Técnicas avanzadas de programación aplicadas a DDS: un nuevo enfoque

Samanta Cueva Carrión[1], Patricia Ludeña González[1], Rommel Torres Tandazo[1]

[1] Departamento de Ciencias de la Computación y Electrónica, Universidad Técnica Particular de Loja, San Cayetano Alto, Loja, Ecuador, 1101608.

Autores para correspondencia: spcueva@utpl.edu.ec, piludena@utpl.edu.ec, rovitor@utpl.edu.ec

Fecha de recepción: 28 de septiembre 2015 - Fecha de aceptación: 12 de octubre 2015

RESUMEN

La construcción de plataformas middleware para sistemas distribuidos en tiempo real supone, a día de hoy, un gran reto tecnológico y de investigación. La falta de técnicas adecuadas de programación hace que el desarrollo de estos sistemas sea un proceso complejo y costoso. En la actualidad, el desarrollo de técnicas de middleware y de programación distribuida ha generado numerosos resultados y propuestas aunque la mayor parte de ellas están centradas en sistemas de llamadas remotas y/o invocación de servicios. Las plataformas middleware más recientes incorporan técnicas de programación avanzada asociadas a Programación Orientada a Aspectos. Este artículo tiene el propósito de desarrollar una plataforma middleware para sistemas distribuidos considerando el modelo publicación/subscripción, que sirva de base para la investigación de técnicas avanzadas para sistemas distribuidos basados en eventos. Nuestra propuesta describe las experiencias en el desarrollo y adaptación del estándar a la plataforma C# en donde se prueban conceptos sobre la viabilidad de aplicar técnicas avanzadas de programación en la construcción práctica de código abierto de un middleware DDS (Data Distribution System).

Palabras clave: Middleware, DDS, Tiempo-Real, middleware, publicación-suscripción, distribución de datos.

ABSTRACT

Actually, the creation of middleware platforms for real time distributed systems is a big challenge to research and technology. There are not enough programing techniques, making the development of these systems a complex and expensive task. Nowadays, the development of techniques of middleware and distributed programing has created important results and proposals, almost all of them oriented to remote calls and service invocation systems. The last middleware platforms involve advanced programing techniques related to Aspect-Oriented Programing. This research shows the development of a distributed systems middleware platform for the publishing/subscription model, we are pretending that our platform will be use for the research of advanced techniques for event based distributed systems. This paper shows the findings found into the standard development and adaptation toward the C# language programing, where feasibility of applied advanced programing techniques into the open source practical development of DDS (Data Distribution System) middleware is probed.

Keywords: Middleware, DDS, Real-Time, middleware, publish-subcribe, data.

1. INTRODUCCIÓN

Actualmente existen algunas soluciones propietarias de DDS (Data Distribution System); desde el año 2004 hay dos empresas encargadas de DDS, la empresa Real-Time Innovations[1] y la empresa Global

[1] https://www.rti.com/

PrismTech Corporation[2]; ambas han participado en las especificaciones aprobadas por el Object Management Group[3] (OMG) en un documento denominado Data Distribution Service for Real-time Systems. Sin embargo, estas soluciones al ser propietarias limitan la explotación y desarrollo de estos tipos de middleware.

Los lenguajes de programación han incorporado nuevas características que no han sido aprovechadas adecuadamente en el desarrollo e implementación de middleware. Por ejemplo incorporación de delegados, utilización de tareas y utilización de LINQ (Language Integrated Query).

Este trabajo de investigación se enfoca en aprovechar estas nuevas características con el objetivo de conseguir que los middleware publicación/suscripción sean más robustos y su utilización sea lo más transparente al usuario.

Este artículo está organizado de la siguiente manera: en el apartado 2, se da una perspectiva de los middleware de publicación/suscripción y los esfuerzos de implementación realizados a la fecha. El apartado 3 muestra nuestra propuesta de middleware, a la que hemos denominado DOOP.EC y su integración con el lenguaje de programación C#. El apartado 4 muestra las principales características del lenguaje C# que hemos aprovechado en la implementación del middleware. El apartado 5 se muestra las pruebas unitarias a las que fue sometido DOOP.EC, finalmente se muestra las principales conclusiones y trabajo futuro derivado del presente artículo.

2. ESTADO DE LA CUESTIÓN

Un Data Distribution Service para sistemas de tiempo real es una especificación para un middleware de tipo publicación/suscripción en sistemas distribuidos (Haoli & Yongming, 2012). Los DDS han sido creados para responder a las necesidades de la industria para estandarizar sistemas centrados en datos (data-centric systems). (Basanta-Val & Garcia-Valls, 2014).

En (Martínez del Valle, 2013) se realiza una comparación entre los estándares de comunicación SOAP (Simple Object Access Protocol), REST (REpresentation State Transfer) y DDS. La Tabla 1. muestra las características generales de los middleware y el grado de cumplimiento de los tres estándares; de lo cual se puede observar que DDS presta mejores servicios comparado con SOAP y REST.

Tabla 1. Comparativa general. (Martínez del Valle, 2013).

Comparativa	DDS	REST	SOAP
Interoperabilidad	2°	1°	2°
Independencia del fabricante	2°	1°	3°
Accesible en fuente	-	-	-
Flexibilidad	1°	2°	3°
Conocimientos	-	-	-
Code Churn	-	3°	-
Seguridad incluida	1°	2°	3°
QoS	1°	3°	2°
Sistema en tiempo real	1°	2°	3°
Tamaño del paquete	1°	2°	3°
Tiempo de envío	1°	2°	3°
Integración	-	-	-

En (Bellavista *et al*., 2013) se realiza una comparación del desempeño de implementación de OpenSplice y RTI (Real-Time), estos sistemas representan soluciones eficientes y de alto rendimiento

[2] http://www.prismtech.com/
[3] http://www.omg.org/

para la comunicación de datos en escenarios desafiantes, con requisitos de tiempo real (Pérez Tijero & Gutiérrez, 2012), tales como la gestión del tráfico aéreo, automatización industrial (Garcia Valls *et al*., 2013), smart grids y, las aplicaciones más recientes como son las financieras (Woochul *et al*., 2012). A pesar de que la última década ha sido testigo de la difusión y consolidación de algunas implementaciones importantes, todavía son muy pocos los estudios de análisis de rendimiento disponibles en la literatura. Para llenar ese vacío y a futuro facilitar procesos, en (Bellavista *et al*., 2013) se realiza un análisis minucioso de las implementaciones de DDS propuesto por PrismTech. Los resultados experimentales reportados señalan los pros y los contras de ambas soluciones en términos de rendimiento de la entrega de datos, también por evaluar precisamente los cuellos de botella, por ejemplo en términos de uso de recursos de CPU y memoria.

DDS no se diseñó teniendo en cuenta técnicas de programación ampliamente disponibles en la mayoría de los lenguajes actuales C#, Scala, Java. Por lo cual uno de los objetivos fundamentales de este proyecto es la oportunidad de trasladar los conceptos de DDS a los mecanismos que ofrece C# y sus técnicas avanzadas de programación. Pero este diseño enfrenta dos fuerzas que en cierto sentido son contrapuestas. Por una parte, el diseño del API (Application Programming Interface) en C# debe reflejar en gran medida el mismo estilo con las ya existentes de forma que un programador experimentado en DDS para Java no tuviera dificultades a la hora de enfrentarse al API en .Net. Pero por otra parte, C# dispone de recursos que no existen en los lenguajes mencionados. Por lo que el diseño del API debe tener en cuenta estas características e incluirlas en el diseño. Por ejemplo, en el diseño tradicional de DDS existe el concepto *listener*; cuando se crean ciertas estructuras (participante) es habitual pasar como parámetro un objeto *listener* que permite capturar eventos relacionados con dicha estructura. Pero en C# este concepto se suele implementar mediante delegados (referencias a métodos) y eventos. La capacidad de C# de utilizar Cálculo Lambda, el cual trata de expresar formalmente la computación basada en las funciones de abstracción y aplicación en la recepción de dichas notificaciones; hace poco atractiva la idea de utilizar el concepto tradicional de *listener*.

Para resolver esta dualidad, en nuestra propuesta hemos optado por ofrecer dos versiones del API. Una tradicional siguiendo el modelo ofrecido por los lenguajes ya normalizados y otra que extiende los métodos con mecanismos propios de C#. Ambos API en realidad suponen un único sistema de interface gracias a la utilización de las clases parciales de C# (permite dividir la declaración de una clase o interface entre varios ficheros).

3. PROPUESTA

3.1. Conceptos claves

El sistema DDS incorpora los siguientes conceptos:

- Agregación de delegados y eventos sustituyendo los conceptos relacionados con las notificaciones (*listener* o similares). Nótese que esta funcionalidad permite la incorporación de Cálculo Lambda.
- Utilización de tareas. Las versiones recientes de C# incorporan extensiones para la programación asíncrona y paralela. C# dispone de palabras reservadas a tal efecto (*async* y *await*). Un ejemplo de tal utilización se produce en los métodos de escritura o lectura de mensajes (*topics*). El programador puede realizar de forma asíncrona una lectura de mensaje. La llamada al método retorna la respuesta y continúa el proceso habitual.
- Utilización de tipos de valor (*structs*). Algunas de las clases definidas en Java realmente tienen una semántica asociada muy propia de tipos de valor. En cambio, en C# se pueden distinguir entre tipos referenciados (*class*) y tipos de valor (*strucs*). En algunos casos hemos redefinido el API para tener en cuenta estos casos.
- Utilización de LINQ y árboles de expresiones. El estándar DDS ofrece la posibilidad de definir filtros y *queries* para el intercambio de datos. De esta forma, un subscriptor puede limitar el volumen de información que recibe. El estándar define un lenguaje de *queries* muy cercano en sintaxis a SQL (Structured Query Language). El programador aplica este tipo de

filtros usando cadenas de *strings* como argumentos de funciones. Nuestra propuesta es utilizar además técnicas basadas en métodos extensores y árboles de expresiones. Ambas son técnicas recientes en C# y que son la base de LINQ.

3.2. Requisitos

Para el diseño de DOOP.EC se ha tenido en cuenta los siguientes requisitos:

En primer lugar, y por la naturaleza del middleware como plataforma en tiempo real, es necesario disponer de un subsistema de comunicaciones que tenga un rendimiento y una capacidad de escalabilidad a niveles extremadamente altos.

En segundo lugar, se desea que Doop.ec sea una plataforma para la investigación de nuevos protocolos. Es por ello que se determinó como requisito que el sistema de comunicaciones subyacente fuera lo suficientemente abierto y flexible para poder añadir nuevos protocolos o mecanismos de transporte en el futuro sin que la arquitectura se viera afectada.

Con estos requisitos en mente, se optó por utilizar la versión de C# de la librería Apache Mina[4]. Esta librería es un framework para la programación de comunicaciones de alto rendimiento y gran escalabilidad. Dicha librería ya soporta transportes basados en TCP/IP, UDP/IP, RS232, entre otros. Además, mediante un mecanismo de apilamiento modular es factible añadir nuevos protocolos o incluir nuevos procesos (por ejemplo: compresión) dentro de un protocolo existente.

Otra de las razones que nos llevó a utilizar esta librería fue el soporte a sockets asíncronos; es conocido que la utilización de hebras (*threads*) para la escritura y lectura de mensajes suele ser un problema si no se diseña la política adecuada. Utilizar una única hebra para el proceso de mensajes suele ser un cuello de botella en sistemas de alto rendimiento. Por el contrario, generar hebras por cada cliente suele ser un problema si el número de éstos es alto o la entrada salida de nuevos clientes es continua. Una solución es utilizar un pool de hebras pero la gestión de tal sistema suele ser muy compleja. Nuestra solución pasa por utilizar los mecanismos asíncronos y de tareas (hebras ligeras) que aporta C# y de la que hace uso la versión en C# de Mina; permitiendo adaptar la política de concurrencia a la carga del sistema; desde utilizar una única hebra hasta la solución paralela que aporta .Net.

Otro de los aspectos que han resultado clave para el rendimiento del sistema es la utilización clases específica para el tratamiento de buffers que reducen la carga habitual en la copia de los mensajes. Se han usado los mismos conceptos que utiliza IObuffer en Java.

Finalmente, la integración de los serializadores con el sistema de comunicaciones ha supuesto una tarea crítica pero que ha sido posible gracias al diseño modular de Mina. El sistema de serialización se ha integrado en la librería como un codecs adicional (un codec es un módulo de Mina que permite la serialización y deserialización de los mensajes de un protocolo). Para demostrar la flexibilidad del diseño del sistema de comunicaciones, se ha realizado e integrado el protocolo RTPS.

3.3. Módulos

Nuestra propuesta está formada por cinco módulos como lo muestra la Figura 1.

- **Módulo de Infraestructura:** define las clases abstractas y las interfaces que son modificadas por los otros módulos. También provee soporte para los dos estilos de interacción (basado en notificaciones y basado en espera) con el middleware.
- **Módulo de Infraestructura:** define las clases abstractas y las interfaces que son modificadas por los otros módulos. También provee soporte para los dos estilos de interacción (basado en notificaciones y basado en espera) con el middleware.
- **Módulo de Dominio:** contiene la clase de *DDS.Domain*: que actúa como un punto de entrada del servicio y actúa como un automatizador para muchas de las clases. La clase *DDS.DomainParticipant* actúa también como un contenedor para otros objetos que componen el servicio.

[4] https://github.com/longshine/Mina.NET

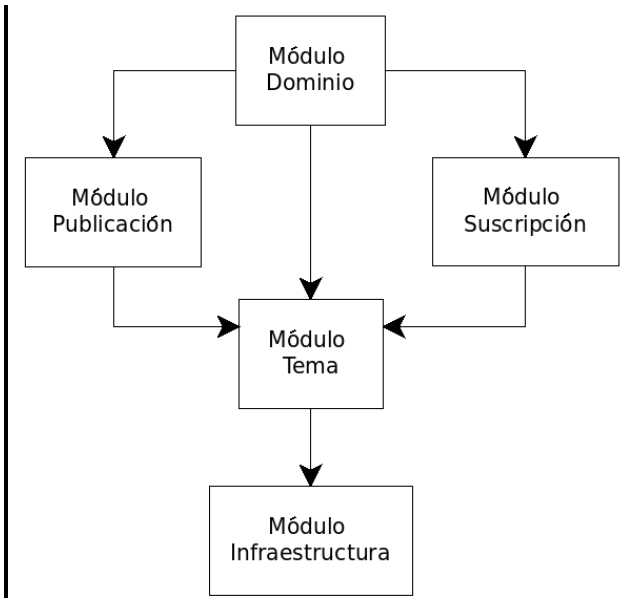

Figura 1. Módulos del DDS y sus dependencias.

- **Módulo de Tema:** contiene la clase *DDS.Topic*, la interfaz *DDS.TopicListener*, y más generalidades, todo lo que se necesita para la aplicación es definir los objetos DDS.Topic y adjuntar las políticas de calidad de servicio de los mismos.
- **Módulo de Publicación:** Contiene las clases *DDS.Publisher* y *DDS.DataWriter* así como también las interfaces *DDS.PublisherListener* y *DDS.DataWriterListener*, y más generalidades, todos esto es necesario en el nodo publicador
- **Módulo Suscriptor:** contiene las clases *DDS.Subscriber*, *DDS.DataReader*, *DDS.ReadCondition* y *DDS.QueryCondition*, también contiene las interfaces *DDS.SubscriberListener* y *DDS.DataReaderListener*, y más generalidades que en conjunto con esto son necesarias en el nodo suscriptor.

Durante el desarrollo del API, se han presentado algunos problemas al momento de adaptarlo al lenguaje de programación C#, los mismos que se describen a continuación:

- Problemas al revisar y definir correctamente las excepciones apropiadas para el API DDS.
- Al revisar el patrón listener actual del API DDS. Se corresponde con el modelo Java pero C# tiene eventos y delegados que hay que identificar para mejorar el API.
- Al revisar las marcas de serializable, cloneable y similares. En c# tiene implicaciones diferentes a la de Java.

4. EVALUACIÓN

La evaluación del modelo propuesto se realizó a través de pruebas unitarias; las pruebas se realizaron en diferentes circunstancias a los métodos que conforman los API DDS. Dichas pruebas unitarias se centraron en la validación de las políticas de calidad de servicio que debe ofrecer una aplicación en tiempo real de acuerdo a la comunidad RTI[5] (DDS Community, RTI Conecct Users), se presenta en la

[5] https://community.rti.com/rti-doc/500/ndds.5.0.0/doc/pdf/RTI_CoreLibrariesAndUtilities_QoS_Reference_Guide.pdf

Tabla 2, y considerando la aplicación de políticas de QoS desarrolladas en (López Vega, *et al*., 2010). Los métodos sujetos a pruebas fueron: CORE, DOMAIN, PUB, SUB, TOPIC, TYPE. Las pruebas fueron superadas al 100% con éxito.

5. CONCLUSIONES

Como resultado de este trabajo se implementó un nuevo middleware de tipo DDS, al que denominamos DOOP.EC, que integra las características descritas en la sección 3. DOOP.EC ha demostrado ser un middleware que cumple con las principales características de un middleware DDS.

Con este desarrollo se han explorado nuevas áreas de investigación entre las cuales podemos mencionar:

- Implementar el API en un sistema real para una correcta validación y verificaciones del sistema, dentro del área de la salud, robótica, redes de comunicación, etc.
- Estudiar modelo de programación paralela y tratar de aplicarlo a la API, para implementar una mayor tolerancia a fallos y mejorar su rendimiento.
- Ampliar las funcionalidades implementando mejoras a nivel de la codificación del API DDS, de manera que se pueda establecer controles de calidad en las publicaciones y suscripciones de los diferentes temas (Topics).
- Utilizar el protocolo de comunicación RTPS para el envío de los datos a través de la red.

Tabla 2. Detalle de pruebas unitarias realizadas sobre Doop.ec.

Políticas QoS	Descripción	Entidad DDS
Presentation	Controla la presentación de datos recibidos a DataReader	Publisher, Subscriber
Entity_Factory	Controla el comportamiento de una entidad como un factor para otras entidades	DomainParticipant Factory, DomainParticipant, Publisher, Subscriber
User_Data	Permite la entrega de información adicional en el descubrimiento de entidades de usuarios	DomaintParticipant, DataReader, DataWriter
Topic_Data	Entrega información adicional de descubrimiento	Topic
Deadline	Separación máxima entre actualizaciones del mismo tópico	Topic, DataReader, DataWriter
Durability	Especifica si el contexto se almacenará y entregará previamente a un dato publicado	Topic, DataReader, DataWriter
Group_Data	Establece información adicional para grupos de datos	Publisher, Subscriber
Partition	Ejecuta la división lógica de dominios	Publisher, Subscriber
Destination_Order	Controla las entregas con datos de múltiples *DataWriters*	DataReader, DataWriter
History	Controla cuantos y como se almacenan los datos	DataReader, DataWriter
Políticas QoS	Descripción	Entidad DDS
Latency_Budget	Especifica el tiempo permitido para la entrega de datos	DataReader, DataWriter
Liveliness	Verifica actividad de cada entidad	DataReader, DataWriter
Ownership	Establece que entidad puede actualizar los datos	DataReader, DataWriter
Realibity	Indica si las muestras perdidas en la red deberían ser reparadas por el middleware.	DataReader, DataWriter
Resource_Limits	Limita la cantidad de datos almacenados en caché	DataReader, DataWriter
Reader_Data_Lifecycle	Controla como un DataReader administra los ciclos de vida de los datos que se han recibido	DataReader

Time_Based_Filter	Número máximo de muestras entregadas por segundo	DataReader
Durability_Service	Configura conexiones con durabilidad persistente o transitoria	DataWriter
Writer_Data_Lifecycle	Controla como un DataWriter maneja el ciclo de vida de una instancia	DataWriter
Lifespan	Tiempo de expiración de la muestra de datos.	DataWriter
Ownership_Strength	Asigna puntuación a entidades para actualizar datos	DataWriter
Transport_Priority	Especifica la prioridad en una par de DataWriter	DataWriter

AGRADECIMIENTOS

Este trabajo se ha podido realizar gracias al financiamiento del proyecto CEPRA VII-2014-06, titulado: Middleware en tiempo real basado en el modelo publicación/suscripción (http://www.utpl.edu.ec/proyectomiddleware/).

REFERENCIAS

Basanta-Val, P., M. Garcia-Valls, 2014. A Distributed Real-Time Java-Centric Architecture for Industrial Systems. *IEEE Transactions on Industrial Informatics*, 10(1), 27-34.

Bellavista, P., A. Corradi, L. Foschini, A. Pernafini, 2013. Data Distribution Service (DDS): A performance comparison of OpenSplice and RTI implementations. *IEEE Symposium on Computers and Communications (ISCC)* (págs. 377-383). Bologna: IEEE.

Garcia Valls, M., I. Lopez, L. Villar, 2013. iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time Systems. *IEEE Transactions on Industrial Informatics*, 9(1), 228-236.

Haoli, R., G. Yongming, 2012. A Study on the Distributed Real-time System Middleware Based on the DDS. *International Conference of Modern Computer Science and Applications* (págs. 1-6). Springer .

López Vega, J.M., J. Povedano Molina, J. Sánchez Monedero, 2010. Políticas de QoS en una Plataforma de Trabajo Colaborativo sobre Middleware DDS. *Jornadas de Tiempo Real.*

Martínez del Valle, B, 2013. *Sistema de comunicaciones de altas prestaciones basado en DDS*. Obtenido de Universidad Carlos III de Madrid: http://e-archivo.uc3m.es/handle/10016/19178.

Pérez Tijero, H., J. Gutiérrez, 2012. On the schedulability of a data-centric real-time distribution middleware. *Computer Standards & Interfaces*, 34(1), 203-211.

Woochul, K., K. Kapitanova, H.S. Sang, 2012. RDDS: A Real-Time Data Distribution Service for Cyber-Physical Systems. *IEEE Transactions on Industrial Informatics*, 8(2), 393-405.