Capítulo 3: Estado del arte de los middleware orientados a mensajería

3.1 Introducción

Middleware es un término acuñado por Lewandowski en 1998 [40] que representa una capa de software ubicada entre la capa más alta (usuarios y aplicaciones), y la más baja (sistemas operativos y mecanismos de comunicación básicos).

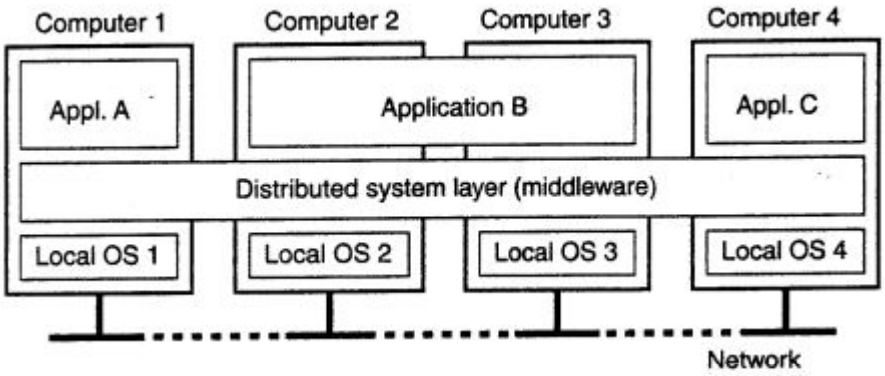

Figura 3.1. Concepto de middleware

Provee un mismo interfaz para todas las aplicaciones, y por tanto de un mismo medio de comunicación entre diferentes aplicaciones. Al mismo tiempo también oculta las diferencias en hardware, lenguajes de programación, plataformas de comunicación, y sistemas operativos, facilitando el desarrollo de sistemas complejos con diferentes tecnologías y arquitecturas (Figura 3.1).

En general, una capa middleware debe ofrecer las siguientes funcionalidades dentro de un sistema distribuido:

- Servicio de descubrimiento de la información: es requerido para la localización de recursos, mensajes y de otros servicios dentro del sistema.
- Seguridad: es necesario proveer un marco de seguridad en la comunicación entre procesos locales y remotos.
- Tratamiento del tiempo: proveen un formato universal para la representación horaria en plataformas diferentes ejecutándose en países y zonas horarias distintas. Este servicio es crítico para el mantenimiento de registros y tareas de sincronización entre procesos.
- Mecanismos de transacciones: estos servicios proveen transacciones semánticas que garantincen la integridad de los datos, siendo fundamentales para asegurar las actualizaciones de una o más bases de datos.

Los middleware están en continua evolución y mejora, extendiendo su rango de influencia de forma paulatina. Actualmente los sistemas middleware se encuentran ampliamente extendidos en entornos militares, tele-vigilancia, sistemas aeronáuticos y aeroespaciales, entre otros. En un futuro no es descartable que lleguen a utilizarse en aplicaciones cotidianas como navegadores o correo electrónico.

En cuanto a los tipos de middleware existentes, algunos autores han realizado en los últimos

años diferentes clasificaciones según su propio criterio. Por ejemplo, Judith Hurwitz distinguió 5 tipos [41]: los middleware RPC (Llamada a Procedimiento Remoto), los middleware orientados a mensajes, los middleware orientados a objetos, los middleware orientados a consulta SQL (Lenguaje Estructurado de Consultas) y los middleware embebidos. En nuestro caso, y por cercanía a la especificación DDS, únicamente trataremos los orientados a mensajes (MOM).

Los middleware orientados a mensajes están diseñados para el intercambio de mensajes entre procesos de forma asíncrona. De esta forma, las aplicaciones únicamente se encargan de "colocar" y "sacar" mensajes de las colas, no se conectan directamente entre ellas. No existen los términos de cliente y servidor, por lo que se pierde el concepto de petición de servicio, y cualquier entidad puede participar en el intercambio de información en cualquier momento, por lo que no necesariamente se requiere respuesta, y por tanto existe un desacoplamiento inherente en la comunicación.

Dentro de este tipo de middleware existen dos tipos de paradigmas de comunicación:

Punto a punto: cuenta con solo dos nodos en la comunicación, uno que envía el mensaje, y otro que lo recibe. Este modelo asegura la llegada del mensaje, ya que si el receptor no está disponible para aceptar el mensaje o atenderlo, se le remite igualmente, enviándose a una cola para que luego pueda ser recibido cuando se conecte.

Publicador/suscriptor: en este tipo de paradigma pueden existir diferentes nodos de comunicación que pueden ejercer los papeles de publicador, suscriptor, o ambos al mismo tiempo. La información es generada o consumida en un mensaje, evento o topic (dependiendo de la tecnología utilizada) que sirve como nexo de unión entre publicadores y suscriptores. Este modelo de comunicación ofrece una potente abstracción para multidifusión y comunicación en grupo. Los publicadores pueden difundir una determinada información a un grupo de suscriptores que están a la escucha de esta difusión de mensajes.

Dentro del paradigma publicador/suscriptor podemos encontrarnos con 4 modelos arquitectónicos de comunicación [42]:

- **Centralizada con intermediarios:** donde un servidor central sirve como punto de enlace entre todas las entidades y todo el tráfico pasa por él (Figura 3.2). Este modelo es implementado por JMS (Servicio de Mensajería de Java) y el Servicio de Notificaciones de CORBA (Arquitectura Común de Agente de Petición de Objetos).

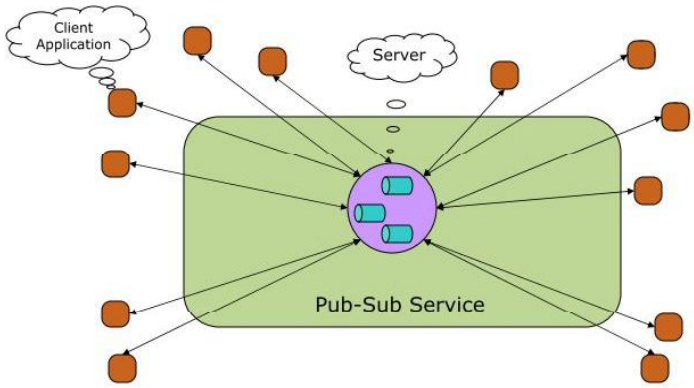

Figura 3.2. Arquitectura centralizada con intermediarios

- **Centralizada multi-intermediarios:** cada cola o topic se encuentra en servidores distintos, interconectados entre ellos (Figura 3.3). Este modelo es implementado por JMS, el Servicio de Notificaciones de CORBA y AMQP (Protocolo Avanzado de Espera de Mensajes).

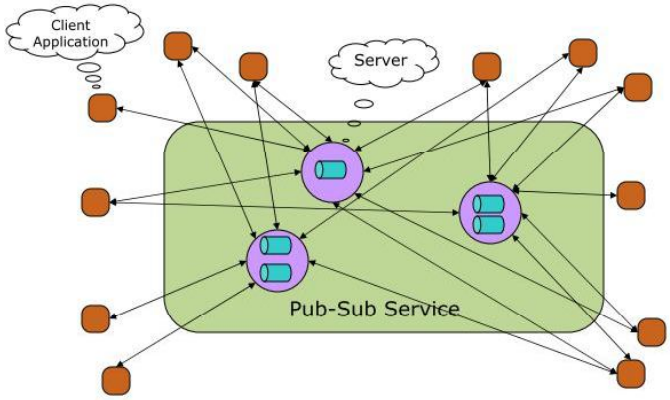

Figura 3.3. Arquitectura centralizada multi-intermediarios

- **Descentralizada multi-intermediarios:** donde el servicio publicador/suscriptor distribuye los mensajes internamente entre Puntos de Acceso (PS) (Figura 3.4). Internamente este servicio puede usar peer-to-peer, hub-and-spoke, multicast, entre otros métodos. Se utiliza en IBM WebSphere usando la funcionalidad *ClientConnection*.

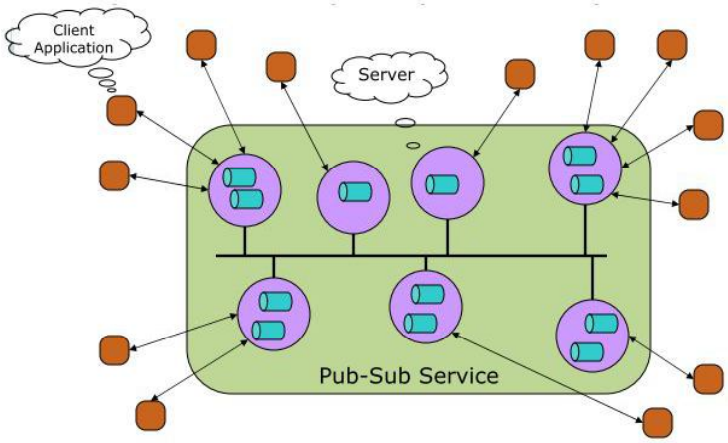

Figura 3.4. Arquitectura descentralizada con intermediarios

- **Descentralizada sin intermediarios:** donde no existen servidores, el proceso de cola ocurre a nivel local, y los clientes se comunican peer-to-peer (Figura 3.5). Es el tipo de arquitectura que se utiliza en DDS y en IBM WebSphere usando *BindingConnection*.

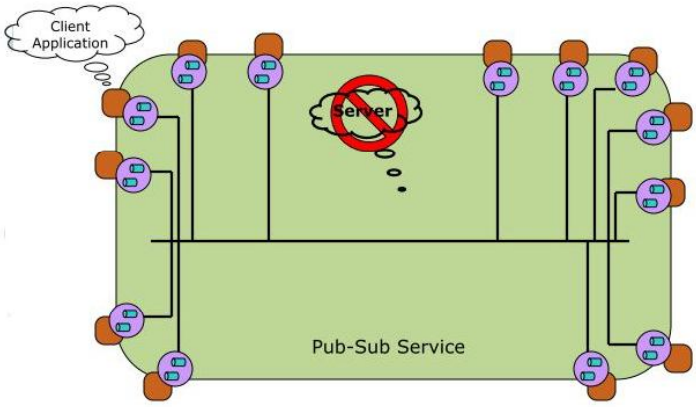

Figura 3.5. Arquitectura descentralizada sin intermediarios

Las ventajas de una arquitectura centralizada es la sencillez de implementación ante todo, aunque surgen varios tipos de inconvenientes entre los que destacan la poca escalabilidad, los cuellos de botella que surgen, la pobre predictibilidad (fundamental para trabajar en tiempo real), puntos "negros" de fallo, entre otros.

En una arquitectura descentralizada, y más concretamente cuando no existen intermediarios o *brokers*, existe una mejora en la latencia, predictibilidad del conjunto, ausencia de puntos de rupturas, y un aumento de la escalabilidad, permitiendo la inclusión de nuevos elementos sin que afecte al rendimiento del sistema. Entre las desventajas que ofrece esta arquitectura destacan su complejidad de implementación, y la necesidad de emplear un protocolo de comunicaciones no orientado a conexión para evitar una ingente cantidad de establecimientos de conexiones entre nodos.

3.2 Tecnologías y normas en middleware orientados a mensajerías

- **AMQP**

AMPQ (Protocolo Avanzado de Espera de Mensajes) [43] es un protocolo estándar abierto para la comunicación mediante un middleware orientado a mensajería, donde su objetivo principal es alcanzar la interoperatividad entre diferentes servicios (particularmente se diseñó inicialmente para servicios financieros).

Se considera el estándar de facto para middleware orientado a mensajería, y surgió en 2004 de la mano de empresas como OpenAMQ, Rabiit MQ Apache Qpid o Red Hat Enterprise MRG. Actualmente la especificación se encuentra en su revisión 0.9.1, aunque un esbozo de la 1.0 ya ha sido liberado.

A diferencia de otras tecnologías, AMQP no solamente define un API (Interfaz de Programación de Aplicaciones), sino también es un protocolo a nivel de cable, es decir, define el formato de los datos que son enviados a través de la red como un flujo de octetos. Por tanto, cualquier aplicación puede crear e interpretar mensajes conforme a este formato de datos, independientemente del lenguaje que se utilice.

Proporciona un *Espacio de Colas Compartido*, accesible para todas las aplicaciones

interesadas en el intercambio de mensajes. El encaminamiento de los mensajes es llevado a cabo por los *brokers* o intermediarios, que redirigen los datos a su destino basándose en una *clave única*, que es un identificador único asociado a cada mensaje. Este mensaje tiene una estructura fija con unas propiedades opcionales relacionadas con la prioridad de los datos, el tiempo de vida de los mismos, el modo de entrega, o la perdurabilidad. Si la aplicación suscriptora no puede procesar los mensajes entrantes, el bróker los almacena en una cola.

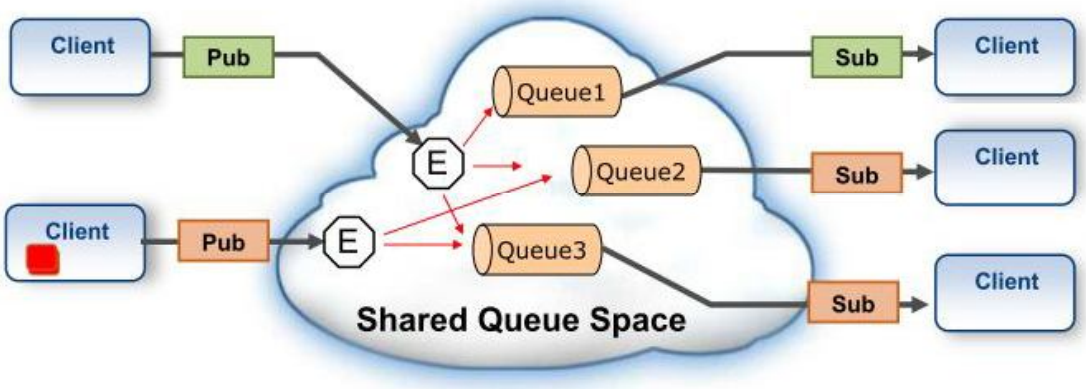

Figura 3.6. Tecnología AMQP

Tal y como se muestra en la Figura 3.6, los consumidores de información son los encargados de suscribirse a estas colas recibiendo los mensajes almacenados en la misma (mediante un mecanismo de *push*). Alternativamente, los publicadores pueden hacer saltar activamente mensajes de la cola como crean conveniente (mecanismo de *pull*). La cola garantiza que los mensajes son entregados en el mismo orden en el que llegaron, siguiendo por tanto el mecanismo FIFO. Además de esto, las colas incluyen una serie de características asociadas a la persistencia de los mensajes, a su exclusividad, autoborrado y otras propiedades.

- **MSMQ**

MSMQ (Cola de Mensajes de Microsoft) [44] es una tecnología propietaria de Microsoft que permite que aplicaciones que están ejecutándose en diferentes lugares puedan comunicarse a través de redes heterogéneas y sistemas que pueden estar temporalmente inactivos. Las aplicaciones envían mensajes a las colas para que puedan ser leídos por otras aplicaciones, tal y como se aprecia en la Figura 3.7.

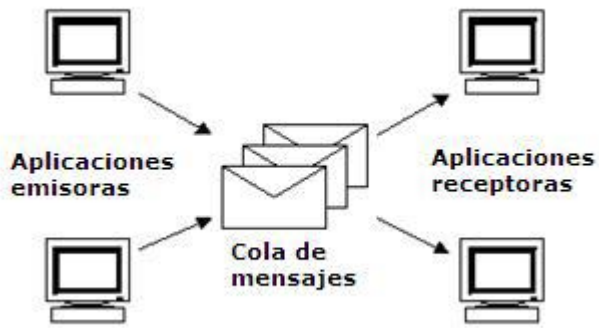

Figura 3.7. Tecnología MSMQ

Algunas características importantes de MSMQ son las siguientes:

- Garantiza la entrega de los mensajes: las aplicaciones que envían mensajes no necesitan estar conectadas en la red al mismo tiempo. MSMQ garantiza que los mensajes serán entregados tan pronto como las conexiones de red sean establecidas y las aplicaciones soliciten los mensajes.
- Comunicación asíncrona. Una vez que una aplicación cliente envía un mensaje, MSMQ permite al cliente realizar otras tareas sin tener que esperar la respuesta de la aplicación receptora.
- Soporte para transacciones: íntimamente integrado con MTS (Servidor de Transacciones de Microsoft). Cuando MTS está presente, sus servicios participarán automáticamente en una transacción MSMQ.
- Entrega ordenada: MSMQ garantiza que cada mensaje será entregado sólo una vez y en el orden en el que fueron enviados.
- Servicios de encaminamiento de mensajes: proporcionan al cliente la habilidad de enviar mensajes a su destino utilizando el camino más corto. El administrador sólo necesita definir el coste de la ruta, y MSMQ automáticamente realizará el resto del trabajo.
- Configuración dinámica.
- Alta escalabilidad.
- Seguridad mediante certificados.

Para acceder a una cola MSMQ mediante .NET existe una API llamada System.Messaging, diseñada para soportar otros sistemas basados en colas. Proporciona clases que permiten conectar, supervisar y administrar las colas de mensajes en la red, así como enviar, recibir o leer mensajes.

- **JMS**

JMS (Servicio de Mensajería de Java) [45] es una API de Java diseñada por Sun y otras compañías en 1998, y cuya última versión (2.1) data del 2002. Permite a las aplicaciones crear, enviar, recibir y leer mensajes, definiendo un conjunto de interfaces y su semántica asociada, posibilitando a los programas escritos en Java comunicarse con otros mediante mensajes.

Minimiza el conjunto de conceptos que un programador debe aprender para producir mensajes, proveyendo suficientes características para desarrollar aplicaciones sofisticadas basadas en el uso de mensajes. También maximiza la portabilidad de aplicaciones JMS a través de proveedores JMS dentro del mismo dominio.

En general, una aplicación JMS está compuesta de las siguientes partes:

- Proveedor JMS: es un sistema de mensajería que implementa las interfaces JMS y provee características administrativas y de control.
- Clientes JMS: son los programas o componentes escritos en Java, que producen y consumen los mensajes.
- Mensajes: son los objetos que comunican información entre clientes JMS.
- Objetos administrados: son objetos JMS preconfigurados creados por un administrador para el uso de clientes. Existen dos tipos: destinos y factorías de conexión.
- Clientes nativos: son programas que hacen uso de una API distinta a la API de JMS para el envío o recepción de mensajes.

`

JMS puede trabajar de las dos siguientes maneras:

- Dominio de mensajes punto a punto: un productor o aplicación es construido alrededor del concepto de mensajes, colas, emisores y receptores. Cada mensaje es direccionado a una cola específica, y los clientes extraen los mensajes de los mismos. Las colas retienen todos los mensajes que son enviados hasta que son consumidos o hasta que el tiempo de expiración termine.

- Dominio de mensajes publicador/suscriptor: en una aplicación productora de información, los clientes direccionan los mensajes a un topic. Los publicadores y suscriptores generalmente son anónimos y pueden publicar y suscribirse a un contenido jerárquico. El sistema se encarga de distribuir los mensajes que se publican y hacérselos llegar a los distintos suscriptores. Los topics retienen los mensajes solo el tiempo necesario para distribuirlos a los actuales suscriptores.

• Servicio de Notificaciones de CORBA

El Servicio de Notificaciones de CORBA [46] es una extensión del Servicio de Eventos de CORBA, por lo que hereda todas sus características. Más concretamente define los interfaces para la comunicación de notificaciones a través de un canal de eventos donde participan productores y consumidores de información mediante dos mecanismos de propagación: *push* y *pull*.

Una de las desventajas que presenta el Servicio de Eventos de CORBA es que no ofrece soporte para el filtrado de eventos o para especificar requisitos de reparto. Sin el uso de filtros, todos los consumidores conectados a un canal tendrán que recibir las mismas notificaciones que cualquier otro. Y sin la posibilidad de especificar requisitos de reparto, todas las notificaciones que se envíen a través de este canal tendrán las garantías de reparto incluidas en la implementación.

Con el objetivo de paliar este contratiempo, el Servicio de Notificaciones de CORBA incluye filtros que especifican qué eventos pueden estar interesados la aplicación. Los filtros pueden adherirse a cada proxy o intermediario en la comunicación, y de esta forma se dirigirá las notificaciones a los consumidores de eventos según las restricciones especificadas en los filtros, como se aprecia en la Figura 3.8.

Figura 3.8. Servicio de Notificaciones de CORBA

- **DDS**

DDS (Servicio de Distribución de Datos) [47] es una especificación adoptada por la OMG para el intercambio de datos en sistemas distribuidos en tiempo real, basándose en el paradigma de comunicación publicador/suscriptor. La OMG es un consorcio internacional sin ánimo de lucro formado por organizaciones que desarrollan estándares de tecnologías orientadas a objetos que se encuentran ampliamente extendidas en la actualidad dentro de la industria informática. DDS es un ejemplo de este tipo de tecnologías, y otros estándares que también han sido promovidos por la OMG son CORBA, UML, o MDA, entre otros.

DDS fue creado en el año 2001 por la empresa americana Real-Time Innovations y la francesa Thales Group, aunque no fue adoptado por la OMG hasta el 2004, convirtiéndose en la primera especificación abierta internacional basada en el modelo de comunicaciones publicación/suscripción. La última revisión disponible hasta la fecha es la 1.2, lanzada en 2007.

Actualmente la OMG se encuentra ultimando la versión 1.3, donde se trabaja en la inclusión a la especificación de un nuevo tipo de topics denominados X-Topics. Dichos topics pueden ser definidos en tiempo de ejecución y declarados no solo mediante el lenguaje de especificación IDL (Lenguaje de Definición de Interfaces) como se venía realizando hasta ahora, sino también utilizando XML (Lenguaje de Marcas Generalizado) o XSD (Lenguaje de Esquema XML). Otros temas candentes que también se están concretando son la integración de DDS con diferentes tecnologías web, tales como HTTP, clientes SOAP (Protocolo de Acceso de Objeto Simple) o REST (Transferencia de Estado Representacional), así como la definición de un marco de seguridad.

Básicamente, el objetivo de la especificación DDS es facilitar el intercambio de datos en sistemas distribuidos en tiempo real, proporcionando una gran variedad de políticas de servicios que permitan configurar la comunicación entre extremos y alcanzar los requisitos demandados por la aplicación. La riqueza de las políticas de QoS (Calidad de Servicio) posibilita que DDS soporte requisitos de tiempo real en un espectro muy amplio, desde necesidades de tiempo real estricto hasta sistemas que no requieren ningún tipo de requisito de este tipo.

La comunicación en DDS conecta por un lado a los productores de datos o publicadores, y por otro a los consumidores de datos o suscriptores, desacoplándolos en tiempo, espacio y flujo [48]. De esta manera, la aplicación no necesita conocer el origen de la información, ni cuándo ha sido producida, ni cómo llega hasta ella, sino únicamente basta con precisar qué tipo de información es la que se quiere, y unas condiciones de comunicación que vienen dadas unas políticas de calidad de servicio.

El contendedor de la información, y nexo de unión entre publicadores y suscriptores, es el Topic, que lleva asociado un determinado nombre y un tipo de dato concreto, descrito por el lenguaje de especificación IDL de forma similar a como se realiza en CORBA.

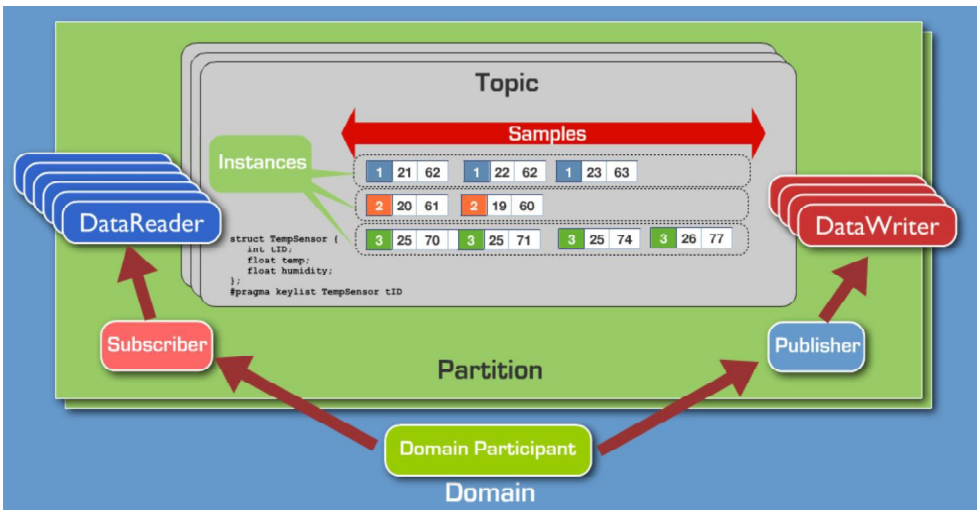

Figura 3.9. Servicio de Distribución de Datos (DDS)

Como se observa en la Figura 3.9, las aplicaciones envían y reciben la información dentro de un dominio DDS, que se trata de un espacio virtual de comunicación que conecta a publicadores, suscriptores y topics. De esta forma, un suscriptor solo podrá consumir información de un publicador que se encuentre en el mismo dominio que aquél.

Dentro de un dominio DDS también existe el concepto de partición, que es una subdivisión lógica dentro del espacio virtual de un dominio. A diferencia de los dominios, una entidad publicadora o suscriptora sí puede cambiar de una partición a otra en tiempo de ejecución, facilitando de esta forma la detección de posibles fallos en la comunicación, y proporcionando robustez en caso de la inserción de nuevas funcionalidades al sistema.

En general, DDS se construye para trabajar sobre protocolos no orientados a conexión y para el descubrimiento de nodos de forma automática, no requiriendo de servidores o sistemas especiales para la comunicación, y ofreciendo una infraestructura descentralizada, eficiente, resistente a fallos, y flexible ante posibles cambios en el sistema.

En la Figura 3.10 se presenta los niveles arquitectónicos de DDS, en donde se especifican dos capas: el DCPS (Publicador/Suscriptor Centrado en Datos), que se encarga de la distribución de datos entre entidades, y el DLRL (Capa de Reconstrucción de Datos Locales), un interfaz opcional que ayuda a la integración de la capa DCPS a nivel de aplicación, permitiendo que los datos localizados remotamente puedan ser tratados por un equipo como si fueran locales.

Figura 3.10. Niveles arquitectónicos en DDS

La capa DCPS está constituida por diversos módulos y las interacciones entre los mismos, distinguiéndose entre entidades productoras de información o escritores de datos, y entidades consumidoras o lectoras de datos, tal y como se aprecia en la Figura 3.11.

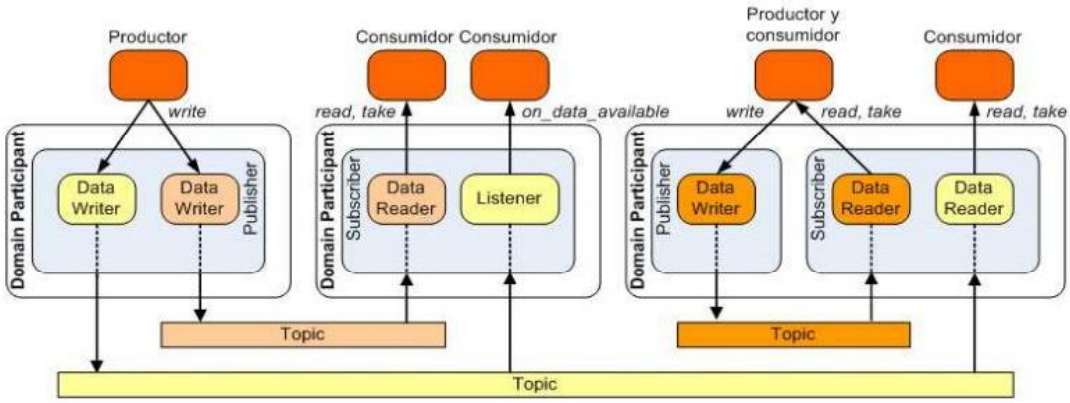

Figura 3.11. Entidades del sistema DDS

A continuación se realiza una breve descripción de cada una de las entidades pertenecientes al nivel DCPS:

1. **Dominio**: es la unidad básica que une a varias aplicaciones que tienen intención de comunicarse. DDS tiene la capacidad de soportar múltiples dominios, y así ofrecer a los desarrolladores de un sistema la posibilidad de separar diferentes tipos de datos, minimizando de esta forma el impacto de nuevas funcionalidades que puedan incorporarse.
1. **DomainParticipant**: es el encargado de crear las diferentes entidades dentro de un dominio, así como sus diferentes políticas de calidad de servicio asociadas. De esta forma, solo los publicadores y los suscriptores dentro de un mismo dominio pueden interactuar entre ellos.
2. **Suscriber**: es el objeto responsable de recibir datos publicados y ponerlos a disposición de la aplicación (en función de las QoS del Suscriber). Para acceder a estos datos, la aplicación debe utilizar la entidad DataReader, asociada al suscriptor
3. **DataReader**: es el punto de entrada del suscriptor, y puede obtener la información mediante dos métodos: a iniciativa de la aplicación a través de las funciones *take*() y *read*(), o por medio de un aviso a través de un Listener cuando se cumplan ciertos criterios determinados por condiciones impuestas durante la creación de la entidad.
4. **Publisher**: es el objeto responsable de la distribución de datos, decidiendo qué información se quiere publicar, en qué momento y dónde. Para publicar estos datos, la aplicación debe utilizar la entidad DataWriter. De esta forma, un publicador puede manejar varias entidades DataWriters, cada una asociada a un tipo de información diferente.
5. **DataWriter**: escribe la información en un Topic empleando la función *write*() basándose en las políticas de calidad de servicio definidas por el Publisher.
6. **Topic**: es la estructura de comunicación por el cual se comunican productores y consumidores. Está compuesto por un *Topicname*, que lo identifica dentro de un dominio y que además debe ser único dentro del mismo, y un *Topictype*, que define el contenido del topic utilizando el lenguaje IDL. Además, dentro de los campos de un topic pueden utilizarse los denominados *Topickey*, para contener canales lógicos independientes que permitan ordenar y diferenciar datos entrantes, asemejándose al concepto de PRIMARY_KEY empleado por las bases de datos convencionales.

- **Comparativa**

En el Anexo I de esta memoria se comparan las distintas tecnologías presentadas con anterioridad, en función de diversas características asociadas a un sistema distribuido.