,* los cuales representan la interfaz entre la capa aplicación y el resto del sistema RMI, son los encargados de proporcionar servicios de distribución transparentes.
- *Remote reference Layer,* el cual se encarga de manipular la semántica de las invocaciones remotas, tanto en la parte del cliente como en el servidor.
- *Capa Transporte,* el cual se utiliza para establecer las conexiones y gestionar los detalles de la comunicación de bajo nivel. La especificación define la conexión del protocolo RMI, que se basa en dos protocolos: Serialización de Objetos Java y HTTP.

Aunque Java es uno de los lenguajes de programación más populares, no define la especificación DRTSJ, aún no se ha difundido, aunque existe un primer borrador (Sun Microsystems, 2012) y en el sitio Web del grupo de trabajo (Sun Microsystems, 2000), la cual describe las características importantes de la futura especificación. Sin embargo, varias líneas de la investigación pretenden adaptar el lenguaje de programación a un modelo determinístico no solo para ambientes con un solo procesador, sino también para los distribuidos (Tejera, Alonso, & de Miguel, 2007) (Basanta-Val, García-Valls, & Estévez-Ayres, 2010).

1.4.4. The Data Distribution Service for Real-Time Systems

La difusión asincrónica de la información ha sido un requisito común para varias aplicaciones distribuidas, tales como sistemas de control, sensores de redes y los sistemas de automatización industrial. El DDS (OMG, Data Distribution Service for Real-Time Systems. v1.2., 2007) tiene como objetivo facilitar el intercambio de datos en este tipo de sistemas a través del paradigma publicador-suscriptor. A diferencia de otros las especificaciones que siguen este paradigma, la comunicación está centrada en los datos propuesto por el modelo DDS, es decir, la atención se centra en los datos en sí. En una arquitectura data-centric se debe formalmente definir el tipo de datos que se comparte en la periferia del sistema, y luego la información se intercambia de forma anónima simplemente escribiendo y leyendo este tipo de datos. Con un enfoque centrado en los datos, el middleware es consciente del contenido de la información intercambiada y de lo que puede manejar directamente él, por ejemplo, la filtración de datos.

Al igual que con la mayoría de los estándares se define dentro de la OMG, el DDS soporta multilenguaje y multiplataforma mediante el uso del lenguaje IDL (OMG, Corba Core Specification. v3.2., 2011) para definir tipos de datos compartidos y el DDSI[13] (OMG, 2009) para interoperar entre diferentes implementaciones, respectivamente. Más allá de esto, la OMG

[13] DDSI, DDS Interoperability Wire Protocol

tiene publicado el Extensible and Dynamic Topic Types specification (OMG, 2012), que proporciona apoyo a los sistemas distribuidos extensibles y evolutivos utilizando DDS. Esta especificación permite a los tipos de datos ser definidos dinámicamente, es decir pueden ser utilizados sin compilación, o modificados, es decir, los campos de los datos se pueden agregar o quitar. Para este fin, el DDS proporciona un sistema de datos estructurales, nuevas representaciones de tipos de datos, diferentes formatos de serialización o de codificación, y una nueva API[14] para la gestión de los tipos de datos en tiempo de ejecución.

El modelo conceptual DDS se basa en la abstracción de un tipo de datos globales, donde el Publicador y el Suscriptor escriben (producir) y leen (consumir) datos, respectivamente; lo que lleva al middleware centralizado a la obtención de datos de forma independiente de su origen.

Para manejar mejor el intercambio de datos, el estándar define un conjunto de entidades que participan en el proceso de comunicación. Las aplicaciones que desean compartir información con otras, pueden utilizar este espacio de datos globales y declarar su intención de publicar los datos a través de la entidad DW[15]. Del mismo modo, las aplicaciones que necesiten recibir información pueden utilizar la entidad DR[16] para solicitar datos particulares. Las entidades Publicadoras y Suscriptoras son contenedoras de los DW y DR, respectivamente, los cuales comparten los parámetros de QoS[17] comunes. Del mismo modo, estas entidades se agrupan en un Dominio. Solo las entidades que pertenecen al mismo dominio pueden comunicarse. En capas superiores, todos las entidades participantes contienen todos los DW y DR, Publicador y Suscriptor que compartan un QoS común en el Dominio correspondiente.

[14] API, Application Programming Interface o La interfaz de programación de Aplicaciones
[15] DW, DataWriter.
[16] DR, DataReader.
[17] QoS, Quality of Service o Calidad de Servicio.

Para intercambiar información entre las entidades, el Publicador solo necesita saber acerca de un tema específico, como por ejemplo, el tipo de dato para compartir y el Suscriptor requiere el registro de su interés en recibir determinados temas, mientras que el middleware puede establecer y gestionar la comunicación de forma transparente. Dentro de la definición de un tema, uno o más elementos pueden ser designados como una *llave.* Esta entidad permite la existencia de múltiples instancias del mismo tema, permitiendo así que los DR diferencien la fuente de origen de los datos, por ejemplo, un grupo de sensores de temperatura que proporcionan información de diferentes áreas. La Figura 1-6 muestra un sistema distribuido que consta de tres participantes en un solo Dominio y dos tópicos.

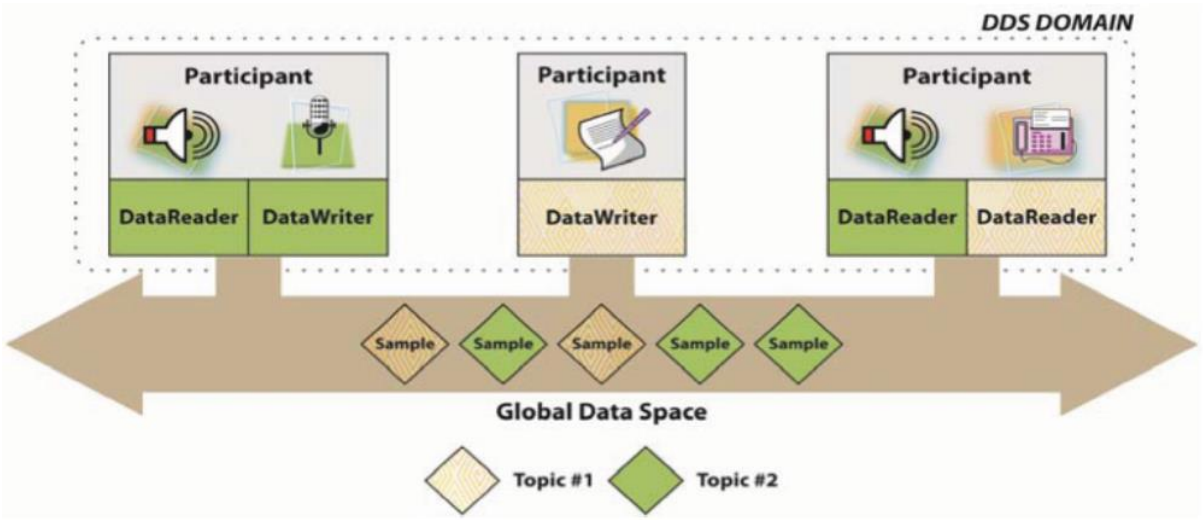


Figura 1-6. Sistema Distribuido que consta de tres participantes en un solo Dominio

Ambos tópicos tienen un solo DW, el cual es el encargado de generar nuevo grupos de datos. Sin embargo, las actualizaciones sucesivas del Tópico #1 solo serán recibidos por un solo 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:

- *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[18] (OMG, 2009) en tiempo real del Publicador-Suscriptor, junto con la Representación de Datos Común o CDR, el cual se define en CORBA (OMG, Corba Core Specification. v3.2., 2011). 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 (Pérez & Gutiérrez, On the schedulability of a data-centric real-time distribution middleware., 2012). Sin embargo se considera una tecnología madura y ya se ha desplegado en varios escenarios en tiempo real, tales como sistemas de Defensa Nacional (Schimidt, Corsaro, & Hag, 2008), Automatización (Ryll & Ratchev, 2008), o Espacio (Gillen, y otros, 2012).

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

Después de analizar los diferentes estándares de distribución orientadas al desarrollo de aplicaciones en tiempo real, se halla las semejanzas y diferencias entre estas tecnologías, así

[18] RTP, Real-Time Transport Protocol

como evaluar 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:

- *El soporte para la planificación en procesadores y redes,* los sistemas de tiempo real requieren un estricto control de la ejecución de hilos y de la transmisión de mensajes. Esto incluye el soporte para las políticas de planificación encargadas de ordenar el acceso concurrente de los hilos y de los mensajes a los procesadores y redes de comunicación, respectivamente.
- *Control de parámetros de planificación,* el Middleware debe proveer mecanismos para configurar los parámetros de planificación de cada hilo que pueden ser ejecutados en el procesador. Igualmente, la asignación de parámetros de planificación a mensajes de red debe ser soportada.
- *Gestión de Hilos o Patrones de concurrencia,* representa el patrón de diseño que trata sobre el paradigma de multi-hilos, que controla y procesa la difusión de la información. Esta característica es especialmente importante en el lado del receptor.
- *El acceso controlado a recursos compartidos,* esto puede lograr a través de la aplicación de protocolos de sincronización.

Hay dos tipos de entidades de planificación que pueden ser identificadas en sistemas de tiempo real: Hilos para procesadores y Mensajes para redes de comunicaciones. Además, esas entidades también pueden producir algún tipo de contención que debe ser tenido en cuenta en el diseño en tiempo real.

1.5.1. Gestión de los recursos del procesador

El comportamiento temporal de middleware de distribución está fuertemente determinado por las políticas de planificación y los patrones de concurrencia (Pérez H. ,

Gutiérrez, Sangorrín, & Harbour, 2008). En el caso de las políticas de planificación, es necesario identificar cuales mecanismos están provistos en 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 trata con las opciones disponibles para determinar cuál hilo es responsable para el envío o recepción de peticiones/ datos remotos.

Primeramente es posible que los planificadores estén soportados directamente en el sistema operativo. Sin embargo mientras estos estándares de distribución tienen por objetivo el desarrollo de sistemas de tiempo real, sería conveniente incluir las operaciones en sus APIs para establecer políticas de planificación específicas y sus correspondientes parámetros de planificación de los hilos del middleware.

Ambas especificaciones Ada y RT-CORBA dan soporte a diferentes políticas de planificación, incluyendo las políticas FPS[19]. Igualmente, el marco de planificación definido por DRTSJ puede ser ampliado con el fin de soportar diferentes políticas de planificación. Sin embargo, el modelo propuesto en DDS no incluye planificación en los procesadores lo cual no está definido en el estándar. Aunque el estándar DDS define varios parámetros temporales ninguno es apropiado para construir hilos en el procesador: parámetro *DeadLine*[20] que puede ser utilizados en algunos casos, como por ejemplo, sistemas EDF[21] (Wikipedia, 2015), pero el estándar en el estándar no se considera su uso. Una situación similar se da con el parámetro Latency_Budget y su definición no es clara, aunque su especificación propone una dosificación de datos, como por ejemplo, reuniendo un conjunto de muestras de datos a ser enviado en un solo paquete de red de gran tamaño.

[19] FPS, Fixed Priorities Schudeling o Planificación de prioridades fijas
[20] Deadline, fija la separación máxima de tiempo entre dos actualizaciones.
[21] EDF, Earliest deadline first, es un algoritmo de planificación dinámico usado en sistemas operativos de tiempo real para procesar consultas de prioridad.

RT-CORBA es la única especificación que proporciona mecanismos para especificar los parámetros de planificación que se utilizarán durante la ejecución de las operaciones solicitadas en el nodo remoto. La especificación para sistemas estáticos define dos políticas, Server_Declared y Client_Propagated, que impone restricciones en la asignación de prioridades y luego reduce la planificabilidad del sistema (Campos, Gutiérrez, & Harbour, 2006). Además, aunque el modelo de transformación de prioridades permite la modificación de las políticas, esta se lleva a cabo dentro del código de aplicación suministrado, por lo que cualquier cambio en los parámetros de planificación debe ser revisado.

El procesamiento de llamadas remotas o datos entrantes representan un proceso de dos etapas que incluye: la escucha de eventos de E/S en las redes de comunicación y el procesamiento de mensajes de red y la ejecución del código de la aplicación asociado a llamadas remotas o datos entrantes, y una posible respuesta. La escucha de eventos de E/S es internamente realizado y controlada por el middleware, mientras que la ejecución del código de aplicación se refiere a la interacción entre el middleware y la aplicación. Los estándares de distribución no definen que patrón de concurrencia debe ser usado en la parte de escucha de eventos pero especifica que implementación debe atender las solicitudes remotas concurrentes, por ejemplo, el Ada DSA indica explícitamente este aspecto, mientras que RT-CORBA implícitamente este promedio de la definición de los *Threadpools*. En cuanto a la ejecución del código de aplicación, RT-CORBA y Ada DSA dirigen la ejecución del código de la aplicación en el contexto de un hilo interno del middleware, mientras DDS permite el uso de hilos internos del middleware, por medio del mecanismo de escucha, o la aplicación de hilos, a través de estructuras de espera y establecimiento o por pedido de la disponibilidad de los datos en un DR. De todos modos, independientemente del hilo o hilos responsables del procesamiento de cada etapa, es importante que el middleware proporcione los mecanismos necesarios para controlar sus parámetros de planificación.

Finalmente, el acceso determinístico de los recursos compartidos evita el problema de inversión de prioridades (Sha, Rjkumar, & Lehoczky, 1990). RT-CORBA, Ada y RTSJ incluyen el uso de protocolos de sincronización para el acceso a secciones críticas, aunque solo los dos últimos especifican que implementaciones necesitan soportar protocolos específicos, por ejemplo, el priority ceiling protocol[22]. (WIKIPEDIA, 2014)

1.5.2. Gestión de recursos de red

En relación a las redes de comunicaciones, ni RT-CORBA ni Ada DSA incluyen la posibilidad de asignar parámetros de planificación; por lo tanto, las implementaciones son responsables de proveer el soporte necesario. En cuanto DRTSJ, su última publicación solo establece que tanto en tiempo real como en redes de propósito general la gestión de recursos de red debe ser soportada. En el caso de DDS, la especificación solo considera redes basadas en políticas de planificación con prioridad fija y excluye cualquier tipo de redes predecibles utilizadas en la industria, por ejemplo, time-triggered networks[23]. Esto puede ser tratado mediante la modificación de la definición del parámetro *Transport_Priority* como propone (Pérez & Gutiérrez, On the schedulability of a data-centric real-time distribution middleware., 2012)

Aunque la mayoría de las normas analizadas se centran en las redes basadas en Ethernet, como por ejemplo, RT-CORBA con TCP/IP y DDS con UDP/IP, esta red de comunicación no es por sí apta para proporcionar tiempos de respuesta determinística.

Sin embargo, la evolución de la tecnología Ethernet en los últimos años, con la definición de nuevos estándares, como 802.1p (IEEE, 2006), el cual prioriza los diferentes

[22] *Priority ceiling protocol*, es un protocolo que se utiliza en tiempo real para gestionar el acceso a recursos compartidos, evitando la inversión de prioridad y la exclusión mutua.

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

mensajes, junto con su bajo costo, ha dado lugar a un creciente interés en la industria en el uso de este enfoque en el desarrollo futuro de los sistemas en tiempo real.

Cuando el middleware de distribución se lleva a cabo en los sistemas operativos y en los protocolos de red con la planificación basada en prioridades, es fácil de transmitir la prioridad en la que un servicio remoto debe ser ejecutado dentro de los mensajes enviados a través de la red, por ejemplo este esquema es utilizado por la política *Client_Propagated* en RT-CORBA. Sin embargo, esta solución no funciona si las políticas de planificación son complejas, tales como marco flexible de planificación basado en los contratos son utilizados (Aldea, y otros, 2006) (FRESCOR, 2006). Envío de los parámetros del contrato a través de la red es ineficiente porque estos parámetros son de gran tamaño. De la misma manera, el cambio dinámico de los parámetros del contrato en el nodo remoto es también ineficiente, por lo tanto, otros esquemas de configuración son requeridos para este tipo de sistema.

Otro factor importante a considerar es el tamaño de los mensajes de red, el cual debe ser delimitada y conocida antes del análisis de temporización. Este punto es particularmente crítico en el diseño de aplicaciones previsibles con DDS, mientras que a un mensaje DDSI puede comprender un número indefinido de mensajes, incluyendo no solo el metatráfico, sino también los datos de usuario.

Además este mecanismo es poco eficiente para minimizar el tiempo de respuesta promedio, no suele ser adecuada 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 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.

1.5.3. Cuadro Comparativo de las diferentes tecnologías

En la tabla 1-2 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, las cuales son requeridas en el peor caso del comportamiento temporal, permanecerá abierto a las implementaciones, como las mostradas en la tabla 1-2. En consecuencia, la elección de un middleware particular, determina no solo el rendimiento de las aplicaciones, sino también su previsibilidad, y por lo tanto la capacidad de cumplir con los deadlines. 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.

Finalmente, en la figura 1-7 se muestra la evolución de estos estándares de distribución de tiempo real.

Tabla 1-2. Capacidades de Tiempo-Real de los Estándares de Distribución *(Pérez & Gutiérrez, 2014).*

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	Implementación definida	Implementación definida	Priority inheritance	Implementación definida	Implementación definida
	Usuario definido			Priority Ceiling		

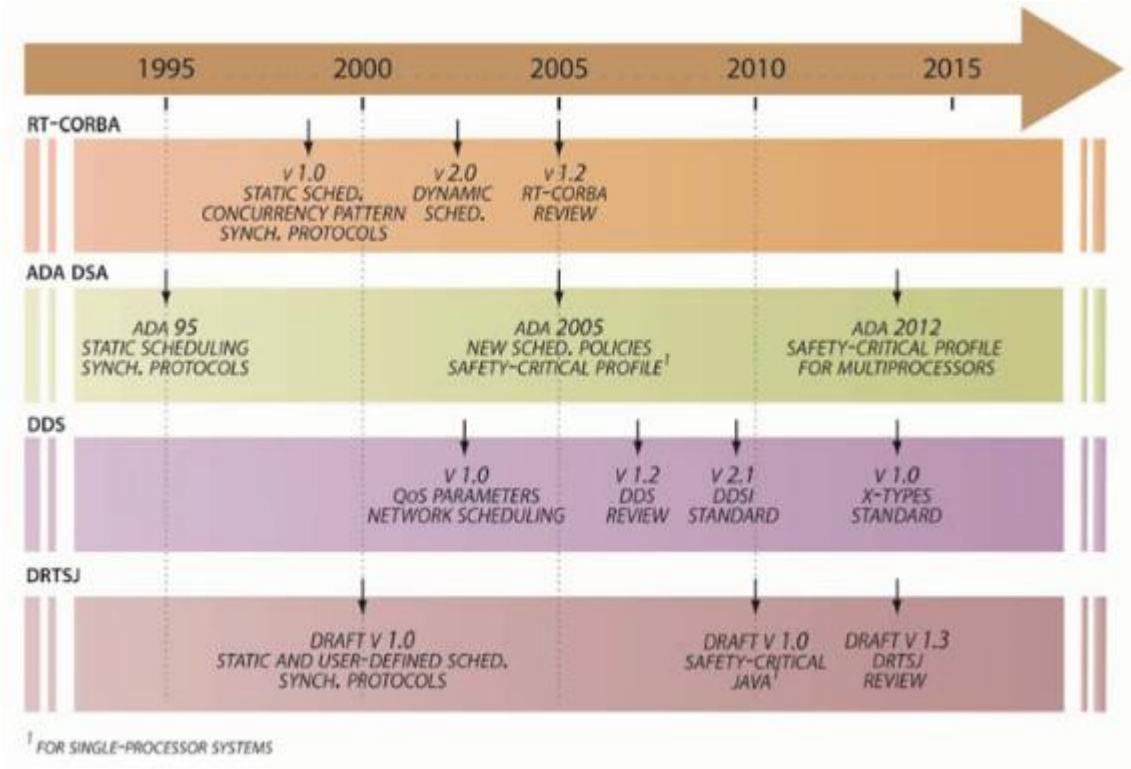


Figura 1-7 Línea de tiempo en los estándares de tiempo real

(Pérez & Gutiérrez, 2014).

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[24] , es decir haciendo su mejor esfuerzo.

Los flujos, representan capturas de valores de un *data-object* que puede ser interpretada 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[25] 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,

[24] IP, Internet Protocol.
[25] Naturalmente, se refiere a que la interfaz puede ser similar a una ya usada por variables locales lectura/ escritura.

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 siguiente Figura 1-8

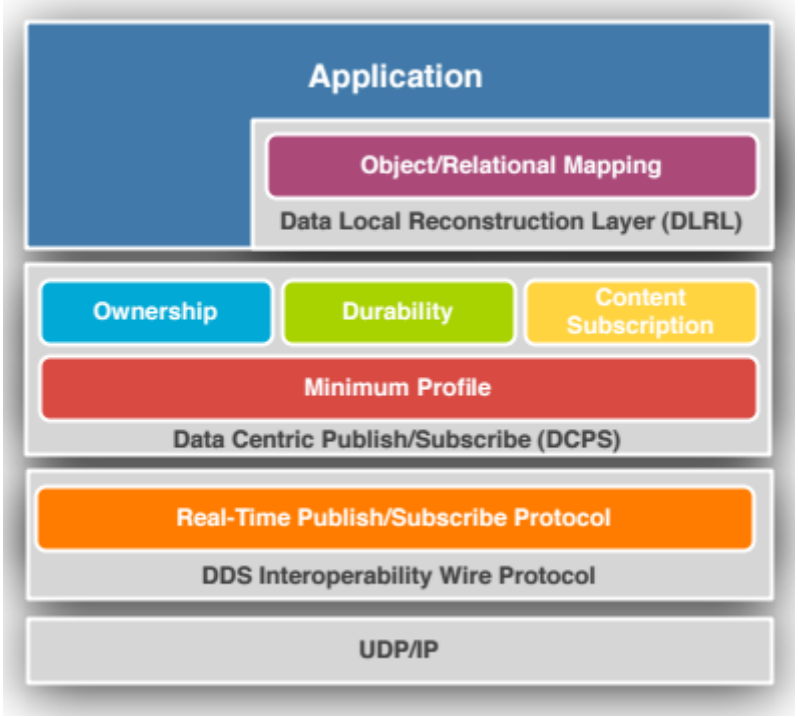


Figura 1-8. Arquitectura del Middleware DDS

(Corsaro)

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

- Un nivel bajo denominado *data-centric Publish-Subscribe o DCPS,* el cual está orientado a la entrega eficiente de información adecuada a los destinatarios adecuados.
- 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.

DCPS o Data-Centric Publish-Subscribe

La comunicación es llevada a cabo con la ayuda de las siguientes entidades: *DomainParticipant, DataWriter, DataReader, Publisher, Subscriber, y Topic*. Todas estas clases extienden la *DCPSEntity,* representando su capacidad de ser configurada a través de las políticas de QoS, ser notificado de eventos vía, y *soportar* condiciones que pueden ser esperadas por la aplicación. Cada especialización de una clase base *DCPSEntity* tiene un correspondiente *Listener* especializado y un conjunto de valores de *QoSPolicy* que son deseables.

El Publicador representa a los objetos responsables para la emisión de datos. Un Publicador debe publicar datos de diferente tipo.

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 comunicado 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 recibe los datos publicados y los hace disponibles al participante. Un Suscriptor debe recibir y despachar datos de diferentes tipos especificados. Para acceder a los datos recibidos, el participante debe utilizar un tipo *DataReader* asociado al suscriptor.

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

La entidad *Topic*, asocia un nombre que es único en el sistema, un tipo de dato, y QoS relacionado a su propia información. La definición de Tipo entrega suficiente información para

que el servicio manipule los datos, por ejemplo, serializa los datos dentro de un formato de red para ser transmitido. La definición de Tipo puede ser hecha por medio de un lenguaje textual, por ejemplo algo como "*float x; float y;*" o por medio de un "*plugin*" operacional, el cual provee los métodos necesarios.

DCPS puede también soportar suscripciones del tipo *content-based* 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.

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 son mejoradas 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 publicación.
- El módulo suscripción contiene las clases suscriptor, el *DataReader*, y el *SubscriberListener,* y generalmente todo lo que es necesario en el lado de suscripción.

- 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.

A continuación en la Figura 1-9 se muestra el modelo DCPS y sus relaciones.

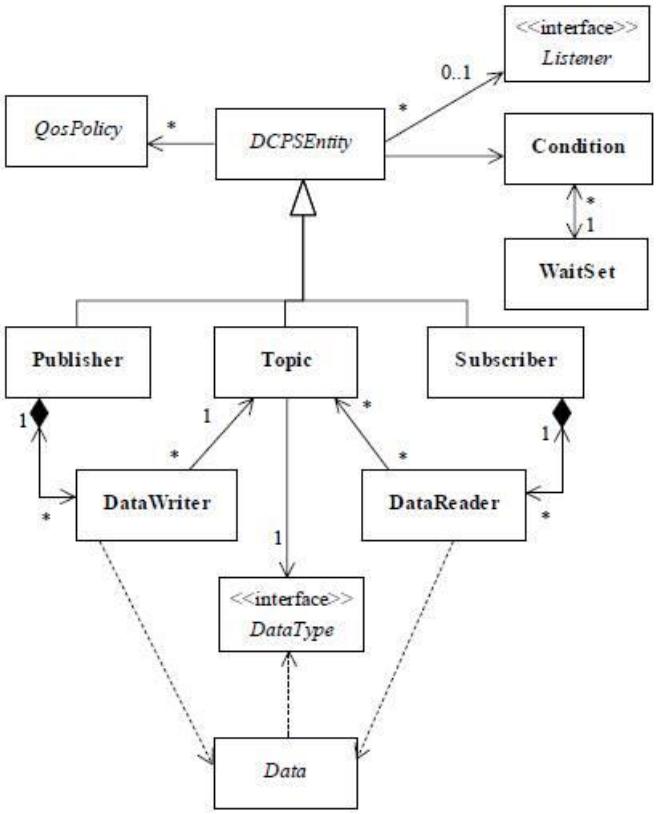


Figura 1-9. Modelo DCPS y sus relaciones

(Pardo-Castellote)

DLRL o Data Local Reconstruction Layer

La capa DLRL es una capa 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 solo utilizadas a través de DCPS y no ser accesibles del 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 solo 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 siguiente Figura 1-10 representa el modelo del DLRL.

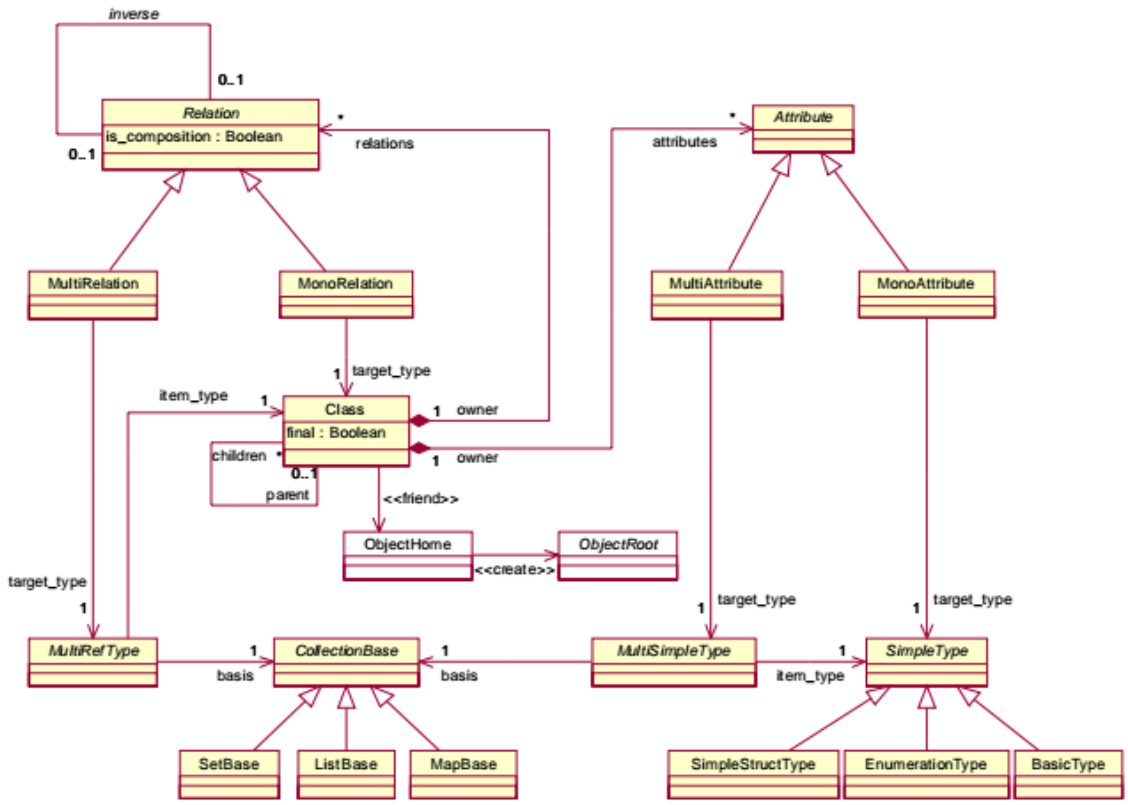


Figura 1-10. Modelo DLRL

(Pérez & Gutiérrez, 2014).

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.

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

Las principales características del protocolo RTPS son:

- Propiedades para el rendimiento y calidad de servicio, los que 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 para permitir la creación de redes sin puntos de fallos.
- Extensibilidad para permitir que el protocolo sea extendido y mejorado con nuevos servicios con compatibilidad hacia atrás e interoperabilidad.
- Conectividad plug-and-play para que las nuevas aplicaciones y servicios estén automáticamente descubiertos y las aplicaciones puedan unirse y dejar la red en cualquier momento sin necesidad de reconfiguración.
- Configurabilidad para permitir el balanceo de requerimientos para la confiabilidad y la puntualidad de cada entrega de datos.
- Modulabilidad para permitir que los dispositivos implementen un subconjunto del protocolo y que aun así participen en la red.
- Escalabilidad para sistemas que potencialmente escalen en redes extensas.
- Seguridad de tipo de datos para prevenir errores en la programación de aplicaciones que puedan comprometer las operaciones en los nodos remotos.

El protocolo RTPS esta descrito en términos de un modelo de plataforma independiente o PIM, o un conjunto de PCMS o Modelo específico de plataforma. El PIM RTPS contiene cuatro módulos los cuales son:

- *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 siguiente figura x se puede observar la interacción del protocolo RTPS con DDS.

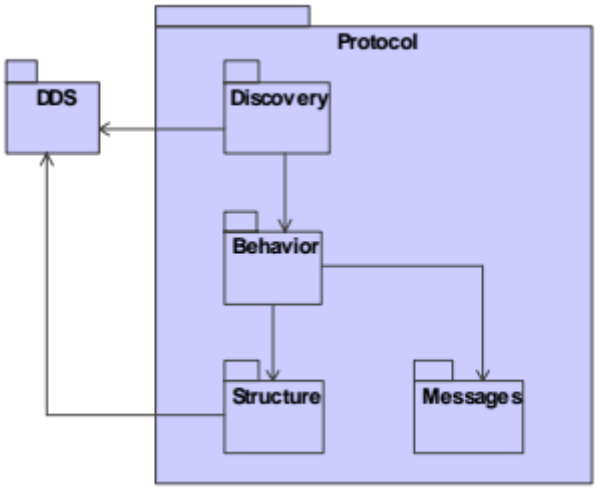


Figura 1-11. Módulos RTPS

(Pérez & Gutiérrez, 2014).

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 son tomados 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 Calidad de Servicio soportadas por el servicio DCPS se encuentran en definidas en el estándar.

Una política de QoS pueden ser establecidas 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 siguiente Tabla 1-3 se mostrará las Políticas de QoS.

Tabla 1-3. Políticas de QoS del DDS *(Corsaro)*.

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

En este modelo el lado del Suscriptor puede especificar una lista ordenada de las peticiones para una política particular de QoS en un orden decreciente. El lado del Publicador 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 en entre los dos objetos *DCPSEntity*, si la QoS solicitada y ofertada no pueden llegar a un acuerdo. En la siguiente Figura 1-12 se muestra el Modelo Suscriptor-Solicitado y el Publicador-Ofertado.

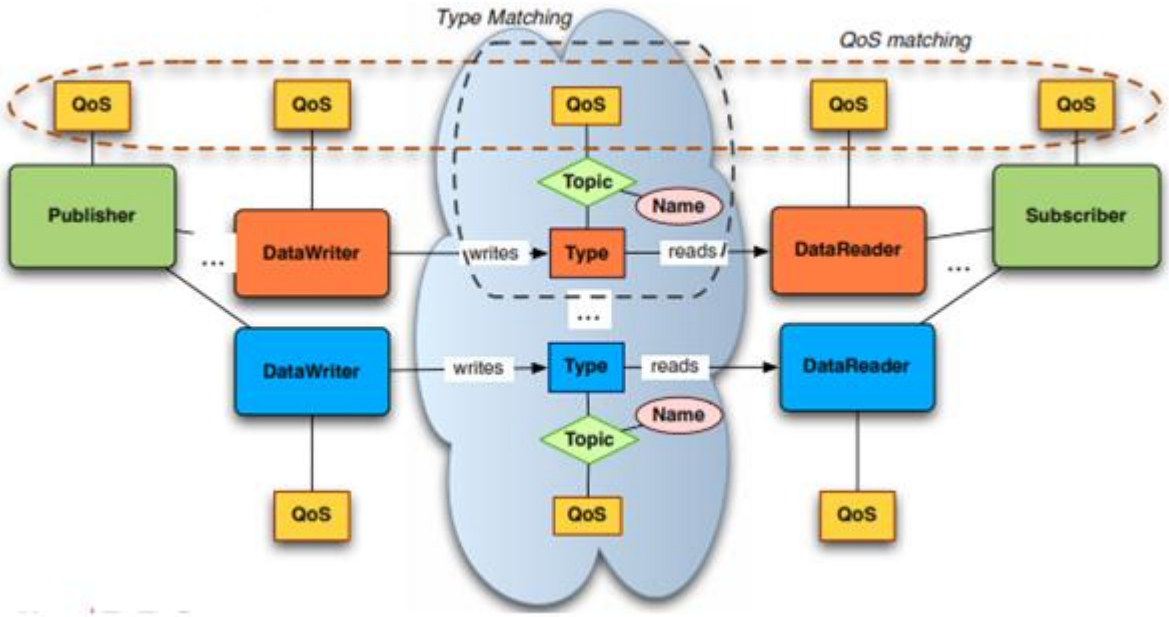


Figura 1-12. Modelo Suscriptor-Solicitado y Publicador-Ofertado *(Corsaro)*.

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.

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 siguiente Figura 1-13 muestra la interoperabilidad del API.

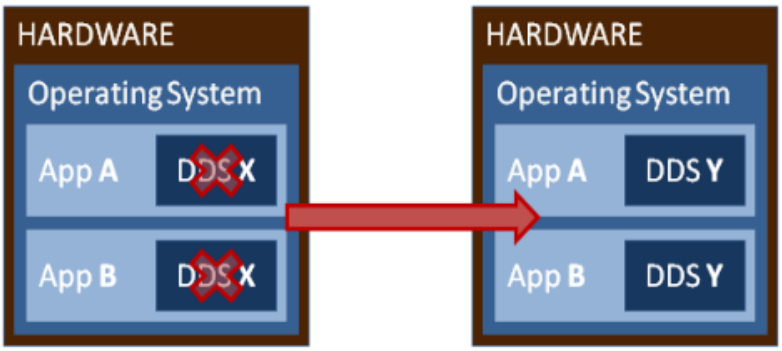


Figura 1-13. Interoperabilidad del API

(Twin Oaks Computing, Inc., 2011).

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 estándar 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 de 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 siguiente Figura 1-14 se podrá ver la interoperabilidad del protocolo de conexión.

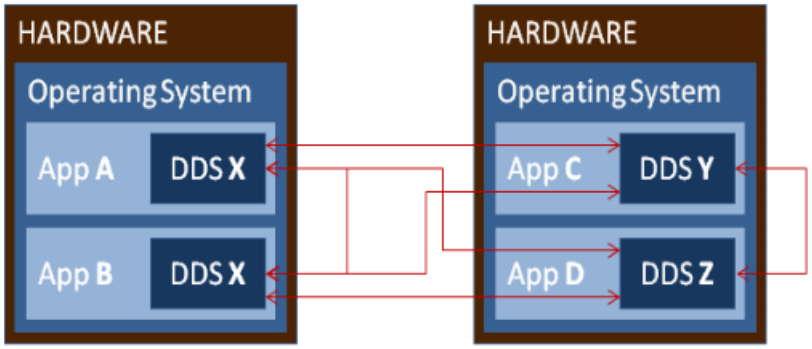


Figura 1-14. Interoperabilidad del Protocolo de Conexión

(Twin Oaks Computing, Inc., 2011).

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 posibles publicaciones futuras?; **historial** y **límites de recursos** es decir, ¿cuáles son los requerimientos de almacenamiento?; **filtrado** y **presentación** es decir, ¿qué información deben ser presentados 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 más de 22 políticas de QoS, y define un conjunto mínimo de políticas de QoS que den 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, 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, 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:

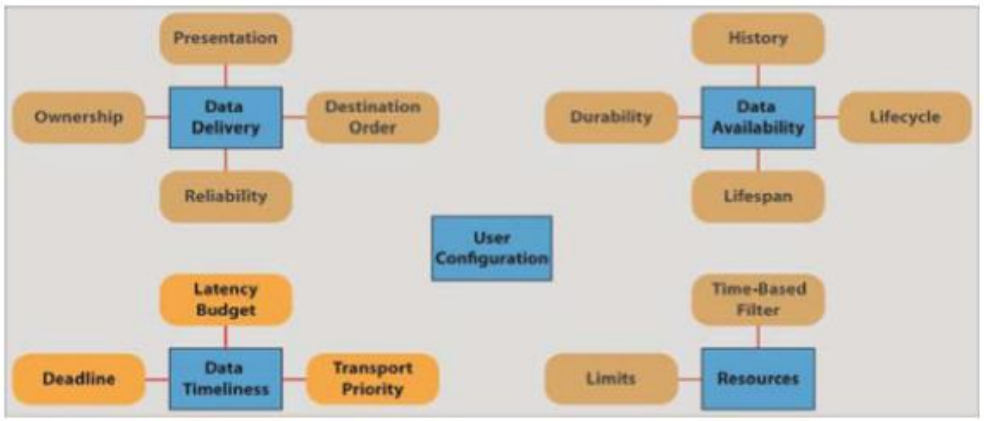


Figura 1-15. Parámetros de QoS definidos por DDS

(Pérez & Gutiérrez, 2014).

- *Data Availability o disponibilidad de los datos,* esto comprende parámetros para el control de políticas de consulta y almacenamiento de la información. Los parámetros que pertenecen a esta categoría son: Durability, Lifespan, History y Lifecycle.
- *Data Delivery o entrega de datos,* está especifica como la información debe ser transmitida y presentada a la aplicación. Los parámetros que pertenecen a esta categoría son: Presentation, Reliability, Partition, Destination_Order, y Ownership.
- *Data Timeliness o Puntualidad de la información,* esta controla la latencia en la distribución de los datos. Los parámetros que pertenecen a esta categoría son: Deadline, Latency_Budget, y Transport_Priority.
- *Maximum Resources o Máximos de Recursos,* esta limita la cantidad de recursos que pueden ser usados en el sistema a través de parámetros tales como: Resource_Limit o Time-Based_Filter.
- *User Configuration o Configuración de Usuario,* estos parámetros permiten que la información extra ser añadida a cada entidad en la capa aplicación.

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 compatible 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, como esta 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 solo 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 parámetro 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.

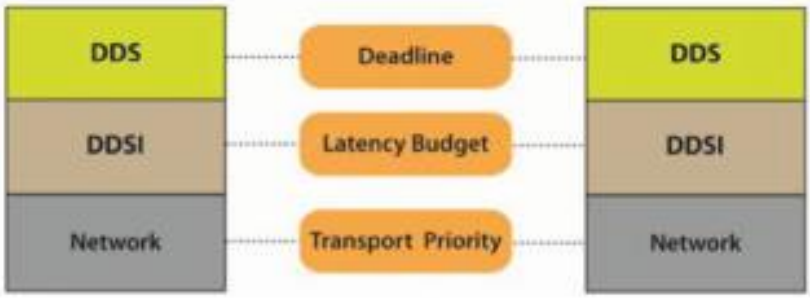


Figura 1-16. Control del tiempo en DDS

(Pérez & Gutiérrez, 2014).

El DDS define diferentes mecanismo 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*[26] (WIKIPEDIA, 2013), 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 la siguientes:

- *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

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

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.

- *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 es llevada 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 solo 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 solo 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 del 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 interoperabilidad, además 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. Esta 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*[27]. Este tipo de tráfico en la red, el cual es interno en el middleware, es llamado metatráfico y puede ser considerado en los análisis temporales.

[27] 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.

2. CAPÍTULO II

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. Finalmente, obtener un análisis detallado de los mismos y definir los requisitos necesarios para la 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.

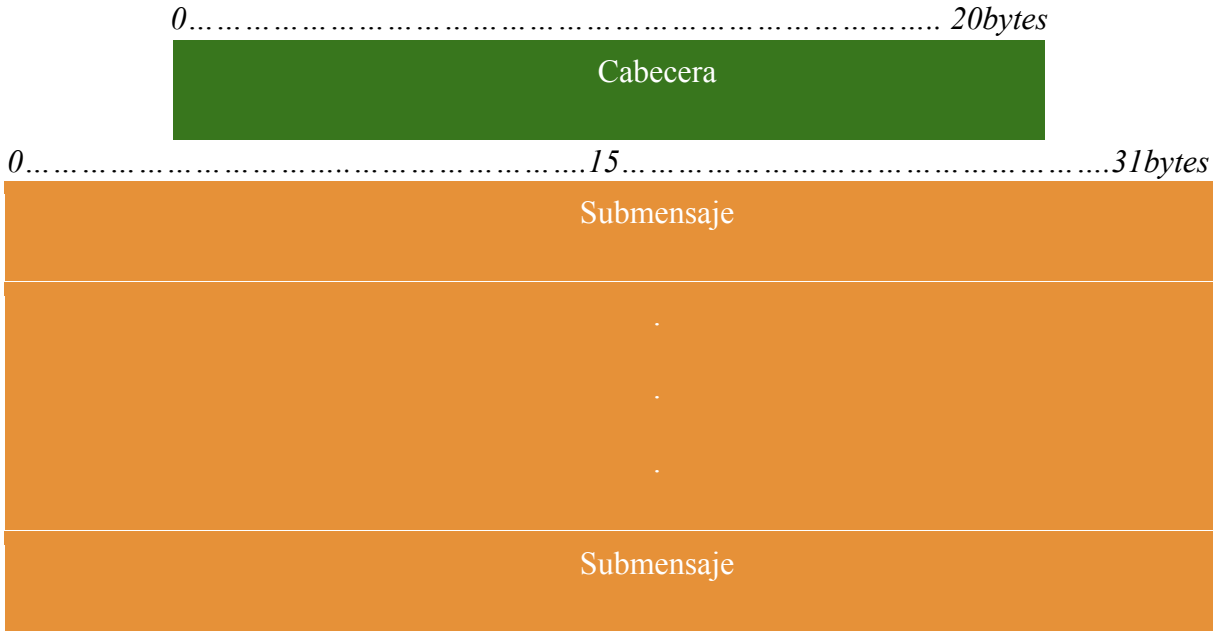


Figura 2-1. Estructura general mensaje RTPS

(OMG, 2014)

RTPS mensajes explícitamente no enviar la longitud, utiliza el transporte subyacente con el cual se envían mensajes, en el caso de la longitud es enviada a la carga UDP UDP/IP.

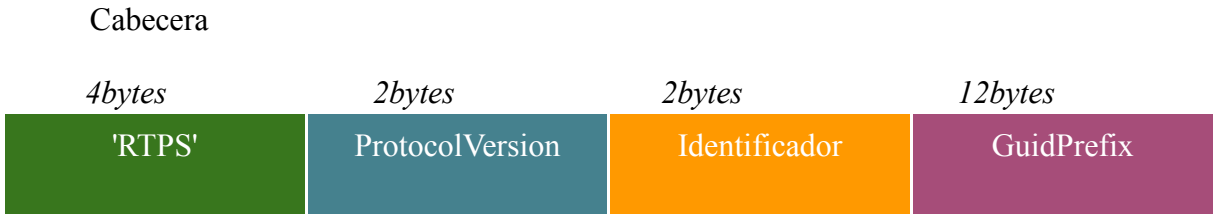


Figura 2-2. Cabecera del Mensaje RTPS

(OMG, 2014)

2.2.2. Estructura de los submensajes RTPS

En la siguiente Figura 2-3 se muestra la estructura del submensaje, la cual está compuesta por una cabecera submensaje y el contenido de submensaje,

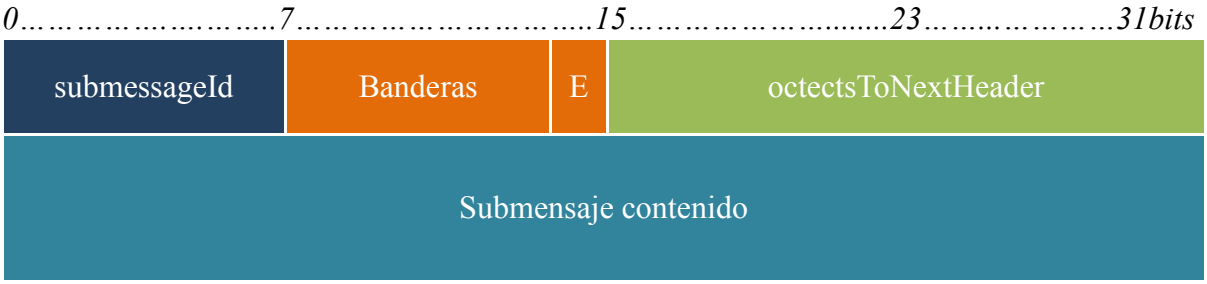


Figura 2-3. Estructura de los submensajes RTPS

(OMG, 2014)

2.2.2.1. Lista de submensajes

- Pad
- AckNack
- Heartbeat

- Gap
- InfoTimeStamp
- Infofuentes
- InfoReply
- InfoDestination
- InfoReplyIp4
- NackFrag
- HeartbeatFrag
- Datos
- DataFrag

2.2.3. AckNackSubmessage

En la Figura 2-4 se muestra la estructura del submensaje AckNack.

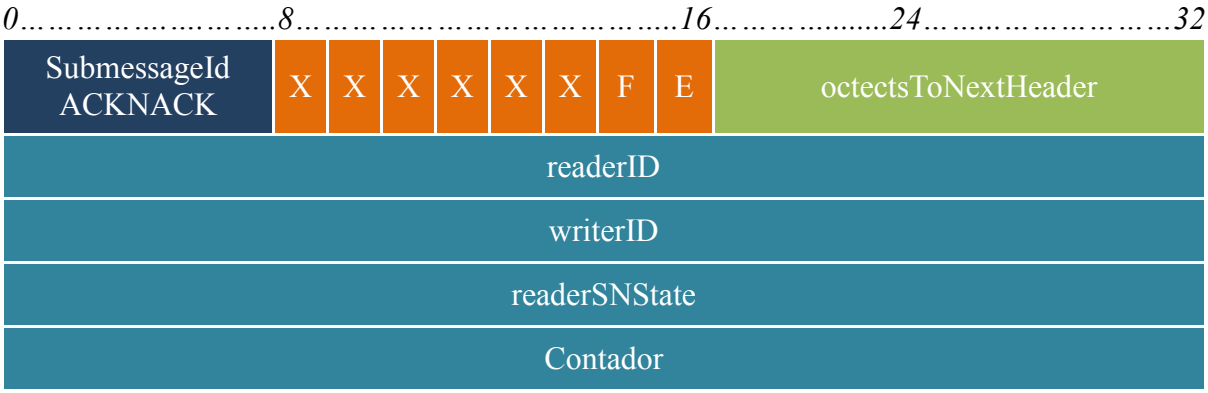


Figura 2-4. Estructura del submensaje AckNack

(OMG, 2014)

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

2.2.3.1. Banderas en el encabezado de submensaje

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

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

2.2.3.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad lector que acusa recibo de cierto número de secuencia o las solicitudes para recibir ciertos números de secuencia.

El **writerId**, identifica la entidad escritor que es el objetivo del mensaje AckNack. Es la entidad del escritor que se piden a re-enviar algunos números de secuencia o está siendo informado de la recepción de ciertos números de secuencia.

El **readerSNState**, se comunica el estado del lector al escritor. Todos los números de secuencia hasta el antes readerSNState.base se confirman como recibida por el lector. Los números de secuencia que aparecen en el conjunto indican falta números de secuencia en el lado del lector. Los que no aparecen en el conjunto son indeterminados (podría ser recibido o no).

El **Contador**, se incrementa cada vez que se envía un mensaje AckNack. Proporciona los medios para un escritor detectar los mensajes duplicados de AckNack que pueden derivarse de la presencia de vías de comunicación redundante.

2.2.3.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- readerSNState no es válida.

2.2.3.4. Cambio en el estado del receptor

Ninguno

2.2.3.5. Interpretación lógica

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

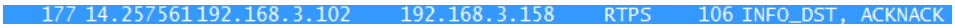
El escritor se identifica únicamente por su GUID. El GUID del escritor se obtiene utilizando el estado del receptor.

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:

- Reconoce el submensaje toda la secuencia de números hasta e incluyendo el que sólo el menor número de secuencia en el SequenceNumberSet.
- El submensaje negativamente-reconoce (peticiones) la secuencia de números que aparecen explícitamente en el conjunto de.

Ejemplo



```
submessageId: ACKNACK (0x06)
  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.. = reserved bit: Not set
    ......0. = Final flag: Not set
    .......1 = Endianness bit: Set
  octetsToNextHeader: 24
  readerEntityId: ENTITYID_BUILTIN_PUBLICATIONS_READER (0x000003c7)
    readerEntityKey: 0x000003
    readerEntityKind: Built-in reader (with key) (0xc7)
  writerEntityId: ENTITYID_BUILTIN_PUBLICATIONS_WRITER (0x000003c2)
    writerEntityKey: 0x000003
    writerEntityKind: Built-in writer (with key) (0xc2)
  readerSNState
    bitmapBase: 0
    numBits: 0
  Counter: 1
```

```
0000  00 27 13 fc cb a2 64 5a  04 2b d8 62 08 00 45 00   .'....dZ .+.b..E.
0010  00 5c 7a 97 00 00 80 11  37 a5 c0 a8 03 66 c0 a8   .\z..... 7....f..
0020  03 9e c6 4e 1c f2 00 48  1d bc 52 54 50 53 02 01   ...N...H ..RTPS..
0030  01 01 c0 a8 03 66 00 00  03 6c 00 00 00 01 0e 01   .....f.. .l......
0040  0c 00 c0 a8 03 9e 00 00  04 14 00 00 00 01 06 01   ........ ........
0050  18 00 00 00 03 c7 00 00  03 c2 00 00 00 00 00 00   ........ ........
0060  00 00 00 00 00 00 01 00  00 00                     ........ ..
```

Una explicación del uso del AckNack ha sido descrita dentro del ejemplo 2 en la sección de ejemplos de RTPS

2.2.4. DataSubmessage

En la Figura 2-5 se muestra la estructura del submensaje Data.

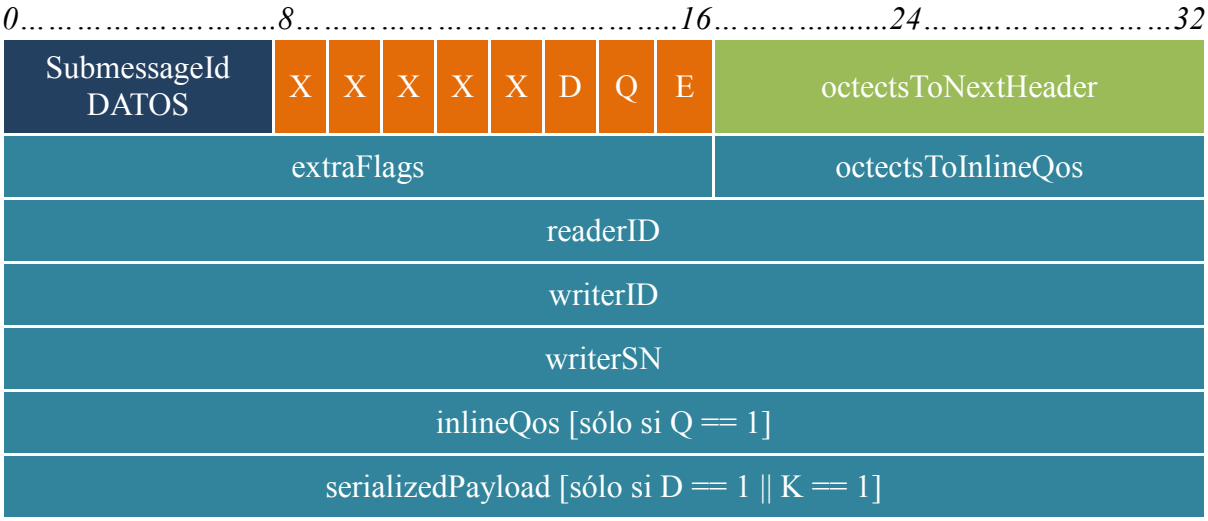


Figura 2-5. Estructura del submensaje Data

(OMG, 2014)

El submensaje notifica al lector RTPS de un cambio a un objeto de datos pertenecientes al escritor RTPS. Los posibles cambios incluyen ambos cambios en valor, así como los cambios en el ciclo de vida del objeto de datos.

2.2.4.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparecen en las banderas de cabecera 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 deben utilizarse para interpretar el mensaje. Se representa con el literal 'p' Q = 1 significa que el submensaje **datos** contiene el inlineQos SubmessageElement.

La **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 serializedPayload SubmessageElement.
- D = 1 y K = 0 significan que el serializedPayload SubmessageElement contiene los datos serializados.
- D = 0 y K = 1 significan que el serializedPayload SubmessageElement contiene la clave serializada.

- D = 1 y K = 1 son una combinación no válida en esta versión del Protocolo de.

2.2.4.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad RTPS **lector** que está siendo informada del cambio con el objeto de datos.

El **writerId** , identifica la entidad RTPS **escritor** que hizo el cambio con el objeto de datos.

El **writerSN**, identifica el cambio y el orden relativo de todos los cambios realizados por el RTPS **escritor** identificados por el writerGuid. Cada cambio obtiene un número de secuencia consecutiva. Cada RTPS **escritor** mantiene es número de secuencia propia.

El **inlineQos**, presente solamente si la InlineQosFlag está situado en la cabecera. Contiene QoS que puedan afectar la interpretación del mensaje.

El **serializedPayload**, presente solamente si el DataFlag o el KeyFlag se encuentra en la cabecera.

- Si el DataFlag está establecido, entonces contiene la encapsulación del nuevo valor del objeto de datos después del cambio.
- Si el KeyFlag está establecido, entonces contiene la encapsulación de la clave del objeto de datos el mensaje se refiere a.

El campo **extraFlags** , ofrece espacio para una 16 bits adicionales de banderas más allá de los 8 bits proporcionados al igual que en la cabecera de submensaje. Estos bits adicionales

apoyará la evolución del protocolo sin comprometer la compatibilidad hacia atrás. Esta versión del protocolo debe establecer todos los bits en el extraFlags a cero.

El **octetsToInlineQos,** la representación de este campo es un CDR unsigned corta (ushort).

El campo octetsToInlineQos contiene el número de octetos a partir del primer octeto inmediatamente después de este campo hasta el primer octeto de la inlineQos SubmessageElement. Si el inlineQos SubmessageElement no está presente (es decir, el InlineQosFlag no está establecida), entonces octetsToInlineQos contiene el desplazamiento al campo siguiente 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 no esperar o entender y seguir procesar el inlineQos SubmessageElement (o el primer elemento submensaje que sigue inlineQos si el inlineQos no está presente). Esta regla es necesaria para que el receptor será capaz de interactuar con los remitentes que utilizan las versiones futuras del protocolo que puede incluir cabeceras submensaje adicional antes de la inlineQos.

2.2.4.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- writerSN.value no es estrictamente positivo (1, 2,...) o es SEQUENCENUMBER_UNKNOWN.
- inlineQos no es válida.

2.2.4.4. Cambio en el estado del receptor

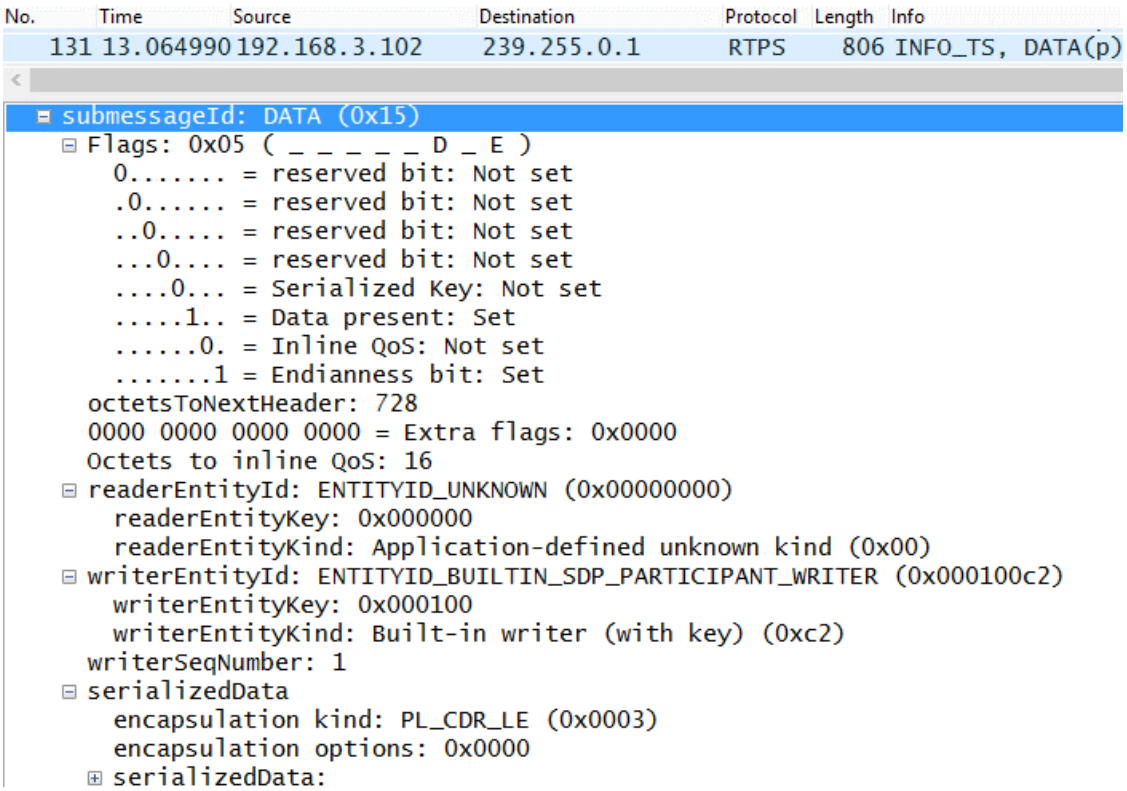
Ninguno

2.2.4.5. Interpretación lógica

El **escritor** envía el submensaje **datos** al **lector** para comunicar los cambios en los objetos de datos dentro de la escritora. Los cambios incluyen dos cambios en valor, así como los cambios en el ciclo de vida del objeto de datos. Éstos se comunican por la presencia de la 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 se establece la DataFlag y el KeyFlag no está establecida, el elemento serializedPayload se interpreta como el valor del objeto dataobject.
- Si se establece la KeyFlag y el DataFlag no está establecida, el elemento serializedPayload se interpreta como el valor de la clave que identifica la instancia del objeto de datos registrada.
- Si el InlineQosFlag está establecida, el elemento inlineQos contiene los valores de QoS que reemplazan a los del escritor RTPS y debe ser utilizada para procesar la actualización de.

Ejemplo



Una explicación del uso del Data ha sido descrito dentro del ejemplo 2 en la sección de ejemplos de RTPS

2.2.5. DataFragSubmessage

En la Figura 2-6 se muestra la estructura del submensaje DataFrag.

extraFlags	octectsToInlineQos
readerID	
writerID	
writerSN	
fragmentStartingNum	
fragmentsInSubmessage	fragmentSize
sampleSize	
inlineQos [sólo si Q == 1]	
serializedPayload	

Figura 2-6. Estructura del submensaje DataFrag

(OMG, 2014)

El submensaje DataFrag extiende el submensaje datos activando el serializedData a ser fragmentado y enviado como múltiples DataFrag submensajes. Los fragmentos contenidos en los submensajes DataFrag re luego son ensamblados por el lector RTPS.

Definir un separado submensaje DataFrag además el submensaje datos, ofrece las siguientes ventajas:

- Mantiene las variaciones en el contenido y estructura de cada submensaje al mínimo. Esto permite más eficientes implementaciones del protocolo como el análisis de paquetes de red se simplifica.
- Evita tener que agregar información de fragmentación como parámetros de QoS en línea en el submensaje datos. Esto puede no sólo más lento desempeño, también dificulta en el cable de depuración más, como ya no es obvio si es fragmentados o no y que el mensaje contiene qué perdidos.

2.2.5.1. Banderas en el encabezado de submensaje

La **InlineQosFlag**, indica que la entidad lector que está siendo informada del cambio con el objeto de datos, cuando se establece en 1 significa que el submensaje DataFrag contiene el inlineQos SubmessageElement.

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

El **KeyFlag**, cuando se establece en 0, significa que el serializedPayload SubmessageElement contiene los datos serializados, cuando se establece en 1 significa que el serializedPayload SubmessageElement contiene la clave serializada.

2.2.5.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad RTPS lector que está siendo informada del cambio con el objeto de datos.

.El **writerId**, identifica la entidad RTPS escritor que hizo el cambio para el objeto dataobject.

El **writerSN**, se identifica el cambio y el orden relativo de todos los cambios realizados por el escritor RTPS identificados por el writerGuid. Cada cambio obtiene un número de secuencia consecutiva. Cada escritor RTPS mantiene es número de secuencia propia.

La **fragmentStartingNum**, indica el fragmento inicial de la serie de fragmentos de serializedData.

Fragmento de numeración comienza con el número 1.

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

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

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

El **inlineQos**, presente solamente si la InlineQosFlag está situado en la cabecera.

Contiene QoS que puedan afectar la interpretación del mensaje.

El **serializedPayload**, presente sólo si DataFlag se establece en la cabecera.

Encapsulación de una serie consecutiva de fragmentos, a partir de las fragmentStartingNum para un total de fragmentsInSubmessage.

Representa parte del valor del objeto de datos-nuevo después del cambio. Actualmente sólo si el DataFlag o el KeyFlag se establece en la cabecera.

- Si el DataFlag está establecido, entonces contiene un conjunto de fragmentos de la encapsulación del nuevo valor del objeto dataobject después del cambio consecutivo.
- Si el KeyFlag está establecido, entonces contiene un conjunto de fragmentos de la encapsulación de la clave del objeto de datos el mensaje se refiere a consecutivos.

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

2.2.5.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- writerSN.value no es estrictamente positivo (1, 2,...) o es SEQUENCENUMBER_UNKNOWN.
- fragmentStartingNum.value no es estrictamente positivo (1, 2,...) o excede el número total de fragmentos de.
- fragmentSize excede dataSize.

- El tamaño del serializedData superior fragmentsInSubmessage * fragmentSize.
- inlineQos no es válida.

2.2.5.4. Cambio en el estado del receptor

Ninguno

2.2.5.5. Interpretación lógica

El submensaje DataFrag extiende el submensaje datos activando el serializedData a ser fragmentado y enviado como múltiples DataFrag submensajes. Una vez que el serializedData es volver a montar el lector RTPS, la interpretación de los submensajes DataFrag es idéntica a la del submensaje datos.

A continuación se describe cómo volver a montar usando la información en el submensaje DataFrag serializedData

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

(fragmentStartingNum - 1) * fragmentSize

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

(dataSize / fragmentSize) + ((dataSize % fragmentSize)? 1: 0)

Tenga en cuenta que cada submensaje DataFrag puede contener múltiples fragmentos. Un escritor RTPS seleccionará fragmentSize basado en el tamaño más pequeño de mensaje soportado a través de todos los transportes subyacentes. Si algunos lectores RTPS puede ser alcanzados a través de un transporte que soporta mensajes más grandes, el escritor RTPS puede embalar los fragmentos múltiples en un solo DataFrag submensaje o incluso puede enviar un submensaje regular de datos si la fragmentación ya no es necesaria.

Ejemplo

```
   15 3.3021170 127.0.0.1        127.0.0.1        RTPS    1150 DATA_FRAG
   16 3.3025280 127.0.0.1        127.0.0.1        RTPS    1122 DATA_FRAG
<
  ⊞ guidPrefix
  ⊞ Default port mapping: domainId=186, participantIdx=114, nature=UNICAST_MET
  ⊟ submessageId: DATA_FRAG (0x16)
    ⊟ Flags: 0x03 ( _ _ _ _ _ _ Q 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
        ......1. = Inline QoS: 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): 0x000
      writerSeqNumber: 6
      fragmentStartingNum: 1
      fragmentsInSubmessage: 1
      fragmentSize: 1024
      sampleSize: 3072
    ⊞ inlineQos:
    ⊞ serializedData
```

```
⊟ 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
```

2.2.6. GapSubmessage

En la Figura 2-7 se muestra la estructura del submensaje Gap.

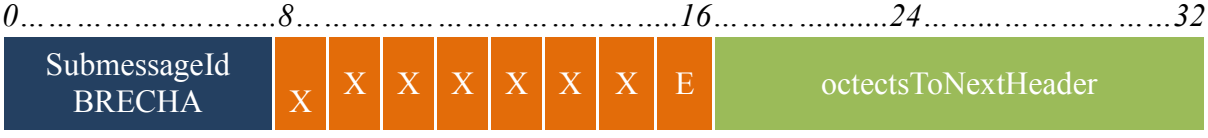


Figura 2-7. Estructura del submensaje Gap

(OMG, 2014)

Este submensaje es enviado de un escritor RTPS a un lector RTPS e indica al lector RTPS 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**, aparecen en las banderas de cabecera submensaje e indica el orden de bits.

2.2.6.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad lector que está siendo informado de la irrelevancia de un conjunto de números de secuencia.

El **writerId**, identifica la entidad escritor al que se aplica el rango de números de secuencia.

El **gapStart**, identifica el primer número de secuencia en el intervalo de números de secuencia irrelevante.

El **gapList**, sirve para dos:

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

2.2.6.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- gapStart es cero o negativo.
- gapList no es válida.

2.2.6.4. Cambio en el estado del receptor

Ninguno

2.2.6.5. Interpretación lógica

El escritor RTPS envía el mensaje de boquete al lector RTPS para comunicar que ciertos números de secuencia ya no son relevantes. Esto es causado típicamente por lado del escritor filtrado de la muestra (temas de filtrado de contenido, basado en el tiempo de filtrado). En este escenario, nuevos datos-valores podrán sustituir a los viejos valores de los objetos de datos que fueron representados por los números de secuencia que aparecen como irrelevantes en la brecha.

Los números de secuencia irrelevante comunicados por el mensaje de 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.

Ejemplo

```
No.  Time  Source  Destination  Protocol  Length  Info
 713 209.72610 192.168.3.158   192.168.3.102   RTPS    110 INFO_DST, GAP
submessageId: GAP (0x08)
  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.. = reserved bit: Not set
     ......0. = reserved bit: Not set
     .......1 = Endianness bit: Set
   octetsToNextHeader: 28
  readerEntityId: 0x80000007 (Application-defined reader (with key): 0x800000)
     readerEntityKey: 0x800000
     readerEntityKind: Application-defined reader (with key) (0x07)
  writerEntityId: 0x80000002 (Application-defined writer (with key): 0x800000)
     writerEntityKey: 0x800000
     writerEntityKind: Application-defined writer (with key) (0x02)
   gapStart: 1
  gapList
     bitmapBase: 96
     numBits: 0
```

2.2.7. HeartbeatSubmessage

En la Figura 2-8 se muestra la estructura del submensaje Heartbeat.

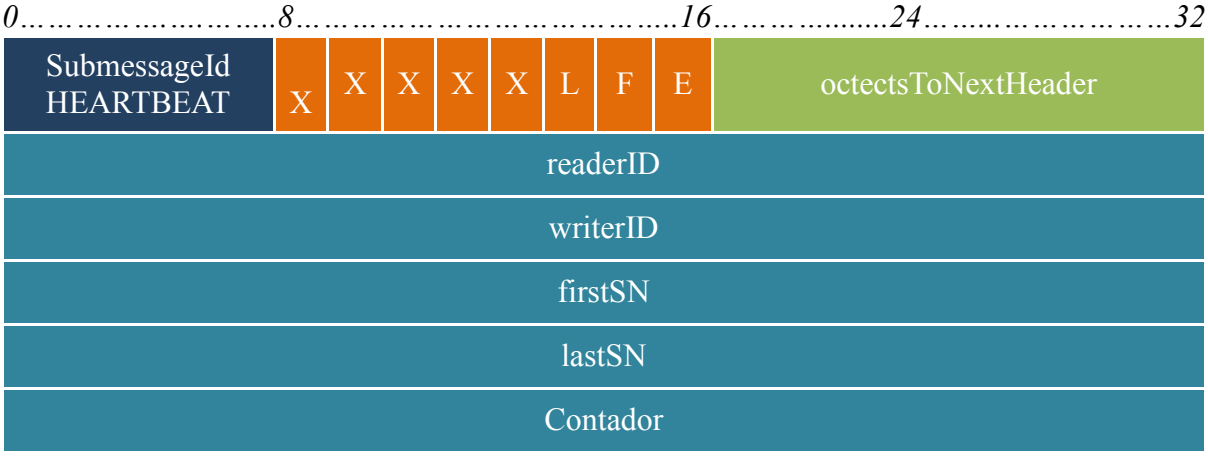


Figura 2-8. Estructura del submensaje Heartbeat

(OMG, 2014)

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 de submensaje

La **FinalFlag**, indica que si el lector es necesaria para responder a los latidos del corazón o si es sólo un latido asesor.

El FinalFlag se representa con el literal 'F'. F = 1 significa que el escritor no requiere una respuesta desde el lector. F = 0 significa que el lector debe responder al mensaje de latidos del corazón.

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

La **LivelinessFlag**, indica que el escritor de datos DDS asociados con el escritor RTPS de la messaage ha afirmado manualmente su vivacidad.

El LivelinessFlag se representa con el literal 'L'. L = 1 significa que el DataReader DDS asociados con el lector de RTPS debe actualizar la viveza 'manual' de la DataWriter DDS asociados con el escritor RTPS del mensaje.

2.2.7.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad lector que está siendo informado de la disponibilidad de un conjunto de números de secuencia. Puede establecerse en ENTITYID_UNKNOWN para indicar que todos los lectores para el escritor que envió el mensaje.

El **writerId**, identifica la entidad escritor al que se aplica el rango de números de secuencia.

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

El **Contador**, un contador que se incrementa cada vez que se envía un mensaje de latido nuevo. Proporciona los medios para un lector detectar los mensajes duplicados de latidos cardíacos que pueden derivarse de la presencia de vías de comunicación redundante.

.

2.2.7.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- firstSN valor es cero o negativo.
- lastSN valor es zeo o negativas.
- lastSN valor < firstSN valor

2.2.7.4. Cambio en el estado del receptor

Ninguno

2.2.7.5. Interpretación lógica

El mensaje del Heartbeat tiene dos propósitos:

- Informa al lector de los números de secuencia que están disponibles en HistoryCache del escritor para que el lector puede solicitar (mediante un AckNack) más que ha faltado a.
- Pide al lector a enviar un acuse de recibo para los cambios CacheChange que han sido introducidos en HistoryCache del lector tal que el escritor conoce el estado del lector.

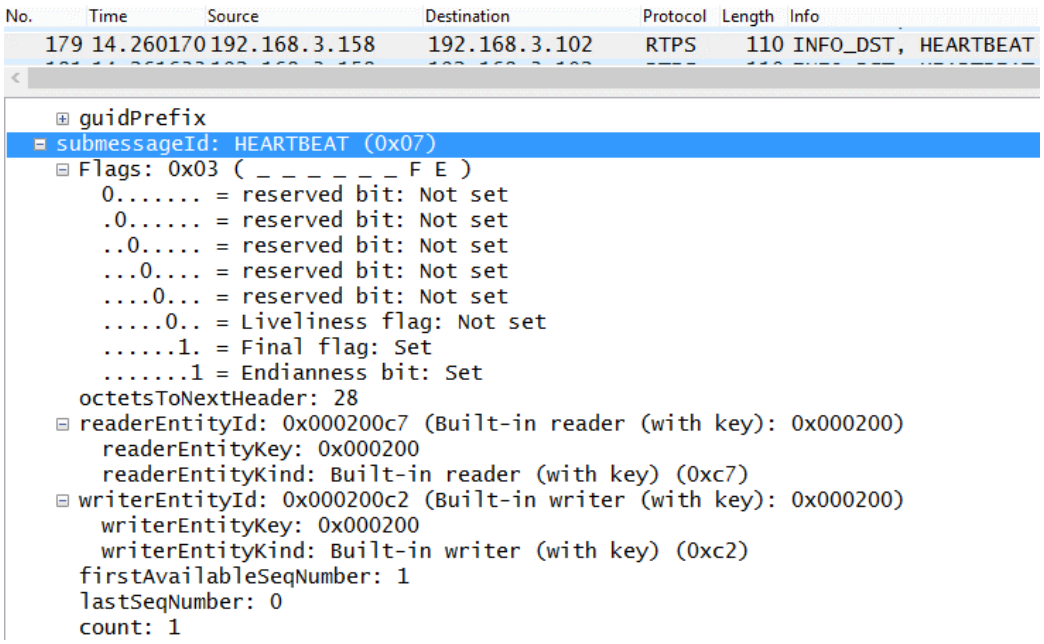
Todos los mensajes de latidos del corazón el primer propósito. Es decir, el lector encontrará siempre el estado de HistoryCache del escritor y puede solicitar lo que ha perdido. Normalmente, el lector RTPS sólo enviaría un mensaje AckNack si le falta un CacheChange.

El escritor utiliza la FinalFlag para solicitar al lector a enviar un acuse de recibo para los números de secuencia que ha recibido. Si el Heartbeat tiene el FinalFlag activo, el lector no

es necesario enviar un mensaje AckNack. Sin embargo, si el FinalFlag no está establecido, entonces el lector debe enviar ha recibido un mensaje de AckNack que indica que CacheChange cambia, aunque el AckNack indica que ha recibido todos los cambios CacheChange en HistoryCache del escritor.

El escritor establece la LivelinessFlag para indicar que la DataWriter DDS asociados con el escritor RTPS del mensaje manualmente ha afirmado su vitalidad mediante la adecuada operación de DDS (consulte la especificación DDS). El lector RTPS por lo tanto debe renovar el arrendamiento manual vivacidad de la correspondiente DataWriter DDS remoto.

Ejemplo



Una explicación del uso del Heartbeat ha sido descrito dentro del ejemplo 2 y 3 en la sección de ejemplos de RTPS

2.2.8. HeartBeatFragSubmessage

En la Figura 2-9 se muestra la estructura del submensaje HeartBeatFrag.

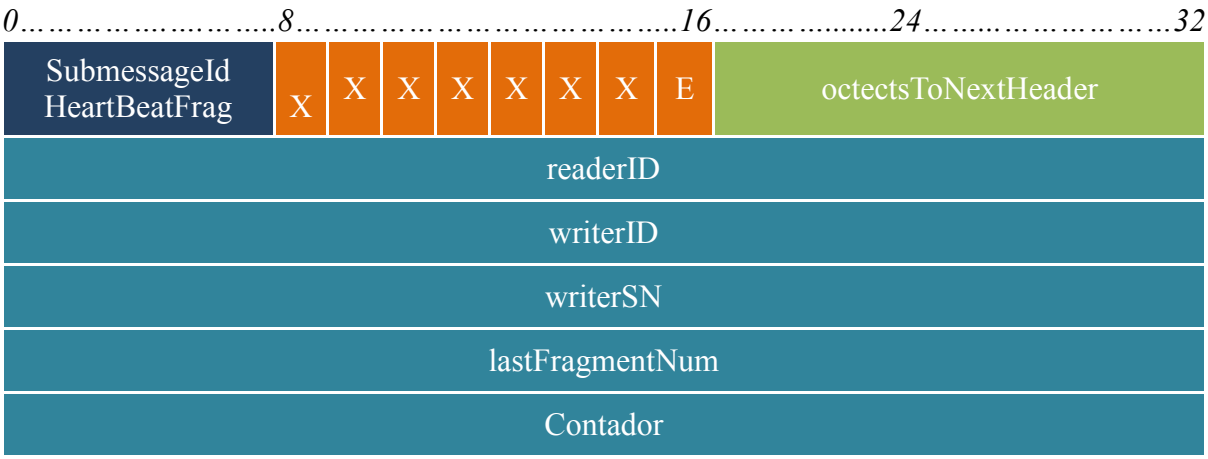


Figura 2-9. Estructura del submensaje HeartBeatFrag

(OMG, 2014)

Fragmentación de datos y hasta que todos los fragmentos están disponibles, el submensaje **HeartbeatFrag** se envía desde una RTPS **escritor** a un RTPS **lector** para comunicar que los fragmentos del escritor tiene 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 **latidos del corazón** .

2.2.8.1. Banderas en el encabezado de submensaje

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

2.2.8.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad lector que está siendo informado de la disponibilidad de fragmentos. Puede establecerse en **ENTITYID_UNKNOWN** para indicar que todos los lectores para el escritor que envió el mensaje.

El **writerId**, identifica la entidad **escritor** que envió el submensaje.

El **writerSN**, identifica el número de secuencia del cambio datos para que los fragmentos están disponibles.

El **lastFragmentNum**, todos los fragmentos hasta e incluyendo este último fragmento (más alto) están disponibles en el escritor para el cambio identificado por writerSN.

El **Contador**, se incrementa cada vez que se envía un mensaje HeartbeatFrag. Proporciona los medios para un lector detectar los mensajes duplicados de HeartbeatFrag que pueden derivarse de la presencia de vías de comunicación redundante.

2.2.8.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- writerSN.value es cero o negativo.
- lastFragmentNum.value es cero o negativo.

2.2.8.4. Cambio en el estado del receptor

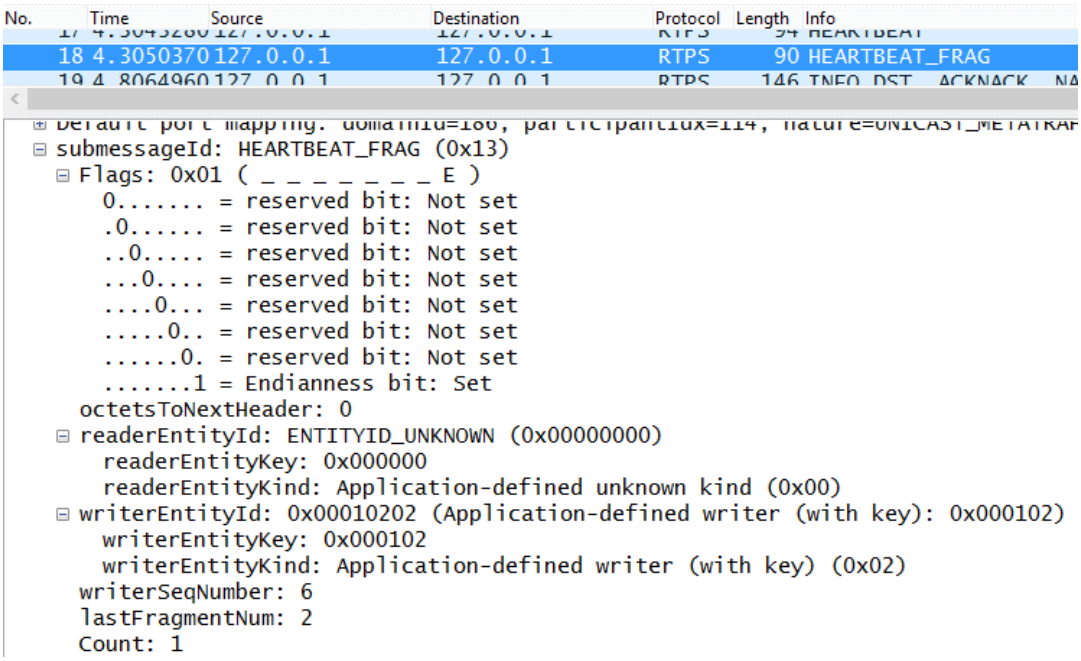
Ninguno

2.2.8.5. Interpretación lógica

El mensaje de HeartbeatFrag sirve el mismo propósito como un mensaje de ritmo cardíaco 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.

Ejemplo



2.2.9. InfoDestinationSubmessage

En la Figura 2-10 se muestra la estructura del submensaje InfoDestination.

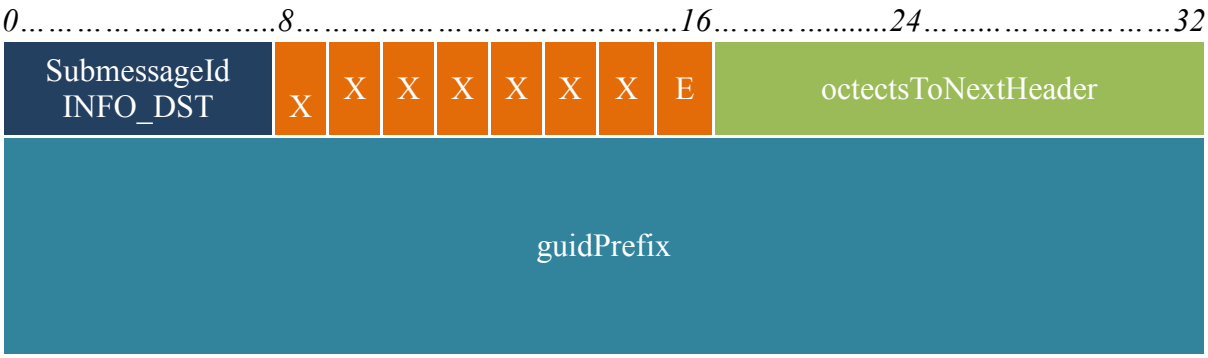


Figura 2-10. Estructura del submensaje InfoDestination

(OMG, 2014)

Este mensaje se envía desde un escritor RTPS a un lector RTPS para modificar el GuidPrefix utilizado para interpretar el lector entityIds que aparecen en los submensajes que le siguen.

2.2.9.1. Banderas en el encabezado de submensaje

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

2.2.9.2. Otros elementos en el encabezado de submensaje

La **guidPrefix**, proporciona la GuidPrefix que debe utilizarse para reconstruir los GUID de todas las entidades de RTPS lector cuyo EntityIds aparece en los submensajes que siguen.

2.2.9.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.

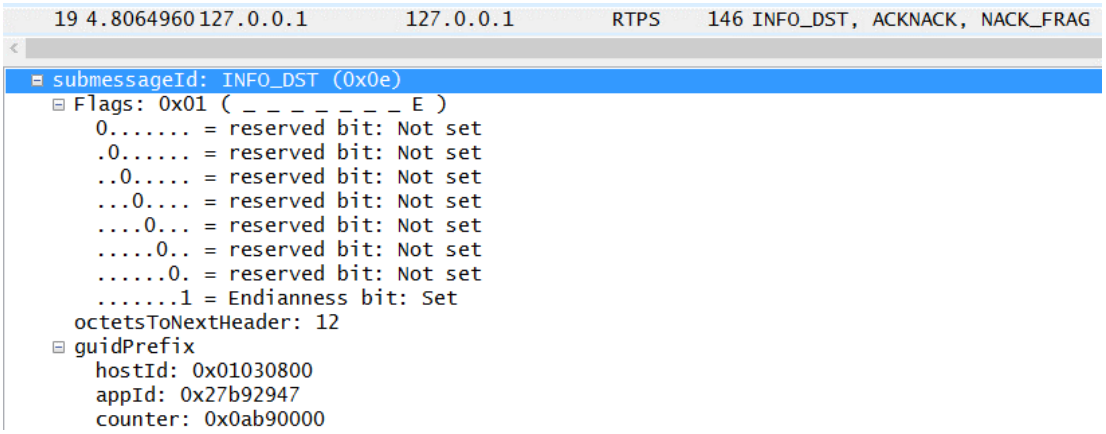
2.2.9.4. Cambio en el estado del receptor

Si (InfoDestination.guidPrefix! = GUIDPREFIX_UNKNOWN) {Receiver.destGuidPrefix = InfoDestination.guidPrefix} más {Receiver.destGuidPrefix = < GuidPrefix_t del participante que reciba el mensaje >}

2.2.9.5. Interpretación lógica

Ninguno

Ejemplo



Una explicación del uso del InfoDestination ha sido descrito dentro del ejemplo 2 en la sección de ejemplos de RTPS

2.2.10. InfoReplySubmessage

En la Figura 2-11 se muestra la estructura del submensaje InfoReply.

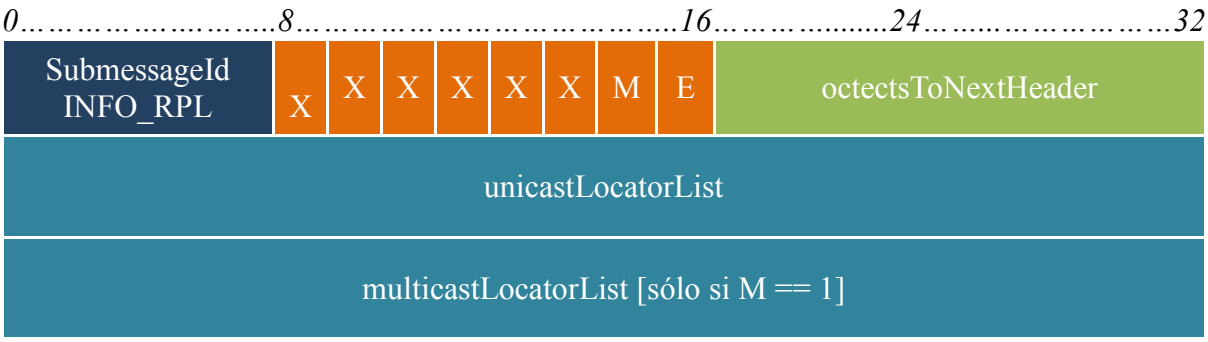


Figura 2-11. Estructura del submensaje InfoReply

(OMG, 2014)

Este mensaje se envía desde un lector de RTPS a un escritor de 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 de submensaje

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

La **MulticastFlag**, indica si el submensaje también contiene una dirección de multidifusión.

El MulticastFlag está representado con el literal estoy '. M = 1 significa que la InfoReply también incluye un multicastLocatorList.

2.2.10.2. Otros elementos en el encabezado de submensaje

La **unicastLocatorList**, indica que el escritor debe utilizar para llegar a los lectores al responder a los submensajes que siguen se dirige a un conjunto alternativo de unidifusión.

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

2.2.10.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.

2.2.10.4. Cambio en el estado del receptor

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

2.2.10.5. Interpretación lógica

Ninguno

2.2.11. InfoSourceSubmessage

En la Figura 2-12 se muestra la estructura del submensaje InfoSource.

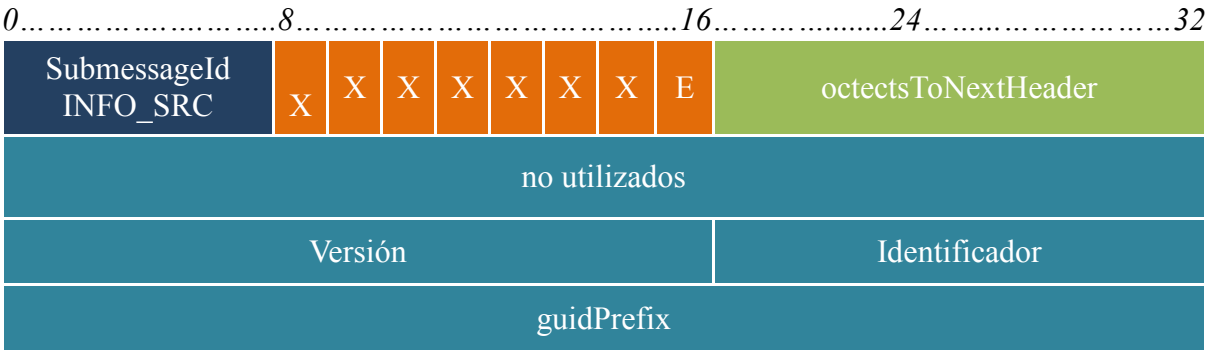


Figura 2-12. Estructura del submensaje InfoSource

(OMG, 2014)

Este mensaje modifica la fuente lógica de los submensajes que siguen.

2.2.11.1. Banderas en el encabezado de submensaje

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

2.2.11.2. Otros elementos en el encabezado de submensaje

La **protocolVersion**, indica el protocolo utilizado para encapsular submensajes posteriores.

El **identificador**, indica el identificador del vendedor que encapsule submensajes posteriores.

El **guidPrefix**, identifica el participante es el contenedor de las entidades RTPS **escritor** que son la fuente de los submensajes que siguen.

2.2.11.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.

2.2.11.4. Cambio en el estado del receptor

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

2.2.11.5. Interpretación lógica

Ninguno

2.2.12. InfoTimestampSubmessage

En la Figura 2-13 se muestra la estructura del submensaje InfoTimestamp.

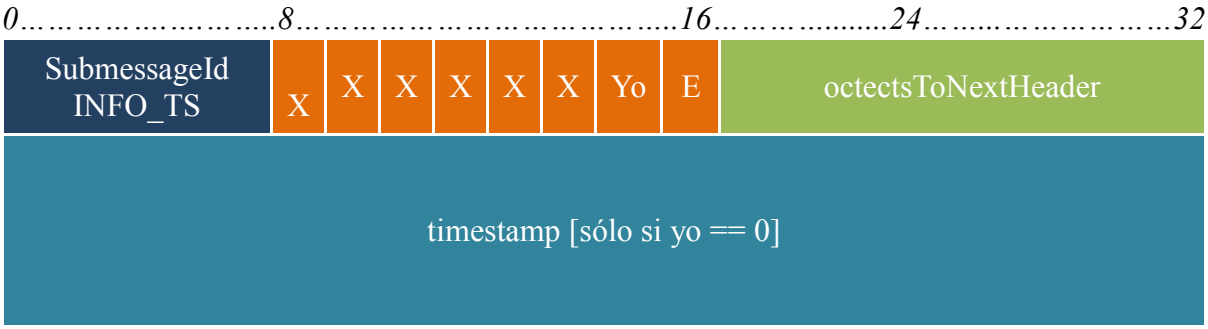


Figura 2-13. Estructura del submensaje InfoTimestamp

(OMG, 2014)

Este submensaje se utiliza para enviar una fecha y hora que se aplica a los submensajes que siguen dentro del mismo mensaje.

2.2.12.1. Banderas en el encabezado de submensaje

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

La **InvalidateFlag**, indica si submensajes posteriores deben considerarse como una marca de tiempo o no.

El InvalidateFlag se representa con el literal 'I'. Yo = 0 significa que el InfoTimestamp también incluye una marca de tiempo. Yo = 1 significa submensajes posteriores no deben considerarse que tienen una marca de tiempo válida.

2.2.12.2. Otros elementos en el encabezado de submensaje

El **timestamp**, presente solamente si el InvalidateFlag no se encuentra en la cabecera. Contiene la fecha y hora que debe utilizarse para interpretar los submensajes posteriores.

2.2.12.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.

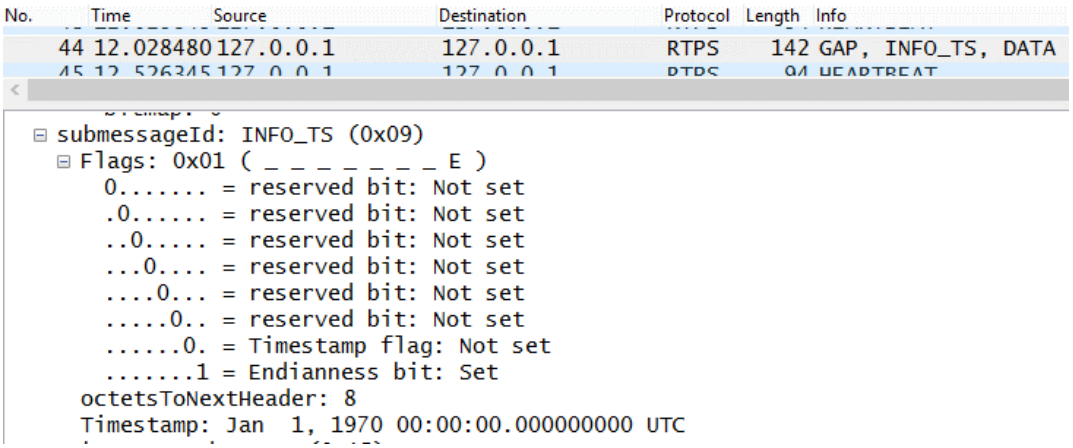
2.2.12.4. Cambio en el estado del receptor

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

2.2.12.5. Interpretación lógica

Ninguno

Ejemplo



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

2.2.13. NackFragSubmessage

En la Figura 2-14 se muestra la estructura del submensaje NackFrag.

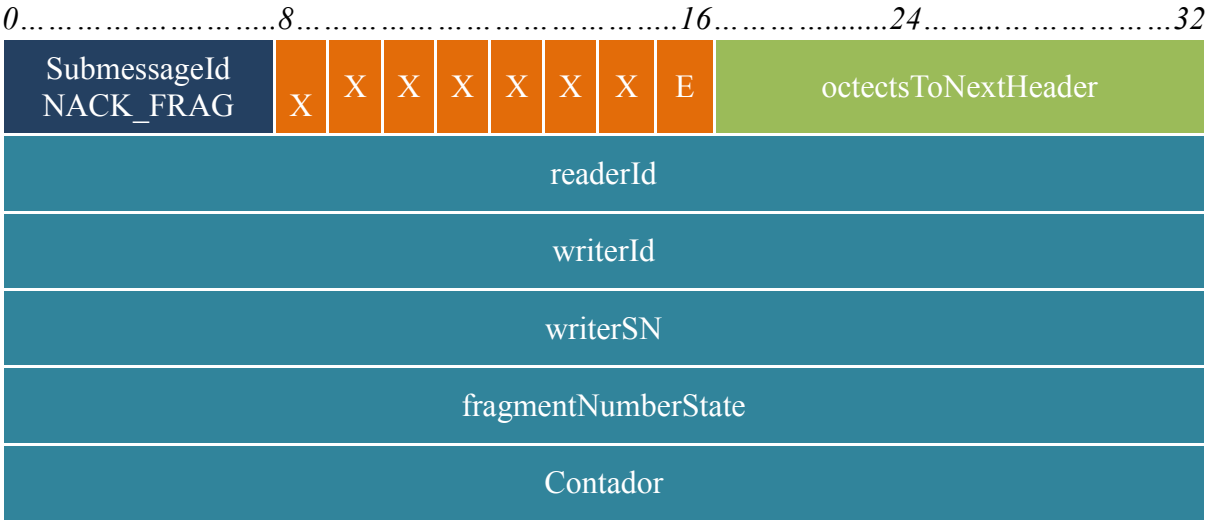


Figura 2-14. Estructura del submensaje NackFrag

(OMG, 2014)

El submensaje NackFrag se utiliza para comunicar el estado de un lector a un escritor. Cuando se envía un cambio de datos como una serie de fragmentos, el submensaje NackFrag permite al lector para dar a conocer al escritor números fragmento específico aún falta.

Este submensaje sólo puede contener caracteres negativos agradecimientos. Tenga en cuenta que esto difiere de un submensaje AckNack, que incluye reconocimientos tanto positivos como negativos. Las ventajas de este enfoque incluyen:

- Elimina la limitación de ventanas introducida por el submensaje AckNack. Dado el tamaño de un SequenceNumberSet se limita a 256, un submensaje AckNack se limita a tocándose la muestra sólo aquellas muestras cuyo número de secuencia no no exceda de los primeros desaparecidos por más de 256. Las muestras por debajo

de las primeras muestras desaparecidas son reconocidas. NackFrag submensajes por otro lado puede ser utilizado para NACK cualquier fragmento números, incluso fragmentos más than256 aparte de esos NACKed en un anterior submensaje AckNack. Esto llega a ser importante cuando manejo las muestras que contienen una gran cantidad de fragmentos de.

- Fragmentos pueden reconocerse negativamente en cualquier orden.

2.2.13.1. Banderas en el encabezado de submensaje

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

2.2.13.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad lector que pide para recibir ciertos fragmentos.

El **writerId**, identifica la entidad escritor que es el objetivo del mensaje NackFrag. Esta es la entidad del escritor que se piden a re-enviar algunos fragmentos.

El **writerSN**, el número de secuencia que faltan algunos fragmentos.

El **fragmentNumberState**, se comunica el estado del lector al escritor. Los fragmento de números que aparecen en el conjunto indican fragmentos perdidos en el lado del lector. Los que no aparecen en el conjunto son indeterminados (podría haber sido recibido o no).

El **Contador**, un contador que se incrementa cada vez que se envía un mensaje NackFrag nuevo. Proporciona los medios para un escritor detectar los mensajes duplicados de NackFrag que pueden derivarse de la presencia de vías de comunicación redundante.

2.2.13.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.
- writerSN valor es cero o negativo.

- fragmentNumberState no es válida.

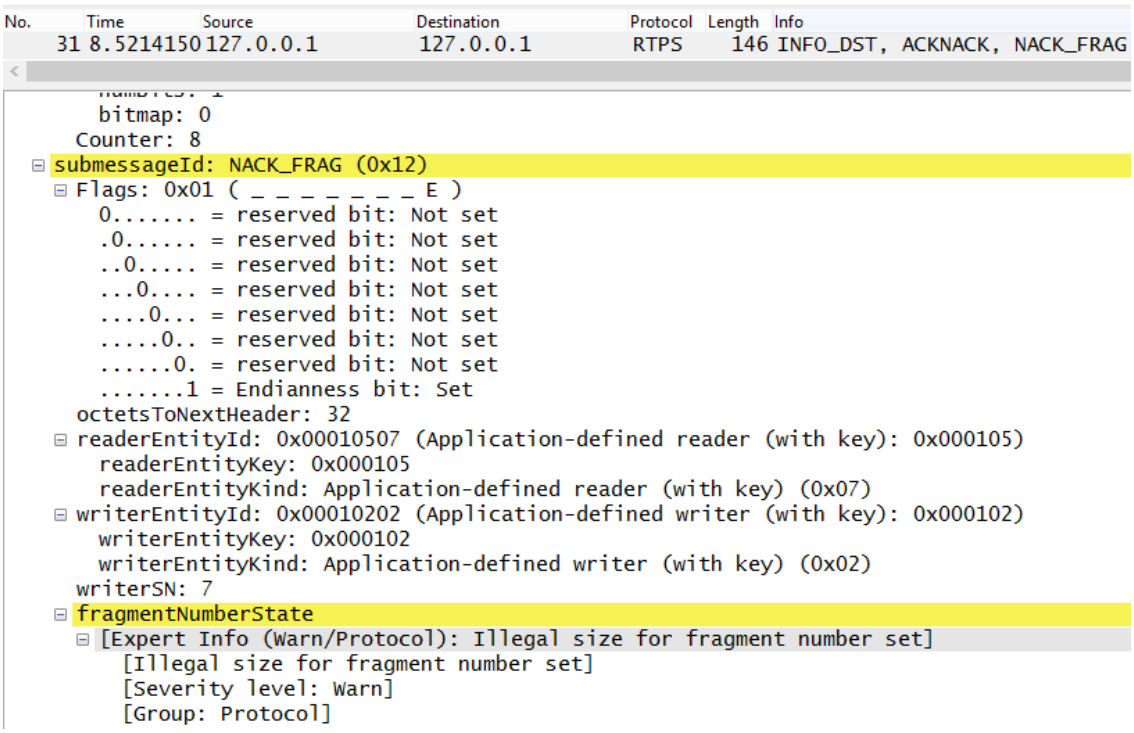
2.2.13.4. Cambio en el estado del receptor

Ninguno

2.2.13.5. Interpretación lógica

El lector envía el mensaje NackFrag al escritor para solicitar fragmentos del escritor.

Ejemplo



2.2.14. PadSubmessage

En la Figura 2-15 se muestra la estructura del submensaje Pad.

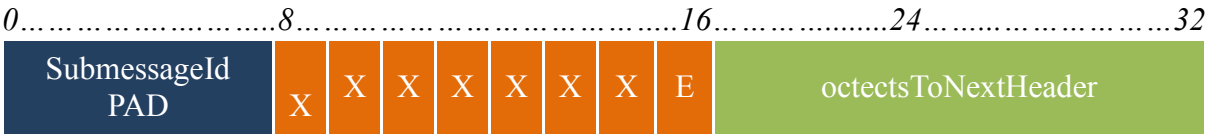


Figura 2-15. Estructura del submensaje Pad

(OMG, 2014)

El propósito de este submensaje es permitir la introducción de cualquier relleno necesario para satisfacer cualquier requisito de memoryalignment deseada. Su no tiene otro significado.

2.2.14.1. Banderas en el encabezado de submensaje

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

2.2.14.2. Otros elementos en el encabezado de submensaje

Este submensaje no tienen otros elementos.

2.2.14.3. Validez

Este submensaje es siempre válido.

2.2.14.4. Cambio en el estado del receptor

Ninguno

2.2.14.5. Interpretación lógica

Ninguno

2.2.15. InfoReplyIp4Submessage

En la Figura 2-16 se muestra la estructura del submensaje InfoReplyIp4.

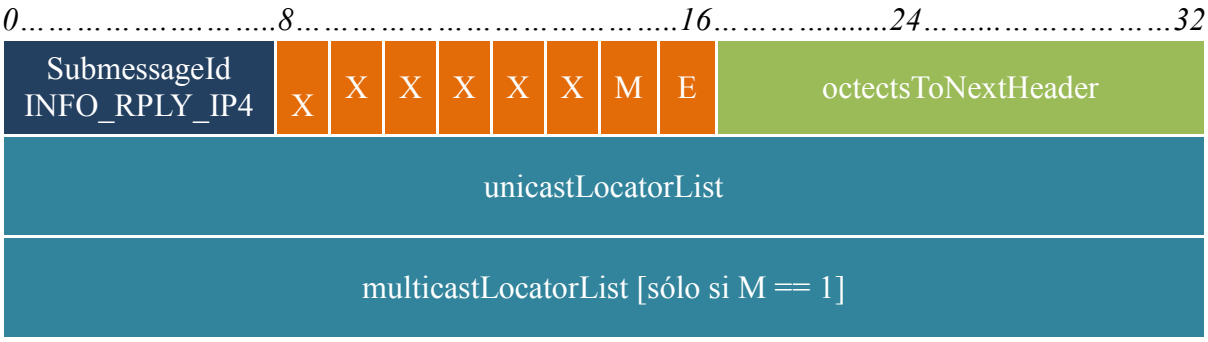


Figura 2-16. Estructura del submensaje InfoReplyIp4

(OMG, 2014)

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

Se proporciona por motivos de eficiencia y puede utilizarse en lugar del submensaje InfoReply para proporcionar una representación más compacta.

2.2.15.1. Banderas en el encabezado de submensaje

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

La **MulticastFlag**, indica si el submensaje también contiene una dirección de multidifusión.

El MulticastFlag está representado con el literal estoy '. M = 1 significa que la InfoReplyIp4 también incluye un multicastLocatorList.

2.2.15.2. Otros elementos en el encabezado de submensaje

La **unicastLocatorList**, indica que el escritor debe utilizar para llegar a los lectores al responder a los submensajes que siguen se dirige a un conjunto alternativo de unidifusión.

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

2.2.15.3. Validez

Este submensaje es inválida cuando cualquiera de las siguientes es verdadera:

- submessageLength en el encabezado de submensaje es demasiado pequeño.

2.2.15.4. Cambio en el estado del receptor

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

2.2.15.5. Interpretación lógica

Ninguno

2.3. ANÁLISIS DE REQUISITOS

Una vez realizado el análisis de los paquetes RTPS y de descubrimiento RTPS, se puede tener una idea clara de los requerimientos necesarios para soportar el protocolo RTPS con el middleware DDS.

Este módulo estará compuesto por cinco componentes principales: Implementación de los módulos DDS, Implementación de los mecanismos y técnicas para el alcance de la información, Lectura y Escritura de datos, Implementación de módulos RTPS, e Implementación de protocolos de descubrimiento RTPS.

2.4. MÓDULO DDS

En el DDS se encuentra el Publicador, el Suscriptor, y el Topic.

2.4.1. Publicador

El **Publicador** es el objeto responsable de la distribución de datos. Puede publicar los datos de los diferentes tipos de datos. Un DataWriter actúa como un *typed*[28] de acceso a una publicación. Un DataWriter es el objeto que la aplicación debe utilizar para comunicar a un publicador de la existencia y el valor de los objetos de datos de un tipo dado. Cuando los valores de los datos de los objetos han sido comunicados al publicador a través de un DataWriter

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

apropiado, es responsabilidad del publicador realizar la distribución (el publicador hará esto de acuerdo a sus propias políticas de calidad de servicio conectados a la correspondiente DataWriter).

Una publicación está definida por la asociación de un DataWriter con un publicador, esta asociación expresa la intención de la aplicación de publicar los datos descritos por el DataWriter en el contexto proporcionado por el publicador.

2.4.2. Suscriptor

El **Suscriptor** es un objeto responsable de recibir los datos publicados ponerlos a disposición de acuerdo con el QoS del suscriptor de la aplicación receptora.

Puede recibir y despachar datos de los diferentes tipos especificados. Para acceder a los datos recibidos, la aplicación debe utilizar un *typed* DataReader adjunto al suscriptor.

Una suscripción está definida por la asociación de un DataReader a un suscriptor, esta asociación expresa el intento de la aplicación para suscribirse a los datos descritos por el DataWriter en el contexto proporcionado por el suscriptor.

2.4.3. Topic

Un **Topic** representa la unidad de información que puede ser producida o consumida; es una triada, compuesta por un tipo, un nombre único y un conjunto de políticas de calidad de servicio, como se muestra en la Figura 2-17, que se utiliza 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 2-17. Objeto Topic y sus componentes.

Los Topic son objetos que conceptualmente encajan entre las publicaciones y suscripciones. Las publicaciones deben ser conocidas del tal manera que las suscripciones puedan referirse a ellas sin ambigüedades. El Topic tiene 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.

Adicionalmente al Topic de QoS, la calidad de servicio del *DataWriter* asociada a este Topic y la QoS del publicador asociado al *DataWriter* controlan el comportamiento de parte del publicador, mientras la QoS de los Topic, *DataReader* y el Suscriptor, controlan el comportamiento en el lado del suscriptor.

Un tipo Topic DDS es descrito por una estructura IDL[29] que contiene un número arbitrario de campos cuyos tipos podrían ser; tipo primitivo como se muestra en la Tabla 2-1, un tipo template como se muestra en la Tabla 2-2, o un tipo compuesto como se muestra en la Tabla 2-3

[29] IDL, Interface Definition Language.

Tabla 2-1. Tipos IDL primitivos.

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

Tabla 2-2. 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<octet> oseq; sequence<octet, 1024> oseq1k;
	sequence<MyType> mtseq; sequence<MyType, 10> mtseq10;
fixed<digits,scale>	fixed<5,2> fp; //d1d2d3.d4d5

Tabla 2-3. 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* deben ser conocidos por el middleware y potencialmente propagados. Los objetos del *Topic* son creados utilizando las operaciones de creación proporcionadas por el *DomainParticipant*.

La interacción es directa por parte del publicador: cuando la aplicación sabe u hacer con los datos disponibles para la publicación, llama a la operación correspondiente al *DataWriter* relacionado.

La interacción con el suscriptor muestra más opciones: la información relevante puede llegar cuando la aplicación está ocupada o cuando la aplicación está a la espera de esa información. Es decir que dependiendo de la forma en que la aplicación está diseñada, se utilizaran notificaciones asíncronas o síncronas. Ambos modos de interacción están permitidos, para llamadas de acceso sincrónicas se utilizan *Listener*, para llamadas de acceso asincrónico se utilizan *WaitSet* asociados con objetos *Condition*.

Estos modos de interacción también se pueden usar para acceder a los cambios que afectan el estado de la comunicación dentro del middleware.

2.5. MECANISMO Y TÉCNICAS PARA EL ALCANCE DE LA INFORMACIÓN

Los dominios y las particiones son una manera de organizar los datos, sin embargo, operan a un nivel estructural. El Topic DDS Content-Filtered permite crear topics, que limitan los valores que pueden tomar sus instancias. Al suscribirse a un **topic content-filtered** una aplicación, sólo recibirá, entre todos los valores publicados aquellos valores que coincidan con el filtro de *topic*. Los operadores de los filtros y condiciones de consultas se muestran en la siguiente Tabla 2-4.

Tabla 2-4. 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 *topic*, limitan la cantidad de memoria utilizada por el middleware para las instancias. Una forma menos usada en lugar de los *topic Content-Filtered* son los *Query Condition* y se los ejecuta en el contexto de una lectura del tipo *read*.

2.6. LECTURA Y ESCRITURA DE DATOS

2.6.1. Escritura de Datos

Escribir datos dentro de DDS es un proceso simple ya que solo se debe llamar al método write del DataWriter.

Un DataWriter permite a una aplicación establecer el valor de los datos para ser publicados bajo un determinado Topic.

Hay que establecer bien la diferencia entre *writers* y *topic-instances* del ciclo de vida, para ello, se usará la analogía que existe entre *topic* y las instancias del *topic* y los objetos de clase en un lenguaje orientado a objetos. Tanto los objetos como los *topic-instances* tienen una identidad proporcionada por su valor de clave única, y un ciclo de vida.

El ciclo de vida del *topic-instances*, puede ser manejado implícitamente a través de la semántica implicada por el *DataWriter*, o puede ser controlado 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.

2.6.2. Ciclo de Vida de los Topic-Instances

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

- ALIVE, si por lo menos hay un *DataWriter* escribiendo.
- NOT_ALIVE_NO_WRITERS, cuando no hay mas *DataWriters*, escribiendo.
- NOT_ALIVE_DISPISED, si ha sido eliminada 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 indica 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 sistemas para almacenar la instancia podrán ser liberados.

2.6.2.1. 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 2-5, se puede notar que solamente es para escribir datos, y existen tres operaciones *write*, las cuales crean tres nuevas instancias de *topic* en el sistema, los cuales están asociados a los id=1, 2, 3.

Estando estas instancias en el estado ALIVE, las instancias serán registradas automáticas, es decir que están asociadas con el *writer*.

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

Tabla 2-5. 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;`
`}`

2.6.2.2. Administración de Ciclo de Vida Explícita

El ciclo de vida de *topic-instances* puede ser manejado explícitamente por el *API* definido en el *DataWriter*. En este caso el programador de la aplicación tiene el control sobre cuando las instancias son registradas, se dejan de registrar o se eliminan. El registro de *topic-instances* es una práctica buena a seguir cuando una aplicación escribe una instancia muy a menudo y requiere la escritura de latencia más baja. En esencia, el acto de registrar explícitamente una instancia permite al middleware reservar recursos, así como optimizar la búsqueda de instancias. Dejar de registrar un *topic-instances* proporciona una manera para decir al DDS que una aplicación se realiza escribiendo un *topic-instances* específico, por lo tanto, todos los recursos asociados locamente pueden ser liberados de forma segura. Finalmente,

eliminar un topic-instances da 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 deben ser liberados tanto de forma local como remota.

2.6.2.3. Topic sin claves

Los *topic* sin claves los convierte en únicos, en el sentido que hay sólo una instancia. En los *topic* sin clave el estado de transición es vinculado al ciclo de vida del *DataWriter*.

2.6.3. Lectura de Datos

DDS tiene dos mecanismos diferentes para acceder a los datos, cada uno de ellos permito controlar el acceso a los datos y la disponibilidad de los mismos.

El acceso a los datos se realiza a través de la clase *DataReader* exponiendo dos semánticas para acceder a los datos: *read* y *take*.

2.6.3.1. Read y Take

La semántica de *read* implementada por el método *DataReader::read*, da acceso a los datos recibidos por el *DataReader* sin sacarlo de su caché local. Por lo cual estos datos serán nuevamente legibles mediante una llamada apropiada al *read*. Así mismo, el DDS proporciona una semántica de *take*, implementado por los métodos *DataReader::take* que permite 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 son más disponibles para subsecuente operaciones *read* o *take*.

La semántica proporcionada por las operaciones *read* y *take* permiten usar el DDS como una caché distribuida 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, otras como tecnología de mensajería de alto rendimiento, y sin embargo, otras veces como una combinación de las dos.

El *read* y el *take* DDS se encuentran siempre no bloqueados. 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. La naturaleza de las operaciones de *read/take* de no-bloqueo asegura que estos pueden utilizarse con seguridad por aplicaciones que sondean para datos.

2.6.4. 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 está a disposición de *DataReader* y pueden ser utilizadas para seleccionar los datos accedidos a través de *read* o ya sea de *take*.

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

- **Estado de la muestra.** El estado de la muestra puede ser READ o NOT_READ dependiendo si la muestra ya ha sido leída o no.
- **Estado de la instancia**. Esto indica el estado de la instancia como ALIVE, NOT_ALIVE_NO_WRITERS, o NOT_ALIVE_DISPOSED.
- **Estado de la vista.** El estado de vista puede ser NEW o NOT_NEW dependiendo de si es la primera muestra recibida por el *topic-instance* determinado o no.

El *SampleInfo* también contiene un conjunto de contadores que permite reconstruir el número de veces que el *topic-instance* ha realizado cierta transición de estado como convertirse en *alive* después de *disposed*.

Finalmente, el *SampleInfo* contiene un *timestamp* (fecha y hora) para los datos y una bandera que dice si la muestra es asociada o no. Esta bandera es importante ya que el DDS puede generar información válida de las muestras con datos no válidos para informar acerca de las transiciones de estado como una instancia de ser desechado.

2.6.5. Notificaciones

Una forma de coordinar con DDS es 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, sin embargo, las aplicaciones podrían querer ser notificadas de la disponibilidad de datos o tal vez esperar su disponibilidad, como lo opuesto al sondeo. DDS apoya la coordinación tanto síncrona y asíncrona por medio de *waitsets* y los *listeners*.

2.6.5.1. Waitsets

DDS proporciona un mecanismo genérico para esperar en condiciones. Uno de los tipos soportados de condiciones son las condiciones de lectura, las cuales pueden ser usadas para esperar la disponibilidad de los datos de uno o más *DataReaders*.

2.6.5.2. Listeners

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

Los mecanismos de control de eventos permiten enlazar cualquier cosa que se quiera a un evento DDS, lo que significa que se puede enlazar una función, un método de clase, etc. El contrato sólo con que necesita cumplir es la firma que se espera por la infraestructura. Por

ejemplo, cuando se trata con el evento *on_data_available* tiene que registrar una entidad exigible que acepta un único parámetro de tipo *DataReader*.

Finalmente, vale mencionar algo en el código, el *handler* se ejecutará en un *thread* de middleware. Como resultado, cuando se utilizan *listeners* se debe probar a minimizar el tiempo del *listener* empleado en el mismo.

2.7. MÓDULO RTPS

Los módulos RTPS están definidos por la [30]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 extremos de los *Writers* y *Readers*. Estos extremos se comunican usando los mensajes descritos en el **módulo de mensajes** que se encuentra descrito más adelante. También es necesario describir el comportamiento de la máquina virtual, por medio del **módulo de comportamiento**, el cual es descrito de igual manera posteriormente y por el cual se observa el intercambio de mensajes que debe tomar lugar entre los extremos. Y finalmente se encuentra el protocolo de descubrimiento usado para configurar la máquina virtual con la información que esta necesita para comunicar pares remotos, este protocolo se encuentra descrito en el **módulo de descubrimiento**.

2.7.1. Módulo estructura

El propósito principal de este módulo es describir las clases principales usadas por el protocolo RTPS, como se puede observar en la Figura 2-18.

[30] PIM, Plataform Independent Model

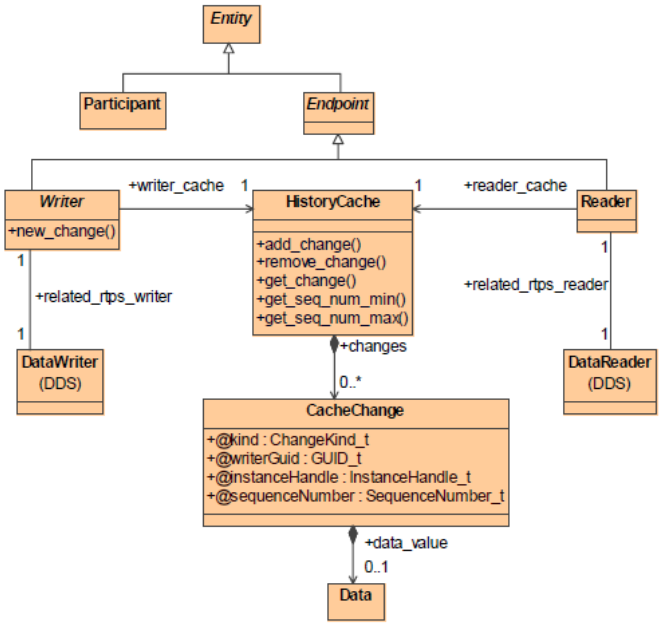


Figura 2-18. Módulo Estructura

Las entidades RTPS muestran a los extremos que son utilizado por las entidades DDS para que estas se comuniquen entre sí. Cada Entidad RTPS tiene correspondencia uno a uno con las Entidades DDS. El **HistoryCache** proporciona una interfaz entre las entidades DDS y su correspondiente entidad RTPS. Por ejemplo, cada operación *write* en un *DataWriter* DDS añade un **CacheChange** al **HistoryCache** en su correspondiente RTPS *Writer*. El RTPS *Writer* subsecuentemente transfiere el **CacheChange** al **HistoryCache** de cada RTPS *Reader* asociado. En el lado recibido, el DDS *DataReader* es notificado por el RTPS *Reader* que un nuevo **CacheChange** ha llegado en el **HistoryCache**.

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

Como podremos observar todas las entidades de RTPS son derivadas de la clase *Entity* del RTPS, como se muestra en la Tabla 2-6.

Tabla 2-6. Clases y Entidades RTPS

Entidades y Clases RTPS

Clase	**Propósito**

Entity	Es la clase base para todas las entidades RTPS. *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	Especialización de 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	Especialización de la representación de objetos *Endpoints* que puede ser origen de mensajes que comunican *CacheChanges*.
Reader	Especialización de la representación de objetos *Endpoints* que puede ser destino de mensajes que comunican *CacheChanges*.
HistoryCache	Clase contenedora usada para almacenar temporalmente y gestionar grupos de cambio a *data-objects*. En el lado del escritor este contiene la historia de los cambios en el objeto de datos hechos por el escritor. El uso de esta historia o historia parcial, depende las políticas de QoS del DDS y del estado de comunicaciones con los lectores asociados. 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 el gráfico xx podemos observar un diagrama de clases del *HistoryCache.*
CacheChange	Representa un cambio individual que se ha hecho al objeto de datos. Estos incluyen a la creación, modificación, y eliminación de objetos de datos.
Data	Representa a los datos que deben ser asociados con un cambio hecho al objeto de datos.

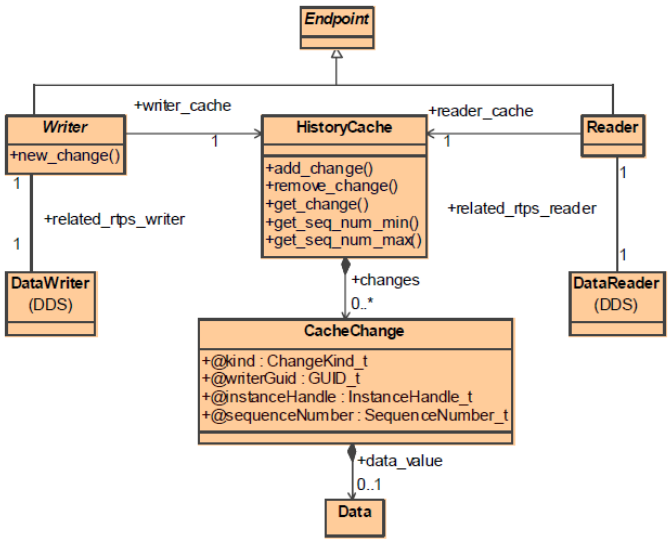


Figura 2-19. HistoryCache

2.7.2. Módulo Mensajes

Este módulo 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 puede ampliar fácilmente para apoyar tanto nuevas características del protocolo, así como extensiones específicas del proveedor.

2.7.2.1. 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 2-20.

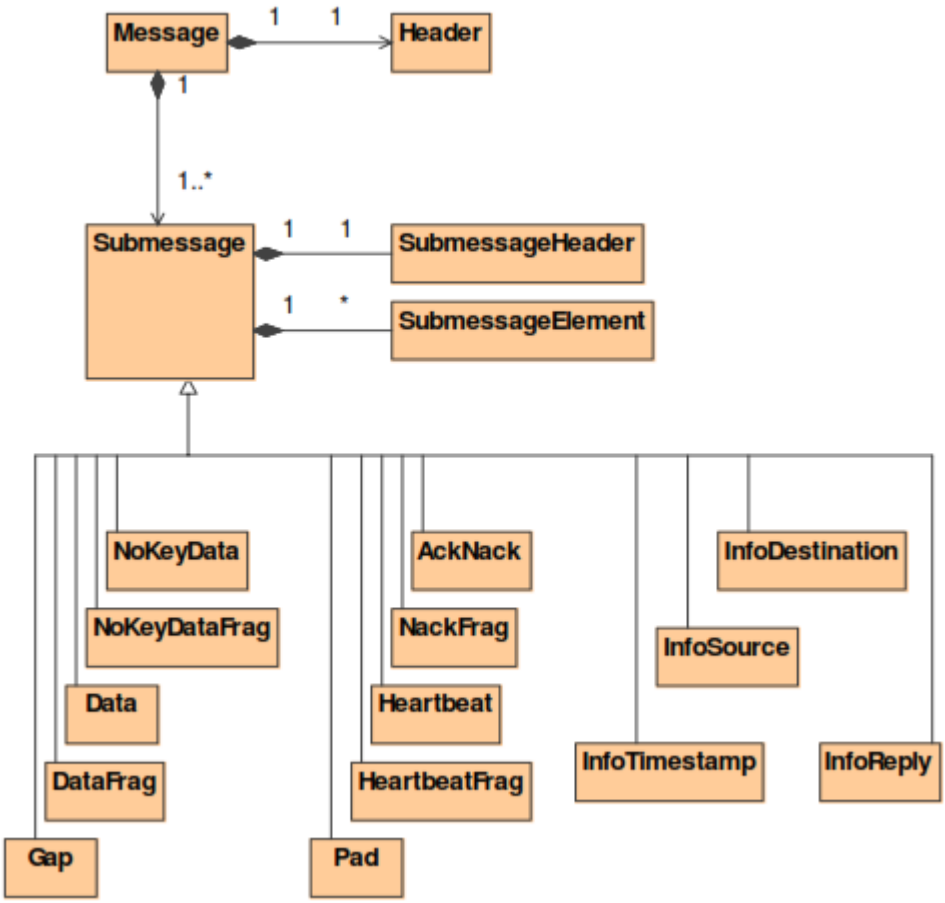


Figura 2-20. Estructura del mensaje RTPS.

Cada mensaje enviado por el protocolo RTPS tiene una longitud finita. Esta longitud no se envía explícitamente por el protocolo RTPS pero es parte del transporte subyacente con la que se envían 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. Un transporte orientado a conexión (como TCP) sería necesario insertar la longitud del mensaje con el fin de identificar el límite del mensaje RTPS.

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 2-21.

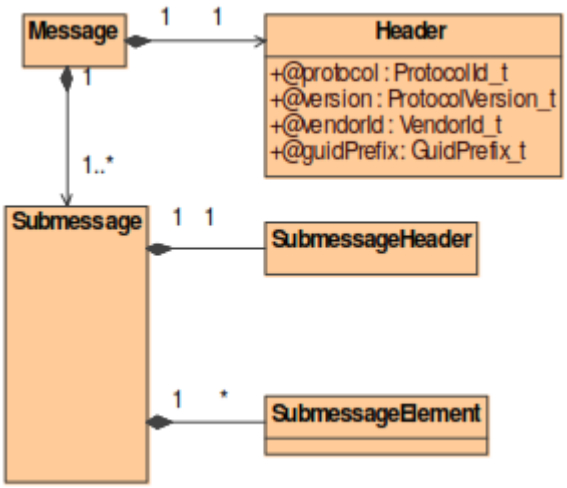


Figura 2-21. Estructura de la cabecera del mensaje RTPS.

Estructura del submensaje

Cada submensaje RTPS consiste 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 2-22.

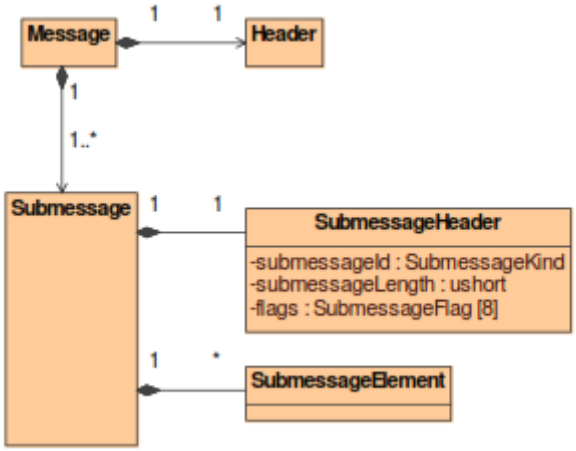


Figura 2-22. Estructura de los submensajes RTPS.

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 *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 (no está incluido en la cabecera del submensaje).

El *SubmessageLength*> 0, o bien es:

- La longitud desde el comienzo de los contenidos del submensaje hasta el comienzo de la cabecera del siguiente submensaje (en este caso el submensaje no es el último submensaje en el mensaje).
- O es la longitud del mensaje restante (en este caso el submensaje es el último submensaje del mensaje).

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

El *SubmessageLength* == 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 (es la longitud máxima que se puede almacenar en el campo *submessageLength*), siempre que sean el último submensaje en el mensaje.

2.7.2.2. RTPS Message Receiver

La interpretación y significado de un submensaje dentro de un mensaje puede depender de los submensajes previos dentro de ese mismo mensaje. Por lo tanto, el receptor de un mensaje debe mantener el estado de 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 2-23.

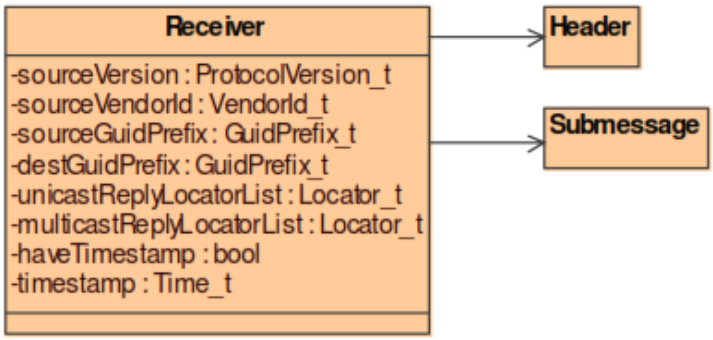


Figura 2-23. Receptor RTPS.

Reglas seguidas por el Receptor del mensaje

El siguiente algoritmo 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. En concreto: una

implementación de RTPS 2.2 debe ignorar cualquier Mensajes complementarios con ID que estén fuera del conjunto *SubmessageKind* definido en la versión 2.2. *SubmessageId* en el rango específico del fabricante que vienen de un *vendorID* que se desconoce también debe ser ignorada y el análisis debe continuar con el siguiente submensaje.

- Las banderas del submensaje, el receptor de un submensaje debe ignorar banderas desconocidos. Una implementación con RTPS 2.2 debe saltar todas las banderas que están marcados como "X" (sin usar) en el protocolo.
- 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.
- Puede afectar el comportamiento del punto final al que está destinado el mensaje. Esto se aplica a los mensajes básicos RTPS, tales como: Datos, DataFrag, HeartBeat, AckNack, Gap, HeartbeatFrag, NackFrag.

2.7.2.3. 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 de construcción atómicas predefinidos 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 2-24.

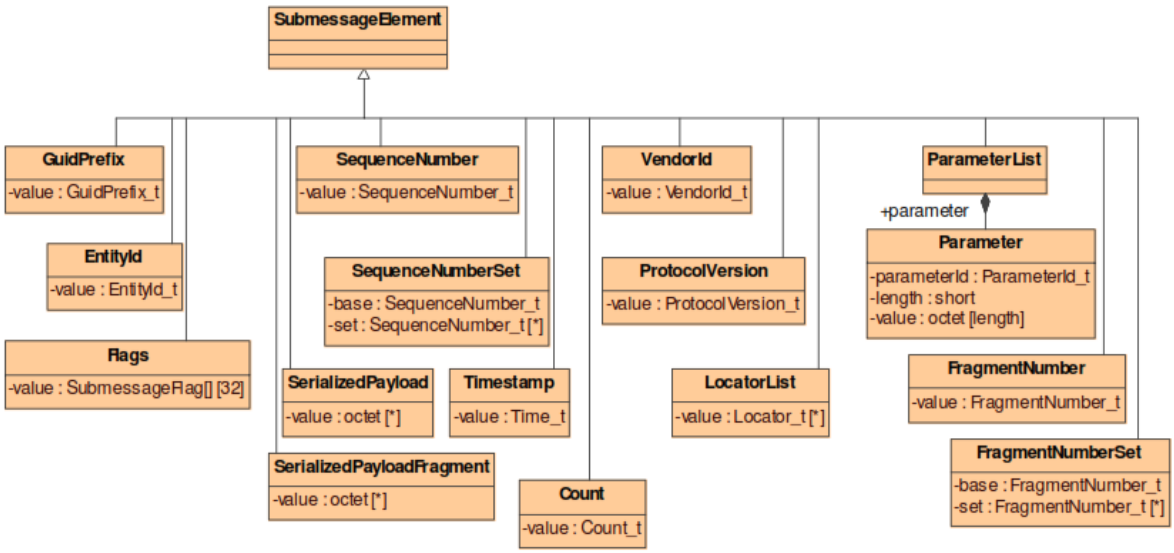


Figura 2-24. Elementos de submensaje RTPS.

El detalle de los elementos del submensaje RTPS se encuentra en la sección del Módulo Mensajes de la norma (OMG, 2014).

2.7.2.4. 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 proporcionar contexto que ayuda a los procesos posteriores del Entity submessage.

Las entidades del submensaje son:

- Los *datos*, contiene información sobre el valor de un objeto fecha de la aplicación. Los submensajes de datos son enviados por *Writer* (**NO_KEY Writer** o **WITH_KEY Writer**) a un *Reader* (**NO_KEY Reader** o **WITH_KEY Reader**).

- El *DataFrag*, equivale a los datos, pero sólo contiene una parte del nuevo 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* (**NO_KEY Writer** o **WITH_KEY Writer**) a uno o más *Reader* **(NO_KEY Reader** o **WITH_KEY** Reader**)**.
- El *HeartbeatFrag*, es para los datos fragmentados, describe que fragmentos están disponibles en un Writer. Los mensajes *HeartbeatFrag* son enviados por un *Writer* **(NO_KEY Writer o WITH_KEY Writer)** a uno o más *Reader* **(NO_KEY Reader o WITH_KEY 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 una o más *Writer*.
- El *NackFrag,* proporciona información sobre el estado de un *Reader* a un *Writer*, más específicamente los fragmentos de información que siguen perdidos en el *Reader*. Los mensajes *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 submensajes RTPS.
- El *InfoDestination*, proporciona información sobre el destino final de Entity Submessage posteriores. 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 un *TimeStamp* origen para *Entity Submessage* posteriores.
- El *Pad,* se utiliza para agregar relleno a un mensaje, si es necesario para la alineación de la memoria.

A continuación se muestran los diferentes submensajes RTPS en la Figura 2-25.

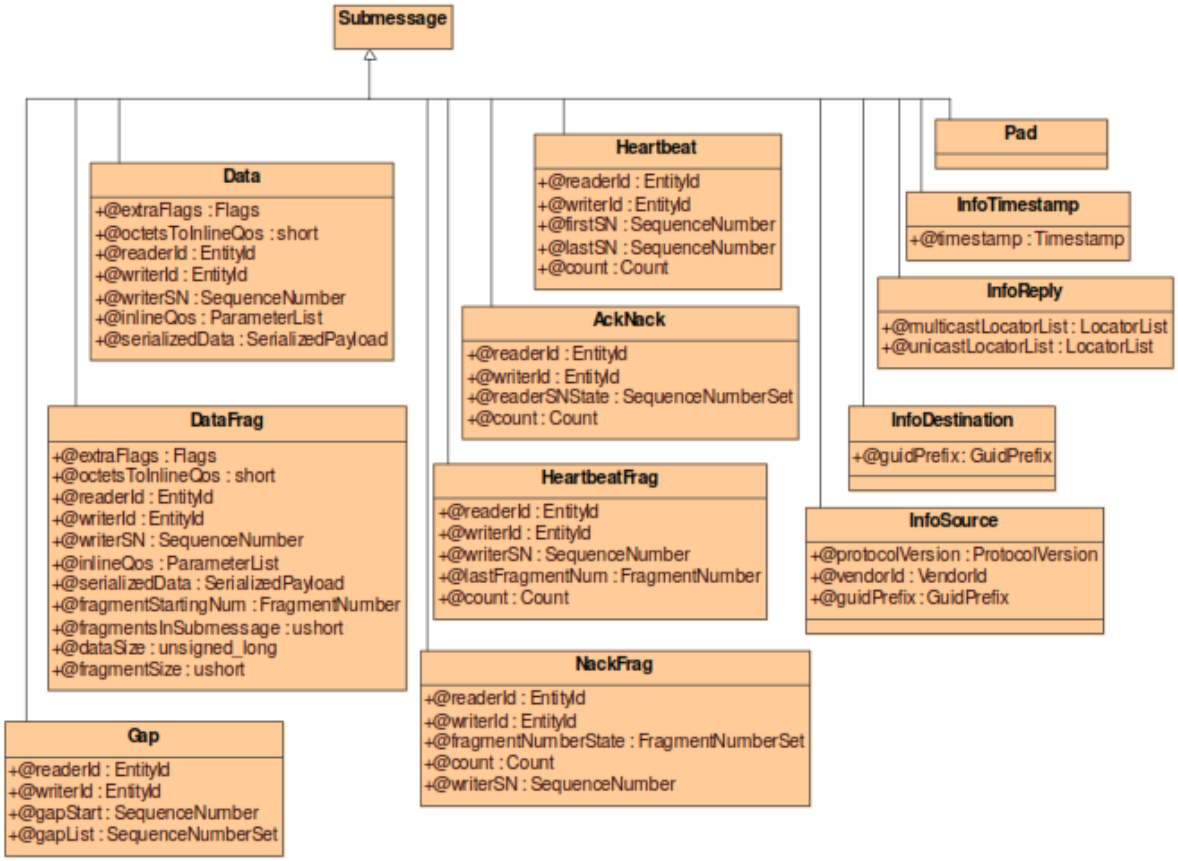


Figura 2-25. Submensajes RTPS.

El detalle de los tipos de submensajes RTPS se encuentra en la sección del Módulo Mensajes de la norma (OMG, 2014).

2.7.3. Módulo Comportamiento

Una ver que el RTPS *Writer* ha sido asociado a un RTPS *Reader*, 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*.

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 anterior punto.

2.7.3.1. Requerimientos Generales

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 esté definido en los mensajes RTPS puede ser usado.
- Todas las implementaciones deben implementar a un RTPS *Message Receiver*, 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.
- Las implementaciones deben implementar al protocolo de descubrimiento denominado *Simple Participant and Endpoint Discovery Protocols*, es decir a los protocolos de descubrimientos que cubre el estándar.

2.7.3.2. Comportamiento requerido de los Writer RTPS

- Los *Writers* no deben enviar datos fuera de orden, es decir que estos deben enviar los datos en el mismo orden en el que fueron añadidos en su *HistoryCache*.

- Los *Writers* deben incluir valores *in-line QoS* si es requerido por un *Reader*, es decir que un *writer* debe respetar las solicitudes de los *Readers* para recibir mensajes de datos con QoS.
- Los *Writers* 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 heartbeats 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 acuses de recibo negativos, cuando un acuse de recibo negativo nos 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.

2.7.3.3. Comportamiento requerido de los Reader RTPS

- 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 tormentas 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 mensaje 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.

2.7.3.4. 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 mejor esfuerzo 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.

2.7.3.5. Comportamiento de un Writer respecto a Reader asociados

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 se muestran las combinaciones posibles.

Tabla 2-7. Combinación de atributos posibles en lectores asociados con escritores *(OMG, 2014)*

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

2.7.3.6. Implementación del Writer RTPS

El escritor RTPS se especializa en los extremos RTPS y representa al actor que envía mensajes al *CacheChange* de los lectores RTPS asociados.

Writer sin estado RTPS

Este escritor no tiene conocimiento del número de lectores asociados, ni tampoco mantiene cualquier estado en sus lectores RTPS asociados. Lo único que un escritor sin estado

mantiene es la lista *Locator_t* RTPS, la cual debería ser usada para enviar información a los lectores asociados.

El escritor sin estado es útil para situaciones donde un *HistoryCache* de un escritor es pequeño, o la comunicación está en modo mejor esfuerzo, o si el escritor se está comunicando vía multicast a un número grande de lectores.

Writer con estado RTPS

Este escritor está configurado con la información de los lectores asociados y mantiene el estado en cada uno de estos. Para mantener el estado con cada lector, el escritor con estado RTPS puede determinar si todos los lectores RTPS han recibido un cambio en particular del *CacheChange*, lo cual hace que esto se optimo en el uso de recursos de red, ya que se evita los anuncios de los lectores que han recibido cambios del *HistoryCache* del escritor.

La información mantenida es también útil para simplificar la implementación de QoS en el lado del escritor. Existe un *ReaderProxy* RTPS, el cual es la clase que representa la información mantenida en el escritor RTPS con cada lector RTPS.

La asociación de un escritor con estado RTPS con un lector RTPS significa que el escritor con estado enviará los cambios del *CacheChange* en el *HistoryCache* del escritor al lector RTPS asociado el cual es representado por el *ReaderProxy*. (OMG, 2014)

2.7.3.7. Comportamiento de Writer sin estado

Comportamiento de Writer sin estado con mejor esfuerzo

En la siguiente Figura 2-26 podremos observar el comportamiento de este tipo de escritor.

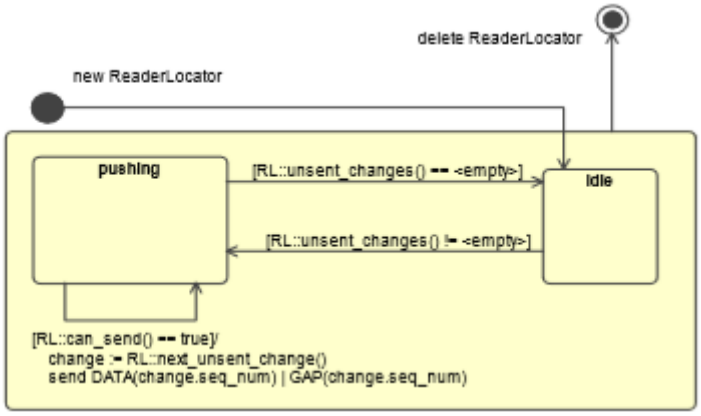


Figura 2-26. Comportamiento de un Writer sin estado con WITH_KEY *Best-Effort* con respecto a cada *ReaderLocator*

(OMG, 2014)

El listado de estados se encuentra descrito en la Tabla 2-8.

Tabla 2-8. 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	*Final*

		más al *ReaderLocator*.	

Comportamiento de Writer sin estado confiable

En la siguiente Figura 2-27 podremos observar el comportamiento de este tipo de escritor.

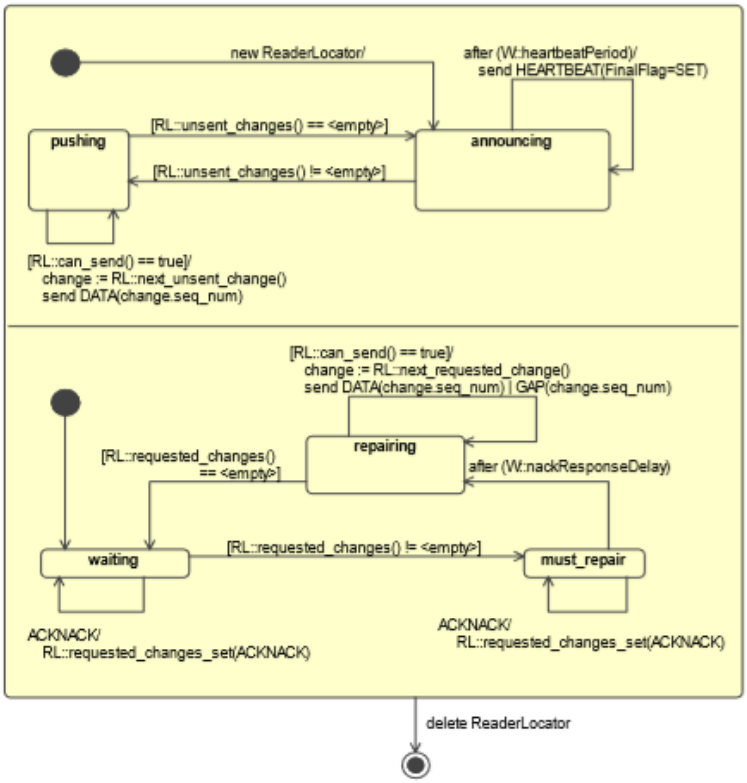


Figura 2-27. Comportamiento de un *Writer* sin estado con WITH_KEY *Reliable* con respecto a cada *ReaderLocator*

(OMG, 2014)

El listado de estados se encuentra descrito en la Tabla 2-9.

Tabla 2-9. 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	*Announcing*

		enviados al *ReaderLocator*.	
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*
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	*Final*

		para no mantener más al *ReaderLocator*.	

2.7.3.8. Comportamiento de Writer con estado

Comportamiento de Writer con estado con mejor esfuerzo

En la siguiente Figura 2-28 podremos observar el comportamiento de este tipo de escritor.

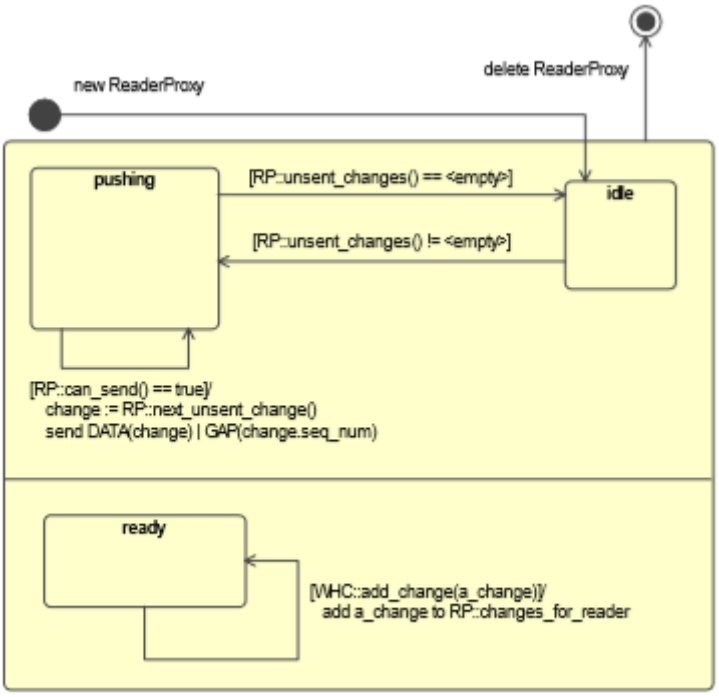


Figura 2-28. Comportamiento de un Writer con estado con WITH_KEY *Best-Effort* con respecto a cada *ReaderLocator*

(OMG, 2014)

El listado de estados se encuentra descrito en la Tabla 2-10.

Tabla 2-10. 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*
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 2-29 podremos observar el comportamiento de este tipo de escritor.

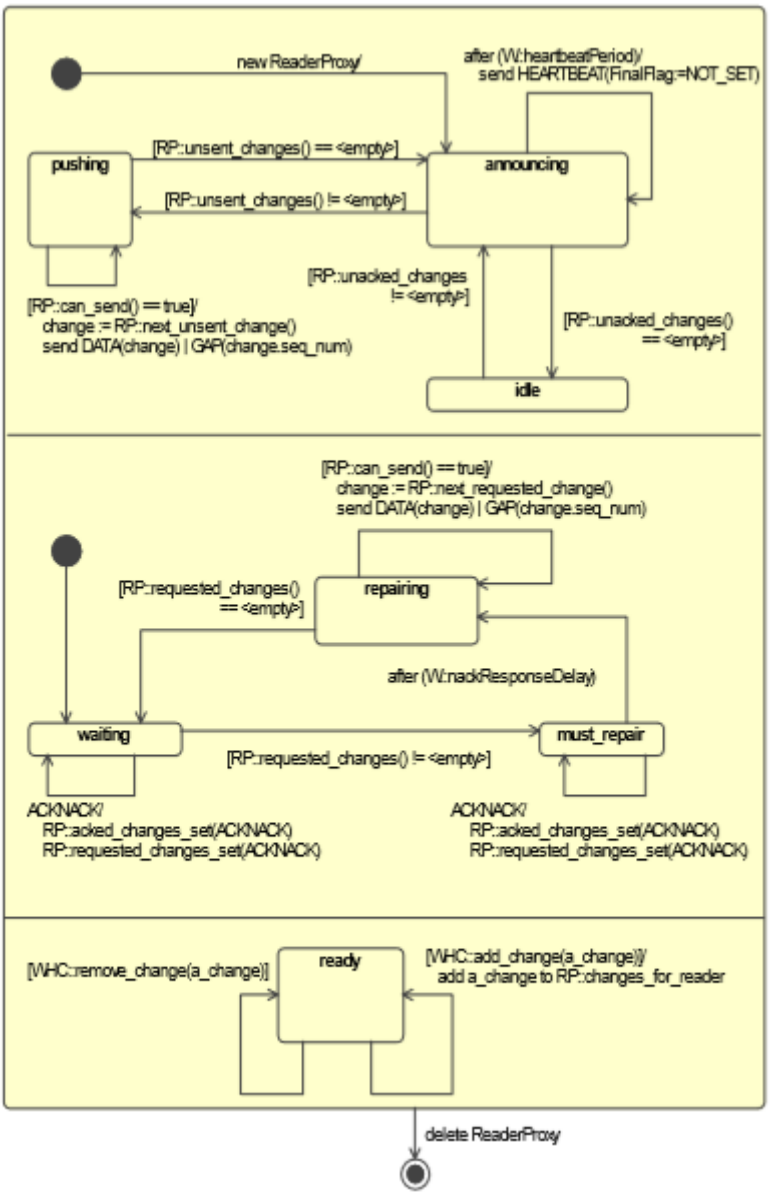


Figura 2-29. Comportamiento de un Writer con estado con WITH_KEY *Reliable* con respecto a cada *ReaderLocator*

(OMG, 2014)

El listado de estados se encuentra descrito en la Tabla 2-11.

Tabla 2-11. 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*	*Final*

		representado por el *ReaderProxy*.	

2.7.3.9. Implementación del Reader RTPS

El lector RTPS se especializa en los extremos RTPS que reciben mensajes del *CacheChange* desde uno o más extremos escritores RTPS.

Reader sin estado RTPS

Este lector no tiene conocimiento del número de escritores asociados, y además no mantiene ningún estado con cada escritor asociado.

Reader con estado RTPS

Este lector mantiene el estado con cada escritor asociado, y además este estado es encapsulado en un *WriterProxy*.

El *WriterProxy* representa la información que un *Reader* con estado mantiene con cada *Writer* asociado.

2.7.3.10. Comportamiento de Reader sin estado

Comportamiento de Reader sin estado con mejor esfuerzo

En la siguiente Figura 2-30 podremos observar el comportamiento de este tipo de lector.

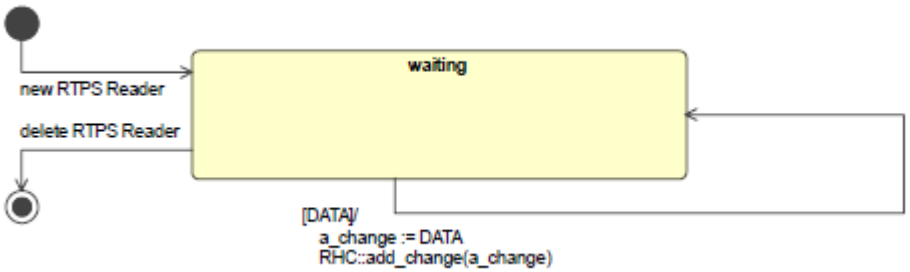


Figura 2-30. Comportamiento de un *Reader* sin estado con WITH_KEY Best-Effort *(OMG, 2014)*

El listado de estados se encuentra descrito en la Tabla 2-12.

Tabla 2-12. 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.

2.7.3.11. Comportamiento de Reader con estado

Comportamiento de Reader con estado con mejor esfuerzo

En la siguiente Figura 2-31 podremos observar el comportamiento de este tipo de lector.

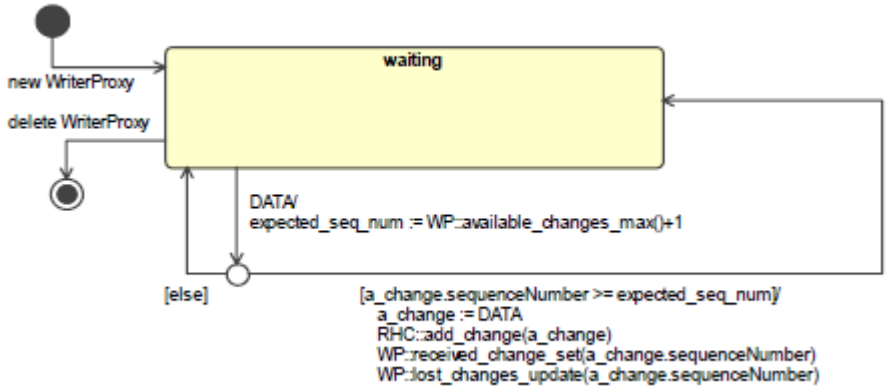


Figura 2-31. Comportamiento de un *Reader* con estado con WITH_KEY Best-Effort con respecto a cada *Writer* asociado

(OMG, 2014)

El listado de estados se encuentra descrito en la Tabla 2-13.

Tabla 2-13. 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 2-32 podremos observar el comportamiento de este tipo de lector.

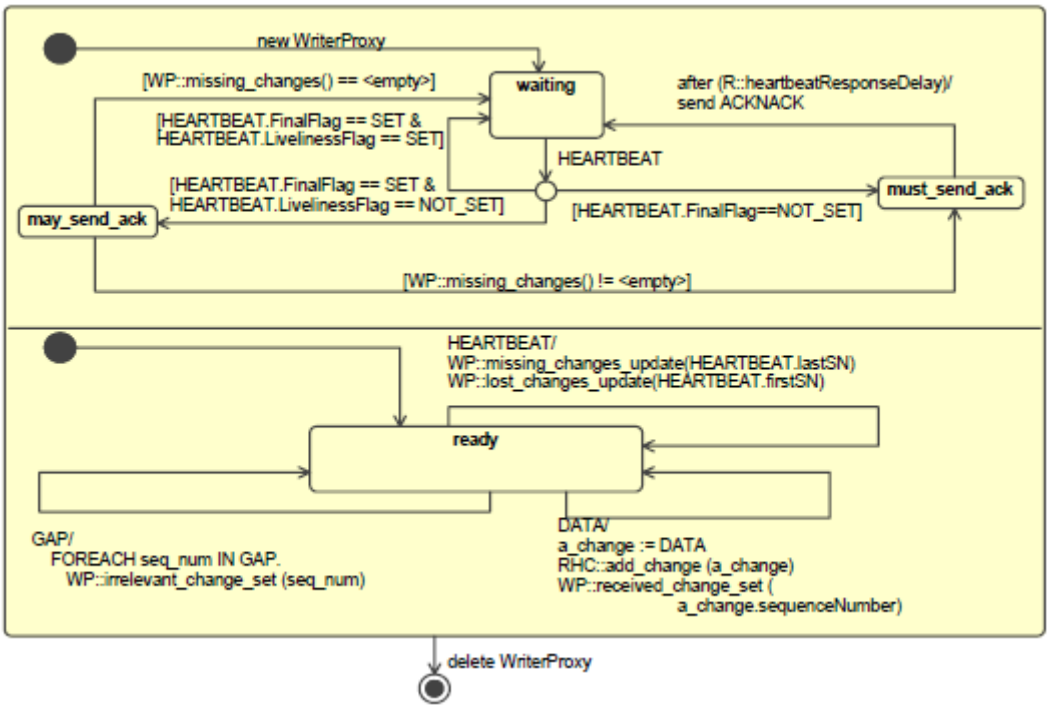


Figura 2-32. Comportamiento de un *Reader* con estado con WITH_KEY *Reliable* con respecto a cada *Writer* asociado

(OMG, 2014)

El listado de estados se encuentra descrito en la Tabla 2-14.

Tabla 2-14. 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* *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	*Final*

		representado por el *WriterProxy*.	

2.7.4. Módulo Descubrimiento

El módulo descubrimiento define el protocolo de descubrimiento RTPS. El propósito del protocolo de descubrimiento es permitir que cada *participante* RTPS descubra otros relevantes *participantes* y sus *endpoint*. Una vez que el *endpoint* ha sido descubierto, las implementaciones pueden configurar *endpoint* locales para establecer comunicación.

La especificación DDS se basa en el uso de un mecanismo de descubrimiento para establecer comunicación entre parejas de DataWriter y DataReader. Las implementaciones DDS automáticamente descubren la presencia de entidades remotas, cuando ellas se unen y dejan la red. La información de descubrimiento se hace accesible al usuario a través del *DDS built-in topics.*

El protocolo de descubrimiento RTPS definido en este módulo, proporciona el mecanismo de descubrimiento requerido para el DDS.

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.

Las implementaciones pueden optar por admitir varias PDP y EDP, posiblemente *vendor-specific*. Hasta dos participantes tienen al menos un PDP y EDP en común, ellos pueden intercambiar la información de descubrimiento requerido. A fin de 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 es proporcionar información sobre *endpoint* remotos descubiertos. Esta información es utilizada por un *participante* para configurar sus *endpoint* locales dependiendo de la aplicación efectiva del protocolo RTPS y no es parte de la especificación del protocolo de descubrimiento. Por ejemplo, para anteriores implementaciones, la información obtenida en el *endpoint* remoto permite a las implementaciones configurar:

- Los objetos RTPS *ReaderLocator* están asociados con cada RTPS *StatelessWriter*.
- Los objetos RTPS *ReaderProxy* asociados con cada RTPS *StatefulWriter*.
- Los objetos RTPS *WriterProxy* asociados con cada RTPS *StatefulReader*.

2.7.4.1. RTPS Built-in Discovery Endpoint

La especificación DDS especifica que el descubrimiento tiene lugar mediante "incorporados" DataReader y DataWriter DDS con predefinidos *topics* y QoS.

Hay cuatro predefinidos *built-in topics*: “DCPSParticipant”, “DCPSSubscription”, “DCPSPublication”, y “DCPSTopic”. El *DataType* asociado con este *topics* son también especificadas por la especificación del DDS y principalmente contiene valores de la entidad QoS.

Para cada uno de los *built-in Topic*, existe un correspondiente DDS *built-in DataWriter* y DDS *built-in DataReader*. Los *built-in DataWriter* se utilizan para anunciar la presencia y QoS del Participante local DDS y las Entidades DDS que contiene (*DataReaders*, *DataWriters* y topics) al resto de la red. Del mismo modo, los *built-in DataReaders* recogen este información de los participantes remotos, que es utilizada por la aplicación DDS para identificar entidades remotas correspondientes. Los *built-in DataReaders* actúan como DataReaders DDS regulares y también se puede acceder por el usuario a través del DDS API.

El enfoque adoptado por los Protocolos RTPS de Descubrimiento simples (SPDP[31] y SEDP[32]) es análogo al concepto *built-in Entity*. Los mapas RTPS en cada *built-in DDS DataWriter* o *DataReader* están asociados a un *built-in RTPS endpoint*. Estos *built-in endpoint* actúan como *Writer* y *Reader endpoint* y proporciona los medios para el intercambio de la información de descubrimiento requerida entre los participantes mediante el protocolo habitual RTPS definida en el módulo Behavior.

El SPDP, se ocupa de cómo los participantes se descubren entre sí , los mapas de DDS *built-in Entity* para el Topic "DCPSParticipant". El SEDP, especifica cómo intercambiar información de descubrimiento sobre los *Topic* locales, *DataWriters* y *DataReaders* , los mapas DDS *built-in Entity* para los *Topic* "DCPSSubscription", "DCPSPublication" y "DCPSTopic".

2.7.4.2. Simple Participant Discovery Protocol

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

Un participante puede soportar varios PDP[33] , pero para el propósito de interoperabilidad, todas las implementaciones deberían soportar al menos SPDP.

[31] SPDP, RTPS Simple Discovery Protocol
[32] SEDP, Simple Endpoint Discovery Protocol
[33] PDP, Participant Discovery Protocol

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

Por cada *participante*, el SPDP crea dos RTPS *built-in Endpoint*: el *SPDPbuiltinParticipantWriter* y el *SPDPbuiltinParticipantReader.*

El *SPDPbuiltinParticipantWriter* es un RTPS *best-effort StatelessWriter.* El *historyCache* del *SPDPbuiltinParticipantWriter* contiene un solo objeto-datos de tipo SPDPdiscoveredParticipantData. El valor de este objeto-dato se establece a partir de los atributos en el participante. Si los atributos cambian, el objeto-dato es reemplazado.

El *SPDPbuiltinParticipantWriter* envía periódicamente estos objetos-datos a una lista pre configurada de localizadores para anunciar la presencia del participante en la red. Esto se consigue llamando periódicamente *StatelessWriter :: unsent_changes_reset*, que hace que el *StatelessWriter* vuelva a enviar todos los cambios presentes en su *HistoryCache* a todos los localizadores. La tasa periódica a la que el *SPDPbuiltinParticipantWriter* envía los valores predeterminados al *SPDPdiscoveredParticipantData* un valor PSM especificado.

La lista pre configurada de localizadores puede incluir localizadores *unicast* y *multicast*. Los puertos son definidos por cada PSM. Estos localizadores simples representan posibles *participantes* remotos en la red, ningún *participante* necesita estar presente. Para el envío de *SPDPdiscoveredParticipantData* periódico, los *participantes* pueden unirse a la red en cualquier orden.

El *SPDPbuiltinParticipantReader* recibe los anuncios *SPDPdiscoveredParticipantData* desde los *participantes* remotos. La información contenida incluye que los EDP[34] soporten al *participante* remoto. El EDP se utiliza para el intercambio de información del *endpoint* con el *participante* remoto.

[34] EDP, Endpoint Discovery Protocol.

Las implementaciones pueden minimizar cualquier retrasos generados por el envío de un *SPDPdiscoveredParticipantData* adicional en respuesta a la recepción de estos objetos-datos de un participante previamente desconocido, pero este comportamiento es opcional. Las implementaciones pueden también permitir al usuario elegir si desea ampliar automáticamente la lista pre configurada de localizadores con nuevos localizadores desde participantes recién descubiertos. Esto permite que las listas de localización asimétricas. Estas dos últimas características son opcionales y no se requiere para el propósito de la interoperabilidad.

SPDPdiscoveredParticipantData

El *SPDPdiscoveredParticipantData* define los datos intercambiados como parte del SPDP.

En la Figura 2-33 se muestra el contenido del ***SPDPdiscoveredParticipantData.*** Como se muestra en la figura, el ***SPDPdiscoveredParticipantData*** se especializa el *ParticipantProxy* y por lo tanto incluye toda la información necesaria para configurar un *participante* descubierto.

El ***SPDPdiscoveredParticipantData*** también especializa al DDS- definido *DDS::ParticipantBuiltinTopicData* proporcionando la información que los correspondientes DDS *built-in DataReader* necesita.

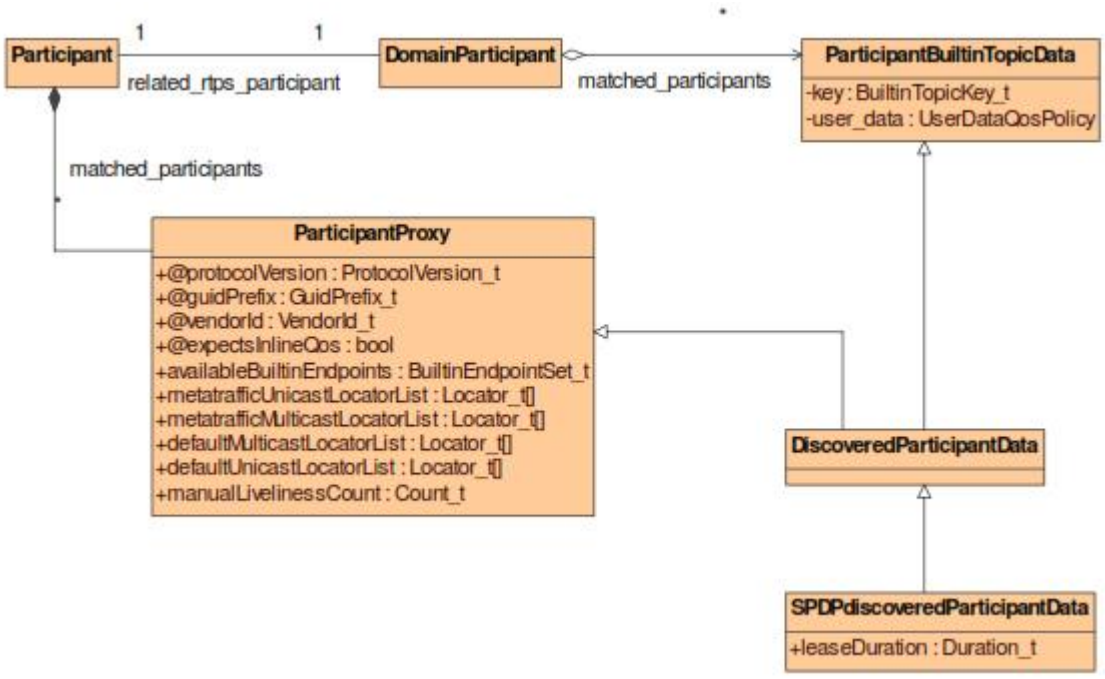


Figura 2-33. SPDPdiscoveredParticipantData.

Built-in Endpoint usado por el SPDP

La Figura 2-34 se muestra el *built-in Endpoint* introducido por el SPDP.

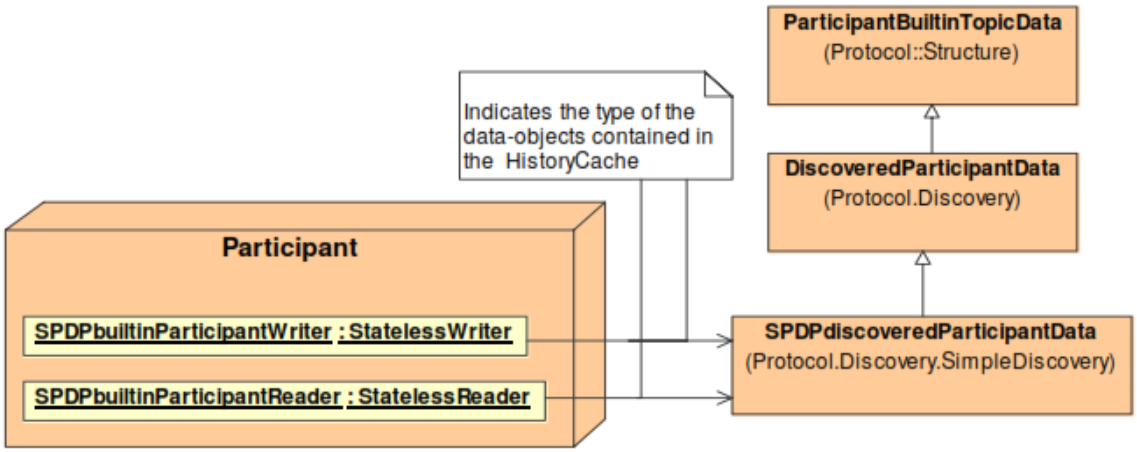


Figura 2-34. El built-in Endpoint usado por el SPDP

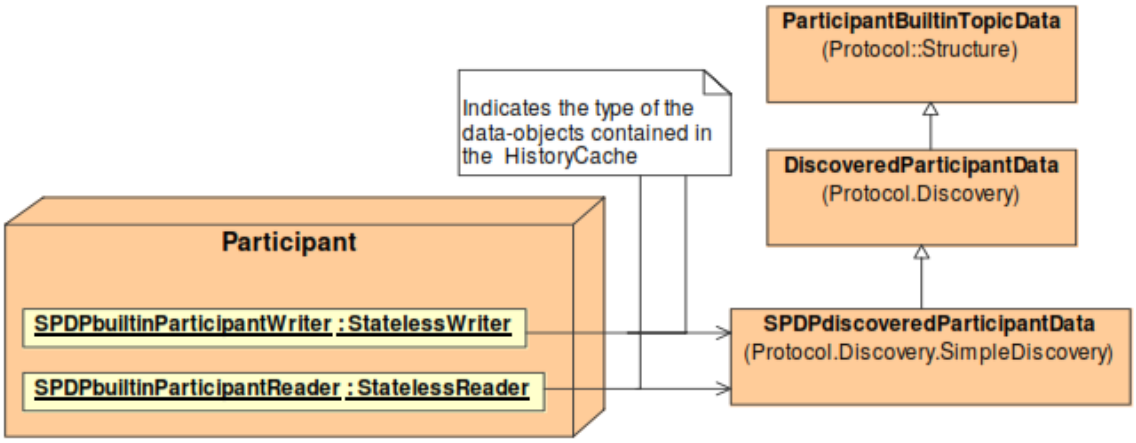


Figura 2-35. El built-in Endpoint usado por el SPDP.

El protocolo reserva los siguientes valores de la ***EntityId_t*** para el SPDP *built-in Endpoint.*

ENTITYID_SPDP_BUILTIN_PARTICIPANT_WRITER

ENTITYID_SPDP_BUILTIN_PARTICIPANT_READER

El *HistoryCache* de la *SPDPbuiltinParticipantReader* contiene información sobre todos los *participantes* descubiertos activos; la llave usada identifica cada objeto-dato correspondiente al GUID *participante.*

Cada información sobre un participante es recibida por el *SPDPbuiltinParticipantReader*, el SPDP examina la *HistoryCache* buscando una entrada con una clave que coincide con el GUID *participante*. Si una entrada con una clave coincidente no está allí, una nueva entrada se agrega tecleando el GUID del *participante*.

Periódicamente, el SPDP examina la *SPDPbuiltinParticipantReader,* la *HistoryCache* busca entradas obsoletas definidas como aquellos que no se han renovado por un período más largo que su *leaseDuration* especificado. Se eliminan las entradas obsoletas.

2.7.4.3. 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*.

Similar al SPDP, el Protocolo de descubrimiento simple *endpoint* utilizamos predefinidos *built-in Endpoint.* La utilización *built-in Endpoint* predefinidos significa que una vez que un *participante* conoce de la presencia de otro *participante*, que puede asumir la presencia de los *built-in Endpoint* puestos a su disposición por el *participante* remoto y establece la asociación con el *built-in Endpoint* localmente asociado.

El protocolo utilizado para la comunicación entre los *built-in Endpoint* es el mismo usado por la aplicación definida *Endpoint.* Por lo tanto, mediante la lectura del *built-in Reader Endpoint*, el protocolo máquina virtual puede descubrir la presencia QoS de las Entidades DDS que pertenecen a las *participantes* remotos. Del mismo modo, escribiendo el *Writer* incorpora criterios de valoración que un *participante* puede informar a los demás de la existencia y calidad de servicio de las entidades locales DDS.

Built-in Endpoint utilizado por el SEDP

Los mapas SEDP de los *built-in Entity* para el "DCPSSubscription", "DCPSPublication" y el Topic "DCPSTopic". Acorde a la especificación DDS, el *reliability* QoS para este *built-in Entity* se establece en "*reliable*". El SEDP, por lo tanto, los mapas de cada uno corresponden al *built-in DDS DataWriter* o *DataReader* en los correspondientes confiables *Writer* y *Reader* RTPS *Endpoint*.

Por ejemplo, como se muestra en la Figura 2-36, el DDS *built-in DataWriter* para el "DCPSPublication", y el Topic "DCPSTopic" puede asignar un *StatefulWriter RTPS* fiables y los DDS *DataReaders* a *StatefulReader* RTPS fiables. Las implementaciones reales no necesitan utilizar la referencia de estado. Para el propósito de la interoperabilidad, es suficiente que una implementación proporcione lo requerido para *built-in Endpoint* y comunicación

confiable que satisfaga a los requerimientos generales que se han nombrado en la sección anterior.

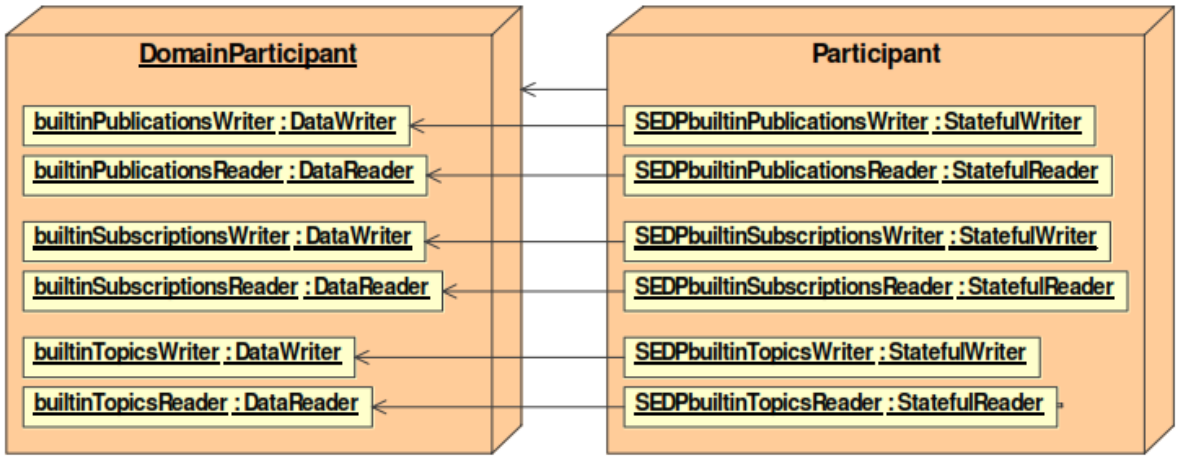


Figura 2-36. Ejemplo de asignación de los DDS built-in Entity correspondientes con RTPS built-in Endpoint.

El protocolo RTPS reserva los siguientes valores del *EntityId_t* por el *built-in Endpoint*:

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER

ENTITYID_SEDP_BUILTIN_TOPIC_WRITER

ENTITYID_SEDP_BUILTIN_TOPIC_READER

Built-in Endpoint requerido para el SEDP

Las implementaciones no son requeridas para proveer *built-in Endpoint*.

El Tema propagación es opcional. Por lo tanto, no se requiere para implementar el *SEDPbuiltinTopicsReader* y *SEDPbuiltinTopicsWriter built-in Endpoint* y con el propósito de la interoperabilidad, las implementaciones no deben confiar en su presencia en los *participantes* remotos.

En lo que respecta a los restantes *built-in Endpoint*, un Participante sólo está obligado a proporcionar *built-in Endpoint* requerido para hacer coincidir los *Endpoint* locales y remotos. Por ejemplo, si un *participante* DDS sólo contendrá *DataWriters* DDS, los únicos RTPS requeridos *built-in Endpoint* son las *SEDPbuiltinPublicationsWriter* y las *SEDPbuiltinSubscriptionsReader*. El *SEDPbuiltinPublicationsReader* y los *SEDPbuiltinSubscriptionsWriter built-in Endpoint* no sirven para nada en este caso.

Tipos de datos asociados con built-in Endpoint utilizado por el SEDP

Cada RTPS *Endpoint* tiene un *HistoryCache* tiene almacenados cambios al objeto-dato asociado con el *Endpoint*. Esto también es aplicado al RTPS *built-in Endpoint.* Por lo tanto, cada RTPS *built-in Endpoint* depende de algunos *DataType* que representa los contenidos lógicos de los datos escritos en su *HistoryCache.*

La Figura 2-37 define los *DataType DiscoveredWriterData, DiscoveredReaderData,* y *DiscoveredTopicData* asociado con el RTPS *built-in Endpoint* para el "DCPSPublication", "DCPSSubscription", y los Topic "DCPSTopic".

El *DataType* asociado con cada RTPS *built-in Endpoint* contiene toda la información especificada por DDS para el correspondiente *built-in DDS Entity.* Por esta razón, *DiscoveredReaderData* extiende el DDS definido *DDS::SubscriptionBuiltinTopicData, DiscoveredWriterData* extiende *DDS::PublicationBuiltinTopicData,* y *DiscoveredTopicData* extiende *DDS::TopicBuiltinTopicData.*

Además de los datos necesarios para la asociación *built-in DDS Entity,* los *DataType* "Descubiertos" también incluyen toda la información que pueda ser necesaria por una implementación del protocolo para configurar los RTPS *Endpoint.* Esta información está contenida en el RTPS *ReaderProxy* y en el *WriterProxy.*

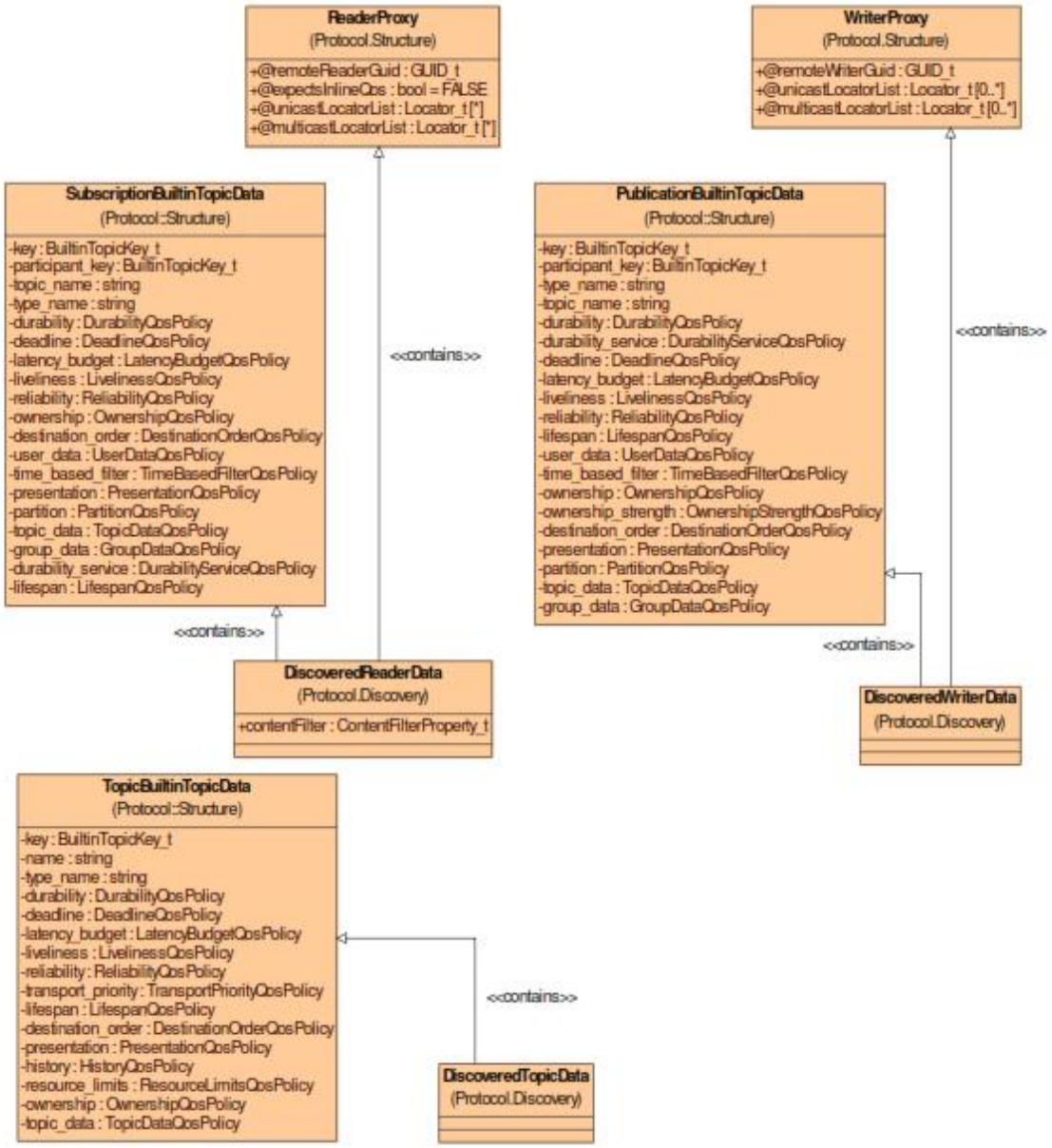


Figura 2-37. Los tipos de datos asociados con built-in Endpoint utilizado por el SEDP.

Una implementación del protocolo no necesita enviar necesariamente toda la información contenida en los *DataType.* Si cualquier información no está presente, la aplicación puede asumir los valores por defecto, tal como se define por el PSM. El PSM también define cómo se representa la información de descubrimiento en el cable. Los RTPS *built-in Endpoint* utilizado por el SEDP y sus *DataType* asociados se muestran en la XXX.

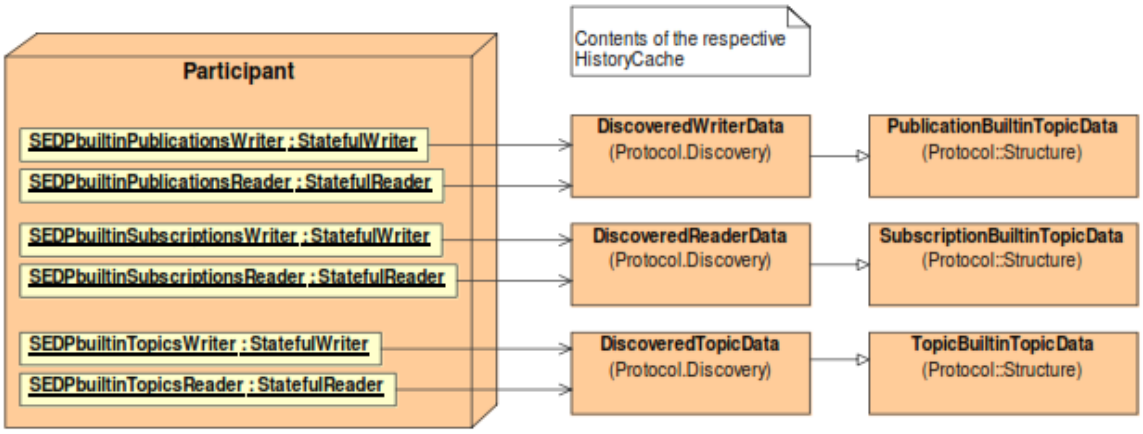


Figura 2-38. El built-in Endpoint y los DataType asociados con su respectivo HistoryCache.

Los contenidos de la *HistoryCache* por cada *built-in Endpoint* puede ser descrita en términos de los siguientes aspectos: *DataType, Cardinality, Data-Object insertion, Data-Object modification,* y *Data-Object deletion.*

- *DataType,* el tipo de dato almacenado en la caché. Esto se debe en parte definida por la especificación DDS.
- *Cardinality,* el número de diferentes datos de objetos (cada uno con una clave diferente) que potencialmente pueden ser almacenadas en la memoria caché.
- *Data-Object insertion,* las condiciones en las que se inserta un nuevo objeto de datos en la caché.
- *Data-Object modification,* las condiciones en las que se modifica el valor de un objeto de datos existentes.
- *Data-Object deletion,* las condiciones en las que se elimina un objeto de datos existentes de la caché.

2.7.4.4. Interacción con la máquina virtual RTPS

Para ilustrar aún más el SPDP y SEDP, describe cómo la información proporcionada por el SPDP se puede utilizar para configurar el SEDP *built-in Endpoint* en la máquina virtual RTPS.

Descubrimiento de un nuevo participante remoto

Usando el *SPDPbuiltinParticipantReader,* un *participante* local "local_participant" descubre la existencia de otro *participante* descrito por el *DiscoveredParticipantData* participant_data. El descubrimiento de *participantes* utiliza el SEDP.

El pseudo código a continuación configura los SEDP *built-in Endpoint* locales dentro del *local_participant* para comunicarse con el correspondiente SEDP *built-in Endpoint* en el Participante descubierto.

Tener en cuenta que la forma de los *Endpoint* están configurados dependiendo de la aplicación del protocolo. Para la referencia *stateful*, esta operación se realiza los siguientes pasos lógicos:

```
IF ( PUBLICATIONS_READER IS_IN participant_data.availableEndpoints ) THEN
guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER>;
writer = local_participant.SEDPbuiltinPublicationsWriter;
proxy = new ReaderProxy( guid,
participant_data.metatrafficUnicastLocatorList,
participant_data.metatrafficMulticastLocatorList);
writer.matched_reader_add(proxy);
ENDIF
IF ( PUBLICATIONS_WRITER IS_IN participant_data.availableEndpoints ) THEN
guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER>;
reader = local_participant.SEDPbuiltinPublicationsReader;
proxy = new WriterProxy( guid,
participant_data.metatrafficUnicastLocatorList,
```

```
        participant_data.metatrafficMulticastLocatorList);
        reader.matched_writer_add(proxy);
        ENDIF
        IF ( SUBSCRIPTIONS_READER IS_IN participant_data.availableEndpoints ) THEN
        guid = <participant_data.guidPrefix,
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER>;
        writer = local_participant.SEDPbuiltinSubscriptionsWriter;
        proxy = new ReaderProxy( guid,
        participant_data.metatrafficUnicastLocatorList,
        participant_data.metatrafficMulticastLocatorList);
        writer.matched_reader_add(proxy);
        ENDIF
        IF ( SUBSCRIPTIONS_WRITER IS_IN participant_data.availableEndpoints ) THEN
        guid = <participant_data.guidPrefix,
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER>;
        reader = local_participant.SEDPbuiltinSubscriptionsReader;
        proxy = new WriterProxy( guid,
        participant_data.metatrafficUnicastLocatorList,
        participant_data.metatrafficMulticastLocatorList);
        reader.matched_writer_add(proxy);
        ENDIF
        IF ( TOPICS_READER IS_IN participant_data.availableEndpoints ) THEN
        guid = <participant_data.guidPrefix,
ENTITYID_SEDP_BUILTIN_TOPICS_READER>;
        writer = local_participant.SEDPbuiltinTopicsWriter;
```

proxy = new ReaderProxy(guid,

participant_data.metatrafficUnicastLocatorList,

participant_data.metatrafficMulticastLocatorList);

writer.matched_reader_add(proxy);

ENDIF

IF (TOPICS_WRITER IS_IN participant_data.availableEndpoints) THEN

guid = <participant_data.guidPrefix, ENTITYID_SEDP_BUILTIN_TOPICS_WRITER>;

reader = local_participant.SEDPbuiltinTopicsReader;

proxy = new WriterProxy(guid,

participant_data.metatrafficUnicastLocatorList,

participant_data.metatrafficMulticastLocatorList);

reader.matched_writer_add(proxy);

ENDIF

Eliminación de un participante previamente descubierto

Basado en leaseDuration del Participante remoto, un participante local "local_participant" llega a la conclusión de que un participante previamente descubierto con GUID_t participant_guid ya no está presente. El Participante "local_participant" debe reconfigurar los *Endpoint* locales que se comunican con los *Endpoint* en el participante identificado por el GUID_t participant_guid.

Para la implementación de referencia de estado, esta operación lleva a cabo los siguientes pasos lógicos:

guid = <participant_guid.guidPrefix, ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER>;

writer = local_participant.SEDPbuiltinPublicationsWriter;

```
        proxy = writer.matched_reader_lookup(guid);
        writer.matched_reader_remove(proxy);
        guid                    =                    <participant_guid.guidPrefix,
ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER>;
        reader = local_participant.SEDPbuiltinPublicationsReader;
        proxy = reader.matched_writer_lookup(guid);
        reader.matched_writer_remove(proxy);
        guid                    =                    <participant_guid.guidPrefix,
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER>;
        writer = local_participant.SEDPbuiltinSubscriptionsWriter;
        proxy = writer.matched_reader_lookup(guid);
        writer.matched_reader_remove(proxy);
        guid                    =                    <participant_guid.guidPrefix,
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER>;
        reader = local_participant.SEDPbuiltinSubscriptionsReader;
        proxy = reader.matched_writer_lookup(guid);
        reader.matched_writer_remove(proxy);
        guid                    =                    <participant_guid.guidPrefix,
ENTITYID_SEDP_BUILTIN_TOPICS_READER>;
        writer = local_participant.SEDPbuiltinTopicsWriter;
        proxy = writer.matched_reader_lookup(guid);
        writer.matched_reader_remove(proxy);
        guid                    =                    <participant_guid.guidPrefix,
ENTITYID_SEDP_BUILTIN_TOPICS_WRITER>;
        reader = local_participant.SEDPbuiltinTopicsReader;
```

```
proxy = reader.matched_writer_lookup(guid);
reader.matched_writer_remove(proxy);
```

2.7.4.5. 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 apoyen al menos el SPDP y SEDP. Se espera que con el tiempo, una colección de protocolos de descubrimiento de interoperabilidad se desarrollarán para atender las necesidades de implementación específicos.

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 soportes *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. En

ese caso, no hay necesidad de crear la SEDP *built-in Endpoint*, o si ya existen, sin necesidad de configurarlos para que coincida con el nuevo *participante* remoto. Este enfoque permite a un *vendor* personalizar el procedimiento de déficit excesivo, si se desea, sin comprometer la interoperabilidad.

3. CAPÍTULO III

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

4. CAPÍTULO IV

PRUEBAS

5. CAPÍTULO V

CONCLUSIONES Y RECOMENDACIONES

5.1. Conclusiones

5.2. Recomendaciones

REFERENCIA BIBLIOGRÁFICA

WIKIPEDIA. (2 de Mayo de 2014). Recuperado el 9 de Marzo de 2015, de Priority Ceiling Protocol: http://it.wikipedia.org/wiki/Priority_ceiling_protocol

Wikipedia. (09 de Marzo de 2015). Recuperado el 09 de Marzo de 2015, de Earliest deadline first scheduling: http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling

Aldea, M., Bernat, G., Burns, A., Dobrin, R., Drake, J. M., Fohler, G., y otros. (2006). FSF: A real-time scheduling architecture framework. *n Proceedings of the IEEE Real Time Technology and Applications Symposium.*, 113-124.

Amoretti, M., Caselli, S., & Reggiani, M. (2006). Designing distributed, component-based systems for industrial robotic applications. (L. K. (Ed.), Ed.) *In Industrial Robotics: Programming, Simulation and Applications*.

Bard, J., & Kovarik, V. J. (2007). Software Defined Radio: The Software Communications Architecture. (Wiley-Blackwell, Ed.)

Basanta-Val, P., García-Valls, M., & Estévez-Ayres, I. (2010). An architecture for distributed real-time Java based on RMI and RTSJ. *In Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation* (págs. 1-8). IEEE.

Blair, G. S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., y otros. (2001). The design and implementation of open ORB 2. *In IEEE Distributed Systems Online, 2*.

Bollella, G., & Gosling, J. (2000). *The real-time specification for Java.* IEEE Computer.

Campos, J. L., Gutiérrez, J. J., & Harbour, M. G. (2006). Interchangeable scheduling policies in real-time middleware for distribution. (I. P.-E. Technologies, Ed.) *Lecture Notes in Computer Science, 4006*, 227-240.

Corsaro, A. (s.f.). *Advanced DDS Tutorial.* (OpenSPlice, Ed.) Obtenido de http://www.prismTech.com/dds-community

Davis, R. I., & Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *43,35*, págs. 43:4, 35:1-35:44. ACM Computing Surveys.

Freeman, E., Hupfer, S., & Arnold, K. (1999). JavaSpaces: Principles, Patterns, and Practice. Addison-Wesley, Reading, MA.

FRESCOR. (2006). *Framework for Real-Time Embedded Systems Based on COntRacts. Project Web page. Retrieved September 2013* . Recuperado el 10 de Marzo de 2015, de http://www.frescor.org

Gillen, M., Loyall, J., Haigh, K. Z., Walsh, R., Partridge, C., Lauer, G., y otros. (2012). Information dissemination in disadvantaged wireless communications using a data dissemination service and content data network. *In Proceedings of the SPIE Conference on Defense Transformation and Net-Centric Systems*, *8405.*

Gokhale, A., Balasubramanian, K., Krishna, A. S., Balasubramanian, J., Edwards, G., Deng, G., y otros. (2008). Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming, 73*, 39-58.

Grelck, C., Julju, J., & Penczek, F. (2012). Distributed S-Net: Cluster and grid computing without the hassle. *In Proceedings of the 12th IEEE /ACM International Symposium on Cluster, Cloud and Grid Computing(CCGrid)*, (págs. 410-418).

IEEE. (2006). *The Institute of Electrical and Electronics Engineers STD 802.1Q. 2006. Virtual bridged local area networks. Annex G.* . Obtenido de http://www.ieee802.org/1/pages/802.1Q.html

ISO/IEC. (2012). Ada 2012 Reference Manual. Language and Standard Libraries—International Standard. *ISO/IEC, 8652*.

ISO/IEC, Taft, S. T., Duff, R. A., Brukardt, R., Ploedereder, E., & Leroy, P. (2006). 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, 4348*.

Kermarrec, Y. (1999). CORBA vs. Ada 95 DSA: A programmer's view. *XIX*, 39-46.

Kim, K. H. (2000). Object-oriented real-time distributed programming and support middleware. *In Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS)*, 10-20.

Klefstad, R., Schmidt, D. C., & O'Ryan, C. (2002). Towards highly configurable real-time object request brokers. *In Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, (págs. 437-447).

Liu, C. L., & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environments. *Journal of the ACM*, *20*, págs. 46-61.

Neumeyer, L., Robbins, B., Nair, A., & Kesari, A. (2010). S4: Distributed stream computing platform. *In Proceedings of the IEEE International Conference on Data Mining (ICDM)* (págs. 170-177). IEEE.

OMG. (2005). Realtime Corba Specification. v1.2.

OMG. (2007). Data Distribution Service for Real-Time Systems. v1.2.

OMG. (2009). *The Real-Time Publish-Subscribe Wire Protocol. DDS interoperability wire protocol specification. v2.1.* . Recuperado el 6 de Marzo de 2015, de http://www.omg.org/spec/DDSI/2.1/

OMG. (2011). Corba Core Specification. v3.2.

OMG. (2012). *Extensible and Dynamic Topic Types for DDS. v1.0.* . Obtenido de http://www.omg.org/spec/DDS-XTypes/1.0/

Pardo-Castellote, G. (s.f.). *OMG Data-Distribution Service: Architectural Overview.* Real-Time Innovations, Inc.

Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Henia, R., y otros. (2007). Influence of different system abstractions on the performance analysis of distributed real-time systems. *In Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07)* (págs. 193-202). New York: ACM.

Pérez, H., & Gutiérrez, J. J. (2012). On the schedulability of a data-centric real-time distribution middleware. *Computer Standards and Interfaces 34.*, 203-211.

Pérez, H., & Gutiérrez, J. J. (Marzo de 2014). A survey on standards for real-time distribution middleware. *ACM Computing Surveys, 46*(49), 39.

Pérez, H., Gutiérrez, J. J., Sangorrín, D., & Harbour, M. (2008). Real-time distribution middleware from the Ada perspective. In Proceedings of the 13th Ada-Europe International Conference on Reliable Software Technologies. En F. Kordon, & T. Vardanega (Ed.), *Lecture Notes in Computer Science. 5026*, págs. 268-281. Springer.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., y otros. (2009). MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Software Engineering for Self-Adaptive Systems. *Lecture Notes in Computer Science, 5525*, 164-182.

Ryll, M., & Ratchev, S. (2008). Application of the data distribution service for flexible manufacturing automation. . *International Journal of Aerospace and Mechanical Engineering* , 193-200.

Schimidt, D. C., Corsaro, A., & Hag, H. V. (2008). Addressing the challenges of tactical information management in net-centric systems with DDS. En *Journal of Defense Software Engineering* (págs. 24-29).

Sha, L., Rjkumar, R., & Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39. 9*, págs. 1175-1185. IEEE.

Sun Microsystems. (2000). *JSR-50: Distributed Real-Time Specification.*

Sun Microsystems. (2002). *JavaTM Message Service Specification. v1.1.*

Sun Microsystems. (2004). *Java Remote Method Invocation (RMI).*

Sun Microsystems. (2012). *Distributed Real-Time Specification (Early draft).* Recuperado el 06 de Marzo de 2015, de http://jcp.org/en/egc/download/drtsj.pdf?id=50&fileId=5028

Tejera, D., Alonso, A., & de Miguel, M. A. (2007). 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).*, 113-120.

Twin Oaks Computing, Inc. (Diciembre de 2011). *Interoperable DDS Strategies.* Recuperado el 17 de Marzo de 2015, de http://www.twinoakscomputing.com

Vergnaud, T., Hugues, J., Kordon, F., & Pautet, L. (4 de Mayo de 2004). PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. (I. P. Ada-Europe, Ed.) *Lecture Notes in Computer Science, 3063*, 106-119.

WIKIPEDIA. (8 de Marzo de 2013). *Polling*. Recuperado el 11 de Marzo de 2015, de http://es.wikipedia.org/wiki/Polling

ANEXOS