Facultad 5

Título:

“Subsistema de comunicaciones para el SCADA SAINUX”.

TRABAJO DE DIPLOMA PARA OPTAR POR EL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS

Autores: Yadiel Martinez González
Yanelys del Rosario Lalcebo

Tutor: Ing. José Antonio Aragón Cáceres
Co-tutor: Ing. Yolier Galán Tassé

2012
“Año 54 de la Revolución”

"El genio comienza las grandes obras, pero sólo el trabajo las acaba"

Joseph Joubert (1754-1824)

DECLARACIÓN DE AUTORÍA

Declaramos ser los únicos autores del presente trabajo de diploma y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmamos la presente a los ___ días del mes de ________ del año _______.

______________________________	______________________________
Firma del Autor	Firma del Autor
Yadiel Martinez González	Yanelys del Rosario Lalcebo

____

Firma del Tutor

Ing. José Antonio Aragón Cáceres

DATOS DE CONTACTO

Nombre y apellidos del tutor: José Antonio Aragón Cáceres.

Institución: Universidad de las Ciencias Informáticas (UCI).

Título: Ingeniero en Ciencias Informáticas.

e-mail: jaaragon@uci.cu

Especialista graduado en la Universidad de las Ciencias Informáticas (UCI), profesor instructor con 3 años de experiencia docente y 6 años de experiencia en la producción de software, específicamente en el desarrollo de sistemas SCADA. Arquitecto principal del SCADA-UX y líder de la línea de desarrollo de Comunicaciones del Departamento de Construcción de Componentes del CEDIN.

AGRADECIMIENTO

A la Revolución cubana por contribuir a mi formación profesional.

A mis padres que con tanto amor y esfuerzo han sabido guiarme siempre por el camino correcto, a quienes le debo todo lo que soy.

A mi novio Yenier por su amor, apoyo y comprensión durante estos 5 años, quien ha sido mi motor impulsor a cosechar cada uno de mis logros en esta carrera.

A toda mi familia por quererme tanto y apoyarme siempre durante toda mi vida.

A mis profesores por darme las herramientas necesarias para desempeñarme como profesional.

A mis tutores Tony y Yolier por su guía y apoyo incondicional para la realización de este trabajo.

A Yadiel por ser un compañero estupendo.

A mis amigos, compañeros de grupo y a todos los que han contribuido a mi graduación.

Muchas gracias.

Yanelys

A mi familia, por su incondicional apoyo.

A Leanys por quererme tanto.

A mi piquete Hecty, Lito y Jonnys por ser amigos y hermanos.

A los que han quedado tras esta etapa de mi vida como Claudia, Yadira, Karel.

A Tony y Yolier por su gran ayuda en la realización de este trabajo.

A Yanelys por ser una excelente compañera.

A todos gracias.

Yadiel

DEDICATORIA

A mis padres, a Yenier y a toda mi familia.

Yanelys

A mi madre, por su extraordinario amor, por ser mi guía, mi luz, mi todo.

A Mima y el Piro, por cada segundo presentes.

A Tania, por aguantar a mis malcriados primos.

Al Yuri, por premiarme con su cariño.

Yadiel

RESUMEN

El presente trabajo muestra el desarrollo de un subsistema de comunicaciones para el SCADA SAINUX, el cual posibilita el envío y recepción de datos entre los distintos módulos distribuidos que componen dicho SCADA.

Inicialmente se llevó a cabo una investigación sobre las tecnologías de comunicación distribuida en sistemas SCADA, lo que permitió obtener una idea general sobre el estado del arte de las mismas. Además la investigación estuvo enfocada principalmente en que dichas tecnologías utilicen herramientas libres que permitan la comercialización, para contribuir a que SAINUX pueda aportar beneficios económicos al país.

Finalmente se seleccionó *OpenDDS*, *middleware* utilizado en sistemas cuyos requisitos incluyen: distribución de datos, robustez, capacidad de tiempo real, tolerancia a fallos, y además está liberado bajo la licencia pública general menor (LGPL), que permite la creación de un software comercializable. Se definieron los requisitos funcionales y no funcionales para el desarrollo del subsistema, así como el modelado de la solución a partir de estos. Se implementó el diseño definido y se realizaron las pruebas a las funcionalidades más importantes, lo que permitió validar la tecnología y el modelado propuesto a través de los resultados satisfactorios de las mismas.

PALABRAS CLAVE:

Comunicación distribuida, *middleware*, publicación-suscripción, SCADA, subsistema de comunicación.

ÍNDICE

INTRODUCCIÓN 1

CAPÍTULO I: "FUNDAMENTACIÓN TEÓRICA Y SELECCIÓN DE TECNOLOGÍAS". 5

1.1. INTRODUCCIÓN. 5
1.2. SISTEMAS SCADA. 5
1.2.1. Principales funcionalidades. 6
1.2.2. Módulos. 6
1.2.3. Principales datos que se manejan en un SCADA. 7
1.3. SISTEMAS SCADA DISTRIBUIDOS. 8
1.3.1. Características. 8
1.3.2. Ventajas. 9
1.4. *MIDDLEWARE* EN SISTEMAS SCADA. 9
1.4.1. Beneficios proporcionados por el Middleware. 11
1.5. TENDENCIAS Y TECNOLOGÍA DE COMUNICACIÓN DISTRIBUIDA ACTUALES. 12
1.5.1. Estándares de comunicación distribuida. 12
- *Java Remothe Method Invocation (Java RMI)* 12
- *Common Object Request Broker Architecture (CORBA).* 13
- *DDS.* 15

1.5.2. Middlewares 17
- *ORBit.* 17
- *ACE + TAO.* 18
- *OpenDDS* 19

1.6. SELECCIÓN DE TECNOLOGÍAS PARA EL DESARROLLO DEL SUBSISTEMA DE COMUNICACIONES.. 20
1.6.1. Criterios de evaluación. 20
1.6.2. Análisis de opciones tecnológicas. 23
1.7. TECNOLOGÍA PROPUESTA PARA EL DESARROLLO DEL SUBSISTEMA DE COMUNICACIONES. 24
1.8. METODOLOGÍA, LENGUAJES Y HERRAMIENTAS PARA EL DESARROLLO DEL SUBSISTEMA DE COMUNICACIONES. 25

CAPÍTULO II: MODELADO DE LA SOLUCIÓN. 27

2.1. INTRODUCCIÓN. 27
2.2. REQUISITOS DEL SUBSISTEMA DE COMUNICACIONES. 27
2.2.1. Requisitos funcionales. 27
2.2.2. Requisitos no funcionales. 28
2.3. DIAGRAMA DE CASOS DE USO DEL SISTEMA. 30
2.3.1. Descripción del caso de uso Enviar_Datos. 30
2.3.2. Descripción del caso de uso Recibir_Datos. 31
2.4. ARQUITECTURA DEL ESTÁNDAR DDS. 32
2.4.1. Capa DCPS 32
2.4.2. Capa DLRL 33
2.5. MODELO DE MENSAJES PUBLICACIÓN/SUSCRIPCIÓN. 34
2.6. MODELO DE DISEÑO DEL SUBSISTEMA DE COMUNICACIÓN DE SAINUX. 34
2.6.1. publisher_subscriber: 35
2.6.2. lifetime_policy: 37
2.6.3. comm_traits: 38
2.6.4. communication_data_types: 40
2.7. PATRONES DE DISEÑO UTILIZADOS. 40
2.7.1. Traits (rasgos o características) 41
2.7.2. Policy (políticas) 41
2.7.3. Facade (fachada) 42
2.8. DIAGRAMA DE SECUENCIA: *PUBLISHER*. 42
2.9. DIAGRAMA DE SECUENCIA: *SUBSCRIBER*. 43

CAPÍTULO III: "IMPLEMENTACIÓN Y PRUEBAS". 44

3.1. INTRODUCCIÓN. 44
3.2. MODELO DE IMPLEMENTACIÓN. 44
3.2.1. Diagrama de componentes del paquete publisher_subscriber. 44
3.2.2. Diagrama de componentes del paquete lifetime_policy. 45
3.2.3. Diagrama de componentes del paquete comm_traits. 45
3.2.4. Diagrama de componentes del paquete communication_data_types. 46
3.2.5. Despliegue del subsistema de comunicaciones de SAINUX. 46
3.3. ESTILO DE CÓDIGO. 47
3.3.1. Nombres. 47
3.3.2. Manejo de Errores. 48
3.3.3. Documentación y Comentarios. 48
3.3.4. Codificación. 49
3.4. VISTAS MÁS SIGNIFICATIVAS DEL CÓDIGO. 50
3.4.1. Enviar datos según el nombre del tópico. 50
3.4.2. Crear tópico y esperar por un publicador. 51
3.5. PRUEBAS. 51
3.5.1. Ambiente de pruebas. 52
3.5.2. Especificación de escenarios de prueba. 52
3.5.3. Casos de pruebas. 53
3.5.4. Análisis de los resultados de pruebas. 55

CONCLUSIONES 56

RECOMENDACIONES 57

REFERENCIAS BIBLIOGRÁFICAS 58

ANEXOS 61

GLOSARIO 63

Índice de figuras

Figura 1: Operarios de un sistema SCADA interactuando con la interfaz hombre-máquina........ 5
Figura 2: Pirámide de Automatización de un SCADA.. 11
Figura 3: Matriz de decisión de tecnología para la implementación del subsistema de comunicaciones de SAINUX 25
Figura 4: Diagrama de casos de uso del sistema 30
Figura 5: Funcionamiento de la capa DCPS... 33
Figura 6: Modelo de Publicación/Suscripción 34
Figura 7: Diagrama de paquetes del subsistema de comunicación de SAINUX 34
Figura 8: Diagrama de clases del paquete: publisher_subscriber 35
Figura 9: Diagrama de clases del paquete lifetime_policy... 37
Figura 10: Diagrama de clases del paquete comm_traits.. 38
Figura 11: Diagrama de secuencia de publisher... 42
Figura 12: Diagrama de secuencia de subscriber... 43
Figura 13: Diagrama de componentes del paquete publisher_subscriber 44
Figura 14: Diagrama de componentes del paquete lifetime_policy 45
Figura 15: Diagrama de componentes del paquete comm_traits 45
Figura 16: Diagrama de componentes del paquete communication_data_types 46
Figura 17: Despliegue del subsistema de comunicaciones 47
Figura 18: Fragmento de código del método send_by_topic_name() 50
Figura 19: Fragmento de código de los métodos create_subscriber_topic() y wait_for_publisher() 51

Índice de tablas

Tabla 1: Actores del sistema. 30
Tabla 2: Descripción del caso de uso Enviar_Datos.. 31
Tabla 3: Descripción del caso de uso Recibir_Datos... 31
Tabla 4: Descripción de la clase: basic_communication. 35
Tabla 5: Descripción de la clase publisher.. 36
Tabla 6: Descripción de la clase write_actions. 36
Tabla 7: Descripción de la clase subscriber.. 37
Tabla 8: Descripción de la clase lifetime_policy... 37
Tabla 9: Descripción de la clase ITopic. 38
Tabla 10: Descripción de la clase Topic. 38
Tabla 11: Descripción de la clase communication_traits. 39
Tabla 12: Descripción de la clase communication_traits_p <publisher_var>. 39
Tabla 13: Descripción de la clase communication_traits_s <subscriber_var>. 39
Tabla 14: Descripción de la clase metadata_traits. 40
Tabla 15: Descripción de la clase metadata_traits_p <point>. 40
Tabla 16: Resultados de pruebas para el envío y recepción de datos complejos de tipo punto. 55
Tabla 17: Elementos de la matriz de decisión. 61
Tabla 18: Valores cuantitativos de las calificaciones a las tecnologías..................................... 61
Tabla 19: Modelo de decisión de la tecnología más apropiada. 62

INTRODUCCIÓN

El desarrollo y evolución de los medios de producción con el paso del tiempo ha sido tal, que actualmente son capaces de sustituir al hombre en disímiles labores, realizando tareas productivas de manera autónoma, con una eficiencia prácticamente total. Estrechamente ligado al desempeño de estos medios, se encuentran los sistemas para la Supervisión, Control y Adquisición de Datos (SCADA), cuyo objetivo fundamental es la vigilancia y manejo de procesos industriales a cualquier escala.

Una de las características fundamentales de este tipo de sistemas es que, por lo general, se encuentran geográficamente distribuidos, es decir, están compuestos por varios componentes tanto de hardware como de software que se encuentran conectados en red y cada uno de ellos tiene una función específica dentro del sistema, los cuales, a través de la comunicación y coordinación entre sí, permiten que este funcione como una entidad única. Sin embargo, en ocasiones, lograr la interacción entre estos componentes, resulta muy engorroso, ya que no necesariamente deben estar soportados por la misma plataforma, ni utilizar el mismo lenguaje de programación o el mismo sistema operativo, por lo que el medio de comunicación que se utilice, debe ser capaz de resolver este tipo de problemas. **[1]**

Los software que se encargan de las comunicaciones en los sistemas distribuidos se conocen como "*middleware*" y su uso es muy amplio a nivel mundial. Los *middleware* son software de conectividad que ofrecen un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. Dependiendo del problema a resolver y de las funciones necesarias, serán útiles diferentes tipos de servicios de *middleware*. Estas funciones les son asignadas durante su desarrollo y dependen mucho de la tecnología que se utilice por lo que es importante realizar una correcta selección de la misma para que estos puedan satisfacer los servicios que debe brindarle al sistema en el cual van a ser acoplados. **[1]**

El Centro de Informática Industrial (CEDIN) perteneciente a la facultad 5 de la Universidad de las Ciencias Informáticas, cuya misión es desarrollar productos y servicios informáticos de automatización industrial, se encuentra inmerso en

el desarrollo de un nuevo producto llamado SAINUX Sistema de Automatización Industrial UX (estas últimas siglas provienen del desarrollo de aplicaciones sobre el sistema operativo Linux). Con este sistema se pretende automatizar procesos de las empresas que lo requieran en el país y además poder comercializarlo en el exterior para obtener divisas y así contribuir al desarrollo económico. SAINUX tuvo un producto antecesor que fue el SCADA UX, el cual se encuentra instalado en el acueducto de Santiago de Cuba para la supervisión y control de este proceso, garantizar la eficiencia y contribuir a la solución del problema de abasto de agua en dicha provincia.

Con el objetivo de ahorrar tiempo y esfuerzos, SAINUX toma varios módulos del SCADA UX que no contengan tecnologías y herramientas privativas en su desarrollo; pero uno de los módulos fundamentales para el funcionamiento de un SCADA es el *middleware*. Este fue desarrollado en un inicio, utilizando la tecnología *Internet Communication Engine* (ICE), una plataforma para desarrollo de aplicaciones de comunicación de alto rendimiento, que incluye varias capas de servicios y *plugin*. ICE es un software liberado bajo la Licencia Publica General (GPL), aunque también está disponible la adquisición de una licencia comercial, para quienes no desean utilizar la tecnología sujetos a los términos de GPL.

El uso de la distribución libre de ICE en la implementación del módulo de comunicación del SCADA UX, imposibilita la utilización del mismo en la implementación de SAINUX debido a los términos de su licencia GPL, impidiendo de esta manera, cualquier apoyo al progreso de la economía nacional que pudiera proporcionar el mercadeo del software, y aunque es posible adquirir una licencia comercial de la tecnología, esta no constituye una opción económicamente viable, debido al alto coste de la misma. Por tales motivos no es factible utilizar el *middleware* del SCADA UX y se hace necesario desarrollar un nuevo subsistema de comunicación entre los diferentes módulos de SAINUX, basado en otra tecnología, que permita gestionar el intercambio de información entre los mismos.

Por todo lo expuesto anteriormente se identificó el siguiente **problema a resolver**: ¿Cómo garantizar una solución de comunicación distribuida para SAINUX que permita la introducción del producto en el mercado nacional e internacional?

Del problema anterior, se define como **objeto de estudio:** las tecnologías de comunicación distribuida en sistemas SCADA y el **campo de acción** es: el subsistema de comunicación de SAINUX. De acuerdo con el problema planteado la **idea a defender** es la siguiente:

Si se desarrolla el subsistema de comunicaciones para gestionar el intercambio de información entre los procesos distribuidos de SAINUX y se utiliza en su desarrollo una tecnología que permita la comercialización, entonces se proveerán los mecanismos de envío y recepción de datos entre los distintos módulos del sistema; y además se obtendrá un producto que beneficiará tanto a empresas cubanas como al desarrollo económico del país.

Se persigue como **objetivo general:** Desarrollar un subsistema de comunicaciones basado en tecnologías *Middleware* existentes, que permita el intercambio de información entre los procesos distribuidos de SAINUX y garantice la instalación del producto en el ámbito nacional, así como su comercialización en otros países.

Definiéndose las siguientes **tareas de la investigación** para lograr un mayor nivel de precisión en la implementación:

- ✓ Estudio de los temas relacionados con la comunicación en sistemas SCADA distribuidos.
- ✓ Estudio de las licencias que poseen las tecnologías de comunicación distribuida.
- ✓ Evaluación y selección de las tecnologías adecuadas para la implementación del subsistema de comunicaciones del sistema SAINUX.
- ✓ Modelación a través del diseño las funcionalidades relacionadas con el subsistema de comunicaciones del sistema SAINUX.
- ✓ Implementación de las funcionalidades relacionadas con el subsistema de comunicaciones del sistema SAINUX.
- ✓ Integración y prueba de las funcionalidades relacionadas con el subsistema de comunicaciones del sistema SAINUX.

Para el desarrollo de las tareas de la investigación se combinan diferentes **métodos de investigación** y técnicas en la búsqueda y procesamiento de la información, los fundamentales son:

A nivel teórico

- ✓ **Análisis-síntesis:** Utilizado para analizar las teorías y documentos generados por desarrolladores que usen herramientas para el tratamiento de aplicaciones distribuidas permitiendo la extracción de los elementos que se relacionen con la implementación del subsistema de comunicaciones del SCADA SAINUX.
- ✓ **Análisis histórico-lógico:** Utilizado para conocer, con mayor profundidad los antecedentes y las tendencias actuales referidas al origen, desarrollo y aplicación de los sistemas de comunicación distribuida en sistemas SCADA.

A nivel empírico

- ✓ **Experimentos:** Empleado en la elaboración de prototipos funcionales, con el objetivo de comprobar la efectividad de la implementación de las funcionalidades del subsistema de comunicaciones del sistema SAINUX.

El presente trabajo de diploma estará estructurado de la siguiente forma:

Capítulo 1 "Fundamentación teórica y selección de tecnologías”. En este capítulo se brinda una serie de definiciones y conceptos de sistema SCADA y *middleware* que servirán como marco teórico para la comprensión del entorno en el cual se desarrollan las actividades, además se realiza un estudio y análisis de las tendencias y tecnologías de comunicación distribuida actuales, con vistas a la selección de la mejor opción tecnológica para el desarrollo del subsistema de comunicaciones de SAINUX.

Capítulo 2 " Análisis y diseño de la solución". Durante este capítulo se analizan los requisitos funcionales y no funcionales con los que debe cumplir el sistema y además se modela el diseño para el desarrollo del subsistema de comunicaciones de SAINUX con el objetivo de crear una entrada apropiada para las actividades de implementación.

Capítulo 3 "Implementación y prueba". Se muestran las vistas de implementación y despliegue, así como las pruebas realizadas al subsistema de comunicaciones de SAINUX.

Capítulo I: “Fundamentación teórica y selección de tecnologías”.

1.1. Introducción.

En el presente capítulo se muestran algunos de los principales elementos relacionados con los sistemas SCADA, como son su descripción, funcionalidades y módulos, así como los tipos de SCADA que existen. Se profundiza en el tema de *middleware*, abordando los beneficios que este brinda en un sistema SCADA. También se realiza un estudio de las tendencias y tecnologías de comunicación distribuida actuales de mayor auge hasta el momento en que se inicia la investigación. Se realiza la selección de la tecnología de comunicación distribuida a usar para la posterior implementación y se describen la metodología, lenguajes y herramientas que se utilizarán durante el desarrollo.

1.2. Sistemas SCADA.

A decir de Javier García Giménez, los sistemas SCADA, acrónimo de *Supervisory Control And Data Adquisition*, utilizan el computador y las tecnologías de comunicación para automatizar el monitoreo y control de procesos industriales. Estos sistemas son partes integrales de la mayoría de los ambientes industriales complejos o geográficamente dispersos ya que pueden recoger la información de una gran cantidad de fuentes rápidamente, y la presentan a un operador en una forma amigable. **[2]**

Se define entonces que un sistema SCADA es la combinación de un conjunto de hardware y software que permite la colección y visualización de los datos proporcionados por dispositivos de adquisición.

Figura 1: Operarios de un sistema SCADA interactuando con la interfaz hombre-máquina

1.2.1. Principales funcionalidades.

Las principales funcionalidades ofrecidas por los SCADA de manera general son las siguientes: **[3]**

- **Adquisición de datos:** para recolectar, procesar y almacenar la información recibida.
- **Supervisión:** para observar desde una computadora el estado de las instalaciones y las labores de producción en el campo.
- **Control:** para realizar ajustes en el proceso actuando sobre los reguladores autónomos básicos (alarmas, eventos, entre otros).
- **Transmisión de datos:** para la transmisión de información entre dispositivos de campo y otras computadoras.
- **Presentación de la información:** para la representación gráfica de los datos. Interfaz Hombre-Máquina (IHM).

1.2.2. Módulos.

Los módulos o bloques software que permiten las actividades de adquisición, supervisión y control en un sistema SCADA son los siguientes: **[4]**

- ✓ **Configuración:** permite al usuario definir el entorno de trabajo de su SCADA, adaptándolo a la aplicación particular que se desea desarrollar.
- ✓ **Interfaz gráfico del operador:** proporciona al operador las funciones de control y supervisión de la planta. El proceso se representa mediante sinópticos gráficos almacenados en el ordenador de proceso y generados desde el editor incorporado en el SCADA o importados desde otra aplicación durante la configuración del paquete.
- ✓ **Módulo de proceso:** ejecuta las acciones de mando pre-programadas a partir de los valores actuales de variables leídas.
- ✓ **Gestión y archivo de datos:** se encarga del almacenamiento y procesado ordenado de los datos, de forma que otra aplicación o dispositivo pueda tener acceso a ellos.
- ✓ **Comunicaciones:** se encarga de la transferencia de información entre la planta y la arquitectura hardware que soporta el SCADA, y entre esta y el resto de elementos informáticos de gestión.

1.2.3. Principales datos que se manejan en un SCADA.

- ✓ **Puntos:**

El flujo principal de información en los sistemas SCADA lo constituyen las variables (puntos). Estas variables pueden representar innumerables indicadores como son: presión, temperatura, flujo, potencia, peso, intensidad de corriente, voltaje, potencial hidrógeno, densidad, carga, resistencia o capacitancia entre otros. Las variables son adquiridas mediante instrumentación o utilizando sensores conectados a autómatas o equipos de control. Después de convertidas a señales eléctricas, estas variables pasan a ser estructuras que contienen datos, los que pueden ser de tipos simples (entero, flotante, cadenas, y otros) o de tipos complejos. La información de estas variables permite conocer el estado del sistema y su historia. **[5]**

- ✓ **Alarmas:**

Las alarmas se basan en la vigilancia de los parámetros de las variables del sistema. Son los sucesos no deseables, porque su aparición puede dar lugar a problemas de funcionamiento. Este tipo de sucesos requieren de la atención de un operario para su solución antes de que se llegue a una situación crítica que detenga el proceso. **[5]**

- ✓ **Eventos:**

El resto de las situaciones normales, tales como puesta en marcha, paro, cambios de consignas de funcionamiento, consultas de datos, entre otras, serán los denominados eventos del sistema o sucesos. Los eventos no requieren de la atención del operador del sistema, registran de forma automática todo lo que ocurre en el sistema. También será posible guardar estos datos para su posterior consulta. **[5]**

- ✓ **Comandos:**

Acción que se ejecuta en sistemas SCADA que modifican los valores recolectados de campo. Estos pueden ser desde acciones para modificar el valor de una variable hasta el reconocimiento de una alarma.

- ✓ **Estado de la comunicación:**

Es la información relacionada con los estados de los dispositivos, canales y sub – canales.

- ✓ **Bitácoras:**

Hechos ocurridos durante la utilización del sistema que puedan ser aprovechados como experiencias en el futuro.

1.3. Sistemas SCADA distribuidos.

Existen diversos tipos de sistemas SCADA en dependencia del fabricante y sobre todo de la finalidad, entre los cuales podemos encontrar: **sistemas monolíticos**, fueron los primeros sistemas SCADA, llevan a cabo las operaciones en una computadora. Debido al poco control que ejercen, estos se limitan solo a una planta o instalación; **sistemas en red**, forman parte de la última generación de sistemas SCADA, por lo general se comunican a través de redes de área amplia, y la trasmisión de datos entre nodos se realiza a través de Ethernet o fibra óptica. Mientras que los primeros de los sistemas SCADA se limitaron solo a una instalación, muchos de estos sistemas, son capaces de ejercer control en ambientes realmente amplios y conectarse a internet, por lo que son potencialmente vulnerables a ataques cibernéticos. Por último se verán los **sistemas distribuidos**, para ello se presentan algunas definiciones:

- ✓ Colección de ordenadores autónomos enlazados por una red y soportados por aplicaciones que hacen que la colección actúe como un servicio integrado. **[6]**
- ✓ Sistema de computación con un número de componentes que cooperan entre sí comunicándose a través de la red. **[6]**

Finalmente, un SCADA distribuido es aquel sistema en que sus componentes tanto de hardware como de software se encuentran conectados en una red de área local, cada uno de estos, con una función específica dentro del sistema, los cuales, mediante la comunicación y coordinación entre sí, permiten el funcionamiento del sistema como una entidad única, cuyo objetivo fundamental es lograr el control y supervisión de los procesos industriales. Algunas de las características y ventajas que conllevan este tipo de sistemas, se exponen a continuación.

1.3.1. Características.

Una de las características más distintivas de los sistemas SCADA distribuidos es la de tener **recursos compartidos**, se refiere a que los recursos están

físicamente encapsulados dentro de uno de sus componentes y pueden ser accedidos por otros mediante comunicación. Otra de las virtudes que estos poseen, es la variedad existente entre los elementos que componen y participan en ellos, tanto en redes, hardware como sistemas operativos, conocida como **heterogeneidad**. Por lo general, son **sistemas abiertos**, es decir, extensibles. También permiten la **concurrencia** y la ejecución en paralelo cuando varios usuarios invocan simultáneamente comandos o interactúan con aplicaciones, así como el incremento de su escala sin la realización de grandes cambios, lo que en términos informáticos se conoce como **escalabilidad**.

Los sistemas distribuidos de manera general, presentan un alto grado de disponibilidad por lo que implementan mecanismos de **tolerancia a fallas**, que consiste en la capacidad del sistema, de continuar brindando servicio aún en caso de producirse algún fallo. La percepción de estos sistemas como un todo, más que una colección de componentes independientes, se conoce como **transparencia** y resulta de vital importancia tanto para el programador como para el usuario durante la construcción y uso del mismo.

1.3.2. Ventajas.

Notables son las mejorías proporcionadas por los sistemas SCADA distribuidos respecto a otros, entre ellas, está la posibilidad de dividir la lógica del sistema en componentes modulares y reusables en lugar de obtener un código monolítico. También, al distribuir el procesamiento, cada nodo del sistema tendrá menos complejidades, por lo que los requerimientos de hardware serán menos exigentes. Además no se requieren costosos sistemas de redes que demanden una configuración exhaustiva. El hecho que cada entidad es programada de tal forma que sea independiente, hace que sean sistemas escalables. Otra importante ventaja de este tipo de sistemas es que ofrecen la necesaria disponibilidad de los dispositivos, ya que cada componente está programado dentro del concepto de tolerancia a fallos.

1.4. *Middleware* en sistemas SCADA.

Con el devenir de los sistemas SCADA distribuidos y la necesidad de gestionar la comunicación entre sus componentes surgen los denominados *middleware*:

Según el doctor Douglas C. Schmidt, el *middleware* es un software de infraestructura que reside entre las aplicaciones y el sistema operativo, redes, y hardware subyacentes, específicamente intentando brindar una plataforma más apropiada para el desarrollo y ejecución de los sistemas distribuidos. **[7]**

Una definición bastante completa, puede encontrarse en el libro "Documentación de ZeroC ICE", donde se plantea: se puede entender un *middleware* como un software de conectividad que hace posible que aplicaciones distribuidas pueden ejecutarse sobre plataformas heterogéneas, es decir, sobre plataformas con distintos sistemas operativos, que usan distintos protocolos de red y que incluso, involucran distintos lenguajes de programación en la aplicación distribuida. Desde otro punto de vista, un *middleware* se puede entender como una abstracción en la complejidad y en la heterogeneidad que las redes de comunicaciones imponen. De hecho, uno de los objetivos de un *middleware* es ofrecer un acuerdo en las interfaces y en los mecanismos de interoperabilidad, como contrapartida de los distintos desacuerdos en hardware, sistemas operativos, protocolos de red y lenguajes de programación. **[8]**

Del análisis de las definiciones anteriores se puede concluir que el *middleware* en un sistema SCADA, es un software de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento integrado de un sistema distribuido y posibilita la abstracción necesaria para la programación y enmascaramiento de la heterogeneidad de las redes, hardware y sistema operativo.

En la pirámide de automatización de un sistema SCADA (ver figura 2), el *middleware*, es la capa de software que se encuentra por encima de los niveles físicos y de red y por debajo de las aplicaciones de usuario.

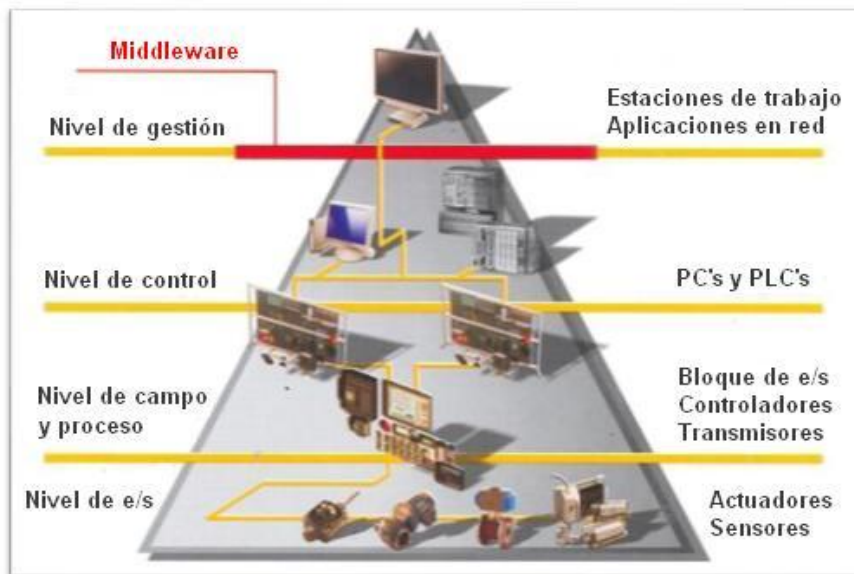

Figura 2: Pirámide de Automatización de un SCADA

1.4.1. Beneficios proporcionados por el *Middleware.*

Una correcta implementación de un *middleware* resulta de vital importancia para las aspiraciones de contar con un sistema SCADA lo suficientemente exitoso. Los beneficios que este ofrece, se abordan seguidamente.

El *middleware* libera a los desarrolladores de la programación de bajo nivel, abstrayéndolos de los minuciosos detalles de la plataforma y por ende, los protege de cometer errores y simplifica la implementación del sistema. También reduce los costos del ciclo de vida del software a través de la reutilización de los conocimientos de desarrollos anteriores, capturando implementaciones de patrones claves en *frameworks* reusables, y de esta forma evitar re-implementaciones innecesarias y provee un amplio y probado rango de servicios orientados al desarrollador para los ambientes de red, tales como, los de autenticación y seguridad.

El *middleware* está capacitado para proporcionar: **[9]**

- ✓ **Independencia entre el cliente y el servidor:** No necesitan saber comunicarse entre ellos, sino cómo comunicarse con el módulo de *middleware*.
- ✓ **Traducción de la información de una aplicación y el paso de dicha información a otra:** Acepta consultas y datos recuperándolos de la aplicación cliente, los transmite y envía la respuesta de regreso. También genera los códigos de error.

- ✓ **Control de las comunicaciones:** Da a la red las características adecuadas de desempeño, confiabilidad, transparencia y administración.

El *middleware* tiene varias funcionalidades, pero las claves de forma genérica serían: **[9]**

- ✓ **Gestión de dispositivos**: Los *middlewares* pueden en su mayoría controlar cualquier tipo de hardware, desde lectores a dispositivos actuales (entrada/salida). Puede controlar el funcionamiento y actualización de los dispositivos y avisar en caso de presentarse algún problema.
- ✓ **Procesamiento de datos**: Funciona como filtro de los datos recolectados para evitar lecturas múltiples y evitar sobrecargo de los sistemas. Puede configurar alertas y añadir información antes de pasarlas a los sistemas de gestión empresarial.
- ✓ **Conectar la información con las aplicaciones de negocio:** Los *middlewares* tienen herramientas que incluyen Interfaz de Programación de Aplicaciones (API) utilizadas por muchas empresas como puentes para conectar los datos con su software de negocios.

1.5. Tendencias y tecnología de comunicación distribuida actuales.

El *middleware*, así como las herramientas de desarrollo, sientan las bases para facilitar la construcción de un sistema SCADA, por lo que, la selección de la tecnología para desarrollar un subsistema de comunicación, es un punto clave en el diseño de la aplicación. Existen variadas tecnologías de comunicación posibles a utilizar por sus potencialidades y nivel de utilización global, entre ellas se encuentran estándares y *middlewares* ya desarrollados, que se analizarán a continuación.

1.5.1. Estándares de comunicación distribuida.

- ***Java Remothe Method Invocation (Java RMI)***

RMI proporciona una potente herramienta al programador Java poniendo a su alcance una capacidad de programación distribuida 100% Java. La idea básica de RMI es que, objetos ejecutándose en una máquina virtual (VM) sean capaces de invocar métodos de objetos ejecutándose en VM's diferentes. Haciendo notar que las VM's pueden estar en la misma máquina o en máquinas distintas conectadas por una red. En RMI, cuando los objetos

remotos ya han sido referenciados, el programador los puede utilizar igual que si estuviesen en la maquina local, accediendo a sus métodos y manejándolos de forma normal. La capa RMI permanece oculta al programador proporcionando mayor claridad al código y mayor comodidad a la hora de programar la aplicación. El paso de objetos por valor se revela como una de las características que diferencian a RMI de las otras tecnologías de programación distribuida. Esta potente posibilidad se basa en el uso del concepto de serialización para el paso de objetos a través de la red. Objetos de todo tipo, tanto nativos Java como definidos por el usuario, pueden ser transferidos de forma totalmente transparente al programador usando esta técnica. **[10]**

Características de RMI.

- ✓ Integra el modelo de objetos distribuidos dentro del lenguaje Java en un modo natural.
- ✓ Soporta invocaciones remotas de objetos en diferentes máquinas virtuales.
- ✓ Soporta paso de objetos por referencia y/o valor.
- ✓ Preserva la seguridad brindada por el ambiente de trabajo de Java en tiempo de ejecución.
- ✓ Su implementación 100% Java, es su principal inconveniente. RMI es muy complicado de emplear por otros lenguajes de programación, por lo que no es posible utilizarlo en aplicaciones distribuidas donde se utilizan distintos lenguajes.
- ✓ Los objetos a ser exportados, tienen que heredar obligatoriamente de un objeto base dado por la API. En general la implementación de RMI tiende a ser bastante invasiva.
- ✓ Depende de una máquina virtual Java para su ejecución.
- ✓ Disminuye el rendimiento con el crecimiento del sistema.

- ***Common Object Request Broker Architecture* (CORBA).**

CORBA es un estándar que establece una plataforma de desarrollo de sistemas distribuidos, facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. CORBA fue definido y está controlado por el Grupo de Gestión de Objetos (OMG) que define las API's, protocolos de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad

entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas. **[11]**

CORBA es una tecnología basada en la clásica arquitectura encuesta/respuesta. Existe una implementación de objetos en el servidor, al cual el cliente encuesta para ejecutar. Los servicios que un objeto provee, son dados por su interfaz, la cual es definida en el Lenguaje de Definición de Interfaces (IDL) del OMG. Los objetos distribuidos son identificados por referencias a objetos, las cuales son definidas por las interfaces IDL. El *Object Request Broker* (ORB) es el servicio distribuido que implementa la solicitud al objeto remoto, localiza el objeto remoto en la red, comunica la solicitud del objeto, espera los resultados y cuando están disponibles los envía de regreso al cliente.

Otra parte importante del estándar CORBA es la definición de un conjunto de servicios distribuidos que soportan la integración e interoperación de objetos distribuidos, tales como:

Ciclo de Vida: Define como los objetos de CORBA son creados, eliminados, movidos y copiados.

Nombre: Define como los objetos de CORBA pueden ser localizados por nombres simbólicos.

Eventos: Provee la comunicación entre objetos distribuidos.

Externalización: Coordina la transformación de objetos CORBA hacia y desde medios externos.

Transacciones: Coordina accesos atómicos a objetos CORBA.

Concurrencia y control: Provee un servicio de bloqueo para objetos de CORBA con el fin de asegurar acceso concurrente.

Registro: Soporta el encuentro de objetos CORBA basado en propiedades, describiendo el servicio ofrecido por el objeto.

Persistencia: Ofrece una interfaz para almacenar objetos.

Propiedades: Soporta la asociación de pares nombre-valor con objetos CORBA. **[12]**

Características de CORBA.

- ✓ Provee soporte a través de varias plataformas.
- ✓ Es escalable de sistemas grandes a enormes.
- ✓ Provee mapeo desde IDL hacia C, C++, ADA, *SmallTalk*, y Java.

- ✓ Transparencia en la localización.
- ✓ Transparencia en la red.
- ✓ Capacidades de tiempo real.
- ✓ Comunicación directamente con los objetos.
- ✓ Los mecanismos de seguridad son divergentes.
- ✓ Hay limitadas herramientas de desarrollo.
- ✓ Aun no hay un estándar real para prioridad de los hilos.
- ✓ Su uso no es tan transparente al programador como sería deseable.

- **DDS.**

El Servicio de Distribución de Datos (DDS) es el estándar de *middleware* de red para aplicaciones de tiempo real distribuidas adoptado por el OMG, está basado en un modelo de comunicaciones publicación/suscripción, proporcionando una forma simple e intuitiva de distribuir datos. DDS desacopla temporal y espacialmente a las entidades que crean y envían datos (los publicadores), de las entidades que los reciben y usan (los suscriptores). De este modo, los publicadores simplemente han de declarar su intención de publicar y posteriormente publicar los datos. Los suscriptores, por su parte, deben indicar su intención de recibir dichos datos, siendo éstos automáticamente entregados a los mismos independientemente de su localización espacial y temporal. **[15]**

DDS gestiona todas las fases de la transferencia: direccionamiento de los mensajes, la serialización y deserialización a formato estándar, entrega, control de ancho de banda y reintentos. Cualquier nodo puede ser publicador, suscriptor, o ambas cosas simultáneamente. Además gestiona de forma automática el cambio en caliente de publicadores redundantes si el primario falla. Los suscriptores siempre obtienen los datos que son aun válidos con la prioridad más alta.

Las especificaciones DDS describen dos niveles de interfaces:

- ✓ Un nivel bajo *Data-Centric Publish-Subscribe* (DCPS) que se encarga de entregar la información a los destinatarios de forma eficiente. DCPS es, además, una formalización (a través de una API estandarizada) del modelo publicación/suscripción.

- ✓ Un nivel superior, denominado *Data Local Reconstruction Layer* (DLRL). El objetivo de esta capa es actuar de interfaz entre la capa de aplicación y la capa DCPS. Se trata de una capa orientada a objetos, lo que permite que las aplicaciones de usuario puedan integrarla de forma simple. **[15]**

Entre los componentes importantes de la capa DCPS cabe destacar los siguientes:

DomainParticipant: Es el punto de entrada de las comunicaciones para un dominio concreto. Representa a la aplicación que emplea DDS dentro de un dominio específico y proporciona los componentes de DDS necesarios para la comunicación.

Topic: Instanciación de un componente *TopicDescription*, que es la descripción básica de un dato que puede ser publicado y al que se pueden suscribir otros componentes.

ContentFilteredTopic o MultiTopic: *TopicDescription* especializados.

Publisher: Objeto responsable de la diseminación de datos cuando estos deben ser publicados. **[16]**

DataWriter: Objeto que permite a una aplicación instanciar un valor de un dato para que sea publicado por un *Topic*.

Subscriber: Objeto responsable de la recepción de los datos resultante de las suscripciones.

DataReader: Objeto que permite a la aplicación declarar los datos en los que está interesada recibir información (creando una suscripción usando un *Topic*, *ContentFilteredTopic o MultiTopic*) y accede a los datos recibidos usando un *Subscriber* asociado.

De manera general el estándar DDS permite: establecer comunicaciones complejas uno a muchos y muchos a muchos haciendo uso de la API estándar para publicar y suscribirse a datos; personalizar aplicaciones con requisitos de tiempo real, fiabilidad y Calidad de Servicio (QoS); proporcionar tolerancia a fallos y a los posibles eventos relacionados con las comunicaciones entre aplicaciones de forma transparente; usar diferentes protocolos de transporte.

Características de DDS.

- ✓ Disminución entre el acoplamiento de entidades debido en parte al empleo de la filosofía publicación/suscripción.

- ✓ Arquitectura flexible y adaptable gracias al empleo del descubrimiento automático.
- ✓ Escalabilidad debido en parte a la disminución del acoplamiento entre entidades.
- ✓ Calidad de servicio altamente parametrizable.
- ✓ Proporciona tolerancia a fallos y a los posibles eventos relacionados con las comunicaciones entre aplicaciones de forma transparente.
- ✓ El débil acoplamiento hace que se dificulten las pruebas.

1.5.2. *Middlewares*

- **ORBit.**

El surgimiento de ORBit está relacionado con el momento en que el proyecto GNOME comenzó a hacer uso de CORBA con la utilización del ORB MICO pero este no se ajustaba muy bien a las necesidades de GNOME, por lo que Elliot Lee y Dick Porter decidieron escribir un nuevo ORB desde cero naciendo de esta forma ORBit.**[13]**

ORBit cumple con los dos requisitos del proyecto GNOME: es libre, y es uno de los ORB's más rápidos que existen, consigue una velocidad casi idéntica a una simple llamada a función cuando se usa en local, pues detecta automáticamente que las comunicaciones se están realizando en la misma máquina, desactivando en ese caso, gran parte de la complejidad del protocolo de comunicaciones usado en CORBA (GIOP/IIOP). Como el resto de partes básicas de la arquitectura de GNOME, ORBit está implementado en C.

Características de ORBit.

- ✓ Está liberado bajo la Licencia Pública General (GPL) y Licencia Pública General Menor (LGPL).
- ✓ Es rápido y ágil permitiendo el uso de CORBA en áreas en las que normalmente no parece práctico.
- ✓ Tiene soporte para lenguajes como C++, Perl y PHP entre otros, aunque esta implementado en C.
- ✓ ORBit es una implementación ligera por sus cortos tiempos de respuesta y que usa menos memoria que otros ORB.
- ✓ No es una implementación completa de CORBA ya que solo implementa algunos de sus servicios.

- ✓ Es incompatible con otros ORB.
- ✓ La frecuencia de transmisión de datos es baja.

- **ACE + TAO.**

Existen muchas implementaciones de ORB dependiendo del tipo de uso que se le desee dar. Para el caso de transmisión de datos en tiempo real existe *Real-Time* CORBA. Hay varios ORB implementados en distintos lenguajes de programación que cumplen dicha especificación. Entre ellos se puede encontrar *The ACE ORB* (TAO).

El Entorno de Comunicación Adaptativa (ACE) se compone de un gran conjunto de utilidades de programación en lenguaje C++ disponibles tanto en Linux como en Windows. Es un *framework* de libre distribución y código abierto. Simplifica el desarrollo de aplicaciones de comunicación en tiempo real. Provee un sistema de comunicación entre procesos, manejo de eventos, enlace dinámico de librerías y concurrencia. Otra gran cualidad de ACE es la automatización de configuraciones de sistema, y su reconfiguración mediante servicios de enlace dinámico en aplicaciones en tiempo de ejecución, y el procesado de dichos servicios en uno o más hilos. Estas facilidades eliminan carga de trabajo al programador. **[14]**

Entre los componentes básicos de TAO se encuentran:

IDL. Compiler. Genera los *stubs* y *skeletons*, necesarios para las llamadas remotas a métodos.

Inter-ORB Protocol Engine**.** Ofrece una capacidad de realizar operaciones entre ORB’s que usen el protocolo IIOP. Permite tanto modelos estáticos como dinámicos de programación en CORBA.

ORB Core. Permite comunicaciones unidireccionales, bidireccionales y comunicaciones fiables unidireccionales, tanto síncronas como asíncronas. También permite varios modelos de concurrencia.

***Portable Object Adapter* (POA).** Diseñado usando patrones que ofrecen un conjunto de estrategias útiles para la detección de referencias de objetos, tanto persistentes como transitivos.

Al ser TAO una implementación de *Real-Time* CORBA, ofrecen los servicios del estándar tratados anteriormente. Fuera de las especificaciones de CORBA se ofrecen los siguientes servicios:

Servicio de balance de carga: Este servicio aplica los algoritmos "*round robin*" y mínima dispersión para balancear las cargas en un grupo de máquinas.

Servicio de eventos en tiempo real: Este servicio aumenta el modelo del servicio de eventos del estándar de CORBA mediante el suministro de la fuente y filtrado basado en tipos, la correlación de eventos, el envío en tiempo real y la comunicación *multicast User Datagram Protocol/Internet Protocol* (UDP/IP).

Servicio de programación: Soporta la programación estática, así como dinámica de máxima urgencia, mediante un sistema de planificación basado en la asignación de prioridades.

Características de TAO.

- ✓ TAO está disponible libremente, de código abierto y compatible con estándares en tiempo real de implementaciones de CORBA con altas prestaciones.
- ✓ Proporciona eficiencia, previsibilidad, escalabilidad y calidad de servicios mediante el uso de patrones estándar.
- ✓ Portabilidad extensa para un gran número de compiladores y sistemas operativos.
- ✓ Es la implementación de CORBA que más servicios implementa por lo que lo hace un poco lento.
- ✓ Presenta algunas anomalías con el manejo de excepciones.

- ***OpenDDS***

OpenDDS es una implementación C++ multiplataforma de código abierto, de la especificación DDS del OMG, para sistemas cuyos requisitos incluyen: tiempo real, robustez, distribución de datos y tolerancia a fallos, utilizando un modelo de publicación/suscripción. En particular, se puede utilizar *OpenDDS* para construir software propietario y no se está bajo ninguna obligación de redistribuir cualquier parte del código fuente que se construye. **[17]**

OpenDDS se basa en la capa de abstracción de ACE para facilitar la portabilidad de la plataforma. Además aprovecha las capacidades de TAO, como su compilador de IDL y la base del Repositorio de Información del DCPS (DCPSInfoRepo). Adicionalmente aprovecha *Make Project Creator* (MPC) para aliviar la carga de mantenimiento apoyando la construcción de múltiples entornos y plataformas.

Características de *OpenDDS*.

- ✓ Ofrece los siguientes protocolos de transporte por defecto: *Transmission Control Protocol* (TCP), *ReliableMulticast, UnreliableMulticast* y UDP.
- ✓ El *framework* de transporte permite a cualquiera crear un transporte para adaptarse a los requerimientos personalizados.
- ✓ *OpenDDS* se ha encontrado para obtener mejores resultados que otros servicios de TAO tales como, *Name Service* (NS) y *Real Time Event Channel* (RTEC).
- ✓ Transferencia de datos alta y a gran velocidad.
- ✓ *OpenDDS* proporciona un *framework* de transporte que facilita añadir un nuevo transporte. **[17]**
- ✓ Genera código para proporcionar una eficiente serialización y deserialización de los tipos de datos definidos por el usuario.
- ✓ Exhibe una escalable arquitectura multi-hilo.
- ✓ Las características ofrecidas por el RTEC y el NS son similares a DDS pero no idénticas, por lo que se debe revisar cuidadosamente los casos de uso antes de elegir un servicio u otro.

1.6. Selección de tecnologías para el desarrollo del subsistema de comunicaciones.

En este punto, teniendo en cuenta: las dificultades que induce el uso de la tecnología ICE, abordadas en la introducción del trabajo y la descripción de las tendencias y tecnologías *middleware* más usadas en el desarrollo de subsistemas de comunicación para sistemas de tipo SCADA y otros sistemas distribuidos, se procederá a realizar un exhaustivo análisis con el fin de seleccionar y documentar la tecnología a usar para la implementación del subsistema de comunicaciones de SAINUX.

1.6.1. Criterios de evaluación.

Otro elemento importante a tener en cuenta antes de la selección de la tecnología más adecuada, es que la implementación de este nuevo subsistema debe mejorar las debilidades del anterior así como igualar o superar en lo posible, las fortalezas de la pasada versión. Con ese fin, el conjunto de características que debe soportar la tecnología seleccionada se define a continuación:

- ✓ **Desarrollo en C++:** C++ es un lenguaje de programación que se utiliza ampliamente en la industria del software. Otro elemento a tener en consideración es que desde la concepción del CEDIN el lenguaje de programación base para el desarrollo de sus productos ha sido el C++. Los recursos que tiene este lenguaje son muy útiles en la rama de la automatización, que es la base del producto a desarrollar**.[11]**
- ✓ **Implementación con herramientas de software libre y código abierto:** Con el propósito de explotar al máximo las herramientas existentes para el desarrollo de las aplicaciones, es muy importante la utilización de las herramientas y tecnologías bajo la Licencia Pública General Menor (LGPL), esta es una modificación de la licencia GPL. Reconoce que muchos desarrolladores de software, no solo utilizarán el código fuente que se distribuya bajo la licencia GPL, debido a su principal desventaja, que determina que todos los derivados tendrán que seguir los dictámenes de esa licencia. La LGPL permite que los desarrolladores utilicen programas bajo la GPL o LGPL sin estar obligados a someter el programa final bajo dichas licencias. Permite entonces, la utilización simultánea de software con este tipo de licencia, tanto en desarrollos libres, como en desarrollos privativos, por lo que permite realizar versiones comerciales de un producto final. Es por ello que se decide utilizar este tipo de herramientas y no otras que pueden resultar mucho más complicadas en adquisición y utilización, de esta forma estamos contribuyendo a alcanzar la independencia tecnológica.
- ✓ **Comunicación distribuida:** Facilita el proceso de comunicaciones en ambientes distribuidos, pues la responsabilidad de estas no depende del tipo de estructura y recae en el sistema que se utilice para gestionar las mismas. Estos sistemas deben permitir la comunicación entre aplicaciones, sin importar dónde estén localizadas. Es importante que la comunicación distribuida no sea solamente a través de peticiones a objetos remotos o interacciones cliente/servidor, sino que también sean a través de mecanismos de publicación/suscripción para las interacciones en tiempo real.
- ✓ **Capacidad de tiempo real:** Se entiende por Sistema en Tiempo Real (STR) a aquel que interactúa activamente en un entorno con dinámica

conocida en relación con sus entradas, salidas y restricciones temporales, para darle un correcto funcionamiento de acuerdo con los conceptos de estabilidad, control y alcance. En otras palabras un STR se define como un sistema informático que tiene la capacidad de interactuar rápidamente con su entorno físico. Es de mucha importancia medir la capacidad de transacciones en tiempo real en cada tecnología a fin de garantizar que la información viaje de forma rápida cumpliendo con los tiempos establecidos en las especificaciones de requisitos del sistema SAINUX.

- ✓ **Manejo de un elevado número de variables:** Las variables son estructuras de datos que como su nombre indica pueden cambiar de contenido durante la ejecución del programa. Los sistemas distribuidos generalmente son muy grandes por lo que el intercambio de variables entre un proceso y otro resulta imprescindible. Estas, requieren un seguimiento especial pues en determinados sistemas una pequeña variable que no se atienda como lo requiere puede ocasionar grandes daños. Es por eso la importancia que amerita lograr el manejo de un elevado número de variables.
- ✓ **Persistencia a las conexiones**: Las conexiones persistentes garantizan que, siempre que haya una interrupción, la conexión entre los elementos sea automáticamente establecida tan pronto como se recupere el servicio. Esto evita que se tengan que reiniciar los componentes para reconocerse nuevamente, debido a que la persistencia asegura que el enlace esté siempre disponible relazando automáticamente la conexión. Su ventaja radica en esta facilidad precisamente, ya que estas son más óptimas en ciertos aspectos, al no tener que conectar y desconectar cada vez que se hace una transacción.
- ✓ **Tolerancia a fallos:** Cuando se producen fallos en el software o en el hardware de algún sistema informático, los programas, podrían producir resultados incorrectos o podrían dejar de brindar un determinado servicio. La tolerancia a fallos es la propiedad de algunos ordenadores o sistemas de funcionar, aun cuando se haya producido una falla en alguno de sus componentes. Es algo propio de sistemas que precisan de una alta disponibilidad en función de la importancia de las tareas que

realizan. De ahí la necesidad de contar con mecanismos de tolerancia a fallas a fin de obtener soluciones capaces de funcionar aun cuando existan algunos contra tiempos o fallas.

- ✓ **Redundancia:** La redundancia en un sistema de comunicación, consiste en intensificar y repetir la información contenida, a fin de que no se provoque una pérdida fundamental de información ante un determinado fallo.
- ✓ **Seguridad:** En todo sistema SCADA resulta muy importante la seguridad, es por eso que se debe evaluar los mecanismos de seguridad que provee cada tecnología a fin de seleccionar los más adecuados. A la hora de prever la seguridad del sistema se tiene que tener en cuenta varios aspectos fundamentales. Uno de ellos es la seguridad lógica que es la que se encuentra a nivel de los datos. El otro aspecto a tener en cuenta es la seguridad en las telecomunicaciones incluyendo las tecnologías de red, servidores del sistema y las redes de acceso entre otras. Es necesario contar con mecanismos de seguridad eficientes que introduzcan un valor agregado a la solución de la capa de comunicación del SAINUX.

1.6.2. Análisis de opciones tecnológicas.

Teniendo en cuenta los criterios de selección previamente definidos, así como la descripción de cada una de las posibles implementaciones de *middleware* a utilizar (Orbit, ACE+TAO y *OpenDDS*), se llega al punto de definir cuál de estas cumple con las exigencias del subsistema a desarrollar. Para ello se realiza un análisis de cada una, valorando sus características más distintivas y además se realiza un modelo de decisión (ver anexo 3) con el objetivo de definir la tecnología de desarrollo para el subsistema de comunicación de SAINUX.

Comenzando con el análisis puede verse que **ORBit** y **ACE+TAO** poseen prestaciones similares y convenientes para su utilización en un sistema, debido a que ambas son implementaciones del estándar CORBA. La primera, se destaca por ser más ligera, por sus cortos tiempos de respuesta y baja utilización de memoria, sin embargo, TAO está concebido para el desarrollo de sistemas en tiempo real debido a la eficiente implementación que realiza del estándar *Real Time* CORBA, además soporta la mayoría de sus

características. No obstante, debe aclararse que estos dos ORB mantienen algunas de las desventajas de CORBA como es la incompatibilidad con otros ORB y no implementan todos los servicios de los que dispone el mismo.

Por último, otra de las tecnologías que ha alcanzado un cierto reconocimiento y madurez, en los últimos tiempos es ***OpenDDS***. Esta integra varios componentes de otras tecnologías para lograr su funcionamiento lo que la convierte en una herramienta muy potente. Además de que es una implementación en C++ del estándar DDS y de código abierto, implementa los perfiles del DCPS basándose en la capa de abstracción de ACE para facilitar la portabilidad. También integra algunas capacidades de TAO como su compilador IDL entre otros elementos. Todo lo anteriormente mencionado le permite obtener mayores resultados que TAO en el NS y el RTEC. Es preciso aclarar que las características ofrecidas por el RTEC y el NS son similares a DDS pero no idénticas, por lo que se debe revisar cuidadosamente los casos de uso antes de elegir un servicio u otro.

1.7. Tecnología propuesta para el desarrollo del subsistema de comunicaciones.

Como consecuencia del estudio y análisis de los resultados arrojados tras la evaluación de cada una de las tecnologías abordadas previamente en el modelo de decisión (ver figura 3), y teniendo en cuenta la opinión de varios especialistas en el tema de tecnologías de comunicación distribuida en sistemas SCADA, Se determina que la tecnología más apropiada para el desarrollo del subsistema de comunicación de SAINUX, es ***OpenDDS*** la cual, en teoría demostró ser mejor que las demás en cuanto a los objetivos que persigue el subsistema a desarrollar.

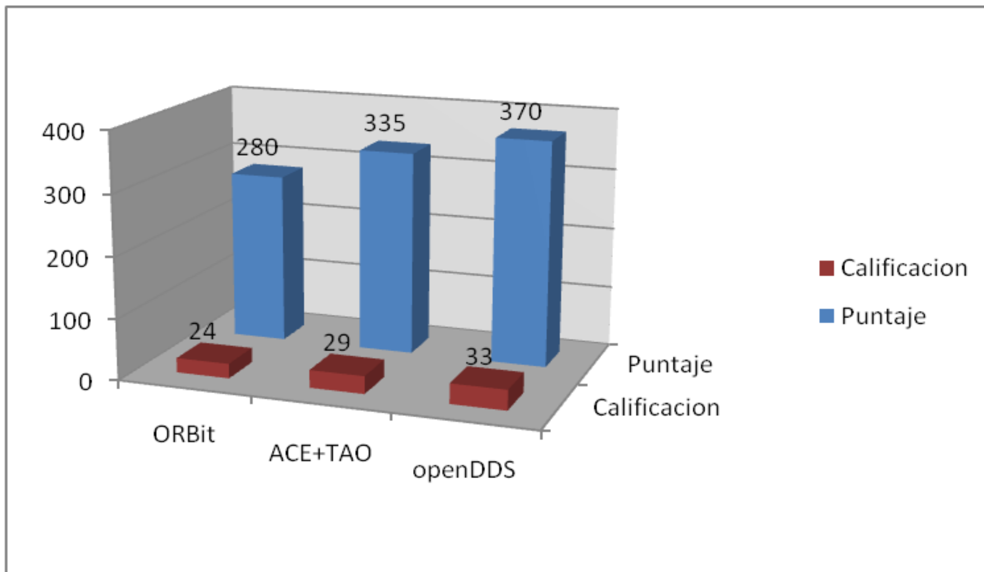

Figura 3: Matriz de decisión de tecnología para la implementación del subsistema de comunicaciones de SAINUX

1.8. Metodología, lenguajes y herramientas para el desarrollo del subsistema de comunicaciones.

En el proceso de desarrollo de software intervienen varios factores que determinan la calidad del producto final, ellos son: la metodología, los lenguajes y las herramientas a utilizar durante el desarrollo del módulo de comunicación, los cuales, en el presente trabajo, no son objeto de selección, pues ya han sido analizados y valorados por analistas, arquitecto y jefe del proyecto de SAINUX.

Como metodología de desarrollo se empleará **RUP** (*Rational Unified Process*). RUP divide el proceso de desarrollo en ciclos, teniendo un producto final al culminarse cada una de las iteraciones. Es más apropiada para proyectos de gran envergadura, dado que requiere un equipo de trabajo capaz de administrar un proceso complejo en varias etapas, por lo que la hace ideal para el sistema SAINUX. El lenguaje de modelado a utilizar será **UML** (*Unified Modeling Language*), es desde finales de 1997 un lenguaje de modelado visual que se utiliza para especificar, visualizar, construir y documentar artefactos de un sistema de software **[18]**. Como herramienta CASE, ***Visual Paradigm*** la cual propicia un conjunto de ayudas para el desarrollo de programas informáticos, desde la planificación, pasando por el análisis y el diseño, hasta la generación del código fuente de los programas y la documentación, ha sido concebida para soportar el ciclo de vida completo del proceso de desarrollo del software a través de la representación de todo tipo de diagramas **[19]**. El

lenguaje de programación y entorno de desarrollo del cual se hará uso son **C++** y **eclipse** respectivamente.

Capítulo II: Modelado de la solución.

2.1. Introducción.

El presente capítulo describe los requisitos funcionales y no funcionales que debe cumplir el subsistema de comunicación, así como los casos de uso del sistema. Además muestra la arquitectura del estándar de *middleware* a utilizar que es DDS. Se explica también el modelo de mensajes Publicación/Suscripción, que se utilizará para lograr el intercambio de información entre los diferentes módulos de dicho sistema; y se define el modelo de diseño para tener una adecuada entrada a las actividades de implementación.

2.2. Requisitos del subsistema de comunicaciones.

Teniendo en cuenta la necesidad de crear un subsistema de comunicación que permita el intercambio de información entre todos los módulos del SAINUX, a continuación se enuncian los requisitos funcionales (RF) y no funcionales (RNF) determinados para desarrollar dicho subsistema.

2.2.1. Requisitos funcionales.

Los requisitos funcionales son capacidades o condiciones que el sistema debe cumplir. Estos no alteran la funcionalidad del producto, lo que quiere decir que se mantienen invariables sin importarle con que propiedades o cualidades se relacionen. A continuación se exponen los requisitos funcionales identificados para desarrollar el subsistema de comunicación.

RF 1: El sistema debe permitir el envió y recepción de puntos.

- ✓ RF 1.1: El sistema debe permitir la transmisión y recepción de datos complejos del tipo punto de forma desacoplada.
- ✓ RF 1.2: El sistema debe posibilitar el envió y recepción de datos complejos del tipo secuencia de puntos de manera desacoplada.

RF 2: El sistema debe permitir el envió y recepción de alarmas.

- ✓ RF 2.1: El sistema debe posibilitar la transmisión y recepción de datos complejos de tipo alarmas de forma desacoplada.
- ✓ RF 2.2: El sistema debe posibilitar el envió y recepción de datos complejos del tipo secuencia de alarmas de manera desacoplada.

RF 3: El sistema debe permitir el envió y recepción de comandos y respuestas de comandos.

- ✓ RF 3.1: El sistema debe posibilitar la transmisión y recepción de comandos de forma desacoplada.
- ✓ RF 3.2: El sistema debe ofrecer el envió y recepción de respuestas de comandos de manera desacoplada.

RF 4: El sistema debe permitir el envió y recepción de eventos.

- ✓ RF 4.1: El sistema debe permitir la transmisión y recepción de eventos de manera desacoplada.

RF 5: El sistema debe posibilitar el envió y recepción de bitácoras.

- ✓ RF 5.1: El sistema debe proveer la transmisión y recepción de bitácoras de usuarios de forma desacoplada.

RF 6: El sistema debe permitir el envió y recepción de Estados de Comunicación.

- ✓ RF 6.1: El sistema debe posibilitar la transmisión y recepción de datos complejos de tipo Estado de Comunicación de manera desacoplada.
- ✓ RF 6.2: El sistema debe posibilitar el envió y recepción de datos complejos del tipo secuencia de Estado de Comunicación de manera desacoplada.

RF 7: El sistema debe ser capaz de Gestionar canales de comunicación.

- ✓ RF 7.1: Agregar canales de comunicación.
- ✓ RF 7.2: Eliminar canales de comunicación.

2.2.2. Requisitos no funcionales.

Los requisitos no funcionales son propiedades o cualidades que el producto debe tener. Debe pensarse en estas propiedades como las características que hacen al producto atractivo, usable, rápido o confiable. Son importantes para que clientes y usuarios puedan valorar las características no funcionales del producto. A continuación se exponen los mismos:

Usabilidad

- ✓ RNF1.1 El sistema debe permitir el acceso a los servicios de manera rápida y segura.

Fiabilidad

- ✓ RNF2.1 Las solicitudes de variables, alarmas, eventos e invocación de los distintos tipos de comandos, como las acciones sobre estos, van a ser siempre dependientes de los mecanismos de suscripción y de la lógica de seguridad del sistema.
- ✓ RNF2.2 Debe existir una comunicación eficiente y fiable, garantizando que no existan pérdidas de información de (puntos, eventos, alarmas, comandos, estado de la comunicación y bitácoras).
- ✓ RNF2.3 Debe proveerse, a los servicios, de mecanismos de tolerancia ante fallos que permitan recuperarse antes errores.
- ✓ RNF2.4 Deben persistir las conexiones entre los módulos del SAINUX.

Eficiencia

- ✓ RNF3.1 Durante el intercambio de variables, debe existir una comunicación eficiente, garantizando una velocidad en la comunicación adecuada para aplicaciones de visualización y cálculos de algoritmos.
- ✓ RNF3.2 El sistema debe manejar 50000 variables, entre puntos y alarmas en un instante de tiempo dado.
- ✓ RNF3.3 Realizar implementaciones que permitan transacciones en tiempo real para procesos críticos.

Soporte

- ✓ RNF4.1 El desarrollo del sistema orientado a la exposición de interfaces y servicios debe brindar alta interoperabilidad.
- ✓ RNF4.2 Se debe implementar un sistema que introduzca atributos de flexibilidad, escalabilidad y adaptación y que permita agregar y/o modificar funcionalidades con bastante facilidad y de forma eficiente sin impactar en los clientes y sin incurrir en grandes cambios.
- ✓ RNF4.3 El sistema debe ser implementado con tecnologías libres que permitan el desarrollo de aplicaciones orientada a servicios y que sean compatibles con todos los sistemas operativos existentes, buscando una solución robusta y universal.

2.3. Diagrama de casos de uso del sistema.

Teniendo en cuenta los requisitos funcionales planteados anteriormente y el uso de la tecnología seleccionada para implementar el subsistema de comunicación, el diagrama de casos de uso del sistema queda de la siguiente manera:

Actor	Descripción
Publicador	Es el encargado de diseminar los datos en el dominio sobre los tópicos asociados.
Suscriptor	Es el encargado de recibir los datos según los tópicos a los que se ha suscrito.

Tabla 1: Actores del sistema.

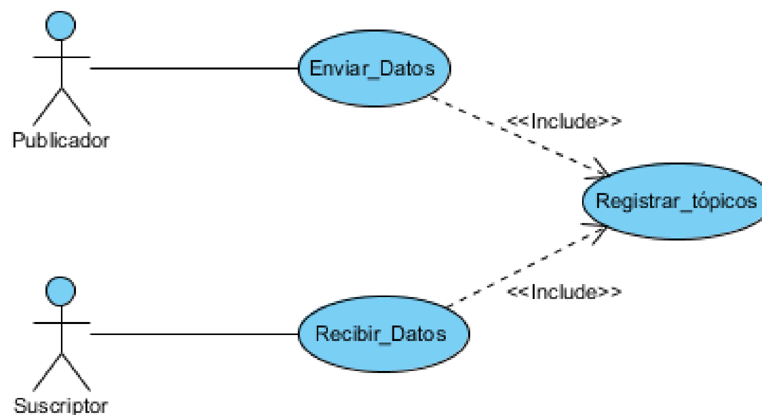

Figura 4: Diagrama de casos de uso del sistema

2.3.1. Descripción del caso de uso Enviar_Datos.

Caso de Uso:	Enviar_Datos
Actores:	Publicador
Resumen:	El caso de uso describe el proceso de publicación en el subsistema de comunicación de SAINUX, comienza cuando un publicador desea enviar datos al dominio.
Precondiciones:	Debe tener acceso a la aplicación y estar registrado en la misma.
Referencias	RF1, RF2, RF3, RF4, RF5, RF6, RF7
Prioridad	Crítico
Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
El publicador inicializa el proceso indicando que desea enviar datos al dominio.	El sistema crea al publicador como un participante de dominio. Crea un objeto *Publisher* para establecer la comunicación.
El publicador registra un tema de publicación con un nombre determinado.	El sistema crea un objeto *Topic* con el nombre que le pasa el publicador y lo

	asocia a un tipo de datos. Crea un objeto *DataWriter* asociado al *Topic* para escribir los datos del publicador.
El publicador envía un dato que puede ser: alarma, evento, punto, comando, respuesta de comando, estado de la comunicación o bitácora.	El sistema se encarga de escribir los datos para que puedan ser publicados en el dominio.
Flujos Alternos	
Acción del Actor	**Respuesta del Sistema**
El publicador envía un dato que puede ser: alarma, evento, punto, comando, respuesta de comando, estado de la comunicación o bitácora, con un nombre determinado.	El sistema se encarga de escribir los datos según el nombre que le pasa el publicador para que puedan ser publicados en el dominio.
Pos-condiciones	Enviar datos a todo el dominio.

Tabla 2: Descripción del caso de uso Enviar_Datos.

2.3.2. Descripción del caso de uso Recibir_Datos.

Caso de Uso:	Recibir_Datos
Actores:	Suscriptor
Resumen:	El caso de uso describe el proceso de suscripción en el subsistema de comunicación de SAINUX, comienza cuando un suscriptor desea recibir datos de su interés.
Precondiciones:	Debe tener acceso a la aplicación y estar registrado en la misma.
Referencias	RF1, RF2, RF3, RF4, RF5, RF6, RF7
Prioridad	Crítico

Flujo Normal de Eventos	
Acción del Actor	**Respuesta del Sistema**
El suscriptor inicializa el proceso indicando que desea recibir datos publicados en el dominio.	El sistema crea al suscriptor como un participante de dominio. Crea un objeto *Subscriber* para establecer la comunicación.
El suscriptor registra un tema con un nombre determinado, sobre el cual desea recibir información.	El sistema crea un objeto *Topic* con el nombre que le pasa el suscriptor y lo asocia a un tipo de datos. Crea un objeto *DataReader* asociado al *Topic* para leer los datos que llegan al suscriptor.
El suscriptor espera por los publicadores para recibir los datos.	
Pos-condiciones	Recibir los datos de interés publicados en el dominio.

Tabla 3: Descripción del caso de uso Recibir_Datos.

2.4. Arquitectura del estándar DDS.

El *middleware OpenDDS* está basado en el estándar DDS e implementa su arquitectura, la cual está separada en dos capas. La capa inferior es la centrada en los datos de publicación y suscripción (DCPS), que contiene interfaces de tipo seguro para el mecanismo de comunicación publicación/suscripción. La capa superior es la capa local de reconstrucción de datos (DLRL), que permite a un desarrollador de aplicaciones la construcción de un modelo de objetos local en la parte superior de la capa de DCPS. Cada capa tiene su propio conjunto de conceptos y patrones de uso, y por lo tanto los conceptos y la terminología de las dos capas se pueden discutir por separado. **[20]**

2.4.1. Capa DCPS

La capa de DCPS es responsable de manera eficiente de la difusión de datos desde los publicadores a los suscriptores interesados. Está implementada utilizando los conceptos de *publisher* y *data writer* en el lado del emisor y *subscriber* y *data reader* en el lado del receptor. La capa DCPS consta de uno o más dominios de datos, cada uno de los cuales contiene un conjunto de participantes (editores y suscriptores) que se comunican a través de DDS. Cada entidad (es decir, el editor o suscriptor) pertenece a un dominio. Cada proceso tiene un participante de dominio para cada dominio de datos del cual es miembro. **[20]**

Dentro de un dominio de datos, los datos se identifican con un *topic* (tema) que es un segmento de dominio de tipo específico que permite a los publicadores y suscriptores referirse a los datos de forma inequívoca. Dentro de un dominio, un *topic* asocia un nombre de tema único, el tipo de datos, y un conjunto de políticas de calidad de servicio (QoS) con los datos en sí. Cada *topic* está asociado con un solo tipo de datos, aunque muchos *topics* diferentes pueden publicar el mismo tipo de datos. **[20]**

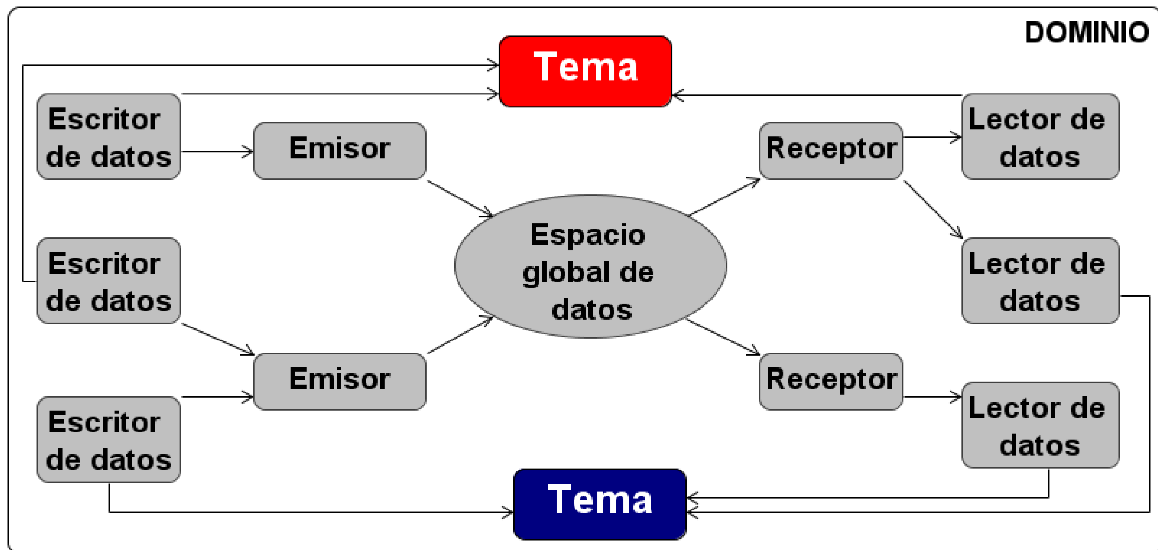

Figura 5: Funcionamiento de la capa DCPS

2.4.2. Capa DLRL

La capa DLRL es una capa orientada a objetos en la parte superior de la DCPS. Un objeto DLRL es un idioma nativo (es decir, C + +) con uno o más atributos compartidos. Cada clase DLRL se asigna a uno o más *topics* de DCPS, cada valor de atributo compartido se asigna a un campo en el tipo de datos de un *topic*, y su valor se distribuye a través de la aplicación de DCPS. Un participante DLRL comunica los datos al resto de la aplicación mediante la modificación de un objeto DLRL, resultando la publicación de una muestra de datos sobre el tema asociado. Un atributo DLRL compartido puede ser un valor simple o una estructura, una referencia a otro objeto DLRL, o una colección (lista, mapa) de ellos. DLRL soporta gráficos de objetos complejos y relaciones complejas entre los objetos DLRL. **[20]**

El desarrollador es responsable de decidir cómo las entidades DCPS se asignan a los objetos DLRL. El modelo se especifica en OMG *Interface Definition Language* (IDL), utilizando los tipos de valor IDL. La especificación DDS cuenta con una asignación predeterminada de la DCPS a la DLRL. **[20]**

2.5. Modelo de mensajes Publicación/Suscripción.

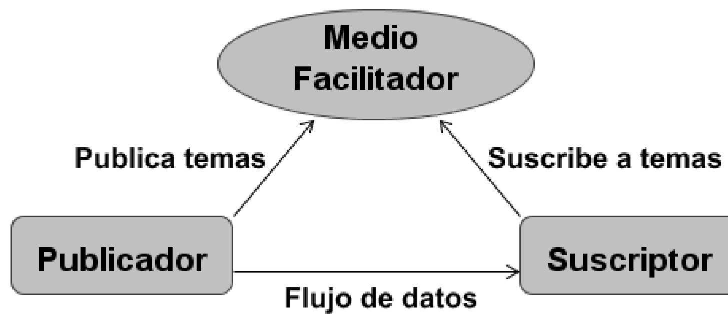

Figura 6: Modelo de Publicación/Suscripción

Este modelo se fundamenta en el intercambio asíncrono de mensajes. Aquí las entidades generadoras declaran una serie de *topics* que están dispuestas a publicar y las consumidoras se suscriben a distintos *topics* de su interés. De este modo, cuando un productor publica un dato con una temática concreta, todos los suscriptores de esta lo reciben de forma transparente a la aplicación. El paradigma adopta una aproximación denominada *data centric*, ya que desacopla (en el tiempo y el espacio) la interacción entre los participantes (consumidores o generadores de información), y enfoca su esfuerzo en la distribución de los datos, con independencia de la localización y el instante de origen o destino de los mismos. Para la distribución de información de acuerdo con el paradigma publicación/suscripción, el OMG ha especificado recientemente el estándar DDS lo que, dada la solvencia y reconocimiento de esta institución, ha venido a consolidar definitivamente este nuevo paradigma como modelo de interacción alternativo. **[21]**

2.6. Modelo de diseño del subsistema de comunicación de SAINUX.

El siguiente esquema muestra los distintos paquetes que componen el modelo de Publicación/Suscripción en el subsistema de comunicaciones de SAINUX, cuyos propósitos se explican a continuación.

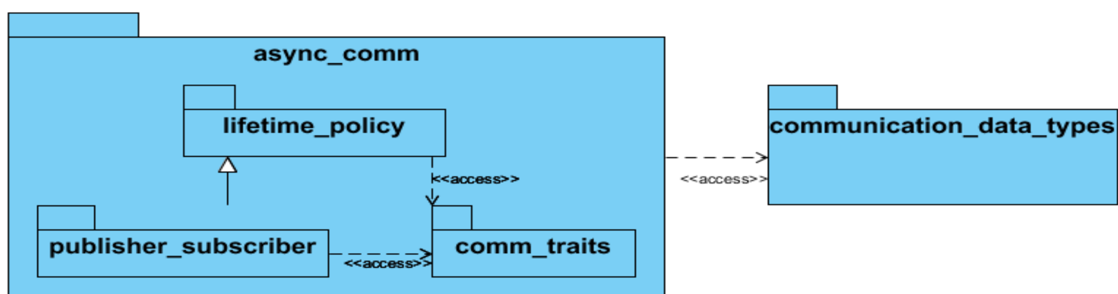

Figura 7: Diagrama de paquetes del subsistema de comunicación de SAINUX

2.6.1. *publisher_subscriber:*

Este paquete engloba el proceso de publicación/suscripción de manera desacoplada a los clientes en tiempo y espacio. A continuación se muestra el diagrama de clases correspondiente a este paquete:

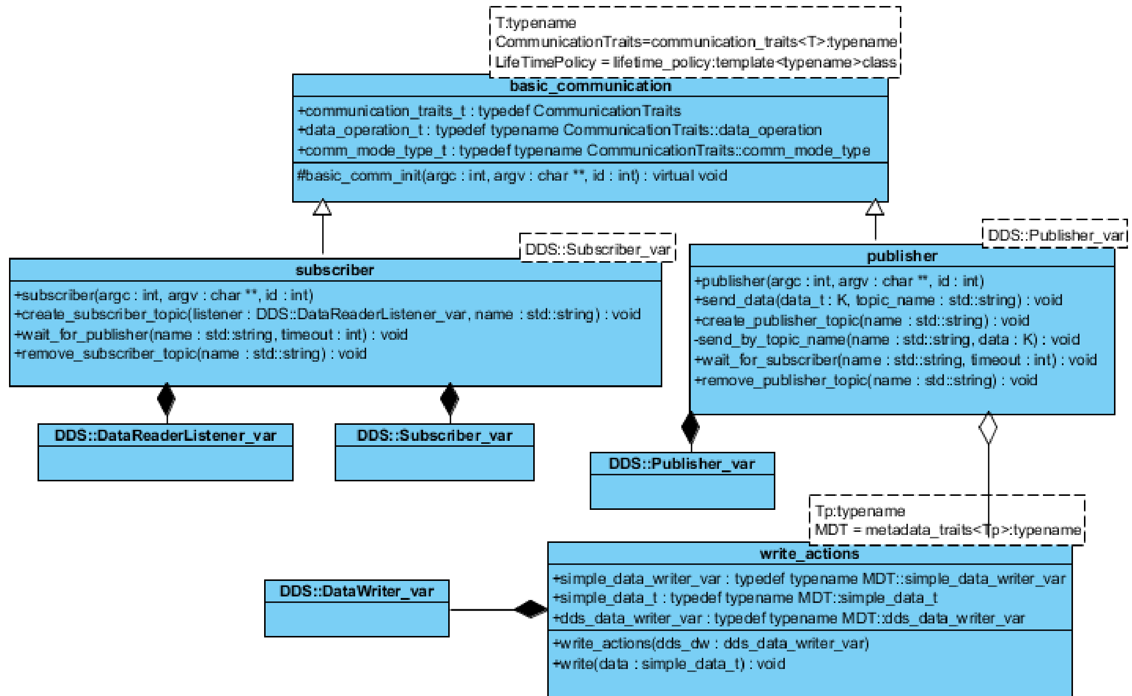

Figura 8: Diagrama de clases del paquete: publisher_subscriber

Descripción de la clase: *basic_comunication*	
Elemento	**Descripción**
basic_comunication	Esta clase engloba las características que son comunes a los publicadores y suscriptores, y actúa específicamente como uno de ellos, en dependencia del parámetro que se le pase (*publisher* o *subscriber*).
	Métodos
*#basic_comm_init(argc: int, argv: char * *, id: int): virtual void*	Inicializa la clase base y crea un objeto *publisher_var* o un *subscriber_var.*

Tabla 4: Descripción de la clase: basic_communication.

Descripción de la clase: *publisher*	
Elemento	**Descripción**
publisher	Es la clase encargada de diseminar los datos en el dominio.

	Métodos
+send_data(data_t : K, topic_name : std::string) : void	Envía un dato de tipo K (puntos, alarmas, eventos, comandos, estado de la comunicación y bitácora) según el nombre del *topic*.
Template<typename K> +create_publisher_topic(name : std::string) : void	Esta función crea un *topic* de nombre *name* asociado al tipo de dato que se especifica como parámetro K.
Template<typename K> -send_by_topic_name(name : std::string, data : K) : void	Esta función envía un dato de tipo K según el nombre del *topic*.
+wait_for_subscriber(name : std::string, timeout : int) : void	Espera un tiempo determinado por un suscriptor.
+remove_publisher_topic(name : std::string) : void	Elimina un *topic* según el nombre.

Tabla 5: Descripción de la clase publisher.

Descripción de la clase: *write_actions*	
Elemento	**Descripción**
Write_actions	Esta clase se encarga de publicar un tipo de dato específico.
	Métodos
+write(data : simple_data_t) : void	Publica un dato de tipo *simple_data_t* que puede ser: puntos, alarmas, eventos, comandos, estado de la comunicación y bitácoras.

Tabla 6: Descripción de la clase write_actions.

Descripción de la clase: *subscriber*	
Elemento	**Descripción**
subscriber	Esta clase se encarga de gestionar las suscripciones relativas a los *topic* sobre los que se ha declarado interés de recibir los datos.
	Métodos
Template<typename K> +create_subscriber_topic(listener :DDS::DataReaderListener_var, name : std::string) : void	Esta función crea un *topic* de nombre *name* asociado al tipo de dato que se especifica como parámetro K. Además crea un objeto *DDS::DataReader_var* pasándole el *listener* especificado, que es el que se encarga de

	leer los datos.
+wait_for_publisher(name : std::string, timeout : int) : void	Espera un tiempo determinado por un publicador.
+remove_subscriber_topic(name : std::string) : void	Elimina un topic según el nombre.

Tabla 7: Descripción de la clase subscriber.

2.6.2. *lifetime_policy:*

Este paquete tiene la función de inicializar y destruir los participantes de dominio, así como los *topic* creados durante la ejecución.

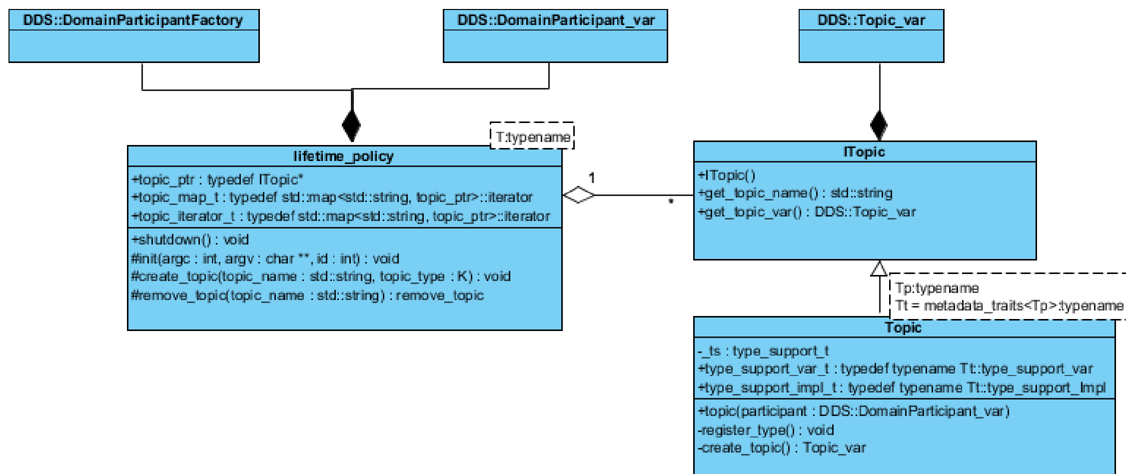

Figura 9: Diagrama de clases del paquete lifetime_policy

Descripción de la clase: *lifetime_policy*	
Elemento	**Descripción**
lifetime_policy	Esta clase inicializa y destruye los participantes de dominio y los *topics*.
	Métodos
*#init(argc : int, argv : char **, id : int) : void*	Inicializa la *factory* y el objeto de participante de dominio: *DDS::DomainParticipant_var*.
+shutdown() : void	Libera toda la memoria de los objetos creados.
Template<typename K> #create_topic(topic_name : std::string, topic_type : K) : void	Esta función crea un *topic* de nombre *topic_name* asociado al tipo de dato que se especifica como parámetro K.
#remove_topic(topic_name : std::string) : remove_topic	Elimina un *topic* por su nombre.

Tabla 8: Descripción de la clase lifetime_policy.

Descripción de la clase: *ITopic*	
Elemento	**Descripción**
ITopic	Esta clase es una interfaz que posee las funcionalidades de los *topic*.
	Métodos
+get_topic_name() : std::string	Muestra el nombre de un *topic.*
+get_topic_var() : DDS::Topic_var	Devuelve la entidad *Topic_var.*

Tabla 9: Descripción de la clase ITopic.

Descripción de la clase: *Topic*	
Elemento	**Descripción**
Topic	Esta es la clase genérica encargada de la creación de *topics*, asociándolos a tipos de datos.
	Métodos
-register_type(): void	Registra el tipo de dato asociado al *topic.*
-create_topic(): Topic_var	Crea un *topic.*

Tabla 10: Descripción de la clase Topic.

2.6.3. *comm_traits:*

Este paquete contiene la especificación del patrón de diseño: *Traits*, esta una técnica que sale naturalmente del uso de los plantillas cuando tienes que manejar varias tipos/clases distintas, consiste en definir clases plantillas intermediarias que contienen, aparte, las características de las clases que se manipulan en el patrón.

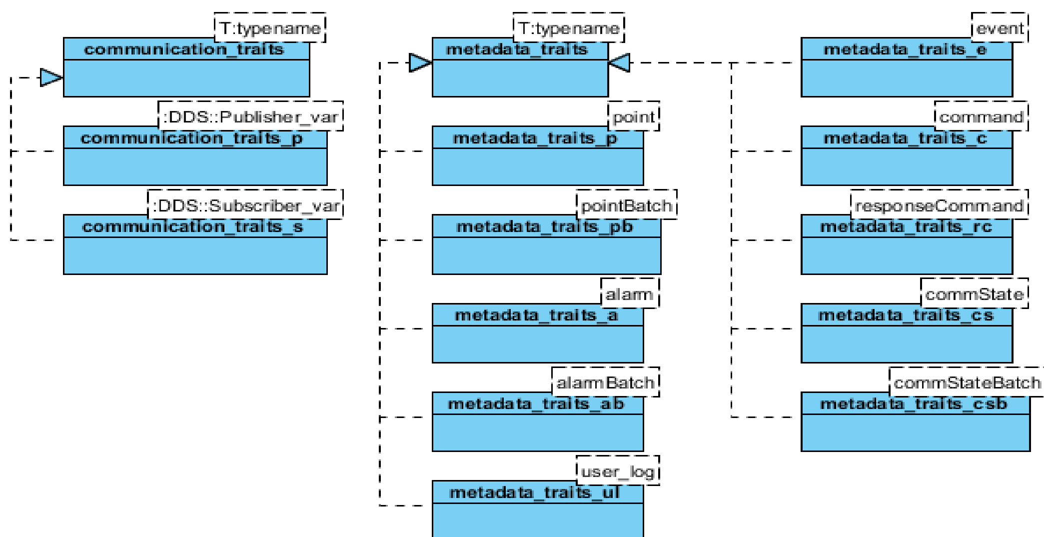

Figura 10: Diagrama de clases del paquete comm_traits

Descripción de la clase: *communication_traits*	
Elemento	**Descripción**
communication_traits	Es una clase genérica que sirve de base para las especializaciones de los *communication_traits.*

Tabla 11: Descripción de la clase communication_traits.

Descripción de la clase: *communication_traits_p<publisher_var>*	
Elemento	**Descripción**
communication_traits_p	Especialización de *communication_traits* para *publisher*.
	Métodos
+create_communication_mode(participant : DDS::DomainParticipant_var) : static comm_mode_type	Crea un objeto *DDS::publisher_var*.
+create_data_operation(topic : DDS::Topic_var, c_mode_type : comm_mode_type, listener : Listener_var = 0) : static data_operation	Crea un objeto *DDS::DataWriter_var*
+wait(data_op : data_operation &, s_timeout : int) : static void	Espera por un *subscriber* disponible.

Tabla 12: Descripción de la clase communication_traits_p <publisher_var>.

Descripción de la clase: *communication_traits_s<subscriber_var>*	
Elemento	**Descripción**
communication_traits_s	Especialización de *communication_traits* para *subscriber*.
	Métodos
+create_communication_mode(participant : DDS::DomainParticipant_var) : static comm_mode_type	Crea un objeto *DDS::subscriber_var*.
+create_data_operation(topic : DDS::Topic_var, c_mode_type : comm_mode_type, listener : Listener_var = 0) : static data_operation	Crea un objeto *DDS::DataReader_var*
+wait(data_op : data_operation &, s_timeout : int) : static void	Espera por un *publisher* para recibir datos.

Tabla 13: Descripción de la clase communication_traits_s <subscriber_var>.

Descripción de la clase: ***metadata_traits***	
Elemento	**Descripción**
metadata_traits	Es una clase genérica que sirve de base para las especializaciones de los *metadata_traits.*

Tabla 14: Descripción de la clase metadata_traits.

Descripción de la clase: ***metadata_traits_p <point>***	
Elemento	**Descripción**
metadata_traits_p	Especialización de *metadata_traits* para el tipo de dato punto. Esta clase se repite por cada tipo de dato que se maneja en el sistema (puntos, secuencia de puntos, alarmas, secuencia de alarmas, eventos, comandos, respuesta de comandos, estado de la comunicación, secuencia de estado de la comunicación y bitácoras).
	Métodos
+get_name() : static std::string	Muestra el nombre del *topic* asociado.
+narrow_reader(reader : dds_data_reader_ptr) : static simple_data_reader_var	Castea un objeto *DDS::DataReader_ptr* como un *pointDataReader_var*
+narrow_writer(writer : dds_data_writer_var) : static simple_data_writer_var	Castea un objeto *DDS::DataWriter_var* como un *pointDataWriter_var*

Tabla 15: Descripción de la clase metadata_traits_p <point>.

2.6.4. *communication_data_types:*

En este paquete se definen los tipos de datos que se manejan en el SCADA SAINUX, como son alarmas, eventos, puntos, comandos, estado de la comunicación y bitácoras. La tecnología DDS posee un lenguaje de definición de interfaces "*IDL*" donde los ficheros creados tendrán la extensión **.idl**, y al usar el compilador *IDL* se generan los *stub* que definen cada tipo de dato como tal, que son los que cada módulo va a utilizar para el manejo de alarmas, eventos, puntos, comandos, estado de la comunicación y bitácoras.

2.7. Patrones de diseño utilizados.

El uso de patrones contribuye a reutilizar diseño, identificando aspectos claves de la estructura de un diseño que puede ser aplicado en una gran cantidad de situaciones. La reutilización del diseño provee numerosas ventajas: reduce los esfuerzos de desarrollo y mantenimiento, mejora la seguridad, eficiencia y

consistencia de los diseños. Además aumenta la flexibilidad, modularidad y extensibilidad, factores internos e íntimamente relacionados con la calidad percibida por el usuario.

2.7.1. *Traits (rasgos o características)*

Los *traits* son una técnica de programación genérica que permite en tiempo de compilación que se tomen decisiones sobre la base de tipos, así como basado en valores durante el tiempo de ejecución. **[22]**

En este caso el patrón se utiliza para definir aparte las características de los tipos de datos a manejar (*metadata_traits*), que son puntos, alarmas, eventos, comandos, estado de la comunicación y bitácoras, así como para el modo de comunicación (*communication_traits*) que se establecerá, ya sea de publicador o de suscriptor.

Ventajas:

- ✓ El uso de los *traits* permite tener un código más limpio, legible y fácil de mantener.
- ✓ Si se utilizan los *traits* correctamente, se pueden obtener las ventajas anteriores sin tener que pagar el costo en el rendimiento, seguridad o acoplamiento que darían otras soluciones.

2.7.2. *Policy (políticas)*

Las *policy* tienen mucho en común con los *traits* pero se diferencian en que ponen menos énfasis en el tipo y más énfasis en el comportamiento. Estas ayudan a la implementación de elementos de diseño reutilizable, seguro, eficiente y altamente personalizable. Una *policy* se define a través de una clase o una clase plantilla en la cual se definen una familia de algoritmos. **[23]**

En este caso se utiliza para definir el comportamiento de creación y destrucción de algunas entidades, tales como: *Topic, DomainParticipant y DomainParticipanFactory* en la clase *lifetime_policy*, la cual actúa en dependencia del parámetro plantilla que recibe, que puede ser: *publisher_var* o *subscriber_var*.

Ventajas:

- ✓ Se puede variar el algoritmo dinámicamente sin afectar el contexto.
- ✓ Elimina las sentencias condicionales.

- ✓ Dispone implementaciones diferentes de un mismo comportamiento. El cliente puede elegir según la necesidad de espacio y tiempo.

2.7.3. *Facade* (fachada)

Proporciona una o varias interfaces unificadas de alto nivel que representa a todo un subsistema y facilita su uso. La “fachada” satisface a la mayoría de los clientes, sin ocultar las funciones de menor nivel a las que necesiten acceder. **[24]**

La fachada del subsistema de comunicación está vista por las clases *publisher* y *subscriber* que separan al cliente de todos los componentes del subsistema.

Ventajas

- ✓ Al separar al cliente de los componentes del subsistema, se reduce el número de objetos con los que el cliente trata, facilitando así el uso del subsistema.
- ✓ Se promueve un acoplamiento débil entre el subsistema y sus clientes, eliminándose o reduciéndose las dependencias.
- ✓ No existen obstáculos para que las aplicaciones usen las clases del subsistema que necesiten. De esta forma se puede elegir entre facilidad de uso y generalidad.

2.8. Diagrama de secuencia: *publisher*.

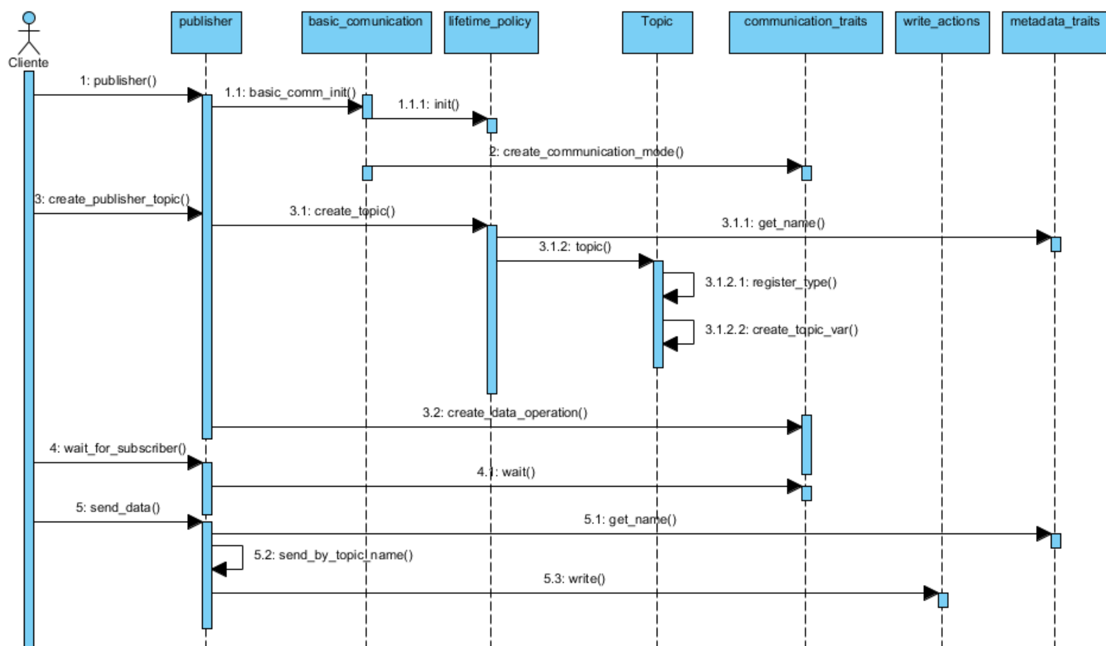

Figura 11: Diagrama de secuencia de publisher

El diagrama de secuencia anterior describe el proceso de publicación en el subsistema de comunicación de SAINUX. El proceso se inicializa creando un objeto *publisher* y con este las entidades DDS necesarias. Después se crea el *topic* a publicar, que puede ser (alarma, evento, punto, comando, estado de la comunicación o bitácora). Para ello se crea además el tipo de operación a realizar, que en este caso de la publicación es un *data_writer*; y por último el publicador comienza a enviar los datos.

2.9. Diagrama de secuencia: *subscriber*.

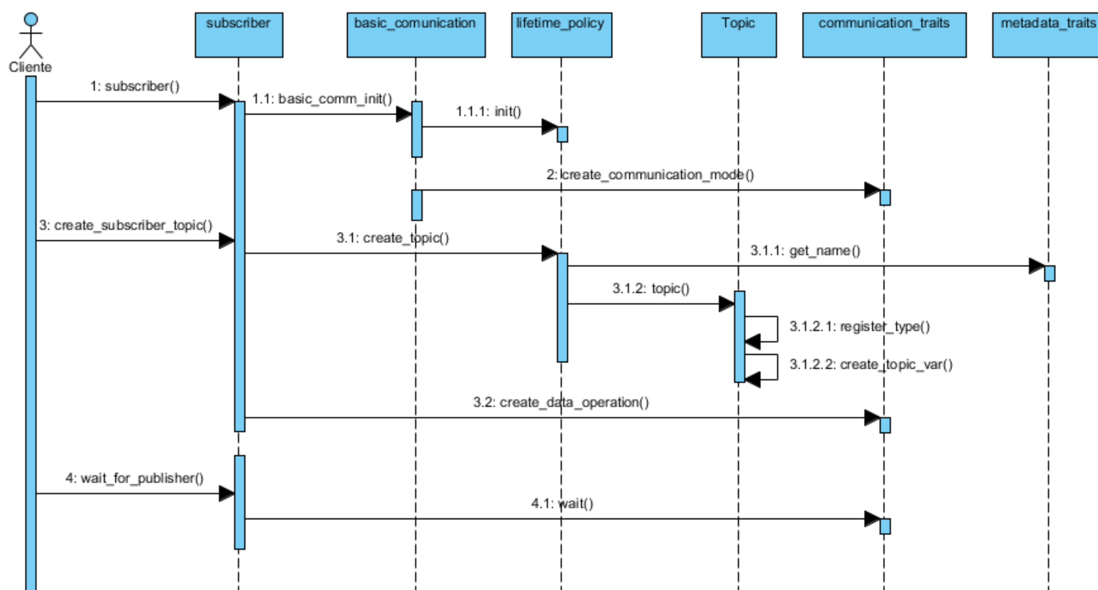

Figura 12: Diagrama de secuencia de subscriber

El diagrama de secuencia anterior describe el proceso de suscripción en el subsistema de comunicación de SAINUX. El proceso se inicializa creando un objeto *subscriber* y con este las entidades DDS necesarias. Después se crea el *topic* a suscribirse, que puede ser (alarma, evento, punto, comando, estado de la comunicación o bitácora). Para ello se crea además el tipo de operación a realizar, que en este caso de la suscripción es un *data_reader*; y luego el suscriptor comienza a esperar por los publicadores para recibir sus datos.

Capítulo III: "Implementación y pruebas".

3.1. Introducción.

En el presente capítulo se muestran las vistas de implementación y despliegue con sus respectivos diagramas de componentes. Además se expone el estilo de código utilizado, las vistas más significativas del código, y las pruebas que validan el funcionamiento del subsistema de comunicación.

3.2. Modelo de Implementación.

La implementación comienza con el resultado del diseño y se implementa el sistema en términos de componentes, es decir, ficheros de código fuente. Además muestra cómo los componentes se organizan de acuerdo a los nodos específicos en el modelo de despliegue.

Como resultado de la implementación del subsistema de comunicaciones de SAINUX, se obtiene una biblioteca llamada *async_comm*, que implementa el envío y recepción de datos de manera asíncrona, siguiendo el modelo de publicación/suscripción de *OpenDDS*, el cual mantiene desacoplados a los clientes, en tiempo y espacio de los mecanismos de comunicación utilizados. A continuación se describen los diagramas de componentes generados a partir del diagrama de paquetes definido en el diseño.

3.2.1. Diagrama de componentes del paquete *publisher_subscriber*.

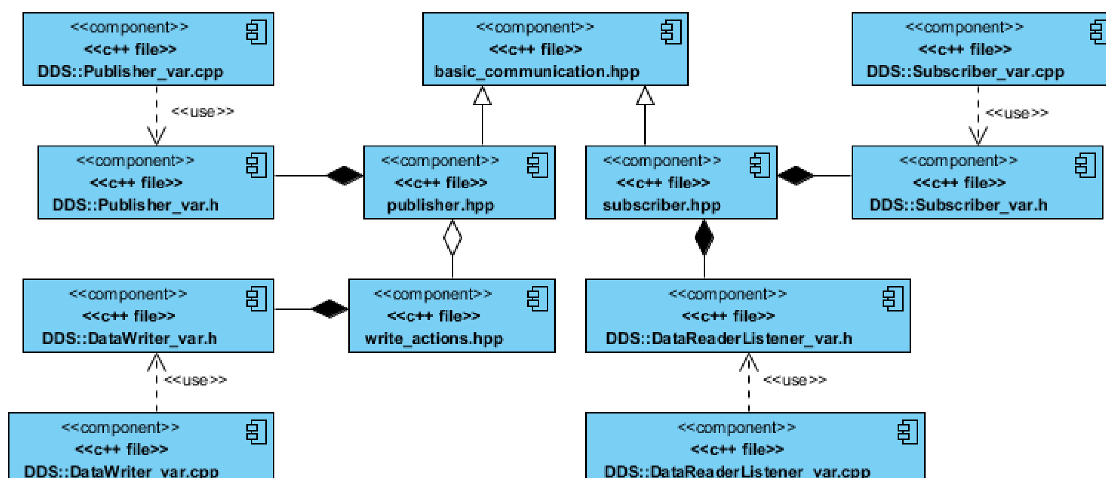

Figura 13: Diagrama de componentes del paquete publisher_subscriber

En este diagrama se muestran los distintos componentes de *OpenDDS* que conforman a *publisher.hpp*, *subscriber.hpp* y *write_actions.hpp*. Además se

puede ver la interacción entre los componentes *publisher.hpp* y *subscriber.hpp* que heredan de *basic_communication.hpp* para llevar a cabo el proceso de publicación/suscripción en el sistema. El componente b*asic_communication.hpp* contiene las características comunes de publicadores y suscriptores; y actúa como uno de ellos, en dependencia del parámetro que se le pase. El componente *write_actions.hpp* se encarga de escribir los datos que publica el componente *publisher.hpp*.

3.2.2. Diagrama de componentes del paquete *lifetime_policy*.

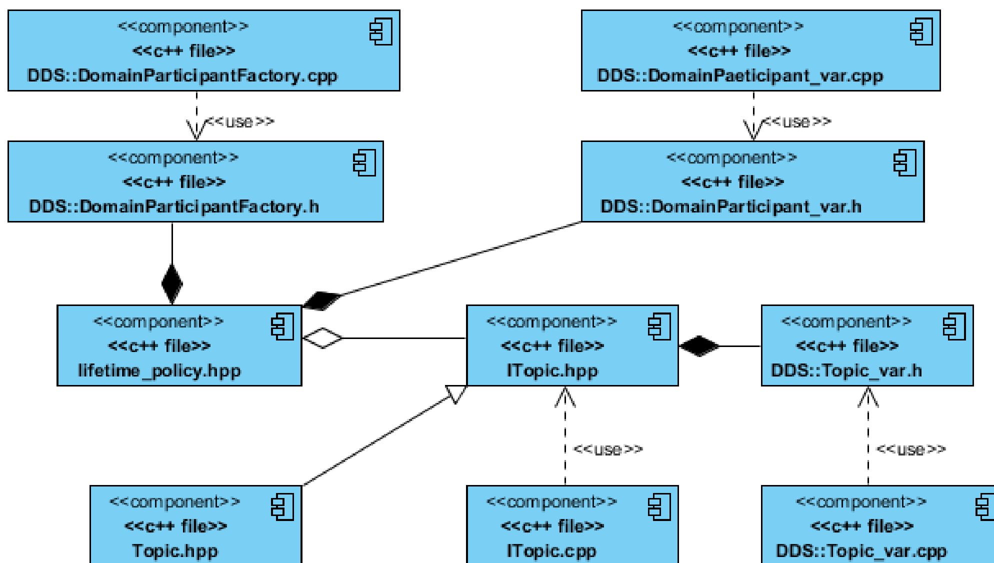

Figura 14: Diagrama de componentes del paquete lifetime_policy

En este diagrama, el componente *lifetime_policy.hpp* es el encargado de crear y destruir la fábrica de participantes, los participantes de dominio y los tópicos a través de su relación con los componentes de *OpenDDS*: *DDS::DomainParticipantFactory.h*, *DDS::DomainParticipant_var.h* y *DDS::Topic_var.h*. En el caso de los tópicos, el componente *ITopic.hpp* actúa como una interfaz para acceder a las funcionalidades de *Topic.hpp*.

3.2.3. Diagrama de componentes del paquete *comm_traits*.

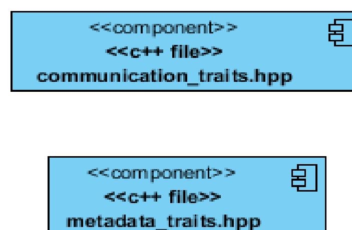

Figura 15: Diagrama de componentes del paquete comm_traits

En este diagrama se muestra el componente *communications_traits.hpp* que maneja tipos de clases distintas que contienen por separado sus características, en este caso se trabaja con las características de publicación y suscripción. En el caso de *metadata_traits.hpp* ocurre de la misma forma, lo que en este caso las características que se manejan por separado son las de puntos, alarmas, eventos, comandos, estado de la comunicación y bitácoras.

3.2.4. Diagrama de componentes del paquete *communication_data_types*.

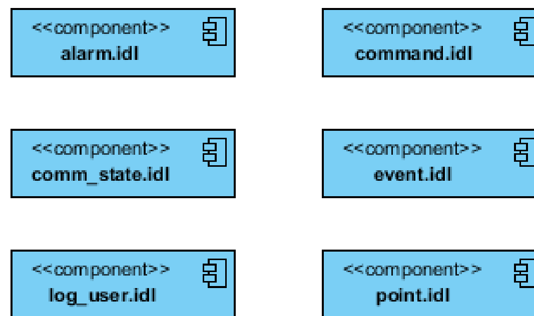

Figura 16: Diagrama de componentes del paquete communication_data_types

En este diagrama se muestran los componentes que conforman los tipos de datos con los que trabaja el subsistema de comunicación (alarmas, eventos, puntos, comandos, bitácoras y estado de la comunicación). Estos componentes son de extensión .idl debido a que utilizan el lenguaje de definición de interfaces IDL.

3.2.5. Despliegue del subsistema de comunicaciones de SAINUX.

El siguiente gráfico muestra las relaciones físicas entre los distintos nodos que componen el sistema y el reparto de los componentes sobre dichos nodos. El subsistema de comunicación de SAINUX tiene la siguiente disposición física:

- ✓ Los clientes pueden ser cualquiera de los módulos de SAINUX que pueden publicar o suscribirse a datos. En estas máquinas deben estar instaladas las bibliotecas *async_comm, communication_data_types* y *OpenDDS*.
- ✓ Hay un servidor que es donde se encuentra el *DCPSInfoRepo* que actúa como el mecanismo de descubrimiento entre los publicadores y suscriptores.

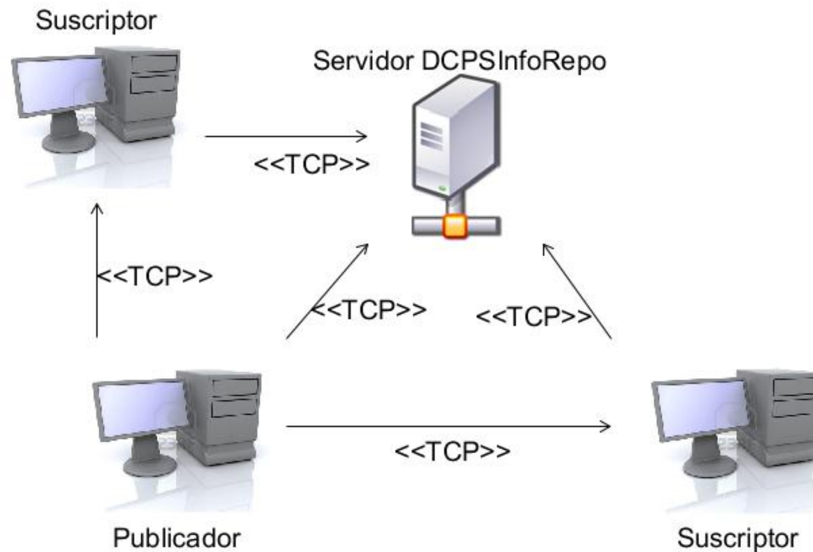

Figura 17: Despliegue del subsistema de comunicaciones

3.3. Estilo de código.

El uso de un estilo de código a la hora de programar, permite revisar y actualizar el mismo de forma fácil y ordenada. Además evita caer en errores que dificulten la comprensión del código. Seguidamente se muestra el estilo utilizado en la implementación del subsistema de comunicación, el cual se definió para ser utilizado en todos los módulos de SAINUX (Estilo de código SAINUX).

3.3.1. Nombres.

- ✓ Los nombres de las clases son sustantivos singulares.
- ✓ Los nombres deben reflejar el ¿qué? y no el ¿cómo?
- ✓ Los nombres no deben revelar detalles de implementación.
- ✓ Escoger nombres lo suficientemente largos para ser expresivos, pero evitando manejar nombres que dificulten la labor de implantación.
- ✓ Evitar nombres que permitan una interpretación subjetiva (evitar ambigüedad y asegurar abstracción).
- ✓ Evitar redundancia no repitiendo nombres de clases en sus elementos.
- ✓ Concatenar calificadores de cómputo a las variables que almacenen el producto de tal cómputo (avg, sum, min, max, index).
- ✓ Dado que los nombres generalmente son el producto de concatenar varias palabras, se emplea minúscula para el inicio de cada palabra y guión bajo para separarlas. Para el caso de los nombres de variables

debe aplicarse la misma convención y los atributos deben comenzar con guión bajo.

- ✓ Variables booleanas deben contener “is” en su nombre.
- ✓ Los nombres de constantes deben contener solo letras mayúsculas.
- ✓ Minimizar el uso de abreviaturas. En caso de ser requeridas, se debe ser consistente en su uso y cada abreviación debe significar solo una cosa. En general agregar a la documentación las abreviaturas.
- ✓ Los nombres de los métodos son frases que incluyen verbos.
- ✓ Los nombres de los atributos y parámetros son frases con sustantivos.
- ✓ Evitar el uso de nombres idénticos para distinto propósito.

3.3.2. Manejo de Errores.

- ✓ Se pueden manejar los errores mediante mecanismos de excepciones o mediante valores de retorno, aunque esto debe ser uniforme dentro de un mismo objeto.
- ✓ Es buena práctica emplear herramientas para identificar errores en la codificación en caliente.

3.3.3. Documentación y Comentarios.

- ✓ En el código debe documentarse en forma explicativa los pasos que se van ejecutando.
- ✓ Emplear oraciones completas al documentar el código.
- ✓ Documentar mientras se programa.
- ✓ Documentar cualquier cosa que no sea obvia en el código.
- ✓ Documentar eliminación de errores y cambios sobre el código.
- ✓ Al modificar el código se deben actualizar todos los comentarios y documentación asociada.
- ✓ Documentar cada rutina agregando: nombre del desarrollador, fecha, parámetros de entrada, valores de retorno, precondiciones, pos-condiciones, dependencia con otros métodos o funciones y descripción general del algoritmo. Además, de realizarse cambios al código, debe indicarse el nombre de la persona que realizó el cambio, la fecha y la descripción del cambio, comenzando desde el o los cambios más recientes.

- ✓ Evitar agregar comentarios al final de líneas de código, salvo en el caso de declaraciones. En este caso tales comentarios deben estar alineados.
- ✓ Antes de la entrega de la aplicación, eliminar todos los comentarios superfluos y/o temporales con la finalidad de evitar confusiones en su mantenimiento.

3.3.4. Codificación.

- ✓ Se establece un tamaño de indentación estándar de tres (3) espacios, sin tabulaciones. Alinear secciones del código.
- ✓ Alinear verticalmente llaves de apertura y cierre.
- ✓ Usar espacios antes y después de los operadores que el lenguaje de programación permita.
- ✓ Emplear líneas en blanco para organizar el código, permitiendo crear párrafos de código para una mejor lectura.
- ✓ Evitar colocar más de una sentencia por línea.
- ✓ Emplear constantes en sustitución de números o cadenas de caracteres literales.
- ✓ Minimizar el alcance de las variables para evitar confusión y facilitar el mantenimiento.
- ✓ Emplear cada variable y rutina solo para un propósito.
- ✓ Evitar el uso de variables públicas, sustituirlas por variables privadas y métodos que proporcionen el valor de tal variable, para mantener el encapsulamiento.
- ✓ Minimizar el uso de conversiones de tipo forzadas (castings), cuando se requiera su uso, debe ser comentada la justificación.
- ✓ Emplear select-case o switch en sustitución de if anidados sobre la misma variables. • Liberar apuntadores de manera explícita.
- ✓ Emplear i, j, k, l, p, q, r para contadores en ciclos.
- ✓ Comentar siempre las llaves que cierran.
- ✓ Emplear al máximo operadores del tipo: +=, *=, /=, -=, ++, --, entre otros.
- ✓ Mantener la modularidad del código bajo el criterio de la lógica que encierra, no exagerar la modularidad.

- ✓ Emplear correctamente los tipos de ciclos: si es al menos una vez usar do-while, si es ninguna o más veces usar while-do, y si se conoce el número exacto de ciclos usar for.
- ✓ Inicializar todas las variables.
- ✓ Emplear líneas en blanco para separar los pasos lógicos (declaraciones, lazos).
- ✓ Siempre asignar NULL a los apuntadores luego de ser destruidos (solo aplica para C).
- ✓ Evitar prácticas que incrementan explosivamente la complejidad, como lo son: objetos y variables globales y saltos tipo go-to.

3.4. Vistas más significativas del código.

En este epígrafe se muestran algunos fragmentos de código significativos para el funcionamiento del subsistema de comunicaciones de SAINUX.

3.4.1. Enviar datos según el nombre del tópico.

En este fragmento se muestra la implementación del método *send_by_topic_name()*, el cual envía datos según el nombre del *topic* que recibe por parámetros.

```
244  * Funcion auxiliar para enviar datos
245  * @param std::string topic_name
246  * @param K data
247  * @param const DDS::InstanceHandle_t instance_handle
248  * @author Yadiel Martinez Gonzalez     ymglez@estudiantes.uci.cu
249  * */
250 template<typename K>
251 void send_by_topic_name(std::string topic_name,      /**< Nombre del topico, puede ser vacio*/
252                         K data,                      /**< Dato a enviar, puede ser alarm, point, event ...*/
253                         const DDS::InstanceHandle_t instance_handle = DDS::HANDLE_NIL /**< Handle para la esc
254                         )throw (common::communication_exception)
255 {
256     typedef typename write_actions<K>::simple_data_writer_var simple_data_writer_var;
257 
258     data_writers_iterator iter = _dw_map.find( topic_name );
259     if(iter != _dw_map.end() )
260     {
261         try
262         {
263             simple_data_writer_var  dw_var;
264             dw_var = boost::any_cast< simple_data_writer_var >(iter->second._any_sdata_writer_var);
265 
266             write_actions<K>::write(dw_var, data, instance_handle);
267         }
268         catch(const boost::bad_any_cast & err)
269         {
270             throw common::communication_exception(err.what());
271         }
272     }
273     else
274     {
275         throw common::communication_exception( "Don't created topic\n" );
276     }
```

Figura 18: Fragmento de código del método send_by_topic_name()

3.4.2. Crear tópico y esperar por un publicador.

En este fragmento de código se puede observar la implementación del método *create_subscriber_topic()*, el cual se encarga de crear un *topic* para un *subscriber*; y además se muestra el método *wait_for_publisher()*, el cual es el encargado de esperar un tiempo determinado por algún *publisher* de un *topic* específico para comenzar a recibir los datos.

```
68 * Funcion para crear los topicos
69 * @param const datareader_listener_callback&
70 * @param std::string topic_name = ""
71 * */
72 template<typename K>
73 std::string create_subscriber_topic(const datareader_listener_callback& drl_callback, /**< datareader_listener
74                                 std::string topic_name = "" /**< Nombre del topico, puede ser vacio*/)
75 {
76 
77     topic_ref topic = create_topic<K>(topic_name);
78     data_operation_t data_reader =
79                         communication_traits_t::create_data_operation( topic.get_topic_var(),
80                                                                        _cm_type,
81                                                                        DDS::DataReaderListener_var(new datare
82                                                                    );
83     _dr_map.insert( std::make_pair(topic.get_topic_name(), data_reader ) );
84     return topic.get_topic_name();
85 }
86 
87 /**
88 * Espera por un publisher disponible para comenzar a recibir datos
89 *@param std::string name
90 * */
91 void wait_for_publisher(std::string name, /**< Nombre del topico*/
92                         int timeout)
93 {
94     data_readers_iterator iter = _dr_map.find( name );
95     if(iter != _dr_map.end() )
96     {
97         communication_traits_t::wait(iter->second , timeout);
98     }
99 }
```

Figura 19: Fragmento de código de los métodos create_subscriber_topic() y wait_for_publisher()

3.5. Pruebas.

Las pruebas constituyen un elemento importante en la calidad del producto final de un proceso de desarrollo de software. Estas tienen como objetivo:

- ✓ Encontrar y documentar los defectos que puedan afectar la calidad del software.
- ✓ Validar que el software trabaje como fue diseñado.
- ✓ Validar y probar los requisitos que debe cumplir el software.
- ✓ Validar que los requisitos fueron implementados correctamente.

Existen dos tipos de pruebas que son los más utilizados: la prueba de caja negra, que pretende demostrar que las entradas se aceptan de forma adecuada y que se produce un resultado correcto; y la prueba de la caja blanca, que comprueba los caminos lógicos del software.

A continuación se muestran las pruebas realizadas al subsistema de comunicación de SAINUX para verificar sus principales funcionalidades.

3.5.1. Ambiente de pruebas.

Para la correcta realización de las pruebas se utilizaron una serie de recursos que se exponen a continuación:

- ✓ **Recursos físicos:**

  Se utilizan 3 máquinas con 1 gigabyte de memoria RAM, 160 gigabyte de capacidad de disco duro, microprocesador Intel Core 2 Duo con una velocidad de 2.2Ghz, *motherboard* Intel y red alámbrica.

- ✓ **Recursos lógicos:**

  El sistema operativo sobre el cual se desarrollaron las pruebas es *Debian squeeze*.

  La velocidad de la red es de 100 megabits por segundo.

  Debe estar instalado en cada máquina el *middleware* de *OpenDDS*, así como las bibliotecas *async_comm* y *communication_data_types*.

3.5.2. Especificación de escenarios de prueba.

La realización de las pruebas tiene como objetivo fundamental determinar si el rendimiento en cuanto a la transmisión de datos es eficiente, según las aspiraciones del centro para el SCADA que se pretende desarrollar. A continuación se define el siguiente escenario de prueba.

- ✓ **Escenario de prueba:**

El presente escenario de prueba permite validar la velocidad de transferencia de datos que soporta la solución del subsistema de comunicaciones basado en el *middleware* de *OpenDDS* tanto de manera remota como local. Se utilizó un escenario lo más real posible solamente para probar el envío y recepción de datos complejos de tipo punto, teniendo en cuenta que el subsistema se comporta de la misma manera para el resto de los tipos definidos. La distribución de las máquinas y el orden de ejecución de las pruebas fue la siguiente:

PC1: Se ejecuta el servicio de *OpenDDS*: DCPSInfoRepo. Se ejecuta además un *publisher* y se pone a enviar puntos por el *topic* “punto”. Va a enviar 1000 puntos cada 1 segundo hasta llegar a los 50000 puntos. Se va a repetir esta operación hasta completar 1000 muestras.

PC2: Se ejecuta un *subscriber* escuchando puntos por el *topic* “punto”. Se mide el tiempo en que el *subscriber* recibe 1000 puntos y se promedian los tiempos de cada uno de los 50 envíos para llegar a los 50000 puntos. Se va a repetir esta operación hasta completar 1000 muestras.

PC3: Se ejecuta un *subscriber* escuchando puntos por el *topic* “punto”. Se mide el tiempo en que el *subscriber* recibe 1000 puntos y se promedian los tiempos de cada uno de los 50 envíos para llegar a los 50000 puntos. Se va a repetir esta operación hasta completar 1000 muestras.

PC4: Se ejecuta el servicio de *OpenDDS*: DCPSInfoRepo. Se ejecuta además un *publisher* y se pone a enviar puntos por el *topic* “punto”. Va a enviar 1000 puntos cada 1 segundo hasta llegar a los 50000 puntos. También se ejecuta un *subscriber* escuchando puntos por el *topic* “punto”. Se mide el tiempo en que el *subscriber* recibe 1000 puntos y se promedian los tiempos de cada uno de los 50 envíos para llegar a los 50000 puntos. Se va a repetir esta operación hasta completar 1000 muestras.

3.5.3. Casos de pruebas.

CP1: Envío y recepción de datos complejos de tipo punto de manera remota.

Descripción: El caso de prueba permite verificar el envío y recepción de datos complejos de tipo punto de manera remota (en varias PC) por un canal de comunicación, utilizando el servicio DCPSInfoRepo de *OpenDDS*. Se conectan dos suscriptores a dicho servicio escuchando por el tópico “punto”, en total recibirán 50000 puntos. El publicador se conecta al servicio para publicar los puntos por un tópico.

Flujo central

- ✓ Se obtiene un publicador al servicio DCPSInfoRepo de *OpenDDS*.
- ✓ Se le especifica el tópico por el cual va a publicar, así como la cantidad de puntos a enviar.
- ✓ Se inicia una instancia del publicador por un canal de comunicación.
- ✓ El publicador espera porque exista un suscriptor al tópico especificado.

- ✓ El publicador comienza a enviar puntos.
- ✓ Se obtiene un suscriptor al servicio DCPSInfoRepo de *OpenDDS*.
- ✓ Se le especifica el tópico por el cual va a recibir los datos.
- ✓ Se inicia una instancia del suscriptor por un canal de comunicación.
- ✓ El suscriptor espera porque exista un publicador para el tópico especificado.
- ✓ El suscriptor comienza a recibir puntos.

Condiciones de ejecución

Debe estarse ejecutando en una de las máquinas el servicio DCPSInfoRepo. Debe estar configurado el fichero /etc/hosts en cada máquina cliente de la siguiente manera:

- ✓ 127.0.0.1 localhost
- ✓ [local ip] [hostname]
- ✓ [server ip] [server name]

En el servidor la configuración debe ser la siguiente:

- ✓ 127.0.0.1 localhost
- ✓ [local ip] [hostname]
- ✓ [client 1 ip] [client 1 name]
- ✓ [client 2 ip] [client 2 name]

CP2: Envío y recepción de datos complejos de tipo punto de manera local.

Descripción: El caso de prueba permite verificar el envío y recepción de datos complejos de tipo punto de manera local (en la misma PC) por un canal de comunicación, utilizando el servicio DCPSInfoRepo de *OpenDDS*. Se conecta un suscriptor a dicho servicio escuchando por el tópico “punto”, en total recibirá 50000 puntos. El publicador se conecta al servicio para publicar los puntos por un tópico. El flujo central, así como las condiciones de ejecución para este caso de prueba se mantienen de la misma manera que para el caso anterior.

3.5.4. Análisis de los resultados de pruebas.

Caso #	Clases válidas	Clases inválidas	Resultado esperados	Resultados obtenidos	Observaciones
1	Cuando se levanta la aplicación, los suscriptores están recibiendo puntos por el tópico “punto”. Se marca el tiempo cuando se recibe la cantidad de puntos que se configuró.		Cada suscriptor debe recibir todos los puntos enviados por el publicador (50000) y el tiempo de recepción en cada muestra no debe superar el tiempo de 1 segundo.	Luego de recolectar 1000 muestras, se pudo determinar que ambos *subscriber* recibieron los 50000 puntos de cada envío con un tiempo promedio de: *Subscriber* PC2, recibió los 50000 puntos de cada envío con un tiempo promedio de: 466.00 milisegundos. *Subscriber* PC3, recibió los 50000 puntos de cada envío con un tiempo promedio de: 432. 85 milisegundos	
2	Cuando se levanta la aplicación, el suscriptor está recibiendo puntos por el tópico “punto”. Se marca el tiempo cuando se recibe la cantidad de puntos que se configuró.		El suscriptor debe recibir todos los puntos enviados por el publicador (50000) y el tiempo de recepción en cada muestra no debe superar el tiempo de 1 segundo.	Luego de recolectar 1000 muestras, se pudo determinar que el *subscriber* recibió los 50000 puntos de cada envío con un tiempo promedio de: *subscriber* PC4, recibió los 50000 puntos de cada envío con un tiempo promedio de: 318.5 milisegundos.	

Tabla 16: Resultados de pruebas para el envío y recepción de datos complejos de tipo punto.

CONCLUSIONES

Luego de finalizada la presente investigación para el desarrollo del subsistema de comunicaciones del SCADA SAINUX, se concluye que:

- ✓ Existen varios estándares e implementaciones de *middleware*, pero el más adecuado para la implementación del subsistema de comunicaciones de SAINUX resultó ser *OpenDDS*.

- ✓ Con el uso de *OpenDDS*, el subsistema de comunicaciones permite el envío asíncrono de mensajes, desacoplando en tiempo y espacio a los participantes (publicadores y suscriptores).

- ✓ El subsistema de comunicaciones desarrollado permite el envío y recepción de datos complejos, como son: puntos, alarmas, eventos, comandos, respuestas de comandos, bitácoras y estado de la comunicación.

- ✓ El desarrollo del subsistema de comunicaciones, contribuye a que SAINUX pueda utilizarse en cualquier empresa de Cuba, que requiera automatización para sus procesos, así como la comercialización del mismo en el exterior.

RECOMENDACIONES

- ✓ Implementar el comportamiento de comunicación cliente/servidor mediante la simulación del mismo a partir del modelo de publicación/suscripción que brinda *OpenDDS*, para permitir la interacción entre los módulos de seguridad y configuración con el resto de los módulos de SAINUX.

REFERENCIAS BIBLIOGRÁFICAS

1. Ger Peña, Raicel. *Sistema de Comunicación (Middleware) aplicable a diferentes entornos distribuidos. Introducción*. 2008 [citado 28/02/2012]; Disponible en: http://repositorio_institucional.uci.cu/jspui/bitstream/ident/TD_1562_08/1/TD_1562_08.pdf
2. García Giménez, Javier. *Calibración, Control y Diseño SCADA de un robot paralelo neumático con el autómata S7-300. SCADA, una primera idea.* 2008 [citado 29/02/2012]; Disponible en: http://repositorio.bib.upct.es/dspace/bitstream/10317/570/1/pfc2683.pdf.
3. Castro Lozano, Carlos y Romero Morales, Cristóbal. *Introducción a SCADA. Asignatura: Interfaz Hombre Máquina.* [citado 29/02/2012]; Disponible en: http://www.uco.es/investiga/grupos/eatco/automatica/ihm/descargar/scada.pdf
4. Portal EcuRed. Categoría: Ciencias Informáticas y Telecomunicaciones. *Sistema SCADA*. 2011 [citado 29/02/2012]; Disponible en: http://www.ecured.cu/index.php/Sistema_SCADA
5. Aragón Cáceres, José y Llanes Jiménez, A. Beatriz. *Servicio de Integración con Terceros para el acceso a variables del sistema SCADA Guardián del ALBA.* 2009 [citado 01/03/2012] Disponible en: http://bibliodoc.uci.cu/TD/TD_2554_09.pdf
6. Catalunya, U.P.d. *EPSC - DAC*. 2007 [citado 01/03/2012]; Disponible en: http://studies.ac.upc.edu/EPSC/FSD/FSD-ConceptosGenerales.pdf.
7. Schmidt, D.C. *Middleware for Distributed Systems*. 2005 [citado 2/03/2012]; Disponible en: http://www.cs.wustl.edu/~schmidt/PDF/middleware-encyclopedia.pdf.
8. Fernández, D.V., *Documentación de ZeroC ICE*. 2006; Disponible en: http://www.esi.uclm.es/www/dvallejo/docs/docICE.pdf
9. Guibert Nápoles, Rosalbis y Cárdenas del Valle, Yusniel. *Mecanismos de seguridad para el middleware del SCADA "Guardián del ALBA".* 2008 [citado 02/03/2012] Disponible en: http://catalogoenlinea.uci.cu/cgi-bin/koha/opac-detail.pl?biblionumber=6612
10. *RMI mano a mano con SSL: construyendo aplicaciones distribuidas seguras.*2011 [citado 2/03/2012]; Disponible en: http://www.programacion.com/articulo/rmi_mano_a_mano_con_ssl:_construyendo_aplicaciones_distribuidas_seguras_79#seccion-UTILIZANDO-SSL.
11. Javier, I.M.P., *Selección de la tecnología adecuada para implementar el Middleware V 2.0*. 2009.

12. *¿Qué es CORBA?* 2011 [citado 2/03/2012]; Disponible en: http://es.scribd.com/doc/31361481/CORBA.
13. GNOME, F. *Programación en el Entorno GNOME.*2002 [citado 2/03/2012]; Disponible en: http://www.calcifer.org/documentos/librognome/index.html.
14. Soto, R.G. *Extensión de xawtv para la transmisión de flujos de video mediante ACE-TAO*. 2003 [citado 2/03/2012]; Disponible en: http://repositorio.bib.upct.es/dspace/bitstream/10317/192/1/pfc1132.pdf.
15. Vega, J.M.L. *Plataforma de Trabajo Colaborativo sobre Middleware DDS*. 2008 [citado; Disponible en: http://dtstc.ugr.es/tl/pdf/pf/PFC_jmlvega_Final_rev04.pdf.
16. Luján, J.L.P., *Revisión de los Sistemas de Comunicaciones más empleados en Control Distribuido*. 2009, Instituto de Automática e Informática Industrial, Universidad Politécnica de Valencia.
17. Inc, O.C. *OpenDDS*. 2012 [citado 3/03/2012]; Disponible en: http://www.OpenDDS.org/.
18. Grady, J.B., *El Lenguaje Unificado de Modelado*. 2007.
19. Portal EcuRed. *Visual Paradigm*. 2012 [citado 6/03/2012]; Disponible en: http://www.ecured.cu/index.php/Visual_Paradigm.
20. *OpenDDS. Articles. Introduction to OpenDDS. DDS Architecture.* 2012 [citado 10/04/2012]; Disponible en: [http://www.OpenDDS.org/Article-Intro.html]
21. García Aranda, Fernando. *Plataforma extensible para la monitorización de sistemas y la detección de intrusiones con DDS. Antecedentes y estado del arte. Paradigma Publicación/Suscripción.* 2012 [citado 12/04/2012]; Disponible en: https://forja.rediris.es/docman/view.php/679/1326/cavecanem-manual_tecnico.pdf
22. Wesley, Addison. *Modern C++ Design: Generic Programming and Design Patterns Applied. Type Traits.* 2001 [citado 14/05/2012]
23. Wesley, Addison. *Modern C++ Design: Generic Programming and Design Patterns Applied. Policy-Based Class Design.* 2001 [citado 14/05/2012]
24. Patrones del "Gang of Four". Unidad docente de Ingeniería del Software. Facultad de Informática-Universidad Politécnica de Madrid. *Facade/Fachada.* [citado 14/05/2012].
25. Tellería Martínez, Ana Silvia. Diseño de un middleware básico para el intercambio de información de un sistema supervisor de procesos automatizados. *Glosario*. 2007 [citado 18/05/2012]; Disponible en: http://catalogoenlinea.uci.cu/cgi-bin/koha/opac-detail.pl?biblionumber=4792
26. Ravelo Hernández, Luis Angel. Propuesta de tecnología adecuada para mejorar la versión 1.0 del middleware del Guardián del Alba. *Glosario de términos*. 2009 [citado

18/05/2012]; Disponible en: http://catalogoenlinea.uci.cu/cgi-bin/koha/opac-detail.pl?biblionumber=8089

ANEXOS

Anexo 1: Elementos a tener en cuenta en el desarrollo del modelo de decisión elaborado para la selección de tecnologías a utilizar en la implementación del subsistema de comunicaciones del SCADA SAINUX.

Una matriz de decisión es una herramienta de decisiones utilizada por los arquitectos de software como parte de sus herramientas a la hora de seleccionar una tecnología o aplicación a utilizar. En la siguiente tabla se detalla la información que posee la matriz de decisión.

Criterios:	Opciones:	Importancia:	Puntaje:
Jerarquía de criterios de decisión, también conocido como modelo de decisión.	Opciones a seleccionar, también llamado soluciones o alternativas	Valor de cada criterio sobre la base de su importancia en la decisión final, es un valor comprendido entre 0 y 100.	Tasa de cada opción en una relación de escala mediante la calificación a cada criterio.

Tabla 17: Elementos de la matriz de decisión.

Como calificar una opción.

Calificación	Descripción.
0	No Aceptable
1	Poco Aceptable
2	Aceptable.
3	Sobresaliente.
4	Excelente.

Tabla 18: Valores cuantitativos de las calificaciones a las tecnologías.

Modelo de Decisión		ALTERNATIVAS					
		ORBit		ACE+TAO		OpenDDS	
Criterios	Importancia	Calificación	Puntaje	Calificación	Puntaje	Calificación	Puntaje
Desarrollo en C++.	15	2	30	4	60	4	60
Implementación con herramientas libres y código abierto.	15	4	60	4	60	4	60
Comunicación distribuida.	15	4	60	4	60	4	60
Capacidad de tiempo real.	10	4	40	3	30	4	40
Manejo de un elevado número de variables.	10	2	20	3	30	4	40
Persistencia a las conexiones.	10	2	20	3	30	4	40
Tolerancia a fallas.	5	2	10	3	15	4	20
Redundancia.	10	2	20	3	30	3	30
Seguridad.	10	2	20	2	20	2	20
Total.	100	24	280	29	335	33	370

Tabla 19: Modelo de decisión de la tecnología más apropiada.

Puntaje = calificación * Importancia.

Importancia = rango entre 1 y 100.

GLOSARIO

API: (*Application Programming Interface*) en español, Interfaz de Programación de Aplicaciones, es un conjunto de definiciones de funciones que abstraen los detalles de la implementación facilitando el desarrollo de aplicaciones.

Cliente/Servidor: Es una arquitectura de red que separa al cliente del servidor. Se le llama cliente al proceso que inicializa la comunicación haciendo una petición; y servidor al proceso que responde a las solicitudes del cliente.

Framework: Es un conjunto de clases que encapsulan diseños abstractos de soluciones a un determinado número de problemas en relación. Los objetivos principales que persigue un *framework* son: acelerar el proceso de desarrollo, reutilizar código ya existente y promover buenas prácticas de desarrollo como el uso de patrones. **[26]**

GPL: (*General Public License)*, es una licencia que regula los derechos de autor de los programas de software libre (*free software*) promovido por la Fundación de Software Libre en el marco de la iniciativa GNU. **[5]**

Hilos: Los hilos en las aplicaciones, son tareas que pueden ser ejecutadas concurrentemente con otras, es decir, que se pueden ejecutar varias tareas a la vez.

IIOP: (*Internet Inter-Orb Protocol*) Protocolo que posibilita la interoperabilidad de las aplicaciones de ambientes heterogéneos en Internet. Adoptada como parte de CORBA por la OMG a partir de 1994. Es un mecanismo subyacente a CORBA y es administrado de forma transparente a los usuarios finales por el ORB para la comunicación con sus homólogos implementados con otras tecnologías. Constituye la individualización para TCP/IP de la especificación más general de **GIOP** (*General Inter-ORB Protocol*). **[25]**

OMG: (*Object Management Group)* o Grupo de Gestión de Objetos, es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos. Es una organización sin ánimo de lucro que

promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas. **[5]**

ORB: (*Object Request Broker)*, constituye el núcleo de la tecnología CORBA. **[5]**

Software Libre: Las cuatro reglas esenciales que deben cumplir las aplicaciones para ser consideradas como software libre son: **[25]**

- Libertad 0: Libertad de ejecutar el programa como quieras.

- Libertad 1: Libertad de estudiar el código fuente y cambiarlo para realizar lo que desees.

- Libertad 2: Libertad de realizar copias y distribuirlas cuando quieras.

- Libertad 3: Libertad de distribuir o publicar versiones modificadas cuando desees.

TCP/IP: Es un conjunto de protocolos de red en la que se basa Internet y que permiten la transmisión de datos entre redes de ordenadores. En ocasiones se le denomina conjunto de protocolos TCP/IP, en referencia a los dos protocolos que la componen: Protocolo de Control de Transmisión (TCP) y Protocolo de Internet (IP), que fueron los dos primeros en definirse, y que son los más utilizados de la familia. **[5]**

UDP: (*User Datagram Protocol)*, es un protocolo del nivel de transporte basado en el intercambio de datagramas, permitiendo el envío de estos a través de la red sin que se haya establecido previamente una conexión. **[5]**

Software privativo: Es todo software que no cumple con alguna de las cuatro reglas del software libre. **[25]**