Tutorial DDS

Proyecto Middleware en tiempo real basado en el modelo publicación/suscripción

Febrero, 2015

Contenido

Índice de Figuras

Índice de Tablas

Servicio de Distribución de Datos (Data Distribution Service), DDS

1. Introducción

Servicio de Distribución de Datos para sistemas en tiempo real (Denominado de forma abreviada también como DDS) es la especificación para un middleware de tipo publicación/suscripción en sistemas distribuidos. DDS ha sido creado en el año 2011, en respuesta a las necesidades de la industria de estandarizar sistemas centrados en datos (en inglés data-centric systems). Existen soluciones propietarias de DDS que están disponibles desde hace ya algunos años, desde el año 2004 hay dos empresas encargadas de DDS, la americana Real-Time Innovations[1] y la francesa Thales Group[2] ambas han participado en las especificaciones aprobadas por el Object Management Group (OMG[3]) en un documento denominado Data Distribution Service for Real-time Systems. (Data Distribution Service, 2013)

El estándar DDS OMG, introducido en el año 2014, es un estándar para Tiempo Real, Confiable y de Alto Rendimiento para Publicación/Suscripción. (Corsaro, 2010)

El DDS OMG para sistemas de tiempo real es el primer estándar abierto internacional middleware, aborda directamente publicación-suscripción de comunicaciones de tiempo real y sistemas embebidos.

La OMG se encuentra ultimando la versión 1.3, donde se trabaja en la inclusión a la especificación de un nuevo tipo de topics (tópicos) denominado X-Topics. Dichos topics pueden ser definidos en tiempo de ejecución y declarados no sólo 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 interesantes 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.

DDS introduce un espacio virtual de datos globales donde las aplicaciones pueden compartir información con la simple lectura y escritura de datos-objetos abordados por medio de un nombre definido por la aplicación (Topic) y una clave. DDS cuenta con un control preciso y extenso de los parámetros de calidad de servicio, incluyendo la fiabilidad, ancho de banda, los plazos de entrega, y los límites de recursos. (OMG, 2011).

---

[1] RTI ofrece al mundo implementaciones líderes en DDS, permite que los dispositivos puedan compartir de forma inteligente la información y trabajar juntos como un sistema integrado.

[2] El Grupo Thales es una compañía francesa de electrónica dedicada al desarrollo de sistemas de información y servicios para los mercados aeroespacial, de defensa y seguridad.

[3] OMG es un consorcio internacional sin ánimos de lucro formado por organizaciones que desarrollan estándares de tecnologías orientadas a objetos.

DDS es ampliamente adoptado a través de varios dominios de aplicaciones diferentes, tales como, Automated Trading, simulaciones, SCADA, telemetría, etc. (Corsaro, 2010)

2. Espacio Global de Datos

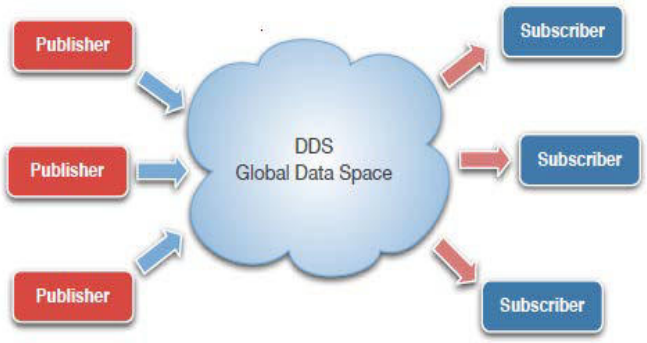

Figura 1. Espacio Global de Datos del DDS. (Corsaro, 2010)

La abstracción clave en la Fundación de DDS es un espacio totalmente distribuido de datos Global (GDS). Es importante comentar que la especificación DDS requiere la implementación del espacio global de datos para ser distribuidos completamente evitando la introducción de puntos particulares de fallo o puntos particulares de cuello de botella. Los GDS también descubren la aplicación, define los tipos de datos y los propaga como parte del proceso de descubrimiento.

En esencia, la presencia de un GDS equipado con descubrimiento dinámico significa que cuando se implemente un sistema, no hay que configurar nada. Todo será descubierto automáticamente y los datos comenzarán a fluir como se evidencia en la *Figura 1*. Por otra parte, desde los GDS se distribuye completamente no hay que temer el desplome de algún servidor, aunque las aplicaciones puedan chocar y reiniciar, o conectar/desconectar, el sistema como un todo sigue en marcha.

DDS proporciona dos mecanismos para determinar el alcance de la información, dominios y particiones como se muestra en la *Figura 2*.

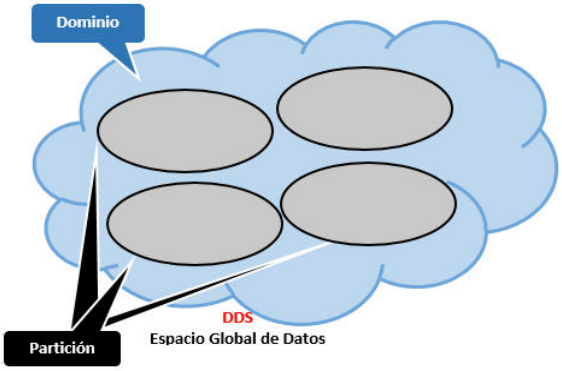

Figura 2. Dominios y Participaciones en el Espacio Global de Datos. (Corsaro, 2010)

- **Dominio:** Un dominio es una instancia del GDS y las entidades siempre pertenecen a un dominio específico. Un dominio establece una red virtual enlazando a todas las aplicaciones DDS que se han unido a ella.

- **Partición:** Es un mecanismo de alcance proporcionado por DDS para organizar una partición. Los dominios pueden ser organizados en particiones, donde cada partición representa un grupo lógico de **Topics**.

  Las particiones DDS son descritas por nombres y tienen que ser explícitamente unidas con el fin de publicar datos en estas o subscribirse a los temas que contienen. El mecanismo previsto por DDS para unirse a una partición es muy flexible como un **publisher** o un **subscriber** pueden unirse al proporcionar su nombre completo o pueden unirse todas las particiones que coincidan con una expresión regular. (Corsaro, 2010)

3. Arquitectura

La especificación DDS describe dos niveles de interfaces, como se muestra en la *Figura 3*, a continuación se detalla cada una:

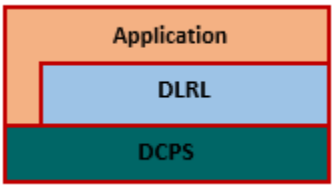

Figura 3. Arquitectura de los DDS. (Pardo-Castellote, 2014)

3.1. DCPS (Data-Centric Publish-Subscribe)

Un DCPS a nivel inferior que tiene por objeto hacer un reparto de la información de forma eficiente a los receptores apropiados. El DCPS consta de cinco módulos como se muestra en la *Figura 4*:

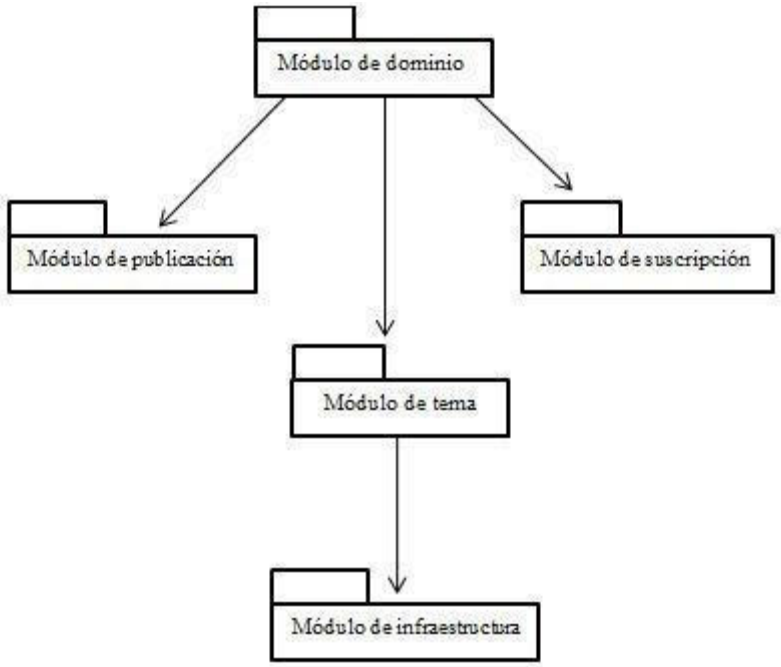

Figura 4. Desglose de los Módulos DCPS (OMG Portal, 2011).

- *Módulo de Infraestructura:* define las clases abstractas y las interfaces que se modifican por los demás módulos. También proporciona soporte para los dos estilos de interacción (basado en notificaciones y basado en espera) con el middleware.

- *Módulo de dominio:* contiene la clase ***DomainParticipant*** (Dominio de Participación) que actúa como un punto de entrada del Servicio y actúa como una fábrica para muchas de las clases. El ***DomainParticipant*** también actúa como un contenedor para los otros objetos que componen el Servicio.

- *Módulo tema:* contiene la clase ***Topic***, la interfaz ***TopicListener***, y más generalmente, todo eso es necesario por la aplicación para definir los objetos ***Topic*** y adjuntar las políticas de calidad de servicio a los mismos.

- *Módulo de publicación:* contiene las clases ***Publisher*** y ***DataWriter***, así como las interfaces ***PublisherListener*** y ***DataWriterListener*** y, en general, todo lo que se necesita en el lado de publicación.

- *Módulo de suscripción:* contiene las clases ***Subscriber***, ***DataReader***, ***ReadCondition***, y ***QueryCondition***, así como las interfaces ***SubscriberListener*** y **DataReaderListener**, y en general, todo lo que se necesita en el lado de suscripción. (DDS for Real-Time, 2014)

3.2. DLRL (Data Local Reconstruction Layer)

Una capa superior opcional **DLRL** que permite una integración simple de DDS en la capa de aplicaciones.

3.3. Ventajas de su empleo

- Disminución del acoplamiento entre entidades*:* debido en parte al empleo de la filosofía publique/suscríbase.
- Arquitectura flexible y adaptable*:* gracias al empleo del 'discovery' automático (RPTS).
- Eficiencia: debido a la comunicación directa entre el publicador y el suscriptor.
- Determinismo: en la consigna de los datos.
- Escalabilidad: debido en parte a la disminución del acoplamiento entre entidades.
- Calidad de servicio: altamente parametrizable.
- Independencia de la plataforma: debido al empleo de estándares como IDL. (Data Distribution Service, 2013)

4. Módulos DDS

El funcionamiento del DDS con sus diferentes módulos se representa en la *Figura:*

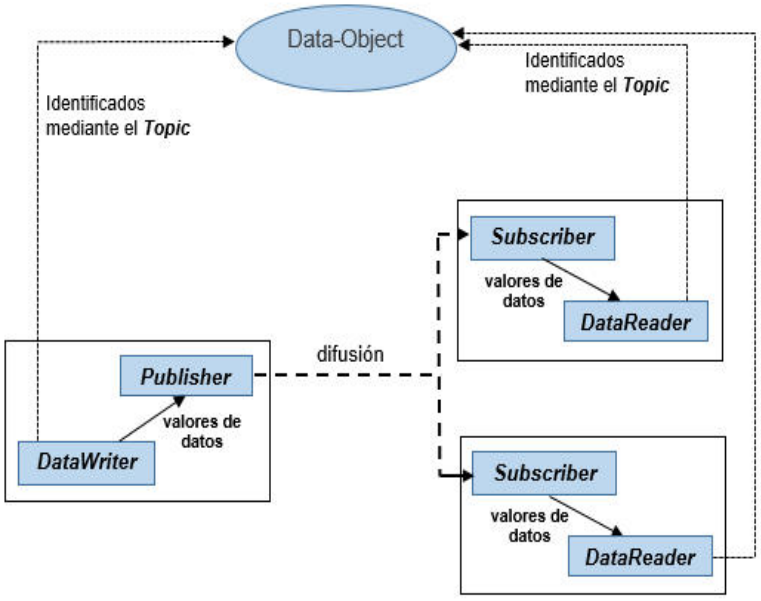

Figura 5. Visión de conjunto de los Módulos DDS. (OMG Portal, 2011)

Como se puede observar en la Figura 5, los flujos de información con la ayuda de las siguientes construcciones: **Publisher** (Publicador) y **DataWriter** (Escritor) en el lado emisor, **Subscriber** (Suscriptor) y **DataReader** (Lector en el lado receptor.

4.1. Publisher (Publicador)

Un **Publisher** 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*[4] (es decir, cada objeto **DataWriter** se dedica a los tipos de datos de una aplicación) de acceso a una publicación. Un **DataWriter** es el objeto que la aplicación debe utilizar para comunicar a un **publisher** 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 **publisher** a través del **data-writer** apropiado, es responsabilidad del **publisher** realizar la distribución (el **publisher** hará esto de acuerdo a sus propias QoS, o calidad de servicio conectados a las correspondiente data-writer). Una *publicación* es definida por la asociación de un **data-writer** a un **publisher**. Esta asociación expresa la intención de la aplicación de publicar los datos descritos por el data-writer en el contexto proporcionado por el **publisher**.

4.2. Subscriber (Suscriptor)

Un **Subscriber** es un objeto responsable de recibir los datos publicados y ponerlos a disposición (de acuerdo con QoS del **Subscriber**) de la aplicación receptora. Puede recibir y despachar datos de diferentes tipos especificados. Para acceder a los datos recibidos, la aplicación debe utilizar un *typed* **DataReader** adjunto al **subscriber**. Así, una *suscripción* se define por la asociación de un **data-reader** con un **subscriber**. Esta asociación expresa el intento de la aplicación para suscribirse a los datos descritos por el **data-writer** en el contexto proporcionado por el **subscriber**.

Cuando una aplicación desea publicar los datos de un determinado tipo, debe crear un **Publisher** (o volver a utilizar uno ya creado) y un **DataWriter** con todas las características de la publicación deseada. Del mismo modo, cuando una aplicación desea recibir datos, debe crear un **Subscriber** (o volver a utilizar uno ya creado) y un **DataReader** para definir la suscripción.

A continuación en la *Figura 6* se muestra el modelo de dominio de una aplicación DDS, donde se observan los principales objetos de comunicación que siguen patrones unificados:

[4] 'typed' significa que cada objeto *DataWriter* es dedicado a una aplicación de tipo de dato.

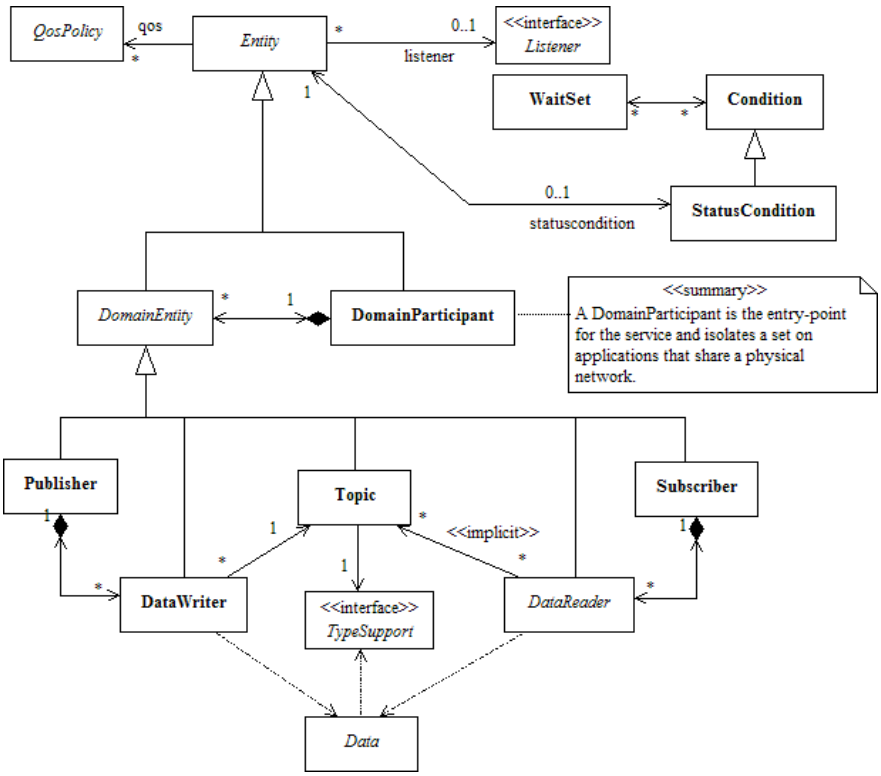

Figura 6. Modelo de Dominio de una aplicación DDS. (OMG Portal, 2011)

Apoyando los QoS (compuesto por varios *QosPolicy*); QoS proporciona un mecanismo genérico para la aplicación para controlar el comportamiento del Servicio y adaptarlo a sus necesidades. Cada **Entity** soporta su propio tipo especializado de las políticas QoS.

Aceptando un **Listener**; los **Subscriber** proporcionan un mecanismo genérico para el middleware de notificar a la aplicación de eventos asíncronos pertinentes, como la llegada de los datos correspondientes a una suscripción, la violación de un ajuste de calidad de servicio, etc. Cada **Entity** soporta su propio tipo especializado de **Subscriber**. Los **Subscriber** están relacionados con cambios en las condiciones de estado.

Se debe considerar que sólo se permite un **Subscriber** por entidad (en lugar de una lista de ellos). La razón de esta elección es que esto permite una más simple (y, por lo tanto, más eficiente) implementación tan lejos como al middleware se refiere. Por otra parte, si se requiere, la aplicación podría implementar fácilmente un **Subscriber** que, cuando se activan, los disparadores provocan un cambio adjunto 'sub-listeners'.

Aceptando un **StatusCondition** (y un conjunto de objetos **ReadCondition** para el **DataReader**; las condiciones (en conjunto con objetos **WaitSet**) proporcionan apoyo a un estilo de comunicación alternativa entre el middleware y la aplicación (es decir, wait-based en lugar de notification-based).

Cabe mencionar que todas las entidades están unidas a un **DomainParticipant**. Un **DomainParticipant** representa los miembros locales de la aplicación en un dominio. Un *dominio* es un concepto distribuido que une todas las aplicaciones capaces de comunicarse entre sí. Representa un plano de comunicación: sólo los **Publishers** y los **Subscribers** conectados al mismo dominio puede interactuar.

DomainEntity es un objeto intermedio cuyo único propósito es declarar que un **DomainParticipant** no puede contener otros participantes de dominio.

A nivel DCPS, los tipos de datos representan la información que se envía de forma atómica. Por motivos de rendimiento, sólo el plano de estructuras de datos es manejado por este nivel.

→ Por defecto, cada modificación de datos se propaga individualmente, independientemente, y no correlacionados con otras modificaciones. Sin embargo, una aplicación puede solicitar que varias modificaciones se pueden enviar como un todo y se interpretan como tales en el lado receptor. Esta funcionalidad se ofrece sobre una base **Publisher**/**Subscriber**. Es decir, estas relaciones sólo se pueden especificar entre objetos **DataWriter** adjuntos al mismo **Publisher** y recuperados entre objetos **DataReader** adjuntos al mismo **Subscriber**. (Data Distribution Service, 2013)

4.3. Topic

Un **Topic** representa la unidad de información que puede ser producida o consumida; es como una tríada compuesta por un tipo, un nombre único, y un conjunto de calidad de servicio (QoS) como se muestra en la *Figura 7,* que se utiliza para controlar las propiedades no funcionales asociadas con el **topic**. Por el momento, se puede decir que si los QoS no se establecen de manera explícita, la aplicación DDS utilizará algunos valores predeterminados prescritos por la norma.

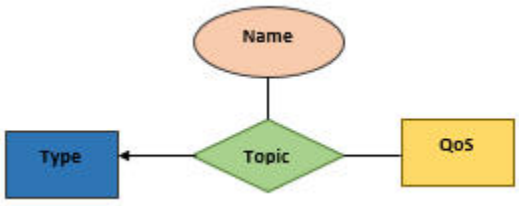

Figura 7.Objeto Topic y sus componentes. (Corsaro, 2010)

Los **Topic** son objetos que conceptualmente encajan entre las publicaciones y suscripciones. Las publicaciones deben ser conocidas de tal manera que las suscripciones pueden referirse a ellos sin ambigüedades. Un **Topic** está destinado para cumplir con ese propósito: se asocia un nombre (único en el dominio es decir, el conjunto de aplicaciones que se comunican entre sí), un tipo

de datos, y la calidad de servicio en relación con los datos en sí. En adición al **Topic** del QoS, la calidad de servicio del **DataWriter** asociado a ese **Topic** y la calidad del servicio del **Publisher** asociado al **DataWriter** controlan el comportamiento de parte del publicador, mientras la calidad del servicio de los correspondientes **Topics**, **DataReader** y **Subscriber** controlan el comportamiento del lado del suscriptor.

Un tipo de **Topic** DDS es descrito por una estructura IDL (Interface Definition Language)[5] que contiene un número arbitrario de campos cuyos tipos podría ser: tipo primitivo (Tabla I), un tipo template (Tabla II) o un tipo compuesto (Tabla III).

Tabla I. Tipos IDL Primitivos.

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

Como se muestra en la Tabla I, el tipo *int* no está allí, esto se debe a que los tipos integrales IDL *short, long, long long* son equivalentes al *int16_t, int32_t y int64_t*, por lo cual no habría problema para usarlos.

Tabla II. 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;

[5] Es un lenguaje de informática utilizado para describir la interfaz de componentes software. Describe una interfaz en un lenguaje neutral, lo cual permite la comunicación entre componentes de software desarrollados en diferentes lenguajes.

	sequence<MyType> mtseq; sequence<MyType, 10> mtseq10;
fixed<digits,scale>	fixed<5,2> fp; //d1d2d3.d4d5

Como se muestra en la Tabla II, el *string* y *wstring* se pueden parametrizar sólo con respecto a su longitud máxima; el tipo *sequence* con respecto a su longitud y tipo contenido; el tipo *fixed* con respecto al número total de dígitos y la escala. El tipo *sequence* abstrae contenedor homogéneo de acceso aleatorio, como el *std:vector* en *C++* o *java.util.Vecto*r en Java. Finalmente, es importante señalar que cuando no se proporciona la longitud máxima del tipo se asume como que tiene una longitud ilimitada, lo que significa que el middleware asignará tanta memoria como sea necesario para almacenar los valores de la aplicación proporciona.

Tabla III. 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;
	};

La Tabla III muestra que el DDS soporta tres diferentes tipos IDL compuestos, *enum, struct* y *union*. Juntándolos todos, un tipo de **Topic** es una *struct* que puede contener campos anidados estructuras, uniones, enumeraciones, tipos template, así como los tipos primitivos. Además de esto, se pueden definir *array* multidimensionales de cualquier tipo soportado por DDS o definido por el usuario.

Cómo se enlaza esto con otros lenguajes de programación como C++, Java o C#? La respuesta no es realmente sorprendente, esencialmente hay un mapeo

específico del idioma de los tipos IDL descritos anteriormente para incorporar los lenguajes de programación. (Corsaro, 2010)

Por definición, un **Topic** corresponde a un único tipo de datos; sin embargo, varios **Topic** pueden referirse al mismo tipo de datos. Por lo tanto, un **Topic** identifica los datos de un solo tipo, que van desde una sola instancia a toda una colección de instancias de ese tipo dado. Esto se muestra a continuación para el tipo de datos hipotéticos **Foo** (Ejemplo de objeto) en la *Figura 8*, mostrando que una colección de instancias de objetos, identificado por sus **key**, puede difundirse sobre un tema.

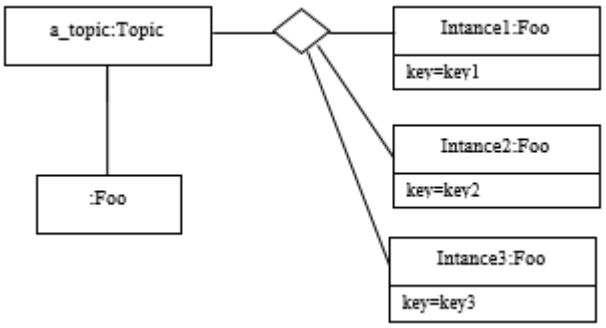

Figura 8. Un Topic con varias instancias. (OMG Portal, 2011)

En caso de que un conjunto de instancias se hayan reunido con un mismo **Topic**, las diferentes instancias deben ser distinguibles. Esto se consigue por medio de los valores de algunos campos de datos que forman la **key (clave)** para ese conjunto de datos. La descripción de la clave (es decir, la lista de campos de datos cuyo valor constituye la clave), tiene que ser indicados al middleware. La regla es simple: *diferentes muestras de datos con el mismo valor de la clave representan valores sucesivos de la misma instancia, mientras que diferentes muestras de datos con diferentes valores de la clave representan diferentes instancias*. Si no se proporciona ninguna clave, el conjunto de datos asociado con el **Topic** se limita a un *solo caso*.

Los **Topic** deben ser conocidos por el middleware y potencialmente propagados. Los objetos del **Topic** son creados utilizando las operaciones de crear proporcionadas por **DomainParticipant**.

El estilo de interacción es directa por parte del **Publisher**: cuando la aplicación sabe qué hacer con los datos disponibles para la publicación, llama a la operación correspondiente en el relacionado **DataWriter** (esto, a su vez, dará lugar a su **Publisher**).

Por el lado del **subscriber** sin embargo, hay más opciones: la información relevante puede llegar cuando la aplicación está ocupada haciendo otra cosa o cuando la aplicación está a la espera de esa información. Por lo tanto,

dependiendo de la forma en que la aplicación está diseñada, notificaciones asíncronas o acceso síncrono puede ser más apropiado. Ambos modos de interacción son permitidos, un **Listener** se utiliza para proporcionar una devolución de llamada para acceso síncrono y un **WaitSet** asociado con uno o varios objetos **Condition** proporcionan acceso de datos asíncronos.

Los mismos modos de interacción síncronos y asíncronos también se pueden utilizar para acceder a los cambios que afectan el estado de comunicación del middleware. Por ejemplo, esto puede ocurrir cuando el middleware detecta de forma asíncrona una inconsistencia.

5. Mecanismo y Técnicas para el alcance de la información

Los dominios y particiones son una manera muy buena de organizar los datos. Sin embargo, operan a nivel estructural. ¿Qué pasaría si se necesita controlar los datos recibidos basados en su situación actual? 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, sólo aquellos que coincidan con el filtro de topic. La expresión de filtro puede operar sobre el contenido de **topic** completos, en lugar de ser capaces de operar sólo en headers como sucede en muchas otras tecnologías pub/sub, como JMS[6]. La expresión de filtro es estructuralmente similar a una cláusula WHERE de SQL. Los operadores soportados se enumeran en la Tabla IV.

Tabla IV. 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

[6] Es un estándar de mensajería que permite a los componentes de aplicaciones basados en la plataforma Java2 crear, enviar, recibir y leer mensajes.

Los **topics Content-Filtered** son muy útiles, en primer lugar que limitan la cantidad de memoria utilizada por el middleware para las instancias y muestras que coincidan con el filtro. Además, el filtrado puede ser usado para simplificar su aplicación delegando al DDS la lógica que comprueba las propiedades de ciertos datos.

Los DDS también soportan las **Query Conditions** (Condiciones de Consulta). La diferencia entre un **Query Condition** y un **Topic Content-Filtered** se debe a que el anterior no filtra los datos entrantes, simplemente consulta los datos recibidos y disponibles en una caché lectora existente. Como resultado la ejecución del Query Conditions está completamente bajo el control del usuario y ejecutado en el contexto de un **read**.

6. Lectura y Escritura de Datos

A continuación se describirán los mecanismos proporcionados por DDS para la lectura y escritura de datos.

6.1. Escritura de Datos

La escritura de datos con DDS es simple 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 **writes** 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:

a) una identidad proporcionada por su valor de clave única, y
b) un ciclo de vida.

El ciclo de vida del **topic-instances** puede ser manejado implícitamente a través 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.

6.2. Ciclo de Vida de Topic-Instances

Antes de detallar como es manejado el ciclo de vida, veamos cuáles son los estados disponibles. Un **topic-instance** está en estado:

- ALIVE si por lo menos hay un **DataWriter** escribiendo.
- NOT_ALIVE_NO_WRITERS cuando no hay más **DataWriters** escribiendo.

- NOT_ALIVE_DISPOSED si esta fue eliminada ya sea implícitamente debido a un defecto de QoS o explícitamente mediante una llamada específica al API **DataWriter**.

El estado NOT_ALIVE_DISPOSED indica que la instancia ya no es relevante para el sistema y que no debería ser escrita más (o algún tiempo pronto) por cualquier **writer**. Como resultado, los recursos asignados en el sistema para almacenar la instancia pueden ser liberados.

6.2.1. Administración del ciclo de vida automática

Se explicará el ciclo de vida de las instancias con un ejemplo. En la *Tabla V* se observa el código de una aplicación, tener en cuenta que éste solamente es para escribir datos, el resultado de las tres operaciones **write** es crear tres nuevas instancias de **topic** en el sistema para los valores de claves asociados con los id = 1, 2, 3 (se ha definido el *TempSensorType* en el primer instanciamiento como tener un particular atributo **key** nombrado *id*). Estas instancias estarán en estado ALIVE siempre que estas aplicaciones estén corriendo y serán registradas automáticamente, o se podría decir que están asociadas con el **writer**. El comportamiento predeterminado de DDS es entonces disponer las instancias de **topic** una vez que se destruye el objeto **DataWrite**, llevando así las instancias a estado NOT_ALIVE_DISPOSED. Se puede reemplazar la configuración predeterminada para inducir a anular los registros de instancias, causando en este caso una transición de ALIVE a NOT_ALIVE_NO_WRITERS.

Tabla V. 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;
}
```

6.2.2. Administración del 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.

La *Tabla VI* muestra un ejemplo de cómo el API **DataWriter** puede ser usado para registrar, dejar de registrar y eliminar **topic-instances**. La *Tabla VI* detalla una aplicación que escribe cuatro muestras pertenecientes a cuatro diferentes **topic-instances**, *id = 0, 1, 2, 3* respectivamente. Las instancias con *id = 1, 2, 3* son explícitamente registradas con la llamada al método **DataWriter::register_instance**, mientras que la instancia con el *id = 0* es automáticamente registrado como resultado de escribir en el **DataWriter**. Para mostrar los diferentes estados de transición posibles, el **topic-instances** con *id = 1* está explícitamente dejado de registrar, causando la transición al estado de NOT_ALIVE_NO_WRITER; el **topic-instance** con *id = 2* es explícitamente eliminado, causando la transición al estado NOT_ALIVE_DISPOSED. Finalmente, el **topic-instances** con el *id = 3*, será automáticamente anulado, como resultado de la destrucción de los `dw` y `dwi3` respectivamente, causando la transición al estado NOT_ALIVE_NO_WRITER.

6.2.2.1. Topic sin claves

Los **topic** sin claves son como únicos, en el sentido de que hay sólo una instancia. Como resultado para los **topic** sin clave el estado de transición es vinculado al ciclo de vida del **data-writer**.

Tabla VI. Gestión explícita del ciclo de vida del topic-instances.

Ejemplo de una aplicación que escribe cuatro muestras pertenecientes a cuatro diferentes **topic-instances.**

```
int main(int, char**) {
dds::Runtime runtime;
dds::Topic<TempSensorType> topic("TempSensorTopic");
//[NOTE #1]: Avoid topic-instance dispose on unregister
dds::DataWriterQos dwqos;
   dwqos.set_auto_dispose(false);
   //[NOTE #2]: Creating DataWriter with custom QoS.
   // QoS will be covered in detail in article #4.
   dds::DataWriter<TempSensorType> dw(topic, dwqos);
   TempSensorType data = {0, 24.3F, 0.5F, CELSIUS};
   dw.write(data);
   TempSensorType key;
   key.id = 1;
   //[NOTE #3]: Using C++0x “auto” to keep code more compact
   //[NOTE #4] Registering topic-instance explicitly
   auto diw1 = dw.register_instance(key);
   key.id = 2;
   auto diw2 = dw.register_instance(key);
   !
   key.id = 3;
   auto diw3 = dw.register_instance(key);
   data = {1, 24.3F, 0.5F, CELSIUS};
   diw1.write(data);
   data = {2, 23.5F, 0.6F, CELSIUS};
   diw2.write(data);
   data = {3, 21.7F, 0.5F, CELSIUS};
   diw3.write(data);
   // [NOTE #5]: unregister topic-instance with id=1
   diw1.unregister();
   // [NOTE #6]: dispose topic-instance with id=2
   diw2.dispose();
   //[NOTE #7]:topic-instance with id=3 will be unregistered
   // as result of the diw3 object destruction
   //[NOTE #8]: topic instance with id=0 will be unregistered
as
   // result of the dw object destruction
   return 0;
}
```

6.3. Lectura de Datos

DDS proporciona dos mecanismos diferentes para acceder a los datos, cada uno de ellos permite controlar qué datos son accedidos y serán informados acerca de la disponibilidad de los datos.

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

6.3.1. Read vs. 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.

6.3.2. 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.

6.3.3. Selección de muestras

Independientemente de leer los datos o quitarlos del DDS, se utiliza el mismo mecanismo para especificar las propiedades de las muestras, para que se entienda mejor, se darán algunos ejemplos utilizando la semántica **read** pero si se desea utilizar la semántica de **take**, simplemente se sustituye cada ocurrencia de **read** por **take**. Específicamente, DDS permite seleccionar datos basados en valores del estado de vista, el estado de la instancia y el estado de la muestra. Por ejemplo, si se quiere conseguir todos los datos recibidos, no importa el estado de la vista, instancia o muestra, se emitiría **read** (o un **take**) como sigue:

```
dr.read(data, info, ANY_SAMLE_STATE, ANY_VIEW_STATE,
                                    ANY_INSTANCE_STATE);
```

Si por el contrario se quiere solo leer (o tomar) todas las muestras que aún no han sido leídas, se emitiría **read** (o un **take**) como sigue:

```
dr.read(data, info, NOT_READ_SAMLE_STATE, ANY_VIEW_STATE,
  ANY_INSTANCE_STATE);
```

Y si se quiere asegurar que siempre sólo lea datos válidos nuevos, se emitiría **read** (o un **take**) como sigue:

```
dr.read(data, info, NOT_READ_SAMLE_STATE, ANY_VIEW_STATE,
    ALIVE_INSTANCE_STATE);
```

Finalmente, si solo se quiere leer datos asociados a instancias que están marcando sus apariciones en el sistema por primera vez, se emitiría **read** (o un **take**) como sigue:

```
dr.read(data, info, NOT_READ_SAMLE_STATE, NEW_VIEW_STATE,
    ALIVE_INSTANCE_STATE);
```

Notifique que con estos tipos de lecturas, se pueden obtener sólo la primera muestra escrita por cada instancia. Aunque parezca un caso extraño, esto es muy útil para todas aquellas aplicaciones que necesitan hacer algo especial cada vez que una nueva instancias hace su aparición en el sistema por primera vez. Un ejemplo podría ser un nuevo avión entrando en una nueva región de control, en este caso el sistema tendría que hacer bastantes cosas que son exclusivas de esta transición de estado específico.

Cabe mencionar que si un estado es omitido, y un **read** (o un **take**) es emitido como sigue:

```
dr.read(data, info);
```

Esto es equivalente a seleccionar muestras con el NOT_READ_SAMPLE_STATE, ALIVE_INSTANCE_STATE y ANY_VIEW_STATE.

Es importante diferenciar los estados con las condiciones de lectura. Las condiciones de lectura son usadas para consultar datos con respecto a su contenido. En cambio, los estados se utilizan para seleccionar datos con respecto a su meta-información.

6.3.4. Iteradores o contenedores

El API del contenedor **read**/**take** se basa en **std::vector** para obtener información de los datos y muestras. El tamaño del vector pasado al **read** (o el **take**) es usado para decidir el número máximo de muestras que deberían ser leídas. Si el tamaño no es cero, a lo sumo se leerá el número de muestras iguales al tamaño pasado. Si el tamaño es cero, entonces el contenedor será redimensionado para almacenar todas las muestras disponibles. El ejemplo siguiente leerá todas las muestras disponibles:

```
std::vector<TempSensorType> data;
std::vector<SamplInfo> info;
read(data, info);
```

Mientras este, solo debería leer a lo mucho 10 muestras:

```
std::vector<TempSensorType> data(10);
std::vector<SamplInfo> info(10);
read(data, info);
```

El API del iterador **read**/**take** soporta iteradores tanto adelante como atrás insertando iteradores. Este API permite leer (o tomar) datos entre cualquier estructura deseada, tan lejos como puedas conseguir un avance o atrás para insertar un iterador por él. Si nos enfocamos en avance del iterador basado en el API, la inserción trasera es bastante similar, entonces se podría leer los datos de la siguiente manera:

```
TempSensorType* data = ...; // Get hold of some memory
SampleInfo* info = ...; // Get hold of some memory
uint32_t samples = dr.read(data, info, max_samples);
```

6.3.5. Bloqueo o no bloqueo de Read/Take

El **read** y el **take DDS** estan 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.

7. Esperando notificados

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**.

7.1. 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**. Esto funcionalmente es proporcionado por el DDS a través de la clase **Waitset** la cual puede ser vista como una versión orientada a objetos del Unix **select**.

Las condiciones DDS pueden estar asociadas con objetos functor que se utilizan para ejecutar aplicaciones específicas lógicas cuando se desencadena la condición.

7.2. 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. (Corsaro, 2010)

8. EJEMPLO: Aplicación Hola Mundo usando el API DDS en JAVA

Para utilizar las librerías se debe realizar los siguientes pasos:

- ✓ Descargar la librería de la página de RTI en el siguiente link: http://www.rti.com/downloads/index.html
- ✓ Instalar la aplicación obtenida.
- ✓ Instalar el archivo de la licencia: para ello se debe copiarel archivo de la licencia (enviado por el RTI) en los directorios C:..RTI y en C:..RTI\ndds.x.x.x
- ✓ Fijar las variables de entorno como se muestra a continuación:
  - RTI_LICENSE_FILE con el valor de C:..RTI.
  - NDDSHOME con el valor de C:..RTI\ndds.x.x.x..
  - PATH anexar C:..RTI\ndds.x.x.x\lib\i86Win32jdk
- ✓ Crear un proyecto en el IDE Eclipse con el nombre **MyDDSProject** y añadir las librerías C:..RTI\ndds.x.x.x\class\nddsjava.jar a las bibliotecas del proyecto (MyDDSProject-> Properties->Java Build Path->Libraries->Add external JARs).

Esta aplicación cuenta con dos clases, un editor que permite a un usuario escribir una línea arbitraria de texto y un abonado que recibirá ese texto tan pronto como esté disponible.

A continuación se detalla la implementación de la aplicación:

A. Implementando el Publisher (Publicador)

Crear la clase Publicador que en este caso se denomina *HelloPublisher*.

```
public class HelloPublisher {
   public static final void main(String[] args) {
```

Crear un *DomainParticipant* (Dominio de Participación). El *DomainParticipant* define en qué dominio de la aplicación pertenece y se utilizará para crear todas las demás entidades necesarias, como el *Publisher*, *DataWriter* y *Topic*. El método *create_participant* () toma cuatro argumentos especificados en la *Tabla VII*:

Tabla VII. Argumentos del método créate_participant().

domainID	Un valor entero que define el dominio que está en *DomainParticipant*.
QoS	Calidad de ajustes de servicio que se utilizará. En este caso se utilizan los parámetros de calidad de servicio por defecto.
listener	Un oyente escucha opcional (rutina de devolución de llamada) que se invoca cuando se producen eventos específicos con respecto al *DomainParticipant*.
mask	Define el tipo de eventos que desencadenan la rutina de devolución de llamada (*listener*).

En el siguiente código se realiza la creación del *DomainParticipant*.

```
// Create the DDS Domain participant on domain ID 0
DomainParticipant participant = DomainParticipantFactory.get_instance().create_participant(
       0,                                                  // Domain ID = 0
       DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT,   // QoS Settings = Default
       null,                                               // listener
   StatusKind.STATUS_MASK_NONE);                           // mask
if (participant == null) {
       System.err.println("Unable to create domain participant");
       return;
}
```

B. Creando el Topic (Tema)

El *DomainParticipant* se usa para crear un *Topic* con el nombre “Hello, World” y el tipo de dato incorporado *String*. Recordar que siempre un *Topic* está compuesto por un nombre y un tipo de dato. El método *create_topic*() tiene cinco parámetros especificados en la *Tabla VIII*:

Tabla VIII. Argumentos del método create_topic().

topic_name	Nombre del *Topic* (String)
type_name	El nombre del tipo de datos de usuario. En este caso, se utiliza el tipo de datos de

	cadena incorporado.
QoS	Ajuste de la calidad de servicio que se utilizará. En este caso se utilizan los parámetros de calidad de servicio por defecto.
listener	Un listener opcional (rutina de devolución de llamada) que se invoca cuando se producen eventos específicos con respecto al *Topic*. Si este es nulo, el listener *DomainParticipant* (si existe) se utilizado.
mask	Define el tipo de eventos que desencadenan la rutina de devolución de llamada (*listener*).

En el siguiente código se crea un *Topic* denominado *helloWorldTopic*.

```
// Create the topic "Hello World" with the built-in String type
Topic helloWorldTopic = participant.create_topic(
        "Hello, World",                             //topic_name
        StringTypeSupport.get_type_name(),   //type_name
        DomainParticipant.TOPIC_QOS_DEFAULT,//QoS
        null,                                       //listener
        StatusKind.STATUS_MASK_NONE);       //mask
if (helloWorldTopic == null) {
        System.err.println("Unable to create topic.");
        return;
}
```

C. Crear el DataWriter (con Publisher por defecto)

El *DomainParticipant* ahora puede crear instancias de un *DataWriter*. Este objeto, posteriormente, puede ser utilizado para escribir los mensajes que se enviarán a los suscriptores. El método *create_datawriter()* tiene cuatro argumentos especificados en la *Tabla IX*:

Tabla IX. Argumentos del método create_datawriter().

topic	El *Topic* para el cual este *DataWriter* escribirá mensajes/datos.
QoS	El nombre del tipo de datos de usuario. En este caso, se utiliza el tipo de datos de cadena incorporado.
listener	Un *listener* opcional (rutina de devolución de llamada) que se invoca cuando se producen eventos específicos con respecto a la *DataWriter*. Si este es nulo, el listener del Publisher (si existe) se utilizará.
mask	Define el tipo de eventos que desencadenan la rutina de devolución de llamada (listener).

Si se ha preguntado por qué no había una instancia del *Publisher*. La creación de un *publisher* es opcional y si no se crea de forma explícita, se utiliza un *publisher* por defecto (esto es una extensión de RTI Connext DDS y no forma parte del estándar DDS). De lo contrario el *Publisher* podría ser utilizado para crear el *DataWriter*.

```
// Create the data writer using the default publisher
StringDataWriter dataWriter = (StringDataWriter)participant.create_datawriter(
        helloWorldTopic,                    // Topic
        Publisher.DATAWRITER_QOS_DEFAULT,   // QoS
        null,                               // listener
        StatusKind.STATUS_MASK_NONE);       // mask
if (dataWriter == null) {
        System.err.println("Unable to create data writer\n");
        return;
}
```

Finalmente se puede utilizar el *DataWriter* para escribir -y publicar de forma automática a través del *Publisher* por defecto- los mensajes que recibimos en este caso como la entrada del usuario. El método *write()* tiene dos argumentos especificados en la *Tabla X*:

Tabla X. Argumentos del método write().

Instance	Una instancia del tipo de datos que se enviará. En este caso un *String*.
InstanceHandle	Un *InstanceHandle*. Esto se utiliza con los tipos de datos con *key*.

Antes de finalizar el programa, se eliminan todas las entidades del *DomainParticipant* y el propio *DomainParticipant*.

```
System.out.println("Ready to write data.");
System.out.println("When the subscriber is ready, you can start writing.");
System.out.print("Press CTRL+C to terminate or enter an empty line to do a clean
shutdown.\n\n");
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
while (true) {
System.out.print("Please type a message> ");
   String toWrite = reader.readLine();
        dataWriter.write(toWrite, InstanceHandle_t.HANDLE_NIL);
        if (toWrite.equals("")) break;
   }
} catch (IOException e) {
        e.printStackTrace();
} catch (RETCODE_ERROR e) {
```

```
        // This exception can be thrown from DDS write operation
        e.printStackTrace();
}
System.out.println("Exiting...");
// Deleting entities of DomainParticipant and DomainParticipant
participant.delete_contained_entities();
DomainParticipantFactory.get_instance().delete_participant(participant);
}}
```

D. Implementando el Subscriber (Suscriptor)

El *Subscriber* es muy similar al *Publisher*. También tiene que haber un *DomainParticipant* y un *Topic* en lugar de un *DataWriter*, ahora habrá un *DataReader*. Un listener será implementado para automáticamente ser ejecutado cuando los mensajes nuevos están disponibles. Para el *Subscriber* para poder recibir mensajes enviados por el *HelloPublisher*, el *DomainParticipant* debería estar en el mismo *dominio* y el *Topic* debería tener el mismo nombre y tipo de dato.

```
public class HelloSubscriber extends DataReaderAdapter {
//                For                clean                shutdown                sequence
private static boolean shutdown_flag = false;
public static final void main(String[] args) {
//      Create      the      DDS      Domain      participant      on      domain      ID      0
DomainParticipant participant = DomainParticipantFactory.get_instance().create_participant(
        0,                                                  //      Domain      ID      =      0
        DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT,  // QoS
        null,                                               // listener
        StatusKind.STATUS_MASK_NONE);                       // mask
if (participant == null) {
System.err.println("Unable to create domain participant");
      return;
}
// Create the topic "Hello World" for the String type
Topic topic = participant.create_topic(
        "Hello, World",                     // Topic Name
        StringTypeSupport.get_type_name(),  // Topic Data Type
        DomainParticipant.TOPIC_QOS_DEFAULT,        // QoS
        null,                               // listener
        StatusKind.STATUS_MASK_NONE);       // mask
if (topic == null) {
        System.err.println("Unable to create topic.");
        return;
}
```

E. Creando el DataReader (Lector)

Una vez creado el *DomainParticipant* y el *Topic*, se procede a crear el *DataReader*. Como puede verse, tiene que ser unido a un *Topic* en instancias de tiempo y en este caso -por primera vez en nuestro ejemplo- utiliza un *listener* así como una máscara. El *listener* es, en este caso, una instancia de *HelloSubscriber*. Esa es también la razón por la clase *HelloSubscriber* se extiende desde la clase *DataReaderAdapter* y reemplaza el método *on_data_available (lector DataReader)* que es el método que se activará una vez que se disponga de datos. La máscara indica que este listener debe lanzar tan pronto como nuevos mensajes/datos están disponibles.

```
// Create the data reader using the default publisher
StringDataReader dataReader = (StringDataReader) participant.create_datareader(
        topic,                              //Topic
        Subscriber.DATAREADER_QOS_DEFAULT,  // QoS
        new HelloSubscriber(),                      // Listener
        StatusKind.DATA_AVAILABLE_STATUS);  // mask
        if (dataReader == null) {
        System.err.println("Unable to create DDS Data Reader");
        return;
}
// Reading User-Input
System.out.println("Ready to read data.");
System.out.println("Press CTRL+C to terminate.");
for (;;) {
try {
Thread.sleep(2000);
               if(shutdown_flag) break;
           } catch (InterruptedException e) {
               // Nothing to do...
           }
       }
System.out.println("Shutting down...");
// Deleting entities and DomainParticipant
participant.delete_contained_entities();
DomainParticipantFactory.get_instance().delete_participant(participant);
}
```

F. Leyendo los datos

Por último en el método *on_data_available()* la lógica para leer datos se implementa. Cada mensaje recibido (llamado muestra) en realidad está compuesto de dos partes. En primer lugar el propio mensaje (en este caso un String) y el segundo de un metarchivo de tipo *SampleInfo* que contiene información adicional acerca de este mensaje.

El *StringReader* creado luego intenta leer la siguiente muestra con el método *take_next_sample()* en la que pasa el archivo de información. Si la muestra contiene datos reales, la propiedad *valid_data* del objeto *info* será igual a la verdadera y la muestra se imprimirá en la consola. (Brunner, 2012)

```
public void on_data_available(DataReader reader) {
// Method Parameter of Type DataReader must be cast to Data Type of DataReader for this Topic
StringDataReader stringReader = (StringDataReader) reader;
SampleInfo info = new SampleInfo();
for (;;) {
try {
        String sample = stringReader.take_next_sample(info);
   if (info.valid_data) {
        System.out.println(sample);
        if (sample.equals("")) {
                shutdown_flag = true;
        }
   }
} catch (RETCODE_NO_DATA noData) {
        // No more data to read
        break;
        } catch (RETCODE_ERROR e) {
        // An error occurred
        e.printStackTrace();
   }
}}}
```

En la *Figura 9* se muestra la ejecución de la clase del Suscriptor denominada *HelloSubcriber.java*

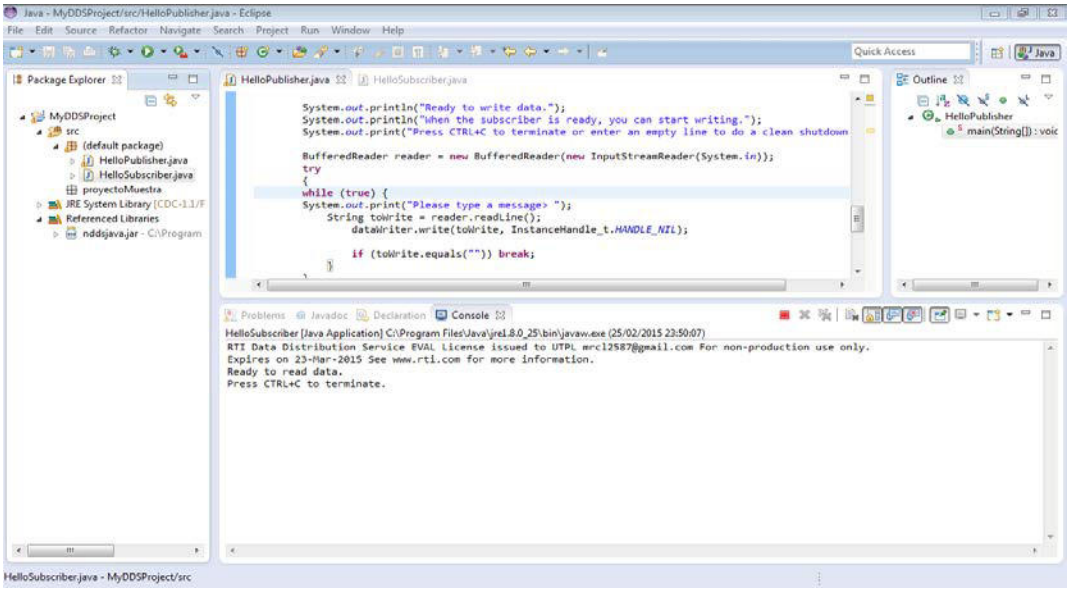

Figura 9. Ejecución de la clase HelloSubscriber.java.

En la *Figura 10* se muestra la ejecución de la clase del Publicador denominada *HelloPublisher.java*, lista para enviar los datos.

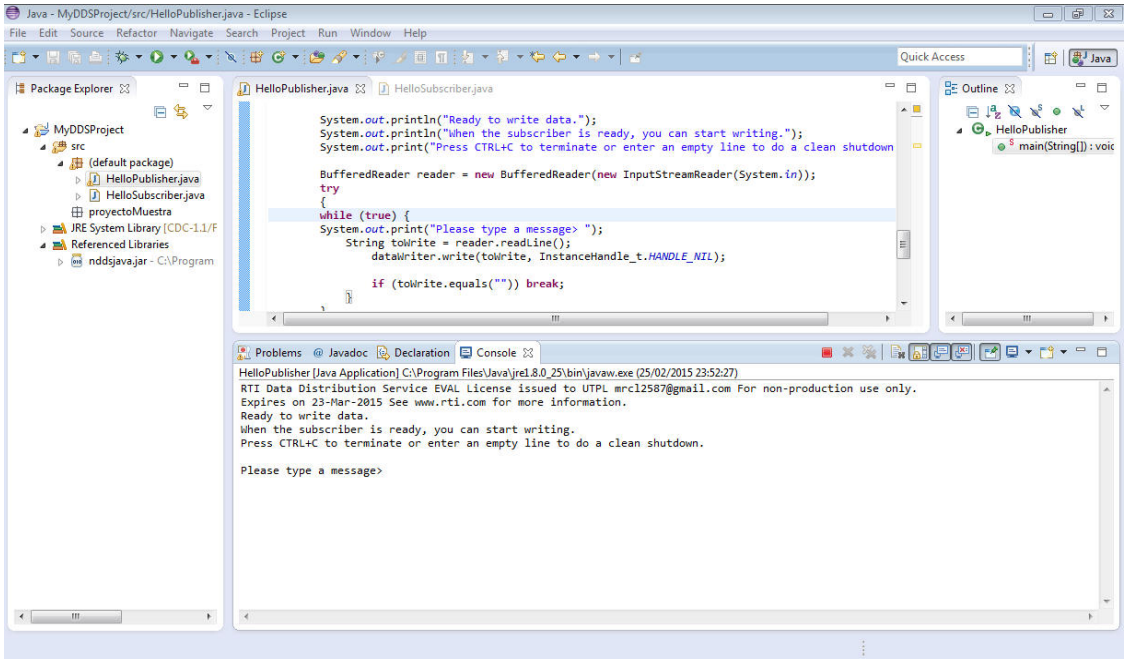

Figura 10. Ejecución de la clase HelloPublisher.java

En la *Figura 11* se muestran los datos enviados por el Publicador “Hola Mundo!! 💻”.

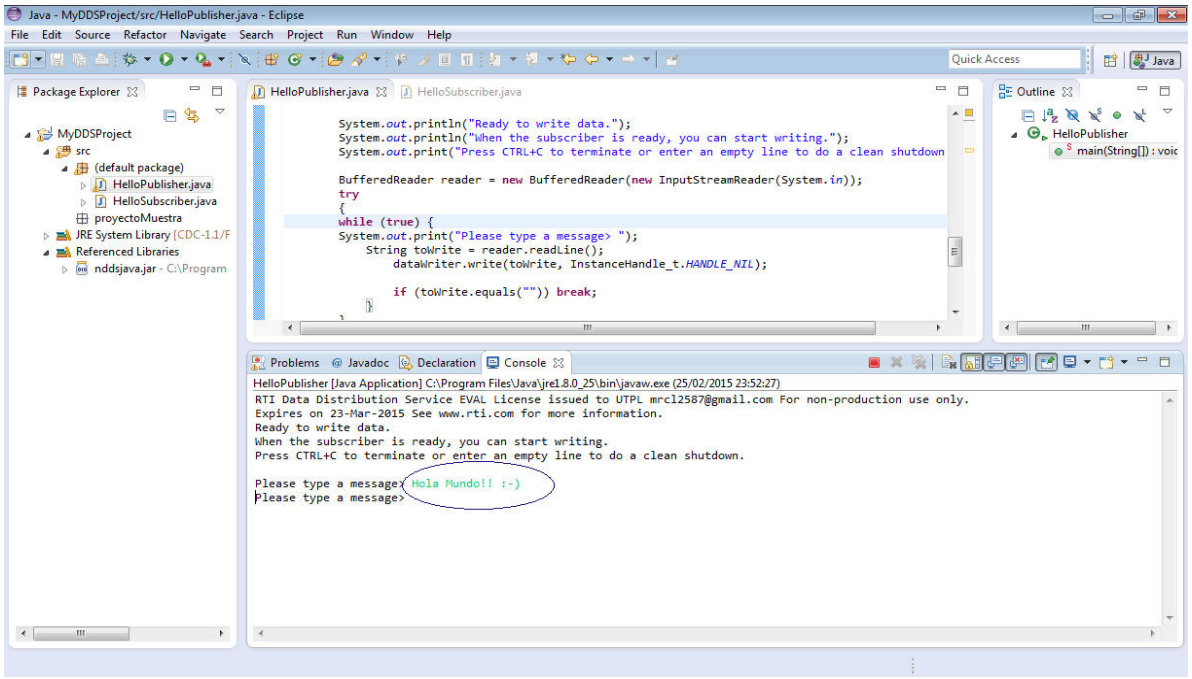

Figura 11. Publicando Hola Mundo!! :-) en la clase HelloPublisher.java

En la *Figura 12* se muestran los datos recibidos por el Suscriptor.

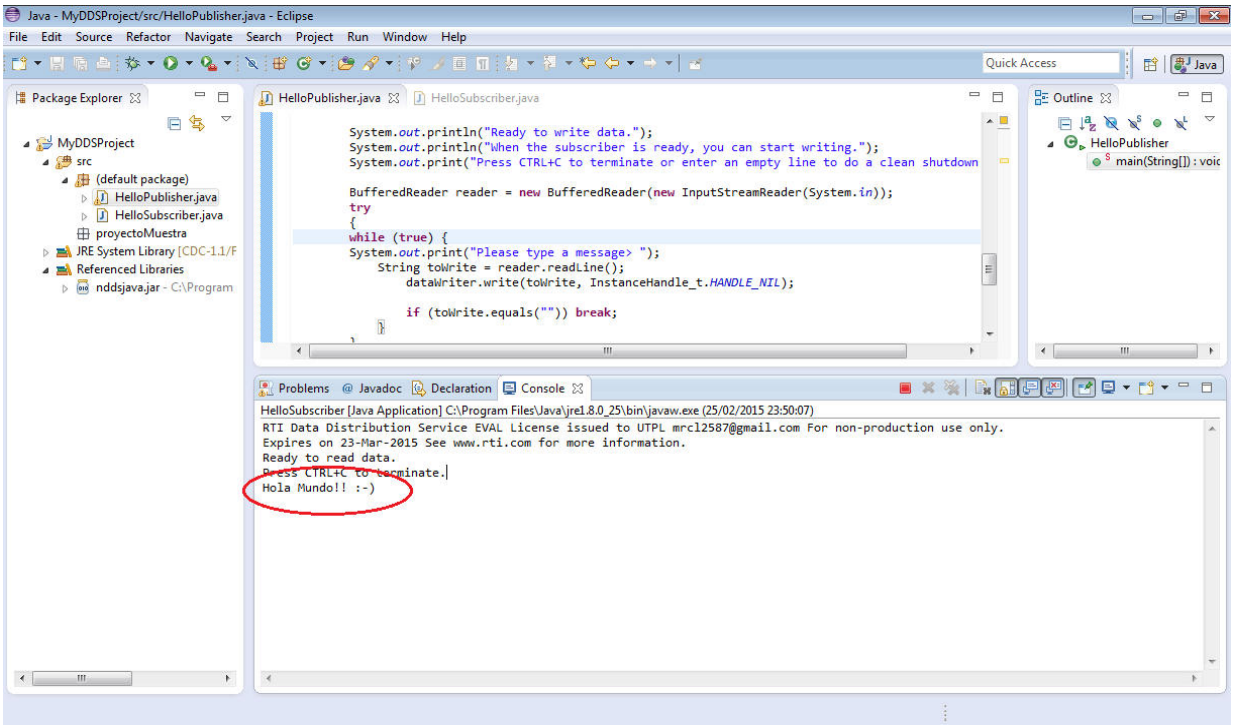

Figura 12. Recibiendo los datos en la clase HelloSubscriber.java enviados por el Publicador (clase HelloPublisher.java)

9. Referencias:

- Data Distribution Service. (2013, Diciembre 20). En *Wikipedia, la enciclopedia libre*. Obtenido de http://es.wikipedia.org/w/index.php?title=Data_Distribution_Service&oldid=70046933
- OMG. (2011, Octubre 23.). OMG Data Distribution Portal [Portal]. Obtenido de http://portals.omg.org/dds/
- Data Distribution Service for Real-Time Systems Specification (DDS for Real-Time), (2014, Junio).
- Corsaro, A. (2010, Enero 27). The DDS Tutorial. PRISMTECH. Otendido de http://www.omgwiki.org/dds/sites/default/files/Tutorial-Part.I.pdf
- Pardo-Castellote, Gerardo, Data Distribution Service, Real-Time Innovations, Inc. (2014). Obtenido de http://www.omg.org/news/meetings/workshops/RT_2004_Manual/00-T6-1_DDS.pdf
- Brunner, Sandro, RTI Connext DDS : Setup and HelloWorld Example (Windows/Eclipse/Java), (2012, Noviembre). Obtenido de http://blog.zhaw.ch/icclab/rti-connext-dds-setup-and-helloworld-example-windowseclipsejava/