Proyecto Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Estudio de la arquitectura DDS y diseño de una solución aplicada sobre arquitectura genérica de vehículos

Autor: Rafael Pajuelo Durán

Tutores: Ramón González Carvajal y
Eduardo Hildalgo Fort

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

Estudio de la arquitectura DDS y diseño de una solución aplicada sobre arquitectura genérica de vehículos

Autor:
Rafael Pajuelo Durán

Tutores:
Ramón Gonzalez Carvajal
Catedrático de Universidad
Eduardo Hidalgo Fort
Investigador postdoctoral

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Grado: Estudio de la arquitectura DDS y diseño de una solución aplicada sobre arquitectura genérica de vehículos

Autor: Rafael Pajuelo Durán

Tutores: Ramón Gonzalez Carvajal
Eduardo Hidalgo Fort

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

En primer lugar, me gustaría dar las gracias a Edu y a Ramón por darme la oportunidad de colaborar con el departamento de electrónica en este proyecto y por las atenciones recibidas.

También quiero agradecer a toda mi familia por estar ahí siempre, dándome todo su apoyo y cariño en las peores de las circunstancias.

Tampoco podría olvidarte de todos aquellos amigos y compañeros que a lo largo de mi vida han estado ahí en todo momento para apoyarme y acompañarme en toda clase de aventuras.

A todos, muchas gracias.

Rafael Pajuelo Durán

Sevilla, 2021

Resumen

En este proyecto se pretende establecer una red de comunicaciones en el entorno de un vehículo genérico, basado en tecnologías distribuidas. Para ello se realizará un estudio del estándar DDS para el recibo y envío de datos en tiempo real. Una vez hecho el análisis teórico se pasará al diseño e implementación de aplicaciones que puedan correr en un sistema embebido basado en microprocesador. Es por ello por lo que se utilizará el lenguaje de programación C++11 corriendo sobre Linux.

Tras el desarrollo se procederá a realizar una demostración en la que varias señales relacionadas con el cuadro de indicadores del vehículo y sus alertas de mantenimiento serán monitorizadas y registradas haciendo uso del sofware RTI Connext DDS.

Abstract

This project aims to establish a communications network in the environment of a generic vehicle, based on distributed technologies. To do this, a study of the DDS standard will be carried out for receiving and sending data in real time. Once the theoretical analysis is done, the design and implementation of applications that can be executed in an embedded system based on a microprocessor will proceed. That is why the C ++ 11 programming language will be used running on Linux.

After the development, a demonstration will be carried out in which various signals related to the vehicle's dashboard and its maintenance alerts will be monitored and logged using the RTI Connext DDS software.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xiv**

Índice de Tablas **xvi**

Índice de Figuras **xviii**

1 Introducción **1**
1.1. Sistemas distribuidos *1*
1.1.1 Sistemas distribuidos en tiempo real 2
1.2. Servicio de Distribución de Datos (Data Distribution Service), DDS *3*
1.3. Objetivos *3*

2 Arquitectura **6**
2.1. DCPS (Data-Centric Publish-Subscribe) *6*
2.1.1 GDS (Global Data Space) 8
2.1.2 Publisher (Publicador) 9
2.1.3 Subscriber (Subscriptor) 9
2.1.4 Topic 10
2.1.5 Mecanismos de alcance 11

3 Descripción **13**
3.1. Common *16*
3.2. Dominio Instrumentación *17*
3.3. Dominio Alertas *18*

4 Implementación **22**
4.1. Publicación *22*
4.2. Subscripción *24*
4.3. Clases Patrón *26*
4.1.1 DDSListener 26
4.1.2 PeriodicUniqueReader 27
4.1.3 DDSBuilder 28
4.1.4 SpecificationWriter 28

5 Demostración **30**
5.1. Dominio Instrumentación *30*
5.2. Dominio Alertas *32*

6 Conclusiones **37**

Glosario **39**

Bibliografía **41**

ÍNDICE DE TABLAS

Tabla I. Algunos tipos de datos admitidos para IDL en Modern C++. 10
Tabla II. Operadores para Filtros DDS. 11
Tabla III. Argumentos de la clase DomainParticipant. 22
Tabla IV.Argumentos de la clase Topic. 23
Tabla V. Argumentos de la clase DataWriter. 23
Tabla VI. Argumentos de la clase DataReader. 25

ÍNDICE DE FIGURAS

Figura 1. Espacio de comunicaciones electrónicas en un vehículo genérico. 4
Figura 2. Arquitectura DDS en un vehículo autónomo. DDS in Autonomous Car Design, 2016, RTI. 6
Figura 3. Comparativa entre modelos centrados en datos y modelos centrados en mensajes. 7
Figura 4. Módulos de un DCPS. 7
Figura 5. Representación del Espacio Global de Datos (GDS). 8
Figura 6. Entidades y flujo de la información en DDS. 9
Figura 7. Directorios de trabajo del proyecto. 13
Figura 8. Trama DDS para datos de tipo Dashboaard. 15
Figura 9.Trama DDS para datos de tipo serviceAlert. 15
Figura 10. Diagrama de clases genérico del proyecto. 16
Figura 11. Diagrama de clases extendido de las clases patrón. 16
Figura 12. Diagrama de la clase "XML". 17
Figura 13. Dominio Instrumentación visto a través del RTI Administration Console. 17
Figura 14. Diagrama de flujo de los subscriptores y publicadores en el dominio Instrumentación. 18
Figura 15. Topic "Cambio_aceite" en el dominio Alertas. 18
Figura 16. Topic "Instrumentación" dentro del dominio Alertas. 19
Figura 17. Diagrama de flujo de los publicadores y subscriptores en el dominio Alertas. 19
Figura 18. Ejecución de las aplicaciones "temperatura_motor" e "Instrumentacion_Main". 30
Figura 19. Resultados del envío de la temperatura del motor a través de DDS. 31
Figura 20. Instrucciones para ejecutar todas las aplicaciones del dominio Instrumentación. 31
Figura 21. Resultado de la ejecución de todas las aplicaciones en el dominio Instrumentación. 32
Figura 22. Instrucciones para simular una alerta de cambio de aceite. 32
Figura 23. Registro del kilometraje para el cambio de aceite. 33
Figura 24. Registro instantáneo de estados de alerta del cambio de aceite. 33
Figura 25. Registro de alertas de cambio de aceite. 34
Figura 26. Instrucciones para la ejecución de todas las aplicaciones en el dominio Alertas. 34
Figura 27. Registro instantáneo de las alertas de revisión general del vehículo. 35
Figura 28. Registro de eventos de alertas de servicio. 35

1 INTRODUCCIÓN

El sector industrial se enfrenta constantemente a toda clase de retos que buscan mejorar los procesos de producción y todos sus derivados, de forma que estos logren una mayor calidad y adquieran las funcionalidades deseadas con el menor coste posible. Esto no es ningún secreto, pues aquellas empresas que avanzan más rápido por este camino son las que más probabilidades tienen de encontrar el éxito.

Dichas mejoras pueden llevarse a cabo en la logística o en el estudio y utilización de ciertos materiales, pero también, en la introducción de nuevas tecnologías basadas en sistemas de tiempo real (RTS). Estas tecnologías son útiles para la monitorización, control y automatización de todos los elementos implicados en cualquier prestación de servicio o proceso de desarrollo del producto. Un claro ejemplo de un RTS muy común es el autómata programable (PLC).

El correcto funcionamiento de estos sistemas recae sobre la capacidad de respuesta en tiempo limitado ante ciertos eventos. Los sistemas funcionarán correctamente si son capaces de realizar respuestas dentro de ciertos límites temporales, mientras que, en caso contrario, su comportamiento será indeseado.

Atendiendo a esto último, cabe destacar que se pueden encontrar dos tipos de RTS:

- **Hard RTS**: donde la vulneración del *deadline* provoca el funcionamiento incorrecto del sistema. Un ejemplo de ello es el caso del sistema de frenado ABS.
- **Soft RTS**: donde la vulneración del *deadline* no implica un funcionamiento incorrecto, pero sí una pérdida de calidad de servicio. Ejemplo de ello es el caso de un reproductor DVD.

De un tipo u otro, los sistemas en tiempo real se encuentran definidos por la cantidad de **eventos extrernos** que pueda admitir, el **tipo de respuesta** que ejecuta tras cada evento y sus respectivos **requerimientos temporales.**

Para la especificación de los requerimientos temporales es habitual hacer uso de diagramas secuencia u otros modelos de computación y lenguajes de modelado como es el caso de los diagramas UML. Luego, para hacer cumplimiento de todos los requisitos de un sistema en tiempo real también es habitual hacer uso de sistemas operativos en tiempo real (RTOS), los cuales se encargan de la planificación, envío y gestión de memoria, propios en un sistema en tiempo real.

Los RTS cuentan comúnmente con configuraciones o ficheros que definen la calidad de servicio del sistema (QoS), constatando cómo ha de ser el comportamiento del sistema. Si se incumple una de estas QoS se provocará un fallo en el sistema y se tendrá que solventar.

Normalmente no se suele integrar un sistema por sí solo, si no que este se encontrará rodeado de más sistemas satélites que intercambian información entre sí y dependiendo de la finalidad última de este conjunto se puede hacer distinción entre **sistemas distribuidos** y **sistemas centralizados**.

Debido al gran auge que está teniendo el IoT en la industria parece tener más sentido que cada vez se busquen soluciones basadas en tecnologías distribuidadas, con alta escalabilidad y confiabilidad.

1.1. Sistemas distribuidos

Los sistemas distribuidos cuentan con varios nodos que actúan como un todo, por lo que son capaces de proporcionar cuantiosas ventajas entre las que se destacan la **alta capacidad de procesamiento**, mayor **tolerancia a fallos**, mayor e**scalabilidad** y mayor **eficiencia** que los sistemas centralizados.

Para remarcar la importancia de este tipo de sistemas es conveniente indagar en una serie de características propias a tener en cuenta. Estas son:

- **Accesibilidad**. Dado el alto número de dispositivos o aplicaciones en la red, se hace necesaria la transparencia del sistema a la hora de representar de datos y tener acceso a recursos.
- **Localización**. Es interesante que la comunicación se establezca desde la propia aplicación cliente, quedando desligada del nodo o nodos en los que se encuentran los datos solicitados.
- **Migración**. Los datos existentes en un nodo pueden ser replicados a otro sin tener que afectar a la aplicación cliente.
- **Relocalización**. A medida que se establece la comunicación entre pares es lógico que se reasignen recursos a diferentes nodos durante el tiempo de ejecución, pues esto es transparente al desarrollador.
- **Sincronización y concurrencia**. Las aplicaciones clientes tienen que ser capaces de compartir recursos, dando consistencia a los datos.
- **Replicación de recursos**. La información puede fluir entre diferentes nodos sin afectar a las aplicaciones clientes.
- **Fallo**. Las aplicaciones clientes quedan ocultas de todo fallo en el sistema, de forma que es este el que se ha de recuperar de errores y volver a un estado consistente.
- **Persistencia**. La forma en la que se accede y se almacena la información queda occulta al usuario. Esta información suele encontrarse en bases de datos, discos duros, memorias USBs, etc.

Por lo general este tipo de sistemas poseen una gran capacidad de abstracción, sin embargo, esta misma capacidad puede llegar a presentarse como inconveniente puesto que se introduce una cierta latencia que, según la tarea que se quiera llevar a cabo con esta tecnología, puede suponer un auténtico problema.

1.1.1 Sistemas distribuidos en tiempo real

Para que un sistema distribuido sea considerado en tiempo real no basta con que las aplicaciones trabajen en tiempo real, si no que además, la información a la cual acceden que estas aplicaciones dentro del sistema distribuido también ha de serlo. Por tanto, puede darse el caso en que únicamente las aplicaciones cliente trabajen en tiempo real, por lo que no se considera al sistema como tal.

Llegados al caso, hay que atender a ciertos aspectos dentro de estos sistemas para considerarlos o adaptarlos a uno que trabaje en tiempo real. Cualquier aplicación en tiempo real debe de tener un comportamiento predecible, de forma que no exista ningín indeterminismo que tenga que ser resuelto en el tiempo de ejecución. Cualquier sistema que se comporte de forma indeterminista consumirá recursos y tiempo, y esto se verá reflejado en la introducción de retrasos en el tiempo ejecución afectando a los *deadlines* de cada evento. Bajo ciertas circunstancias críticas se pueden llegar a cometer errores y/o un mal funcionamiento el sistema completo, por lo que no es de extrañar que estos sistemas deban de contar con una mayor limitación en sus características que un sistema distribuido asecas.

Un sistema distribuido en sí no es más que un sistema no determinista, dado que se abstraen a las aplicaciones cliente del funcionamiento de la red de nodos. Estos sistemas cuentan con un *middleware*, que se encuentra estandarizado por la OMG para la abstracción de sistemas. El *middleware* es el encargado de combatir estos no determinismos, como pueden ser la localización de recursos o la migración.

Tal como se indicó anteriormente, los sistemas distribuidos presentan una serie de características que otorgan un cierto nivel de abstracción a costa de introducir una cierta latencia a la hora de acceder a los datos. Es por ello que, un sistema distribuido en tiempo real ha de prescindir de características como la **localización**, **migración** y **relocalización**. Todo esto ocurre por la necesidad de tener localizado en todo momento los recursos dentro de la red.

Otros aspectos de menor importancia que varían en los sistemas en tiempo real son el servicio de nombres que se suele sustituir por un sistema más sencillo y cerrado, la integración de mecanismos de control de sobrepaso del tiempo de ejecución o la simplicidad de los elementos de seguridad del sistema al encontrarse en sistemas normalmente cerrados.

Por otro lado ganan importancia el modelo de concurrencia, la planificación y la sincronización; que se hace más compleja si se ha de acceder a recursos remotos. Además, aparecen nuevos aspectos a tener en cuenta como son el uso de redes de comunicación predecibles, sincronización de relojes o la inclusion de nuevas calidades de servicio.

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

Como ya se indicaba anteriormente, es común utilizar un *middleware* para llevar a cabo la realidad de un sistema distribuido en su conjunto. *Middlewares* hay muchos, pero ninguno de ellos es nada si no sigue un protocolo. Al igual que uno los protocolos más conocidos para sistemas no distribuidos dentro del IoT, como es MQTT, se puede encontrar un concepto similar en uno de los protocolos estándar que define el OMG para sistemas distribuidos en tiempo real: DDS (*Data Distribution Service for real-time systems*), el cual proporciona una solución al paradigma centrado en datos para sistemas distribuidos. En esta solución existe un entorno virtual denominado *Global Data Space* (GDS), donde aplicaciones se unen y abandonan dicho espacio cada vez que quieran compartir información. Esta comunicación sigue una arquitectura *publish-subscriber* en la que se identifica un *topic* (es el objeto que hace de portador de datos), y según si es tramitado mediante el publicador o el subscriptor dichos datos son enviados o recibidos respectivamente.

DDS fue creado en el año 2011 para dar solución a las necesidades de la industria de estandarizar sistemas centrados en datos (data-centric systems). Todavía sigue en evolución, y un ejemplo de ello son los denominados X-Topics. Dichos topics pueden ser definidos en tiempo de ejecución y pueden no solo ser definidos por el lenguaje de especificación IDL (Lenguaje de Definición de Interfaces) como se hacía hasta el momento, sino que también se podría hacer uso de XMLs (Lenguaje de Marcas Generalizado) o XSDs (Lenguaje de Esquema XML). Otras posibles inclusiones en el estándar es la integración de tecnologías web como HTTP, clientes SOAP (Protocolo de Acceso a Objeto Simple) o REST (Transferencia de Estado Representacional).

La utilización del estándar DDS hace que exista la abstracción necesaria requerida en los sistemas distribuidos en tiempo real que anteriormente se comentaba en este documento, de forma que los terminales de red que participan en la comunicación son transparentes para el programador, haciendo irrelevante el número de equipos involucrados en el sistema, lo que lo hace altamente escalable.

DDS es considerado como el primer protocolo que realmente lleva a cabo una comunicación distribuida en tiempo real, que cumple con las características descritas en el apartado "1.1.1". Y dado que su objetivo principal es establecer un método eficaz para la comunicación entre procesos que funcionan en tiempo real en sistemas embebidos, hace que su utilización sea indispensable para el propósito del proyecto.

1.3. Objetivos

El proyecto busca comunicar los diferentes sensores y actuadores presentes en un vehículo genérico, como podría ser un coche; y en el que participarían varias unidades de control electrónico (ECUs) como podría ser una unidad de control motor. La información tomada de estas diferentes unidades de control será monitorizada y registrada para su posterior tratamiento, haciendo uso de aplicaciónes DDS que correrán sobre un RTOS.

Como DDS es un estándar, existen numerosas compañías que venden soluciones software (librerías) que lo incorporan. Ejemplo de alguna de estas herramientas son la MilSOFT DDS Middleware (Milsoft), Vortex OpenSplice DDS (PrismTech) o la usada para la realización del presente proyecto: RTI DDS (Real-Time Innovations). También existen versiones de software libre como la llamada OpenDDS.

La elección de uso de RTI DDS se ha llevado a cabo por la facilidad de uso en sistemas embebidos y sistemas operativos en tiempo real, además de poseer descubrimiento vía multicast (realmente *peer-to-peer*) y no depender del uso de demonios informáticos (*DAEMONs*) ejecutándose en segundo plano.

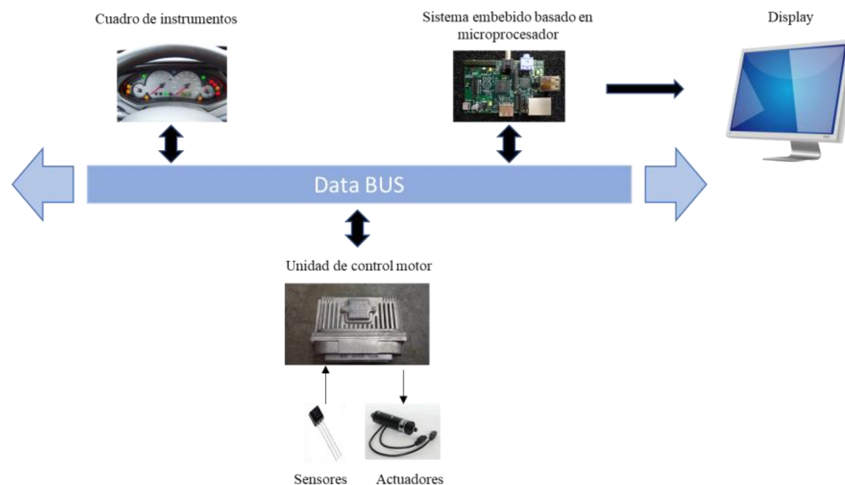

Figura 1. Espacio de comunicaciones electrónicas en un vehículo genérico.

El entorno sobre el que centra este proyecto es mostrado en la *Figura 1*, donde se tienen varias ECUs (solo se muestran la unidad de control motor y la asociada al cuadro de instrumentos), que obtienen lecturas de los diferentes sensores distribuidos en el vehículo y aplican acciones a sus respectivos actuadores según les sean necesarias. Estas unidades de control se encuentran interconectadas por un bus de datos que bien podría tratarse del bus CAN y/o bus DDS, por el que ambos dispositivos entablan comunicación con el sistema embebido en el que se halla el RTOS. Aunque CAN y DDS pueden convivir juntos, es necesario que se aplique alguna especie de pasarela que convierta los mensajes CAN a datagrmas DDS, ya que el sistema embebido solo es compatible con DDS. Los datos serán analizados y procesados por el sistema embebido. La idea es que el usuario tenga la posibilidad de acceder a los registros de estos datos y visualizarlos en algún dispositivo externo como podria ser un ordenador portátil. La realización de esta tarea requiere de la creación de inferfaces gráficas y métodos de exportación, por lo que escapa del alcance del proyecto.

CAN es un protocolo que fue creado y es utilizado para permitir que dispositivos y microcontroladores se comuniquen entre sí dentro de un vehículo **sin una computadora host**, basado en el envío de mensajes bajo ciertas prioridades. Estas funcionalidades siguen siendo replicadas en DDS, pero dada su arquitectura, es capaz de **eliminar** la posibilidad de que existan posibles **retardos en la comunicación** que sí pueden darse a través del CAN (debido al alto número de elementos que intercambian datos del vehículo al mismo tiempo). La inclusion de DDS otorga, por lo tanto, un plus de confiabilidad a los lazos de control de los sistemas y permite una mayor escalabilidad de estos.

Para el desarrollo del proyecto se hará un estudio de la propia arquitectura de DDS y se pondrá en marcha la implementacion de una serie de aplicaciones que lleven a cabo la monitorización y registro de las señales del vehículo. Se tomarán lecturas de la instrumentación (**velocidad instantánea del vehículo**, **par motor**, **nivel de combustible**, **consumo instantáneo** y **temperatura del motor**), y se definirán ciertas alertas de mantenimiento (**sustitución del filtro de combustible**, **cambio de aceite**, **revision general del vehículo** y **reemplazo de las pastillas de freno**). El registro de todas estas señales se almacenarán en ficheros de texto, pero se deja la posibilidad de su incorporación a una base de datos. El desarrollo software se llevará a cabo en el lenguaje de programación Modern C++ 11, corriendo sobre Linux, quedando disponible para su integración en un sistema embebido basado en microprocesador.

Como apunte, cabe remarcar que señales como la del comsumo instantáneo de combustible pueden no llegar directamente desde la instrumentación en un caso real, pero se tratará de igual forma.

2 ARQUITECTURA

RTI brinda arquitecturas de diseño afines a la motivación del proyecto, donde se puede encontrar esquematizado los diferentes bloques que toman parte en la comunicación y conectividad de un vehículo autónomo. Este proyecto sigue la misma arquitectura, sólo que en lugar de tomar lecturas de sensores como el radar o LIDAR se toman otra clase de lecturas anteriormente ya comentadas. El resto de bloques mostrados en la *Figura 2* pueden no aparecer en la arquitectura de otros vehículos, como es el caso del bloque encargado del análisis del tráfico. Es por esto, y por el amplio alcance que tendría desarrollar todos y cada uno de estos bloques, que se van a focalizar los recursos disponibles en realizar únicamente un bloque dedicado a "Monitoring & Logging" con el cual cualquier arquitectura de vehículos debería de contar.

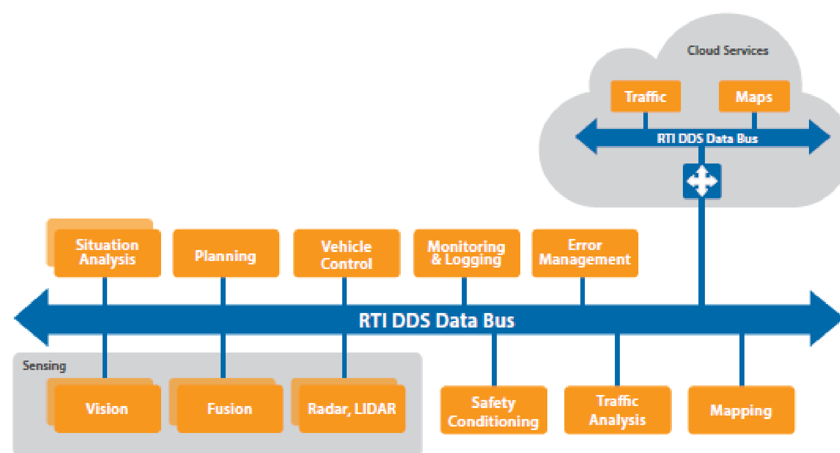

Figura 2. Arquitectura DDS en un vehículo autónomo. DDS in Autonomous Car Design, 2016, RTI.

Dada dicha arquitectura, parece evidente que el proyecto se encuentre ampliamente fundamentado en DDS. Es por esto que se hace necesario el análisis detallado de la propia arquitectura del estándar, para lograr así una mayor comprensión de lo que se realiza en el proyecto.

DDS presenta dos niveles de interfaces. El primero de estos niveles es el DLRL (Data Local Reconstruction Layer). Este se ubica en el nivel superior y tiene como objetivo permitir realizar una integración simple de DDS en la capa de aplicación. La mayoría de las veces es innecesario su uso, por lo que se hace más relevante hablar del segundo de estos dos niveles: el DCPS (Data-Centric Publish-Subscribe).

2.1. DCPS (Data-Centric Publish-Subscribe)

El DCPS se encuentra en el nivel inferior y su objetivo es que la información sea intercambiada de forma eficiente entre los diferentes receptores. Aquí radica la clave de DDS, ya que al contrario que ocurre con los modelos de conectividad centrados en mensajes, el DCPS ofrece una mayor modularidad, simplicidad y escalabilidad.

Mientras que en otros modelos de conectividad es la aplicación la que tiene que decidir sobre el parseo, filtrado, generación de mensajes y almacenamiento de la caché; DDS ya lo hace por sí mismo, restando complejidad a la aplicación que se sitúe por encima suya.

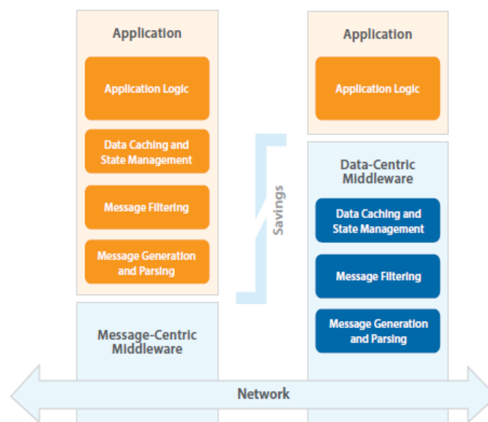

Figura 3. Comparativa entre modelos centrados en datos y modelos centrados en mensajes.

DDS in Autonomous Car Design, 2016, RTI.

Dentro del DCPS, se puede hacer distinción entre una serie de módulos, que compondrían su arquitectura:

- **Módulo de infraestructura**. Define las clases abstractas y las interfaces que son modificadadas por el resto de los módulos. Proporciona soporte para dos tipos de interacción con el middleware, una basada en notificaciones y otra en espera.
- **Módulo de tema**. Contiene todo lo necesario por la aplicación para definir los objetos ***Topic*** y adjuntar las políticas de calidad de servicio para los mismos. En este módulo destacan la clase ***Topic*** en sí, y la interfaz ***TopicListener***.
- **Módulo de dominio**. Contiene la clase ***DomainParticipant***, siendo el punto de entrada al Servicio. Actúa como fábrica de otras clases además de de actuar como contenedor de otros objetos.
- **Módulo de publicación**. Contiene todo lo que se necesita en el lado de la publicación. En él se destacan las clases ***Publisher*** y ***DataWriter***, así como las interfaces ***PublisherListener*** y ***DataWriterListener***.
- **Módulo de subscripción**. Contiene todo lo que se necesita en el lado de la subscripción. En él se destacan las clases ***Subscriber***, ***DataReader***, ***ReadCondition***, y ***QueryCondition***, así como las interfaces ***SubscriberListener*** y ***DataReaderListener***.

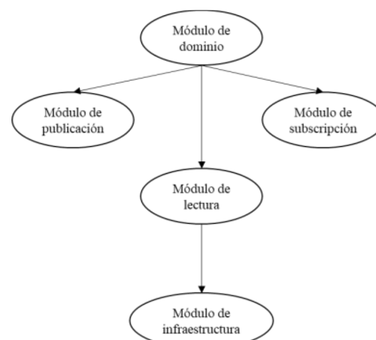

Figura 4. Módulos de un DCPS.

Cabe decir una vez más, que gracias al empleo de esta arquitectura se reduce el acoplamiento entre entidades (simplemente por el hecho que sea una arquitectura publicador/subscriptor ya se gana mucho), es fácilmente adaptable gracias al descubrimiento automático de los pares, es eficiente debido a que se establece comunicación directa entre publicador y subscriptor, es determinista, escalable, altamente parametrizable gracias al manejo de las calidades de servicio y además es independiente de la plataforma en la que se desarrolle, gracias al empleo

de estándares como IDL anteriormente comentado.

2.1.1 GDS (Global Data Space)

Lo primero de lo que se tiene que hablar a la hora de describir el funcionamiento de DDS es del Espacio Global de Datos (GDS). Este representa la abstracción clave que hay que entender para llegar a implementar DDS en un entorno distribuido evitando la introducción de puntos particulares de fallo o de cuello de botella. El GDS es el encargado de realizar el multicast, de definir los tipos de datos y de propagarlos.

La cualidad intrínseca del GDS es el propio descubrimiento dinámico que integra. La conexión de cualquier equipo a la red no supone la configuración de ningún parámetro, ya que todo será descubierto automáticamente y los datos comenzarán a moverse por este espacio virtual.

Otra característica importante que presenta la existencia de un GDS es el hecho que no hay que temer la caída de ningún servidor, aunque llegase a existir problemas de colisiones entre aplicaciones y reinicios inesperados, el sistema como un todo seguiría en marcha.

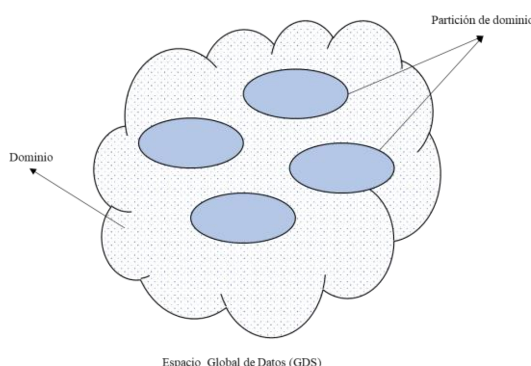

Figura 5. Representación del Espacio Global de Datos (GDS).

El alcance de la información es controlado por dos agentes esenciales, estos son los dominios y particiones:

- **Dominio**. Este representa una instancia del propio GDS, y todas las entidades existentes siempre pertenecerán a uno. De esta forma, el dominio establece una red virtual poniendo en contacto a las aplicaciones DDS que se han unido a ella. Esto garantiza el aislamiento entre grupo de aplicaciones y minimiza las posibles colisiones.
- **Partición**. Dentro de un dominio se puede hacer distinción entre diferentes grupos de Topics, estos grupos de Topics son las denominadas particiones y es proporcionado por DDS como otro mecanismo de control de alcance.

  Las particiones se encuentran descritas por nombres y han de ser llamadas explícitamente desde el código, con el fin de publicar o subscribirse a los Topics que estos contienen. El mecanismo que se usa para unirse a una partición es realmente flexible, tanto publicador como subscriptor pueden unirse únicamente proporcionando el nombre del participante, o incluso estos pueden unirse a varias particiones si existe presente una cierta expresión regular común a todos ellos.

Como se muestra en la *Figura* 6, puede verse que el flujo de información va desde publicadores a subscriptores, siempre bajo un mismo objeto “Topic”. Para realizar dicho intercambio de información es necesario que tanto el Topic que se envía como el esperado sean iguales (mismo nombre, mismo tipo y misma QoS). La QoS en cada bloque puede ser configurada de forma independiente, pero ha de ser compatible. Si se realiza una configuración de QoS en una entidad que englobe a otra, esta QoS funcionará igual para los bloques que se encuentren contenidos en ella por defecto.

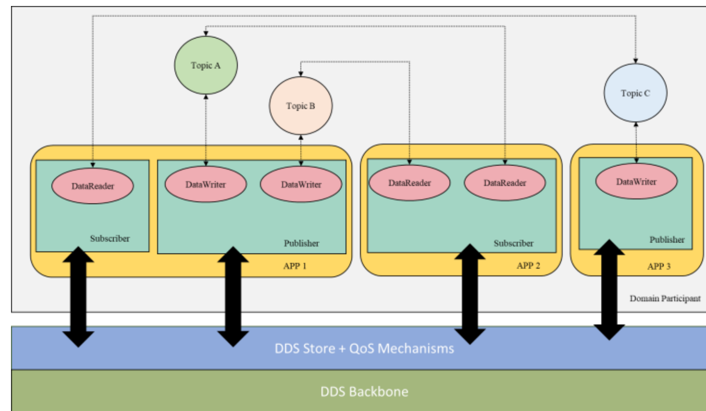

Figura 6. Entidades y flujo de la información en DDS.

2.1.2 Publisher (Publicador)

El **Publisher** es el objeto responsable del envío de datos. Su objetivo no es otro que publicar los datos de los Topics anteriormente definidos. El objeto Publisher necesita de otro objeto llamado **DataWriter**, que es el encargado de hacer de puerta de entrada en la publicación. Cada aplicación ha de tener un DataWriter, de forma que cuando una comunicación vaya a ser realizada, dicho DataWriter deberá de advertir de su existencia y hacer entrega del valor de los objetos de datos recibidos al Publisher. Una vez que los datos han sido proporcionados al Publisher desde el DataWriter concreto, es el publisher quien tiene la responsabilidad de realizar su correcta distribución a través del dominio preseleccionado. Esta distribución será acorde con la QoS que incorpore dicho publisher, que deberán de estar en sintonía con las del data-writer.

En conclusion, la aplicación hace uso de datos descritos en el data-writer y los publica en el contexto proporcionado por el publisher.

2.1.3 Subscriber (Subscriptor)

El **Subscriber** es el objeto responsable de la recepción de los datos publicados. Su objetivo es poner a disposición de una aplicación receptora dichos datos, de acuerdo con una QoS. Para la recepción de diferentes tipos de datos es necesario contar con otro objeto **DataReader**, creando así la asociaciación característica en una subscripción. De forma similar al caso anterior descrito para un publisher, es en este caso el data-reader el que describe los datos a los que hay que subscribirse en el contexto proporcionado por el subscriber.

Si una aplicación precisa de publicar datos de un tipo determinado, esta deberá de crear un **Publisher** o reutilizar uno anteriormente creado en conjunción con un **DataWriter** con las características deseadas de la publicación. De la misma manera, si una aplicación require de la recepción de datos, se debe de crear un objeto **Subscriber** (o reutilizar uno creado anteriormente) y un **DataReader** para establecer la subscripción.

Los subscriber disponen de objetos **Listener** que proporcionan a DDS la capacidad de notificar a la aplicación la existencia de eventos asíncronos. También existen otros objetos **StatusCondition**, **ReadCondition**, y **WaitSet** que proporcionan apoyo en comunicaciones basadas en tiempo de espera.

Cabe recordar que la comunicación entre publishers y subscribers únicamente puede hacerse si estos se encuentran conectados bajo un mismo dominio.

2.1.4 Topic

Un **Topic** representa la unidad minima de información que puede ser producida o consumida dentro del GDS. Un Topic se define por tres cosas básicas: un **tipo de dato**, un **nombre único** y un **conjunto de calidades de servicio** (QoS). Si la QoS no es ofrecida explicitamente, la aplicación DDS tomará algunos valores predeterminados prescritos por la norma.

Conceptualmente los Topics son objetos que deben de ser reconocidos a ambos lados de la comunicación, por lo que tienen que estar perfectamente predefinidos para que no se produzcan ambigüedadades en el proceso referencia hacia ellos.

El manejo de los Topics es relativamente simple, ya que tienen asociados un nombre único en el dominio, por lo que se hace fácilmente distinguible por las aplicaciones que lo usan. La QoS proporcionada por el Topic en adición a las presentadas por el data-witer y el publisher asociado a su vez, constituyen el conjunto de comportamientos desempeñados por parte de la publicación; mientras que, la QoS del Topic, data-reader y subscriber controlan el comportamiento de la subscripción.

El tipo de Topic es definido por un IDL, donde número de campos es arbitrario y los tipos de datos admitidos pueden ser de tipo primitivo, tipo template, o tipo compuesto. Algunos de los elementos más representativos compatibles con el lenguaje de progración **Modern C++** se muestran en la *Tabla* I.

Tabla I. Algunos tipos de datos admitidos para IDL en Modern C++.

Tipos de datos reconocidos por el IDL	
char	long long or int64
wchar	unsigned long long or uint64
octet	float
int8	double
uint8	boolean
long double	enum
constant	pointer
short ot int16	struct
unsigned short or uint16	union
long or uint32	typedef
unsigned long or uint32	bounded string

Un Topic queda definido por un tipo de dato, pero pueden existir varios Topics que se encuentren referidos a un mismo tipo de dato. Este caso puede crear algo de controversia por lo que se tiene que tener especial cuidado en hacer distinguibles las diferentes instancias. Para este tipo de casos se suele tomar una **key** (clave) que debe de ser proporcionada a DDS normalmente desde el propio IDL. De esta forma se consigue que *diferentes muestras de datos con el mismo valor de clave representen 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 DDS no realizará ninguna instanciación "automática". Esto tampoco supone un problema ya que existen otras formas de lograr el mismo resultado sin tener que introducer ninguna key en el IDL, como recurrir al uso de filtros.

2.1.5 Mecanismos de alcance

Tanto los dominios como las particiones actúan de barrera para el alcance de la información, sin embargo, estos dos objetos trabajan de forma estructural y para necesidades concretas como el filtrado de datos dentro de un Topic no pueden ser realizados. DDS posee mecanismos en forma de objetos como el **ContentFilteredTopic** que permiten crear instancias que tomen solo ciertos valores determinados tal y como se establezca en el filtro. En este caso la expression del filtro tiene la capacidad de operar sobre el contenido de los Topics al completo, en vez de analizar ciertos campos de los mensajes como ocurre en otras tecnologías basadas en pub/sub. La expresion del filtro se corresponde con una expresión estructuralmente similar a las cláusulas WRERE de SQL. Estas cláusulas se muestran en la *Tabla II*.

Tabla II. Operadores para Filtros DDS.

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

Los Topics Content-Filtered son útiles como se adelantaba anteriormente para hacer distinción entre diferentes instancias, pero también ayudan a limitar la cantidad de memoria usada por el middleware para estas. El uso del filtro simplifica el tamaño de la aplicación, dejando a DDS la lógica pertinente.

DDS posee además ortros mecanismos de alcance como son las Query Conditions (condiciones de consulta). Estas, al contrario que el Topic Content-Filtered, no filtra nada, sino que realiza una consulta a los datos anteriormente recibidos y disponibles en una caché lectora, dejando bajo control del usuario las acciones a tomar.

3 DESCRIPCIÓN

De acuerdo con los objetivos propuestos al comienzo de este documento, se ha realizado un desarrollo en Linux, recreando el sistema operativo del sistema embebido que se utilizaría en la aplicación real. El sistema real sería un sistema basado en microprocesador, programado a través del lenguaje de programación Modern C++ 11.

Para el desarrollo de las aplicaciones, se define el entorno de trabajo del proyecto. Este se ha desarrollado a través de la herramienta Visual Code Studio.

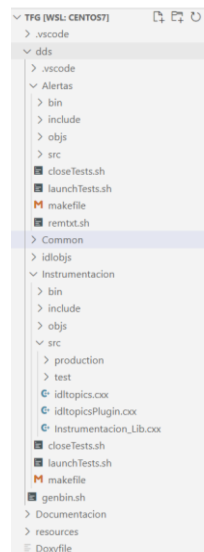

Figura 7. Directorios de trabajo del proyecto.

El proyecto se distribuye de igual forma que la mostrada en la *Figura 7*. Dentro de la carpeta “dds” se encuentra todo el desarrollo del proyecto. Una vez dentro, se puede destacar la existencia dos dominios (“Alertas” e “Instrumentación”), y se encuentran acompañados de otra carpeta “Common”, donde se ubican las reglas y patrones comunes a ambos dominios, como es el caso de la QoS. Cada dominio se encuentra distribuido de forma idéntica al resto, reservando espacio para los archivos binarios, para los “includes”, para el código fuente y para los archivos temporales. Junto a estos, se encuentran por separado el “Makefile” para poder realizar la compilación del proyecto, y algunos archivos para facilitar la ejecución de los subscriptores y publicadores, así como limpiar los registros que se hubieran guardado en formato de texto.

Una vez introducida la organización del proyecto se hace necesario hablar de los tipos de datos que utilizarán los Topics para establecer la comunicación, esto vendrá dado por un IDL. Dicho IDL ha sido un diseño propio, ideado para utilizar dos tipos de datos: datos tipo ***dashboard*** y datos tipo ***serviceAlert***, siendo el primero creado para datos instantáneos, compatibles con medidas de diferentes magnitudes físicas (**temperatura del motor**, **velocidad instantánea**, **consumo instantáneo**, **par motor** y **nivel de combustible**), queriendo llevar a cabo una colecta de los datos que llegan al cuadro de indicadores del vehículo. En el segundo caso, se ha querido llevar a cabo un diseño que propicie la implementación de alertas de necesidades de servicio, para llevar a cabo un registro por eventos de cambio. Estas alertas necesitarán de la lectura del número de kilómetros recorridos y desarrollarán su pequeña prognosis para publicar alertas de necesidad de mantenimiento. Los mantenimientos llevados a cabo serán el **cambio de aceite**, **reemplazo de las pastillas de freno**, **filtro de combustible** y **revisión general**.

El IDL utilizado se muestra a continuación:

```
enum maintenanceState{
    UNNECESSARY,
    CLOSE,
    REACHED
};

struct identifier{

    long resourceID;
    long sourceID;
};

struct itemReplacement{

    long long distanceAvailable;
    maintenanceState state;
};

struct serviceAlert{

    identifier ID;
    long long timeData;
    itemReplacement itemAlert;
};

struct dashboard {

    identifier ID;
    long long timeData;
    float variable;
};
```

Nótese que se incluye un campo “identifier” cuyo objetivo es reconocer los diferentes dispositivos conectados a la red y así también, poder filtrar alguno de los datos por su resourceID y sourceID.

Las alertas de servicio contarán con el campo “mantenanceState”, que avisará del estado en el que se encuentra el mantenimiento de los diferentes elementos de desgaste, y se diferenciarán tres posibles estados de mantenimiento: **innecesario**, **cerca** y **alcanzado**. Además, cada elemento de alertas de servicio llevará la cuenta de los kilómetros que quedan hasta su mantenimiento.

Para el caso de los elementos del cuadro de instrumentos es más sencillo, ya que únicamente poseen los identificadores anteriormente mencionados, el valor de la variable y un marcado temporal. Este marcado temporal o timestamp también está presente en las alertas de servicio y servirá de verificación para ver que los datos llegan según el tiempo esperado y saber cúando una alerta ha dado lugar.

Llegado hasta aquí, hay que decir que recordar que cuando llegue el caso de la implementación, dicho IDL deberá de convertirse a lenguaje C++ bajo las reglas marcadas por RTI DDS. Para esto es necesario recurrir a la herramienta “Code Generator” provista en RTI Connext Launcher, la cual generará los archivos PSM necesarios para su compilación.

Una trama DDS para un dato de tipo ***dashboard*** se muestra a continuación:

```
rtiddsspy.bat - CALL "C:\Program Files\rti_connext_dds-6.0.1\bin\rtiddsspy.bat" -domainId 1 -printS...
RTI Connext DDS Spy built with DDS version: 6.0.1 (Core: 1.9a.00, C: 1.9a.00, C++: 1.9a.00)
Copyright 2012 Real-Time Innovations, Inc.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
rtiddsspy is listening for data, press CTRL+C to stop it.

source_timestamp   Info  Src HostId  topic              type
----------------   ----  ----------  -----------------  -----------------
1631407764.295265  W +N  AC1C5FA0    Instrumentacion    dashboard
1631407765.295784  d +N  AC1C5FA0    Instrumentacion    dashboard
ID:
   resourceID: 0
   sourceID: 3
timeData: 1631407765295
variable: 1

1631407766.296269  d +M  AC1C5FA0    Instrumentacion    dashboard
ID:
   resourceID: 0
   sourceID: 3
timeData: 1631407766296
variable: 2

1631407767.296737  d +M  AC1C5FA0    Instrumentacion    dashboard
ID:
   resourceID: 0
   sourceID: 3
timeData: 1631407767296
variable: 3
```

Figura 8. Trama DDS para datos de tipo Dashboaard.

Por consiguiente, se muestra otra trama DDS para datos de tipo ***serviceAlert***:

```
rtiddsspy.bat - CALL "C:\Program Files\rti_connext_dds-6.0.1\bin\rtiddsspy.bat" -domainId 2 -printSample -qosFile C:\Use
RTI Connext DDS Spy built with DDS version: 6.0.1 (Core: 1.9a.00, C: 1.9a.00, C++: 1.9a.00)
Copyright 2012 Real-Time Innovations, Inc.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
rtiddsspy is listening for data, press CTRL+C to stop it.

source_timestamp   Info  Src HostId  topic              type
----------------   ----  ----------  -----------------  -----------------
1631407548.881074  R +N  AC1C5FA0    Instrumentacion    dashboard
1631407548.881473  W +N  AC1C5FA0    Cambio_aceite      serviceAlert
1631407564.886807  d +N  AC1C5FA0    Cambio_aceite      serviceAlert
ID:
   resourceID: 0
   sourceID: 1
timeData: 1631407564886
itemAlert:
   distanceAvailable: -600
   state: REACHED

1631407565.887252  d +M  AC1C5FA0    Cambio_aceite      serviceAlert
ID:
   resourceID: 0
   sourceID: 1
timeData: 1631407565887
itemAlert:
   distanceAvailable: -600
   state: REACHED

1631407566.887663  d +M  AC1C5FA0    Cambio_aceite      serviceAlert
ID:
```

Figura 9.Trama DDS para datos de tipo serviceAlert.

Ambas tramas han sido captadas a través de la herramienta *RTI DDS Spy*, bajo el entorno de pruebas cerrado del proyecto. La primera muestra se encontraba en el dominio 1, mientras que la segunda estaba en el dominio 2. Al comienzo de la comunicación, en ambas figuras puede apreciarse el nombre y tipo de los Topics que se encuentran activos en cada dominio, junto a su marca temporal de su descubrimiento. Como se comentará en capítulos sucesivos a este, el Topic "Instrumentacion" aparece duplicado en ambos dominios, esto no significa que sean el mismo, sino que existen dos nombres iguales para un mismo Topic. Esto se hace para comprobar que los dominios crean aislamiento y evitan que existan colisiones entre datos del mismo nombre circulando por la red virtual.

Dado que la base del desarrollo del proyecto se encuentra fundamentada en clases patrones ubicadas dentro del directorio "Common", se procede a su descripción en el siguiente apartado.

3.1. Common

Para llegar a comprender el completo funcionamiento de las clases dentro del proyecto es necesario analizar los ficheros “DDSPaterns.hpp” y “DDSPatterns.cxx” dentro de la carpeta “Common”. En dichos ficheros se encuentran las declaraciones y desarrollos de las clases patrón que servirán de guía para la elaboración de los diferentes publicadores y subscriptores. Sin estas clases patrón la declaración de objetos sería menos eficiente y gracias a ellas, el desarrollo de la estructura publicador/subscriptor se vuelve mucho más mecánica.

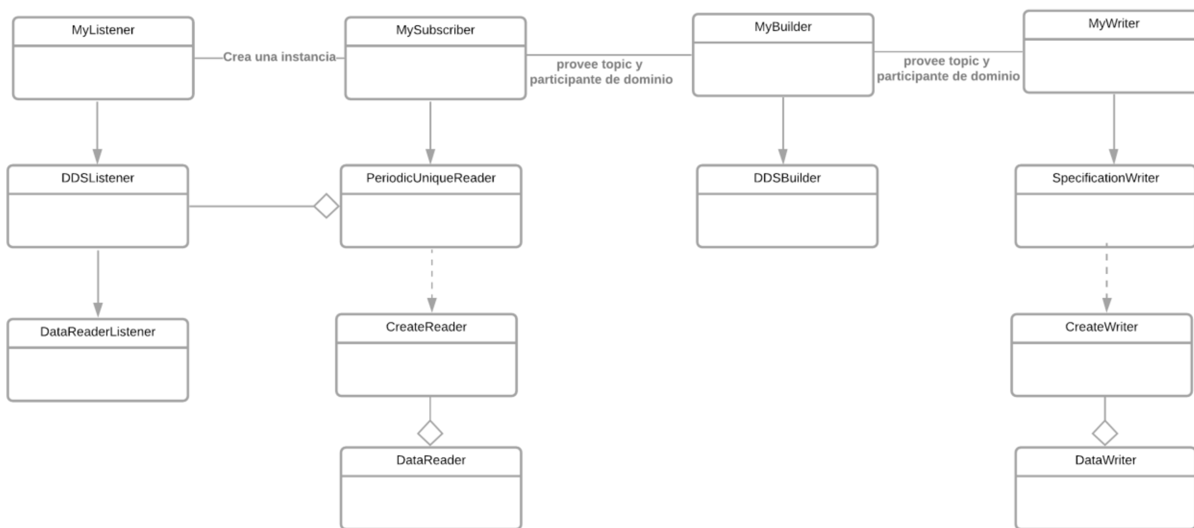

Figura 10. Diagrama de clases genérico del proyecto.

El flujo seguido se muestra en la *Figura* 10. En ella se pueden apreciar la existencia de clases RTI DDS como DataReaderListener, DataReader, CreateReader, DataWriter y CreateWriter. También se podría añadir las clases Topic y DomainParticipant que entrarían en juego en las clases DDSBuilder y MyBuilder. Luego se puede encontrar las clases DDSListener, PeriodicUniqueReader, DDSBuilder y SpecificationWriter. Estas clases son las que incorpora el patrón y a la hora del desarrollo a un mayor nivel, sólo se tendrían que crear las clases MyListener, MySubscriber, MyBuilder y MyWriter por herencia de las clases patrón.

Para ampliar más la información de las clases patrón con sus atributos y métodos, se adjunta la *Figura 11*.

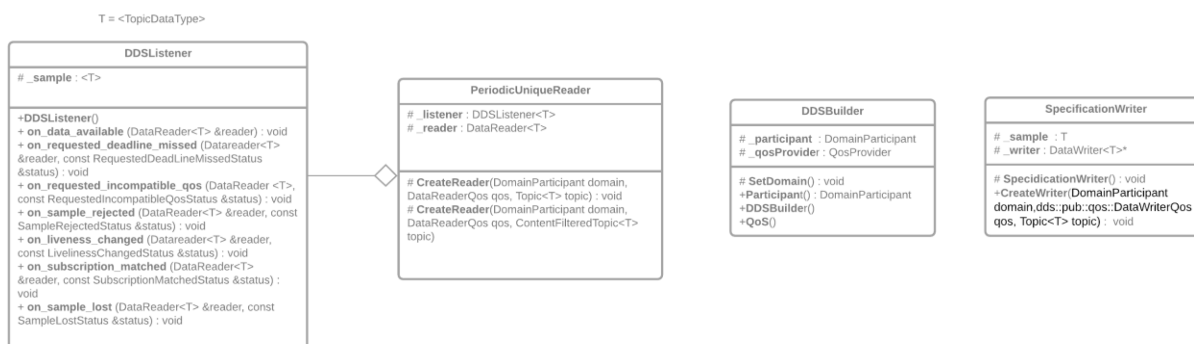

Figura 11. Diagrama de clases extendido de las clases patrón.

Dentro de "Common" también se tienen los ficheros denominados "DDSXml.hpp" y "DDSXml.cxx". Estos tienen el objetivo de habilitar la posibilidad de leer información de ficheros XML desde cualquier entorno del proyecto. El principal uso que se le ha dado ha sido el de la lectura de la configuración de QoS para las diferentes entidades hayan requerido de su uso.

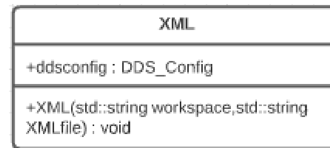

Figura 12. Diagrama de la clase "XML".

3.2. Dominio Instrumentación

En este dominio se trabaja, como su propio nombre indica, con las señales provenientes de la instrumentación. Para ello se ha creado un único Topic del mismo nombre, ya que, aunque las señales a tratar tengan diferentes magnitudes físicas, estan pueden representarse por un mismo tipo de dato. Es de este modo por lo que se recurre a la instanciación de dicho Topic y se le da uso al filtro de DDS.

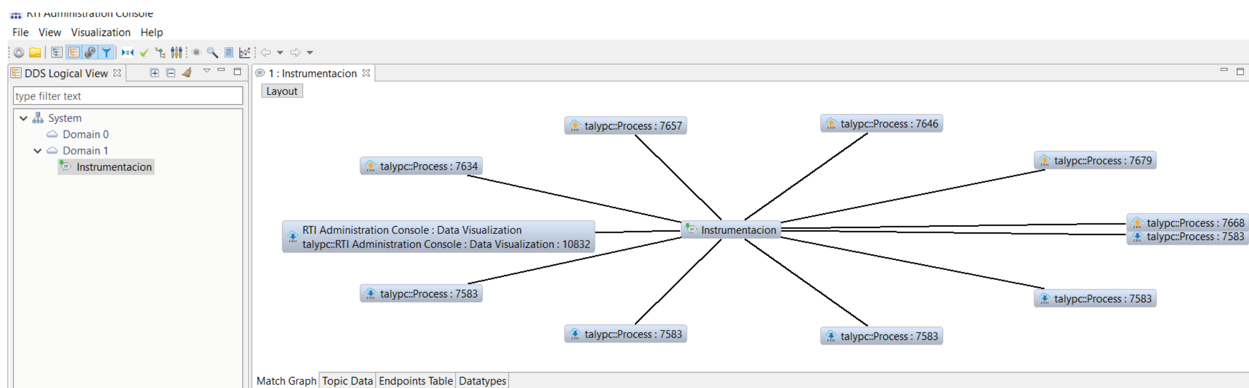

Figura 13. Dominio Instrumentación visto a través del RTI Administration Console.

Otra herramienta útil que ofrece RTI es el Administration Console. Con ella se puede ver de forma gráfica las entidades que componen el dominio. En la *Figura 13* puede verse cómo existe un único Topic ("Instrumentacion") con 11 procesos conectados a él. Estos procesos son los 5 subscriptores y sus publicadores asociados, además de la propia subscripción que hace la herramienta para poder ver dicho contenido.

Al tratarse de instancias, los subscriptores y publicadores siguen todos un mismo diseño. Para diferenciar a dichas instancias se toma como *key* el identificador único del que se hablaba cuando se introdujo el funcionamiento del IDL.

Los publicadores han sido creados como tests que verifiquen la correcta implementación de los subscriptores, por lo que han sido configurados con valores aleatorios, simplemente para verificar que el envío y recibo de datos se hace de la forma adecuada. Dichos publicadores toman el tipo de dato con el ID asociado a la magnitud a medir, la hora de publicación (en milisegundos y en formato *Epoch)*, y un valor incremental que hará del valor de la variable. Todo esto será publicado al GDS cada segundo y desde el otro extremo los subscriptores, a la espera con el método *on_data_available()* de sus Listeners asoaciados , recibirán datos cada vez que se detecte de su existencia en el GDS, almacenando los valores obtenidos en su correspondiente fichero de texto. Este flujo queda patente en la *Figura 14*.

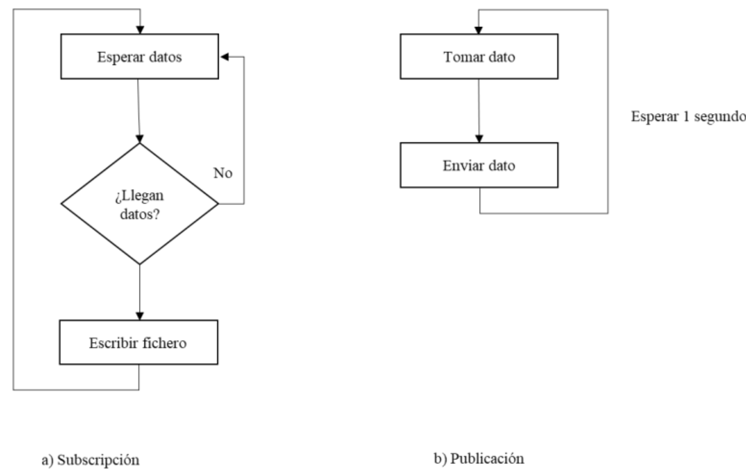

Figura 14. Diagrama de flujo de los subscriptores y publicadores en el dominio Instrumentación.

3.3. Dominio Alertas

Este dominio trabaja de forma diferente al anterior. En el dominio Alertas se puede encontrar que no existen ninguna estancia de ningún Topic, sino que existen 4 Topics de tipo *serviceAlert* y 1 Topic de de tipo *dashboard*. Cada alerta será totalmente independiente a la otra, únicamente que para la evaluación de estas será necesario tener un Topic que lleve asociado el kilometraje del vehículo y sea publicado. Todas las alertas contarán con un subscriptor a este kilometraje. Este Topic kilometraje llevará el nombre de *Instrumentacion*.

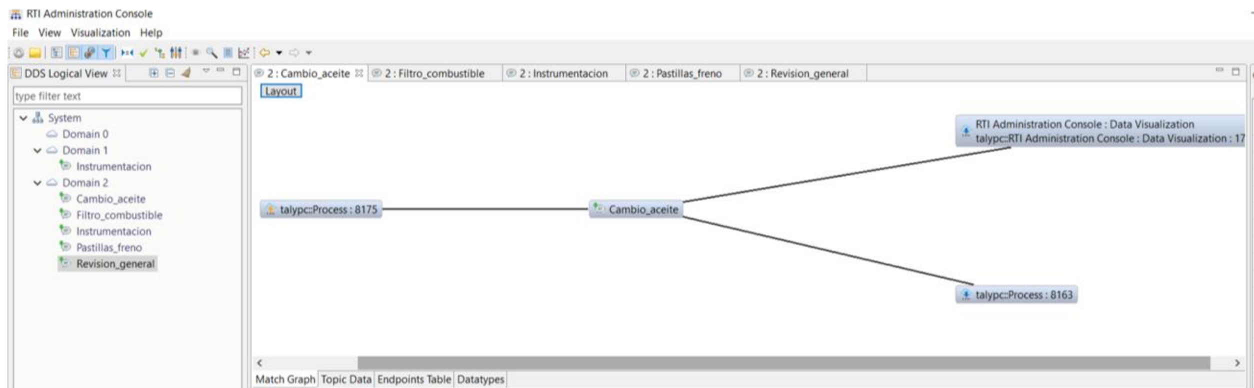

Figura 15. Topic "Cambio_aceite" en el dominio Alertas.

En la *Figura 15* puede aprecirarse que en torno al Topic "Cambio_aceite" existen un subscriptor con su publicador asociado y uno de más que pertenece al usado por la herramienta para su descubrimiento. Al igual que ocurre con este Topic ocurrire con los demás, excepto el Topic Instrumentación, cuya estructura puede verse en la *Figura 16*. En dicha figura, pueden verse representados el publicador único y sus 6 subscriptores. Esto es porque 4 son los referentes a las alertas de mantenimiento, mientras que otro queda reservado para la propia herramienta, y el último es un subscriptor independiente para realizar un registro por separado del kilometraje. Esto ocurre para evitar problemas de memoria a la hora de acceder a un fichero desde varias aplicaciones a la vez o de intentar la escritura al mismo tiempo que se requiere leer de este.

Hay que prestar atención a una apreciación. En el panel de navegación de la herramienta puede verse que la representación de los dominios se hace de forma separada, lo que ya sirve de antesala para entender que los dominios actúan como entes totalmente aislados los unos de los otros. En el caso de la *Figura 16*, puede verse que aparece un icono de advertencia en el dominio 1. Esto no es preocupante puesto que la imagen fue tomada posteriormente cuando ya se encontraban desconectados los publicadores de dicho dominio y la

herramienta advertía de sus ausencias.

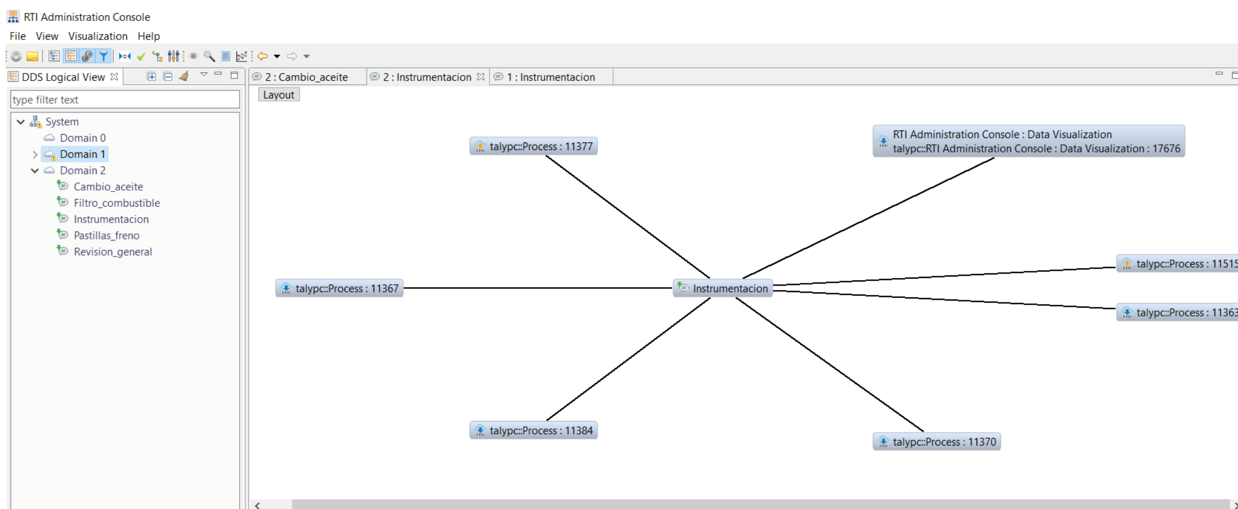

Figura 16. Topic "Instrumentación" dentro del dominio Alertas.

Dado que el dominio Alertas se encuentra diseñado de una forma un poco diferente al dominio Instrumentacion, su puesta en marcha también lo será. El dominio anterior se encontraba principalmente fundamentado en subscriptores, mientras este lo hace principalmente en publicadores. Aquí el papel de tester lo hace el kilometraje. A efectos prácticos este se lleva a cabo de la misma forma que los elaborados en el dominio anterior, pero sin la obligatoria inclusión de un ID para facilitar el filtrado, dado que aquí no lo hay. La subscripción siempre es igual, existen 5 subscriptores y cada uno de ellos subscribe a una alerta excepto el correspondiente al kilometraje. La parte más elaborada es la que confiere a la creación de dichas alertas, que como se comentaba anteriormente, requiere de una subscripción previa. El publicador, una vez recibido los datos de kilometraje calculará la necesidad de servicio (en el propio publicador se inicializa el número de kilómetros a los que se debe de realizar el servicio). Si la distancia disponible antes del servicio es inferior al 5% de la distancia total prevista por este se establecerá una alarma con un estado "CLOSE", si esta distancia sobrepasa el límite indicará una alerta "REACHED", y si por el caso contrario no ocurre nada de lo anterior se marcará como "UNNECESARY".

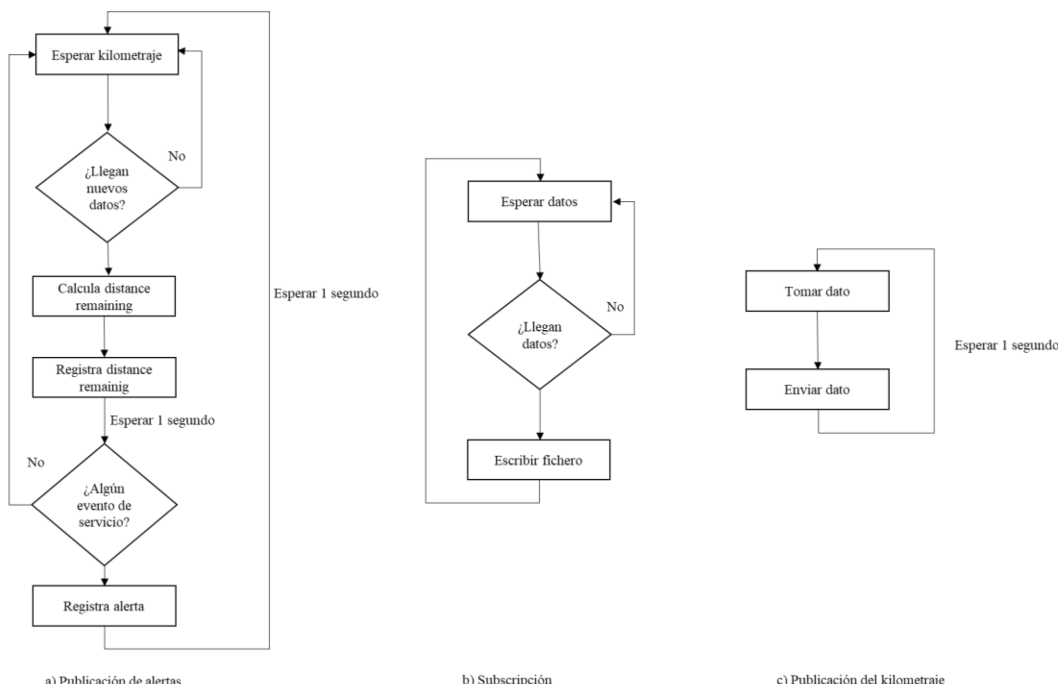

Figura 17. Diagrama de flujo de los publicadores y subscriptores en el dominio Alertas.

Es necesario comprender que el registro de alertas sólo se produce por cambios de evento, por lo que se grabará únicamente el primer evento de cambio que se produzca. Antes de dicha comprobación se tiene que almacenar el kilometraje y esperar 1 segundo para tener seguridad que los procesos de cálculos y registros ya han tenido lugar. El flujo que siguen los publicadores y subscriptores en este dominio queda reflejado en la *Figura 17*.

Se tiene que hacer también una apreciación referente al restablecimiento de alertas. Una vez estas quedan disparadas han de ser resueltas a mano. La mejor forma de hacerlo es importando el número de kilómetros hasta nuevo servicio desde un XML externo. En el proyecto esto ha quedado hardcodeado para asegurar un buen funcionamiento en su demostración académica.

4 IMPLEMENTACIÓN

Para la implementación lo primero será registrarse en RTI para que se faciliten las librerías y licencia de prueba. Esto se consigue a través de la siguiente dirección: https://www.rti.com/free-trial. Una vez descargado, se ha de instalar la aplicación RTI Launcher e indicar desde el apartado "Configuration" de la herramienta, la ruta en la que se encuentra la licencia. Después de ello, habría que hacer lo mismo con el directorio de licencia de la propia librería. Más tarde, desde la ruta en la que se vaya a desarrollar el proyecto, será necesaria la apertura de un terminal y configurar las variables de entorno: RTI_LICENSE_FILE, NDDSHOME y PATH. La ruta de la primera variable será la ruta en la que se encuentre la licencia dentro de la librería proporcionada. La ruta de NDDSHOME irá donde se establezca el rti_connext_dds-X-X-X, dentro de la carpeta de la librería y para el PATH se anexará el compilador usado para la versión de sistema operativo.

4.1. Publicación

Lo primero a tener en cuenta para llevar a cabo la publicación es la creación de un **DomainParticipant** (Dominio de Participación), el cual define en qué dominio va a trabajar la aplicación y se utilizará para crear el resto de las entidades necesarias como son los Publisher, Datawriter y Topic. La clase DomainParticipant tiene los argumentos especificados en la *Tabla III*.

Tabla III. Argumentos de la clase DomainParticipant.

domainID	Valor entero que sirve como identificador de dominio al que pertenece DomainParticipant.
QoS	Calidad de servicio que utilizará. Si el campo queda vacío se usarán los parámetros por defecto.
listener	Es opcional. Se añade si se precisa de algún tipo de información asíncrona. El listener responderá con Callbacks (rutinas de devolución de llamadas) a eventos específicos que se produzcan en el uso de DomainParticipant.
mask	Define los tipos de eventos que dan lugar a las rutinas de devolución de llamada del listener.

El código para su creación sería el siguiente:

```
_participant = new DomainParticipant(0,
                                     the_qos,
                                     NULL,
                                     dds::core::status::StatusMask::none()
);
```

En el ejemplo, el dominio usado es el número 0, se utiliza una QoS que debe de ser definida, no se utiliza ningún listener y se restringe todo tipo de evento.

La declaración de un participante de dominio es imperativa para la creación de un **Topic**, pero pueden existir construcciones ofrecidas por RTI DDS que no recojan al participante de dominio como argumento, dado

que existen varios métodos para inicializar el Topic en dicho participante. Al no existir una única construcción válida, para el proyecto se ha escogido la que utiliza al participante como argumento, pero no tiene entrada para el tipo de nombre, ya que este se ha de pasar en forma de *template*. Los argumentos para la creación de un topic se muestran en la *Tabla IV*.

Tabla IV.Argumentos de la clase Topic.

participant	Participante de dominio donde va a trabajar el Topic.
topic_name	Nombre del Topic, este debe ser único para todo el dominio.
QoS	Calidad de servicio que utilizará. Si el campo queda vacío se usarán los parámetros por defecto.
listener	Es opcional. Se añade si se precisa de algún tipo de información asíncrona. El listener responderá con Callbacks (rutinas de devolución de llamadas) a eventos específicos que se produzcan en el uso del topic.
mask	Define los tipos de eventos que dan lugar a las rutinas de devolución de llamada del listener.

Un ejemplo de la implementación en código de un Topic se muestra a continuación:

```
_InstrumentationTopic = new Topic<dashboard> (
              *_participant,
              "Instrumentacion",
              _qosProvider->topic_qos("DDS_QOS::DDSPeriodicPatternQosProfile"),
              NULL,
              dds::core::status::StatusMask::none()
        );
```

Nótese que tipo de dato del Topic ha sido pasado mediante un *template*, por lo que el topic en este caso es de tipo "dashboard".

Para cerrar el apartado de la publicación sólo queda por definir el **DataWrite**r. Se ha de remarcar que la creación de un **Publishe**r realmente es **opcional** y si este no se crea de forma explícita, se utiliza un Publisher por defecto. Esto se da únicamente en RTI Connext DDS y no es parte del estándar DDS, de lo contrario Publisher podría utilizarse para la creación de DataWriter.

Para la creación del data-writer es neceserario la inclusión de cinco argumentos. Estos se muestran en la *Tabla V*.

Tabla V. Argumentos de la clase DataWriter.

publisher	El publicador al que pertenece el DataWriter.
topic	Topic asociado con el DataWriter.
QoS	Calidad de servicio que utilizará. Si el campo queda vacío se usarán los parámetros por defecto.
listener	Es opcional. Se añade si se precisa de algún tipo de información asíncrona. El listener responderá con Callbacks (rutinas de devolución de llamadas) a eventos específicos que se produzcan en el uso del DataWriter.
mask	Define los tipos de eventos que dan lugar a las rutinas de devolución de llamada del listener.

Un ejemplo de la implementación en código de un DataWriter se muestra a continuación:

```
DataWriter<serviceAlert> mywriter(
    Publisher(participant),
    mytopic,
    qosProvider.datawriter_qos("DDS_QOS::DDSPeriodic1SecondPatternQosProfile"),
    NULL,
    dds::core::status::StatusMask::none()
);
serviceAlert dato;
mywriter.write(dato);
```

Al igual que ocurría en el caso de Topic, DataWriter también reconoce el tipo de dato a través de un *template*. Publisher actúa de forma que pone en contexto al DataWriter, ayudándole a saber en qué dominio se encuentra. La publicación tiene lugar cuando se utiliza el método *write()*.

4.2. Subscripción

El proceso de subscripción es similar al de publicación en sus inicios, puesto que se hace necesaria la definición de un DomainParticipant y de un Topic. Estos no tienen por qué volver a definirse en el código, puesto que los anteriormente definidos pueden y deben de ser reutilizados para asegurar el buen funcionamiento del servicio DDS, haciendo compatibles ambos extremos de la comunicación.

En este caso, el objeto fundamental en la comunicación por parte de la subscripción será el **DataReader**. El objeto **Subscriber** se utilizará, una vez más, para poner en contexto al DataReader en qué dominio participante se encuentra. Los argumentos de entrada para la creación de un DataReader se muestran en la *Tabla VI*.

Tabla VI. Argumentos de la clase DataReader.

subscriber	Subscriptor al que obedece el DataReader.
topic	Topic asociado al DataReader.
QoS	Calidad de servicio que utilizará. Si el campo queda vacío se usarán los parámetros por defecto.
listener	Es opcional. Se añade si se precisa de algún tipo de información asíncrona. El listener responderá con Callbacks (rutinas de devolución de llamadas) a eventos específicos que se produzcan en el uso del DataReader.
mask	Define los tipos de eventos que dan lugar a las rutinas de devolución de llamada del listener.

Un ejemplo de su implementación en código se muestra a continuación:

```
void CreateReader(DomainParticipant domain,dds::sub::qos::DataReaderQos qos,
                  Topic<T> topic){

                _reader = new DataReader<T>(
                Subscriber(domain),
                topic,
                qos,
                _listener,
                dds::core::status::StatusMask::all()
            );
        }
```

El código mostrado representa la creación de un DataReader genérico, en el que los argumentos deberán de ser inicializados en el desarrollo. Una vez más el tipo de dato deberá de ser pasado en forma de *template* y ahora sí, tomará mayor importancia el uso de los listeners, en concreto el uso de **DataReaderListene**r. Por defecto se aprueba el uso de todo tipo de eventos en la máscara.

Si se desea hacer uso del filtro DDS, será necesario cambiar el formato de la implementación de forma que el objeto ContentFilteredTopic reemplace al objeto Topic.

Volviendo de nuevo al **DataReaderListener**, este posee siete posibles callbacks:

- **On_requested_deadline_missed**. Produce una respuesta cuando se sobrepasa un timeout.
- **On_requested_incompatible_qos**. Produce una respuesta cuando se detecta una QoS incompatible.
- **On_sample_rejected**. Produce una respuesta cuando el sample ha sido rechazado.
- **On liveness_changed**. Produce una respuesta cuando se ha cambiado el estado de la muestra.
- **On_data_available**. Produce una respuesta cuando llega una nueva muestra.
- **On_subscription_matched**. Produce una respuesta cuando ha encontrado un Publisher compatible.
- **On_sample_lost**. Produce una respuesta cuando se detecta que una muestra se ha perdido.

Lo ideal en la utilización de estos callbacks es la creación de una clase “DDSListener” que los englobe a todos y así disponer todo y cada uno de ellos según se requiera. Esto se ha llevado a cabo en la elaboración del proyecto, pero a la hora de la implementación se mostrará un ejemplo del uso de *on_data_available()*, puesto que ha sido el método más usado y el resto de callbacks se utilizan de la misma forma.

```
void InstrumentationListener::on_data_available(DataReader<dashboard>&reader)
{
    _sample =  reader.take()[0];
}
```

En este ejemplo puede verse que se llama al método “on_data_available” a través de un objeto “InstrumentationListener”, este a su vez es una instancia de DDSListener anteriormente comentado. Lo importante en este ejemplo es conocer cómo se lleva a cabo la extracción de datos por parte de los subscriptores, y esto tiene lugar aquí, a través del método *take()* o *read()*. La diferencia entre ambos métodos reside en que *read()* proporciona una copia de datos a la aplicación y deja en cola los datos, mientras que *take()* elimina los datos de la cola antes de ser entregados a la aplicación. Se escoge el argumento “[0]” de la serialización porque es el que transporta la muestra buscada (Topic).

4.3. Clases Patrón

Como ya se comentó en el capítulo anterior, el directorio “Common” del proyecto está relacionado con las clases de uso general. Es por ello que se va a dar a conocer las declaraciones de las clases que aparecen en la *Figura 11*.

4.1.1 DDSListener

Esta clase es la fundamental para realizar las “escuchas” de los DataReader, constituyendo el primer pasi para la creación de un subscriptor. El Código de su declaración se muestra a continuación:

```
template<typename T>
class DDSListener : public DataReaderListener<T>{
    protected:
        T _sample;
    public:
        DDSListener(){};
        void on_data_available (DataReader<T> &reader){}
        void on_requested_deadline_missed(DataReader<T> &reader, const dds::core::status::RequestedDeadlineMissedStatus &status){}
        void on_requested_incompatible_qos (DataReader<T> &reader, const dds::core::status::RequestedIncompatibleQosStatus &status){}
        void on_sample_rejected (DataReader<T> &reader, const dds::core::status::SampleRejectedStatus &status){}
        void on_liveliness_changed (DataReader<T> &reader, const dds::core::status::LivelinessChangedStatus &status){}
        void on_subscription_matched (DataReader<T> &reader, const dds::core::status::SubscriptionMatchedStatus &status){}
        void on_sample_lost (DataReader<T> &reader, const dds::core::status::SampleLostStatus &status){}
};
```

4.1.2 PeriodicUniqueReader

Esta es la clase encargada de crear los DataReaders que se encargarán de recibir los datos con ayuda de los Listener y en el contexto proporcionado por el subscriptor. El Código de su declaración es el siguiente:

```
template<typename T>
class PeriodicUniqueReader{
    protected:
        DDSListener<T>* _listener;
        DataReader<T>* _reader;
        void CreateReader(DomainParticipant domain,dds::sub::qos::DataReaderQos qos, Topic<T> topic){
            _reader = new DataReader<T>(
                Subscriber(domain),
                topic,
                qos,
                _listener,
                dds::core::status::StatusMask::all()
            );
        }
        void CreateReader(DomainParticipant domain,dds::sub::qos::DataReaderQos qos, ContentFilteredT
opic<T> topic){
            _reader = new DataReader<T>(
                Subscriber(domain),
                topic,
                qos,
                _listener,
                dds::core::status::StatusMask::all()
            );
        }
    public:
        PeriodicUniqueReader(){};
};
```

Nótese que la clase PeriodicUniqueReader posee dos posibles métodos para la creación de los DataReaders. Estos métodos estás disponibles para hacer uso del Topic al completo o hacer uso de un Topic filtrado, a través de la clase *ContentFilteredTopic*.

4.1.3 DDSBuilder

Esta clase es esencial para la creación de los Topics y su inicialización en los dominios. También ofrece soporte para las calidades de servicio (QoS).

```
class DDSBuilder{
    protected:
        DomainParticipant* _participant;
        dds::core::QosProvider* _qosProvider;
        void SetDomain(int DomainID);
    public:
      DDSBuilder(){};
      DomainParticipant Participant(){return *_participant;}
         dds::core::QosProvider QoS(){return *_qosProvider;}
};
```

4.1.4 SpecificationWriter

Esta clase realmente no dista mucho de la creación de un simple CreateWriter, pero se hace fácilmente configurable a la hora de necesitar cualquier modificación en los posibles publicadores. El Código de su declaración se muestra a continuación:

```
template<typename T>
class SpecificationWriter{
    protected:
        T _sample;
        DataWriter<T>* _writer;
    public:
        SpecificationWriter(){};
        void CreateWriter(DomainParticipant domain,dds::pub::qos::DataWriterQos qos, Topic<T> topic){
            _writer = new DataWriter<T>{
                Publisher(domain),
                topic,
                qos,
                NULL,
                dds::core::status::StatusMask::none()
            };
        }

};
```

5 DEMOSTRACIÓN

Como se anticipaba al comienzo de este documento, se pretende demostrar que DDS es realmente implementable en soluciones de la vida real, y más concretamente en la industria. Para ello se escogió el sector automovilístico como motivación, en que un sistema operativo en tiempo real fuera el gobernante de las comunicaciones en un vehículo, interconectando las diferentes ECUs, monitorizando y registrado dicha actividad. Una vez desarrolladas las aplicaciones para tales propósitos se procede a su utilización.

Los ensayos que se van a llevar a cabo son simples. Primero se va a proceder a la ejecución una de las aplicaciones en el dominio Instrumentación con su correspondiente test. Se va a comprobar que los datos llegan en el tiempo marcado, así como sus valores predefinidos. Más tarde se procederá a ejecutar todos los tests a la vez para comprobar que los datos fluyen correctamente en agrupación, y como Instrumentación posee varias instancias del mismo Topic, se podrá comprobar el correcto funcionamiento del filtro de DDS.

Una vez terminen los ensayos sobre el primer dominio, se procederá a hacer lo mismo con el segundo, y cuando se compruebe que todos los datos discurran a la vez sin ningún tipo de problema se realizará la ejecución de todas las aplicaciones de todos los dominios a la vez. Esto último realmente no será necesario comprobarlo, ya que si se han cumplido los dos puntos anteriores este caso se cumplirá, ya que como se pudo comprobar por la *Figura 13*, los dominios permanecen efectivamente aislados.

5.1. Dominio Instrumentación

Para hacer una demostración básica del funcionamiento de este dominio se tomará la temperatura del motor como referente y se procederá a ejecutar el publicador frente al subscriptor como en la *Figura 18*.

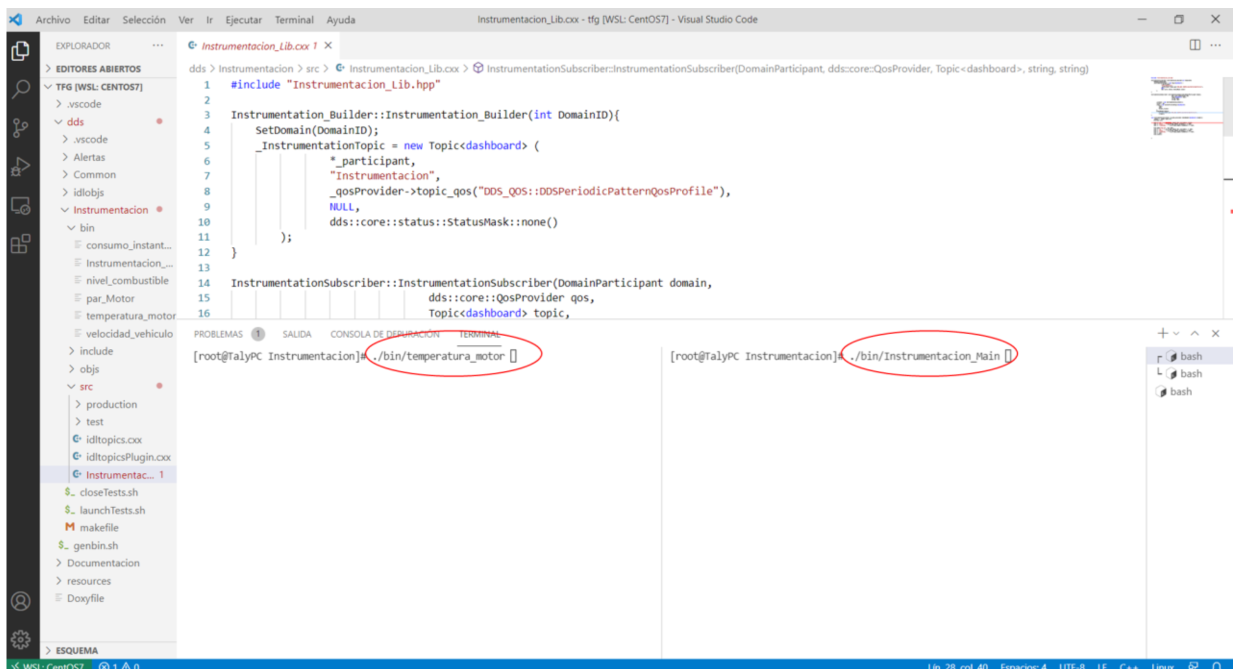

Figura 18. Ejecución de las aplicaciones "temperatura_motor" e "Instrumentacion_Main".

En la *Figura 19* puede verse el resultado de correr dichas aplicaciones. Los datos se registran en un fichero de texto llamado "log.txt" y una vez abierto puede comprobarse que las dos últimas muestras que se han recibido difieren exactamente en 1 segundo como se tenía previsto, además de comprobar que el filtro ha actuado bien, ya que el ID=3 es el configurado para la temperatura del motor.

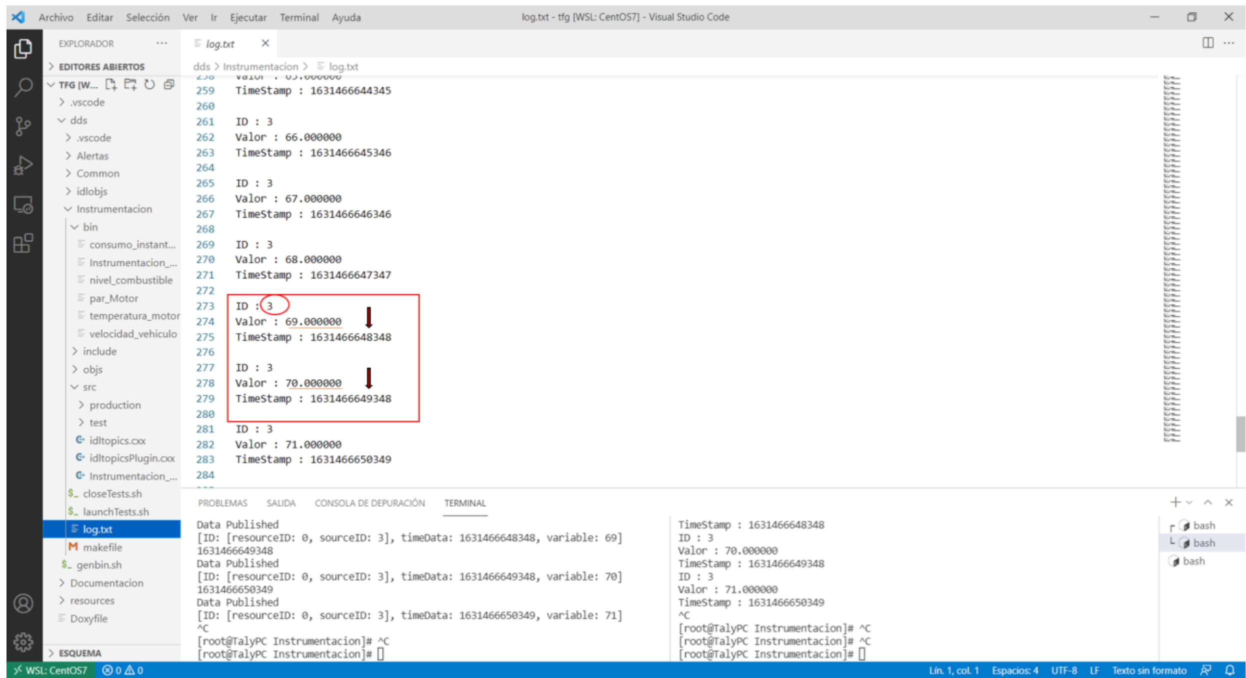

Figura 19. Resultados del envío de la temperatura del motor a través de DDS.

Una vez comprobado el buen funcionamiento del sistema para una señal, se procede a hacer lo mismo con todas las señales del dominio. El programa realizado en Linux denominado "launchTests" ayuda para tal propósito.

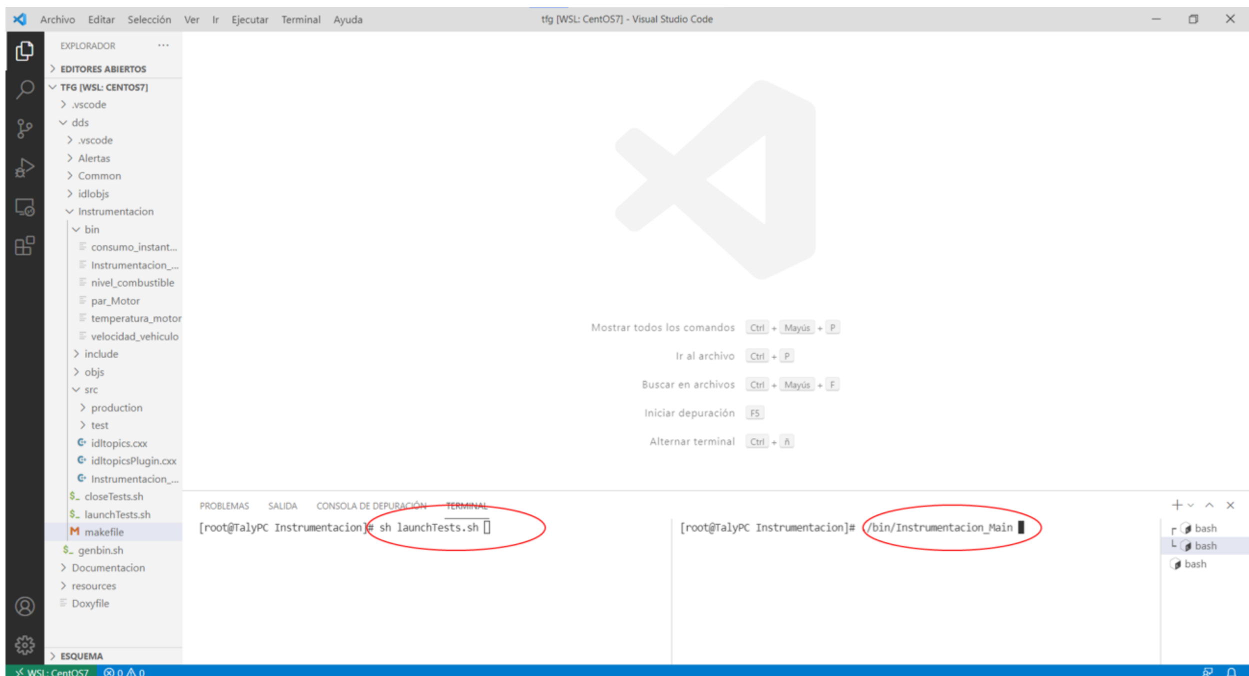

Figura 20. Instrucciones para ejecutar todas las aplicaciones del dominio Instrumentación.

Una vez terminada la ejecución de las aplicaciones, se procede a analizar el comportamiento de los datos registrados. Se remarca que, a pesar de no llegar a ejecutarse, por razones obvias, todos los programas al mismo tiempo; se mantiene el orden prestablecido, y las variaciones de tiempo entre muestra y muestra tienen un error inferior a la milésima de segundo.

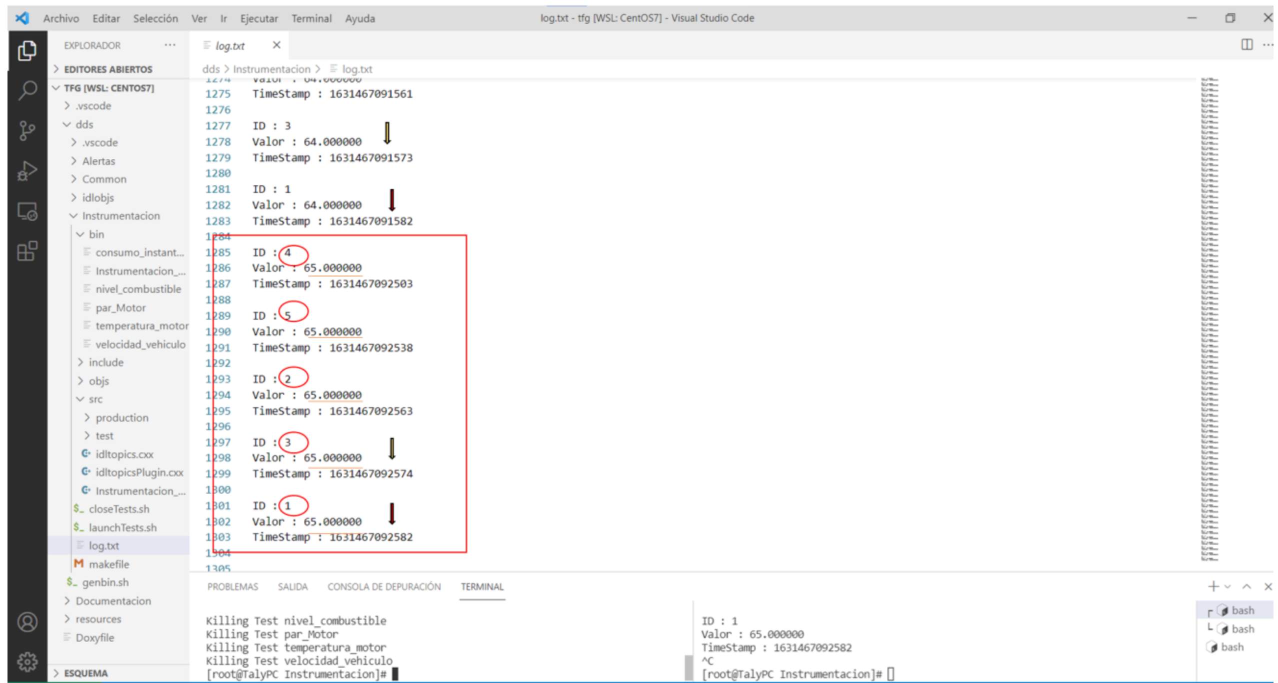

Figura 21. Resultado de la ejecución de todas las aplicaciones en el dominio Instrumentación.

5.2. Dominio Alertas

Al igual que con el anterior dominio, se pretende ejecutar primero las aplicaciones asociadas a una única señal y extraer las primeras conclusiones sobre esta. La señal escogida ha sido la del cambio de aceite.

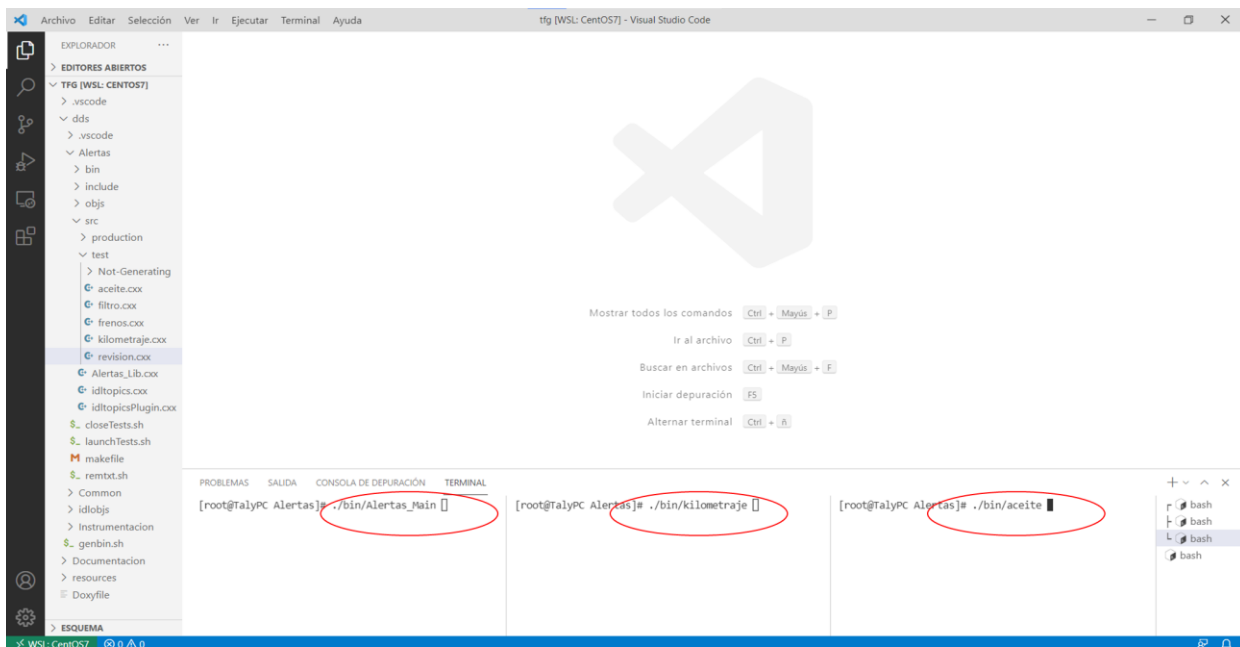

Figura 22. Instrucciones para simular una alerta de cambio de aceite.

Se ha de decir que los mantenimientos han sido programados de forma que el primero es el cambio de aceite a 15000 km, luego vendría el cambio de pastillas de freno a los 20000 km, le seguiría el reemplazo del filtro de combustiblea los 25000 km y terminaría por la revisión general del vehículo a los 30000 km. El publicador del kilometraje está configurado para que comience a contar a partir de 13000 km de 100 en 100 km, pero cada cierto tiempo deja de ser lineal y cuenta en aumento para acelerar el proceso de pruebas.

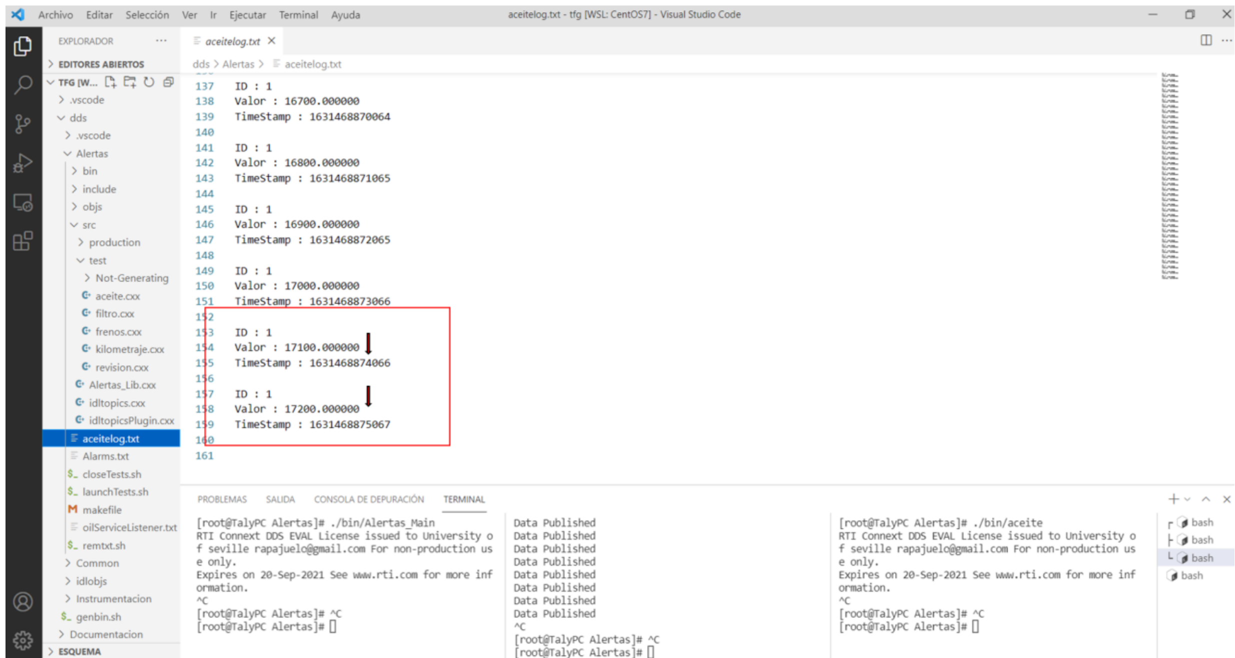

Figura 23. Registro del kilometraje para el cambio de aceite.

Como se venía diciendo desde capítulos anteriores, cada servicio de mantenimiento tiene su propio registro de kilometraje para evitar problemas en la lectura y escritura de datos. Es importante remarcar la sincronicidad que debe tener todo para que el sistema pueda funcionar perfectamente en tiempo real, y en la *Figura 23*, puede verse como el error en tiempo vuelve a ser menor a la milésima de segundo.

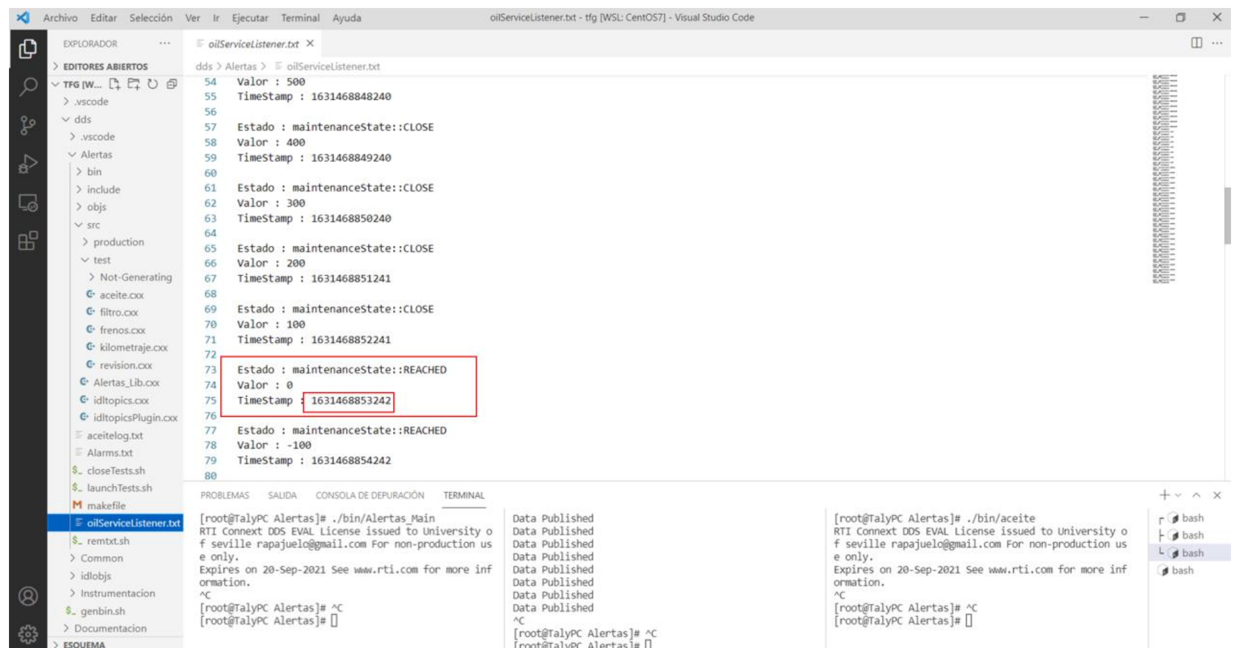

Figura 24. Registro instantáneo de estados de alerta del cambio de aceite.

Para comprobar su correcto funcionamiento, no se registran únicamente los eventos de las alertas de mantenimiento o el kilometraje, sino que también se puede tener acceso a un registro instantáneo de los datos recibidos.

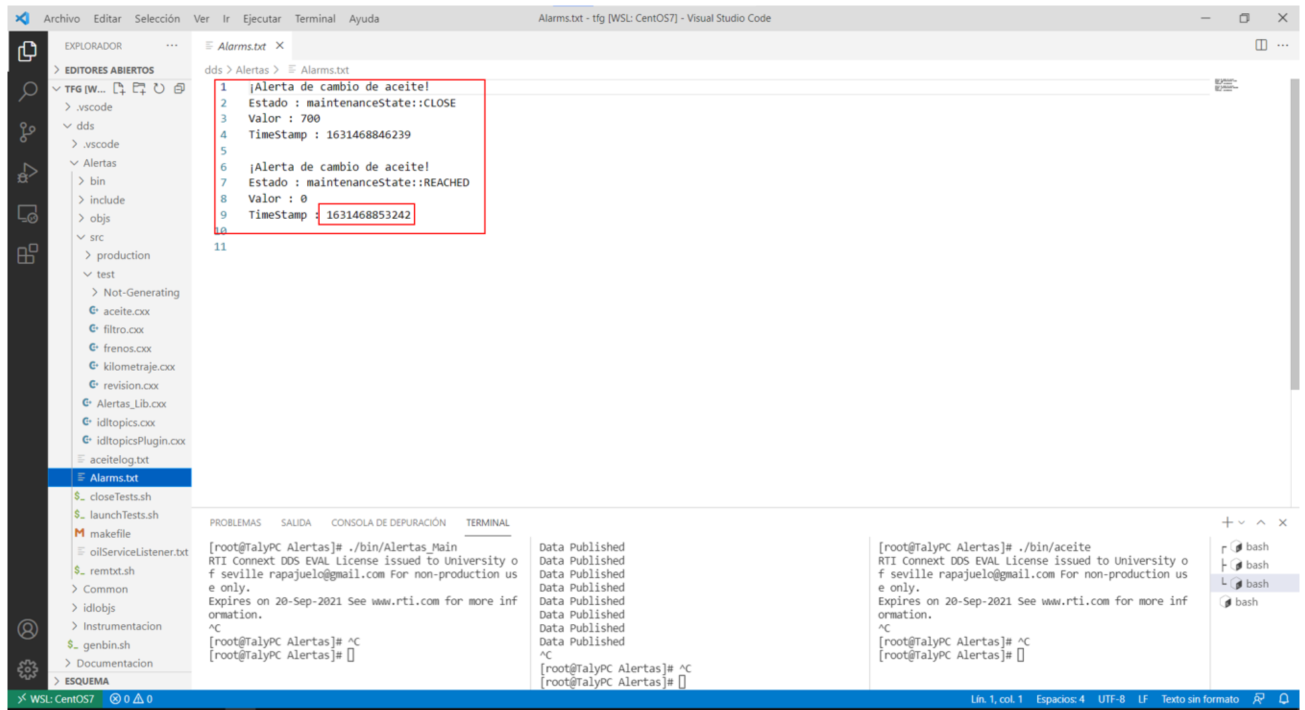

Figura 25. Registro de alertas de cambio de aceite.

Caza vez que se produce un evento de cambio en el estado de cualquier alerta de mantenimiento esta se guarda en el fichero "Alarmas.txt". Si se comparan los timestamps de las *Figuras 24* y *25*, puede verse que la alerta registrada tiene el mismo marcado temporal.

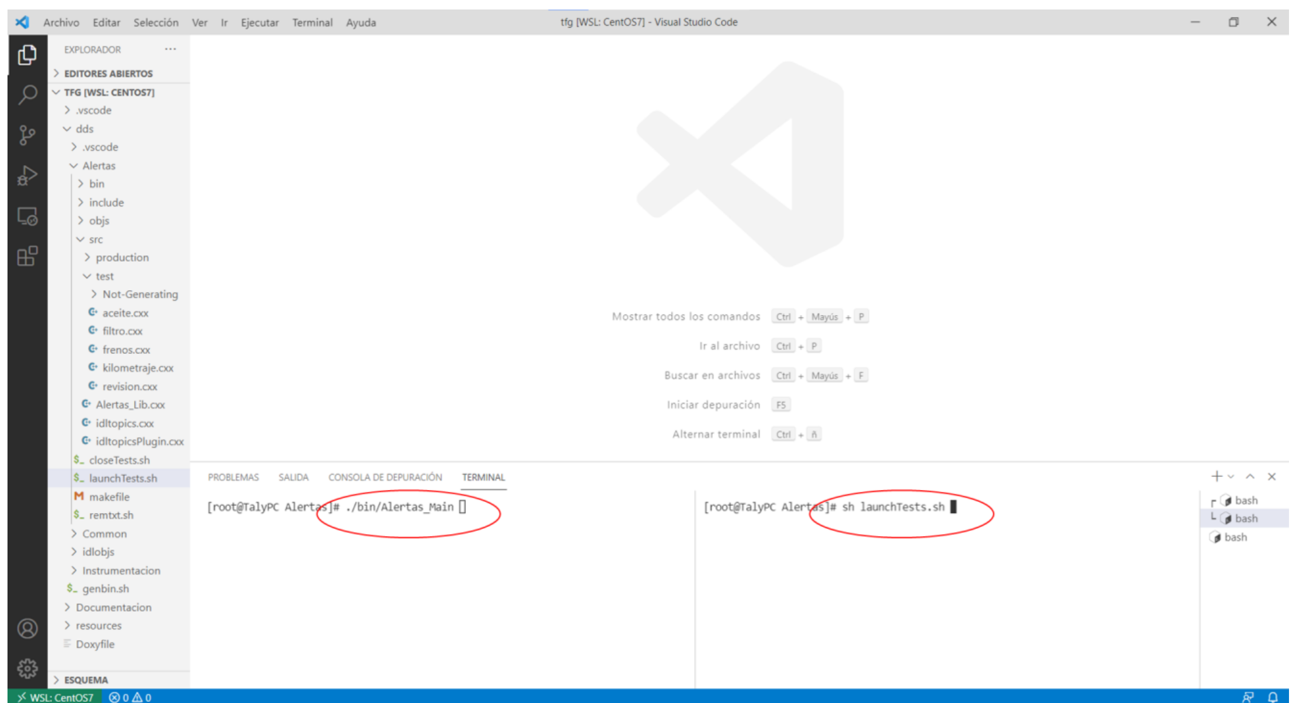

Figura 26. Instrucciones para la ejecución de todas las aplicaciones en el dominio Alertas.

Ahora que se ha podido comprobar el correcto funcionamiento con una única señal, se procede a realizar un ensayo con todas las señales del dominio activas, para ello es necesario escribir las instrucciones mostradas en la *Figura 26*.

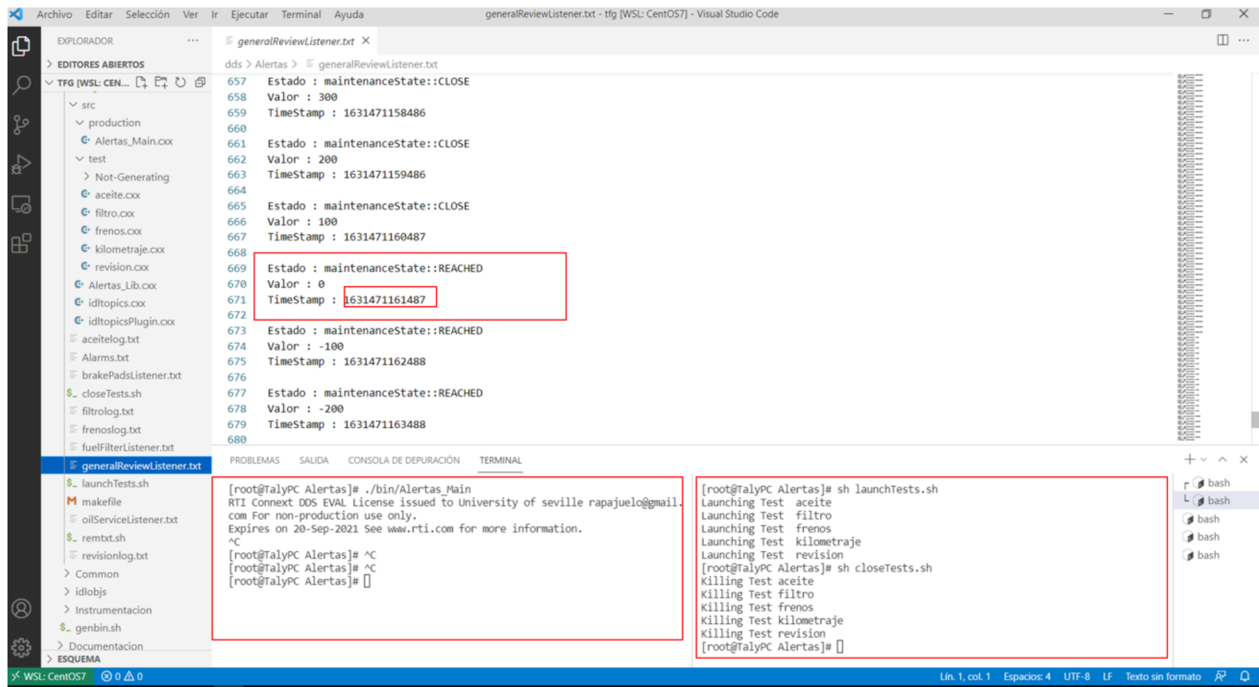

Figura 27. Registro instantáneo de las alertas de revisión general del vehículo.

Al igual que antes, puede comprobarse que ambas muestras cuando alcanzan la última alerta de servicio coinciden en tiempo.

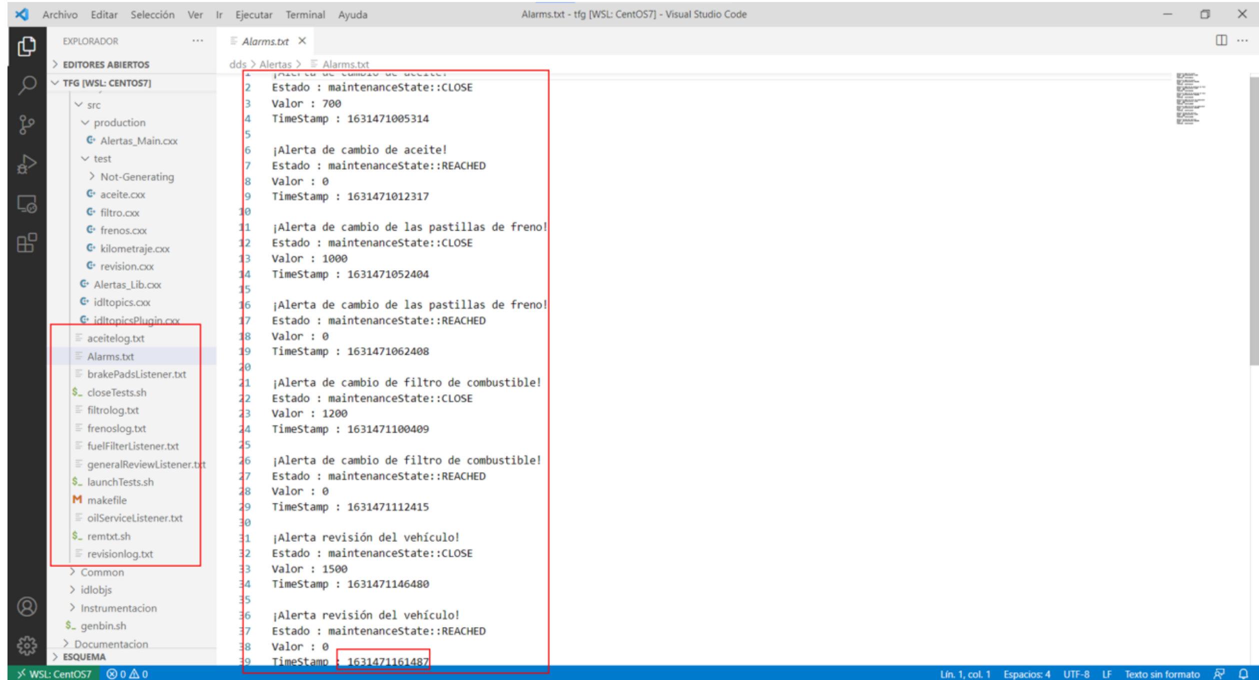

Figura 28. Registro de eventos de alertas de servicio.

Tras la ejecución del ensayo, puede verse la cantidad de ficheros que se han fabricado, siendo esto buena señal de que todo el proceso ha ido según lo esperado. Echando un vistazo a la *Figura 28*, puede verse que toda y cada una de las alertas programadas en el ensayo ha ido según lo esperado, verificando el correcto funcionamiento del sistema en su totalidad.

6 CONCLUSIONES

En el transcurso del proyecto se han abordado numerosos temas, empezando por qué es un sistema distribuido, qué tipos hay y por qué son tan importantes los sistemas distribuidos en tiempo real. Luego, se ha puesto en conocimiento qué es DDS y por qué este estándar es de suma importancia en comunicaciones del mundo real. Se han definido unos objetivos en el entorno de un vehículo genérico, en el que un sistema embebido concectado a diferentes unidades de control electrónicas del es capaz de monitorizar y registrar la información proveniente de los diferentes sensores asociados a estas. Para realizar dichas tareas ha sido necesario el desarrollo de varias aplicaciones que puedan correr sobre un sistema operativo en tiempo real basado en microprocesador. Es por esto por lo que todo el desarrollo se ha hecho utilizando el lenguaje de programación C++ 11 bajo el sistema operativo Linux. Una vez definida la arquitectura de trabajo se propuso un diseño asociado a la programación orientada a objetos, con la inclusión de clases que facilitasen la mecánica del desarrollo. Tras el diseño se pasó a su implementación, gracias al sofware de prueba RTI Connext DDS, y finalmente se propuso una demostración en la que varias aplicaciones ubicadas en diferentes dominios DDS podían compartir y registrar datos en tiempo real, de forma confiable y cumpliendo con todas las especificaciones temporales propuestas.

Una vez finalizado el proyecto, se ha podido comprobar que se ha conseguido implementar con éxito una arquitectura DDS. Para ello, como se ha visto anteriormente, se ha tenido que recurrir a muchos conocimientos sobre el middleware, sobre los sistemas distribuidos en tiempo real, programación en C++ y programación orientada a objetos entre otros; proporcionando al autor una excelente formación en el desarrollo de aplicaciones distribuidas. Esta formación nunca hubiese tenido lugar sin contar con la colaboración del GIE.

El trabajo realizado en este proyecto es perfectamente adaptable a soluciones reales dentro de la industria, y en concreto dentro de la industria automotriz como ha podido verse. En el proyecto queda muy bien reflejado el flujo de trabajo que hay que seguir desde el estudio del propio estándar DDS hasta su implementación, por lo que servirá de mucha utilidad a aquellos que se adentren en la exploración de soluciones que impliquen comunicaciones entre sistemas distribuidos en tiempor real. Dicho trabajo podrá tener aplicaciones más allá de la industria automotriz, ya que podrá usarse en aeronáutica, defensa, bolsa, etc …

Para líneas futuras de investigación se propone hacer uso de bases de datos para el registro de señales, así como la implementación sobre equipos verdaderamente remotos y en entornos de mucha susceptibilidad a interferencias.También se propone ampliar la comunicación a servicios basados en la nube y comunicaciones 5G.

DDS ha venido al mundo para quedarse, y gracias a trabajos como este, hoy queda más cerca de posibles futuros usuarios para emprender soluciones más eficientes en sus proyectos.

GLOSARIO

RTS: Real Time System	1
PLC: Programmable Logic Controller	1
UML: Unified Modeling Languaje	1
RTOS: Real Time Operating System	1
USB: Universal Serial Bus	2
OMG: Object Managmnt Group	3
DDS: Data Distribution Service for real-time systems	3
GDS: Global Data Space	3
ECU: Electronic Control Unit	3
MQTT: Message Queing Telemetry Transport	3
DAEMON: Disk And Execution Monitor	3
XML: Extensible Markup Language	3
IDL: Interfaz Definition Languaje	3
XSD: XML Schema Definition	3
HTTP: Hypertext Transfer Protocol	3
SOAP: Simple Object Access Protocol	3
REST: Representational State Transfer	3
CAN: Controller Area Network	4
SQL: Structured Query Languaje	11
PSM: Platform Specific Model	16

BIBLIOGRAFÍA

[1] https://es.wikipedia.org/wiki/Data_Distribution_Service

[2] https://www.rti.com/products/dds-standard

[3] https://www.utpl.edu.ec/proyectomiddleware/sites/default/files/files/tutorial-dds.pdf

[4] https://www.omg.org/spec/DDSI-RTPS/2.3/Beta1/PDF

[5] https://www.omg.org/news/meetings/workshops/RT-2010-Presentations/QoSUnleashedTutorial.pdf

[6] https://es.wikipedia.org/wiki/Sistema_de_tiempo_real

[7] http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion_de_referencia_ISE4_2_2.pdf

[8] http://bibing.us.es/proyectos/abreproy/12055/fichero/Cap%C3%ADtulo_3.pdf

[9] https://www.ingenieriaymecanicaautomotriz.com/que-es-el-can-bus-y-como-funciona/

[10] https://www.rti.com/products/what-is-a-databus

[11] http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion_de_referencia_ISE4_3_3.pdf

[12] https://www.rfwireless-world.com/Terminology/DDS-protocol-architecture.html

[13]https://community.rti.com/static/documentation/connext-dds/6.0.1/doc/api/connext_dds/api_cpp2/modules.html